Università degli Studi di Napoli
"Federico II"

FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

Integrazione e gestione di servizi batch nell'ambiente pervasivo UbiSystem

RELATORE
Ch.mo Prof. Antonio d'Acierno

CANDIDATO
Rita Fusco
Matr.: 041/1829

CORRELATORE
Ing. Esposito Massimo

ANNO ACCADEMICO 2004 / 2005

RINGRAZIAMENTI

Quando dopo il liceo ho dovuto fare la difficile scelta di quale facoltà frequentare all'università, sapevo poco o niente del mondo in cui mi apprestavo ad entrare,né conoscevo bene tutti gli sviluppi del campo nel quale mi accingevo a compiere i miei studi. Tuttavia quello che sapevo era stato sufficiente a suscitare in me una crescente curiosità e voglia di imparare. Era un mondo nuovo, a cui mi avvicinavo consapevole che sarebbe stato impegnativo. Sentivo però in cuor mio che ero in grado di farcela,pur sapendo che il cammino non sarebbe stato certo sempre facile.

Dei miei anni di università conserverò un buon ricordo,portando con me tutte le esperienze fatte ,quelle positive e quelle negative,perché tutte hanno contribuito a formare il mio carattere di adesso. Già da parecchio non sono ovviamente più la ragazzina che si iscrisse al primo anno. Mi ritrovo ora più consapevole,certamente più matura,più disillusa e spero almeno un po' più sicura di me. Li trovo cambiamenti tutto sommato positivi.

Ho vissuto,come tutti,momenti di soddisfazione e altri di delusione o avvilimento:anche in questi momenti,non ho mai,neppure per un secondo messo in dubbio la decisione che avevo preso.

Queste circostanze belle e brutte certamente non sono relative al solo ambito prettamente accademico:sono legati ad avvenimenti e a persone che, ciascuna a suo modo, sono state determinanti nel mio percorso universitario.

Ho conosciuto tantissime persone;con diversi ho stretto amicizie più profonde con altri ho condiviso magari soltanto un corso o un esame;tutte,in ogni caso mi hanno dato qualcosa.

Inizierò allora a ringraziare il mio relatore ,il prof. Antonio d'Acierno. Oltre all'indubbio

interesse per l'oggetto di studio del mio lavoro di tesi,alla scelta del relatore ha contribuito anche l'impressione che del professore avevo avuto seguendo con lui un corso :una persona seria,intelligente,pratica. Questi ultimi mesi non hanno fatto altro che confermare la mia prima impressione. Inoltre ha contribuito a darmi fiducia,e ad ogni incontro con lui mi sono sempre sentita spronata nel lavoro. Lavoro che non sarebbe stato possibile portare a termine senza la pazienza dell'ing.Massimo Esposito,che ha resistito alle mie numerose domande "fiume",fatte ora di persona ,ora via e-mail,ora per telefono: per ogni mio problema mi ha sempre aiutato a trovare la soluzione,dedicandomi il tempo necessario.

Non sembrerà retorico se le persone a cui più sento di essere grata sono i miei genitori,Antonio e Anna che anche di fronte alle difficoltà incontrate,mi hanno sempre sostenuto, hanno creduto in me e mi hanno dato tutto il supporto necessario:spero di poterli ripagare con altre sempre maggiori soddisfazioni. Un discorso a parte merita il "piccolo " di casa,mio fratello Carlo, che oggi credo sia più emozionato di me,per quanto ha atteso il giorno della mia laurea. Non gli dirò mai troppo spesso quanto lui sia importante e determinante per me ;non è un mistero per nessuno che non saprei immaginare la mia vita senza di lui.

Un pensiero particolare va a zia "Bobba",che ha seguito con partecipazione affettuosa tutte le fasi di questo percorso universitario.

Un grazie sentito va ad un ragazzo conosciuto relativamente da poco, ma di cui ho già una altissima opinione e verso il quale nutro una profonda stima, Carlo Mele:a lui va tutta la mia riconoscenza e spero di aver posto le basi,anche se in maniera un po' insolita ,per una duratura amicizia.

L'ultimo periodo ho frequentato poco rispetto agli anni precedenti la facoltà,ma questo non mi ha fatto certo dimenticare i miei insostituibili "compagni di viaggio",Marco ,Peppe e Giovanna: col tempo si sono dimostrati tra i migliori amici che ho:hanno condiviso le mie gioie, hanno sopportato le mie lagnanze, sono stati i primi a starmi vicino nei momenti più bui.

Rispetto ai primi anni di università non sono soltanto cambiata io,sono cambiate anche le persone intorno a me. Penso allora sia giusto esprimere un pensiero anche per quanti hanno condiviso con me questi ultimi anni,diventando pian piano dei punti di riferimento importanti sui quali spero di poter contare ancora in futuro. Sicuramente non ringrazierò mai

abbastanza Luca, che mi ha aiutato con discrezione e pazienza a superare le mie delusioni più grandi, e che ancora oggi mi incoraggia e mostra di credere in me e in quello che faccio. Un affettuoso pensiero va anche alle mie amiche di sempre,Imma e Graziella,con le quali sono praticamente cresciuta,e a coloro, tra quelli che ho conosciuto in questi ultimi anni, che da semplici conoscenti sono diventati degli amici insostituibili per me e dai quali spero di essere ricambiata:Giovanna, che con la sua esuberanza riempie di colore la vita di chi le sta attorno;Giusi, con cui condivido spesso il modo di pensare e di vedere le cose e della quale apprezzo sempre i consigli sinceri; e Salvatore, che è più riservato e che quindi faccio più fatica a decifrare e a conoscere a fondo, ma che stimo tanto, e so di non sbagliarmi in questo giudizio: senza di loro la mia vita sarebbe senz'altro molto più monotona.

Ai miei genitori e a mio fratello

Carlo

INDICE

INTRODUZIONE

Lo sviluppo sempre più rapido di nuove tecnologie nel campo dell'hardware e delle comunicazioni, ha fatto assistere negli ultimi anni al proliferare di computer sempre più piccoli e poco costosi e parallelamente alla diffusione di nuove tecnologie di comunicazione. Questi eventi stanno lentamente cambiando il modo di concepire l'elaborazione informatica e le modalità d'interazione fra l'uomo ed i computer. In breve, ci troveremo a vivere in ambienti popolati da una serie di dispositivi "intelligenti" che comunicano e cooperano tra di loro per assisterci nello svolgimento delle nostre comuni attività. A questo tipo di scenari ci si riferisce quando si parla di *Ubiquitous* (o *Pervasive*) *Computing*: scenari in cui capacità di calcolo e di elaborazione assumono caratteristiche di ubiquità, si diffondono nella nostra realtà e penetrano all'interno degli oggetti che ci circondano. Il *Pervasive Computing* si basa su ambienti, dotati di proprietà computazionali e comunicative, che contengono un gran numero di componenti hardware e software, autonomi ed eterogenei, che necessitano di cooperare e che tendono ad essere altamente dinamici.

All'interno dei laboratori dell'ICAR-CNR,è stato definito un modello architetturale di sistema pervasivo, ed è stato implementato un prototipo,di nome *UbiSystem*, con il fine di realizzare un ambiente dinamico e scalabile per l'erogazione, la ricerca e la fruizione di servizi.

L'obiettivo di questa tesi è l'espansione del suddetto prototipo, affinché soddisfi nuovi requisiti oltre a quelli che sono stati considerati prima del presente lavoro. In particolare è stata rivista la gestione degli utenti del sistema alla luce di una nuova possibilità, che è quella di introdurre nell'ambiente servizi di tipo computazionale. Precedentemente sono stati

considerati soltanto servizi di tipo interattivo che venivano chiusi all'atto dell'uscita dell'utente da *UbiSystem*:da questa nuova integrazione sono sorte nuove problematiche relative alla gestione delle sessioni di utente. E' stata infatti contemplata la nuova possibilità di non chiudere completamente una sessione nel caso ci siano dei servizi batch ancora in attività e di conseguenza di riconoscere, al rientro nell'ambiente ,un utente che aveva precedentemente richiesto un servizio computazionale,al fine di fornirgli i risultati della elaborazione richiesta. A tal fine è stato anche sviluppato un nuovo componente al fine di fornire dei messaggi. Al momento questo servizio serve per poter comunicare all'utente l'evento di terminazione delle computazioni richieste,ma più in generale potrà in futuro gestire anche altri tipi di messaggio che si riterranno opportuni. E' stato infine proposto un modello a cui attenersi per introdurre in *UbiSystem* un qualsiasi servizio batch, affinchè che esso possa correttamente interagire con gli altri componenti già esistenti.

I componenti del sistema sono realizzati come servizi CORBA e solo se necessario, ovvero quando espongono funzionalità che devono essere rese accessibili dall'esterno, sono esposti come Web Services.

L'architettura si articola su due livelli:al primo si trovano i componenti che attuano le politiche di gestione per l'intero sistema, al secondo si trovano sia i servizi destinati alla fruizione da parte dell'utente, sia i servizi di supporto alle attività svolte all'interno dell'ambiente.

Il lavoro di tesi è così articolato: nel primo capitolo vengono introdotti i concetti di Pervasive ed Ubiquitous Computing, discutendo problematiche, aspetti innovativi e stato dell'arte; nel secondo capitolo vengono descritte le principali tecnologie che sono alla base del Pervasive Computing, in particolar modo, quelle impiegate per lo sviluppo della tesi; nel terzo capitolo viene descritto il modello architetturale di sistema pervasivo proposto;nel quarto capitolo viene illustrato il progetto dei moduli implementati e i dettagli implementativi; nel quinto capitolo viene descritto il modello da seguire per integrare servizi computazionali in *UbiSystem*; nel sesto capitolo viene illustrato il funzionamento del sistema relativamente ai nuovi componenti introdotti ; nelle conclusioni e sviluppi futuri,infine, vengono discussi i risultati ottenuti e prospettate le possibili evoluzioni del sistema.

Il presente lavoro è stato sviluppato presso l'ICAR-CNR nei laboratori di Napoli siti in Via Pietro Castellino.

I. IL PERVASIVE COMPUTING

I.1 INTRODUZIONE

Era il 1991 quando Mark Weiser, direttore scientifico delle ricerche tecnologiche allo Xerox PARK, in un suo articolo d'avanguardia, utilizzò per la prima volta il termine *Ubiquitous Computing* [36]. Nel suo articolo, Weiser annunciava un cambiamento nel modo di concepire l'elaborazione automatica e descriveva scenari in cui i computer, onnipresenti, entravano sempre più a far parte della vita di tutti i giorni.

I *computer* ed il *computing* in generale, infatti, stanno lentamente ed inesorabilmente navigando verso nuovi paradigmi. Negli anni sessanta, alla parola *"computer"* venivano associati grandi e costosi *mainframe*, caratterizzati da un grosso numero di utenti che ne condividevano le risorse. Si parlava di paradigma *"many people per computer"*: molti utenti per una sola macchina. Il progresso tecnologico ha poi consentito la realizzazione dei *personal computer* che hanno significativamente modificato il tipo di utilizzo dei sistemi di calcolo, trasformando il paradigma in *"one person per computer"*: ogni persona poteva disporre di un proprio calcolatore. Nell'ultimo decennio, la diffusione di *laptop*, *Personal Digital Assistant* (PDA), telefoni cellulari multifunzione, dispositivi portatili dotati di microprocessori e di capacità di immagazzinare dati, ha mutato ulteriormente il rapporto uomo-computer, aprendo le porte all'era dei *"many computers per person"*: tanti elaboratori per una singola persona [31].

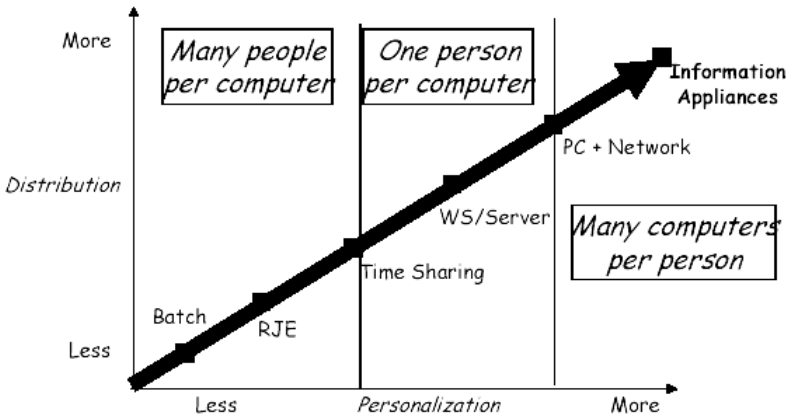


Fig. I.1 L'evoluzione del computing dai mainframes ad oggi.

Il proliferare di questi dispositivi intelligenti, sempre più piccoli e meno costosi, insieme con i progressi delle tecnologie di comunicazione, ha indotto Mark Weiser ad immaginare un futuro, non troppo lontano, nel quale i computer entrano a far parte integrante di ogni oggetto della realtà quotidiana, favorendo la comunicazione e l'elaborazione di informazioni in maniera naturale: *everywhere, all the time*.

L'essenza della sua idea era *"the creation of environments saturated with computing and computational capability, yet gracefully integrated with human users"* [32].

Un sistema di Ubiquitous Computing è caratterizzato da due attributi fondamentali [22]:

- *Ubiquità*: l'interazione con il sistema è disponibile dovunque l'utente ne abbia bisogno;
- *Trasparenza*: il sistema non è intrusivo ed è integrato negli ambienti della vita quotidiana.

In accordo con questa visione è possibile identificare due dimensioni che forniscono una più chiara definizione degli *ubiquitous system* ed esprimono le relazioni esistenti con le altre aree di ricerca emergenti:

- *Mobilità dell'utente*: esprime la libertà che l'utente ha di muoversi quando interagisce con il sistema;
- *Trasparenza di interfaccia*: riflette lo sforzo consapevole e l'attenzione che il sistema richiede all'utente, sia per operare su di esso che per percepirne i suoi output.

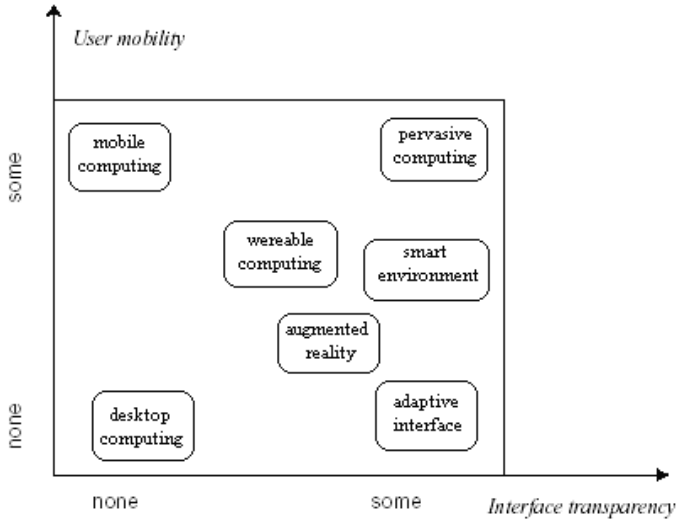


Fig. I.2 Attributi di un sistema pervasivo.

Quindi, nell'ottica di Weiser, l'*Ubiquitous Computing* mira alla realizzazione di un mondo non più vincolato alle scrivanie, ma composto da ambienti dotati di capacità computazionali e comunicative, talmente integrati con l'utente da diventare una "tecnologia che svanisce" [32], utilizzabile in maniera trasparente ed inconscia. Non una realtà virtuale, in cui le persone sono inserite in un mondo generato dai computer, ma piuttosto una virtualità reale che porta i computer a vivere nel mondo reale, insieme con le persone [36].

Tuttavia, gli scenari dipinti da Weiser 13 anni fa erano anacronistici; la tecnologia hardware necessaria per la loro realizzazione semplicemente non esisteva. E così, i tentativi compiuti allo Xerox PARC fallirono.

Dopo diversi anni, i recenti sviluppi tecnologici hanno dato nuovo impulso alle ricerche sull'*Ubiquitous Computing*, di recente ribattezzato anche col nome di *Pervasive Computing*, per suggerire il carattere pervasivo con cui l'"intelligenza elaborativa" si diffonde e si manifesta negli oggetti che ci circondano.

Probabilmente, i tempi non sono ancora maturi e la visione di Weiser resta ancora futuristica: "*The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it*". Di sicuro, però, oggi ci troviamo in una posizione migliore, rispetto a quella dei ricercatori della Xerox del 1991, per affrontare le problematiche relative all'*Ubiquitous*, o *Pervasive, Computing*.

Da qualche anno, di fatto, in diverse Università e Centri di Ricerca del mondo sono sorte numerose iniziative che seguono la direzione tracciata da Weiser; ciascun progetto affronta i vari problemi da differenti punti di vista, talvolta contrastanti, ma tutti sono impegnati nello sforzo comune di rendere il *Pervasive Computing* una realtà.

I.2 L'EVOLUZIONE VERSO IL PERVASIVE COMPUTING

Il Pervasive Computing può essere visto come una nuova forma di computazione altamente dinamica e disaggregata: gli utenti sono mobili e i servizi sono forniti da una serie di componenti distribuiti che collaborano tra loro [5]. Le applicazioni necessarie per supportare queste nuove esigenze sono, da un punto di vista architetturale, non monolitiche bensì costituite da moduli allocati in numerosi nodi della rete.

In tal senso, il *Pervasive Computing* costituisce una nuova tappa nel percorso evolutivo dell'elaborazione e del calcolo distribuito [32].

I.2.1 SISTEMI DISTRIBUITI

Il *Distributed Computing* mira a ripartire dati e capacità computazionali su componenti indipendenti, residenti su macchine diverse, che comunicano tra loro attraverso reti di interconnessione.

Le metodologie attraverso le quali comunicare ed accedere ai dati e agli strumenti di calcolo da postazioni remote, le tecniche sulla replicazione e ridondanza dei dati, che favoriscono la disponibilità e la reperibilità delle informazioni e aumentano l'affidabilità del sistema nel complesso, rappresentano l'oggetto di studio per questa forma di computing.

I.2.2 SISTEMI MOBILI

Con l'introduzione di vincoli e problematiche legate al concetto di mobilità, è stato necessario pensare a nuove soluzioni tecnologiche che hanno portato alla creazione di una nuova forma di *computing*, ossia il *Mobile Computing* [24].

In questi nuovi scenari di calcolo distribuito, non si hanno più nodi di rete fissi, con connessioni stabili e veloci, ma nodi costituiti da dispositivi mobili che accedono alla rete e

la abbandonano continuamente ed in maniera del tutto imprevedibile, dotati di connessioni precarie e contraddistinte da forti cambiamenti sulle caratteristiche di banda.

Le limitate capacità di calcolo e di memoria dei dispositivi mobili, le esigenze di risparmio energetico, rappresentano ulteriori aspetti di cui il *Mobile Computing* si sta occupando.

I.2.3 SISTEMI PERVASIVI

I sistemi di *Pervasive Computing* sono a loro volta anche sistemi distribuiti e mobili. Pertanto, le problematiche inerenti il *mobile* ed il *distributed computing* vengono riprese in questo nuovo paradigma ma, in certo senso, amplificate oltremodo a causa dei requisiti stringenti e dei particolari contesti definiti in questi nuovi ambienti.

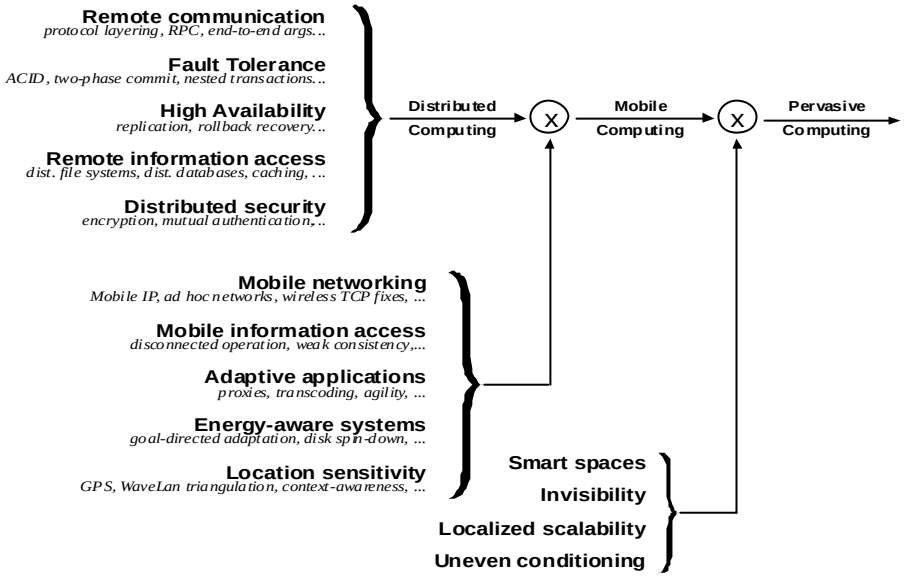


Fig. I.3 Dal Distributed Computing al Pervasive Computing.

La figura I.3 mostra come i problemi di ricerca nel *Pervasive Computing* sono relazionati a quelli del *Distributed* e del *Mobile Computing*. Muovendoci da sinistra verso destra, incontriamo nuove incognite. Inoltre, le soluzioni di problematiche precedentemente affrontate diventano ancora più complesse: come suggeriscono i simboli in figura, l'aumento di complessità è moltiplicativo piuttosto che additivo. E' molto più difficile progettare e realizzare un sistema pervasivo che un sistema distribuito di comparabile robustezza e maturità [32].

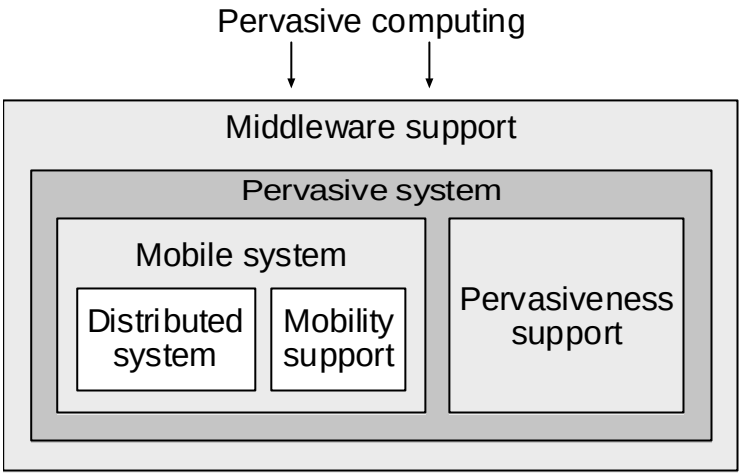


Fig. I.4 Un modello architetturale per sistemi pervasivi.

Concludiamo questo paragrafo con la figura I.4 che riassume quanto finora detto, illustrando come si compone l'architettura di un sistema pervasivo a partire da sistemi mobili e distribuiti [30].

I.3 BACKGROUND E STATO DELL'ARTE

I numerosi progetti realizzati in tutto il mondo per implementare un'infrastruttura in grado di trasformare un semplice ambiente in un ambiente di *ubiquitous computing* evidenziano chiaramente l'ingente impiego di tempo e di risorse che è stato profuso negli ultimi anni.

Ognuna di queste iniziative ha focalizzato la propria attenzione su uno specifico contesto con determinati applicazioni e servizi (ad esempio abitazioni, uffici, *meeting-room*), essendo ancora irrealizzabile l'idea di sviluppare un'unica infrastruttura di tipo generale capace di essere usata in tutti gli ambienti esistenti.

Di seguito sono analizzati, singolarmente, gli esempi più significativi di sistemi pervasivi implementati, evidenziando, per ciascuno di essi, le caratteristiche principali e i modelli architetturali.

I.3.1 AURA

Aura [27], il progetto seguito alla Carnieg Mellon University in Pittsburgh, mira a fornire all'utente un ambiente di *computing* "libero da distrazioni" (*distraction free ubiquitous computing*), dove le persone possono accedere ai servizi o svolgere le proprie

attività senza interventi sul sistema o sull'ambiente. Aura assume un ruolo proattivo anticipando i bisogni degli utenti.

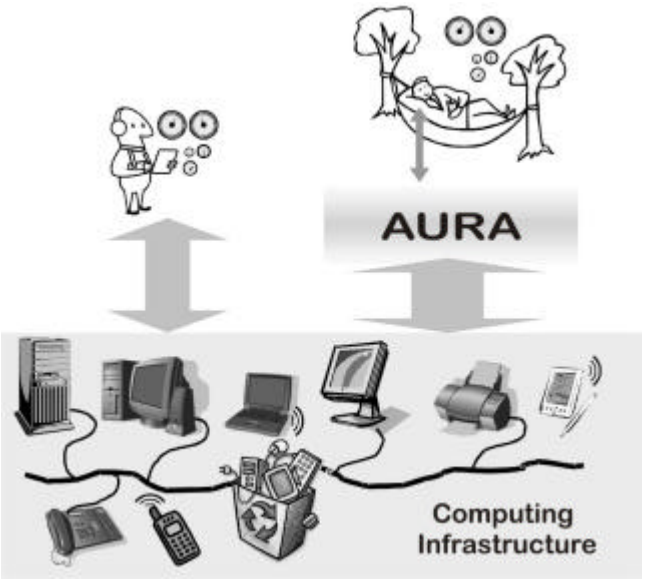


Fig. I.5 AURA: "proxy for peaple"

Aura utilizza essenzialmente due concetti per perseguire i suoi obiettivi:

Pro-activity: rappresenta la capacità, a livello di sistema, di anticipare richieste provenienti da un livello più alto;

Self-tuning: rappresenta la capacità di adattarsi osservando le richieste fatte e di modificare le prestazioni e l'utilizzo delle risorse in base ad esse.

In questa architettura è di fondamentale importanza il concetto di " *user's task*", ovvero l'attività che l'utente vuole svolgere nell'ambiente in cui si trova; ad esempio proiettare una presentazione, stampare, oppure operazioni più complesse [47].

Lo scopo che AURA si prepone è quello di favorire l'utente nell'ottenimento dei suoi obiettivi utilizzando al meglio le risorse presenti nell'ambiente.

Per favorire un'azione del genere c'è bisogno di una struttura architetturale che sia in grado di individuare la natura del task dell'utente, le preferenze personali e le intenzioni.

Tali conoscenze sono la chiave per configurare e monitorare l'ambiente in modo da rendere trasparente all'utente l'eterogeneità degli ambienti informatici e la mutevolezza delle risorse [49].

I.3.1.1 ARCHITETTURA DI AURA

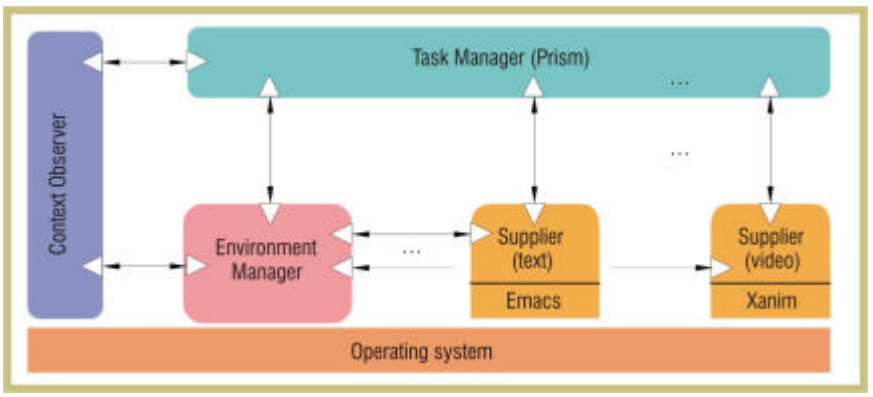


Fig. I.6 L'architettura di AURA.

I componenti principali di AURA sono:

- *Task Manager (Prism)* – Esso ha il compito di rendere l'ambiente libero da distrazioni per l'utente, inoltre si occupa della gestione dei cambiamenti relativi: alla locazione dell'utente; alla conformazione dell'ambiente; alla composizione del task. Quindi Prism fornisce un supporto di alto livello per affrontare gli aspetti di *proactivity* e di *self-tuning*.
- *Environment Manager* – Esso rappresenta la via di accesso all'ambiente. Incapsula i meccanismi per l'accesso a file distribuiti. E' a conoscenza delle risorse disponibili in ogni momento nell'ambiente in quanto, quando un servizio si rende disponibile deve darne comunicazione all' *Environment Manager*. Inoltre, per servizi dotati di risorse condivise limitate, come ad esempio dispositivi input/output, esso tiene traccia anche della capacità disponibile. In aggiunta a meccanismi di discovery di un singolo servizio, un *Environment Manager* più sofisticato è in grado di valutare le possibili configurazioni di un servizio per selezionare quella che combacia meglio con l'esigenza dell'utente.
- *Context Observer* – Offre informazioni riguardo al contesto fisico e notifica gli eventi relativi ad esso a *Environment Manager* e a *Prism*. Esempi di tali informazioni sono: locazione dell'utente, autenticazione, attività, etc. In ogni ambiente tale componente può avere diversi gradi di complessità, a seconda dei mezzi a disposizione (sensori).

- *Supplier* – Ogni *Supplier* fornisce un servizio astratto. E' realizzato incapsulando le applicazioni esistenti. Ad esempio: Emacs, Microsoft Word e Notepad possono essere incapsulati per diventare il servizio: "*text editing*".

I.3.1.2 ARCHITETTURA DI UN AURA CLIENT

Affinché AURA possa offrire i servizi per cui è stato progettato è necessario che anche il client sia dotato di una struttura sofisticata.

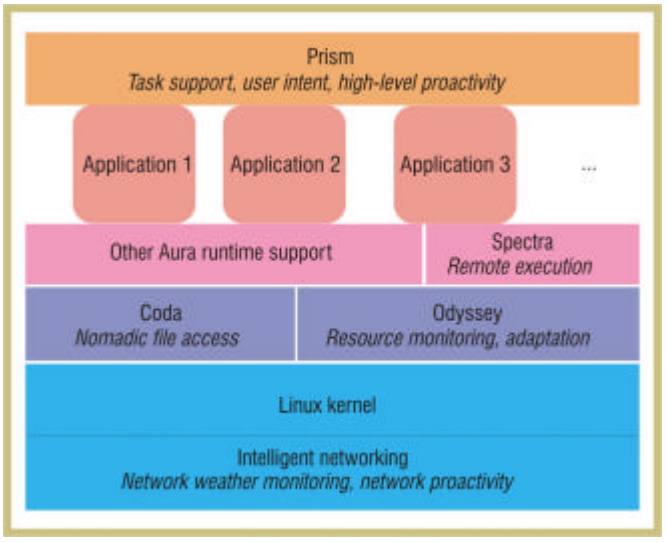


Fig. I.7 Struttura di un AURA client.

I componenti di un AURA client risultano:

- *Odyssey* – è un'estensione del sistema operativo, che consente al sistema di monitorare le risorse ed alle applicazioni di adattarsi in base alla loro disponibilità;
- *Coda* – è un sistema di gestione di file distribuiti che consente l'accesso continuato ai dati anche in presenza di malfunzionamenti della rete o dei server;
- *Spectra* – è un meccanismo di esecuzione remota ed adattativa che utilizza il contesto per decidere come meglio eseguire le invocazioni remote.

Nello scenario descritto è possibile notare che sia i componenti hardware sia le tecnologie software non sono una novità, infatti Coda e Odyssey sono stati creati prima di Aura, ma sono stati modificati per andare incontro alle esigenze del pervasive computing.

Aura ricicla, in pratica, tecnologie esistenti per creare nuovi sistemi adatti ad essere inseriti in ambienti pervasivi.

I.3.1.3 AURA A LAVORO

Allo scopo di illustrare come AURA supporta la mobilità dell'utente sarà descritto un semplice scenario.

Fred sta lavorando a casa all'organizzazione di una conferenza. Egli sta raccogliendo informazioni sulle possibili località e sui costi del servizio di catering attraverso Internet.

Fred lascia la casa e si dirige verso il suo ufficio. Quando Fred intende continuare a lavorare all'organizzazione della conferenza, Aura imposta un task in ufficio in modo che Fred possa riprendere il suo lavoro non appena entra in ufficio: un web browser sulle pagine visitate di recente, i video scaricati in pausa allo stesso punto in cui Fred li aveva fermati, e un foglio di lavoro contenente tutte le richieste inoltrate. Da quando sul muro dell'ufficio di Fred è stato montato un grande schermo, egli preferisce visualizzare su di esso i video e le pagine web, lasciando il monitor per la visualizzazione del foglio di lavoro.

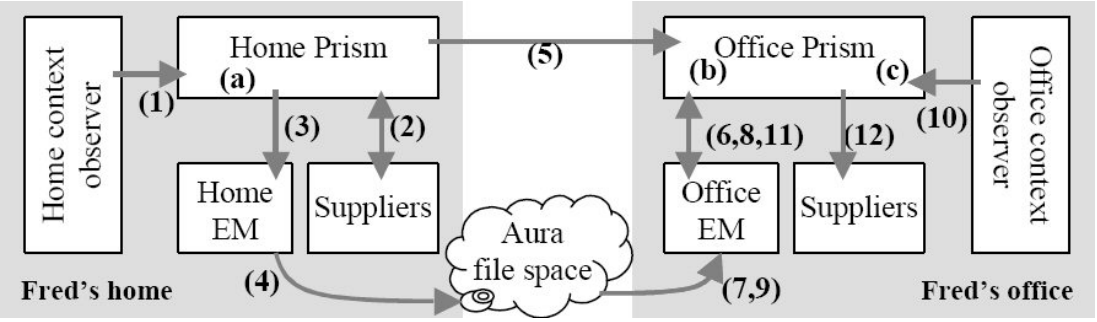


Fig. I.8 Fred va da casa in ufficio.

Fred sta lavorando a casa quando *l'"Home Context Observer"* nota che Fred sta andando via ed informa *Prism* – interazione (1) in *Fig.I.9* – che subisce una transizione di stato (a), e comprende che bisogna sospendere il task in corso a casa.

A questo punto *Prism* richiede lo stato corrente di ogni servizio in uso nel task corrente – interazione (2). Nell'interazione (3), *"Home Prism"* ordina all'*"Home*

Environment Manager" di deallocare le risorse impiegate e memorizzare i file coinvolti in un file server globalmente accessibile – interazione (4).

Dopo la verifica del programma di Fred, *Prism* deduce che probabilmente si dirigerà in ufficio, e (5) trasmette l'informazione all'*"office Prism"*. Questo provoca la transizione di stato (b) in *"office Prism"*, che richiede all'*"Office EM"* (6) di recuperare la descrizione aggiornata dei tasks su cui Fred ha lavorato – interazione (7). Dalla descrizione esso capisce quali file servono a Fred per continuare a lavorare e chiede all'*"Office EM"* di recuperarli – interazione (8). Quest'ultimo esegue l'operazione richiesta – interazione (9).

Non appena l'*"Office Context Observer"* rileva la presenza di Fred in ufficio informa *Prism* (10) causandone la transizione di stato (c). Quest'ultimo richiede ai suppliers i servizi coinvolti nel task (11) e successivamente ne ripristina lo stato di esecuzione (12).

I.3.2 GAIA: UN MIDDLEWARE PER PERVASIVE ENVIRONMENTS

GAIA ([14], [28],[29]) è un'infrastruttura middleware creata dai ricercatori del Dipartimento di Computer Science nell'Università dell'Illinois, che mira a gestire "Spazi Attivi" *(Active Spaces)*.

Nell'ambito di questo progetto di ricerca, vengono definiti i concetti di: "Spazio fisico" (*Physical Space*) una regione geografica con confini fisici limitati e ben definiti, contenente oggetti e dispositivi eterogenei collegati in rete e popolato da utenti che svolgono attività; "Spazio attivo" uno luogo fisico coordinato da un'infrastruttura software sensibile al contesto che consente agli utenti mobili di interagire e configurarsi con l'ambiente fisico e digitale in maniera automatica [29].

L'idea di base è quella di estendere la portata dei sistemi di calcolo tradizionali per includere le apparecchiature e lo spazio fisico che circondano le macchine e permettere, alle entità fisiche e virtuali, di interagire con il sistema: gli Spazi Fisici diventano sistemi interattivi, in altri termini, Spazi Attivi!

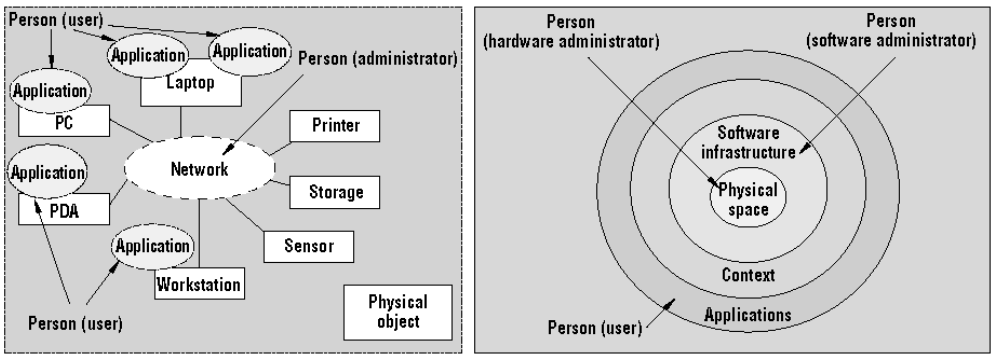


Fig. I.9 Physical Space (sinstra) and Active Space (destra) in GAIA.

Gaia cerca di portare le funzionalità di un *sistema operativo* (come gli eventi, i segnali, il file system, la sicurezza, i processi, i gruppi di processi) in uno spazio attivo, naturalmente estendendo questi concetti ed aggiungendo i nuovi concetti di contesto, ubicazione, dispositivi di calcolo mobili ed attuatori.

L'obiettivo è quindi progettare e implementare un *"sistema operativo middleware"* che gestisca le risorse contenute in uno Spazio Attivo.

Tale sistema deve essere capace di:

- localizzare il dispositivo più adatto;
- rilevare nuovi dispositivi che si aggiungono al sistema;
- adattare il contenuto quando i formati dei dati non sono compatibili con i dispositivi di output.

Il prototipo realizzato opera in spazi fisici usati per insegnare: classi, uffici e stanze per conferenze.

L'ambiente è dotato di attrezzature quali: un sistema audio-sonoro programmabile, cinque pannelli al plasma, HDTV, webcams, bluetooth, ethernet senza fili, apparecchiature di impronta digitale, telefoni intelligenti e la tecnologia di localizzazione Ubisense.

L'architettura di Gaia è formata essenzialmente da quattro componenti:

- *Gaia Kernel*: è un sistema di gestione e sviluppo di oggetti distribuiti (in particolare oggetti CORBA) ed è costituito da un insieme interconnesso di servizi di base di supporto per le applicazioni: *context service, event manager, presence service, security service* e *component repository*;
- *Gaia Application Framework*: modella le applicazioni come collezioni di componenti distribuiti, analizzando le risorse hardware disponibili nello spazio

attivo o quelle relative ad un particolare dispositivo; fornisce funzionalità per alterare la composizione delle applicazioni dinamicamente; è *context–sensitive*; implementa un meccanismo che supporta la creazione di applicazioni indipendenti dallo spazio attivo e fornisce politiche per adattarsi a differenti aspetti delle applicazioni inclusa la mobilità.

- *QoS Service Framework:* si occupa della gestione delle risorse per le applicazioni sensibili alla QoS e adatta dinamicamente tali applicazioni, determinando i nodi appropriati per la loro istanza, in base ad una selezione tra configurazioni in accordo con la disponibilità di risorse, con il servizio di *discovery* e con protocolli di assegnazione di risorse multiple, ossia traducendo i requisiti di alto livello relativi alla QoS in requisiti di sistema;
- *Application Layer:* che contiene le applicazioni e fornisce le funzionalità per registrare, gestire e controllarle attraverso i servizi del Kernel di Gaia.

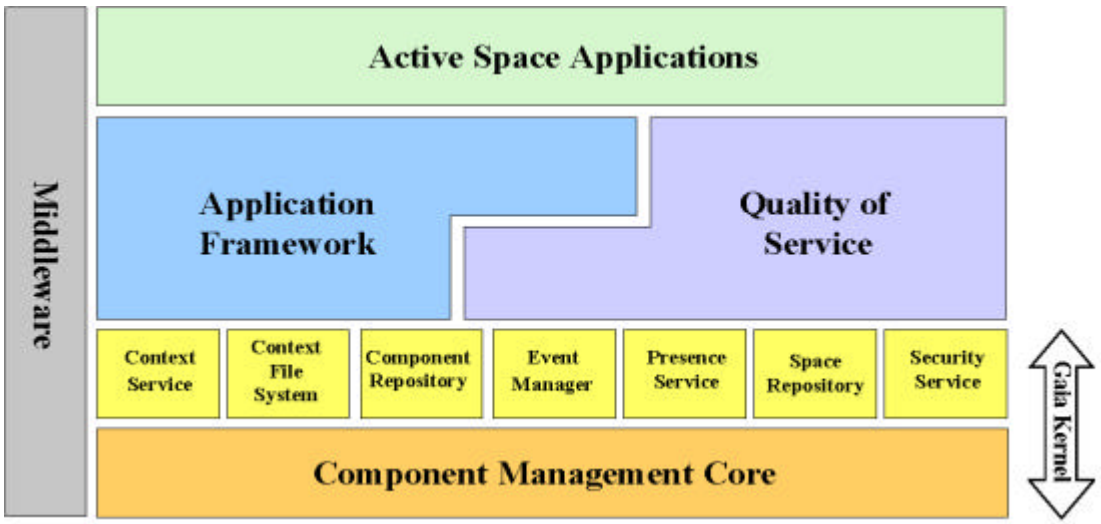


Fig. I.10 L'infrastruttura di GAIA.

In definitiva, GAIA cerca di portare le funzionalità di un sistema operativo in uno spazio attivo (come gli eventi, i segnali, il file system, al sicurezza, i processi, gruppi di processi), naturalmente estendendo questi concetti ed aggiungendo il nuovo concetto di contesto.

I.3.3 COOLTOWN

Il progetto realizzato presso i laboratori della Hewlett-Packard, Cooltown [10], offre un modello di sistema di *pervasive computing* che combina tecnologie web, reti wireless e dispositivi portatili per creare un ponte virtuale tra utenti mobili e entità fisiche e servizi elettronici [20].

La ragione per cui HP ha scelto il web come tecnologia alla base del suo prototipo di ambiente pervasivo è dovuta alla sua facile adozione, essendo l'ambiente di *distributed computing* più diffuso, rispetto ad una qualsiasi altra tecnologia da sviluppare ex novo. Inoltre il web offre la possibilità di accesso dovunque, possiede requisiti software leggeri, permette di operare sia localmente che su scala globale .

E' un progetto focalizzato soprattutto sul livello dell'infrastruttura necessaria per supportare utenti mobili con device portatili e non tanto sul livello dei device e della comunicazione wireless. Si lavora al livello "web server" con lo scopo di *estendere i modelli del web a nuove aree*.

Nella visione di Cooltown, ogni entità, oggetto, posto, persona, è dotata di una rappresentazione web (web presence). Un oggetto fornisce la propria URL, questa punta alla homepage, localizzata su un web server dedicato, attraverso la quale può essere controllato dall'utente: si può fare "click" sulle cose.

Un posto è una collezione di oggetti che hanno presenza web. Una persona è rappresentata da una web page con links che rappresentano servizi cui possono accedere altri individui per comunicare con lei.

In Cooltown, dove il protocollo di comunicazione è HTTP, è possibile comunicare con i dispositivi anche se non si è direttamente agganciati ad una rete globale adoperando ad esempio tecnologie come Bluetooth o IrDA, che risultano sufficienti per individuare, ad esempio, una stampante ed avviare il processo di stampa di un documento.

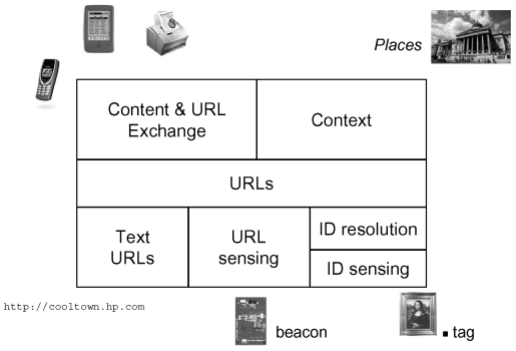


Fig. I.11 La visione di Cooltown.

L'infrastruttura per supportare tale web presence è articolata in tre livelli:

- ✍ *Acquisizione URL (Bottom layer)*: tale livello permette all'utente di acquisire URLs dall'ambiente e da entità fisiche nei dintorni. L'acquisizione dell'URL può avvenire in diversi modi:
  - *Text URLs* : l'utente digita l'URL di un oggetto o posto;
  - *Auto Service Discovery*: si usano protocolli di service discovery;
  - *URL Sensing* : i PDA hanno almeno un dispositivo short-range, con il quale, puntandolo o posizionandolo opportunamente, si possono leggere URL ed altri identificatori associati a posti ed oggetti grazie a: segnali *IR & RF , Codici a barre, Etichette elettroniche*. Il *sensing* può avvenire in maniera diretta: il segnalatore o etichetta presenta direttamente l'URL di una risorsa web; oppure in maniera indiretta: viene presentato un identificatore come ad es. ISBN o un UPc barcode. Poi per ottenere un URL si usa un *resolver*, un servizio che ritorna un URL a partire da un dato identificatore.
- ✍ *Content exchange (Middle layer)*. Lo scambio di Contenuti è opposto al browsing: l'utente nomade può inserire del contenuto nell'infrastruttura pervasiva piuttosto che estrarne soltanto. L'inserimento (pushing o 'posting') di contenuto lavora tra un source (sorgente) di contenuto e un sink (pozzo) di contenuto. Ad esempio: *un utente entra in una sala riunione con una fotocamera (source), fotografa il contenuto di una lavagna, stampa l'immagine su una stampante (sink) della stanza*. L'operazione di *post* può essere fatta sia in modo diretto che in modo indiretto. Nel primo caso si tratta di una semplice operazione di inserimento: la sorgente apre una connessione verso

il sink e scrive l'effettivo contenuto. L'interazione richiede dei metadati relativi ai dati del contenuto. Il formato dei metadati deve essere concordato e stabile nel tempo: in tale approccio si usa un formato XML con codifica ASCI trasportato in una MIME entity. Nel secondo caso il contenuto inserito contiene links ipertestuali ad altri contenuti, quindi si ha lo scambio di URL invece che di contenuti diretti .

- *Context and Physical discovery(Top layer)*. A questo livello viene preso in considerazione il contesto dell'utente: i servizi differiscono a seconda del posto (place) da cui sono acceduti. La rappresentazione di un posto fornisce un contesto: un insieme di risorse correlate. L'implementazione del contesto di un posto è un web server con l'appropriato contenuto che descrive il posto. Nella web page del posto ci sono i link agli oggetti del posto dotati di presenza web. Un *place manager* fornisce accesso e configurazione per i dati associati ad uno o più posti:

  - mantiene directories di risorse di un posto e offre un'interfaccia per aggiungere, richiedere, rimuovere risorse;
  - agisce da resolver cercando risorse a partire dai loro identificatori.

Da quanto detto è evidente che una caratteristica fondamentale dell'architettura in esame è la *"location awareness"*, cioè la consapevolezza della locazione sia delle risorse che dell'utente.

Con XML si possono descrivere degli attributi semantici associati al link della locazione (location representation). In questo modo possono essere offerti servizi personalizzati ai clienti sulla base della loro locazione.

Il servizio web associato all'oggetto conosce la locazione fisica dell'oggetto e può aiutare l'utente a bcalizzare il servizio richiesto. Però, eseguire un'operazione di tracking dell'utente non è sempre attuabile in quanto:

a) Il mapping inverso da locazione semantica a locazione fisica non porta ad un'unica locazione, quindi il servizio non può usare le informazioni sulla locazione semantica per localizzare i client;

b) L' utente non deve necessariamente presentarsi al posto dove c'è l'oggetto fisico per poter accedere al servizio.

Le caratteristiche dei dispositivi dell'ambiente HP sono descritte in una web-form indipendente dal tipo di dispositivo stesso; ciò consente una semplice interazione tra dispositivi, mediante semplici URL, senza la necessità di dover istallare driver per ogni tipo di dispositivo incontrato.

Tuttavia, l'interazione macchina-macchina tra dispositivi sconosciuti è un aspetto ancora irrisolto poiché le web-interface sono essenzialmente rivolte all'uomo e non sono adatte ad una elaborazione automatica.

La sicurezza in Cooltown è gestita da un servizio di autorizzazione centrale che applica una certa politica di accesso a gruppi di servizi. A tale proposito è possibile individuare tre fasi:

- *Setup Phase*: il gestore genera delle chiavi e le comunica ai servizi di un dato gruppo. Tali chiavi hanno lo scopo di descrivere le credenziali che devono essere fornite dall'utente per accedere a tali servizi.
- *User Registration Phase*: quando l'utente entra nell'ambiente il gestore determina il gruppo di servizi ai quali egli può accedere. Genera un documento criptato di credenziali e solo i servizi appartenenti al gruppo accessibile dall'utente saranno in grado di decifrarlo. Le credenziali sono memorizzate sul dispositivo utente e sono richieste ogni qual volta si renda necessaria l'identificazione dell'utente, cioè quando egli richiede l'uso di un particolare servizio.
- *User Access Phase*: L'utente presenta le sue credenziali al servizio a cui vuole accedere. Il servizio decifra le credenziali per determinare la chiave di autenticazione. Inoltre il servizio deve verificare, sempre a partire dalle credenziali, se esso è il servizio effettivamente richiesto dall'utente.

I.3.4 CENTAURUS

Centaurus [19] definisce un'infrastruttura ed un protocollo di comunicazione per servizi software ed hardware eterogenei che devono essere resi disponibili agli utenti ovunque ne abbiano bisogno. Tali servizi intelligenti devono essere capaci di comprendere le necessità

dell'utente per fornirgli migliore supporto ed agevolarne le attività. Tale architettura si comporta da proxy attivo eseguendo servizi per conto dei client che li richiedono.

L'architettura di Centaurus consiste di quattro componenti:

- *Communication Manager*: è responsabile della comunicazione tra il client ed altri componenti Centaurus. Quando riceve informazioni da un client le invia al Service Manager; quando riceve dati dal Service Manager li valida e analizza l'header per decidere a quale client inoltrali. Può usare diversi moduli di comunicazione con il client a seconda del mezzo trasmissivo: Bluetooth, IR, CPDP,… Per la comunicazione si usa un linguaggio proprietario basato su XML, il *Centaurus Communication Markup Language* (CCML). Tale linguaggio è integrabile con linguaggi semantici come, ad esempio DAML+OIL
- *Service Manager:* controlla l'accesso ai servizi ed agisce da *"gateway"* tra i servizi ed i client. Quando un servizio fa lo start up, si registra presso il Service Manager, mandandogli il suo CCML file. Quando un nuovo cliente entra, il SM gli manda un oggetto *"ServiceList"*, che verrà dinamicamente aggiornato. Attraverso questa lista il client potrà selezionare un servizio e il SM gli invierà la descrizione CCML per quel servizio. In alternativa il client potrà invocare il servizio mandando un nuovo file CCML al SM e questo, se il servizio è disponibile, gli inoltra la richiesta.
- *Services*: sono oggetti che offrono funzionalità ai *client* Centaurus: controllare un interruttore di una lampada, stampare… Ogni servizio si registra presso il SM inviandogli un CCML file, con nome, identificatore, locazione, una breve descrizione, il suo "leasing period". Ogni volta che il suo stato cambia deve informare il Service Manager, o deve rinnovare il suo leasing. Accetta richieste solo dal Service Manager con cui si è registrato. I servizi contengono informazioni necessarie a localizzare il più vicino *Service Manager* ed a registrarsi ad esso. Una volta registrato, un servizio può essere acceduto da qualsiasi *client* attraverso il *Communication Manager*.
- *Clients*: implementano un'interfaccia utente per interagire con i servizi. Un *client* può accedere ai servizi forniti dal più vicino sistema Centaurus che si comporta come un proxy soddisfacendo le sue richieste.

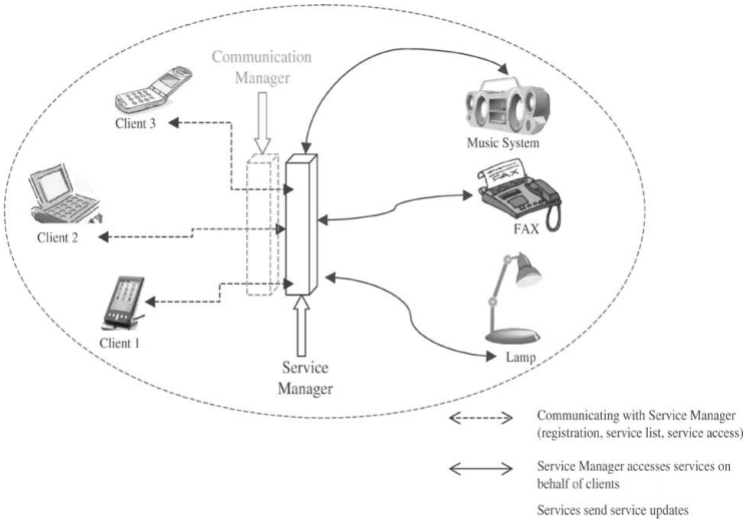


Fig. I.12 Architettura di Centaurus.

Tutti i componenti del modello, per la comunicazione usano un linguaggio proprietario, basato su XML, il Centaurus Communication Markup Language (CCML). In questo modo offrono al sistema un'interfaccia uniforme ed adattabile. Si tratta di un protocollo di trasporto efficiente, basato su messaggi.

E' costituito da due livelli: i moduli del livello I sono dipendenti dal mezzo fisico, mentre il livello II è indipendente dal mezzo. E' progettato per essere eseguito su un'ampia gamma di dispositivi di limitate capacità elaborative: non sfrutta caratteristiche di sistemi operativi avanzati come segnali e multithreading, al contrario di TCP che necessita del supporto per il signaling da parte del S.O.

In ogni caso, in presenza di reti wireless con banda limitata ed elevata latenza funziona meglio di TCP.

I.3.5 PROGETTO COBRA

All'Università del Maryland in Baltimora è stato sviluppato un modello architetturale per *pervasive environments* con particolare attenzione alle problematiche di context-aware computing.

Il cuore dell'architettura è caratterizzato da un'entità server specializzata ed intelligente, il Context Broker (da cui il nome CoBrA: Context Broker Architecture), che riceve informazioni legate al contesto dai dispositivi e dagli agenti presenti, le relaziona, definendo un modello centralizzato e condiviso dell'ambiente e di tutto ciò che si trova ad operare al

suo interno, e si preoccupa di mantenerlo, nel tempo, coerente e privo di inconsistenze in seguito a nuove acquisizioni di informazioni.

Un punto chiave nella realizzazione di questa architettura è lo sviluppo e l'utilizzo di una serie di ontologie comuni, attraverso le quali agevolare la comunicazione fra i diversi agenti e rappresentare lo stato del sistema.

Tale architettura è caratterizzata dal fatto di definire ontologie OWL per abilitare agenti a processare e ragionare sul contesto, includere un motore inferenziale per ragionare sull'informazione di contesto e individuare e risolvere inconsistenze, inoltre, applicare un approccio policy-based per controllare come condividere tra gli utenti le informazioni di contesto.

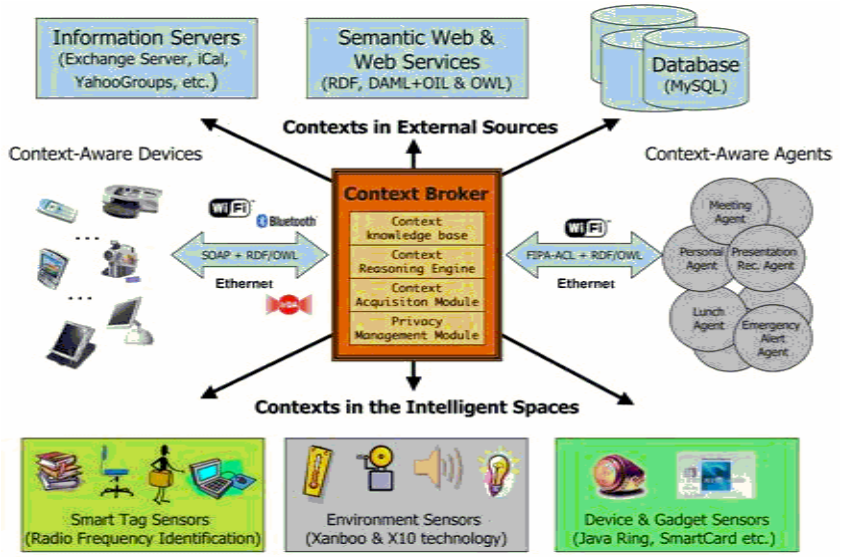


Fig. I.13 Architettura di COBRA.

L'architettura del context broker è costituita da quattro componenti funzionali ([8],[9]):

- *Context Knowledge Base*: immagazzina in maniera persistente la conoscenza del contesto. Fornisce un set di API per consentire agli altri componenti di accedere alle informazioni memorizzate. Contiene anche l'ontologia dello specifico ambiente e le regole euristiche associate ad esso;
- *Context Reasoning Engine*: è un motore che applica algoritmi di inferenza sulle informazioni di contesto immagazzinate. Si possono avere inferenze che usano le ontologie per dedurre conoscenza di contesto e inferenze che usano la conoscenza per rilevare e risolvere inconsistenze;

- *Context Acquisition Module*: è una libreria di procedure che rappresenta una sorta di *middleware* per l'acquisizione di informazioni di contesto;
- *Policy Management Module*: è un insieme di regole di inferenza definite sia per valutare i privilegi di accesso, associati alle differenti entità, sia per condividere particolari frammenti delle informazioni di contesto e ricevere notifiche dei cambi di contesto.

Il modello con broker centralizzato affronta due problemi chiave del pervasive computing: supportare dispositivi con limitate capacità e gestire la *privacy dell'utente.* Grazie ad un context broker che opera su un computer fisso, la complessità di acquisire e ragionare su informazioni contestuali è spostata dai dispositivi mobili limitati verso il broker dotato di elevate risorse, inoltre le complicazioni nel garantire security, trust, and privacy polices sono semplificate dalla presenza di un manager centralizzato.

In base al progetto di tale architettura è stato realizzato il prototipo di una *"intelligent meeting room"* che usa CoBrA.

Tale modello di prova offre servizi e informazioni ai partecipanti *sulla base delle loro necessità correnti* e permette agli utenti di controllare l'uso e la condivisione della loro locazione e del contesto sociale.

In futuro, gli sviluppatori di questa architettura prevedono di migliorare il meccanismo di inferenza logica, a tal proposito si sta investigando sull'uso del frameworkTheorist, un metainterprete Prolog: secondo tale approccio le premesse consistono sia di ***fatti*** (assiomi dati per certi) che di ***assunzioni*** (istanze di ***possibili ipotesi*** che possono essere assunte se sono coerenti con i fatti), in altre parole si affianca il default reasonig all'abduction reasoning.

Tutte le informazioni di contesto acquisite dal Context Broker sono viste come sue osservazioni dell'ambiente. Quando riceve un'osservazione, il Context Broker, prima usa l'abduzione per determinare le possibili cause, poi usa il ragionamento di default per predire cos'altro conseguirà dalla causa.

I.4 CARATTERISTICHE DI UN SISTEMA PERVASIVO

Nei paradigmi di *ubiquitous computing*, servizi ed informazioni sono virtualmente accessibili dovunque, in ogni istante attraverso qualsiasi dispositivo, ma considerazioni di carattere amministrativo, territoriale e culturale ci inducono ad analizzare l'*ubiquitous computing* in ambienti discreti, dai confini ben definiti, come per esempio case, uffici, sale convegno, aeroporti, stazioni, musei, etc. In altre parole, è bene considerare il mondo suddiviso in tanti domini pervasivi, piuttosto che vederlo come un unico enorme sistema [21].

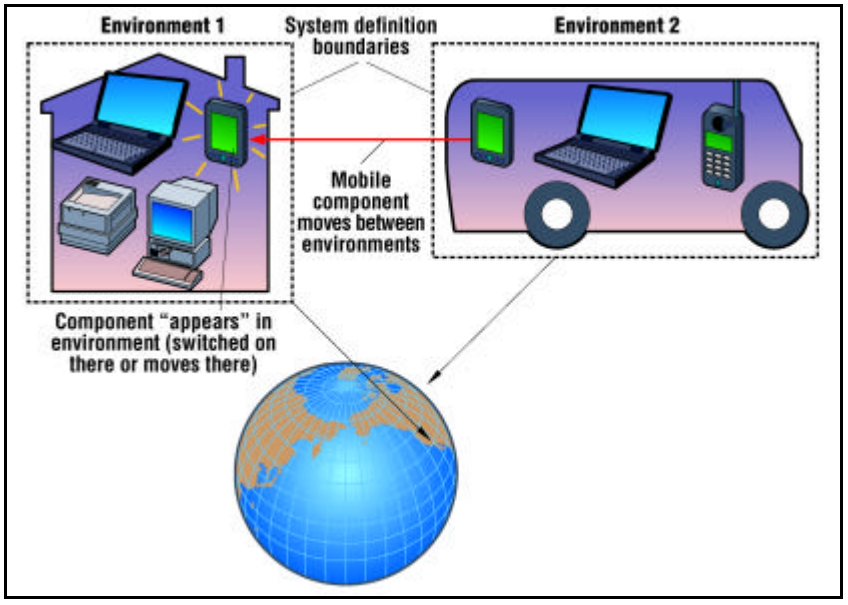


Fig. I.14 Il mondo suddiviso in ambienti pervasivi con confini ben definiti.

Ogni singolo ambiente è contraddistinto da componenti o unità software che implementano astrazioni di servizi, clienti, risorse o applicazioni ed, in generale, ogni ambiente è costituito da un'infrastruttura fissa e da una serie di elementi mobili che, in maniera del tutto impredicibile, entrano a far parte del sistema e lo abbandonano continuamente, talvolta migrando fra i diversi domini.

A partire da queste considerazioni e dai progetti presentati nei paragrafi precedenti è possibile individuare una serie di aspetti di carattere generale e indicazioni di fondamentale importanza per l'analisi, lo sviluppo e l'implementazione di un sistema di pervasive computing [35].

I.4.1 CONTEXT INFORMATION E RUN-TIME ADAPTATION

Una prerogativa di un sistema pervasivo è la capacità di ottenere informazioni sugli utenti e sullo stato dell'ambiente, come ad esempio, la posizione e l'identità dei singoli utenti e la disponibilità delle risorse.

Questo viene realizzato collezionando dati grezzi provenienti da una moltitudine di sorgenti. Tali dati saranno poi processati e trasformati in informazioni di contesto da condividere tra le diverse applicazioni in esecuzione sui vari dispositivi, cercando di mantenere la scalabilità, di garantire la sicurezza delle informazioni, inibendo accessi non autorizzati e rispettando la privacy individuale.

In base a queste informazioni, le applicazioni possono adattare il loro comportamento in maniera diversa a seconda dei casi. E' possibile, a tal proposito, definire due tipi di adattamento: funzionale o strutturale.

Un esempio di adattamento funzionale può essere rappresentato da un'applicazione che fornisce news in un ambiente: il tipo di notizie che vengono fornite potrebbe essere determinato in base a chi si trova nell'ambiente o all'ora della giornata.

Un esempio di adattamento strutturale, invece, può essere rappresentato da un'applicazione musicale: a seconda se l'utente è solo oppure no all'interno dell'ambiente, l'applicazione potrebbe utilizzare il sistema audio del laptop dell'utente, oppure quello dell'ambiente stesso.

Per realizzare quanto appena detto, occorre necessariamente avere una percezione dell'ambiente ragionevolmente accurata, e disporre di meccanismi per il rilevamento e la correzione di informazioni di contesto inattendibili o contrastanti.

I.4.2 TASK RECOGNITION E PRO-ACTIVITY

Un sistema di *pervasive computing* dovrebbe essere capace, a partire dalle informazioni di contesto collezionate, di elaborare lo stato corrente dell'ambiente e le intenzioni dell'utente e modificare dinamicamente il proprio comportamento per assistere l'utente stesso nelle sue attività.

Diversamente dai sistemi di *computing* convenzionali in cui il comportamento del computer è principalmente composto di risposte all'interazione con l'utente, il *pervasive*

computing mira alla realizzazione di un modello in cui i dispositivi sono la parte attiva nell'interazione con l'utente. Quindi se la tecnologia corrente si basa su persone che indicano ai computer ciò che devono fare, la nuova generazione di tecnologie dovrebbe essere basata su computer capaci di comprendere quello che le persone stanno facendo e quello che esse desiderano.

I modelli che tengono conto dell'esperienza, del passato, rappresentano un importante strumento per la caratterizzazione di un sistema pervasivo, perché favoriscono la *pro-activity* del sistema, ossia consentono di determinare, in accordo con i precedenti comportamenti dell'utente, le azioni ottimali che l'utente stesso deve eseguire in determinate situazioni.

I.4.3 RESOURCE ABSTRACTION E DISCOVERY

Le risorse di un sistema pervasivo dovrebbero essere rappresentate in maniera astratta (magari attraverso le caratteristiche anziché attraverso i nomi) cosicché possano facilmente essere selezionate in base a requisiti di tipo generale oltre che specifico. Potrebbe essere necessario definire alcune convenzioni di nomi per evitare che differenti risorse descrivano se stesse adoperando gli stessi termini offrendo, però, servizi diversi.

Dovrebbero essere previsti anche dei meccanismi per scoprire, interrogare ed interagire con le risorse nell'ambiente, per consentire l'introduzione di nuovi componenti senza onerose operazioni di configurazione e di riconfigurazione dei componenti esistenti.

I.4.4 ETEROGENEITY E SERVICE UI ADAPTATION

Un ambiente pervasivo è caratterizzato da una moltitudine di dispositivi eterogenei, in numero variabile, come laptop, PDA, telefoni mobili.

Alla riduzione delle dimensione dei dispositivi dell'ambiente corrisponde una maggiore crescita del numero di dispositivi connessi ed una intensificazione delle interazioni uomo-macchina.

Lo sviluppo tradizionale fornisce servizi tipicamente distribuiti e istallati separatamente per ogni classe di dispositivo e famiglia di processore. Gli scenari di *computing* pervasivo,

invece, portano alla conclusione che distribuire ed installare servizi per ogni classe e famiglia diventa ingestibile, specialmente in un'area geografica molto estesa.

E' importante, quindi, che i servizi forniscano agli utenti interfacce che possano adattarsi alle caratteristiche del dispositivo *client*, senza stravolgere le funzionalità del servizio stesso [13]. Ad esempio, un servizio adibito al controllo dell'illuminazione in una stanza è libero di fornire un'interfaccia utente di tipo grafico per un *client* PDA, ma deve necessariamente garantire anche una rappresentazione testuale della stessa UI per un utente che adopera un telefono cellulare, senza, peraltro, dover cambiare l'implementazione del servizio stesso.

I.4.5 SECURITY E PRIVACY

Un sistema pervasivo generalmente gestisce grosse quantità di informazioni, molte delle quali acquisite tramite sensori e riguardanti gli utenti.

Dal punto di vista dell'utente, è desiderabile che vengano rispettati i principi di privacy e che sia garantita la sicurezza di queste informazioni.

Ad esempio, risulta fastidioso sapere che la stazione di polizia più vicina possa conoscere in quale stanza ci si trovi nella propria casa, attraverso i rilevatori di moto del sistema d'allarme, o quanto alcool si sta consumando, deducendo questa informazione dal sistema che gestisce l'inventario degli alimenti.

D'altra parte, in molte situazioni, una certa perdita di privacy può essere tollerata, come ad esempio in situazioni di pericolo.

Per aggiungere sicurezza alle informazioni, i servizi nell'ambiente non dovrebbero consentire accessi non autorizzati: ad esempio, a casa propria, non dovrebbe essere possibile per un ospite, aumentare il riscaldamento o controllare il funzionamento del forno [4].

I.4.6 FAULT-TOLERANCE E SCALABILITY

Gli ambienti pervasivi costituiscono sistemi "perennemente" attivi. Pertanto, un componente che subisce un guasto non deve compromettere il funzionamento generale dell'intero sistema, né richiedere una complessa tecnica di gestione.

I componenti che cadono dovrebbero automaticamente ripartire, laddove possibile, magari adoperando, ad esempio, memorie di stato persistenti che consentano di effettuare *resume* rapidi ed efficaci.

Abbiamo visto che gli ambienti pervasivi sono caratterizzati anche da una forte dinamicità: dispositivi possono aggiungersi all'ambiente ed abbandonarlo in qualsiasi momento; alcuni servizi possono cadere e presentarsene altrettanti nuovi; gli stessi utenti possono entrare ed uscire dall'ambiente secondo la propria volontà.

Il sistema deve garantire scalabilità, ossia essere in grado di gestire e assicurare il suo funzionamento anche in seguito all'aggiunta di componenti; allo stesso tempo, i nuovi dispositivi e servizi introdotti nell'ambiente non dovrebbero interferire con quelli esistenti.

II. TECNOLOGIE PER IL PERVASIVE COMPUTING

II.1 LA COMUNICAZIONE NEI SISTEMI DISTRIBUITI

Un sistema distribuito consiste in un insieme di computer che comunicano su di una rete per coordinare le azioni e i processi di un'applicazione [25]. Le tecnologie per la realizzazione di sistemi distribuiti hanno suscitato molto interesse negli ultimi anni, anche grazie alla proliferazione dei sistemi e servizi basati sul Web.

Tecnologie consolidate come la comunicazione tra processi e l'invocazione remota, i naming service, la sicurezza e la crittografia, i *file system* distribuiti, la replicazione dei dati e i meccanismi di transazione distribuita, forniscono una solida infrastruttura di *run-time* che supporta le applicazioni di rete odierne.

Il modello dominante è ancora la tradizionale architettura *client-server*, ma lo sviluppo di applicazioni in ambito distribuito si sta basando sempre di più sia sull'impiego di supporti *middleware*, che forniscono astrazioni di alto livello, come oggetti distribuiti condivisi, sia su altri generi di servizi, come per la comunicazione sicura o per l'autenticazione.

Inoltre sia Internet, con i suoi protocolli base, sia il World Wide Web, ad un livello più alto, stanno diventando una piattaforma standard per le applicazioni distribuite. Difatti Internet e le sue risorse possono essere viste come un ambiente globale in cui ha luogo l'elaborazione informatica. Di conseguenza l'attenzione si focalizza su nuovi standard e

protocolli di alto livello, come XML, mentre passano in secondo piano aspetti di basso livello, come ad esempio le peculiarità di un sistema operativo.

II.2 PARADIGMI DI COMUNICAZIONE

Esistono diversi modi attraverso i quali componenti di applicativi software risiedenti su macchine differenti possono comunicare fra loro adoperando una rete. Una tecnica di basso livello è quella di utilizzare direttamente le interfacce offerte dal livello trasporto, come il meccanismo delle *socket*, assieme ad un protocollo di comunicazione pensato ad hoc per l'utilizzo specifico.

Comunque programmare a questo livello di astrazione è consigliabile solo in particolari circostanze poiché demanda completamente al programmatore la gestione di complessi problemi come la sicurezza, l'eterogeneità e la concorrenza.

Nella maggior parte dei casi, invece, è preferibile scegliere tra una serie di protocolli e ambienti di più alto livello quello che meglio si adatta alle proprie esigenze. Alcuni di questi protocolli sono *self-contained* e possono essere usati in qualsiasi programma applicativo con un *overhead* addizionale basso o addirittura nullo. Altri protocolli ed ambienti, invece, sono vincolati a specifici linguaggi di programmazione o a particolari piattaforme di esecuzione [25].

II.2.1 REMOTE PROCEDURE CALL

Un classico schema di comunicazione, che ben si addice al modello *client-server*, è la chiamata a procedure remote (RPC). In questo modello, un componente agisce da client quando richiede un servizio ad un altro componente, da server quando, invece, è lui a rispondere alle richieste. RPC effettua una chiamata ad una procedura esterna che risiede in un differente nodo della rete quasi con la stessa semplicità con cui invoca una procedura locale. Argomenti e valori di ritorno sono automaticamente impacchettati in un formato definito dall'architettura e spediti tra procedure locali e remote.

Per ogni procedura remota, il *framework* RPC sottostante ha bisogno di una procedura stub dal lato client (che agisce da *proxy*) e di un oggetto simile lato server. Il ruolo dello

stub è prendere i parametri passati attraverso una regolare procedura locale ed inviarli al sistema RPC (che deve risiedere su entrambi i nodi). Dietro le quinte, il sistema RPC coopera con gli stub di ambo i lati per trasferire argomenti e valori di ritorno sulla rete.

Per facilitare la creazione di stub, sono stati realizzati speciali tool. Il programmatore fornisce i dettagli di una chiamata RPC in forma di specifiche, espresse attraverso l'*Interface Definition Language* (IDL), poi viene utilizzato un compilatore IDL che genera, a partire da tali specifiche, gli stub in maniera automatica.

I framework RPC, anche se sono tipicamente invisibili ai programmatori, sono diventati una tecnica consolidata poiché rappresentano i meccanismi di trasporto su cui si basano le più generali piattaforme *middleware* di cui si parlerà in seguito.

II.2.2 XML-BASED RPC

Sebbene i sistemi RPC trattano esplicitamente aspetti di interoperabilità in sistemi aperti, i programmi *client* e *server* che fanno uso di questo principio sono vincolati ad un singolo *framework* RPC. La ragione principale è che ciascun *framework* definisce la propria tecnica di codifica per le strutture dati. Nonostante queste differenze, le semantiche base della maggior parte dei sistemi RPC sono simili poiché si fondano su chiamate a procedure sincrone in un formato espresso in sintassi C-like.

L'idea nuova è stata quella di utilizzare XML per definire la sintassi delle richieste e delle risposte RPC, consentendo a differenti sistemi RPC di poter comunicare tra loro. In pratica XML è usato per definire un sistema di tipi che può essere adoperato per scambiare dati tra *client* e *server*. Questo sistema specifica tipi primitivi, come interi, *floating point*, stringhe di testo e fornisce i meccanismi per aggregare istanze di tipi primitivi in tipi composti per ottenere nuove categorie di dati.

Uno dei primi *framework* RPC basati su XML è stato SOAP (Simple Object Access Protocol) [40] definito inizialmente da un consorzio di compagnie tra le quali figuravano Microsoft, IBM, SAP, ma che oggi è invece divenuto un progetto *open source* in fase di standardizzazione presso il World *Wide Web Consortium* (W3C).

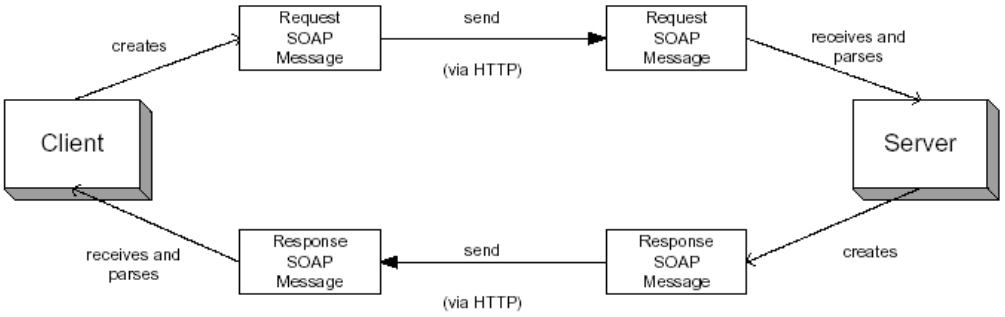


Fig. II.1 Interazione client-server usando SOAP.

SOAP [40] definisce solo la struttura del messaggio ed alcune regole per elaborarlo, rimanendo quindi ad un livello alto e completamente indipendente dal protocollo di trasporto sottostante. Aspetti chiave di SOAP sono, quindi, la sua estendibilità dovuta all'uso di schemi XML e l'utilizzo del protocollo HTTP come meccanismo di trasporto tra *client* e *server*, e quindi del Web come infrastruttura di comunicazione.

II.2.3 REMOTE METHOD INVOCATION

Mentre RPC è ragionevolmente ben adattabile al paradigma di programmazione procedurale, non è direttamente applicabile allo stile di programmazione *object oriented* che ha riscosso molta popolarità negli ultimi anni.

In questo contesto fa la sua comparsa RMI, una nuova tecnica di comunicazione remota per sistemi *Java-based*. Simile a RPC, integra il modello ad oggetti distribuiti nel linguaggio Java in modo naturale, lavorando direttamente sugli oggetti esistenti ed evitando di descriverne i metodi in un file di definizione diverso.

In un classico sistema RPC, il codice dello stub lato client deve essere generato e linkato nel client prima che una procedura remota possa essere chiamata; RMI, invece, è più dinamico perché, sfruttando la mobilità del codice Java, gli stub necessari per l'invocazione possono essere scaricati a tempo di compilazione da una locazione remota, per esempio direttamente dal server.

Internamente, RMI fa uso della serializzazione degli oggetti per trasmettere tipi di oggetti arbitrari sulla rete e poiché il codice scaricato può essere dannoso per il sistema, usa un *security manager* per operare dei controlli di attendibilità.

II.3 MIDDLEWARE ED INFRASTRUTTURE DI COMUNICAZIONE

I middleware e le infrastrutture software per sistemi distribuiti forniscono strumenti per la comunicazione ai componenti applicativi e gestiscono aspetti come l'eterogeneità fra le varie piattaforme, dovuta a differente hardware, software, sistema operativo e linguaggio di programmazione.

Inoltre forniscono un set di servizi standard che in genere sono indispensabili per lo sviluppo di applicazioni distribuite, come servizi di directory, sicurezza e crittografia.

II.3.1 CORBA

Una delle più usate infrastrutture per sistemi distribuiti basata sul modello *object-oriented* è CORBA (*Common Object Request Broker Architecture*), che è supportata da un esteso consorzio di industrie.

Il primo standard CORBA è stato introdotto nel 1991 ed è stato oggetto di continue e significative revisioni.

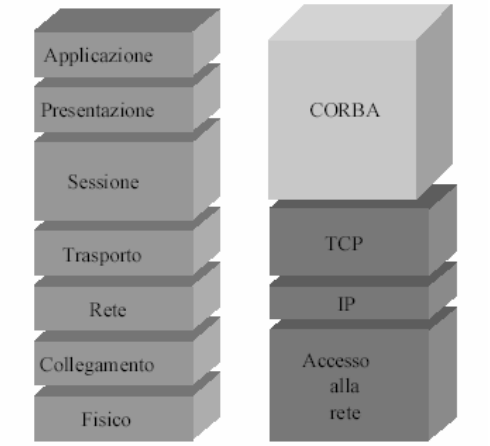


Fig. II.2 CORBA rispetto alla pila OSI.

Una parte delle specifiche descrive l'*Interface Definition Language* (IDL) che deve essere supportato da tutte le implementazioni di CORBA. L'IDL di CORBA si rifà al C++ ed è usata dalle applicazioni per definire i metodi di un oggetto che è possibile utilizzare esternamente.

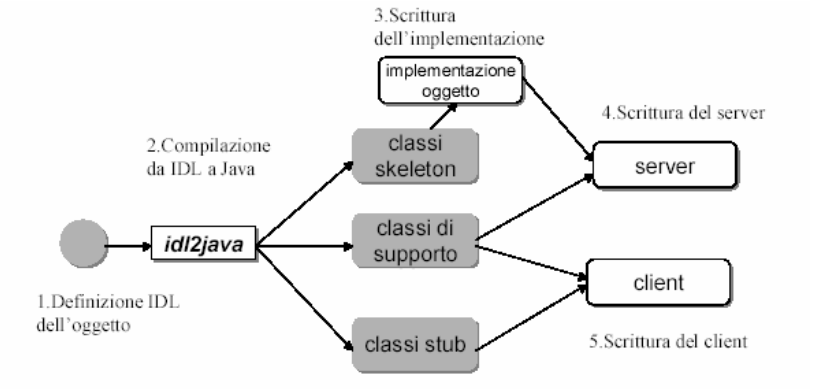


Fig. II.3 Ciclo di sviluppo di un'applicazione CORBA attraverso Java.

E' simile all'IDL di RMI. Tramite un compilatore IDL si ottengono file contenenti del codice di supporto per l'aggancio all'ORB, lo *stub* (per il client), lo *skeleton* (per il server). Lo *stub* e *skeleton* realizzano le funzionalità di *marshalling* e di *unmarshalling* dei dati. Lo *skeleton*, inoltre, dal lato server, coopera con un *Object Adapter* per le operazioni che riguardano l'attivazione dell'oggetto.

Il componente centrale di un sistema CORBA è l'O*bject Request Broker* (ORB), una sorta di bus software. Esso fornisce un meccanismo per comunicare in maniera trasparente richieste di client alle implementazioni degli oggetti server.

Questo semplifica la programmazione distribuita, disaccoppiando il client dai dettagli di invocazione dei metodi: quando, infatti, un client invoca un'operazione, l'ORB si preoccupa di trovare l'implementazione dell'oggetto, di attivarlo se necessario adoperando l'OA, di inoltrargli la richiesta, e ritornare una eventuale risposta al chiamante.

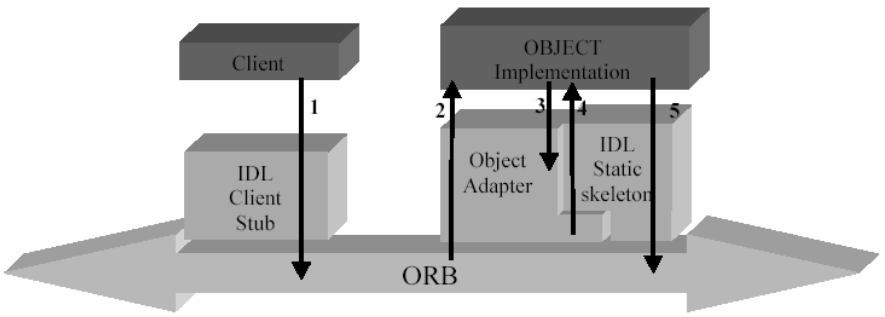


Fig. II.4 Interazione client-server in CORBA.

CORBA prevede sia collegamenti di tipo statico che di tipo dinamico tra client e server. Nel caso statico, codice proveniente dal server (stub del client) viene compilato con il client stesso. Di conseguenza l'interfaccia dell'oggetto server deve essere conosciuta dal client a tempo di compilazione. Nel caso dinamico è introdotta la possibilità di costruire ed invocare richieste su oggetti non conosciuti a tempo di compilazione.

CORBA supporta, quindi, sia la trasparenza di comunicazione, mascherando le chiamate remote come semplici invocazioni locali, sia quella di locazione, nascondendo al client tutti i dettagli sulla localizzazione degli oggetti sulla rete e sul loro linguaggio di implementazione.

La comunicazione tra ORB nella rete avviene tramite i protocolli GIOP e IIOP che implementano il marshalling, l'unmarshalling e la rappresentazione esterna dei dati.

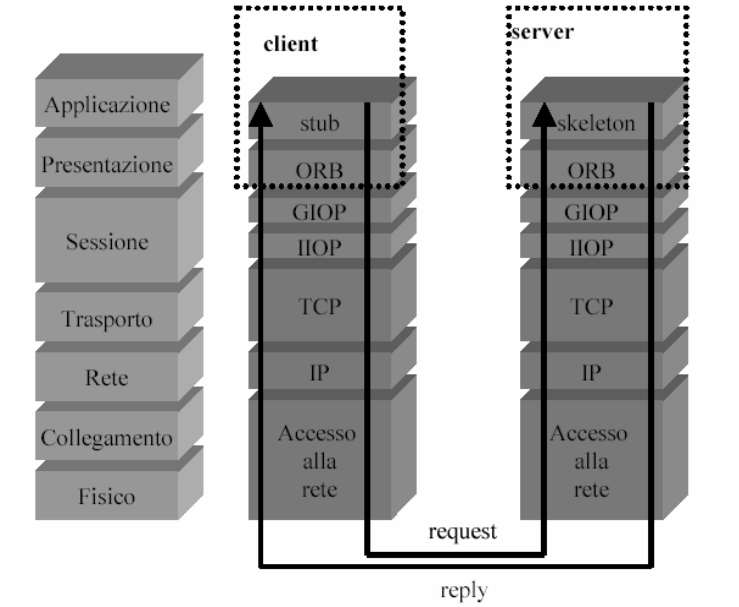


Fig. II.5 Collocazione dei protocolli GIOP-IIOP rispetto alla pila OSI.

GIOP è un protocollo astratto che specifica l'insieme di tipi di messaggi utilizzabili tra un client e un server, una sintassi standard nel trasferimento dati delle interfacce IDL, un formato standard per ogni messaggio utilizzabile. In più GIOP richiede un sistema di trasporto con connessione: IIOP è il mapping specifico di GIOP sul protocollo TCP/IP.

In aggiunta all'ORB, CORBA definisce un framework di oggetti supplementari, limitandosi a fornire le specifiche ma non le implementazioni:

- gli oggetti dei servizi (*Object Services*) costituiscono un insieme di servizi di uso generico per l'implementazione di applicazioni distribuite;
- le facilitazioni comuni (*Common Facilities*) sono un insieme di componenti che forniscono funzionalità applicative di uso comune, come stampa, accesso a DB, accesso a servizi di e-mail;
- le interfacce di dominio (*Domain Interfaces*) specificano componenti che forniscono servizi specifici per particolari domini applicativi;
- gli oggetti delle applicazioni (*Application Interfaces*) corrispondono alla nozione classica di applicazione prodotta da una particolare organizzazione e di conseguenza non sono standardizzati.

Di seguito è fornita una descrizione sommaria di alcuni *object service* particolarmente importanti per i sistemi distribuiti: il *Naming Service*, il *Trading Service*, l'*Event Service* ed il *Notification Service*.

Il *Naming Service* permette ad un client di trovare un oggetto attraverso il suo nome e ad un server di registrare i suoi oggetti dandogli un nome.

Il *Trading Service* è un servizio di locazione di oggetti, così come il *Naming Service*, ma lavora ad un livello più astratto. Nel *Naming Service* gli oggetti vengono ricercati attraverso un nome. Nel *Trading Service* si fanno delle richieste per ricevere liste di oggetti che soddisfano certe caratteristiche o proprietà (inserite nella richiesta) [34].

L' *Event Service* è il servizio che si occupa della gestione di eventi CORBA, dove un evento è definito come una occorrenza di oggetto che è di interesse per uno o più oggetti [44], fornendo in tal modo un supporto per la comunicazione asincrona ad accoppiamento lasco tra oggetti. Per disaccoppiare la comunicazione tra un produttore di eventi (supplier) ed un consumatore (consumer) l'*Event Service* prevede l'astrazione di un canale di eventi (*Event Channel*) che funge da consumer per i supplier e da supplier per i consumer. In tal modo un supplier notifica l'evento senza conoscere i consumers interessati.

La figura II.6 illustra i componenti dell' *Event Service* sopra descritti.

Fig. II.6 Componenti di CORBA Event Service

Il *Notification Service* è una estensione dell' *Event Service* ed affronta le seguenti problematiche:

- strategie di filtraggio eventi (filtering);
- mantenimento di informazioni sullo stato dell' *Event Channel*;
- definizione di parametri di qualità del servizio di consegna degli eventi;

La figura II.7 [45] illustra i componenti del *Notification Service* e mette in evidenza gli elementi che fanno del *Notification Service* una estensione dell'*Event Service*.

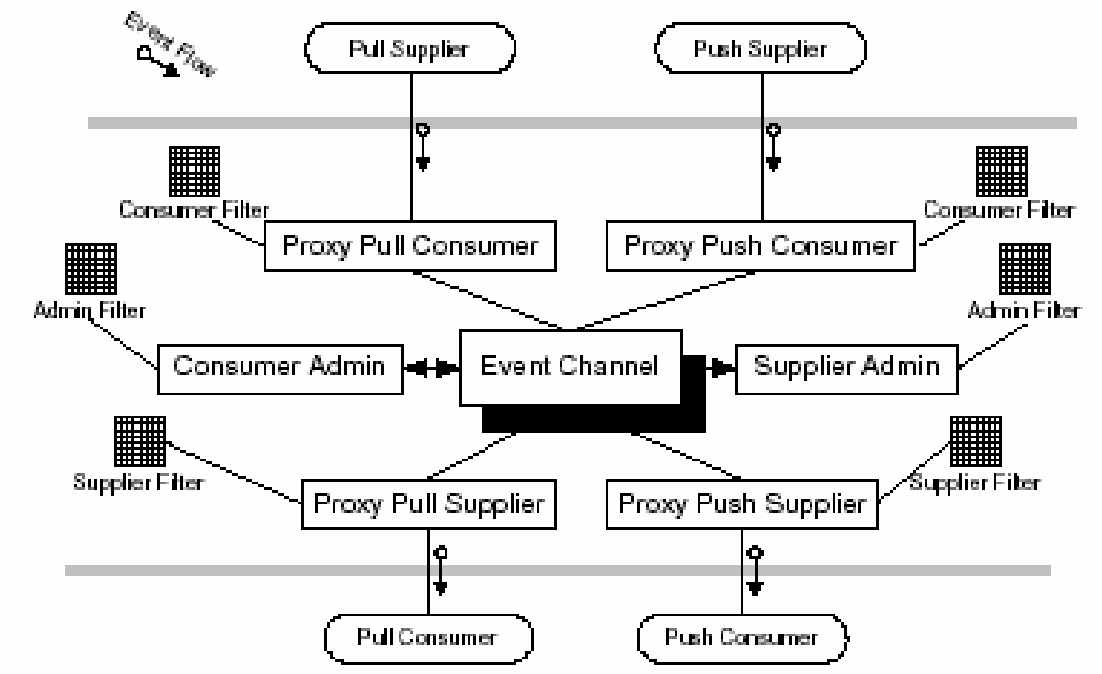


Fig. II.7 Componenti del CORBA Notification Service

CORBA ha riscontrato molto successo nelle industrie e nel campo della ricerca: infatti, implementazioni dello standard sono disponibili presso un gran numero di rivenditori ed esistono persino versioni freeware.

CORBA supporta tutti i maggiori linguaggi di programmazione ed è adattabile per quasi tutte le combinazioni di hardware e sistemi operativi. Tuttavia non è adatto a piccoli dispositivi ed a sistemi altamente dinamici, per i quali sono adottati altri sistemi.

II.3.2 JINI

Jini è un'infrastruttura basata sul top di Java ed RMI per creare una federazione fra dispositivi e componenti software che implementano servizi. Jini rende possibile per qualsiasi dispositivo in grado di eseguire una Java Virtual Machine di interoperare con gli altri, offrire ed utilizzare servizi.

Un servizio è definito come un'entità che può essere usata da una persona, da un programma o da un altro servizio. Tipici esempi di servizi sono la stampa di un documento o la traduzione di dati da un formato ad un altro, ma è possibile considerare servizi anche dispositivi con particolari funzionalità hardware.

I client possono usare un particolare servizio senza avere una conoscenza a priori della sua implementazione, scaricando quando necessario un oggetto proxy.

L'abilità di Jini di creare spontaneamente una federazione di servizi che sia robusta e tollerante ai guasti, è basata su una serie di concetti fondamentali:

- *Discovery*: processo attraverso il quale un client o un servizio localizza il sistema di nomi all'interno della rete;
- *Join*: meccanismo che permette di registrare un nuovo servizio sul sistema di nomi locale o remoto;
- *Lookup*: processo attraverso il quale il client interroga il sistema di nomi per ricercare il servizio richiesto.

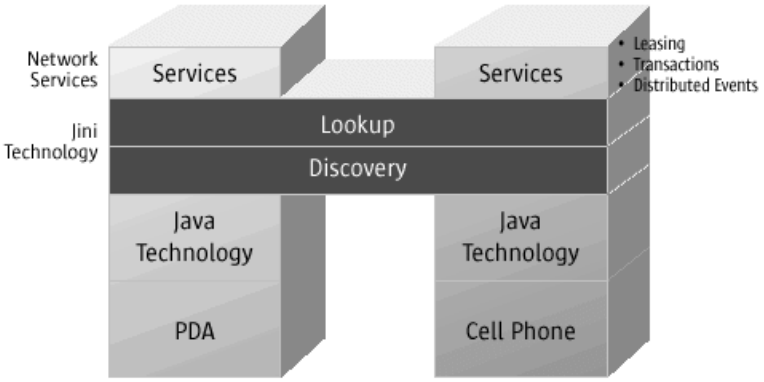


Fig. II.8 Architettura di Jini.

II.3.2.1 DISCOVERY

I servizi usano un meccanismo standard per registrarsi ed entrare a far parte della federazione: innanzitutto, interrogano la rete locale per localizzare il Lookup Service.

Il Lookup Service è un sistema che tiene traccia dei servizi che hanno notificato la loro presenza sulla rete.

Poiché questo componente è di vitale importanza, è previsto che, per aumentare la robustezza e l'affidabilità del sistema, vi possano essere più Lookup Service in esecuzione contemporaneamente su macchine diverse.

La ricerca avviene, a seconda che si cerchi in una rete locale (LAN) o geografica (WAN), attraverso richieste *multicast* o *unicast*. Il multicast discovery è utilizzato per trovare un Lookup Service che contiene servizi che appartengono alla locale comunità Jini, mentre l'unicast discovery permette di localizzare Lookup service che contengono servizi remoti.

II.3.2.2 JOIN

Quando un servizio ha terminato la fase di Discovery e ha trovato un Lookup Service, ha l'obbligo di registrasi presso il Lookup Service stesso. Il servizio invia un oggetto proxy *(service object)* e gli attributi ad esso associati. Il service object contiene la descrizione dell'interfaccia per l'utilizzo del servizio, nonché l'implementazione dei metodi che verranno utilizzati dalle applicazioni.

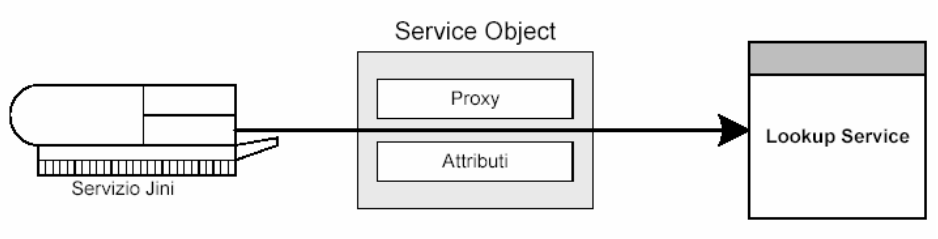


Fig. II.9 Registrazione del servizio.

I servizi sono tenuti a rinnovare la propria registrazione dopo un tempo stabilito al momento della registrazione stessa (*leasing*). Allo scadere del periodo di leasing, se il servizio non rinnova la registrazione, viene cancellato dalle entry del servizio di Lookup.

II.3.2.3 LOOKUP

Questa fase avviene quando un client ha già effettuato la fase di Discovery con successo e vuole richiedere un servizio Jini che abbia determinate caratteristiche. La richiesta al Lookup Service avviene specificando uno o più dei seguenti campi:

- *attributi*: viene effettuata una comparazione con gli attributi registrati all'atto del Join dei servizi;
- *interfaccia*: si specifica un'interfaccia che il servizio poi implementa. Si può, ad esempio, specificare un'interfaccia stampante per poi scegliere tra i *service provider* che la implementano;
- *ID number*: ad ogni servizio Jini, all'atto della registrazione, è assegnato un identificativo che può essere utilizzato nella fase di Lookup per richiedere un servizio ben preciso.

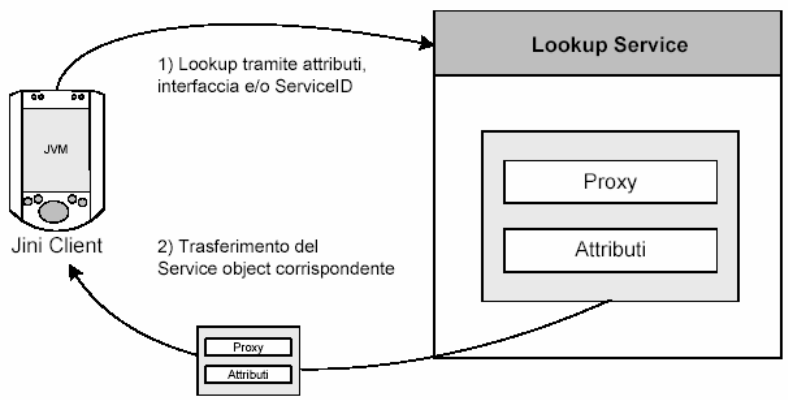


Fig. II.10 Lookup di un servizio.

Se gli attributi specificati dal client corrispondono ad un servizio registrato nel Lookup Service, quest'ultimo restituisce il service object corrispondente che verrà usato per invocare il servizio [23].

La tecnologia Jini, anche se particolarmente adatta per lo sviluppo di sistemi dinamici, presuppone che ogni componente della federazione sia in grado di eseguire una Java Virtual Machine. L'ipotesi non è banale e non è sempre possibile, non solo per mancanza di risorse, ma anche per questioni di conflittualità di interessi: vedi Microsoft contro SUN.

Per questi motivi, sono stati ricercati nuovi modelli basati su tecnologie aperte e popolari, come ad esempio i *Web Services*.

II.3.3 WEB SERVICES

Il modello dei *Web Services* [41] è un insieme di standard per la pubblicazione, il discovery e la composizione di servizi indipendenti in una rete aperta [6].

Il concetto di web service è molto simile a quello di oggetto, nel significato con cui lo si usa nella programmazione Object-Oriented: un oggetto è un modulo software che offre una serie di funzioni utilizzabili dall'esterno da parte di altro software, tramite una interfaccia di comunicazione dichiarata dall'oggetto stesso.

Così come per CORBA ed RMI, anche un Web Service offre una funzionalità (servizio), ad altri client sulla rete attraverso una interfaccia ben definita. La differenza è che in questo caso il servizio è posto sul web, e l'integrazione con il servizio avviene attraverso lo scambio di messaggi sulla rete.

La forza dei Web Services è di utilizzare un set base di protocolli disponibili ovunque, permettendo l'interoperabilità tra piattaforme molto diverse e mantenendo comunque la possibilità di utilizzare protocolli più avanzati e specializzati per effettuare compiti specifici. I protocolli alla base del modello dei Web Services sono quattro:

- XML è lo standard usato per rappresentare i dati trasportati;
- SOAP è lo standard usato per definire il formato dei messaggi scambiati;
- WSDL è lo standard usato per descrivere il formato dei messaggi da inviare al Web Service, quali sono i metodi esposti, quali sono i parametri ed i valori di ritorno.

- UDDI è lo standard promosso dall'omonimo consorzio, che ha come scopo quello di favorire lo sviluppo, la scoperta e l'interoperabilità dei servizi Web.

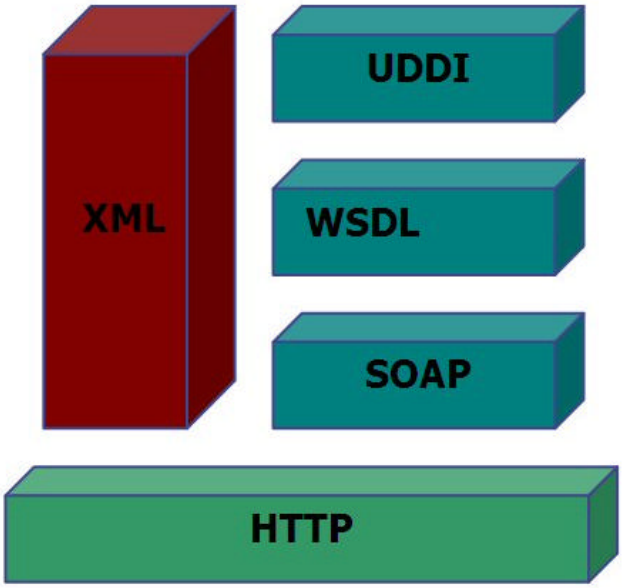


Fig. II.11 Protocolli standard dei Web Services.

II.3.3.1 WSDL

I web services sono visti come software disponibile sul web, utilizzabile da altri web services o da utenti: di conseguenza essi devono essere descritti cosicché gli altri componenti possano usarli facilmente, disaccoppiando interfaccia ed implementazione.

Il Web Service Description Language [42] è il linguaggio standard per la descrizione delle interfacce dei web services (l'equivalente di IDL per CORBA).

Esso presenta le caratteristiche architetturali di molti altri linguaggi basati su XML nati di recente: infatti fa riferimento e si integra con standard esistenti, evitando di ridefinire ciò che è già stato definito ed inoltre predilige l'uso di XML Schema per il type system e di SOAP per la definizione dei messaggi.

WSDL si presenta, quindi, come un formato XML per descrivere servizi di rete come un insieme di punti terminali, detti *porte*, operanti su messaggi contenenti informazioni di tipo "documentale" o "procedurale".

Esso opera una chiara distinzione tra i messaggi e le porte: i messaggi, ossia la sintassi e la semantica proprie di un servizio Web, sono sempre astratti, mentre le porte, l'indirizzo di

rete grazie al quale è possibile richiamare il servizio Web desiderato, sono sempre concrete.

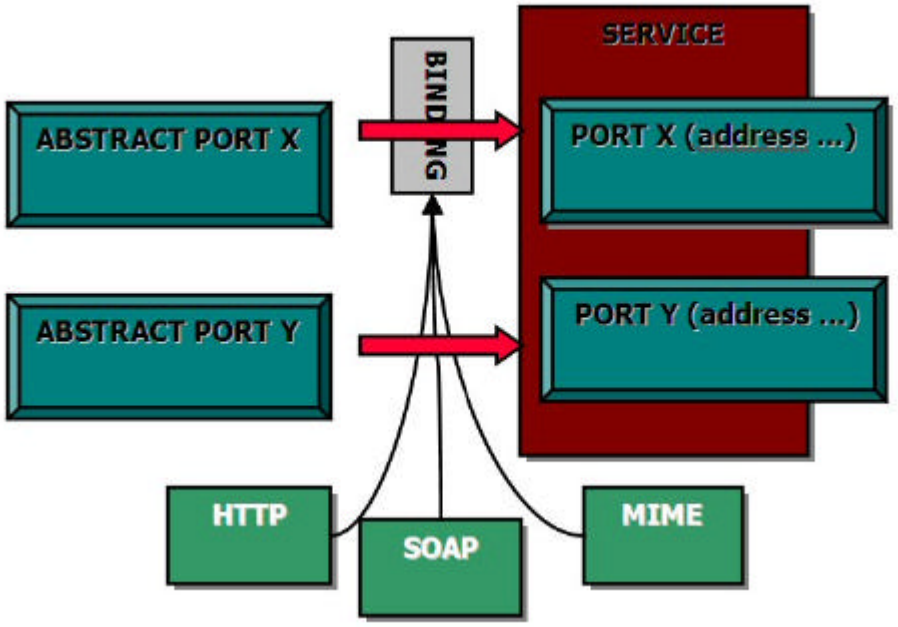


Fig. II.12 Dalla descrizione astratta a quella concreta.

All'utente non viene chiesto di fornire informazioni sulla porta in un file WSDL. Un file WSDL può contenere unicamente informazioni di interfaccia astratte e non può fornire dati di implementazione concreta. Sono questi i requisiti da rispettare affinché un file WSDL sia considerato valido.

WSDL separa, dunque, gli aspetti "astratti" della descrizione di un servizio da quelli "concreti" che lo legano a particolari protocolli di rete o di RPC, consentendo ad uno stesso servizio di poter avere implementazioni diverse, basate sulla stessa descrizione, garantendo in questo modo che i sistemi possano comunicare tra di loro e favorendo il riutilizzo delle descrizioni astratte per la creazione di nuovi servizi.

II.3.3.2 UDDI

L'Universal Description, Discovery and Integration (UDDI) [33] Service è un'iniziativa supportata da IBM, Microsoft, e HP, e fornisce ai client un meccanismo per trovare dinamicamente web services.

UDDI è un registro pubblico progettato per contenere informazioni strutturate sulle aziende e i rispettivi servizi. Con UDDI, è possibile pubblicare e individuare informazioni su un'azienda e i servizi Web da essa proposti.

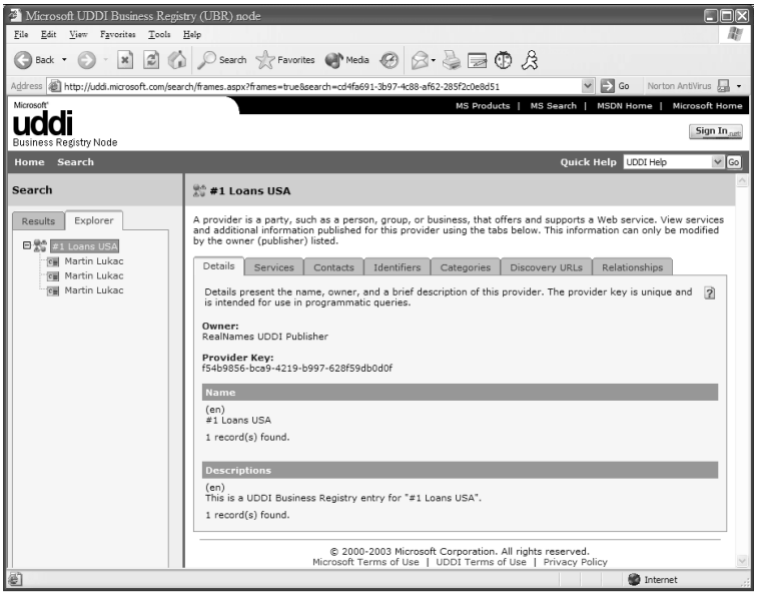


Fig. II.13 Esempio di registro UDDI della Microsoft.

Questi dati possono essere classificati mediante tassonomie standard, in modo da facilitare la ricerca delle informazioni per categoria.

Particolare di primaria importanza, in UDDI sono disponibili informazioni sulle interfacce tecniche dei servizi delle aziende. Mediante un gruppo di chiamate di API XML basate su SOAP, è possibile interagire con UDDI, sia in fase di progettazione che in fase di esecuzione, per rilevare dati tecnici che consentono l'attivazione e l'utilizzo di tali servizi.

Utilizzato in questo modo, UDDI funge da infrastruttura per una configurazione software basata su servizi Web.

Lo scenario nel quale trova pieno utilizzo l'UDDI può essere ricondotto a tre fasi fondamentali:

- pubblicazione (*publishing*): il fornitore del servizio per renderlo pubblico contatta il *service registry* che provvede ad inserirlo nel registry tramite UDDI;
- ricerca (*finding*): alla richiesta di un servizio, il service registry provvede a cercare quelli che meglio rispondono alle esigenze del richiedente;

- collegamento (*binding*): nel momento in cui il service registry fornisce la risposta può stabilirsi il collegamento tra il richiedente del servizio web e il fornitore.

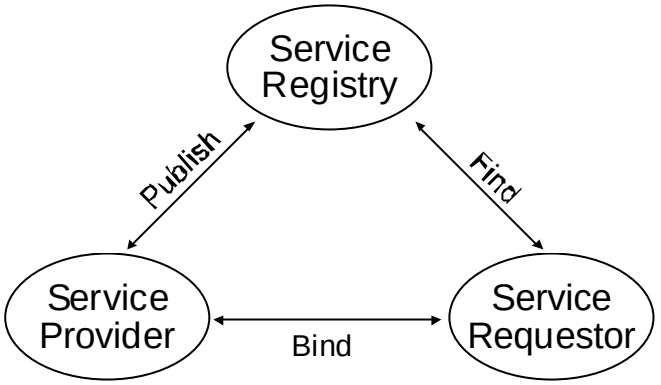


Fig. II.14 Scenario di utilizzo di UDDI.

La pubblicazione sul registro UDDI è un processo relativamente semplice e consiste nel raccogliere alcune informazioni di base relative al modo in cui si desidera modellare in UDDI la voce che definisce l'azienda e i relativi servizi, dopo di che si passa alla registrazione vera e propria, adoperando un'interfaccia utente basata su Web o tramite un programma.

Le informazioni alla base della registrazione sono così classificate:

- *white pages*, che forniscono il nome dell'azienda, una breve descrizione, se necessario, in più lingue, le persone da contattare per ottenere informazioni sui servizi Web offerti ed altri identificativi;
- *yellow pages*, che classificano azienda e servizi attraverso tassonomie standard. Le tassonomie attualmente supportate sono NAICS (*North American Industry Classification System*), UNSPSC (*Universal Standard Products and Services Codes*), ISO 3166, SIC (*Standard Industry Classification*) e GeoWeb Geographic Classification;
- *green pages*, che contengono informazioni tecniche circa i web service che sono esposti attraverso il concetto di tModel. La struttura tModel, abbreviazione di "Technology Model", rappresenta le tracce tecniche, le interfacce e i tipi astratti dei metadati. I file WSDL sono esempi perfetti dei tModel UDDI.

Per ricercare il servizio sono utilizzate una serie di API che vengono esposte per fare browsing su un repository site e per prelevare le informazioni che occorrono riguardo al servizio desiderato. Esistono anche API, destinate a chi pubblica nuovi web services, per l'inserimento di nuove informazioni in questo repository.

Infine, una volta trovato il web service di cui si aveva bisogno, viene effettuato il binding tra interfaccia e implementazione. Anche UDDI, infatti, opera una distinzione simile a quella fatta da WSDL tra astrattezza e implementazione e lo fa attraverso il concetto di tModel. A supporto dei tModel, vi sono i modelli di binding, ossia le implementazioni concrete di uno o più tModel. E' nel modello di binding che si registra il punto di accesso per un'implementazione specifica di un tModel.

Il registro UDDI e il linguaggio WSDL fungono da specifiche complementari per la definizione di configurazioni software basate sui servizi Web.

Il linguaggio WSDL fornisce un metodo formale, non legato a produttori specifici, per definire i servizi Web, rendendo possibile l'utilizzo delle chiamate di procedure remote dell'ultima generazione, mentre UDDI fornisce una vasta infrastruttura standardizzata che consente agli utenti di descrivere e individuare i servizi Web. L'utilizzo combinato dei due standard consente la nascita di un vero ecosistema di servizi Web.

II.4 LA COMUNICAZIONE IN AMBIENTI PERVASIVI

Gli ambienti pervasivi o ubiqui sono caratterizzati da un gran numero di entità autonome dette *agenti,* che contribuiscono a trasformare uno spazio fisico in un ambiente intelligente e computazionalmente attivo. Queste entità possono essere dispositivi, servizi, utenti.

Nonostante siano stati sviluppati vari tipi di *middleware* ed infrastrutture per rendere possibile la comunicazione tra agenti, tra cui quelli presentati precedentemente, nessuno di questi prevede meccanismi per supportare l'interoperabilità semantica tra essi [26].

Nei sistemi distribuiti, lo scambio di messaggi rappresenta, come si è visto, un aspetto fondamentale. I messaggi, che contengono descrizioni di entità, servizi, eventi ed altri concetti, possono essere raggruppati essenzialmente in tre categorie:

- *Advertisement*: corrisponde all'invio di descrizioni di servizi offerti, di dispositivi, interfacce per registri.

- *Notification*: corrisponde all'invio di descrizioni di eventi, ad esempio l'arrivo di un'entità nell'ambiente.
- *Query*: corrisponde alla richiesta di servizi o entità che rispondono a certi requisiti e alla ricezione delle descrizioni di servizi ed entità che li soddisfano.

Per consentire lo scambio di tali messaggi tra due agenti occorre necessariamente che il loro formato sia comune ed inoltre che siano comuni interfacce e protocolli. E' difficile, poi, aspettarsi che le diverse entità comprendano la semantica dell'ambiente e delle altre entità quando interagiscono fra loro. Deve essere quindi conosciuta o scoperta anche la semantica dei messaggi, ossia il vocabolario dei messaggi, che include i nomi e i valori validi degli elementi.

Mentre per sistemi semplici o chiusi, tutti gli schemi richiesti per interpretare il contenuto dei messaggi sono compilati nei componenti del sistema stesso, in un sistema aperto le parti che comunicano sono autonome, eterogenee ed evolvono nel tempo.

Occorre, quindi, un modello aperto che renda possibile la scoperta e l'utilizzo degli schemi quando occorrono e mentre il sistema è in esecuzione.

II.4.1 L'INTEROPERABILITÀ: ESIGENZA DI NUOVI STRUMENTI

I registri di oggetti, come il CORBA Naming Service, forniscono un meccanismo base per la ricerca di servizi della cui esistenza si è già a conoscenza. Brokers, come il CORBA Trading Service, forniscono la capacità di localizzare servizi in base ad alcuni attributi degli stessi.

Altri servizi che forniscono caratteristiche simili sono, ad esempio, JINI, LDAP, Microsoft's Registry. Questi standard definiscono interfacce e formati per le descrizioni ma non ne definiscono il contenuto, la semantica. Ad esempio, il CORBA Trading Service è un'interfaccia standard per un broker, ma non definisce né le proprietà dei servizi registrati né i valori legali di tali proprietà. Le regole di matching sono limitate e applicabili solo agli attributi definiti esplicitamente. Inoltre, mentre per le query esiste una sintassi, non esiste un linguaggio dichiarativo per definire le tipologie di servizi e le loro istanze.

Allo stesso modo, Jini definisce architettura, protocolli e interfacce per lo scambio di tutte le tipologie di messaggi viste precedentemente. I servizi e le entità sono descritte dai

Service Entry, ma come nel caso di CORBA, la loro definizione è lasciata agli applicativi. Non sono definiti, inoltre, né un linguaggio standard per la definizione di schemi né meccanismi per la gestione e la validazione delle Service Entry.

I Web Services, sebbene siano nati soprattutto per superare i problemi di interoperabilità tra piattaforme, stanno cercando di risolvere anche questi problemi di service discovery e di gestione di descrizioni provenienti da entità autonome. Tuttavia anche per i Web Services non è ancora stato definito il livello semantico, ossia non sono stati definiti standard per la definizione, validazione e scambio di schemi per le descrizioni di entità e servizi e dei loro modelli tecnici. Il Web Semantico è destinato a definire questi aspetti.

II.5 WEB SEMANTICO

Il termine Web Semantico [7] fu coniato per la prima volta da Tim Berners-Lee, l'ideatore del WWW.

Si tratta di un insieme di standard aperti, indipendenti dalle tecnologie, nati per consentire lo scambio di descrizioni di entità e relazioni allo scopo di migliorare le capacità di ricerca nel Web. Comunque questi standard sono ben adattabili ad alcuni dei requisiti di un sistema pervasivo.

Il Web Semantico rappresenta un'estensione del Web attualmente esistente che, in aggiunta, mira alla cooperazione tra uomo e calcolatori.

Gran parte del contenuto di Internet è progettato per essere letto da esseri umani e non per essere trattato da programmi ed è ben lontano dal poter fornire una solida piattaforma che renda possibile un'interpretazione e una comprensione semantica da parte di agenti automatici.

Questo è ciò che viene fornito dal Web Semantico e che può essere riassunto come segue:

- l'informazione deve essere *machine-readable*, ossia non più pensata per essere letta direttamente dall'uomo, ma mirata ad essere in un formato facilmente elaborabile dalla macchina, da agenti intelligenti, servizi specializzati, siti web personalizzati e motori di ricerca potenziati semanticamente;

- deve essere fornito un supporto per *l'interoperabilità sintattica*, intesa come la facilità di leggere dati e ottenere una rappresentazione utilizzabile dalle applicazioni;
- deve essere fornito un supporto per *l'interoperabilità a livello semantico*: non sono più sufficienti standard per la forma sintattica dei documenti, ma anche per il loro contenuto semantico; interoperabilità semantica significa definire *mapping* tra termini sconosciuti e termini conosciuti nei dati;
- il formato utilizzato per lo scambio dei dati deve permettere di poter esprimere qualsiasi forma di dati, poiché non è possibile anticiparne tutti i suoi usi potenziali (*potere espressivo universale*). Per raggiungere questo obiettivo, è necessario basarsi su un modello comune di grande generalità. Solo così qualsiasi "prospettiva" può trovare espressione all'interno del modello.

Il Web Semantico, quindi, mira a riportare chiarezza, formalità e organizzazione dei dati, collegando le informazioni a concetti astratti organizzati in una gerarchia, a sua volta descritta in un meta-documento. In questo modo vari agenti automatici hanno la possibilità di cogliere il contesto semantico di una fonte informativa interpretando le varie relazioni esistenti tra le risorse, formulando asserzioni sulle stesse, nonché controllando la loro attendibilità.

II.5.1 ARCHITETTURA E LINGUAGGI DEL WEB SEMANTICO

Nella visione di Tim Berners-Lee, il Web Semantico è un'architettura strutturata su almeno quattro livelli:

- il livello dei dati (un semplice modello dei dati e una sintassi per i metadati);
- il livello schema (una base per la definizione di una vocabolario);
- il livello ontologico (per la definizione delle ontologie);
- il livello logico (supporto al ragionamento).

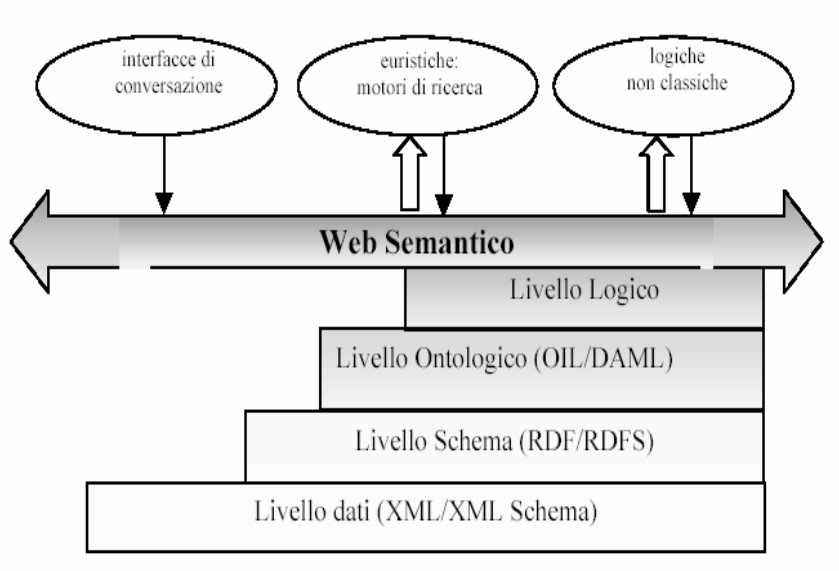


Fig. II.15 Architettura del Web Semantico.

Il Web Semantico, a differenza del Web che si fonda su documenti, si fonda su risorse. Esse sono descritte adoperando i linguaggi RDF e RDF Schema, entrambi basati su tecnologie XML. Questo tuttavia non è sufficiente perché occorre catturare la semantica delle risorse descritte e delle relazioni tra esse e per fare ciò occorre introdurre vocabolari ontologici.

Una volta che le risorse sono state ben definite, non solo sintatticamente ma anche semanticamente, è possibile estendere questi vocabolari ontologici con logiche che supportano ragionamenti automatici.

II.5.2 XML E XML SCHEMA

XML (eXtension Markup Language) [37] è un linguaggio di markup destinato alla descrizione di documenti, strutturati in maniera arbitraria. A differenza di HTML, che è utilizzato per descrivere la modalità con cui documenti ipertestuali con strutture fisse sono visualizzati, XML separa il contenuto del documento dalla sua modalità di visualizzazione. E' dunque uno standard elaborato per garantire l'interoperabilità sintattica, ossia per fare in modo che le informazioni non siano semplicemente formattate per facilitarne il reperimento per un utente umano, ma anche facilmente elaborabili da agenti software.

```
<?xml version="1.0" ?>
<person>
        <name>Marco</name>
        <personal_info>
                <office_number>18</office_number>
                <phone_number>1024</phone_number>
        </personal_info>
</person>
```

Fig. II.16 Esempio di file XML.

XML definisce una struttura ad albero per i documenti dove ciascun nodo individua un tag ben definito (*elemento*) mediante il quale è possibile in qualche modo interpretare le informazioni che esso racchiude (*attributi*).

E' possibile imporre delle restrizioni con cui i tag XML possono essere usati e quali annidamenti di tali tag sono permessi mediante l'uso di due tecnologie:

- DTD (*Document Tye Definition*): Contengono le regole che definiscono i tag usati nel documento XML, in altre parole ne definiscono la struttura. Questi possono essere dei file esterni o specificati direttamente all'interno del documento.
- XML Schema: Spesso i documenti condividono una struttura specifica per un certo dominio, per consentire sia al software sia agli umani di riconoscere il contenuto dell'XML si richiede che tali strutture siano documentate in un formato comprensibile ad entrambi. A tal scopo è stato introdotto XML Schema, il quale permette di definire un vocabolario utilizzabile per descrivere documenti XML facendo uso della loro stessa sintassi. Le sue specifiche assumono che si faccia uso di almeno due documenti, un'istanza ed uno schema. Il primo contiene le informazioni che interessano realmente, mentre il secondo descrive struttura e tipo del precedente.

XML, tuttavia, non gestisce la semantica dei contenuti: essa non è specificata in modo esplicito, ma è "incorporata" nei nomi dei tag, ossia il vocabolario degli elementi e le loro combinazioni non sono prefissati, ma possono essere definiti ad hoc per ogni applicazione. Tale semantica, quindi, non è definita formalmente e può risultare eventualmente

comprensibile solo all'uomo e non alla macchina: un'entità software riconosce i contenuti, ma non è in grado di attribuire loro un significato.

XML, quindi, fornisce l'interoperabilità sintattica ma non riesce a garantire quella semantica né a fornire meccanismi di classificazione o ragionamento; pertanto, deve necessariamente essere affiancato ad altri linguaggi più potenti.

II.5.3 RDF E RDF SCHEMA

RDF (Resource Description Framework) [39] è lo standard che consente l'aggiunta di semantica a un documento e quindi si pone ad un livello direttamente superiore rispetto ad XML. RDF è, in un certo senso, un'applicazione di XML: se XML è un'estensione del documento, RDF può essere visto come un'estensione dei dati introdotti da XML.

Il modello base dei dati RDF è composto da:

- *Risorse*: con questo termine si intende qualsiasi cosa possa essere descritta. Una risorsa può essere ad esempio una pagina Web oppure un qualsiasi oggetto anche se non direttamente accessibile via Web (ad esempio un libro, una persona). Una risorsa viene identificata univocamente attraverso un URI.
- *Proprietà*: una proprietà è una caratteristica, una relazione che descrive una risorsa. Il significato, l'insieme di valori che può assumere, i tipi di risorse a cui può riferirsi sono tutte informazioni reperibili dallo schema RDF in cui essa è definita.
- *Asserzioni*: un'asserzione è costituita da un soggetto (la risorsa descritta), un predicato (la proprietà) e un oggetto (il valore attribuito alla proprietà), dove l'oggetto può essere una semplice stringa o un'altra asserzione.

Volendo fare un paragone con un database relazionale potremmo dire che una riga di una tabella è una risorsa RDF, il nome di un campo (una colonna) è il nome di una proprietà RDF, il valore del campo è il valore della proprietà.

Un semplice esempio di utilizzo del modello di RDF può essere fornito dalla seguente asserzione:

Jim's phone_number is 1024

che è stato graficamente schematizzato in figura II.17.

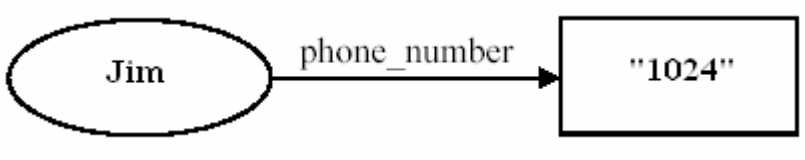


Fig. II.17 Semplice esempio di un'asserzione RDF.

La precedente asserzione è formalizzata in RDF/XML come segue:

```
<?xml version="1.0" ?>

<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:s ="http://www.example.com/props/"
xmlns:base="http://www.organization.com/people">

        <rdf:Description rdf:about="Jim">
                <s:Phone_number>1024</s:Phone_number>
        </rdf:Description>

</rdf:RDF>
```

Fig. II.18 Esempio di documento RDF/XML.

Sintatticamente, i concetti espressi con RDF vengono serializzati mediante XML. RDF definisce, quindi, un semplice modello dei dati per descrivere proprietà e relazioni fra risorse. In RDF però non esistono livelli di astrazione: ci sono le risorse e le loro relazioni, tutte organizzate in un grafo piatto. In altri termini non è possibile definire tipi (o classi) di risorse con loro proprietà specifiche.

Vista l'utilità di poter definire classi di risorse, RDF è stato arricchito con un semplice sistema di tipi detto RDF Schema.

Il sistema di tipi RDF Schema ricorda i sistemi di tipi dei linguaggi di programmazione object-oriented (Java-like). Una risorsa può, per esempio, essere definita come istanza di una classe (o di più classi) e le classi possono essere organizzate in modo gerarchico. RDF Schema utilizza il modello RDF stesso per definire il sistema di tipi RDF, fornendo un insieme di risorse e proprietà predefinite che possono essere utilizzate per definire classi e proprietà a livello utente. E' possibile, inoltre, definire vincoli di dominio e di range sulle

proprietà ed alcuni tipi di relazioni (comprese quelli di sottoclasse di una risorsa e sottotipo di una proprietà). L'insieme di tali elementi è detto vocabolario dell'RDF Schema.

Il linguaggio di specifica RDFS è un linguaggio dichiarativo poco espressivo ma molto semplice da implementare. Si esprime attraverso la sintassi di serializzazione RDF/XML e si avvale del meccanismo dei *namespace* XML per la formulazione delle URI che identificano in modo univoco le risorse (definite nello schema stesso) consentendo il riutilizzo di termini definiti in altri schemi.

Concetti e proprietà già dichiarati per un dominio possono essere impiegati di nuovo o precisati per incontrare le esigenze di una particolare comunità di utenti.

Sebbene, in prima battuta, RDF e RDF Schema sembrano essere uno strumento buono per la definizione di un linguaggio di markup per il Web Semantico (ad esempio, per determinare le relazioni semantiche tra termini differenti), in realtà essi mostrano di non avere sufficiente potere espressivo: non consentono, infatti, di specificare le proprietà delle proprietà, le condizioni necessarie e sufficienti per l'appartenenza alle classi e gli unici vincoli che si possono definire sono quelli di dominio e range delle proprietà.

Inoltre, il loro utilizzo nei sistemi di rappresentazione della conoscenza è limitato da un'altra caratteristica: non permettono di specificare meccanismi di ragionamento, ma rappresentano semplicemente un sistema a frame.

I meccanismi di ragionamento devono essere costruiti, quindi, ad un livello superiore.

II.5.4 DAML+OIL

Il linguaggio DAML+OIL [12] è uno standard che consente la rappresentazione delle informazioni in modo che il loro significato sia comprensibile alle macchine.

I due risultati più interessanti degli ultimi anni in fatto di sviluppo verso il Web Semantico, DAML-ONT e OIL, si sono fusi in un unico linguaggio, DAML+OIL appunto, che incorpora caratteristiche provenienti dal lavoro del gruppo americano (DARPA) e del gruppo europeo (progetto On-To-Knowledge, IST).

DAML+OIL riesce a garantire l'interoperabilità sintattica e semantica sfruttando la sintassi e le caratteristiche dei linguaggi che si trovano ai livelli più bassi dello stack del WEB Semantico, ossia XML e RDF, ed in più aggiunge, rispetto a questi, una maggiore

forza espressiva ottenuta a partire da primitive di modellazione ereditate dai sistemi basati su Frame e la capacità di produrre asserzioni in logica formale ed effettuare ragionamenti in modo automatico, traendo spunto dalle Description Logics.

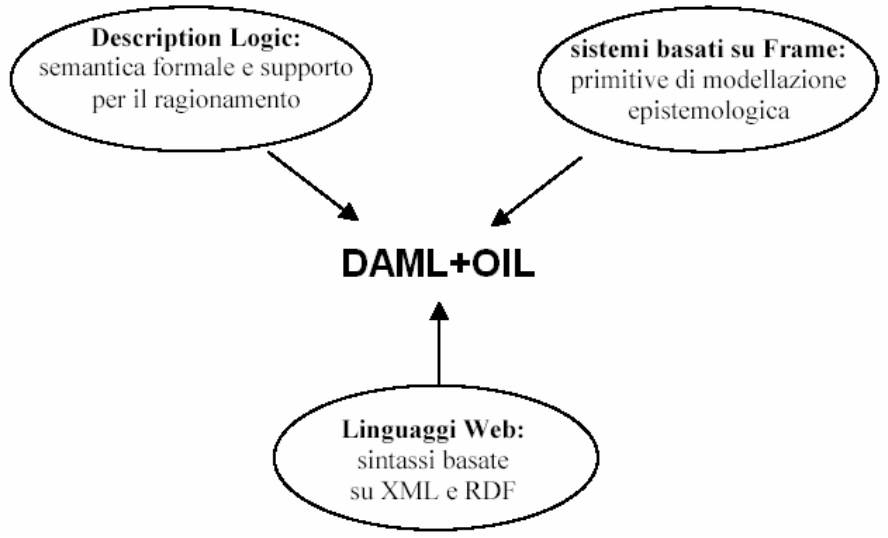


Fig. II.19 Il modello di DAML+OIL.

II.5.4.1 SISTEMI BASATI SU FRAME

I sistemi basati su *Frame* somigliano agli approcci orientati agli oggetti, ma considerano le cose da un punto di vista differente.

Le primitive di modellazione sono le classi (o frame), ognuna delle quali ha determinate proprietà (o slot), chiamate attributi. Essi non hanno una visibilità globale, ma sono applicabili alle sole classi per cui sono definiti.

Un frame fornisce un certo contesto per la modellazione di un aspetto del dominio in esame. Dai sistemi basasti su Frame, DAML+OIL eredita le primitive di modellazione essenziali.

Esso, infatti, è basato sulla nozione di classe, la definizione delle sue proprietà e delle sue superclassi e sottoclassi. Le relazioni possono essere definite come entità indipendenti aventi un certo dominio e intervallo. Come le classi, anche le relazioni possono essere organizzate in una gerarchia.

Ad incrementare il potere espressivo contribuiscono, poi, due aspetti: i costruttori e i tipi di assiomi.

I costruttori permettono la creazione, oltre che di classi intese nel senso classico, con un nome ed associate ad un URI, anche di classi intese come espressioni, cioè come il prodotto di un certo numero di operatori.

Una classe può essere ottenuta dall'unione o dall'intersezione di altre classi, elencandone gli elementi, può essere complementare di un'altra, può essere definita come collezione degli elementi che hanno almeno o al più un certo numero di valori distinti di una proprietà.

Il secondo aspetto è rappresentato dagli assiomi, che permettono la dichiarazione di relazioni di classificazione o equivalenza tra classi e/o proprietà, la disgiunzione tra classi, l'equivalenza o meno di oggetti diversi, le proprietà delle proprietà.

II.5.4.2 DESCRIPTION LOGICS

Le *Description Logics* rappresentano una classe di logiche che sono specificatamente designate a modellare vocabolari. Esse descrivono la conoscenza in termini di concetti (paragonabili ai frame) e restrizioni (paragonabili agli slot) che sono utilizzate per derivare automaticamente classificazioni tassonomiche.

DAML+OIL eredita dalle Description Logics la semantica formale e il supporto per il ragionamento, stabilendo una corrispondenza tra concetti logici e tag DAML+OIL che li rappresentano.

A partire da un documento DAML+OIL è possibile, infatti, creare una *Knowledge Base*: una Kb è un database con capacità di ragionare in maniera automatica, in grado non solo di rispondere a query grazie a dei match ma anche effettuando ragionamenti.

Una KB è formata da due componenti:

- *Intensionale*: uno schema che definisce classi, proprietà e relazioni tra classi (Tbox, ossia *Terminological knowledge*);
- *Estensionale*: un'istanza (parziale) dello schema, contenente asserzioni su individui (Abox, ossia *Assertional knowledge*).

Il Tbox è il modello di ciò che può essere vero, l'Abox è il modello di ciò che correntemente è vero. Effettuando ragionamenti sulla KB, è possibile ottenere risposte a questioni importanti come:

- la *satisfiability* di un concetto: se un concetto esiste;
- la *subsumption*: se un concetto è un caso di un altro concetto;
- la *consistence*: se l'intera KB verifica la satisfiability;
- l'*instance checking*: se un'asserzione verifica la satisfiability.

Uno dei motivi principali per cui si usano le DL come supporto per il ragionamento in DAML+OIL è che la satisfiability e la subsumption possono essere ricavate in modo computazionalmente efficiente, attraverso algoritmi non complessi: essi convertono le asserzioni in una forma normale e poi costruiscono un insieme di vincoli che sono analizzati per verificare se esistono contraddizioni o meno.

II.5.5 OWL

OWL (Ontology Web Language) è un'estensione di DAML+OIL ottenuta attingendo da esso gli insegnamenti tratti dal suo uso applicativo e, in virtù di ciò, molto simile ad esso. Come DAML+OIL, anche OWL estende RDF ed RDF Schema e usa la sintassi XML. Fornisce tre sottolinguaggi:

- *OWL Full*: consente di combinare OWL con RDF e RDF Schema. E' destinato agli utenti che vogliono la massima espressività e la libertà sintattica di RDF senza alcuna garanzia computazionale. Il suo vantaggio è che è pienamente compatibile con RDF , sia sintatticamente che semanticamente, cioè qualsiasi documento RDF legale è anche un documento OWL Full legale.
- *OWL DL*: inserisce alcuni vincoli sul modo di combinare OWL con RDF Schema. E' destinato agli utenti che vogliono la massima espressività senza perdere l'efficienza e la completezza computazionali, e i benefici dei sistemi che adottano tecniche di reasoning. E' stato realizzato per supportare le Description Logics esistenti.
- *OWL Lite*: è un sottoinsieme di OWL DL che è facilmente utilizzabile ed implementabile. Aggiunge ad RDF Schema molte funzionalità utili per supportare le

applicazioni web. Comprende molte delle caratteristiche più usate di OWL ed è destinato agli sviluppatori che vogliono utilizzare OWL ma vogliono iniziare con un set relativamente semplice di caratteristiche del linguaggio.

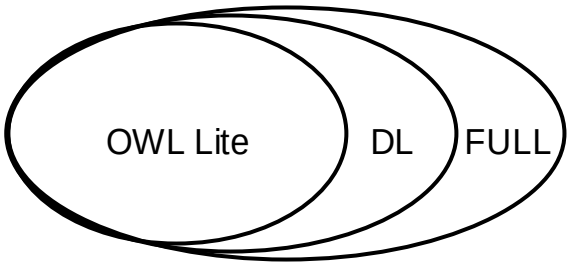


Fig. II.20 Sottolinguaggi di OWL

III. MODELLO ARCHITETTURALE

III.1 INTRODUZIONE

Dopo l'analisi effettuata nei primi capitoli, siamo pronti per presentare la nostra idea di modello architetturale per ambienti di *pervasive computing*.

Le principali scelte di progetto sono state prese coerentemente con lo stato dell'arte e si è tentato di non introdurre nuova complessità al panorama esistente, ricorrendo a tecnologie, protocolli e standard già esistenti ed affermati, senza definirne di nuovi. D'ora in avanti il sistema verrà anche chiamato col nome di *UbiSystem*.

III.2 REQUISITI DEL SISTEMA

L'architettura dovrà essere sufficientemente generale, in modo da poter garantire nuovi sviluppi ed evoluzioni future senza comportare l'esclusione a priori di determinate soluzioni e tecnologie.

Per quanto visto nei capitoli precedenti, il sistema dovrà garantire i seguenti requisiti funzionali:

- supporto per l'eterogeneità di dispositivi, piattaforme e linguaggi;
- definizione e condivisione dello stato del contesto del sistema;
- discovery semantico dei servizi e supporto per l'integrazione;
- gestione della dinamicità e della disponibilità delle risorse presenti.

Non vanno trascurati, inoltre, tutta una serie di requisiti non funzionali, alcuni dei quali strettamente legati agli aspetti di distributed computing, quali: scalabilità, tolleranza ai guasti, sicurezza ed attendibilità, semplicità, etc.

III.3 LO SCENARIO E GLI ATTORI

La figura seguente dipinge lo scenario all'interno del quale è stato sviluppato il prototipo. L'ambiente è costituito da una rete LAN con estensioni di tipo wireless: ad una serie di nodi fissi se ne affiancano altri di natura completamente mobile.

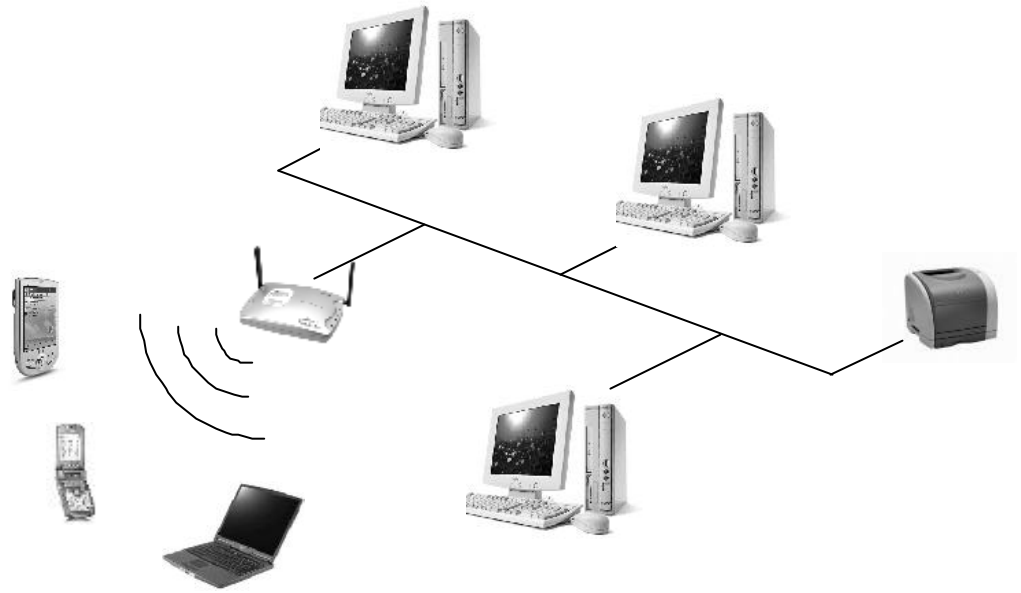


Fig. III.1 Scenario dell'ambiente pervasivo

Gli attori che interagiscono con *UbiSystem* sono principalmente due:

- Gli utenti (client) - che accedono al sistema mediante dispositivi portatili (palmari, cellulari di ultima generazione, laptop) e che sono interessati ad utilizzare i servizi offerti;
- I fornitori di servizi (service provider) - costituiti da moduli software che offrono servizi e compiono azioni per conto degli utenti.

Dal punto di vista dell'utente è desiderabile poter interagire con il sistema mediante il proprio dispositivo senza dover procedere all'installazione di software aggiuntivo né dover eseguire procedure di configurazione di alcun genere. Per questo motivo l'entry-point degli utenti sarà un'applicazione web e quindi accessibile mediante un comune web browser.

Per quanto riguarda i service provider, è stato stabilito che i servizi offerti all'interno dell'ambiente devono essere preferibilmente dei Web Services.

Questa scelta, adottata coerentemente con la nostra linea progettuale di utilizzare tecnologie e standard web-based, non rappresenta una forte limitazione: difatti, per riutilizzare moduli già esistenti (pensiamo, ad esempio, a servizi CORBA) è sempre possibile creare appositi wrappers che espongano i servizi come Web Services.

III.3.1 PERCHÉ I WEB SERVICES

La scelta di modellare i servizi all'interno del nostro ambiente come Web Services è stata motivata da una serie di considerazioni che proponiamo di seguito.

Un Web Service costituisce un sistema software disegnato per supportare l'interoperabilità e l'interazione fra macchine attraverso la rete. E' caratterizzato da un'interfaccia descritta mediante formati machine-processable. E' possibile interagire con un servizio Web mediante linguaggi e protocolli standardizzati o in via di standardizzazione quali WSDL, SOAP, HTTP, SMTP, FTP, etc. e quindi secondo tecnologie web-based (cfr. § II.3.3).

Come i Web Services, anche CORBA si propone di garantire interoperabilità multipiattaforma e multilinguaggio ma, trattandosi soltanto di specifiche, spesso si riscontrano problemi di incompatibilità fra ORB delle reali implementazioni. Inoltre, la forte limitazione di CORBA è quella di non poter usare Internet come infrastruttura di comunicazione poiché il suo traffico viene sistematicamente bloccato dai firewall presenti sulla rete.

Per quanto riguarda i problemi di Service Discovery e di integrazione di servizi, esistono in letteratura diverse soluzioni. Probabilmente la più famosa è la soluzione Jini proposta dalla SUN (cfr. § II.3.2). Ma come abbiamo visto precedentemente, la tecnologia Jini è strettamente legata alla piattaforma Java e si basa su RMI come middleware.

Tornando ai Web Services, c'è da sottolineare che, accanto alla grande opera di standardizzazione che sta compiendo il World Wide Web Consortium, ingenti sono gli investimenti in ricerca e sviluppo in tal senso da parte delle maggiori industrie e aziende dell'Information Technology, che scommettono su questa nuova ed emergente tecnologia.

Inoltre, insieme a DAML-OIL e OWL (cfr. § II.5.4 e § II.5.5), stanno nascendo ontologie come DAML-S ed OWL-S pensate appositamente per arricchire le descrizioni formali dei Web Services con contenuti di natura semantica.

III.4 IL MODELLO ARCHITETTURALE

L'ambiente *UbiSystem*, come mostra la figura III.2, offre le seguenti funzionalità:

- ottenere la connessione;
- richiedere la lista dei servizi applicativi presenti nel sistema;
- utilizzare un servizio applicativo presente nel sistema;
- registrare un servizio fornito da un utente;
- gestire i profili degli utenti registrati presso il sistema;
- monitorare l'ambiente.

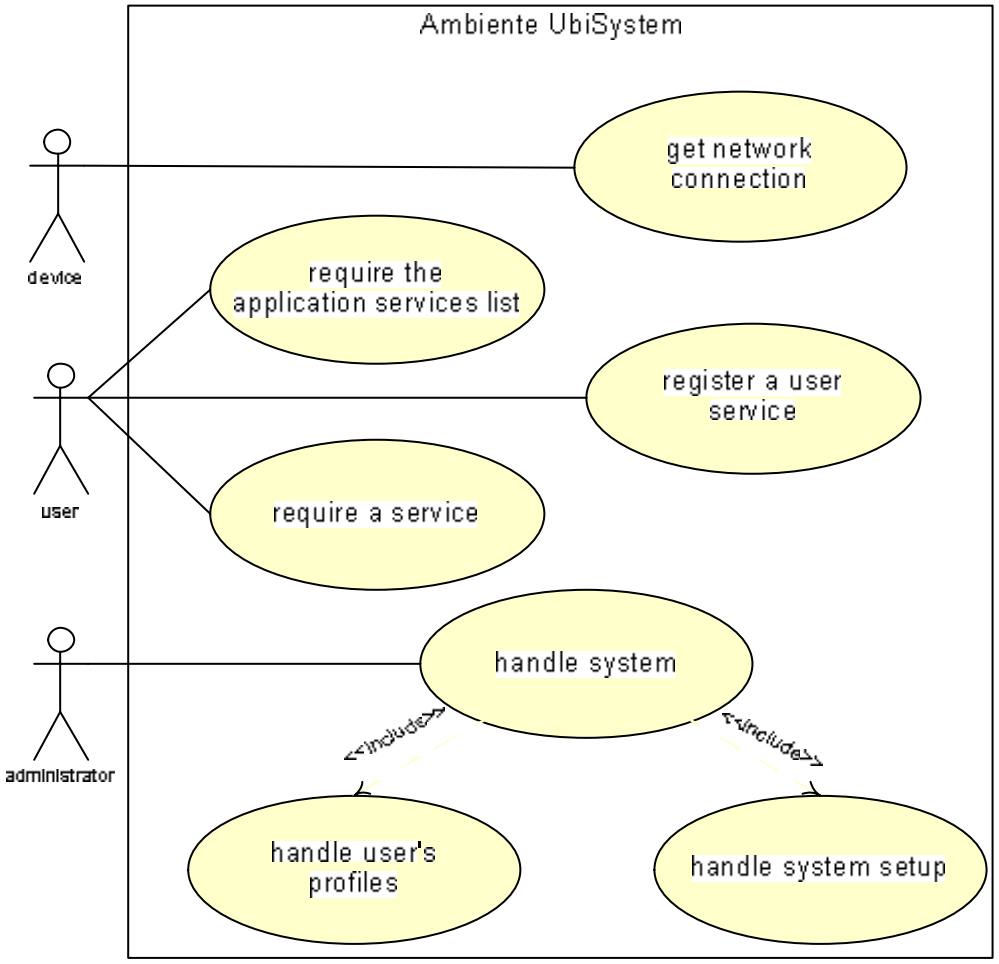


Fig. III.2 Funzionalità offerte dal sistema

Nel definire il framework è stata fatta una distinzione fra i servizi di supporto, finalizzati alla gestione generale del sistema, ed i servizi applicativi di utilità, ovvero i servizi pervasivi.

In figura III.3 si mostra l'architettura generale dell'intero sistema.

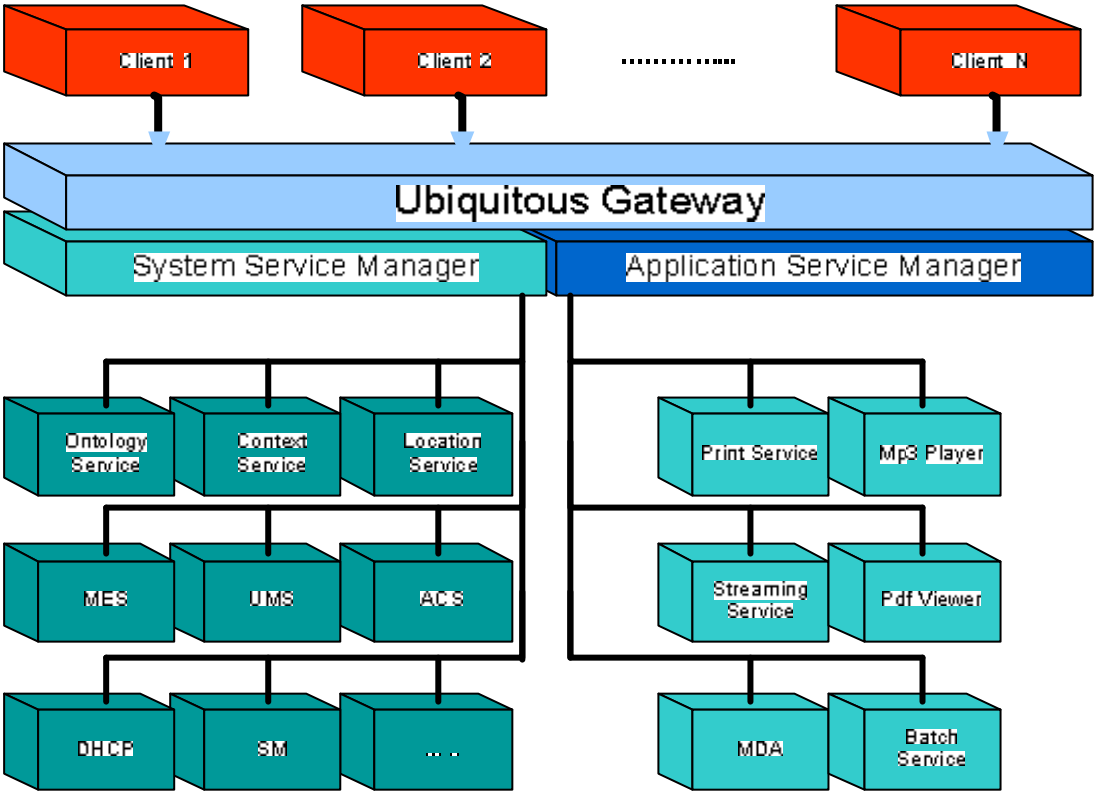


Fig. III.3 Architettura logica del sistema

L'architettura si sviluppa su due piani. Al primo abbiamo:

- *Ubiquitous Gateway*: entry point per gli utenti del sistema pervasivo;
- *System Services Manager*: gestore dei servizi di supporto del sistema;
- *Application Services Manager*: gestore dei servizi pervasivi di utilità.

Al secondo, si trovano i servizi di sistema, che fanno riferimento al *System Service Manager*:

- *DHCP Service*: assegna dinamicamente gli indirizzi della rete e notifica al sistema il momento ed il punto di accesso in cui un dispositivo di utente si aggancia o si sgancia dalla rete, in collaborazione con il *Location Service*;
- *Ontology Service*: gestisce le ontologie dell'ambiente;
- *Asynchronous Communication Service*: gestisce i canali di comunicazione;

- *Context Service*: definisce un modello di contesto e lo condivide con il resto dell'ambiente;
- *Location Service*: estrae informazioni sulla posizione dei dispositivi installati nell'ambiente e del dispositivo utente;
- *Session Manager*: gestisce le sessioni d'utente e d'ambiente;
- *User Manager Service*: svolge servizi per la gestione degli utenti.
- *Message Service* :fornisce un servizio di notifica di messaggi per l'utente.

Dall'altro lato, invece, si trovano i servizi applicativi, che fanno riferimento all'*Application Service Manager*:

- Servizio di stampa localizzata;
- Servizio di Mp3 JukeBox;
- Servizio di musica d'ambiente;
- Servizio di videoproiezione;
- Servizio di streaming;
- Servizi computazionali (batch).

L'architettura così disegnata consente di disaccoppiare le funzionalità del sistema ed è facilmente scalabile.

Nella rimanente parte del capitolo verranno approfondite nel dettaglio tutte le funzionalità e le responsabilità di ogni singolo componente.

III.5 TASSONOMIA DEI SERVIZI IN UBISYSTEM

Analizzando la tassonomia definita nella progettazione del prototipo,possiamo giungere ad una classificazione molto dettagliata dei servizi (figura III.4).

All'interno dei servizi applicativi, a seconda dell'interazione con l'utente, si distinguono:

- *servizi d'utente*: servizi applicativi a cui l'utente può accedere;
- *servizi d'ambiente*: servizi applicativi con quali l'utente non può interagire direttamente (ad esempio, servizio di musica d'ambiente).

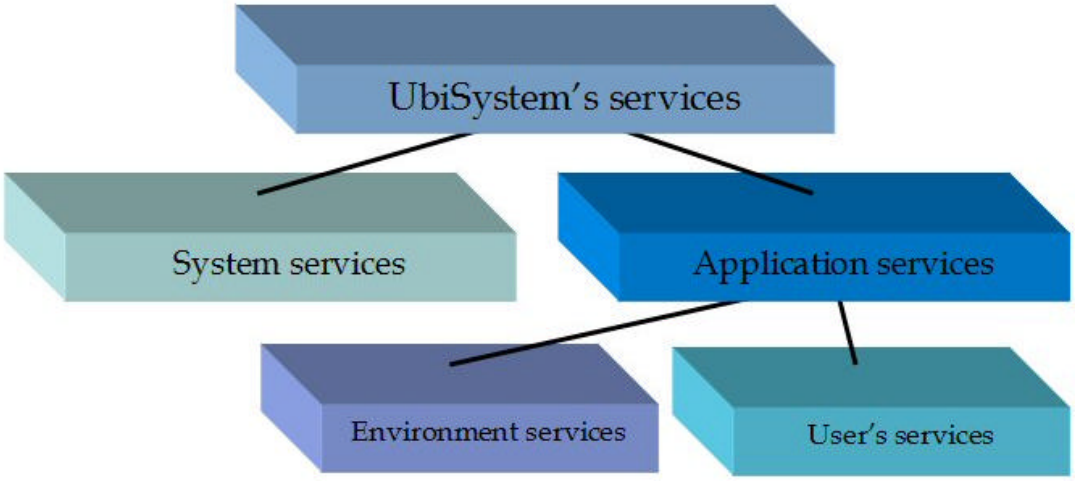


Fig. III.4 Classificazione dei servizi in UbiSystem

Tutti i servizi, inoltre, possono essere soggetti a differenti classificazioni:

- a seconda del tipo di elaborazione fornita si distinguono:
  - *servizi batch*: servizi computazionali;
  - *servizi interattivi*: servizi che richiedono interazione con l'utente.
- a seconda della politica di attivazione applicata si distinguono:
  - *servizi attivi*: servizi che si registrano presso il manager nel momento della loro attivazione, quindi sono su al momento delle richieste provenienti dall'utente;
  - *servizi non attivi*: servizi che vengono attivati automaticamente solo quando richiesti; la registrazione presso il manager viene fatta manualmente dall'amministratore e la descrizione indica il comando da eseguire per portare su il servizio.
- a seconda della dislocazione fisica si distinguono:
  - *servizi centralizzati*: servizi per i quali esiste un unico server;
  - *servizi distribuiti*: servizi per i quali esistono n server su n macchine diverse, in questo caso il manager dei servizi dovrà applicare politiche di load balancing, prima di concedere il servizio.
- a seconda della gestione delle istanze si distinguono:
  - *servizi static*: servizi per i quali esiste una sola istanza utilizzata da tutti i client;
  - *servizi dynamic*: servizi per i quali esiste una istanza per ogni client istanziata dinamicamente da una factory.

III.6 SYSTEM SERVICES MANAGER

E' il componente presso il quale vengono registrati i servizi di sistema, ovvero tutti quei servizi che sono di supporto per l'ambiente pervasivo. Tali servizi possono essere acceduti da parte dell'*Ubiquitous Gateway* e dell'*Application Services Manager* soltanto attraverso questo componente (ad esempio, immaginiamo un servizio di stampa che prima di avviare il processo necessita dell'autenticazione dell'utente che ne faccia la richiesta). Grazie al *System Services Manager,* distinto dal gestore dei servizi applicativi, è possibile avere a disposizione un meccanismo semplice per aggiungere nuovi servizi di supporto per l'ambiente (ad esempio servizi di policy e sicurezza, o magari di QoS) e, allo stesso tempo, di mantenere distinte le due tipologie di servizi sia per ragioni pratiche che amministrative.
Presso il *System Service Manager* dovrà essere registrato anche il nuovo servizio di sistema aggiunto in questo lavoro,il *Message Service*.

Le funzionalità e le responsabilità del *System Services Manager* sono:

- gestire la pubblicazione dei servizi di sistema presenti nell'ambiente all'atto della loro attivazione;
- mantenere e rendere disponibili le descrizioni di tutti i servizi di sistema attivi;
- offrire meccanismi per il discovery dei servizi di sistema disponibili.

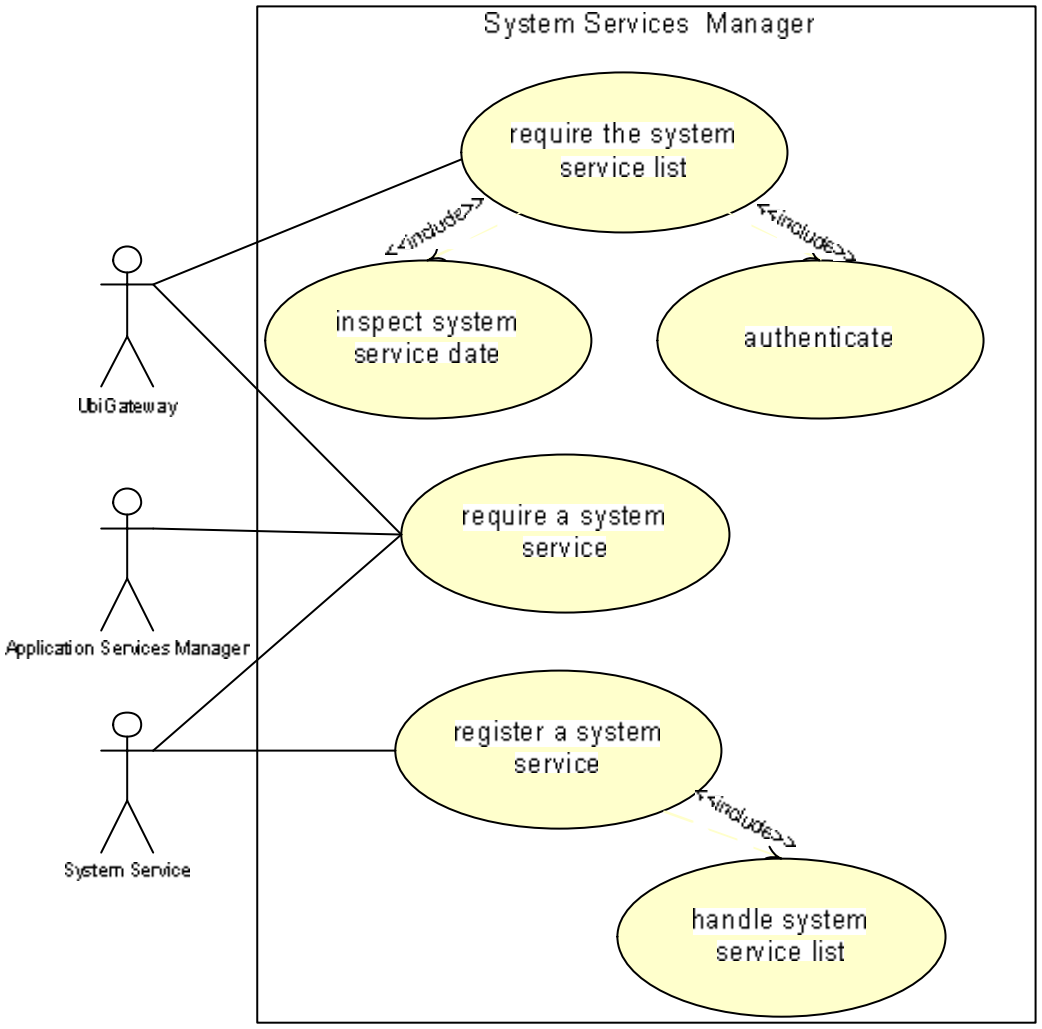


Fig. III.5 Funzionalità offerte dal System Services Manager

III.6.1 SYSTEM SERVICES

I tipi di servizi necessari per coordinare e gestire l'intero sistema pervasivo possono essere molteplici.

Ne sono stati individuati alcuni che riportiamo di seguito e che si propongono di soddisfare i requisiti di un sistema pervasivo.

III.6.1.1 DHCP SERVICE

Le funzioni del *DHCP Service* vanno al di là della semplice assegnazione automatica degli indirizzi ai nuovi elementi che entrano a far parte della rete. Infatti, esso ha la

peculiarità aggiuntiva di notificare al sistema il momento e il punto di accesso in cui un dispositivo si aggancia o si sgancia dalla rete.

Questa caratteristica è molto importante per garantire la proprietà di location-awareness e context-awareness all'interno dell'ambiente pervasivo.

III.6.1.2 ONTOLOGY SERVICE

L'*Ontology Service* è il componente il cui compito è quello di gestire le ontologie utilizzate all'interno dell'ambiente.

Una Ontologia è la descrizione formale di concetti appartenenti ad un determinato dominio applicativo.

L'adozione di modelli ontologici all'interno di sistemi di pervasive computing ha fondamentalmente lo scopo di rendere possibile l'interazione e la collaborazione tra le differenti parti che compongono il sistema.

In particolare, attraverso l'uso di ontologie è possibile:

- definire i termini usati all'interno dell'ambiente, per stabilirne in maniera univoca sintassi e semantica;
- verificare le descrizioni di ciascuna entità affinché rispettino gli schemi definiti;
- intraprendere discovery semantico, per rendere più efficiente la ricerca di oggetti cercando di scoprire tutte e sole le entità rilevanti;
- garantire interoperabilità fra i componenti, fissando definizioni formali ed universali che in particolare descrivono i singoli servizi offerti da ogni risorsa.

Nel progetto, vengono definite ontologie per descrivere i vari aspetti dell'ambiente. Attraverso di esse, viene fissato uno schema logico ed una tassonomia standard interpretabile, oltre che dall'uomo, anche da software automatici.

In particolare, sono state sviluppate ontologie di:

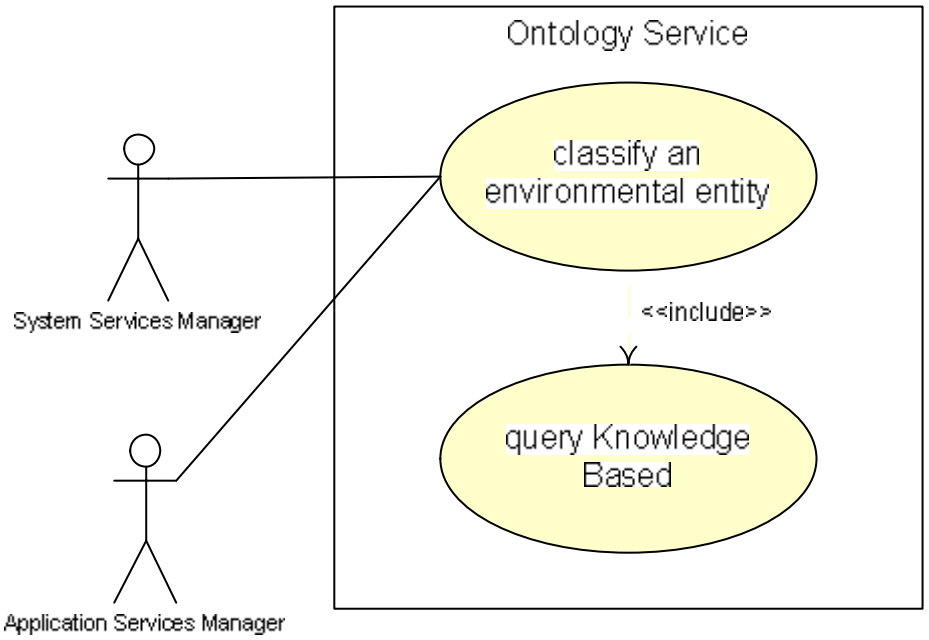
- Contesto: stabilisce concetti inerenti lo stato dell'ambiente: persone presenti, luminosità, temperature, attività in esecuzione, etc.;
- Servizio: stabilisce concetti e caratteristiche che le descrizioni dei servizi disponibili dell'ambiente devono rispettare;
- Device: stabilisce un modello per la caratterizzazione e classificazione dei dispositivi presenti nell'ambiente, comprese peculiarità hardware e software;

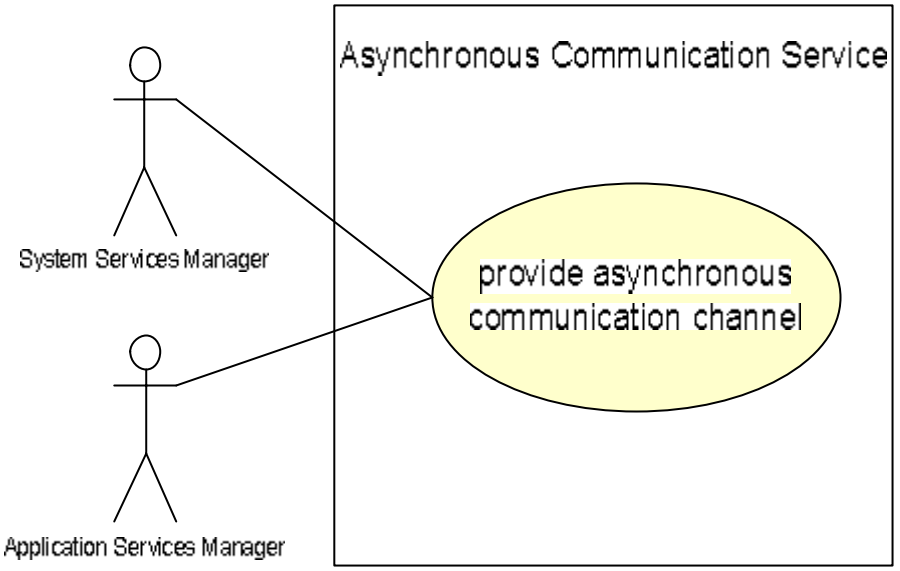
- Utente: stabilisce lo schema concettuale caratterizzante gli utenti degli ambienti pervasivi.

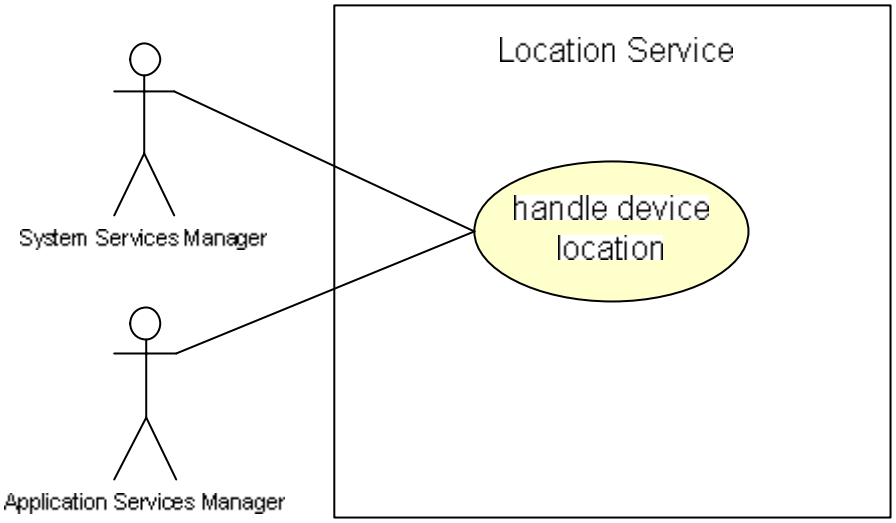
Tutte le ontologie sono sviluppate in linguaggio DAML+OIL (cfr. § II.5.4).

L'*Ontology Service* mette a disposizione un insieme di API (Application Programming Interface) per caricare le ontologie e costruire una Knowledge Base, verificare le descrizioni delle entità, interrogare la Knowledge Base e ricercare definizioni di concetti e relazioni.

La figura III.6 riportata di seguito dipinge le funzionalità offerte da questo componente.




Asynchronous Communication Service
System Services Manager
provide asynchronous communication channel
Application Services Manager


Location Service
handle device location
System Services Manager
Application Services Manager

Si immagini, ad esempio, il seguente scenario: un sensore segnala l'ingresso di una persona nell'ambiente; nel momento in cui la persona entra, è in atto una presentazione PDF; se in un'ontologia è stato definito che durante una presentazione tutte le persone presenti sono dei "partecipanti alla presentazione", si potrebbe concludere che la persona appena entrata è un nuovo partecipante alla presentazione.

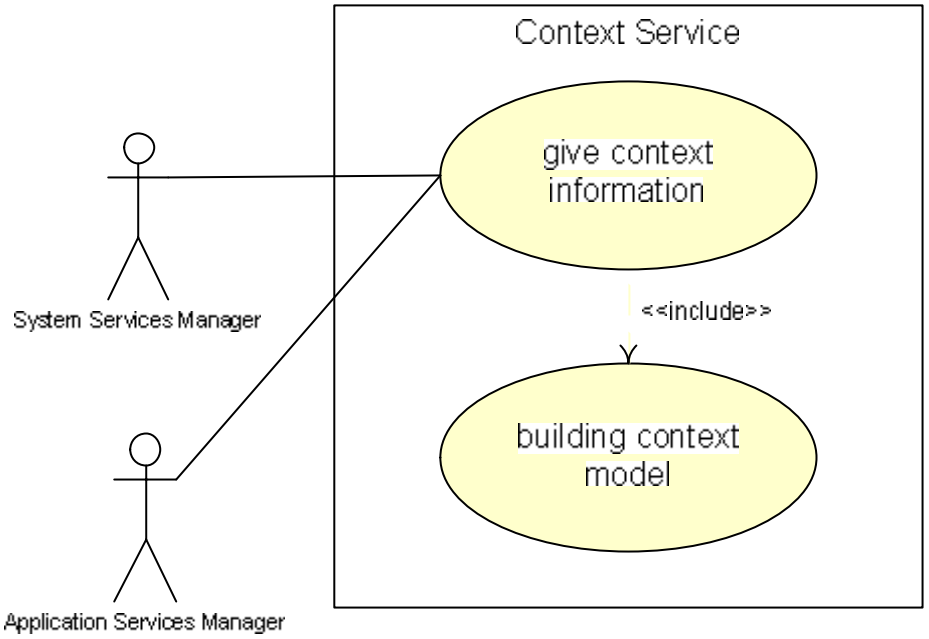
Il *Context Service* definisce il contesto dell'ambiente, sia a partire dai dati raccolti da appositi sensori, sia da quelli provenienti dagli altri componenti del sistema, alcuni dei quali, ad esempio, sono l'*Ubiquitous Gateway*, il *Session Manager* ed il *Location Service*.

In questo modo, il *Context Service* è informato sul numero e sull'identità degli utenti correntemente presenti all'interno dell'ambiente, dei servizi disponibili, dello stato delle applicazioni.

Le applicazioni ed i servizi dell'ambiente, a partire dalle informazioni di contesto ricevute dal *Context Service*, possono adattare il proprio comportamento secondo le diverse situazioni. Per esempio, un'applicazione di gestione dell'illuminazione di un ambiente potrebbe automaticamente accendere le luci regolandone l'intensità secondo le preferenze delle persone presenti all'interno dell'ambiente; oppure, potrebbe regolare l'intensità dell'illuminazione in dipendenza dall'ora del giorno.

Anche in questo caso, l'impiego di ontologie rende più semplice per gli sviluppatori specificare il comportamento dei propri applicativi secondo le diverse esigenze.

La figura III.9 riportata di seguito dipinge le funzionalità offerte da questo componente.


Context Service
give context information
<<include>>
building context model
System Services Manager
Application Services Manager

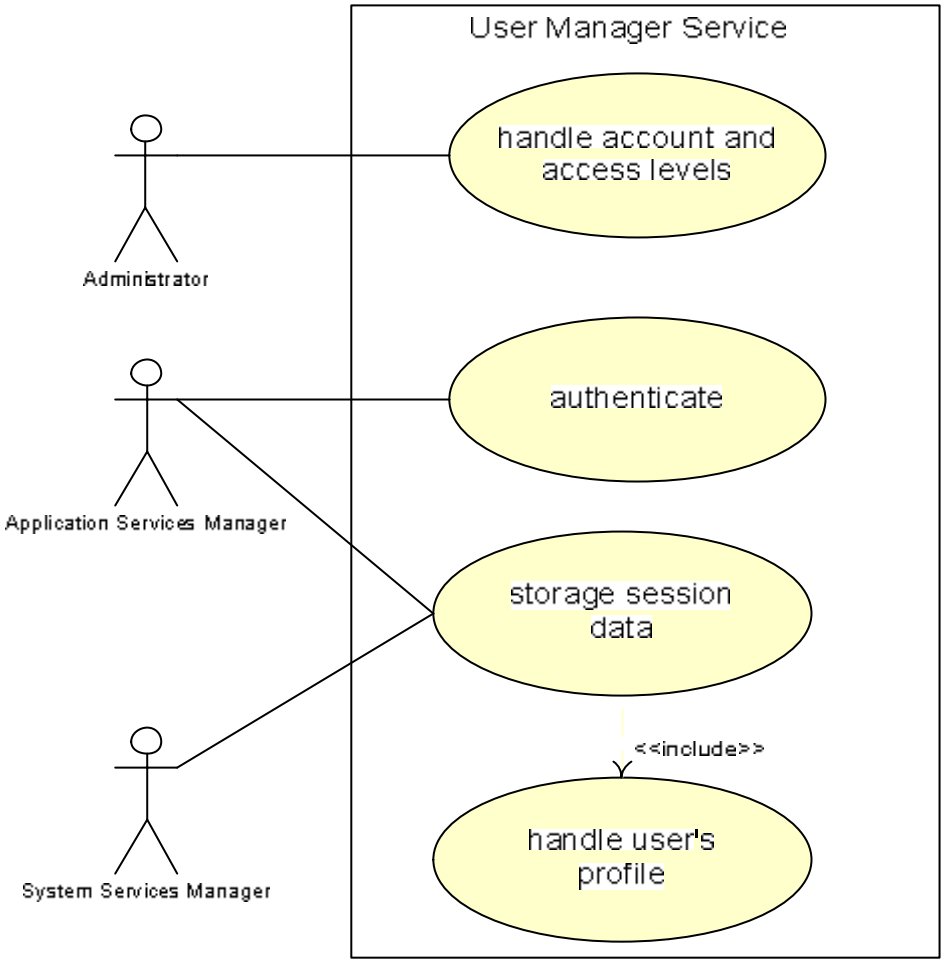


Fig. III.10 Funzionalità offerte dall'User Manager Service

III.6.1.7 SESSION MANAGER

Il *Session Manager* rappresenta il componente che si occupa della gestione delle sessioni.

In particolare il servizio deve:

- gestire una sessione per i servizi applicativi d'ambiente attivi nel sistema;
- gestire una sessione di lavoro per ciascun utente presente nell'ambiente;
- verificare la validità delle sessioni d'utente.

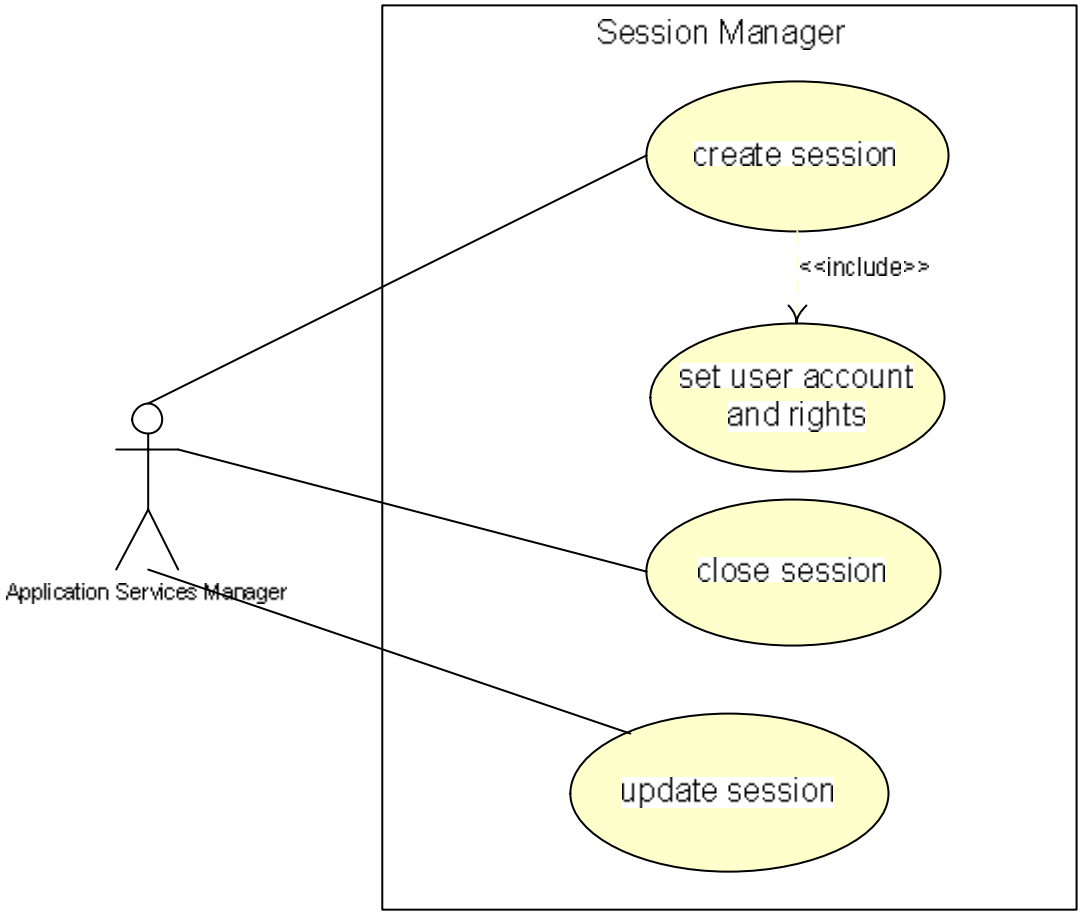


Fig. III.11 Funzionalità offerte dal Session Manager

III.6.1.8 MESSAGE SERVICE

Il *Message Service* rappresenta il componente che si occupa della gestione dei messaggi.

In particolare il servizio deve:

- notificare l'arrivo di nuovi messaggi per l'utente;
- consentire all'utente di poter gestire i propri messaggi prevedendo eventualmente la cancellazione degli stessi,una volta che siano stati letti.

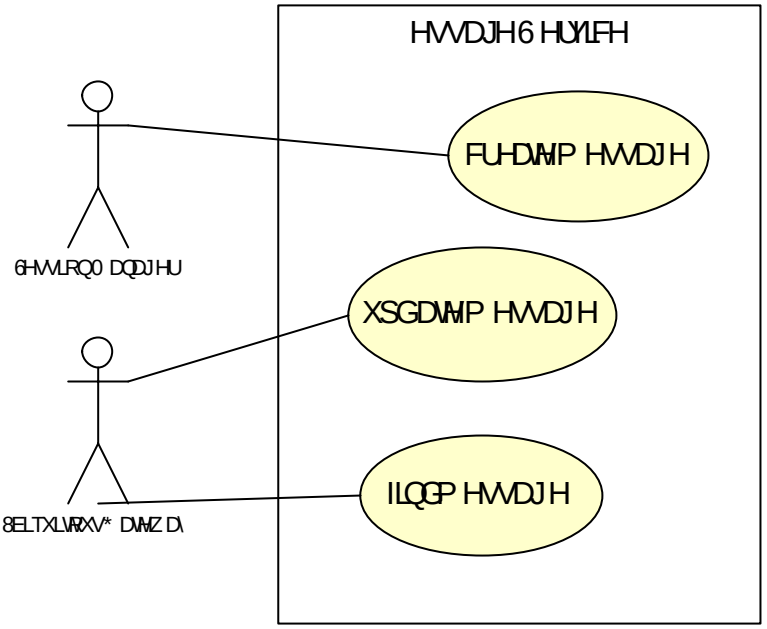


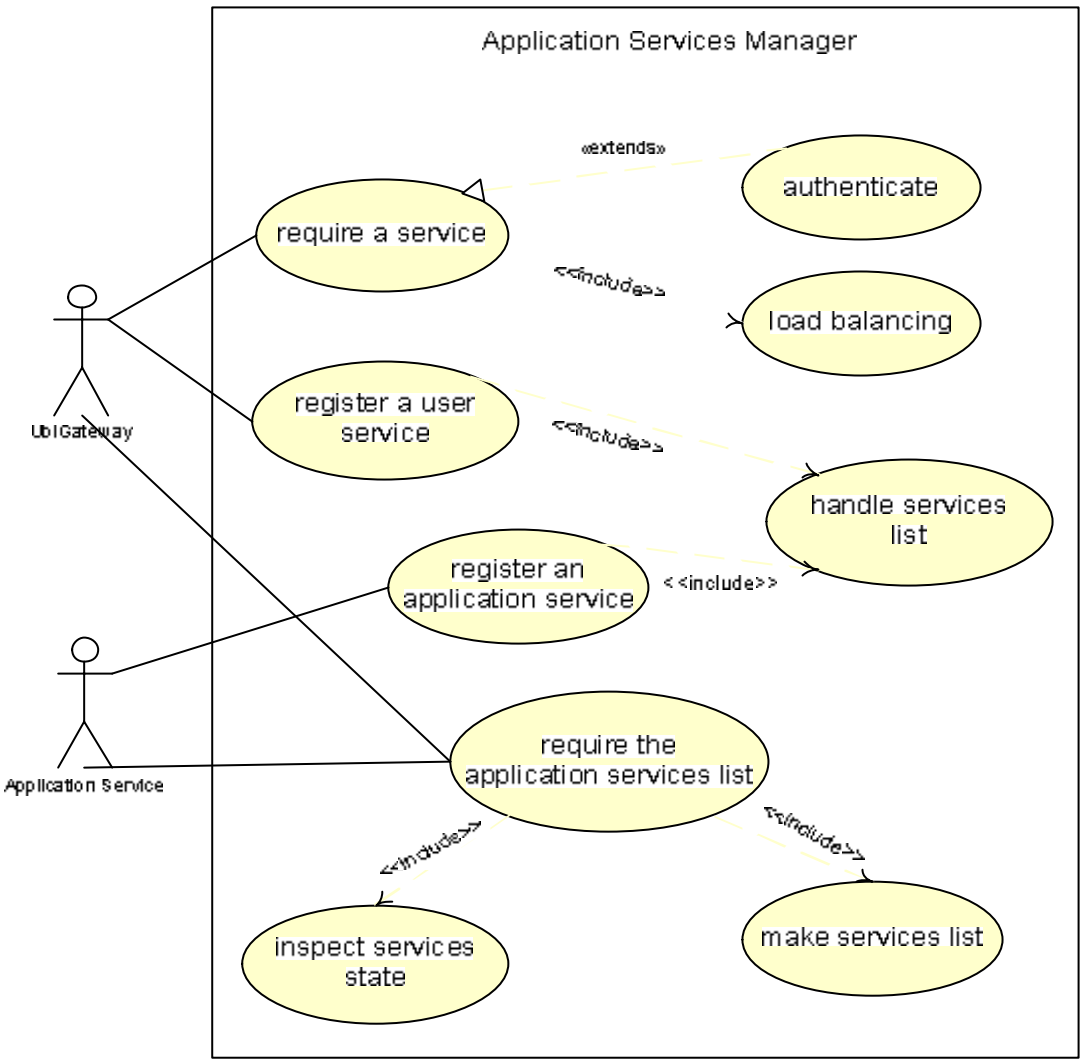
Fig. III.12 Funzionalità offerte dal Message Service

III.7 APPLICATION SERVICES MANAGER

Rappresenta il componente referente per i *service providers* che intendono pubblicare un nuovo servizio.

Le funzionalità e le responsabilità dell'*Application Services Manager* sono:

- gestire la pubblicazione dei servizi offerti dall'utente;
- gestire la pubblicazione dei servizi applicativi presenti nell'ambiente all'atto della loro attivazione;
- mantenere e rendere disponibili le descrizioni di tutti i servizi applicativi attivi nel sistema;
- offrire meccanismi per il discovery dei servizi disponibili;
- applicare politiche di bilanciamento del carico prima di consentire l'uso di un servizio.


Application Services Manager
«extends»
authenticate
require a service
<<include>>
load balancing
register a user service
<<include>>
handle services list
register an application service
<<include>>
UbiGateway
Application Service
require the application services list
<<include>>
<<include>>
inspect services state
make services list

Si tratta di un servizio applicativo d'utente e interattivo.

III.7.1.2 SERVIZIO DI MP3 JUKEBOX

Il servizio consente agli utenti di creare ed ascoltare playlist di brani musicali. Quest'ultima può includere sia brani che si trovano sul server che brani proposti dall'utente stesso.

Si tratta di un servizio applicativo d'utente e interattivo.

III.7.1.3 SERVIZIO DI MUSICA D'AMBIENTE

Il servizio provvede alla gestione della musica di sottofondo di una determinata stanza a tema in dipendenza della presenza degli utenti.

Il servizio provvede alla gestione della musica di sottofondo di una determinata stanza a tema.

Si tratta di un servizio applicativo d'ambiente e non espone funzionalità accessibili direttamente dagli utenti.

III.7.1.4 SERVIZIO DI VIDEOPROIEZIONE

Il PdfViewer è un servizio che permette ad un utente di tenere una presentazione per una conferenza. Il relatore attraverso, il suo palmare, una volta avuto accesso al servizio, può inviare il file pdf da visualizzare e comandare la presentazione attraverso il proprio dispositivo.

Si tratta di un servizio applicativo d'utente e interattivo.

III.7.1.5 SERVIZIO DI STREAMING

Il servizio si occupa di far visualizzare all'utente, sul proprio dispositivo, un filmato diverso a seconda della sua locazione.

Si tratta di un servizio applicativo d'utente e interattivo.

III.7.1.6 SERVIZI BATCH

Di tale tipo di servizi viene dato un modello generale,che prescinde dalla logica di una particolare applicazione. Viene fornito quindi un paradigma da seguire se si vuole aggiungere un servizio applicativo di utente di tipo batch nell'ambiente. Di tale tipologia di

applicazioni forniamo una descrizione delle funzionalità mediante i seguenti casi d'uso (figura III.14).

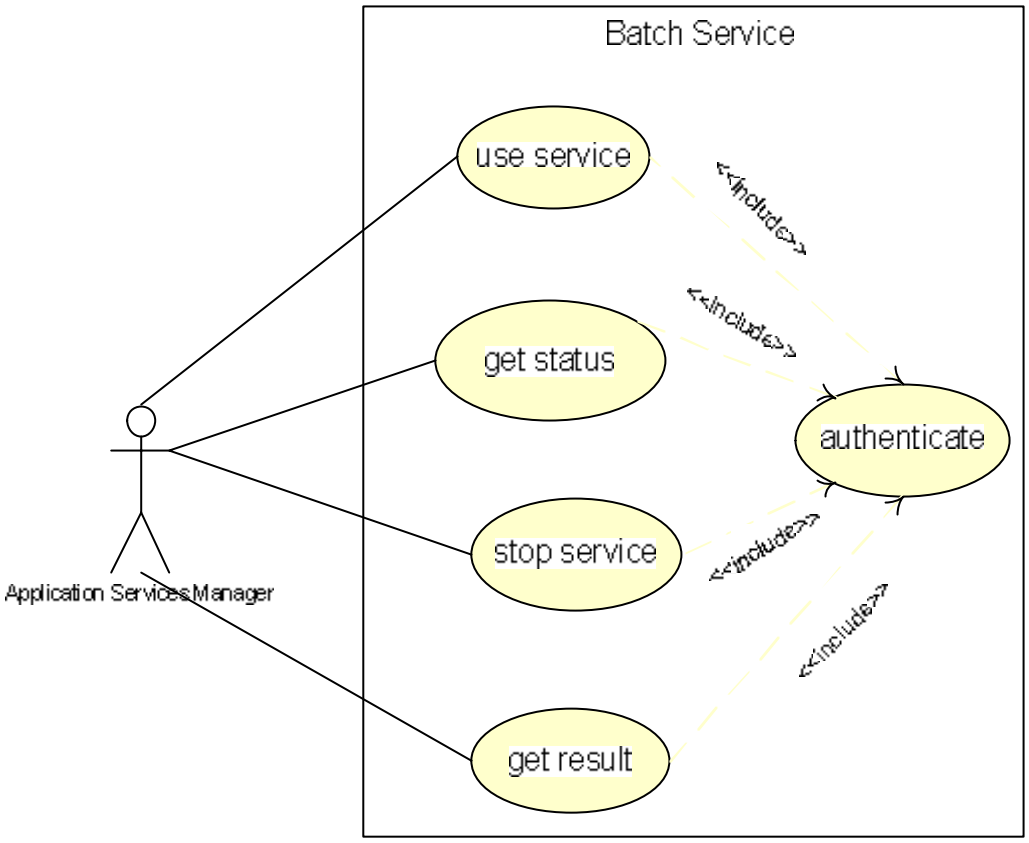


Fig. III.14 Funzionalità offerte dal generico servizio batch

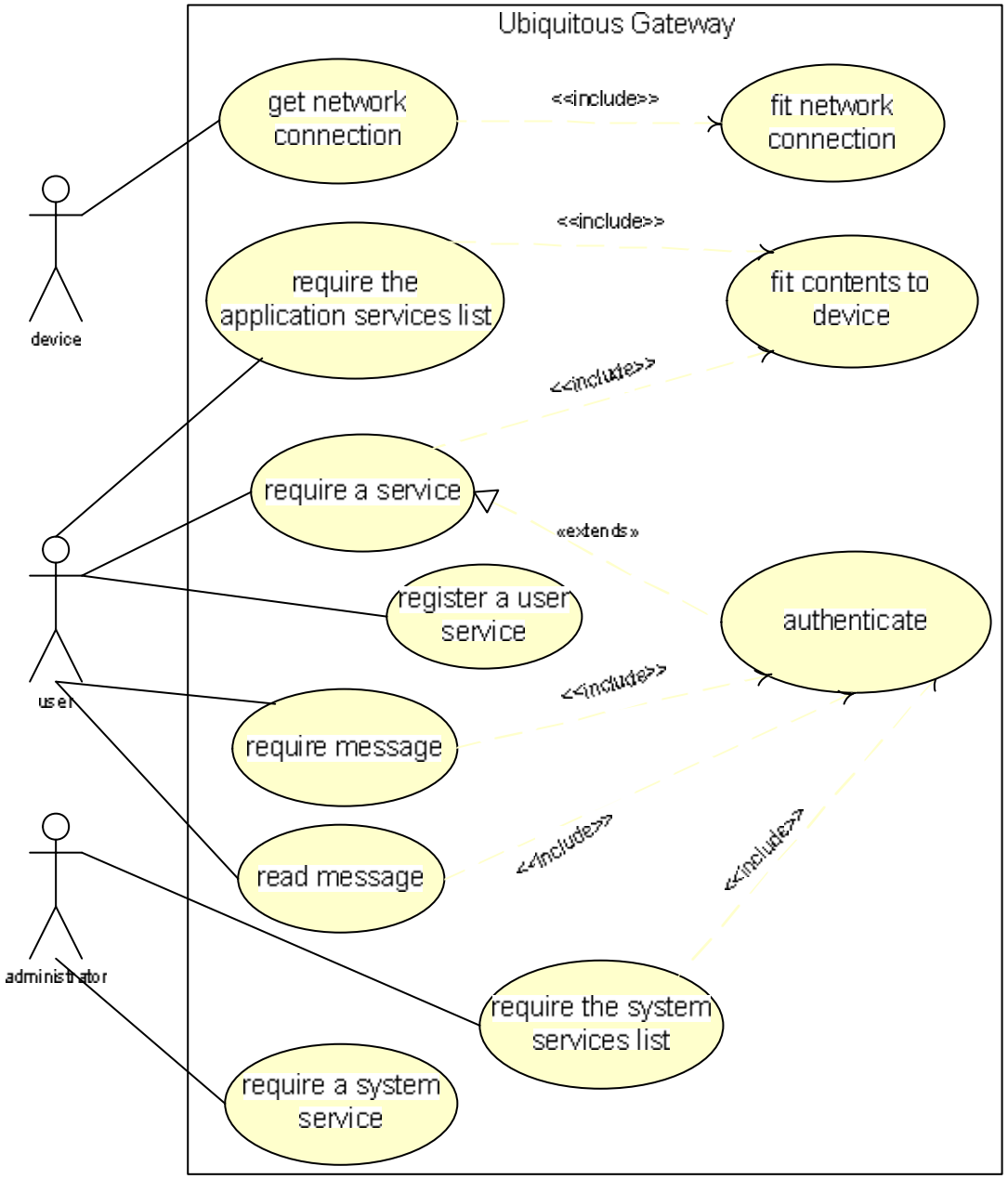
III.8 UBIQUITOUS GATEWAY

L'*Ubiquitous Gateway* rappresenta l'entry point per gli utenti del sistema verso l'ambiente pervasivo.

Da un punto di vista funzionale, l'*Ubiquitous Gateway* dovrà:

- Registrare e autenticare gli utenti che effettuano l'accesso;
- Reperire la descrizione del dispositivo mediante il quale l'utente interagisce con l'ambiente;
- Offrire meccanismi per ricercare ed utilizzare le risorse disponibili nell'ambiente.
- Offrire all'utente delle funzionalità di gestione dei messaggi.

La figura III.15 dipinge le funzionalità offerte da questo componente.



IV. COMPONENTI DI SISTEMA IMPLEMENTATI

IV.1 INTRODUZIONE

UbiSystem è un'infrastruttura Web-based che mira a gestire e rendere possibile la comunicazione tra entità di un ambiente pervasivo, cercando di garantire tutta una serie di requisiti, quali:

- Supporto per l'eterogeneità di dispositivi, piattaforme e linguaggi;
- Definizione e condivisione dello stato del contesto dell'ambiente;
- Discovery semantico dei servizi e supporto per l'integrazione;
- Gestione della dinamicità e della disponibilità delle risorse presenti.

Esso si preoccupa, fondamentalmente, di consentire ad un utente, che accede al sistema mediante un qualsiasi dispositivo portatile (palmare, cellulare, laptop), di poter usufruire, in seguito ad una fase di autenticazione, di una serie di servizi e risorse disponibili nell'ambiente senza dover procedere all'installazione di alcun software aggiuntivo né di dover eseguire procedure di configurazione di alcuna sorta, ma adoperando un comune internet browser.

I componenti del sistema sono realizzati come servizi CORBA e solo se necessario, ovvero quando espongono funzionalità che devono essere rese accessibili dall'esterno, sono esposti come Web Services.

La scelta di CORBA è dovuta al fatto che si tratta di una tecnologia sicura ed efficiente, ed inoltre anche implementazioni gratuite di piattaforme CORBA offrono servizi, ad esempio l'Event ed il Notification Service, in grado di offrire meccanismi per la realizzazione di tipologie di comunicazione, tra oggetti, diversa da quella sincrona.

Per la realizzazione dei componenti che saranno descritti nel prosieguo del capitolo si è scelto di utilizzare, come implementazione della infrastruttura CORBA, JacORB.

JacORB è un Java object request broker open source, molto popolare (usato da enti di ricerca e in campo aziendale) ed efficiente che, progettato nel 1995 in un dipartimento della Freie Universit¨at Berlin (FUB), continua ad essere usato per ricerche alla FUB, specialmente nel campo della sicurezza di oggetti distribuiti.

La versione utilizzata è JacORB 2.2, comprensiva di alcuni servizi tra i quali il Name Service ed il Notification Service.

Nel precedente capitolo abbiamo descritto i servizi, sia applicativi che di sistema,che caratterizzano l'attuale implementazione del prototipo *UbiSystem*.

A partire da tale scenario,nel presente capitolo e nel successivo verranno illustrati i nuovi componenti inseriti nel prototipo, oggetto della presente tesi, e verrà descritta l'intera fase di progettazione svolta, nonchè le scelte implementative adottate.

I nuovi componenti progettati e implementati nell'ambito della presente tesi di laurea sono:

- *Message Service:* si occupa della gestione dei messaggi da notificare all'utente. Attualmente questo componente provvede a gestire soltanto i messaggi relativi alla terminazione dei servizi computazionali,creandone di nuovi e inserendo in essi tutte le informazioni necessarie affinché l'utente possa,nel nostro caso,prelevare il risultato dell'elaborazione batch avviata in un precedente momento; non è escluso comunque che sviluppando in futuro nuove funzionalità del prototipo questo componente possa gestire anche altri tipi di messaggi che possano essere utili all'utente.

- *Batch service:* E' un tipo di servizio non previsto nel precedente prototipo di *UbiSystem*.Di esso viene dato un modello generale,che prescinde dalla logica di una particolare applicazione. Viene fornito quindi un paradigma da seguire se si vuole aggiungere un servizio applicativo di utente di tipo batch nell'ambiente. Di questo tipo di applicazioni e sulla maniera di integrarli nell'ambiente si parlerà diffusamente nel prossimo capitolo.

Oltre a ciò,una non minore parte del lavoro è stata dedicata con molta attenzione a rivedere quegli aspetti della precedente architettura del prototipo,che devono essere considerati per permettere le interazioni dei nuovi componenti nel sistema. Alla luce di questa analisi sono state effettuate delle modifiche ai moduli preesistenti,al fine di permettere un funzionamento coerente con l'introduzione dei nuovi componenti.

Procediamo quindi ad esporre tutti i passi della progettazione fino a giungere ai dettagli implementativi,sia per quanto riguarda i moduli nuovi,sia per quanto riguarda la revisione di alcuni degli elementi preesistenti.

IV.2 BUSINESS CLASS DIAGRAMS

Lo scopo di questo tipo di diagramma è l'individuazione dei metodi e degli attributi di cui deve essere dotato ogni componente. A partire dagli Use Case Diagrams si cerca di tradurre le funzionalità esterne individuate in modo da ottenere i metodi necessari all'interazione dei diversi moduli del sistema.

IV.2.1 UBIQUITOUS GATEWAY

L'*Ubiquitous Gateway* rappresenta l'entry point per gli utenti del sistema e per il loro dispositivo verso l'ambiente pervasivo.

Le funzionalità che questo componente presenta sono:

- *Get network connection.* Permette, a livello logico, la connessione fisica del dispositivo utente.

- *register a user device*. Quando un utente entra nel sistema lo fa accedendo alla home page di UbiSystem. In questo stesso momento *Ubiquitous Gateway* deve provvedere alla registrazione del dispositivo. Per fare ciò è necessario prelevare le caratteristiche del dispositivo e modellarle in un file scritto in linguaggio DAML+OIL. Tale file dovrà essere inviato all'*Ontology Service* per essere validato e registrato. Il metodo che realizza la funzionalità esposta è: *set_characteristics()*.

- *require the application services list*. Il componente in esame offre all'utente la possibilità di visualizzare la lista dei servizi disponibili nell'ambiente. Se tali servizi sono disponibili l'utente potrà utilizzarli, altrimenti viene indicato il motivo che ne impedisce l'utilizzo. Egli avrà anche la possibilità di autenticarsi per ampliare il numero di servizi che può utilizzare. Il metodo che realizza la funzionalità esposta è: *get_all_services()*, ma se l'utente si autentica può invocare direttamente il metodo: *authenticate()*.

- *require a service*. Se il servizio risulta disponibile, il metodo che ne permette l'utilizzo sarà: *require_application_service()*.

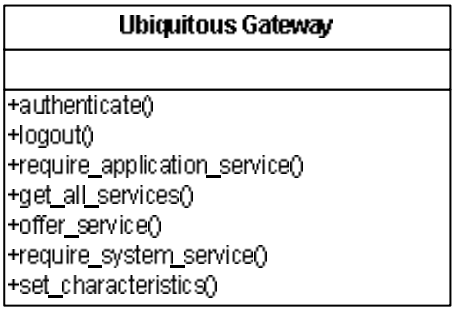
- *register a user service*. Tale funzionalità permette ad un utente di poter mettere a disposizione, all'interno dell'ambiente pervasivo, un proprio servizio. Il metodo previsto per tale scopo è: *offer_service()*.

- *require the system services list*. L'amministratore deve avere la possibilità di visualizzare la lista dei servizi di sistema per poi poterne sfruttare le funzionalità. Per fare questo, però, è necessario che egli si autentichi. Il metodo necessario sarà: *authenticate()*.

- *require a system service*. Analogamente a quanto visto per i servizi applicativi, *Ubiquitous Gateway* dovrà esporre un metodo per l'utilizzo dei servizi di sistema. Questo si chiama: *require_system_service()*.

- *logout*. Tale funzionalità offre all'utente un modo per comunicare la sua uscita dal sistema,attraverso un metodo *logout* ():

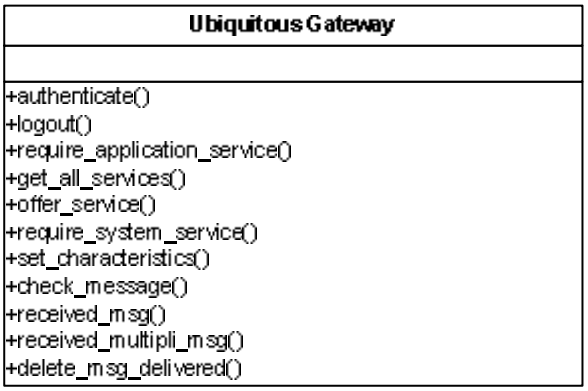
Quanto esposto può essere schematizzato come segue (figura IV.1) :



- *received_message:* è una funzionalità che consente di gestire la lista dei messaggi ,in particolare permette di distinguere i messaggi nuovi(cioè non ancora letti) da quelli già letti ,ma non ancora cancellati. E' infatti previsto che i messaggi possano essere riproposti all'utente se egli sceglie di prenderne visione ma di non cancellarli. Il metodo dell' *Ubiquitous Gateway* che consente tali operazioni sarà: *received_msg().*

- *delete_message_delivered:* l'utente può decidere di cancellare i messaggi che ha già letto e che non vuole gli siano riproposti successivamente. Ciò avviene attraverso il metodo esposto *delete_msg_delivered().*

Le nuove funzionalità descritte vanno dunque integrate alle precedenti;di conseguenza la schematizzazione vista prima diventa (figura IV.2) :



Il *Session Manager* gestisce le sessioni d'utente e una sessione di "*Environment*".

La sessione di *"Environment"* viene creata all'atto della registrazione del primo servizio d'ambiente e viene aggiornata man mano che si registrano gli altri. Essa tiene traccia di ciascun servizio di questo tipo attivo nel sistema e viene distrutta quando tali servizi vengono tutti disattivati.

Le sessioni d'utente, invece, tengono traccia degli utenti presenti nell'ambiente.

Ogni sessione viene creata all'ingresso dell'utente nel sistema, il che si verifica con la prima connessione alla home page di *UbiSystem*. L'*UG* provvede al reperimento del file di descrizione del device e alla registrazione di quest'ultimo.

Se tale operazione va a buon fine esso richiede al *Session Manager* la creazione di una nuova sessione.

Ogni qualvolta un utente utilizza un servizio, la relativa sessione viene aggiornata con il riferimento della particolare istanza assegnata all'utente.

Nel momento in cui l'utente smette di utilizzare un servizio interattivo, oppure un servizio batch completa la sua elaborazione, questo evento viene notificato all'interno dalla sessione.

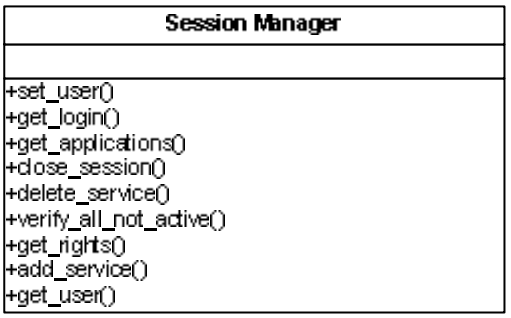
Per quanto riguarda la chiusura della sessione si possono verificare due casi:

1. l'utente comunica al sistema, per mezzo del logout, di voler abbandonare l'ambiente;
2. l'utente lascia il sistema senza una comunicazione esplicita.

E' da rilevare che nella versione precedente del prototipo, con il logout, vengono chiusi tutti i servizi applicativi aperti dall'utente,che nella fattispecie sono soltanto di tipo interattivo e la sessione viene distrutta.

Per il secondo caso, invece, è previsto che ad ogni sessione sia associato un agente che, periodicamente, verifica il tempo trascorso dall'ultima interazione con il sistema. Se vi è stato un certo periodo di inattività, la sessione scade e ne viene effettuata la chiusura.

Le funzionalità del componente *Session Manager*,così come esso era stato concepito nel precedente prototipo, sono elencate nel diagramma seguente (figura IV.3) :


Session Manager
+set_user()
+get_login()
+get_applications()
+close_session()
+delete_service()
+verify_all_not_active()
+get_rights()
+add_service()
+get_user()

L'introduzione di un nuovo tipo di sessione ,che abbiamo chiamato *sleeping*, appunto per distinguerla dalla normale sessione utente attiva, comporterà,come vedremo più diffusamente nei dettagli implementativi,la risoluzione di nuovi problemi di gestione della sessione,che prima ovviamente non ricadevano. In particolare è interessante evidenziare che la nuova possibilità di non dover chiudere necessariamente una sessione per un utente che esce dal sistema,ma di considerare un logout definitivo solo nel caso in cui egli non lasci applicazioni batch in esecuzione, implica il dover gestire diversamente rispetto alla soluzione preesistente non solo il logout,ovvero il momento dell'uscita dell'utente,ma anche il suo successivo rientro in *UbiSystem;* infatti, a differenza di quanto accadeva prima, all'atto dell'autenticazione non si deve soltanto creare una nuova sessione, bensì controllare che un utente che si autentica non abbia lasciato in precedenza una sessione in *sleeping*. In quest'ultimo caso la creazione di una sessione ex novo si rivela inutile quanto scorretta,mentre risulterà più giusto ed efficace,dopo opportuni controlli, andare a tramutare la sessione *sleeping* in questione, in una sessione nuovamente attiva.

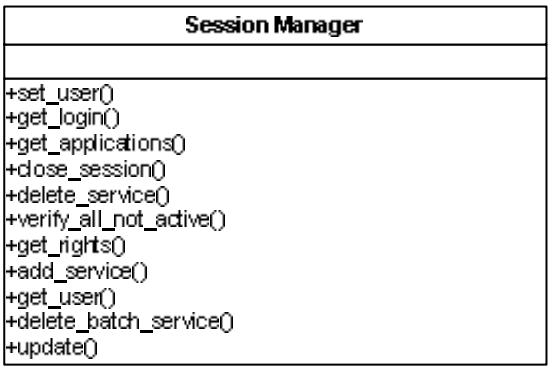
Per il problema simmetrico dell'uscita dell'utente dal sistema e dell'eventuale trasformazione della sessione in *sleeping* ,dobbiamo tener presente che ,in linea di principio, uno stesso utente può interagire con il sistema con più dispositivi contemporaneamente. Ciò accade perché già nel preesistente prototipo, come identificativo di ciascuna sessione è stato scelto l'indirizzo ip del dispositivo stesso:di conseguenza uno stesso utente ,connesso con più dispositivi,avrà più sessioni attive,una per ciascun dispositivo. Cercando di mantenere questa scelta di progetto,ci è sembrato allora opportuno considerare che ,se l'utente ha più sessioni attive,dovute a diversi device in suo possesso, e disattiva una sola delle sue sessioni, i servizi eventualmente sospesi per quest'ultima vengono accodati ad una delle sessioni ancora attive dello stesso utente.In questo modo per il generico utente ci sarà sempre al più una sola sessione di *sleeping*.

Nel progettare la soluzione ai problemi esposti abbiamo quindi provveduto alla modifica di alcune funzionalità del *Session*

Manager ,tenendo comunque conto del prototipo preesistente, che costituiva la base di studio del presente lavoro: di conseguenza le scelte adottate sono volte a rispettare,nei limiti del possibile, quelle fatte in precedenza.

Seguendo tale principio e tenendo conto del fatto che la tesi presente riguarda l'integrazione e la gestione delle applicazioni batch in *UbiSystem*, si è ritenuto opportuno non modificare al momento la gestione dei servizi applicativi di tipo interattivo,adottando per ora solo per quelli di tipo batch ,nuove soluzioni di gestione che ci sembrano più appropriate.

Dall'aggiunta delle nuove funzionalità descritte in questo componente ricaviamo allora il seguente diagramma (figura IV.4) :



Nel momento in cui viene chiusa la sessione di un utente autenticato, il *Session Manager* provvede ad inviare all'*User Manager Service* tutti i dati relativi all'attività svolta. In questo modo l'*User Manager Service* ha la possibilità di eseguire degli studi statistici sulle preferenze dei diversi utenti.

Con la gestione delle sessioni d'utente, il sistema è informato sul numero e sull'identità degli utenti correntemente presenti al suo interno e sulle applicazioni e i dispositivi da essi in uso. Tali informazioni sono accessibili da parte dall'amministratore, attraverso un'interfaccia grafica.

Per memorizzare le sessioni si utilizza una struttura denominata *SessionsDirectory* che permette di inserire, rimuovere, aggiornare e prelevare le sessioni in essa contenute.

IV.2.3 MESSAGE SERVICE

Il *Message Service* rappresenta il componente che si occupa della creazione dei messaggi, inserendo tutte le informazioni necessarie affinché l'utente che riceve il messaggio possa,nel nostro caso, prelevare il risultato dell'elaborazione batch avviata in un precedente momento.

Le funzionalità che questo componente deve offrire sono:

- *find message*. Un utente deve poter verificare se ci sono nuovi messaggi che notificano la terminazione di qualche applicazione avviata in precedenza. Questo componente offre questa possibilità esponendo un metodo opportuno che si chiamerà: *find_message()*. Dopo aver eseguito questa operazione l'utente sarà a conoscenza di quanti sono i nuovi messaggi e potrà visualizzarli.

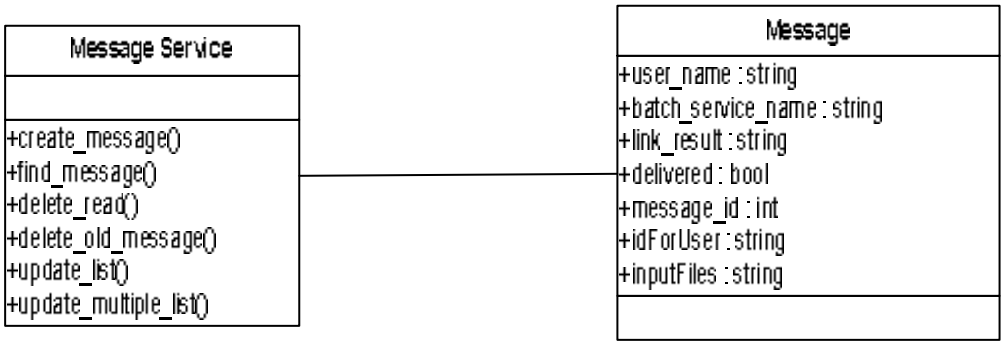
- *update message list*. Consente di gestire la lista dei messaggi: in particolare permette di distinguere i messaggi non ancora letti da quelli già letti ,ma non ancora cancellati. E' infatti previsto che i messaggi possano essere riproposti all'utente se egli sceglie di prenderne visione ma di non cancellarli. Il metodo che consente tali operazioni sarà: *update_list()*.

- *delete read message.* Possono,a richiesta, essere cancellati i messaggi che l'utente ha già letto .Può essere scelta tale opzione se non si vuole che i messaggi letti siano conservati e riproposti in seguito con quelli nuovi. Ciò avviene attraverso il metodo esposto *delete_read().*

- *delete old message.* Questa funzionalità può essere creata per consentire all'amministratore del sistema di cancellare periodicamente i messaggi precedenti ad una certa data.Ciò può avvenire attraverso un metodo *delete_old_message().*

- *create message.* Serve a creare i messaggi da inviare poi all'utente.Il metodo che si occupa di ciò sarà *create_message_user(),* e si farà carico di costruire e conservare il messaggio per l'utente finale,fornendogli tutte le informazioni utili affinché egli possa poi prelevare il risultato dell'elaborazione.In particolare,poiché è previsto che uno stesso utente possa richiedere contemporaneamente diverse esecuzioni dello stesso servizio batch ,nel messaggio deve essere previsto anche un identificativo della particolare esecuzione terminata.

Le funzionalità progettate possono essere schematizzate come segue (figura IV.5):



IV.3 SEQUENCE DIAGRAMS

Un *Sequence Diagram* rappresenta un'interazione visualizzata rispetto a una sequenza temporale necessaria a portare a termine una procedura. Esso mostra gli oggetti che partecipano all'interazione in termini del loro tempo di vita e dei messaggi che essi si scambiano.

Lo scopo di questo paragrafo è quello di mostrare le interazioni tra i vari moduli del sistema per portare a termine le nuove funzionalità integrate che il sistema *UbiSystem* deve offrire. Al termine di questa analisi saranno eventualmente individuati ulteriori metodi che i vari componenti dovranno implementare per soddisfare i requisiti del sistema.

IV.3.1 REGISTRAZIONE DI UN SERVIZIO DI SISTEMA

Come accade per gli altri servizi di sistema già esistenti,anche il nuovo servizio di messaggistica *Message Service* deve registrarsi presso il proprio manager per poter fornire i suoi servizi.

Affinché un servizio di sistema possa registrarsi è necessario che il *System Services Manager* sia attivo e abbia reso noto il proprio riferimento.

Inoltre consideriamo che all'atto della registrazione di un qualsiasi servizio di sistema, l'*Ontology Service* è già registrato ed attivo;infatti senza tale componente sarebbe impossibile registrare gli altri servizi.

Quando l'amministratore attiva il servizio questo provvede a registrarsi presso l'*SSM*. La registrazione coinvolge anche l'*Ontology Service*; infatti esso ha il compito di validare la descrizione del servizio, cioè deve verificare che essa sia coerente con il resto del sistema. Di seguito (figura IV.6) si riporta il sequence relativo generalizzato per un qualsiasi servizio di sistema.Ovviamente per quanta riguarda il presente lavoro il nostro generico System Service si particolarizza in *Message Service*.

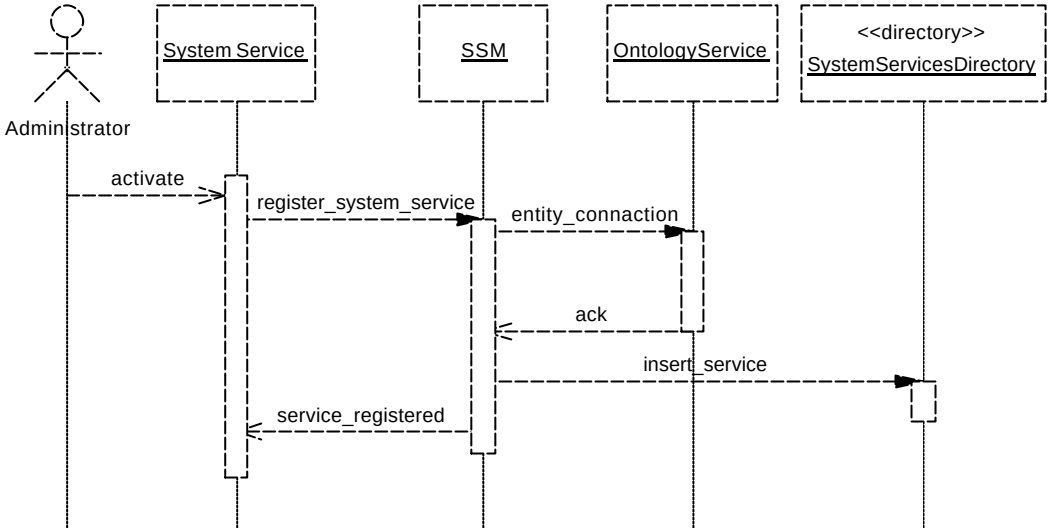


Fig. IV.6 Registrazione di un servizio di sistema

Al momento dell'attivazione da parte dell'amministratore il servizio reperisce il reference dell'*SSM* e richiede la propria registrazione. Affinché l'operazione vada a buon fine i dati di cui ha bisogno il manager sono: il file di descrizione del servizio, il suo riferimento e il nome simbolico con il quale sarà registrato e successivamente reperibile nel sistema.

IV.3.2 REGISTRAZIONE DI UN SERVIZIO APPLICATIVO

Similmente a quanto detto prima ,anche il nuovo servizio applicativo batch introdotto dovrà registrarsi presso il proprio manager. In tal caso però è necessario che, oltre all'*SSM* e all'*Ontology Service*, sia già attivo anche l'*Application Services Manager*, perché proprio tale componente è preposto alla gestione dei servizi applicativi. I passi relativi a tale operazione di registrazione sono riportati nella figura IV.7 seguente:

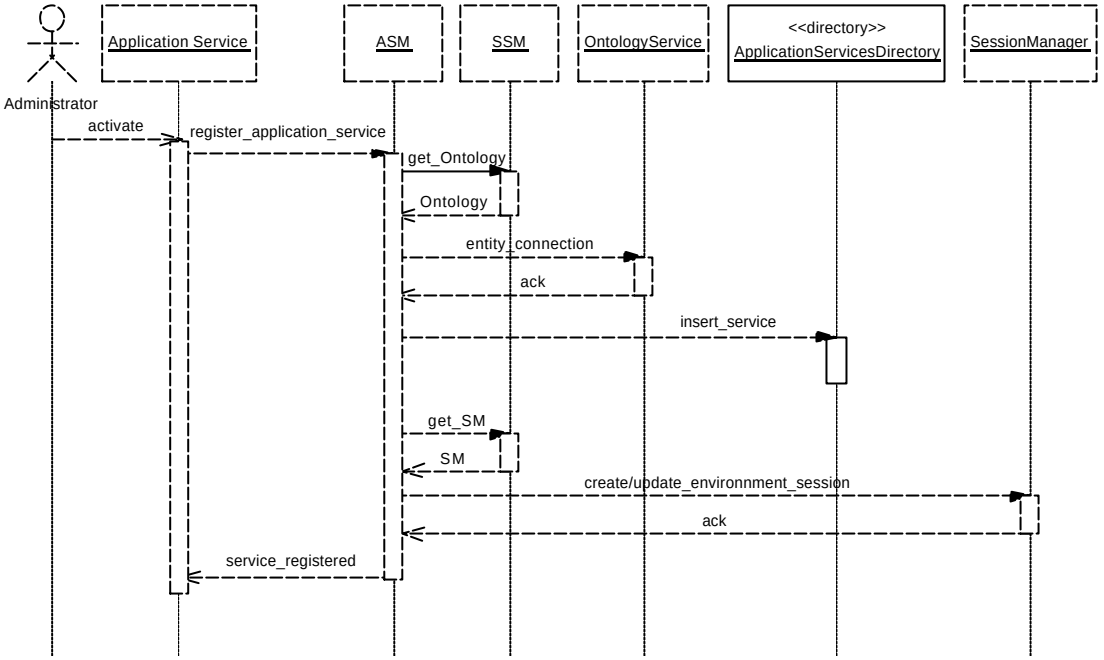


Fig. IV.7 Registrazione di un servizio applicativo

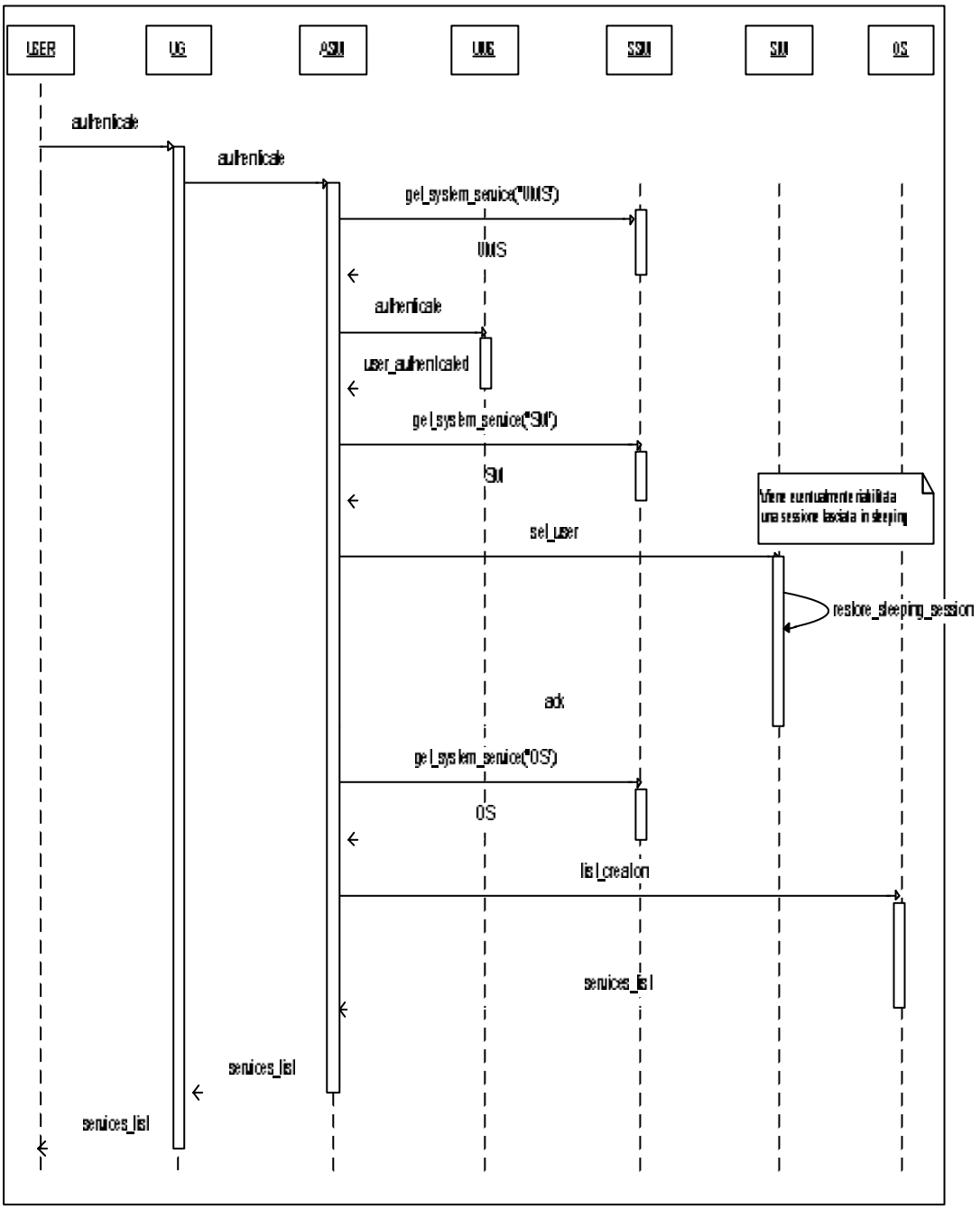
Al momento dell'attivazione, il servizio reperisce il reference dell'*Application Services Manager* ed esegue la richiesta di registrazione inviando il proprio file di descrizione, il reference e il nome simbolico. Successivamente l'*ASM* provvede a richiedere all'*Ontology* la validazione della descrizione del servizio e, se questa va a buon fine, esegue la registrazione presso l'*Application Services Directory*.

IV.3.3 AUTENTICAZIONE DI UN UTENTE

L'autenticazione di un utente prevede ovviamente che quest'ultimo si sia precedentemente registrato presso l'amministratore del sistema. Nel momento in cui l'utente ha effettuato la connessione al sistema ,attraverso la home page di *UbiSystem*,egli non risulta ancora autenticato e la sessione associata sraà una sessione di "*guest*". Successivamente l'utente registrato potrà immettere i suoi dati di login e password ,accedendo quindi ai propri servizi privati. All'atto della autenticazione viene creata la specifica lista dei servizi disponibili. Di questo compito si occupa l'*Ontology Service*, e nel farlo terrà conto dei diritti dell'utente, delle caratteristiche del dispositivo con cui sta interagendo e della sua locazione.

Oltre a ciò,l'autenticazione richiede di eseguire delle operazioni aggiuntive rispetto alla progettazione precedente.In particolare si deve controllare se l'utente che ha richiesto l'autenticazione non abbia lasciato in precedenza una sessione in *sleeping*, nel qual caso la sessione diviene nuovamente attiva. In caso contrario si provvederà ad aprire una sessione nuova.

Di seguito il sequence relativo (figura IV.8) :

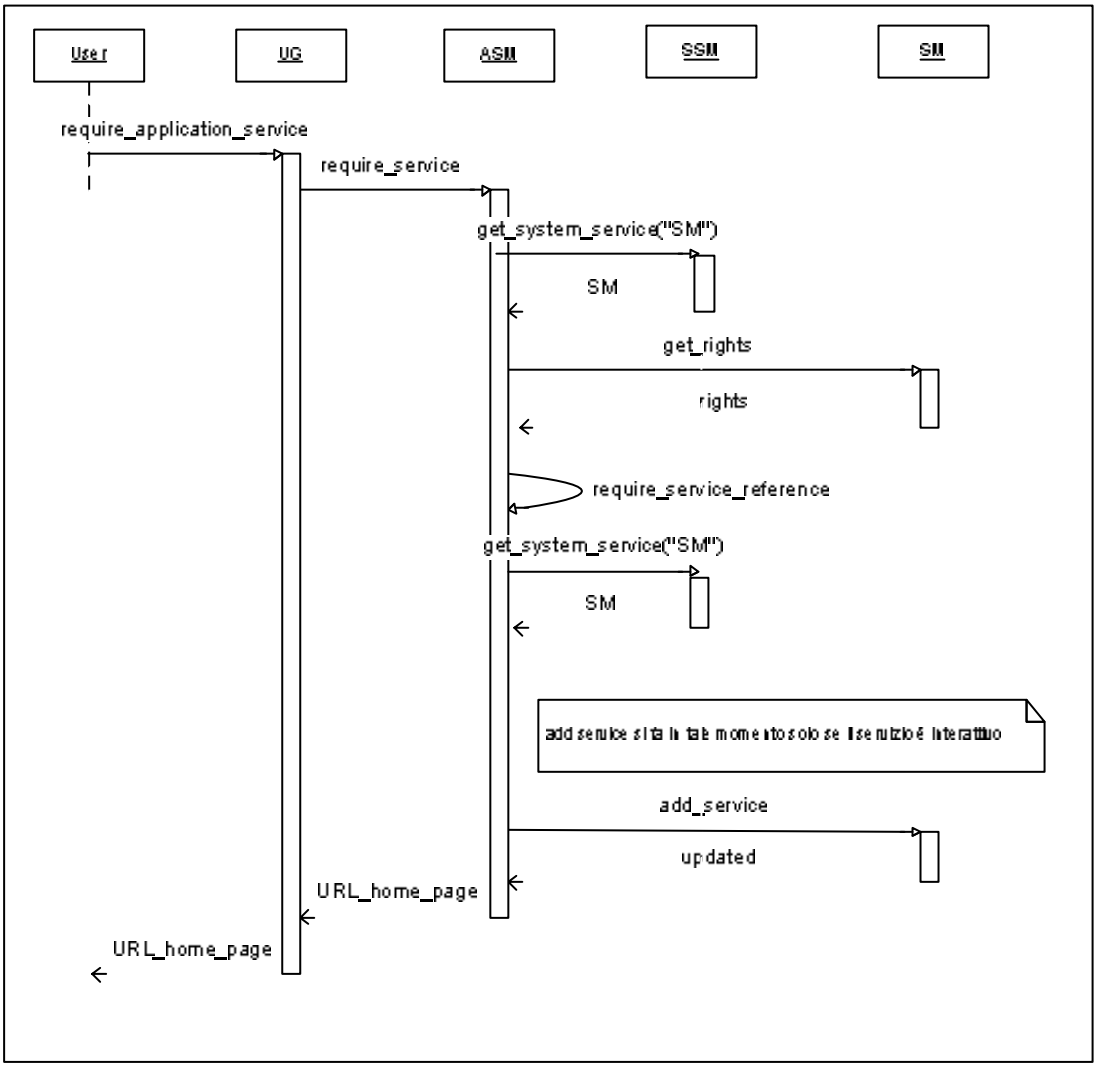


IV.3.4 RICHIESTA DI UN SERVIZIO APPLICATIVO BATCH

Per utilizzare un servizio applicativo è necessario passare attraverso il manager relativo, dato che solo esso conosce il riferimento e lo stato di tali servizi.

La richiesta di un servizio applicativo batch risulta leggermente differente rispetto all'attuale richiesta di un servizio interattivo,in quanto per le applicazioni batch si è pensato di aggiungere il servizio tra quelli in uso dall'utente soltanto nel momento in cui il servizio stesso inizia la sua esecuzione.

Le operazioni da compiere risultano quindi (figura IV.9) :



IV.3.5 UTILIZZO DI UN SERVIZIO APPLICATIVO BATCH

Dopo aver ottenuto l'home page del servizio richiesto l'utente è pronto ad utilizzarlo :deve quindi eseguire una serie di operazioni preliminari richieste. Il servizio batch infatti,per sua natura, inizia ed esegue le proprie istruzioni ,senza interazione con l'utente durante la sua esecuzione.Inoltre tipicamente servizi di questo tipo elaborano dei file ottenuti in ingresso,per produrre risultati su altri file di output.

Per questo c'è bisogno che prima di avviare il servizio vero e proprio ,l'utente prepari i dati di input che vuole siano elaborati e che avrà posto in uno o più file.

Oltre a questo ,come si è accennato precedentemente ,tra il servizio e il *Session Manager* viene istituito un canale asincrono,che servirà nel momento in cui il servizio termina, per notificare tale evento. Il canale verrà ottenuto facendone richiesta al componente *Asynchronous Communication Service*.

Il produttore di nuovi eventi sul canale sarà lo stesso servizio batch,mentre il consumatore sarà il *Session Manager* (figura IV.10).

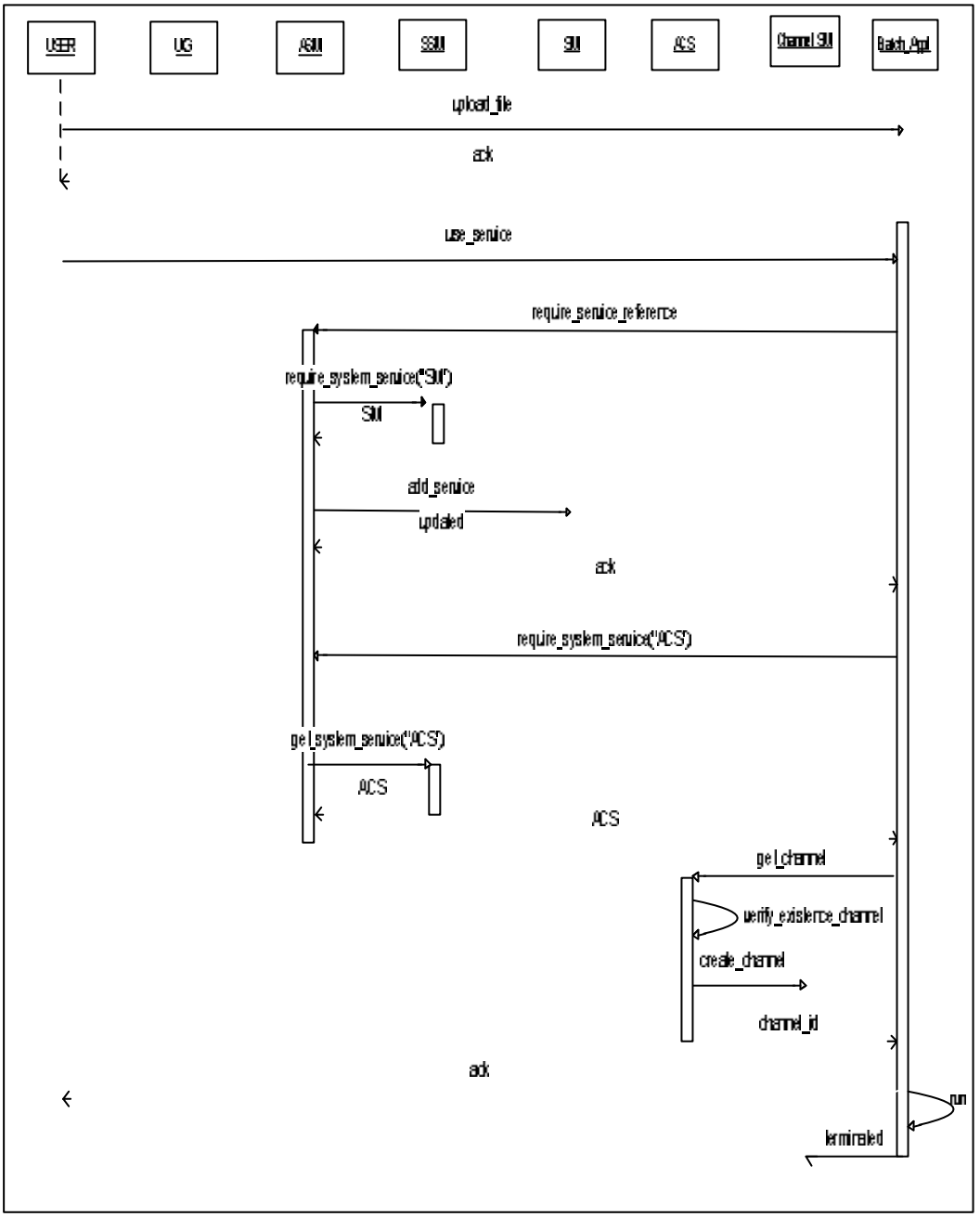


Fig. IV.10 Utilizzo di un servizio applicativo batch

IV.3.6 TERMINAZIONE DEL SERVIZIO BATCH

Quando il servizio batch precedentemente avviato termina deve produrre un nuovo evento sul canale,per la notifica all'utente. Il *Session Manager* sarà il consumatore dell'evento prodotto e in seguito a questo :

- Comunicherà col *Messenger Service* per consentire la creazione di un nuovo messaggio per l'utente.
- Provvederà ad aggiornare la sessione dell'utente(sia essa attiva oppure *sleeping*) indicando il servizio terminato come non più attivo.

Si riporta di seguito il sequence relativo (figura IV.11) :

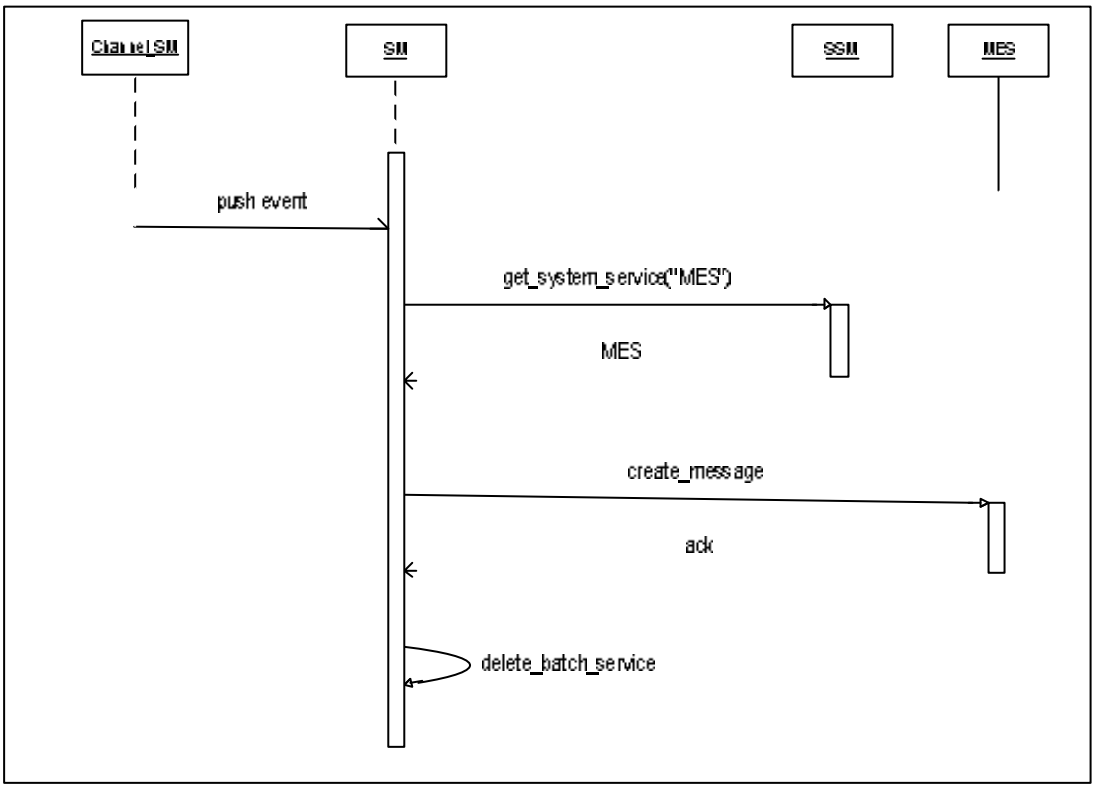


Fig. IV.11 Terminazione del servizio batch

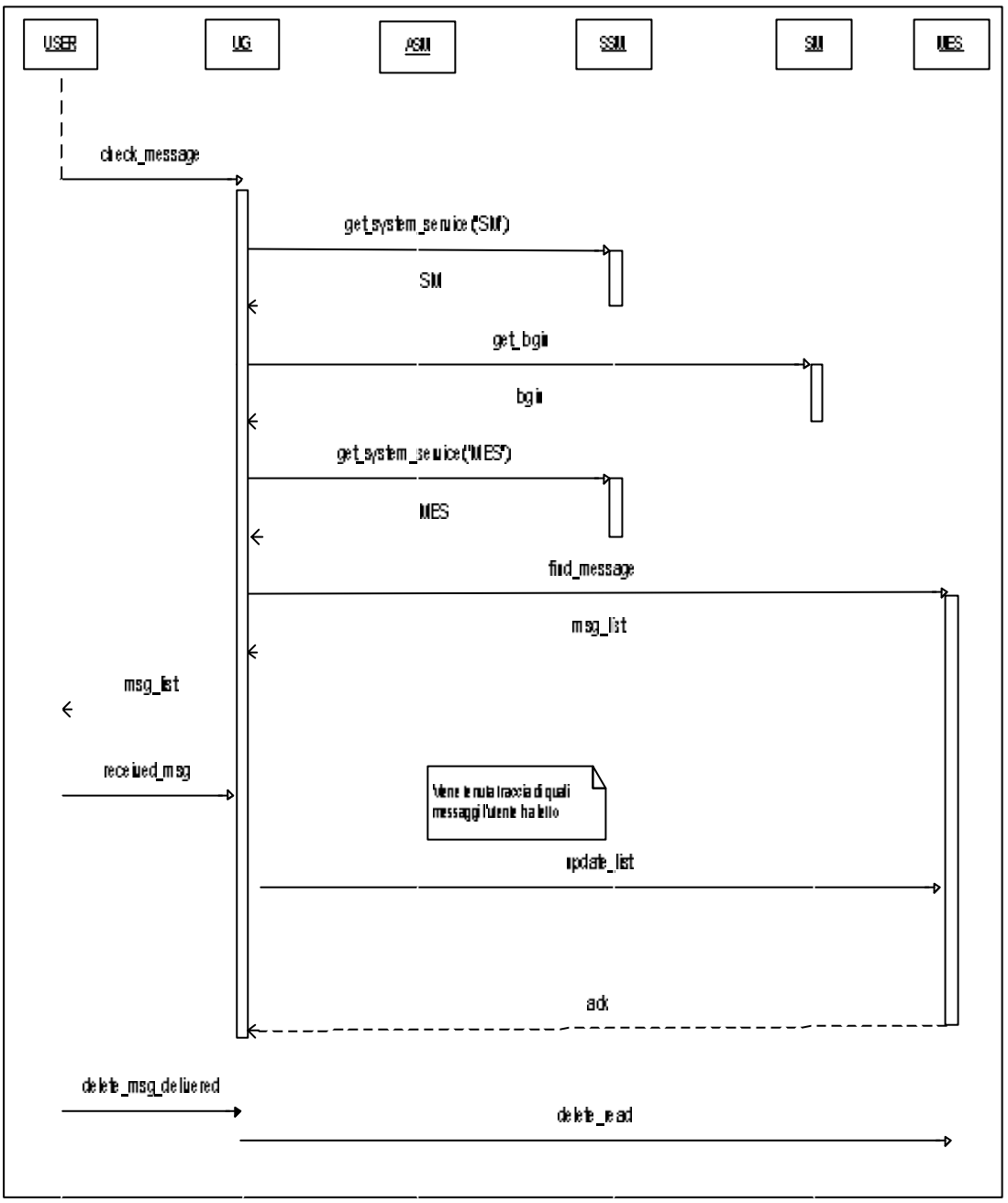
IV.3.7 LETTURA DEI MESSAGGI DA PARTE DELL'UTENTE REGISTRATO

Quando un utente si autentica dopo aver lasciato dei servizi in esecuzione,o in ogni caso al momento della terminazione dei servizi stessi,di tale avvenimento viene informato l'utente mediante un nuovo messaggio,.fornito dal componente di messaggistica. La presenza di nuovi messaggi viene notificata attraverso la pagina dei servizi privati dell'utente stesso,che può così proseguire con la lettura degli stessi.

In ciascun messaggio sono inserite tutte le informazioni necessarie affinché l'utente possa prelevare il risultato dell'elaborazione batch avviata in un precedente momento. In particolare uno stesso utente può richiedere contemporaneamente diverse esecuzioni dello stesso servizio batch per cui nel messaggio deve essere previsto anche un identificativo che distingua le varie richieste fatte allo stesso servizio,magari con input diversi.

All'utente è infine consentito aggiornare la propria pagina dei messaggi scegliendo di indicare o meno i messaggi come "letti" e di cancellare o meno i messaggi di cui ha già preso visione. Nel caso in cui si scelga di non cancellarli subito essi verranno riproposti in seguito (evidenziati come vecchi messaggi),fino alla loro definitiva cancellazione.

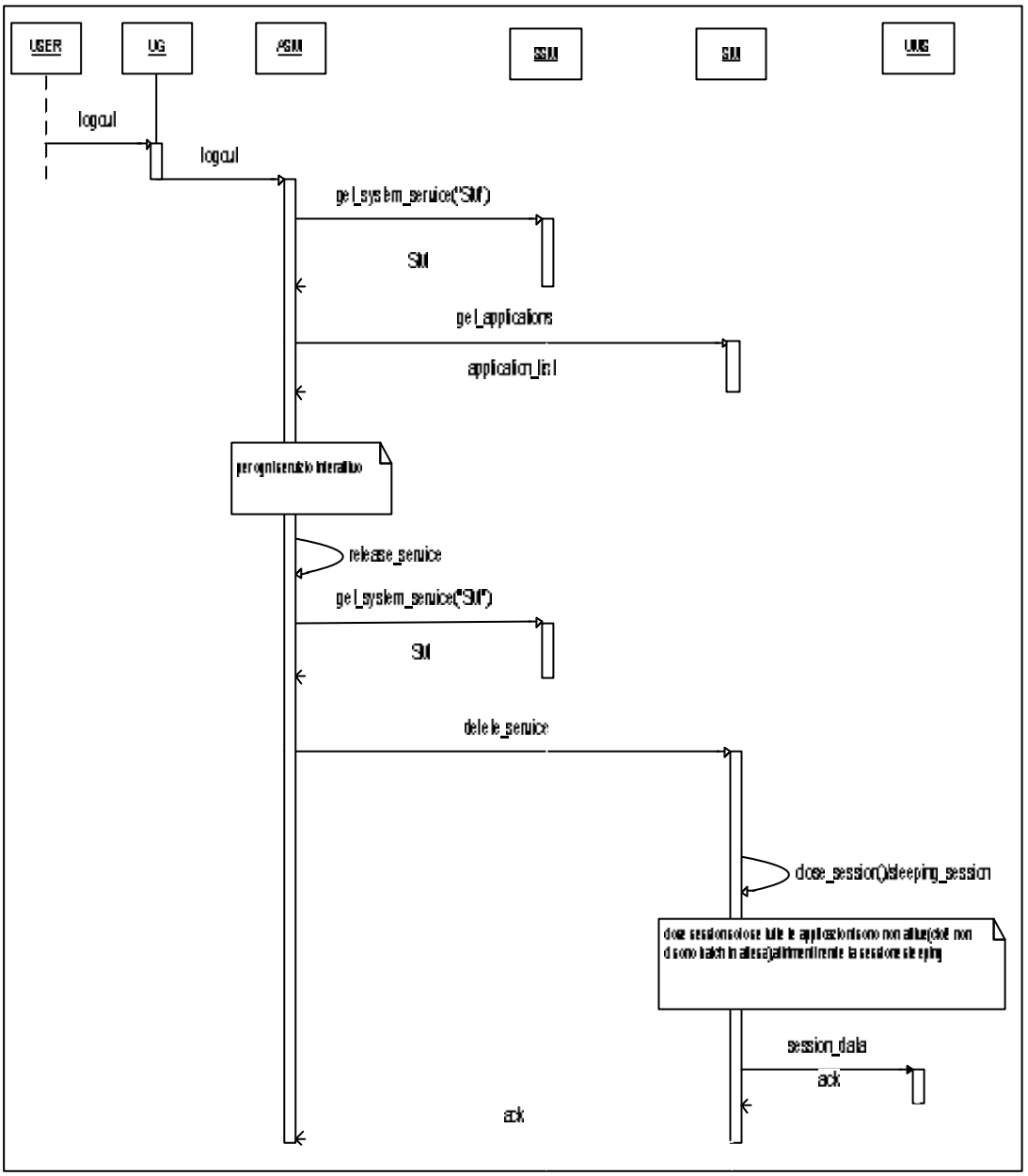
Le operazioni descritte saranno più chiare analizzando il sequence in figura IV.12.


USER
UG
ASM
SSM
SM
MES
check_message
get_system_service(SM)
SM
get_bgm
bgm
get_system_service("MES")
MES
find_message
msg_list
msg_list
received_msg
Viene tenuta traccia di quali messaggi l'utente ha letto
update_list
ack
delete_msg_delivered
delete_read

IV.3.8 LOGOUT

Con il logout l'utente comunica al sistema di voler abbandonare l'ambiente. In tal caso vengono chiusi tutti i servizi applicativi di tipo interattivo aperti dall'utente,mentre ,a differenza della progettazione precedente, quelli di tipo computazionale, non ancora terminati, vengono lasciati attivi. Di conseguenza devono essere deallocate tulle le risorse relative ai soli servizi interattivi in uso nella sessione dell'utente e devono essere chiamati i distruttori delle relative istanze, lasciando proseguire i servizi di tipo batch. Quindi nel caso in cui l'utente effettua un logout lasciando dei servizi batch in esecuzione la sessione non verrà chiusa,ma diventerà una sessione in *sleeping,* cioè sospesa in attesa che l'utente ritorni a prelevare i risultati delle proprie elaborazioni. Per tale motivo il logout è da considerarsi definitivo solo nel caso in cui l'utente non lascia applicazioni batch in esecuzione. E' necessario però far presente che la sessione diviene eventualmente sospesa solo se essa è l'unica sessione attiva in quel momento per l'utente.Se invece l'utente ha più sessioni attive,dovute a diversi device in suo possesso, e disattiva una sola delle sue sessioni, i servizi eventualmente sospesi per quest'ultima ,vengono accodati ad una delle sessioni ancora attive dello stesso utente In questo modo per il generico utente ci sarà sempre al più una sola sessione di *sleeping.*

Le azioni svolte all'interno del sistema si schematizzano come segue (figura IV.13) :


USER
UG
ASM
SSM
SM
UMS
logout
logout
get_system_service("SM")
SM
get_applications
application_list
per ogni sessione interattiva
release_service
get_system_service("SM")
SM
delete_service
close_session()/sleeping_session
close session=close tutte le applicazioni=no non attive(dot non
ci sono batch in attesa)altrimenti la sessione sle eping
session_data
ack
ack

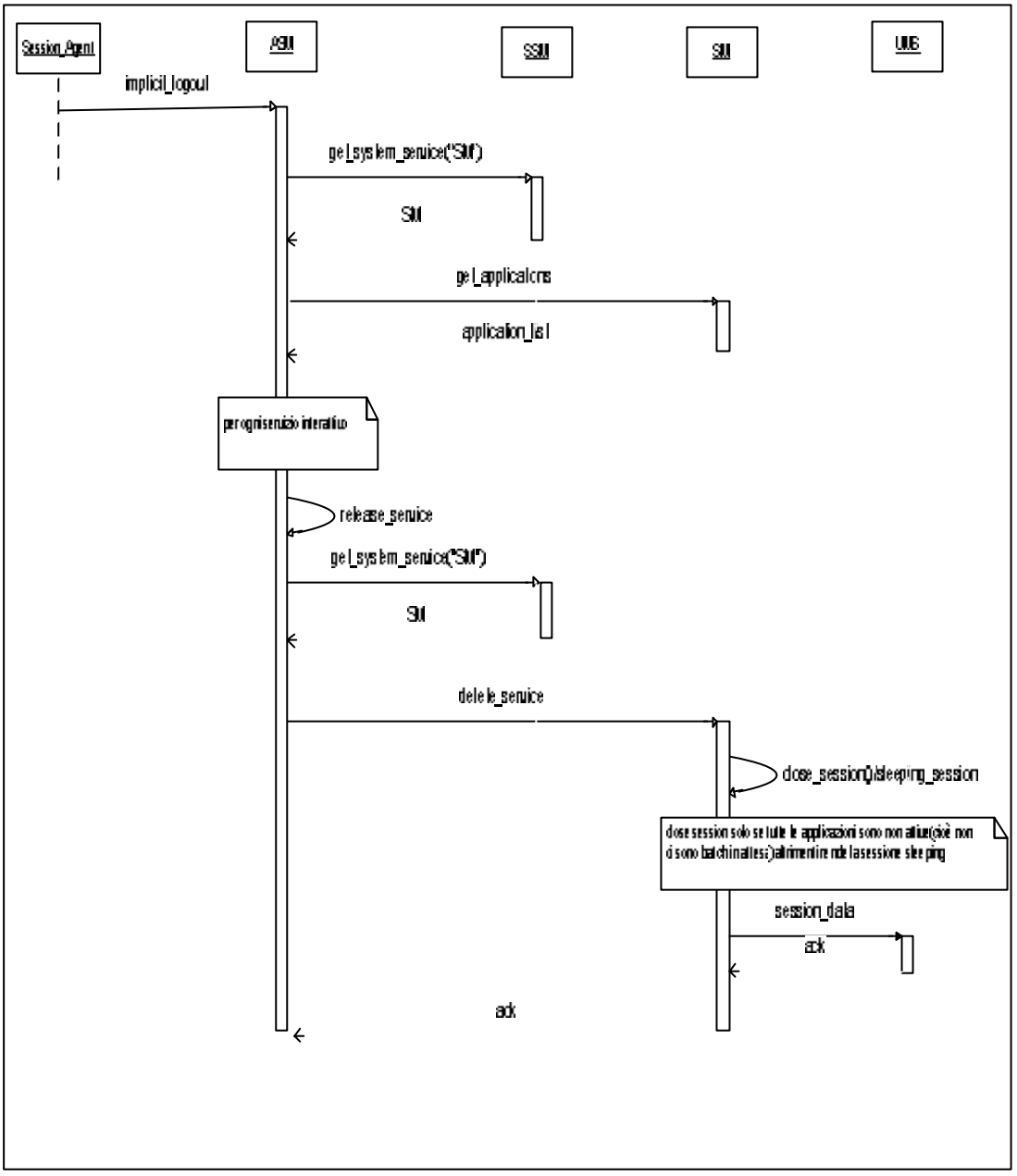
IV.3.9 ABBANDONO DELL'AMBIENTE SENZA LOGOUT

L'utente potrebbe abbandonare il sistema senza dare una comunicazione esplicita.

Allo scopo di individuare un evento di questo tipo, ad ogni sessione è associato un agente che, periodicamente, verifica il tempo trascorso dall'ultima interazione con il sistema. Dopo un certo periodo di inattività esso provvede a notificare la cosa all'*ASM*.

Si può verificare il caso in cui l'utente abbia richiesto un servizio di tipo batch, e quindi la sessione non andrà chiusa, secondo un comportamento analogo a quello del logout descritto nel paragrafo precedente.

Il diagramma seguente (figura IV.14), che sintetizza le operazioni svolte già analizzate nel diagramma relativo al logout,viene riportato per evidenziare che le operazioni sono svolte su iniziativa dell'agente associato alla sessione,che ha verificato un certo periodo di inattività.


Session_Agent
ASI
SSII
SII
UIS
implicit_logout
get_system_service("SII")
SII
get_applicaions
application_list
progressivo interattivo
release_service
get_system_service("SII")
SII
delete_service
close_session()/sleeping_session
close session solo se tutte le applicazioni sono non attive(cioè non ci sono batchinattesa)altrimenti rende la sessione sleeping
session_data
ack
ack

IV.3.10 MODIFICHE IMPLEMENTATIVE CONSEGUENTI

Dall'analisi dei Sequence Diagrams sorge la necessità di prevedere, per i componenti considerati, qualche ulteriore metodo, oltre a quelli individuati a partire dagli Use Case Diagrams. In più appare evidente la necessità di diverse modifiche implementative ad alcuni dei componenti già esistenti. In particolare con l'aiuto dei sequence precedenti sono stati riprogettati:

- il logout (esplicito e implicito) dell'utente dal sistema ,nell'implementazione relativa all'*Application Services Manager*
- l'autenticazione dell'utente nell'implementazione relativa al *Session Manager*
- la chiusura (momentanea o definitiva)della sessione nell'implementazione relativa al *Session Manager*

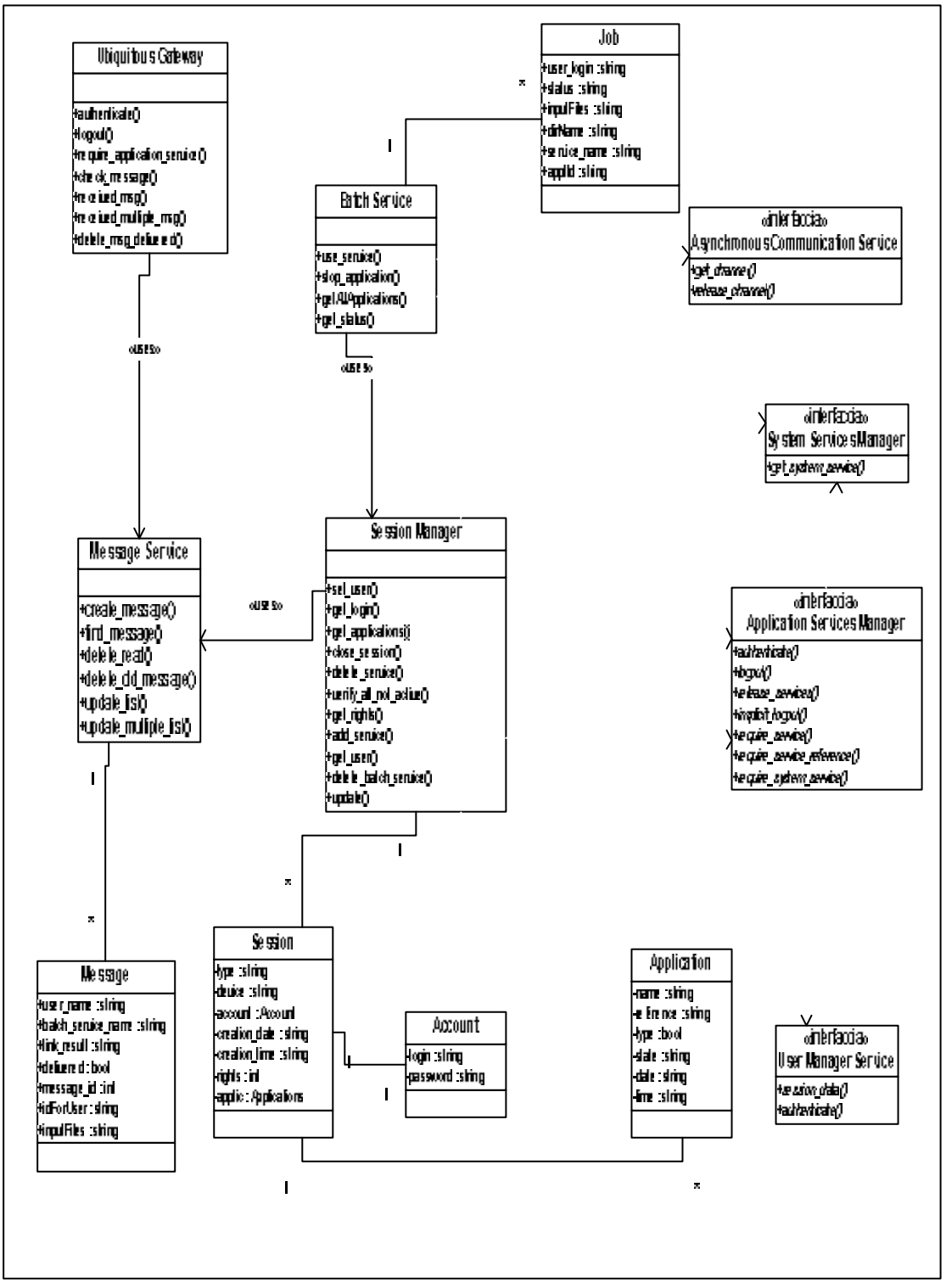
Tali modifiche saranno spiegate in maggiore dettaglio nel paragrafo riguardante l'implementazione.

IV.4 CLASS DIAGRAMS

Un *Class Diagram* mostra la struttura statica del modello, in altre parole, le cose che esistono (classi e tipi), la loro struttura interna e le relazioni occorrenti tra loro e con altri oggetti. Questi diagrammi, al contrario dei *Sequence Diagrams* analizzati nel precedente paragrafo, non riportano informazioni temporali.

Dopo aver analizzato le interazioni tra i vari moduli di *UbiSystem* e aver individuato i metodi che ognuno di essi deve esporre è possibile a questo punto mostrare il diagramma delle classi dei componenti coinvolti e implementati.

Si tratta di un diagramma a livello di astrazione ancora abbastanza alto,che aiuta a fare chiarezza sulle interazioni tra i moduli del sistema (figura IV.15).


Ubiquitous Gateway
+authenticate()
+logout()
+require_application_service()
+check_message()
+received_msg()
+received_multiple_msg()
+delete_msg_delivered()
Job
+user_login :string
+status :string
+inputFiles :string
+dirName :string
+service_name :string
+appId :string
Batch Service
+use_service()
+stop_application()
+getAllApplications()
+get_status()
«interfaccia»
AsynchronousCommunication Service
+get_channel()
+release_channel()
«use»
«use»
«interfaccia»
System ServicesManager
+get_system_service()
Message Service
+create_message()
+find_message()
+delete_read()
+delete_old_message()
+update_list()
+update_multiple_list()
«use»
Session Manager
+set_user()
+get_login()
+get_applications()
+close_session()
+delete_service()
+verify_all_not_active()
+get_rights()
+add_service()
+get_user()
+delete_batch_service()
+update()
«interfaccia»
Application Services Manager
+authenticate()
+logout()
+release_services()
+implicit_logout()
+require_service()
+require_service_reference()
+require_system_service()
Message
+user_name :string
+batch_service_name :string
+link_result :string
+delivered :bool
+message_id :int
+idForUser :string
+inputFiles :string
Session
-type :string
-device :string
-account :Account
-creation_date :string
-creation_time :string
-rights :int
-applic : Applications
Account
-login :string
-password :string
Application
-name :string
-reference :string
-type :bool
-state :string
-date :string
-time :string
«interfaccia»
User Manager Service
+session_data()
+authenticate()

IV.4.1 UBIQUITOUS GATEWAY

Questo componente rappresenta il punto di ingresso per gli utenti di *UbiSystem* e per l'amministratore.

A seconda di chi inoltra la richiesta il sistema reagisce in maniera diversa.

A questo livello sono state individuate delle strutture dati utili per la gestione del sistema che sono usate anche come parametri di ingresso-uscita (figura IV.16) :

- *Service*. Questa classe raggruppa le caratteristiche principali di un servizio di *UbiSystem* necessarie per far conoscere il servizio all'utente: nome, descrizione testuale e diritti d'accesso.
- *Message*. E' la classe che contiene le informazioni su ciascun messaggio tra cui login dell'utente,nome del servizio,collegamento alla pagina del risultato e identificativo del messaggio.

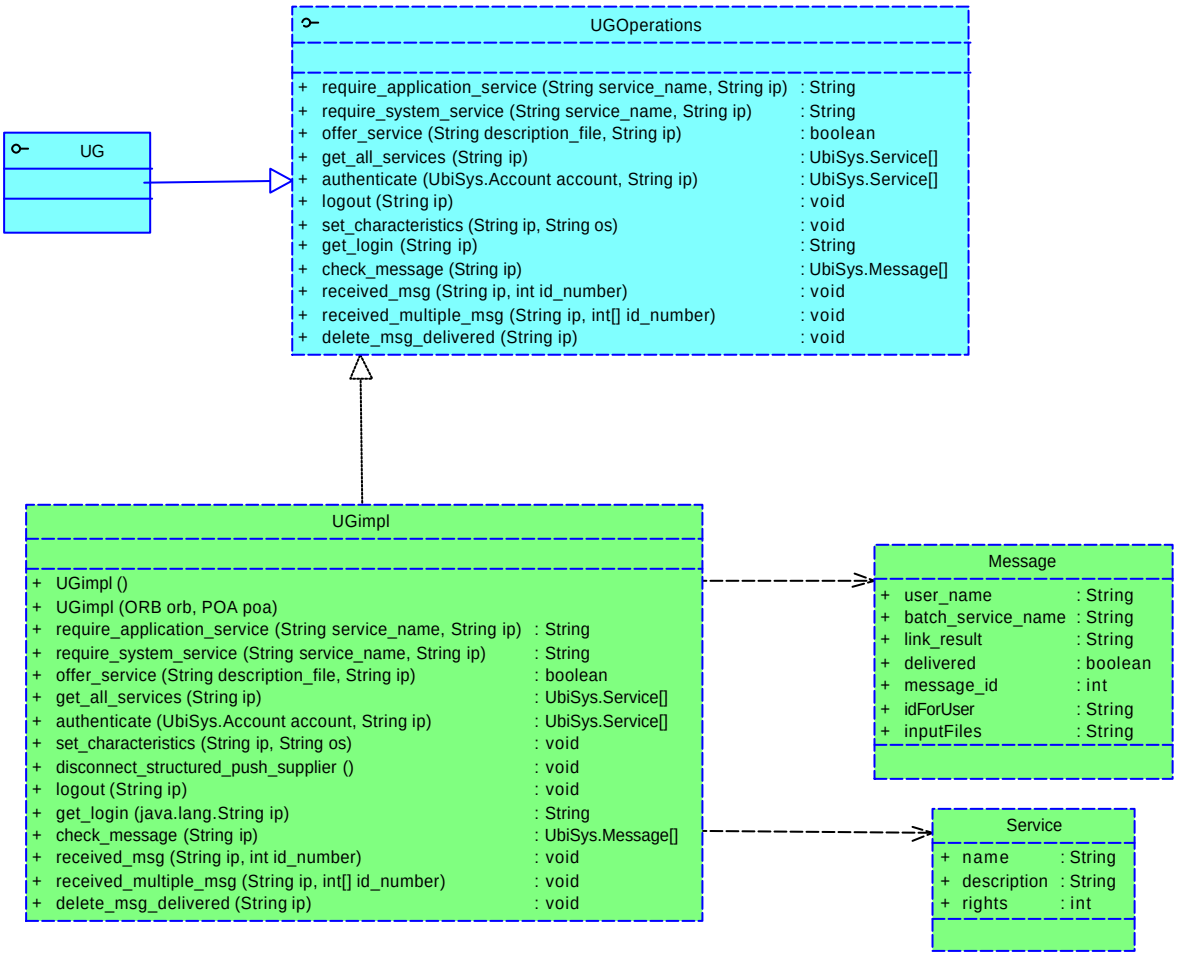


Fig. IV.36 Diagramma delle classi di Ubiquitous Gateway

Risulta utile ricordare la differenza tra un utente generico e l'amministratore, infatti quest'ultimo può interagire anche con i servizi di sistema e può monitorare i servizi applicativi. Per questo motivo sono presenti due interfacce; tale argomento però è già stato sviluppato nei lavori di tesi precedenti,ai quali rimandiamo per un approfondimento [50].

IV.4.2 SESSION MANAGER

Come per tutti gli altri componenti , anche per il manager delle sessioni è possibile individuare un'interfaccia, che espone le funzionalità da esso offerte,utilizzabili dai moduli con cui interagisce (figura IV.17) .

Si noti che la molteplicità dell'associazione tra la classe SMImpl e la classe SM_ConsumerThread indica che è anche possibile istanziare n canali,indicandone il nome. Infatti si può decidere di utilizzare un canale distinto per ciascun servizio batch(dando ad esempio ad ogni canale un nome che si riferisca in maniera inequivocabile al servizio).

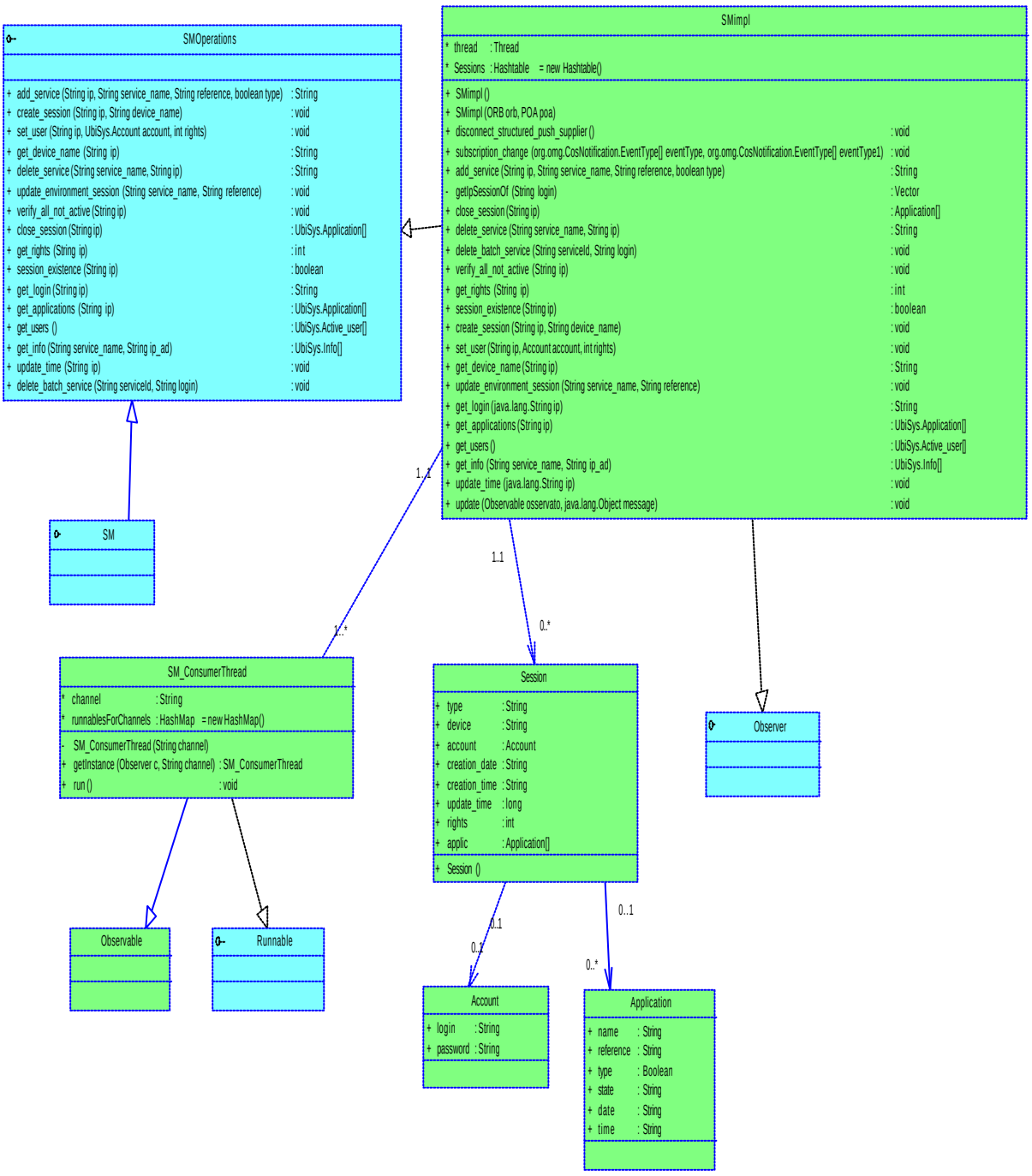


Fig. IV.47 Diagramma delle classi di Session Manager

IV.4.3 MESSAGE SERVICE

La struttura del *Message Service* è riportata nel diagramma seguente (figura IV.18).

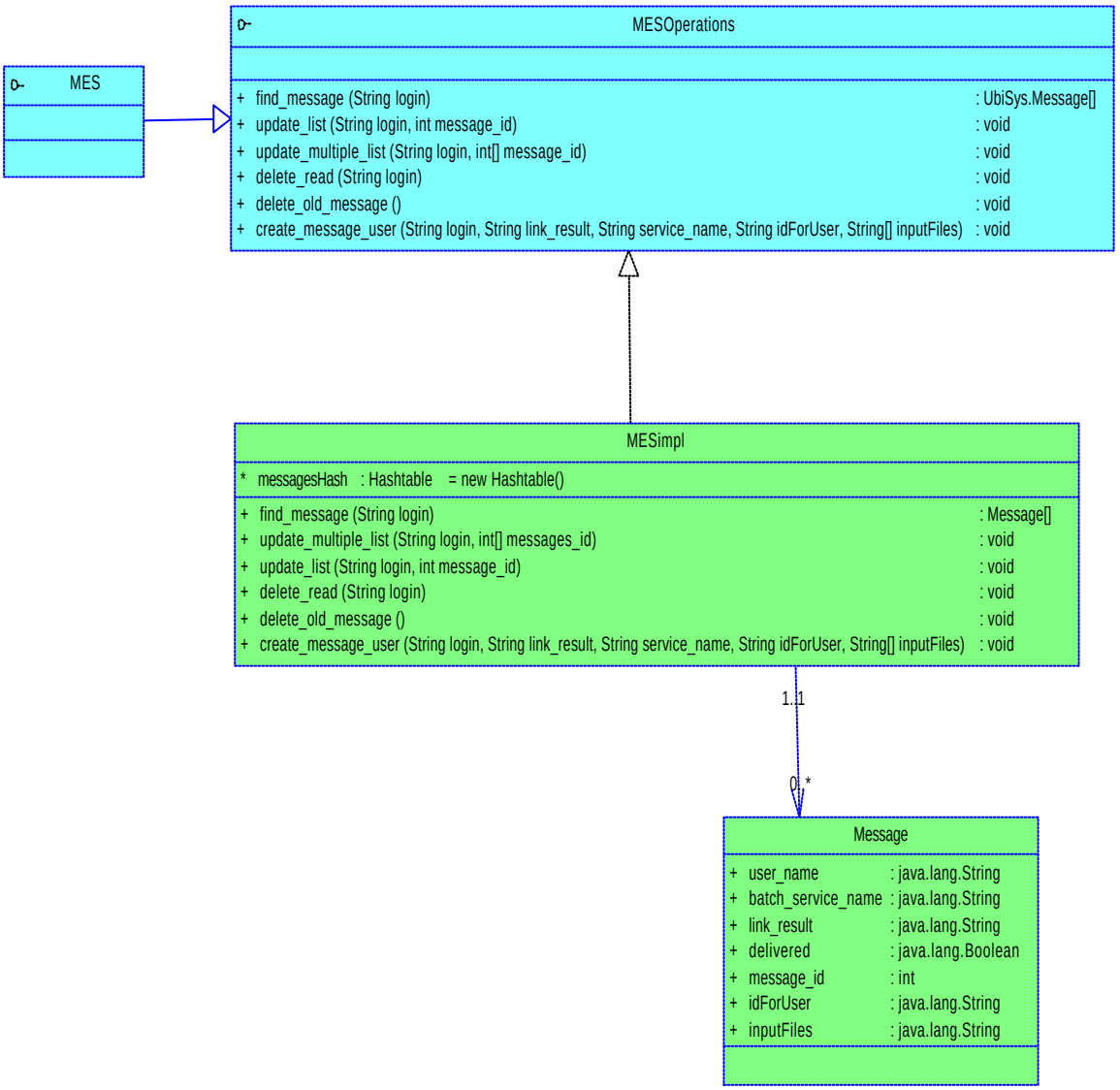


Fig. IV.58 Diagramma delle classi di Message Service

IV.5 DETTAGLI IMPLEMENTATIVI

Al termine della fase progettuale è possibile passare alla fase implementativa.

Tutti i componenti visti sono stati implementati come oggetti CORBA e solamente *Ubiquitous Gateway* è stato esposto come servizio web. Solo in questo modo, infatti, le funzionalità che espone potranno essere accessibili dall'esterno.

IV.5.1 IMPLEMENTAZIONE DI UN OGGETTO CORBA

La prima cosa da fare per realizzare un oggetto Corba scrivere la sua interfaccia. Questa sarà scritta in un apposito linguaggio per la definizione delle interfacce: *Interface Definition Language (IDL).*

IDL permette la definizione del nome dell'oggetto Corba, la dichiarazione dei metodi, la costruzione di nuovi tipi di dati, le modalità di passaggio di parametri e la definizione delle eccezioni.

Un compilatore IDL riceve la descrizione dell'interfaccia ed esegue una corrispondenza tra l'IDL ed il linguaggio in cui è realmente implementato l'oggetto.

La compilazione di una interfaccia produce almeno due file: lo *stub* (lato client) e lo *skeleton* (lato server). Lo stub del client rappresenta l'interfaccia attraverso cui un client può richiedere servizi a un server. Cioè contiene tutte le definizioni delle operazioni dell'interfaccia IDL dell'oggetto in forma di definizioni di metodo del linguaggio di programmazione utilizzato. Dal punto di vista del client, rappresenta la parte dell'oggetto CORBA (la più *vicina)* al client con cui esso interagisce. Il client cioè esegue di fatto una invocazione locale allo stub, che "impacchetta" (*marshalling*) i dati dell'invocazione in un messaggio che viene consegnato all'ORB e da questo inviato al server. Di conseguenza allo stub del client è associato a tempo di esecuzione un riferimento remoto all'oggetto (Object Reference), che è il mezzo utilizzato dal client per invocare operazioni sull'oggetto.

Lo skeleton è l'equivalente dello stub dal lato dell'implementazione dell'oggetto. Esso rappresenta il vero oggetto CORBA poiché fornisce lo scheletro dell'oggetto lasciando vuoto lo spazio riservato all'implementazione dei metodi che dovrà essere riempito da un

programmatore. Lo skeleton si differenzia dallo stub poiché coopera con un Object Adapter per le operazioni che riguardano l'attivazione dell'oggetto. Lo skeleton riceve la richiesta dall'Object Adapter, ne estrae i dati (*unmarsahlling*) e li passa all'implementazione dell'oggetto.

La forma in cui un oggetto si presenta al client è chiamata *Object Reference* (OR). Un OR è, nel senso più generale del termine, un puntatore all'oggetto, cioè un mezzo per poter accedere all'oggetto senza di fatto dover "utilizzare" l'oggetto vero e proprio. Un OR per un oggetto viene creato insieme all'oggetto stesso e continua a esistere lungo tutto il suo tempo di vita, durante il quale continuerà a riferirsi solamente a quell'oggetto. Cioè l'OR smette di essere valido solo quando l'oggetto viene permanentemente distrutto e da quel momento in poi non sarà più riutilizzabile in seguito per altri oggetti diversi. Un OR è quindi associato ad un solo oggetto, ma non è vero il viceversa, un oggetto cioè può avere diversi OR.

In CORBA è definito un formato standard per gli OR grazie al quale viene garantita l'interoperabilità tra ORB diversi. OR che segue questo formato è chiamato *Interoperable Object Reference* (IOR).

Affinché il client possa ottenere l'OR di un oggetto, prima questo deve essere stato creato e reso disponibile da un server. Questo può accadere in diversi modi:

- Restituendo l'OR come il risultato di un'operazione.
- Utilizzando un servizio CORBA, tipicamente il *Naming Service*
- Convertendo l'OR in una stringa che potrà essere salvata su un file system condiviso o inviata via rete. L'interfaccia dell'ORB fornisce le opportune operazioni per convertire OR a e da stringhe.

Per casi particolari, come ad esempio proprio il *Naming Service* (che è esso stesso un oggetto CORBA), è l'ORB che restituisce l'OR attraverso un'operazione della sua interfaccia.

Per rendere più chiaro il meccanismo si considera la realizzazione di un generico oggetto avente le stesse caratteristiche di quelli realmente implementati.

Il primo passo da fare è la scrittura dell'interfaccia IDL salvata nel file: *"Manager.idl"*. Questa è data in pasto al compilatore idl-java che genera tre gruppi di classi:

- Classi lato Client: *_ ManagerStub.java, Manager.java*

- Classi Lato Server: *ManagerPOA.java*, *ManagerPOATie.java*, *ManagerOperations.java*
- Classi di supporto: *ManagerHelper.java*, *ManagerHolder.java*

Successivamente sarà necessario scrivere le classi rappresentanti il server e l'implementazione (*Server.java e Impl.java*), dal lato server, e la classe client (*Client.java*) dal lato client.

Il diagramma delle classi dell'oggetto corba in esame risulta (figura IV.19) :

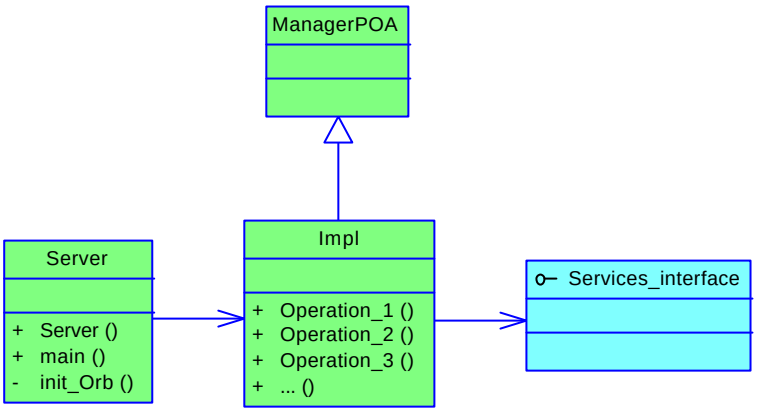


Fig. IV.69 Diagramma delle classi di un oggetto Corba

La classe *Impl.java* ha lo scopo di implementare tutte le operazioni definite nell'interfaccia IDL.

La classe *Server.java* costituisce il processo server e contiene le chiamate CORBA necessarie per attivare l'implementazione all'interno dell'ORB. La sequenza di operazioni svolte all'interno del server è questa:

- *Inizializzazione ORB.* Prima di ogni operazione CORBA è necessario inizializzare l'oggetto ORB attraverso l'operazione statica init(). In questo modo si ottiene anche il riferimento all'ORB necessario per il successivo utilizzo.
- *Creazione riferimento al POA*. Il riferimento iniziale al POA viene restituito dall'ORB stesso attraverso l'operazione resolve_initial_reference(). Questa operazione è in grado di restituire il riferimento di alcuni oggetti speciali che possono essere specificati attraverso il loro nome.
- *Attivazione oggetto nel POA*. Un'istanza dell'implementazione dell'oggetto viene associata dal POA all'oggetto stesso, di cui viene creato e restituito l'Object Reference. L'oggetto a questo punto si manifesta attraverso il suo

Object Reference, attraverso il quale i client possono effettuare invocazioni risalendo al servant che lo incarna.

- *Pubblicazione Object Reference*. L'OR dell'oggetto viene reso disponibile ai client registrandolo nel *Naming Service* o trasferendolo in forma di stringa (ad esempio mediante scrittura in un file condiviso o e-mail). La conversione dell'OR a e da stringa avviene attraverso opportune operazioni dell'ORB.
- *Attivazione ORB*. L'ORB viene esplicitamente attivato e posto in uno stato di ascolto delle richieste.

Per quanto riguarda gli oggetti implementati la pubblicazione dell'OR avviene presso il *Naming Service*.

IV.5.2 UBIQUITOUS GATEWAY

IV.5.2.1 UG COME OGGETTO CORBA

Si riporta di seguito l'interfaccia IDL dell'*Ubiquitous Gateway*.

```
module UbiSys{
      struct Account {
            string login;
            string password;
      };
      struct Service {
            string name;
            string description;
            long rights;
      };
      struct Message {
            string user_name;
            string batch_service_name;
            string link_result;
            boolean delivered;       //true= messaggio consegnato
            long message_id;
            string idForUser;
            files_input inputFiles;
```

```
};

//Definizione lista di servizi
typedef sequence <Service> Services;

//Definizione lista di messaggi
typedef sequence <Message> Messages;
typedef sequence <long> list_id;

module UbiquitousGateway{
      interface UG {
            //Eccezioni sollevate
            exception UGerror{};

string require_application_service(in string service_name,
in string ip) raises (UGerror);

string require_system_service(in string service_name,
      in string ip) raises (UGerror);

boolean offer_service(in string description_file,
      in string ip) raises (UGerror);

Services get_all_services(in string ip)
      raises (UGerror);

Services authenticate(in Account account, in string ip)
      raises (UGerror);

void logout(in string ip)
      raises (UGerror);

void set_characteristics(in string ip, in string os)
      raises (UGerror);

string get_login(in string ip)
      raises (UGerror);
```

```
      Messages check_message (in string ip)
            raises (UGerror);

      void received_msg (in string ip,in long id_number)
            raises (UGerror);

      void received_multiple_msg (in string ip,in list_id id_number)
            raises (UGerror);

      void  delete_msg_delivered (in string ip)
            raises (UGerror);

       };//end interface UG
    };//end module UbiquitousGateway
};//end module UbiSys
```

Le strutture dati definite sono:

- *Account*. Si tratta di una struttura dati atta a definire l'account di un utente. E' definita in questo componente in quanto il metodo *Autenticate()* prevede un parametro di ingresso di tipo *Account*.

- *Service*. Tale struttura dati definisce le caratteristiche di un servizio presente nell'ambiente. In particolare:

  - il campo *name* rappresenta il nome con cui il servizio è conosciuto nell'ambiente;

  - il campo *description* contiene una breve descrizione testuale riguardante il servizio;

  - il campo *rights* indica il minimo valore del livello di accesso che deve possedere l'utente per poter usufruire delle funzionalità del servizio.

- *Services*. Rappresenta una lista di servizi. In altri termini, un array di *Service*.

- *Message*. E' la struttura che definisce i campi di un generico messaggio da recapitare all'utente all'atto della terminazione del servizio applicativo computazionale. Più in dettaglio:

  - il campo *user_name* rappresenta la login con la quale l'utente è registrato in *UbiSystem*;

  - il campo *batch_service_name* è il nome con cui il servizio è conosciuto nell'ambiente;

  - il campo *link_result* è una stringa che contiene l'informazione sul link dal quale l'utente può prelevare il risultato della sua elaborazione;

  - il campo *delivered* serve a tener traccia della consegna del messaggio all'utente;

  - il campo *message_id* è un identificativo del messaggio e serve alla gestione delle operazioni da compiere su di esso(cancellazione,aggiornamento ,ecc…);

  - il campo *idForUser* identifica la particolare esecuzione del servizio,in quanto ,ripetiamo, un utente può richiedere diverse esecuzioni parallele dello stesso servizio ,con input diversi,e deve poter distinguere i vari job che ha avviato.Questo campo può essere usato sia insieme a quello successivo dei file di input,sia come sua alternativa,nel caso i file di input siano in numero eccessivo.Al momento questo campo è previsto,ma per far riconoscere all'utente le applicazioni in corso,abbiamo preferito mostrare i file di input con cui è stato avviato il servizio.

  - il campo *inputFiles* è un array di stringhe contenente l'elenco dei file di input immessi per eseguire il servizio.

- *Messages*. Rappresenta una lista di messaggi. In altri termini, un array di *Message*.

I metodi aggiuntivi implementati oltre a quelli esistenti (per i quali si rimanda ai lavori di tesi precedenti) sono:

- *Messages check_message (in string ip).* Tale metodo riceve in ingresso l'ip del dispositivo da cui proviene la richiesta e restituisce una lista di messaggi da consegnare all'utente.Esso,ricavata la login dell'utente, inoltra la richiesta al *Message Service* invocando l'opportuno metodo di questo componente (*find_message)* per trovare i messaggi dell'utente.Restituisce la lista dei messaggi.

- *void received_msg (in string ip,in long id_number).* Questo metodo riceve in ingresso l'ip del dispositivo da cui proviene la richiesta e l'identificativo del particolare messaggio. Esso inoltra la richiesta al servizio di messaggistica al fine di tener conto che il messaggio è stato letto dall'utente.

- *void received_multiple_msg (in string ip,in list_id id_number).* Questo metodo svolge azioni analoghe al precedente,ma è riferito a una lista di messaggi,per permettere appunto la gestione di una intera lista ,anziché di un solo messaggio alla volta.

- *void delete_msg_delivered (in string ip).* Tale metodo riceve in ingresso l'ip del dispositivo da cui proviene la richiesta ed ha come effetto quello di cancellare i messaggi che risultano letti. Esso non fa altro che invocare il metodo opportuno(*delete_read())* del *Message Service*.

Queste nuove funzionalità,come già anticipato sono state aggiunte al componente *Ubiquitous Gateway* esistente.Ricordiamo che tale modulo ha anche un altro compito fondamentale. Nel momento in cui un utente si connette ad *UbiSystem* è necessario che il sistema venga a conoscenza delle caratteristiche del dispositivo con cui egli interagisce, quindi bisogna fare in modo che il dispositivo si registri presso l'*Ontology Service*.

Nella realizzazione di *UbiSystem* si cerca di evitare l'installazione di software aggiuntivo per la fruizione dei servizi, in accordo con le scelte progettuali, quindi la registrazione non può essere a carico del dispositivo client.

Visto che il ruolo che *Ubiquitous Gateway* gioca in *UbiSystem* è quello di entry point, è suo compito costruire la descrizione del dispositivo a partire dalle informazioni prelevate dall'header della richiesta http giunta ad esso.

IV.5.2.2 UG COME WEB-SERVICE

Per la realizzazione del servizio web sono state utilizzate le librerie di Apache Axis [2] che mettono a disposizione una serie di toolkit che consentono la realizzazione di Web Services a partire da applicazioni Java.

Il primo passo da eseguire è la creazione di due classi Java che rappresentino rispettivamente l'interfaccia e l'implementazione del servizio: *UG.UG_W.java* e *UG.UG_W_impl.java.*

```
package UbiSys.UG_Wrapping.UG;

public interface UG_W {

  public java.lang.String require_application_service
      (java.lang.String service_name, java.lang.String ip);

  public java.lang.String require_system_service
      (java.lang.String service_name, java.lang.String ip);

  public boolean offer_service
      (java.lang.String description_file, java.lang.String ip);

  public UbiSys.Service[] get_all_services(java.lang.String ip);

  public UbiSys.Service[] autenticate
      (UbiSys.Account account, java.lang.String ip);

  public void logout(java.lang.String ip);

  public void set_characteristics
      (java.lang.String ip , java.lang.String os);

  public java.lang.String get_login(java.lang.String ip);

  public UbiSys.Message[] check_message (java.lang.String ip);
```

```
  public void received_msg (java.lang.String ip , int id_number);

  public void received_multiple_msg (java.lang.String ip,
       int[] id_number);

  public void delete_msg_delivered (java.lang.String ip);

}
```

La classe UG_W_impl sarà client dell'oggetto Corba descritto nel precedente paragrafo.

```
package UbiSys.UG_Wrapping.UG;

import java.io.*;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import UbiSys.UbiquitousGateway.*;

public class UG_W_impl {

    UG UG_Reference;
    /** Creates a new instance of UG_W_impl */
    public UG_W_impl() {
      //Questa classe sarà "client" di UG_Server
      try{
        // initialize the ORB.
        ORB orb = ORB.init( (String[])null, null );

        NamingContextExt nc = NamingContextExtHelper.narrow(
           orb.resolve_initial_references("NameService"));

        //legge il riferimento di UG dal naming service
        UG_Reference=UGHelper.narrow(nc.resolve(nc.to_name("UG")));

      }catch (Exception e){
        e.printStackTrace();
```

```
    }
  }
  public String require_application_service
   (String service_name, String ip)throws java.rmi.RemoteException
  {
    String result;
    try{
      result = UG_Reference.require_application_service
          (service_name, ip);
    }catch(Exception e){
      throw new java.rmi.RemoteException();
    }
    return result;
  }

  public String require_system_service(String service_name, String
    ip)throws java.rmi.RemoteException
  {
    String result;
    try{
      result = UG_Reference.require_system_service
          (service_name, ip);
    }catch(Exception e){
        throw new java.rmi.RemoteException();
    }
    return result;
  }

  public boolean offer_service(String description_file, String
    ip)throws java.rmi.RemoteException
  {
    boolean result;
    try{
        result = UG_Reference.offer_service(description_file, ip);
    }catch(Exception e){
        throw new java.rmi.RemoteException();
    }
    return result;
```

```
    }

    public UbiSys.Service[] get_all_services(String ip)
      throws java.rmi.RemoteException
    {
      UbiSys.Service[] result;
      try{
          result = UG_Reference.get_all_services(ip);
      }catch(Exception e){
          throw new java.rmi.RemoteException();
      }
      return result;
    }

    public UbiSys.Service[] authenticate(UbiSys.Account account,
      String ip)throws java.rmi.RemoteException
    {
      UbiSys.Service[] result;
      try{
          result = UG_Reference.authenticate(account, ip);
      }catch(Exception e){
          throw new java.rmi.RemoteException();
      }
      return result;
    }

    public void logout(String ip)throws java.rmi.RemoteException{
        try{
            UG_Reference.logout(ip);
        }catch(Exception e){
            throw new java.rmi.RemoteException();
        }
    }

    public void set_characteristics(java.lang.String ip,
      java.lang.String os)throws java.rmi.RemoteException
    {
      try{
```

```
        UG_Reference.set_characteristics(ip, os);
      }catch(Exception e){
        throw new java.rmi.RemoteException();
      }
    }

    public String get_login(java.lang.String ip)
      throws java.rmi.RemoteException
    {
      String login = null;
      try{
        login = UG_Reference.get_login(ip);
      }catch(Exception e){
        throw new java.rmi.RemoteException();
      }
      return login;
    }

     public UbiSys.Message[] check_message (String ip)
      throws java.rmi.RemoteException {
        UbiSys.Message[] result;
        try{
            result = UG_Reference.check_message(ip);
        }catch(Exception e){
            throw new java.rmi.RemoteException();
        }
        return result;
    }

     public void received_msg (java.lang.String ip , int id_number)
       throws java.rmi.RemoteException {
         try{
            UG_Reference.received_msg (ip,id_number);
        }catch(Exception e){
            throw new java.rmi.RemoteException();
        }
    }
```

```
     public void received_multiple_msg (java.lang.String ip, int[]
       id_number)  throws java.rmi.RemoteException {
         try{
            UG_Reference.received_multiple_msg (ip,id_number);
        }catch(Exception e){
            throw new java.rmi.RemoteException();
        }
    }

     public void delete_msg_delivered (java.lang.String ip)
      throws java.rmi.RemoteException {
         try{
            UG_Reference.delete_msg_delivered (ip);
        }catch(Exception e){
            throw new java.rmi.RemoteException();
        }
    }
}
```

Una volta definite queste due classi viene generata l'interfaccia WSDL del servizio Web, mediante il tool Java2WSDL di axis:.

I parametri necessari sono:

- il nome del file di output: *ugw.wsdl*;
- l'URL del Web Service: *http://localhost:80/axis/services/UG*;
- il target namespace per la WSDL: *urn: UG*;
- il mapping fra il package java ed il namespace: *UG = urn: UG*
- l'interfaccia java del servizio: *UbiSys.UG_Wrapping.UG.UG_W*.

Il comando completo da eseguire risulta:

```
java org.apache.axis.wsdl.Java2WSDL
  -o ugw.wsdl
  -l"http://localhost:80/axis/services/UG"
  -n urn:UG
  -p"UG" urn:UG
  UbiSys.UG_Wrapping.UG.UG_W
```

Una volta eseguito il programma verrà generato il file *ugw.wsdl*.

Il prossimo passo è quello di generare, a partire dalla WSDL del servizio, il codice necessario per effettuarne il deploy. Questa volta verrà adoperato il tool WSDL2Java.

I parametri necessari sono:

- la directory base di output: .;
- lo scope del servizio: *Session, Application, Request*;
- la generazione anche del codice server-side: *-s*;
- il package dove andrà il codice generato: *UbiSys.UG_Wrapping.UG.ws*;
- il file dove reperire la wsdl: *ugw.wsdl* (ma potrebbe esse anche una URL).

Il comando completo risulta allora:

```
java org.apache.axis.wsdl.WSDL2Java
  -o .
  -d Session
  -s
  -p UbiSys.UG_Wrapping.UG.ws ugw.wsdl
```

Nel package *ws* saranno generati i file:

- *UGSoapBindingStub*: client-side stub;
- *UG_W*: l'interfaccia remota;
- *UG_WService*: interfaccia di servizio per il Web Service;
- *UG_WServiceLocator*: implementa *UG_WService*. E' una Helper Factory che fornisce l'accesso al servizio;
- *Account*: classe che ha lo scopo di fare il wrapping dell'omonima classe utilizzata dall'oggetto corba come tipo per parametri di in-out*;*
- *Service*: classe che ha lo scopo di fare il wrapping dell'omonima classe utilizzata dall'oggetto corba come tipo per parametri di in-out;
- *Message:* classe che ha lo scopo di fare il wrapping dell'omonima classe utilizzata dall'oggetto corba come tipo per parametri di in-out;
- *UGSoapBindingImpl*: la classe il cui codice implementa il Web Service. Sarà necessario apportare delle modifiche per fare in modo che i suoi metodi chiamino i corrispondenti della classe *UG_W_impl*. Inoltre si dovrà prevedere il mapping relativo alle classi *Account* e *Service*.
- *deploy.wsdd*: file contenente le informazioni necessarie per effettuare il deploy del Web Service sul sistema Axis;
- *undeploy.wsdd*: file necessario per effettuare l'undeploy del servizio da Axis.

Il passo successivo da eseguire è quello di generare degli archivi di file che saranno necessari affinché il servizio web funzioni in maniera corretta.

In particolare gli archivi da generare sono:

- *UG.jar*: contiene tutte le classi interne ai package *UG* e *UG.ws*, cioè quelle relative al Web Service;

- *UbiSys.jar*: contiene il server Corba invocato dal Web Service. In realtà contiene le classi relative all'intero sistema Corba.
- *sGn.jar*: contiene il servizio Corba *Ontology Service* invocato per la registrazione del device.

Tali file devono essere copiati nella cartella *webapps\axis\WEB-INF\lib* situata all'interno di *Tomcat*.

I comandi usati sono:

```
jar cvf UG.jar UbiSys/UG_Wrapping/UG/UG_W.class
      UbiSys/UG_Wrapping/UG/UG_W_impl.class
      UbiSys/UG_Wrapping/UG/ws/*.class
jar cvf UbiSys.jar UbiSys\*.*
jar cvf sGn.jar sGn\*.*
```

A questo punto è possibile procedere con la pubblicazione del servizio (*deploy*) utilizzando il comando:

```
org.apache.axis.client.AdminClient
  -p 80
  UbiSys/UG_Wrapping/UG/ws/deploy.wsdd
```

Quando viene eseguito quest'ultimo comando, *Tomcat* dovrà essere in esecuzione ed in ascolto sulla porta 80 (-p 80).

L'utente finale interagisce con Ubiquitous Gateway e, quindi, con l'intero sistema, mediante pagine *jsp*. Affinché queste possano utilizzare il servizio web è necessario copiare il file *UG.jar* anche nella cartella *WEB-INF\lib* situata all'interno della directory in cui sono locate le pagine in questione.

IV.5.3 SESSION MANAGER

Bisogna tener presente che alla versione preesistente di questo componente è stato aggiunto un solo metodo pubblico,per quanto riguarda l'interfaccia;tuttavia si riporta di seguito l'interfaccia completa,per facilitare il lettore e permettere un suo migliore orientamento nel comprendere i dettagli implementativi di questo componente, oggetto della presente tesi. E' in tale componente infatti che si trovano gli spunti più interessanti della gestione dei servizi computazionali in *UbiSystem*.

```
module UbiSys{

        struct Account {
                string login;
                string password;
        };

        struct Application {
            string name;
            string reference;
            string state;
            boolean type; //true = interactive
            string date;
            string time;
        };

        struct Active_user{
            string ip;
            string log;
            string device;
            string location;
        };

        struct Info{
            string ip;
            string log;
            string date;
            string time;
        };
```

```
typedef sequence<Application> Applications;
typedef sequence<Active_user> Active_users;
typedef sequence<Info> Infos;
typedef sequence<string> Utenti;

module SessionManager{
interface SM {
            exception SessionNotUpdated{};
            exception SessionNotFound{};
            exception NobodyFound{};
            exception InfoNotAvailable{};


string add_service(in string ip, in string service_name, in
   string reference, in boolean type)
     raises (SessionNotUpdated);

void create_session(in string ip,in string device_name)
     raises (SessionNotUpdated);

void set_user(in string ip, in Account account, in long
   rights)
     raises (SessionNotUpdated);

string get_device_name(in string ip);

string delete_service(in string service_name, in string ip)
     raises (SessionNotUpdated);

void update_environment_session(in string service_name, in
   string reference)
     raises (SessionNotUpdated);

void verify_all_not_active(in string ip)
     raises (SessionNotUpdated);

Applications close_session(in string ip)
```

```
            raises (SessionNotUpdated);

        long get_rights (in string ip)
            raises (SessionNotFound);

        boolean session_existence(in string ip);

        string get_login(in string ip)
            raises (SessionNotFound);

        Applications get_applications(in string ip)
            raises (SessionNotFound);

        Active_users get_users()
            raises (NobodyFound);

        Infos get_info(in string service_name, in string ip_ad)
            raises (InfoNotAvailable);

        void update_time(in string ip)
            raises (SessionNotUpdated);

        void delete_batch_service (in string service_name,in string
          login)
            raises (SessionNotUpdated);

      };//End interface SM
   };//End module SessionManager
};//End module UbiSys
```

Le strutture dati definite sono:

- *Account*. Si tratta di una struttura dati atta a definire l'account di un utente.
- *Application.* Tale struttura dati definisce le caratteristiche di un servizio applicativo utilizzato dall'utente. In particolare:
  - il campo *name* rappresenta il nome con cui il servizio è conosciuto nell'ambiente;

- il campo *reference* rappresenta il riferimento al servizio;
- il campo *state* rappresenta lo stato del servizio;
- il campo *type* indica il tipo di servizio;
- il campo *date* indica il giorno in cui è stato richiesto il sevizio;
- il campo *time* indica l'ora in cui è stato richiesto il sevizio.

✍ *Active_user*. Tale struttura dati definisce le informazioni dell'utente presente nel sistema. In particolare:

- il campo *ip* rappresenta l'indirizzo ip del device con cui l'utente si è connesso al sistema;
- il campo *log* contiene la login dell'utente;
- il campo *device* rappresenta il tipo di device;
- il campo *location* indica la locazione dell'utente.

✍ *Info*. Tale struttura dati definisce le informazioni dell'utente presente nel sistema. In particolare:

- il campo *ip* rappresenta l'indirizzo ip del device con cui l'utente si è connesso al sistema;
- il campo *log* contiene la login dell'utente;
- il campo *date* indica il giorno in cui è stato utilizzato il sevizio;
- il campo *time* indica l'ora in cui è stato richiesto il sevizio.

✍ *Applications*. Rappresenta una lista di servizi utilizzati dall'utente. In altri termini, un array di *Application*.

✍ *Active_users*. Rappresenta la lista degli utenti correntemente presenti nell'ambiente. In altri termini, un array di *Active_user;*

✍ *Infos*. Rappresenta la lista degli utenti che stanno utilizzando un dato servizio. In altri termini, un array di *Info;*

Per la descrizione dei metodi già implementati in precedenza si rimanda ai lavori di tesi svolti precedentemente [50] [51] .

In tale contesto provvederemo invece ad analizzare in dettaglio quali modifiche è stato necessario apportare per gestire la sessione utente.Come abbiamo detto infatti si deve prevedere la presenza di servizi computazionali, che non terminano necessariamente prima che l'utente effettui il logout per uscire dal sistema. Da questo punto di vista la nostra gestione ha previsto le seguenti modifiche:

- *string add_service (in string ip, in string service_name ,in string reference, in boolean type).* Tale metodo riceve in ingresso il nome del servizio richiesto, l'ip del dispositivo con cui l'utente si è connesso al sistema,il riferimento al servizio e il tipo del servizio (*true* per interattivo, *false* per batch).Questo metodo viene invocato nel momento in cui un utente richiede un servizio,e provvede ad aggiornare la sessione relativa.E' da notare che ,per quanto riguarda le applicazioni batch, questo metodo è coinvolto solo all'atto dell'uso effettivo del servizio stesso e non all'atto della semplice richiesta della home page del servizio.Inoltre facciamo notare espressamente che la versione precedente di tale metodo prevedeva una uscita *void*. Nella nuova gestione invece viene restituita una stringa che costituisce un identificativo univoco per il servizio aggiunto.

  Vale la pena soffermarci un po' più a lungo su questo aspetto per capirne le motivazioni. Prima delle modifiche apportate con il presente lavoro,l'aggiunta nella sessione di un servizio utilizzato da un utente prevedeva un controllo che verificasse se il servizio era stato già utilizzato o meno in quella stessa sessione. In caso affermativo il servizio non veniva aggiunto ,ma ne veniva fatto solo un aggiornamento circa la data e l'orario di utilizzo. Questa ipotesi diviene inaccettabile soprattutto per quanto riguarda i servizi batch di cui ci stiamo occupando;infatti sarebbe una logica scorretta in primo luogo perché ,ai fini di una eventuale futura tariffazione del servizio ,questo risulterebbe sempre utilizzato al più una volta soltanto,mentre noi prevediamo che un utente possa effettuare contemporaneamente richiesta di più elaborazioni alla stessa

applicazione,magari con input differenti.In secondo luogo,ed ancora per la possibilità di più richieste contemporanee da parte di uno stesso utente,è necessario distinguere le varie esecuzioni parallele nell'ambito della stessa sessione. Questi motivi giustificano la restituzione da parte di questo metodo, di un identificativo per il servizio aggiunto, che tra l'altro servirà anche alla stessa applicazione batch per la gestione di più lavori contemporanei. Infine nella scelta dell'identificativo da assegnare si è tenuto conto che la struttura della sessione prevede,per ogni servizio aggiunto, che sia specificata la data e il tempo di utilizzo con precisione fino ai millisecondi.E' proprio questo tempo quindi che si è scelto per la sua univocità, avendo ritenuto praticamente nulla la probabilità che un utente possa manualmente avviare più esecuzioni nello stesso millisecondo.Questo ci ha consentito di non aumentare con altri campi la complessità della struttura di una sessione.

- *void set_user (in string ip, in Account account,in long rights).* Tale metodo viene invocato nel momento in cui un utente si autentica.A seconda che l'utente abbia lasciato o meno in precedenza una sessione in *sleeping* provvede in alternativa a:
  - risvegliare la sessione se l'utente aveva lasciato servizi in esecuzione;
  - settare nella sessione relativa le informazioni di account e i diritti che ha ricevuto come parametri di ingresso insieme all'ip;

- *Applications close_session (in string ip).* Questo metodo viene invocato nel momento in cui viene richiesta la chiusura della sessione.Anche qui dobbiamo sottolineare alcune importanti modifiche apportate. Nella versione precedente del metodo la sessione veniva in ogni caso chiusa in seguito alla richiesta di logout. Ora invece la richiesta di uscita dall'ambiente prevede ancora l'invocazione del metodo *close_session(),*ma le azioni che esso esegue sono diverse. Anzitutto si deve controllare che l'utente non abbia altre sessioni attive nello stesso momento,relative a dispositivi diversi,nel qual caso le applicazioni

sospese vengono aggiunte a una di queste sessioni. La sessione a cui accodare le suddette applicazioni può essere scelta secondo politiche e criteri diversi. Ad esempio si può scegliere la sessione più remota,quella più recente oppure quella con il maggiore o il minore numero di applicazioni attive. Tale criterio di scelta può essere facilmente modificato andando ad agire sul solo metodo privato *Vector getIpSessionOf(string login)* creato appunto per separare la politica di selezione.

Se invece l'utente sta chiudendo la sua ultima sessione attiva e sta aspettando la terminazione di qualche computazione la sessione viene trasformata in *sleeping session*,pronta per essere risvegliata alla successiva autenticazione dell'utente. Resta immutato ovviamente il caso in cui l'utente non lascia esecuzioni in sospeso,per cui la sessione può essere definitivamente chiusa.

Dal punto di vista prettamente implementativo,è da notare che si è scelto di lasciare immutata la struttura della tabella che tiene conto delle sessioni.Essa prevede che ciascuna sessione venga riconosciuta in base ad una stringa identificativa.Tale stringa sarà una informazione relativa all'ip,nel caso di utente presente nel sistema,mentre sarà una informazione relativa alla login,nel caso di utente che ha lasciato la sessione sospesa.

- *void delete_batch_service(in string serviceId, in string login)*.Questo metodo provvede a tener traccia dello stato attivo o non attivo dei servizi utilizzati da un utente nell'ambito di una data sessione. In particolare provvede a notificare nella sessione la terminazione di un servizio batch..Ha bisogno in ingresso dell'identificativo della specifica esecuzione della applicazione,che è quello restituito dal metodo *add_service()* di cui si è detto in precedenza,per poterne aggiornare correttamente lo stato.Oltre a ciò bisogna fornire anche il parametro relativo alla login,in quanto la terminazione del servizio può anche avvenire in un momento di assenza dell'utente;come conseguenza di ciò la sessione in *sleeping* non può essere recuperata in base all'ip (inesistente se l'utente non è presente nell'ambiente),per cui deve essere ritrovata in base alla login.Questo segue il criterio seguito nella scelta dell'identificativo della

sessione,di cui si è parlato spiegando il metodo precedente.E' da notare infine che all'atto del logout potrebbe essersi verificato un accodamento delle applicazioni batch sospese in qualche altra sessione attiva dello stesso utente.In seguito a questa eventualità il metodo che stiamo descrivendo potrebbe non trovare una sessione in *sleeping:*di conseguenza deve ricercare la corretta applicazione da aggiornare in tutte le sessioni dell'utente attive in quel dato momento.

Oltre alle funzionalità già esistenti nel *Session Manager* ,ne è stata aggiunta una nuova. Abbiamo in precedenza accennato al fatto che è proprio questo componente che si fa carico della ricezione degli eventi che indicano la terminazione di un servizio computazionale,perché questa informazione gli occorre per tenere nota delle successive applicazioni sospese o terminate e per comunicare col *Messenger Service* per permettere la creazione di un nuovo messaggio per l'utente. Il *Session Manager* è dunque il consumatore degli eventi prodotti sul canale asincrono istituito con il servizio batch.Il canale è creato usufruendo dell' *Asynchronous Communication Service,* componente creato per questo scopo. Il *Session Manager* è dunque continuamente in ascolto sul canale per la ricezione di nuovi eventi.Dal punto di vista implementativo questa funzionalità è realizzata utilizzando la classe java Observable. Essa consente di prevedere un oggetto osservatore e un oggetto osservato. Nel nostro caso l'osservatore è costituito dallo stesso *Session Manager* ,che costituisce l'oggetto che viene richiamato allorquando l'osservato rileva qualche cambiamento sul canale,ovvero la presenza di un nuovo evento. L'osservato da parte sua è costituito da un thread che si pone in ascolto sul canale,effettuando periodicamente una ricerca degli eventi.In tal modo si evita che il Session Manager debba bloccarsi ogni volta che deve controllare l'eventuale presenza di nuovi eventi sul canale.
Quando è presente un nuovo evento ,questo fatto viene rilevato dal thread del *Session Manager* che sta in ascolto sul canale;in conseguenza di questo viene invocato il metodo :

- *void update (Observable osservato, java.lang.Object message)*: questo metodo provvede anzitutto ad indicare il servizio come non più attivo nella

sessione che lo ha richiesto,e successivamente passa al *Message Service* le informazioni utili alla creazione di un nuovo messaggio per l'utente.

IV.5.3.1 MODIFICHE IMPLEMENTATIVE ALL' APPLICATION SERVICES MANAGER

Si è già accennato alla necessità di apportare alcune importanti modifiche implementative all'interno dei componenti già esistenti. In particolare facciamo notare che all'interno del componente *Application Service Manager* è previsto un metodo che richiede il logout dell'utente ed è nello specifico:

- *void logout (in string ip)*

Nella precedente versione del logout venivano rilasciati tutti i servizi adoperati nella sessione,in quanto si prevedeva che essi potessero essere soltanto interattivi;come conseguenza di ciò la sessione veniva senz'altro chiusa in ogni caso.In seguito alla nostra analisi ,l'integrazione di servizi di tipo computazionale ha richiesto una revisione di tale metodo,il quale va a rilasciare soltanto i servizi di tipo interattivo.Se restano attivi servizi batch successivamente al logout,il metodo *close_session()* di cui si è ampiamente parlato in precedenza ,provvede a discriminare eventuali sessioni di *sleeping*.

IV.5.4 MESSAGE SERVICE

Si riporta di seguito l'interfaccia IDL del *Message Service*:

```
module UbiSys{

        typedef sequence <string> files_input;

        struct Message {
            string user_name;
            string batch_service_name;
            string link_result;
```

```
                boolean delivered;   //true = consegnato
                long message_id;
                string idForUser;
                files_input inputFiles;
            };

            typedef sequence<Message> Messages;
            typedef sequence <long> list_id;



    module MessageService{

            interface MES {
              exception MessageNotFound{};

             Messages find_message(in string login);

             void update_list(in string login,in long message_id)
                  raises (MessageNotFound);

             void update_multiple_list(in string login,in list_id
                message_id)
                  raises (MessageNotFound);

             void delete_read(in string login);

             void delete_old_message();

             void create_message_user(in string login,in string
                link_result,in string  service_name,in string
                idForUser,in files_input inputFiles);

    };//End interface MES
  };//End module MessageService
};//End module UbiSys
```

Le strutture dati definite sono:

- *Message*. E' la struttura che definisce i campi di un generico messaggio da recapitare all'utente all'atto della terminazione del servizio applicativo computazionale. Più in dettaglio:

  - il campo *user_name* rappresenta la login con la quale l'utente è registrato in *UbiSystem*;

  - il campo *batch_service_name* è il nome con cui il servizio è conosciuto nell'ambiente;

  - il campo *link_result* è una stringa che contiene l'informazione sul link dal quale l'utente può prelevare il risultato della sua elaborazione;

  - il campo *delivered* serve a tener traccia della consegna del messaggio all'utente;

  - il campo *message_id* è un identificativo del messaggio e serve alla gestione delle operazioni da compiere su di esso(cancellazione,aggiornamento ,ecc…);

  - il campo *idForUser* identifica la particolare esecuzione del servizio,in quanto ,ripetiamo, un utente può richiedere diverse esecuzioni parallele dello stesso servizio ,con input diversi,e deve poter distinguere i vari job che ha avviato.Questo campo può essere usato sia insieme a quello successivo dei file di input,sia come sua alternativa,nel caso i file di input siano in numero eccessivo.Al momento questo campo è previsto,ma per far riconoscere all'utente le applicazioni in corso,abbiamo preferito mostrare i file di input con cui è stato avviato il servizio.

  - il campo *inputFiles* è un array di stringhe contenente l'elenco dei file di input immessi per eseguire il servizio.

- *Messages*. Rappresenta una lista di messaggi. In altri termini, un array d*i Message*.

- *list_id* .Tale struttura definisce una lista di interi ,che servono per gestire liste di messaggi anziché un messaggio alla volta.Ciascun componente della lista sarà un identificativo per un messaggio.

I metodi implementati sono:

- *Messages find_message(in string login).* Tale metodo riceve in ingresso la login dell'utente e restituisce una lista di messaggi da consegnare.

- *void update_list(in string login,in long message_id).* Il parametro di ingresso è la login dell'utente oltre all'identificativo del messaggio. Tiene conto se il messaggio è stato letto o meno dall'utente,andando a modificare un valore booleano che indica l'eventualità messaggio/messaggio non letto.

- *void update_multiple_list(in string login,in list_id message_id).* Questo metodo svolge azioni analoghe al precedente,ma è riferito a una lista di messaggi,per permettere appunto la gestione di una intera lista ,anziché di un solo messaggio alla volta.

- *void delete_read(in string login).* Tale metodo riceve in ingresso la login dell'utente ed ha come effetto quello di cancellare i messaggi che risultano letti.

- *void delete_old_message().*Tale metodo al momento non è implementato,ma è stato previsto per consentire eventualmente all'amministratore di cancellare messaggi che hanno superato un certo periodo di permanenza nel sistema. L'implementazione di questo metodo risulta utile nel momento in cui si prevederà di rendere permanenti le informazioni relative alle sessioni e ai messaggi,memorizzandole in opportune strutture.

- *void create_message_user(in string login,in string link_result,in string service_name,in string idForUser,in files_input inputFiles).* Questo è il metodo che permette di creare nuovi messaggi da recapitare all'utente. I

messaggi contengono tutte le informazioni utili,come i file utilizzati in input al servizio e il link al risultato,per consentire di scaricare i file di output.

V. SERVIZI APPLICATIVI DI TIPO BATCH

V.1 INTRODUZIONE

Durante il lavoro di tesi svolto, il sistema *UbiSystem* è stato integrato con nuovi componenti di sistema e con nuove funzionalità di gestione inserite in componenti esistenti.Di questo si è diffusamente trattato nel capitolo precedente. Dal punto di vista dei servizi applicativi invece è stata resa possibile l'integrazione dei servizi di tipo computazionale,che nella precedente versione del prototipo non erano presenti.

Lo scopo di questo capitolo è allora quello di mostrare un paradigma da seguire per introdurre un servizio applicativo batch nell'architettura. In altre parole se si vorranno in seguito inserire nuovi servizi di questo tipo nell'ambiente ci si potrà attenere al modello proposto,per garantire una corretta gestione del servizio nonché una appropriata interazione con gli altri componenti del sistema. Facciamo presente che nella progettazione si è ritenuto opportuno introdurre la specifica di consentire l'uso di qualsiasi servizio computazionale ai soli utenti muniti di account.Questa limitazione agli utenti registrati è infatti indispensabile per la gestione progettata.

Nel modello da noi proposto ci riferiremo in un primo momento ad una generica applicazione di tipo computazionale,senza entrare nel dettaglio del servizio offerto. Mostreremo quindi quale deve essere a nostro parere la struttura nella quale deve essere incapsulato un qualunque servizio batch che si voglia integrare nell'ambiente,a prescindere

da cosa effettivamente faccia il servizio stesso. In un secondo momento mostreremo un esempio che renda più chiara la struttura proposta.

A questo proposito si è scelto di inserire una semplice applicazione che effettua la compressione di file,la cui durata di esecuzione dipenderà ovviamente dalla dimensione dei file da comprimere. Per consentire di fornire un esempio più chiaro e una dimostrazione maggiormente significativa, il servizio è stato rallentato opportunamente per consentire di mostrare chiaramente la nuova gestione delle sessioni di utente.

Il servizio,al pari di quelli già esistenti,è stato implementato con tecnologia CORBA e poi, generato l'apposito wrapper, è stato esposto all'interno dell'ambiente sottoforma di servizio WEB.

V.2 DESCRIZIONE FUNZIONALE DEI SERVIZI DI TIPO BATCH

Lo scopo di questo paragrafo è quello di descrivere le funzionalità che deve esporre una applicazione batch in *UbiSystem*, indipendentemente dalla particolare logica del servizio stesso.

Innanzitutto bisogna tener presente che la natura stessa del tipo di servizio che stiamo trattando è tale da indurci senz'altro a fare l'ipotesi che occorrano dei dati di input da sottomettere alla applicazione affinché questa possa elaborarli per produrre dei risultati. Per tale motivo non si può pensare che l'utente,una volta scelto di usare un servizio batch, possa poi avviarne subito l'esecuzione:è necessaria una fase iniziale in cui sia possibile inviare, sul nodo della rete dove si trova il servizio, i dati di input scelti dall'utente necessari all' elaborazione. E' appena il caso di far notare esplicitamente che i dati di input,organizzati in appositi file dipendenti dalla particolare applicazione da eseguire,si trovano ,nel nostro modello ,sul dispositivo con cui l'utente si collega ad *UbiSystem*.

Le funzionalità che deve offrire all'esterno una applicazione di questo tipo sono :

- *use_service.* Come appena detto deve essere possibile gestire una fase iniziale,che precede l'esecuzione vera e propria,in cui è per forza di cose richiesta una certa interattività con l'utente.La funzionalità deve consentire quindi l'upload dei dati di input dal device dell'utente al server su cui si trova il

servizio,ma deve anche permettere lo svolgimento di altre azioni preliminari alla corretta esecuzione.In particolare si deve tener presente che a conclusione dell'esecuzione,si deve informare il *Session Manager* di tale evente,affinché possano essere eseguite le azioni per una corretta gestione della sessione.Di conseguenza questa funzionalità deve anche garantire al servizio l'uso di un canale asincrono su cui produrre gli eventi descritti. Si tratterà del canale asincrono descritto nel precedente capitolo,quando abbiamo introdotto le funzionalità del *Session Manager*. Ricordiamo a tale proposito che la gestione da noi pensata prevede che tra il generico servizio batch e il manager delle sessioni venga creato un canale asincrono per le notifiche di terminazione delle applicazioni. A tale scopo si utilizzano i servizi esposti da un altro componente di *UbiSystem* , l' *Asynchronous Communication Service*,che gestisce appunto un meccanismo di comunicazione asincrona mediante appositi canali a eventi.Queste funzionalità potranno essere esposte ad esempio con un metodo di nome *use_service()*.

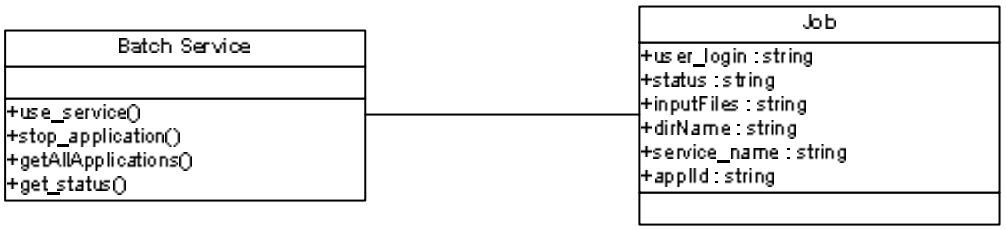
- *get applications*. Poiché la gestione prevede che un utente possa avviare parallelamente diverse esecuzioni di uno stesso servizio,è opportuno che egli possa distinguere tra le varie elaborazioni in corso.Questa funzionalità permetterà allora di riconoscere le diverse operazioni avviate in un momento precedente e ancora in esecuzione.Questo potrebbe essere possibile grazie a un metodo del tipo *getAllApplications()*.

- *stop application*. Attraverso tale funzionalità sarà possibile fermare l'esecuzione di una o più elaborazioni in corso.Ciò deve rendersi possibile perché l'utente potrebbe essere ad esempio non più interessato al risultato di una elaborazione richiesta precedentemente.Ciò avverrà attraverso il metodo esposto *stop_application()*.

- *get status*. Questa funzionalità può essere creata per consentire all'utente di avere traccia dell'andamento dell'elaborazione,al fine di conoscere ad esempio

il tempo rimanente previsto per la conclusione.Questa funzionalità potrà essere offerta attraverso un metodo *get_status()*,la cui implementazione -è già evidente in questa fase della progettazione – dipenderà dal particolare servizio integrato.

Quanto esposto potrà essere schematizzato come segue (figura V.1) :



/* IDL Batch Service */

```
module BatchService{
     module  BatchAppl {

          typedef sequence <string> path_name ;

          struct Job {
          string user_login;
          string status;  //indica quanto manca alla fine del
                            lavoro batch
          path_name inputFiles;
          string dirName;
          string service_name;
          string applId;

          };

          typedef sequence<Job> job_active;

           interface application {

              void use_service(in string service_name,in string
                               ip,in path_name batch_input,in
                               string dirName);

              string  get_status(in string ip,in long id);

              void stop_application(in string ip,in string
                                    applId);

              job_active getAllApplications (in string ip);

           };
        };
};
```

La struttura dati definita è:

- *Job*. Si tratta di una struttura dati atta a gestire le diverse elaborazioni di un servizio computazionale,in quanto conserva tutte le informazioni necessarie al riguardo. In particolare:
  - il campo *user_login* rappresenta la login dell'utente per il quale l'applicazione è in esecuzione.
  - il campo *status* indica il tempo rimanente per la terminazione del servizio;
  - il campo *path_name* è un array di stringhe atte a contenere i nomi dei file di ingresso per l'elaborazione;
  - il campo *dirName* è il riferimento alla directory nella quale sono caricati i file forniti dall'utente;
  - il campo *service_name* fornisce il nome del servizio applicativo batch;
  - il campo *applId* è un identificativo univoco per ciascuna elaborazione.

Per quanto riguarda i metodi esposti,essi sono:

- *void use_service (in string service_name ,in string ip,in path_name batch_input,in string dirName).* Tale metodo riceve in ingresso il nome del servizio richiesto, l'ip del dispositivo con cui l'utente si è connesso al sistema,oltre ai nomi dei file di input e della directory nella quale questi ultimi sono stati caricati precedentemente.Questo metodo si occupa di gestire tutta la fase di preparazione all'esecuzione del servizio:tra l'altro infatti provvede alla creazione di un canale ad eventi asincrono con il *Session Manager*.Questo canale servirà per notificare al gestore della sessione l'evento di terminazione dell'elaborazione.E' ancora tale metodo che provvede all'invio di detto evento.

- *string get_status(in string ip, in long id).* Tale metodo viene esposto per consentire all'utente di poter interrogare il servizio circa lo stato di una certa

elaborazione.In particolare è stato pensato che esso possa fornire il tempo rimanente alla conclusione della applicazione.Tale informazione dipende strettamente dalla logica del servizio,per cui il metodo ,nel nostro caso generale non è stato implementato.

- *job_active getAllApplications(in string ip).* Serve per consentire all'utente di prendere visione di tutte le elaborazioni che sono in corso in un dato momento.Permette di distinguere le diverse esecuzioni mediante un identificativo univoco, contenuto nella struttura dati *Job*.

- *void stop_application(in string ip, in string applId).* Una volta che l'utente ha visualizzato tutte le elaborazioni in corso,questo metodo rende la possibilità di forzare l'arresto di qualche elaborazione.Il parametro *applId* fornito, permette di bloccare il processo corretto.

V.2.3 CLASS DIAGRAM

Sarà più interessante a questo punto l'analisi del diagramma delle classi che ci consente di comprendere nei dettagli l'architettura di una applicazione di tipo batch,se la si vuole integrare nell'ambiente *UbiSystem* (figure V.2 e V.3).

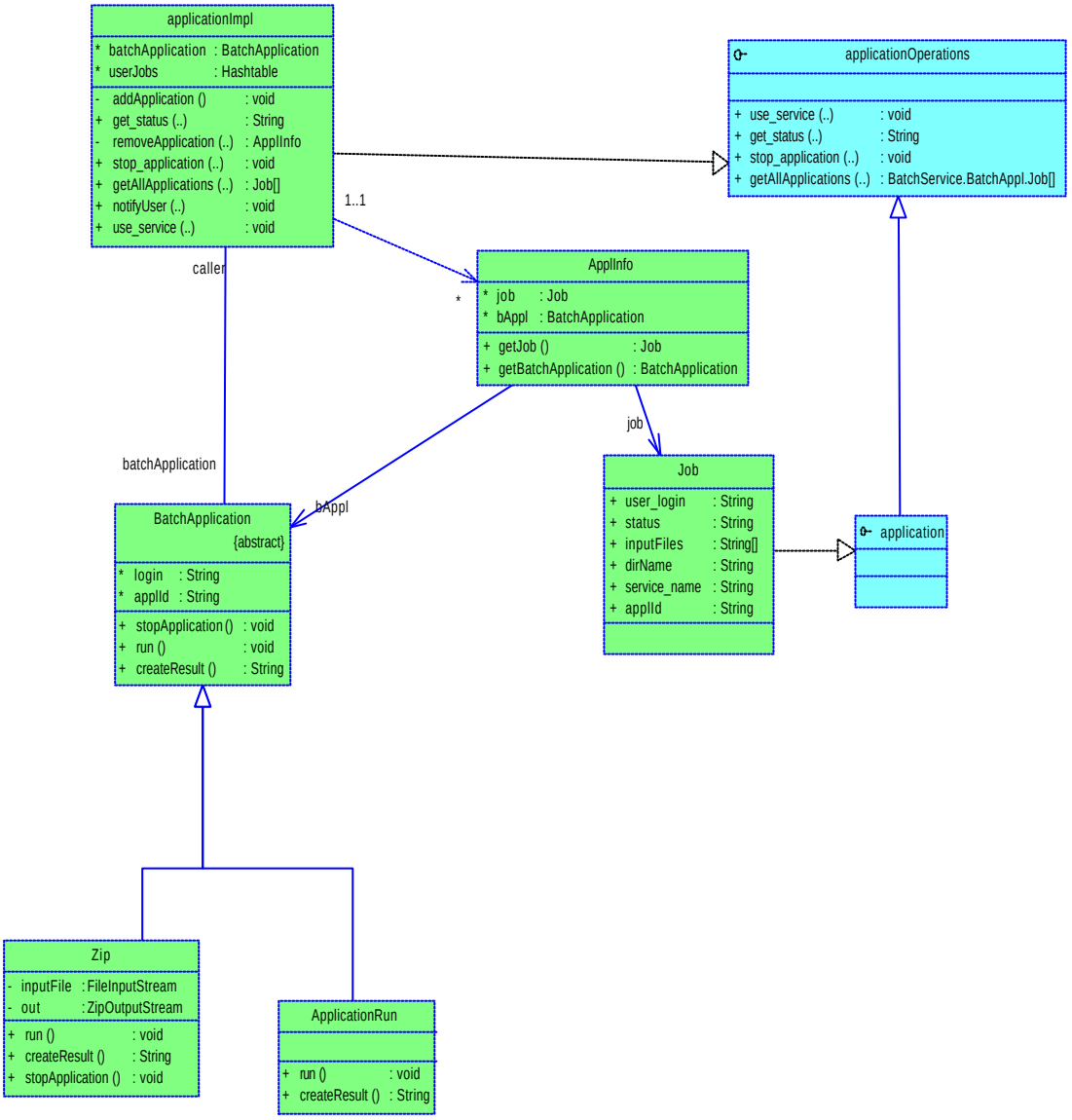


Fig. V.2 Diagramma delle classi del servizio batch (I)

Innanzitutto notiamo che l'interfaccia *applicationOperations* fornisce le funzionalità esposte dal servizio,che sono messe a disposizione dell'utente. Di esse si è già parlato diffusamente nei precedenti paragrafi. L'implementazione di tali funzionalità è invece fornita ad opera di *applicationImpl*. L'implementazione deve aver cura di memorizzare e di distinguere,per ogni utente, tutte le diverse esecuzioni

richieste per ciascun servizio e per questo si serve della struttura *ApplInfo*.

Quest'ultima da una parte contiene informazioni sull'utente che ha richiesto l'elaborazione e sulle specifiche della richiesta effettuata (questo grazie alla classe *Job* descritta in precedenza),dall'altra contiene il riferimento alla elaborazione stessa.

In figura è mostrato anche un esempio di come poter introdurre praticamente un servizio in questa architettura. Nel caso specifico ci riferiamo, come anticipato nell'introduzione del capitolo,ad una applicazione che effettua la compressione di file.

La classe relativa è quella che in figura prende il nome di *Zip*.

Sull'introduzione del servizio nell'ambiente,possiamo soffermarci per qualche chiarimento. Precisiamo anzitutto che ,come si evince dalla figura,è stata progettata una classe astratta che fornisse un paradigma da seguire per tutti i tipi di servizio che si volessero inserire. In particolare quando si implementa la logica del servizio specifico si deve provvedere ad implementare opportunamente i metodi *stopApplication()*, *run()* e *createResult()*. Ora, la presenza di questa classe astratta può apparire superflua ad una prima interpretazione. C'è però la possibilità di considerare due tipi di soluzione al problema dell'integrazione di un servizio batch.Una prima ipotesi potrebbe essere quella di far sì che ogni servizio computazionale vada ad implementare tutta l'intera architettura proposta.In tal modo ogni applicazione avrà una copia delle classi di supporto appena descritte. Resta chiaro ovviamente che in tal caso, la classe che in figura appare come astratta perde di significato,e le si può sostituire direttamente quella che implementa la logica del servizio. A tale proposito notiamo che è certamente più corretto considerare un diagramma del tipo mostrato in figura V.3.

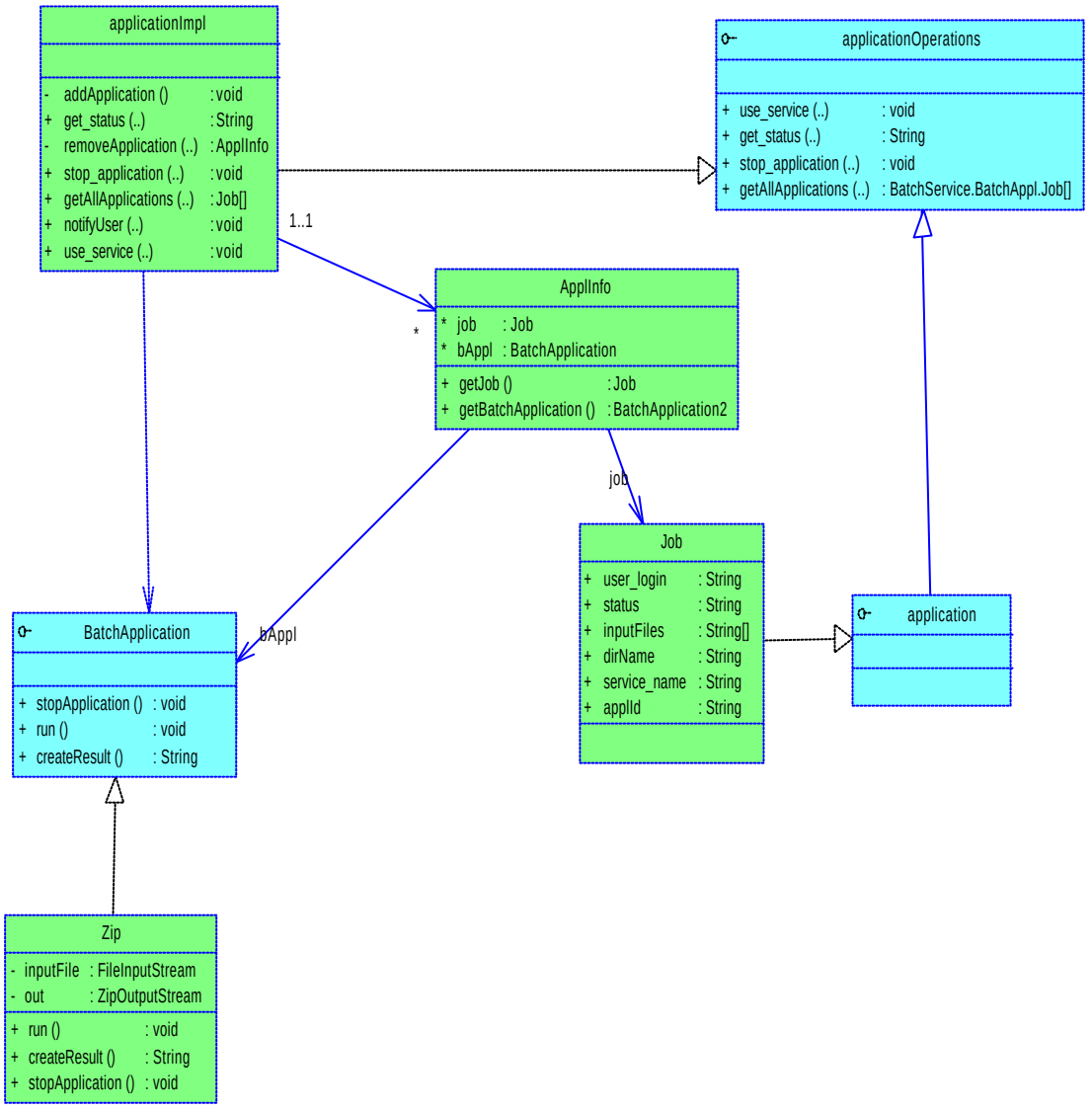


Fig. V.3 Diagramma delle classi del servizio batch (II)

Una seconda soluzione può essere invece rappresentata proprio dallo schema precedente della figura V.2 : si potrebbe cioè prevedere che qualsiasi servizio computazionale si voglia integrare nell'ambiente,lo si può introdurre come estensione della classe astratta *BatchApplication*, così come accade per la classe *Zip* e per la classe *ApplicationRun*. La scelta della corretta applicazione che si vuole utilizzare avverrebbe mediante l'uso di un abstract factory pattern.

Questo modo di procedere presenta l'indubbio vantaggio di non richiedere ulteriore implementazione di codice e di consentire una integrazione molto semplice di qualsiasi servizio. Tuttavia seguendo questo principio si andrebbe a ledere l'idea di progetto della architettura di *UbiSystem*,che ,sappiamo,prevede tra l'altro la presenza di un manager dei servizi applicativi. Infatti con la soluzione proposta si darebbe all'*Application Services Manager* (ASM) la conoscenza e la responsabilità soltanto di un servizio batch generico,nascondendogli i dettagli dei singoli servizi (ad esempio la registrazione ,il reference ecc…).La seconda soluzione è stata quindi riferita per completezza,e per fornire una nuova idea di progettazione da poter tenere eventualmente in conto nel momento in cui si volesse riprogettare e ristrutturare l'intero sistema.

Con la prima soluzione (figura V.3) invece le applicazioni possono essere differenziate, e dipendono direttamente dal manager ASM, così come prescrivono le attuali specifiche di progetto. In tal modo i servizi vengono tutti distinti e ad essi si possono applicare differenti condizioni di funzionamento,ad esempio una diversa tariffazione.

V.3 IMPLEMENTAZIONE DEI SERVIZI

I servizi applicativi sono stati implementati come oggetti Corba ed in seguito sono stati wrappati ed esposti nell'ambiente come servizi web.

Affinché un utente possa accedere alle funzionalità da essi esposte è necessario che i servizi si siano registrati presso il manager dei servizi applicativi.

Di seguito, si mostra il procedimento seguito per la realizzazione di un generico servizio applicativo batch.

V.3.1 IMPLEMENTAZIONE DI UN OGGETTO CORBA

Analogamente a come fatto nel capitolo precedente, si considera la realizzazione di un generico servizio applicativo: *BatchService*.

Il primo passo da fare è la scrittura dell'interfaccia IDL salvata nel file: *"BatchService.idl"*. Questa è data in input al compilatore idl-java che genera tre gruppi di classi:

- Classi lato Client: *_BatchServiceStub.java, BatchService.java*
- Classi Lato Server: *BatchServicePOA.java, BatchServicePOATie.java, BatchServiceOperations.java*
- Classi di supporto: *BatchServiceHelper.java, BatchServiceHolder.java.*

Successivamente si dovrà scrivere il codice relativo alle classi *Server*, *Impl* e *Client*, che hanno struttura simile a quelle descritte per i componenti di sistema nel capitolo precedente.

V.3.1.1 REGISTRAZIONE DI UN SERVIZIO APPLICATIVO

Come per i servizi di tipo interattivo,anche per quelli computazionali è prevista una procedura di registrazione nell'ambiente.Più in dettaglio la registrazione di un servizio applicativo avviene presso l'*Application Services Manager*. Infatti, la richiesta di registrazione dovrà essere inoltrata all'*ASM* che provvederà a memorizzare il riferimento al servizio e a richiedere all'*Ontology Service* la validazione della descrizione del servizio stesso. La registrazione sarà resa possibilie grazie alla classe *Server.java.* Facciamo notare che in questa sede si riportano soltanto i passi fondamentali della registrazione di un servizio,per consentire una lettura più completa;in ogni caso per approfondire la procedura di registrazione rimandiamo ai lavori di tesi precedenti [50], alle cui direttive ci siamo rifatti e che hanno trattato in maniera esaustiva l'argomento.

Il codice che implementa il server è riportato di seguito per comodità:

```
public class Server {

    public Server(args) {
      initOrb(args);
    }

    public static void main(String[] args) {
      new Server(args);
    }

    private void initOrb (String args[]){
      Service service;
        try{
            // initializza l'ORB.
```

```
            ORB orb = ORB.init( args, null );

            // ottiene il riferimento al Naming Server
            NamingContextExt nc = NamingContextExtHelper.narrow(
            orb.resolve_initial_references("NameService"));

            //Ottiene rif POA
            POA rootPOA = POAHelper.narrow
                  (orb.resolve_initial_references("RootPOA"));

            //Attiva POA
            rootPOA.the_POAManager().activate();

            //Crea l'obj servant
            Impl servant=new Impl();

            //registra obj nel POA
            service = servant._this(orb);

            //legge il riferimento dell'ASM dal naming service
            ASM_Reference=ASMHelper.narrow
                  (nc.resolve(nc.to_name("ASM")));

            Reference = orb.object_to_string(pdf);
            String[] ref = new String[1];
            ref[0] = Reference;

            new ApplicationServiceProvider(ASM_Reference, ref,
                                    "BATCHServiceConfig.xml");

            //mette l'appl servente in stato di attesa
            orb.run();

        } catch (Exception e){
        e.printStackTrace();
        }
    }
}
```

Di seguito si riporta anche il codice della classe ApplicationServiceProvider:

```
public class ApplicationServiceProvider implements Runnable {

      //private ontoWS.ws.OntologyServerInterface server = null;
      private String id = new String();
      private int delay = 6000;
      private Thread thread = null;
      private String file = new String();
      private String xmlDescription = new String();
      private ASM MyASM_Reference;
      private String[] references;
      private String service_name = new String();
      private String xmlFile = new String();
      private int rights = 0;


      public ApplicationServiceProvider() {
            thread = new Thread(this);
            thread.start();
      }

        public ApplicationServiceProvider(ASM ASM_Reference,
String[] references, String xmlFile) {
            thread = new Thread(this);
            thread.start();
            MyASM_Reference = ASM_Reference;
            this.references = new String[references.length];
            this.references = references;
            this.xmlFile = xmlFile;
      }
      public static void main(String [] args) throws Exception {
            new ApplicationServiceProvider();
      }

      public void run()  {
            // Legge i parametri di configurazione
```

```
            readConfig();
            // Registra il servizio presso l'SSM
            if (buildDescription())
                  subscriveService();
      }

      private boolean buildDescription() {
            boolean res = false;
            try {
                  BufferedReader in = new BufferedReader(new
FileReader(file));
                  xmlDescription = "";
                  String buf = null;
                  while ( (buf = in.readLine()) != null)
                      xmlDescription = xmlDescription + buf + "\n";
                  res = true;
            } catch (Exception e) {
                  System.out.println ("Impossibile trovare il file "
+ file);
            }
            return res;
      }
      private void subscriveService() {
            boolean go = true;
            while (go) {
              try {
                System.out.print("Tentativo di registrazione del
                  servizio... ");

MyASM_Reference.register_application_service(xmlDescription,
service_name, rights, true, references) ;
                System.out.print("Registrazione del servizio
                  terminata!");
                 go = false;
                 }catch (UbiSys.ApplicationServiceManager.
                         ASMPackage.ServiceNotRegistered e){
                       System.out.println("Service not registered");
                 }
```

```
                catch (Exception e) {
                       System.out.println("non riuscito.\nProblemi
di connessione con l'Ontology Server");
                }
                try {
                     thread.sleep(delay);
                } catch (Exception e) {}
           }
     }

     public void readConfig() {
           Hashtable config = new Hashtable();
           String[] xml = {"name", "description", "delay",
"rights"};
           System.err.println (xmlFile);
           DOMParser parser = new DOMParser();
           try {
             parser.parse(xmlFile);
           } catch (Exception se) {
              System.err.println ("File Config.xml not found or
                malformed");
              return;
           }
           Document document = parser.getDocument();
           Element element = document.getDocumentElement();
           for (int i=0; i<xml.length; i++) {
             NodeList nodelist =
                 element.getElementsByTagName(xml[i]);
             if (nodelist.getLength()==1)
                 config.put(xml[i], (String)
                   nodelist.item(0).getFirstChild().getNodeValue());
             else {
                 System.err.println("Parsing error: " + xml[i] + "
not found");
             }
           }
           service_name = (String) config.get("name");
           file = (String) config.get("description");
```

```
            rights=Integer.parseInt((String)config.get("rights"));
            try {
              delay = Integer.parseInt((String)config.get("delay"));
            } catch (Exception e) {
                  delay = 60000;
            }
      }
}
```

Il file di configurazione BATCHService.xml utilizzato in questa classe riporta il nome simbolico del servizio con cui esso sarà registrato nell'ambiente e il livello di accesso necessario per utilizzarlo. Esso ha struttura:

```
<?xml version="1.0"?>
<config>
    <name>
          BatchService_name
    </name>
    <description>
          BatchService.daml
    </description>
    <delay>
          80000
    </delay>
    <rights>
          2
    </rights>
</config>
```

Il file *BatchService.daml* è scritto in linguaggio DAML+OIL e contiene le informazioni riguardanti il servizio, oltre alle precondizioni sia hardware che software che il dispositivo utente deve rispettare per usufruirne.

Per chiarezza si riporta, in forma schematica la descrizione del generico servizio applicativo computazionale:

DESCRIZIONE DEL SERVIZIO BATCH GENERICO	
ResourceName	BATCHService
ResourceTextDescr	Questa risorsa consente di usufruire di un servizio batch.
ResourceType	User
ResourcePageUrl	http://192.168.0.8/servizi/

				BatchService.jsp
ResourceUrl				http://192.168.0.133/axis/services/BATCHService
ResourceInteractive				false
ResourceSingleInstance				true
ResourceCommand				Start
offeredBy				Mach_08
Resource Profile	**Hw Precondition**	**Audio-output**		true
		Audio-imput		False
		Video-input		false
		Connection Description	**bandwidth**	100
			bandwidthunit	Mbit/s
		Resolution Description	**resolutionwidth**	240
			resolutionheight	320
			resolutionunit	pixel
			resolutionbpp	32
			resolutiongraphics	true
		MemoryType Description	**memoryamount**	6
			memoryunit	Gb
			memoryusagetype	Storage
		Cpu Description	**cpuspeed**	1.33
			cpuunit	MHz
	Sw Precondition	**Os Description**	**Name**	Windows
			Vendor	Microsoft
			Version	4.0
		application	**Name**	Windows Media Player
			Vendor	Microsoft
			Version	8.0

V.3.2 WRAPPING DI UN SERVIZIO

Per l'esposizione di un servizio Corba come Web Service si segue lo stesso procedimento impiegato per wrappare l'*Ubiquitous Gateway*.

La classe *W_Impl.java* sarà client del servizio Corba, quindi all'interno del suo costruttore deve eseguire le operazioni:

- *Inizializzazione dell'ORB.* Così come nel server, anche nel client ogni operazione CORBA deve essere preceduta dall'inizializzazione dell'oggetto ORB.
- *Dichiarazione variabile per l'oggetto.* Le invocazioni all'oggetto remoto avvengono in forma di invocazioni di metodo su un'istanza di una classe Java avente lo stesso nome dell'oggetto CORBA da invocare. Prima di tutto occorre quindi dichiarare una variabile che in seguito conterrà il riferimento remoto all'oggetto.
- *Reperimento dell'Object Reference*. L'OR dell'oggetto server desiderato viene reperito da dove era stato preventivamente memorizzato dal processo server (ad es. Naming Service o file system).
- *Creazione del riferimento remoto all'oggetto.* La variabile precedentemente dichiarata viene inizializzata con l'Object Reference reperito. Viene instaurata la connessione remota all'oggetto e si possono effettuare le richieste.

L'*ASM* espone un metodo per richiedere l'IOR di un servizio applicativo.

Di seguito è riportato il codice Java del costruttore della classe *W_Impl.java*:

```
public W_Impl() {
      Service Reference = null;
      try {
         //Inizializzazione ORB
         org.omg.CORBA.ORB orb=org.omg.CORBA.ORB.init(args,null);

         // ottiene il riferimento al Naming Server
         NamingContextExt nc = NamingContextExtHelper.narrow(
         orb.resolve_initial_references("NameService"));

         ASM_Reference = ASMHelper.narrow
            (nc.resolve(nc.to_name("ASM")));
         String server_ref = ASM_Reference.
```

```
            require_service_reference("Service_name",ip);
          org.omg.CORBA.Object ob = orb.string_to_object(server_ref);
          Reference = ReferenceHelper.narrow(ob);




          //ora è possibile invocare direttamente i metodi
    }
    catch (Exception e){
        e.printStackTrace();
     }
}
```

In questo modo i nuovi servizi batch possono essere completamente integrati in *UbiSystem*.

VI. ESEMPIO D'USO

Dopo aver analizzato i vari componenti di *UbiSystem* e la loro interazione, risulta interessante analizzare il comportamento del sistema dal punto di vista dell'utente.

VI.1 LO SCENARIO IN CUI OPERANO I SERVIZI

Dal punto di vista del presente lavoro di tesi e del contributo apportato al prototipo *UbiSystem*, lo scenario in cui i servizi realmente operano riveste un ruolo marginale,in quanto le problematiche affrontate riguardano aspetti di gestione della sessione di utente e di integrazione di un certo tipo di servizi applicativi (quelli batch) e ciò prescinde dall'ambiente in cui i servizi operano. Per mostrare quindi il comportamento del sistema possiamo riferirci ad uno scenario tipico ,che è poi quello in cui è stato inizialmente progettato il prototipo. Esso è costituito da due stanze,che rifacendoci ai lavori precedenti chiameremo ancora Stanza della Segreteria e Stanza di Motion Capture,così come mostrato in figura VI.1.

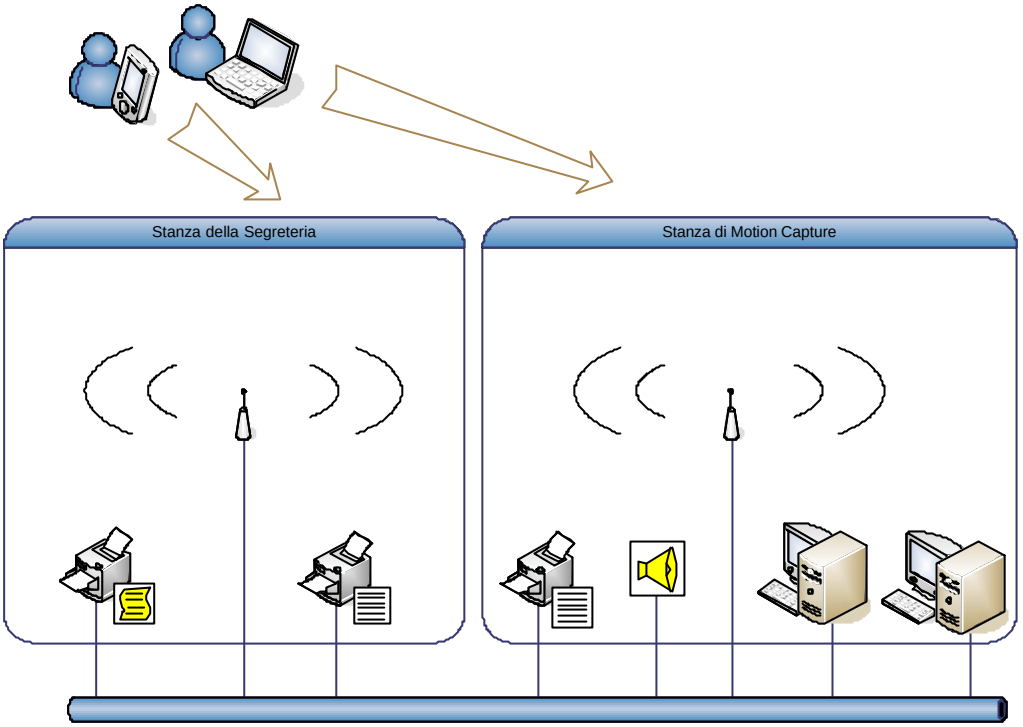


Fig. VI.1 Scenario relativo all'ambiente pervasivo implementato.

Ciascuna stanza comprende un sensore, un access-point D-link, per il rilevamento dei dispositivi mobili che entrano a far parte dell'ambiente.

Nella Stanza della Segreteria sono installate due stampanti,mentre la stanza di Motion Capture comprende invece una stampante per la stampa in bianco e nero ed un impianto sonoro per la riproduzione, ad esempio, di brani musicali. L'intero ambiente è coperto da una rete LAN con access-point wireless.

Oltre a questi ambienti c'è una stanza per le conferenze, dotata di schermo gigante, vicino la segreteria, e quindi nel raggio di copertura dello stesso access-point.

I componenti software di *UbiSystem* sono poi dislocati sui nodi della rete nel modo illustrato nella figura VI.2 riportata di seguito.

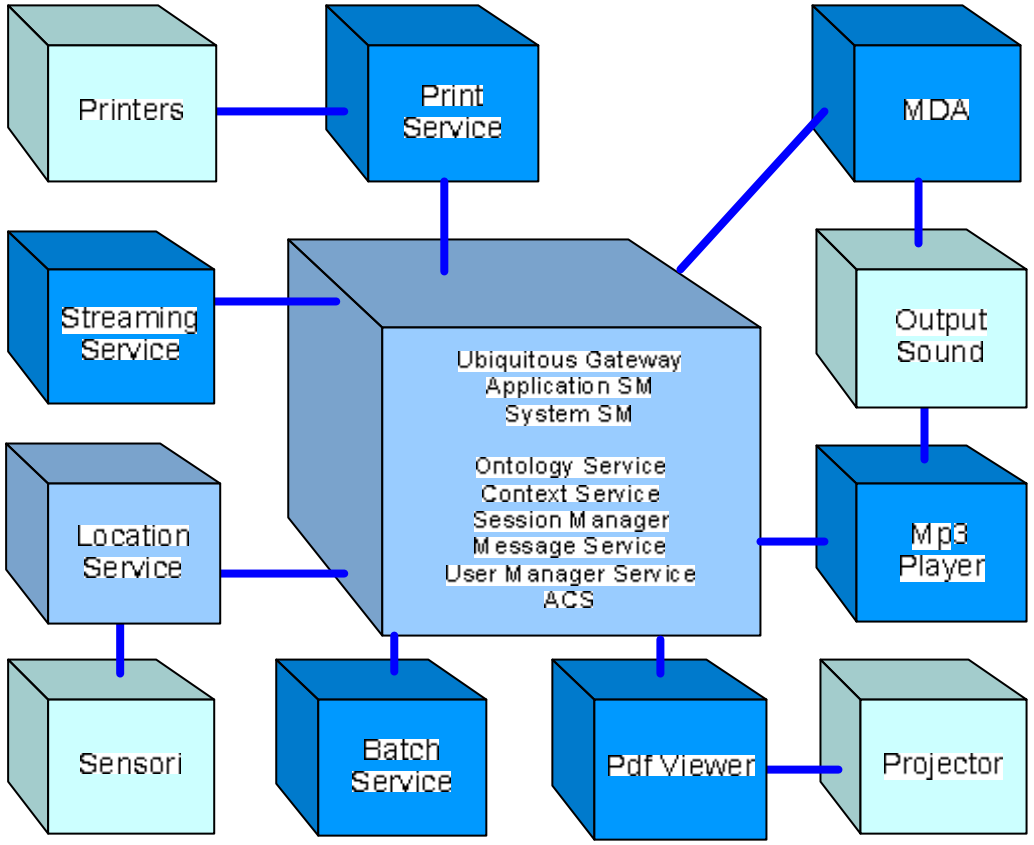


Fig. VI.2 Dislocazione dei Programmi sui nodi della rete

I componenti potrebbero essere dislocati anche in maniera diversa.

VI.1.1 LO START-UP DI UBISYSTEM

Per avviare il sistema è necessario seguire un certo ordine.

Innanzi tutto è necessario avviare i manager, in modo da consentire la successiva registrazione dei servizi.

Il primo servizio a dover essere attivato deve essere l'*Ontology Service*, in modo che possa verificare le descrizioni dei servizi per validarne la correttezza sintattica e semantica secondo quanto asserito nelle ontologie. Per avviare questo servizio è necessario che sia già attivo il Fact-client, e successivamente bisognerà caricare l'ontologia di base.

Dopo aver eseguito queste operazioni potranno essere attivati gli altri servizi. Bisogna, però, fare attenzione che il *Location Service* sia attivato prima del *Context Service* e che all'attivazione di un qualunque servizio d'ambiente il *Session Manager* sia su.

Ad esempio, allo start-up di ACS, al termine della corretta registrazione del servizio viene generato il seguente output (figura VI.3):

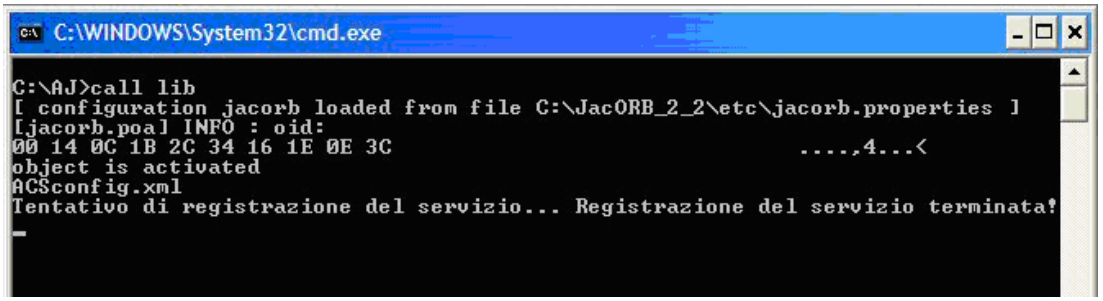


Fig. VI.3 Attivazione e registrazione dell'ACS

Prima che l'utente possa accedere al sistema, ovviamente, è necessario che *Ubiquitous Gateway* sia stato attivato.

VI.1.2 L'INTERFACCIA UTENTE

Una volta attivato il sistema occorre analizzare il problema dell'interazione con l'utente che, per ipotesi di progetto, dovrà avvenire tramite browser.

Attualmente è lo stesso sviluppatore che mette a disposizione una serie di pagine Web attraverso le quali interagire per sfruttare le funzionalità offerte dal proprio servizio.

In alternativa, sarebbero possibili altri due tipi di scenari:

- nel momento in cui il servizio effettua la registrazione fornisce, oltre alla propria descrizione, anche l'interfaccia Web mediante la quale consumare il servizio; l'interfaccia verrebbe ospitata all'interno di un Web Application Server del sistema;
- lo sviluppo di un modulo capace di generare dinamicamente l'interfaccia Web per un servizio a partire dalla descrizione offerta.

Le due soluzioni proposte non sono esclusive ma complementari: si potrebbe adottare la seconda quando non è disponibile anche l'interfaccia utente.

Come già per ciascuno degli altri servizi pubblici è stata sviluppata un'applicazione web che offre una semplice interfaccia grafica per richiamare le funzionalità del relativo Web Service o semplicemente per mostrare la sua Home Page. Per tale scopo è stata adoperata la tecnologia delle Java Server Page e delle Servlet Java, che costituiscono una soluzione

multipiattaforma per la realizzazione di pagine HTML dinamiche. JSP e Servlet permettono, infatti, di svincolarsi sia dal tipo di sistema operativo che dal tipo di browser, in quanto il linguaggio Java viene interpretato solamente dal lato server sollevando il browser da questo compito e limitando la sua funzione a semplice interprete di pagine HTML.

Affinché l'utente possa interagire, attraverso le pagine *jsp*, con l'ambiente è necessario che *Tomcat* (*servlet container* utilizzato) sia attivato.

VI.2 UBISYTEM IN AZIONE

L'entry point per l'ambiente è l'indirizzo http://www.ubinet.it/. La relativa pagina è mostrata in figura VI.4:

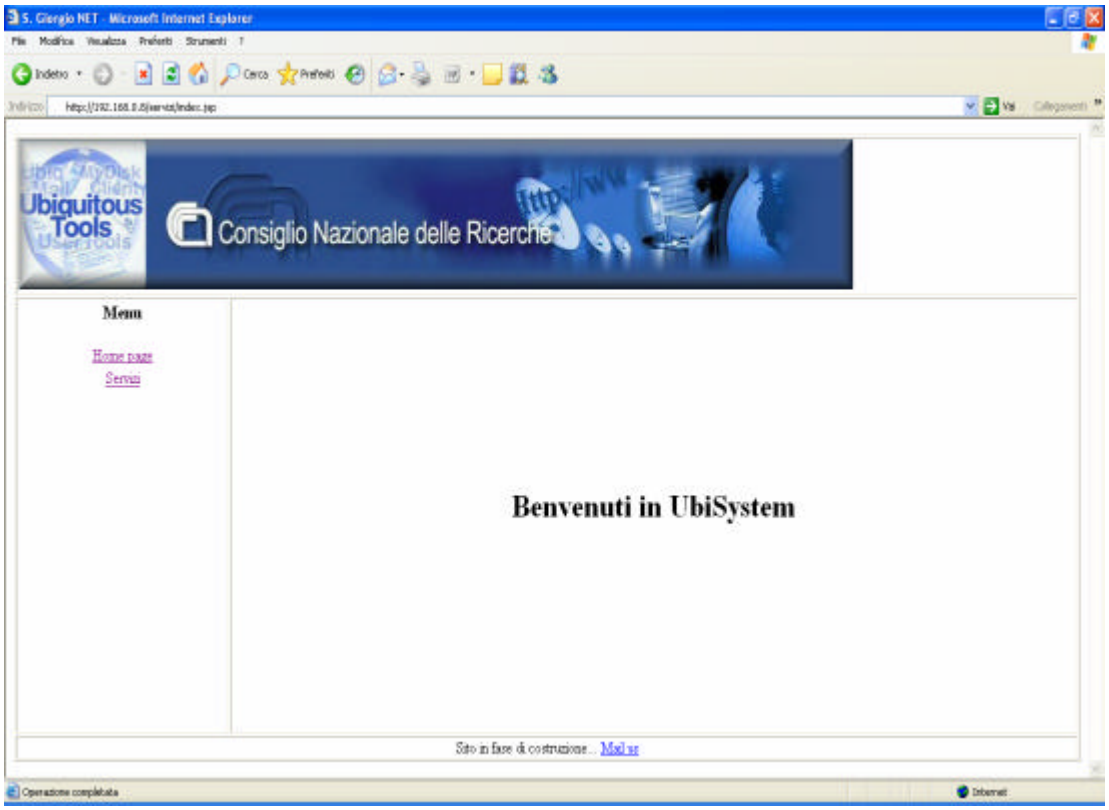


Fig. VI.4 Home Page dell'ambiente pervasivo

Si tenga presente che in questo capitolo verranno presentati solo gli esempi d'uso relativi al contributo apportato dal presente lavoro di tesi,rimandando ai lavori precedenti i casi non espressamente trattati in questa sede ([50],[51]). Si fa inoltre notare che per

quanto riguarda l'esempio d'uso del servizio batch,si è scelto di adoperare una semplice applicazione che effettua la compressione di file.

La durata dell'applicazione dipende ovviamente dalla dimensione dei file da comprimere,ma è il servizio è stato in ogni caso rallentato opportunamente per consentire di mostrare chiaramente la nuova gestione delle sessioni di utente.

Scegliendo la voce *Servizi* del Menù a sinistra, viene visualizzata la pagina in cui vengono elencati tutti i servizi correntemente attivi nell'ambiente insieme con una breve descrizione. A seconda delle caratteristiche dell'utente, del suo dispositivo e della sua locazione i servizi sono resi disponibili o meno. Per i primi viene fornito il relativo link, per i secondi l'indicazione del motivo per cui non sono accessibili (figura VI.5).

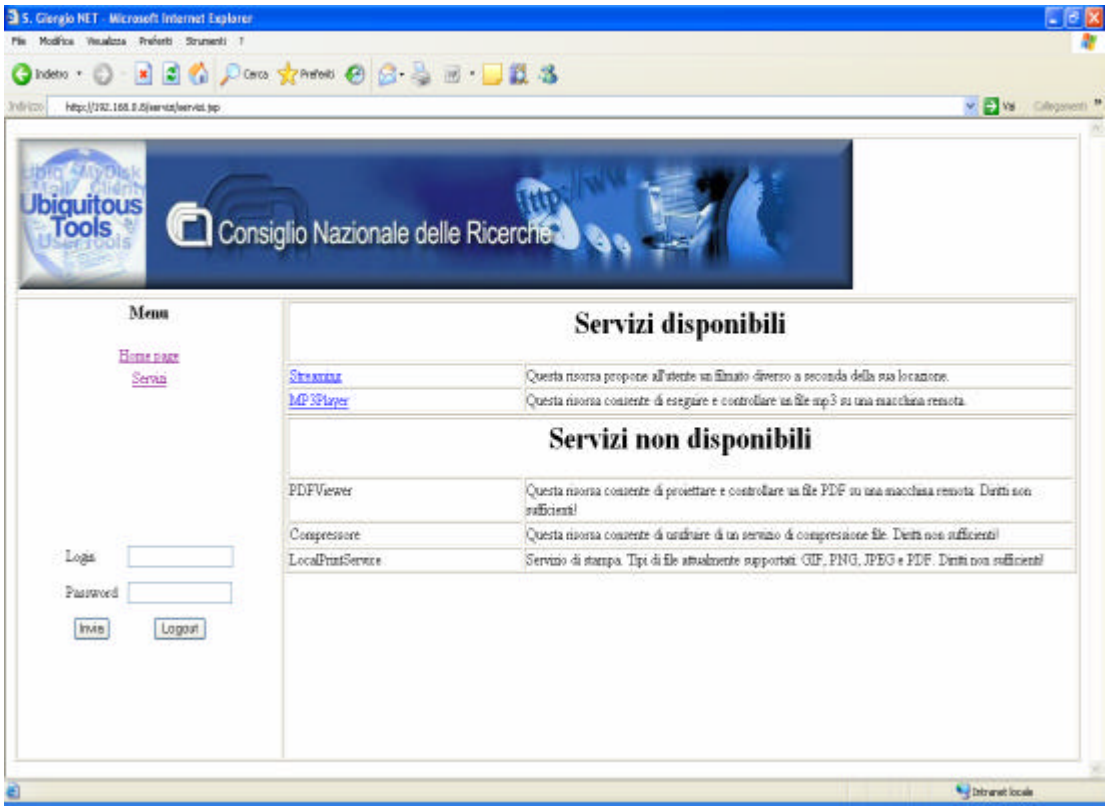


Fig. VI.5 Elenco dei servizi dell'ambiente

Nella lista non compare il servizio di musica d'ambiente in quanto non offre funzionalità fruibili direttamente dall'utente.

VI.2.1 AUTENTICAZIONE DI UN UTENTE

Se un utente è munito di Account ha la possibilità di poter accedere ad un maggior numero di servizi. L'utente con login *"Pluto"* potrà usufruire di tutti i servizi.

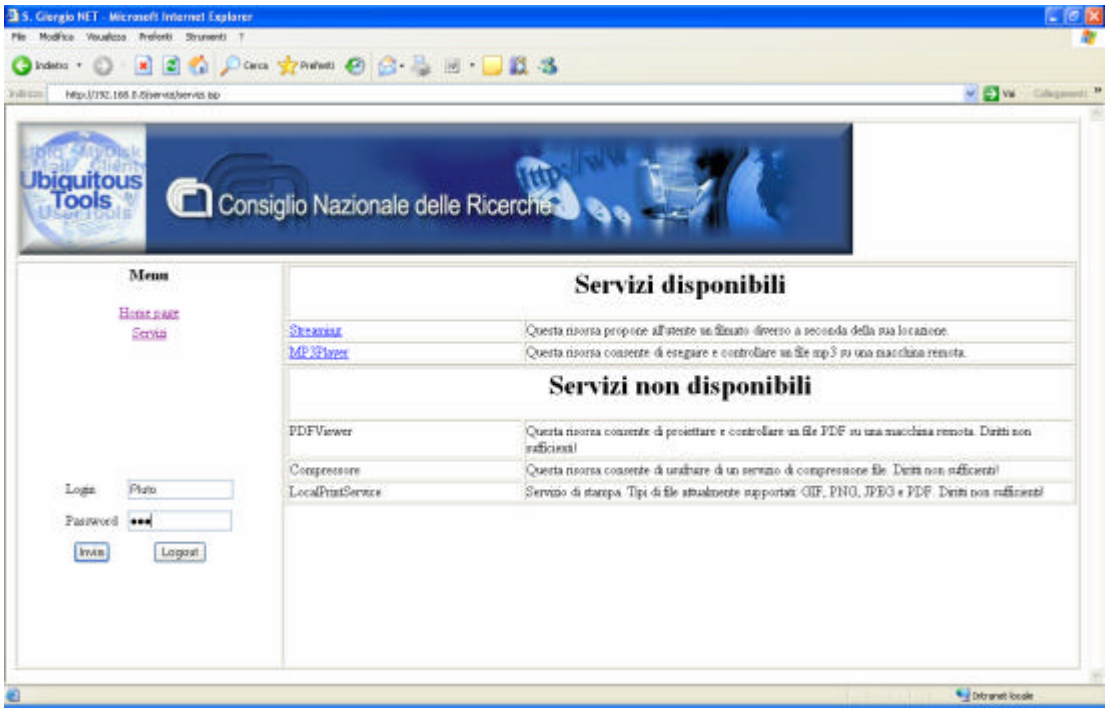


Fig. VI.6 Autenticazione di Pluto

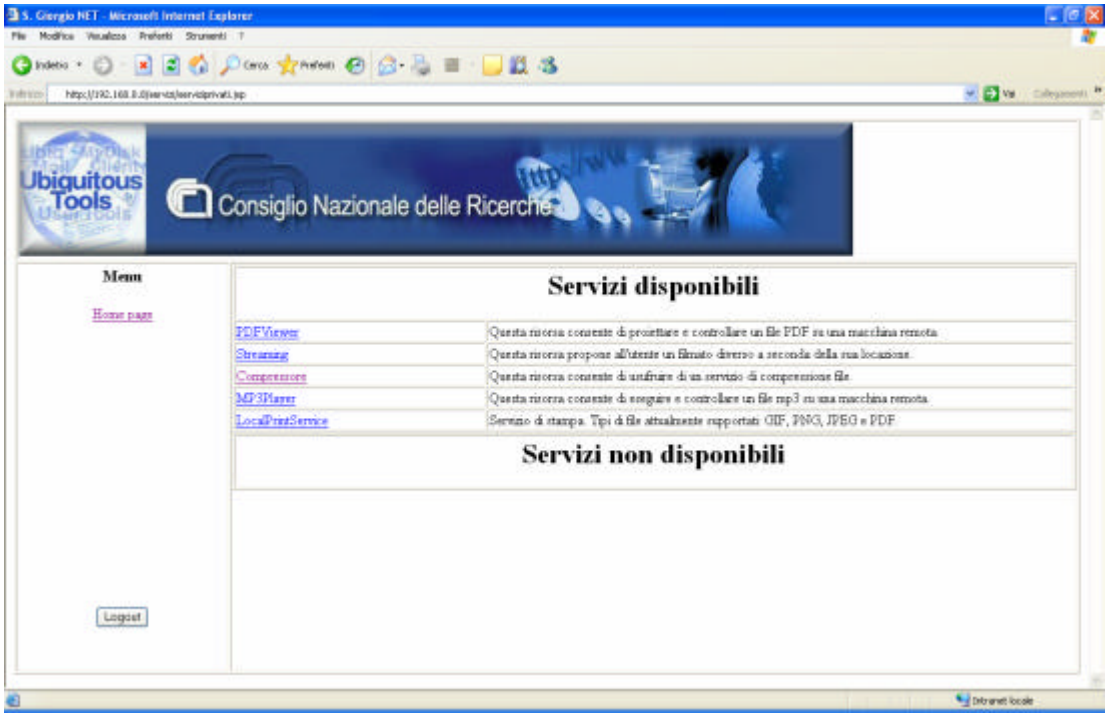


Fig. VI.7 Lista personalizzata per Pluto

Nel caso in cui i dati inseriti non fossero validi, sarà visualizzata una pagina di errore apposita (figura VI.8):

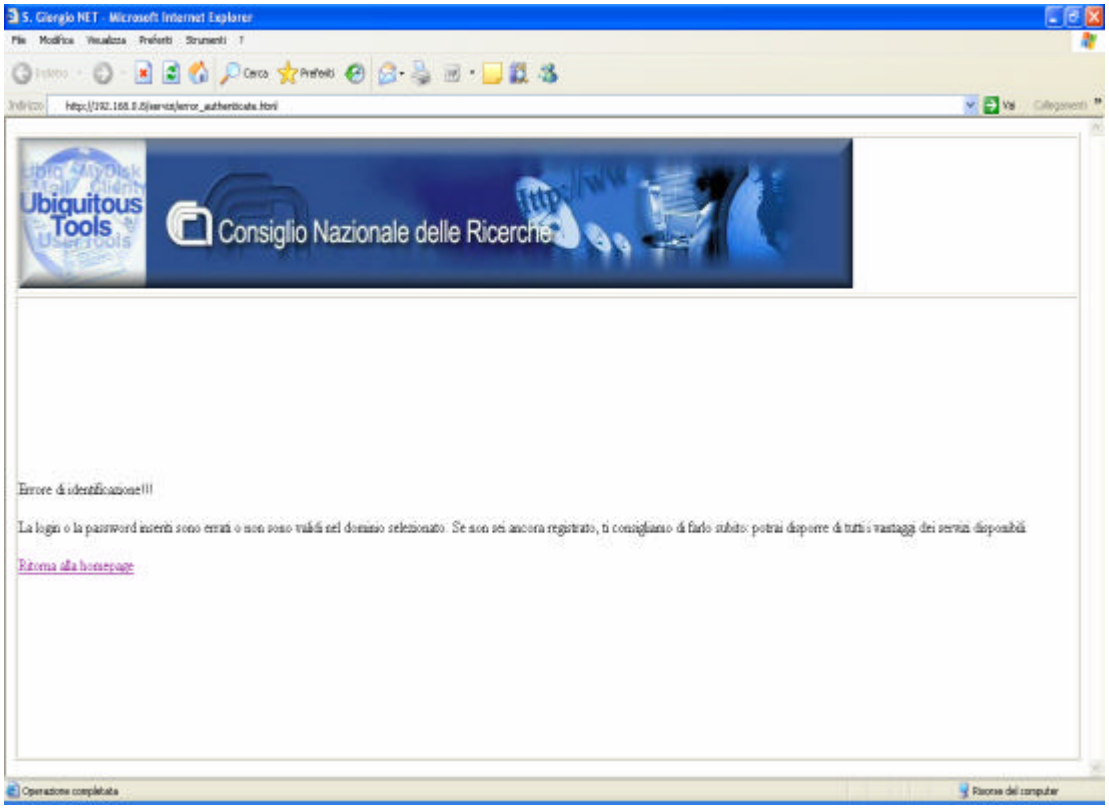


Fig. VI.8 Errore durante l'autenticazione

VI.2.2 IL SERVIZIO COMPRESSORE DI FILE

Dalla pagina web contenente l'elenco dei servizi disponibili è possibile accedere alla home page del servizio batch mediante il link relativo (figura VI.9).

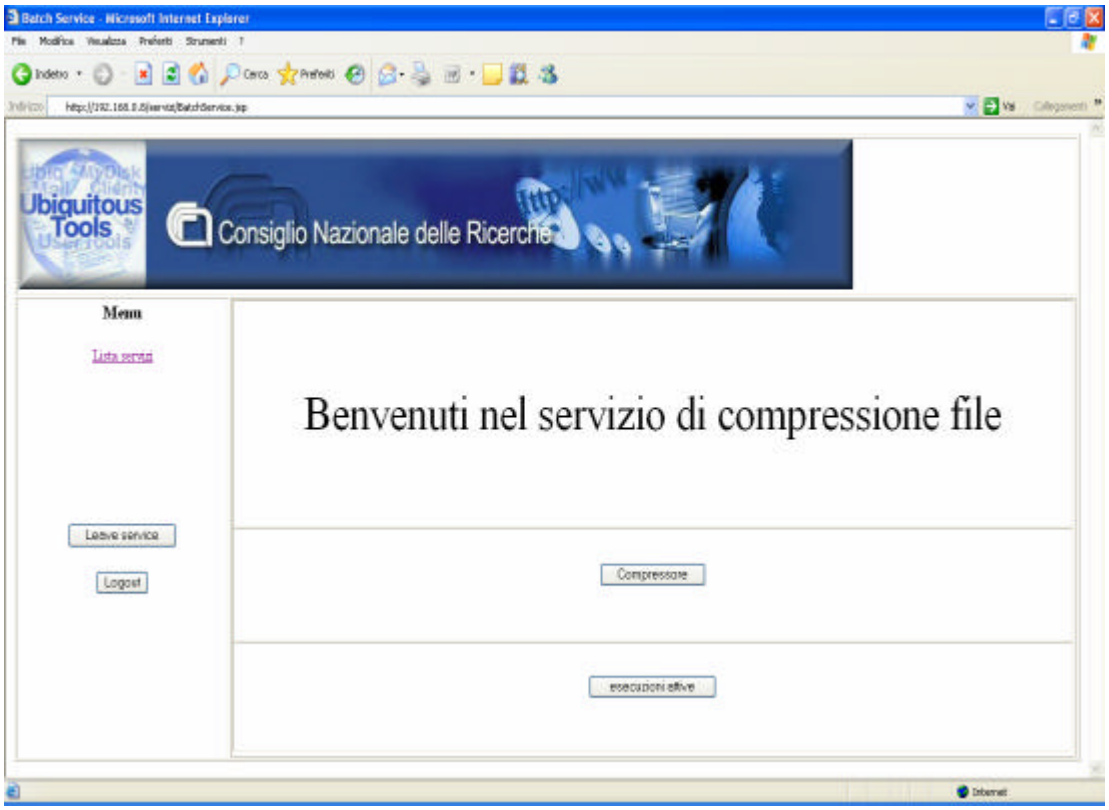


Fig. VI.9 Home page del servizio di compressione file

Da qui è possibile passare all'utilizzo del servizio (pulsante "*Compressore*"), controllare eventuali servizi sospesi (pulsante *"esecuzioni attive"*), abbandonare il servizio (pulsante "*Leave service*") oppure lasciare l'ambiente (pulsante "*Logout*").

Cliccando sul bottone relativo al servizio si accede alla pagina (figura VI.10):

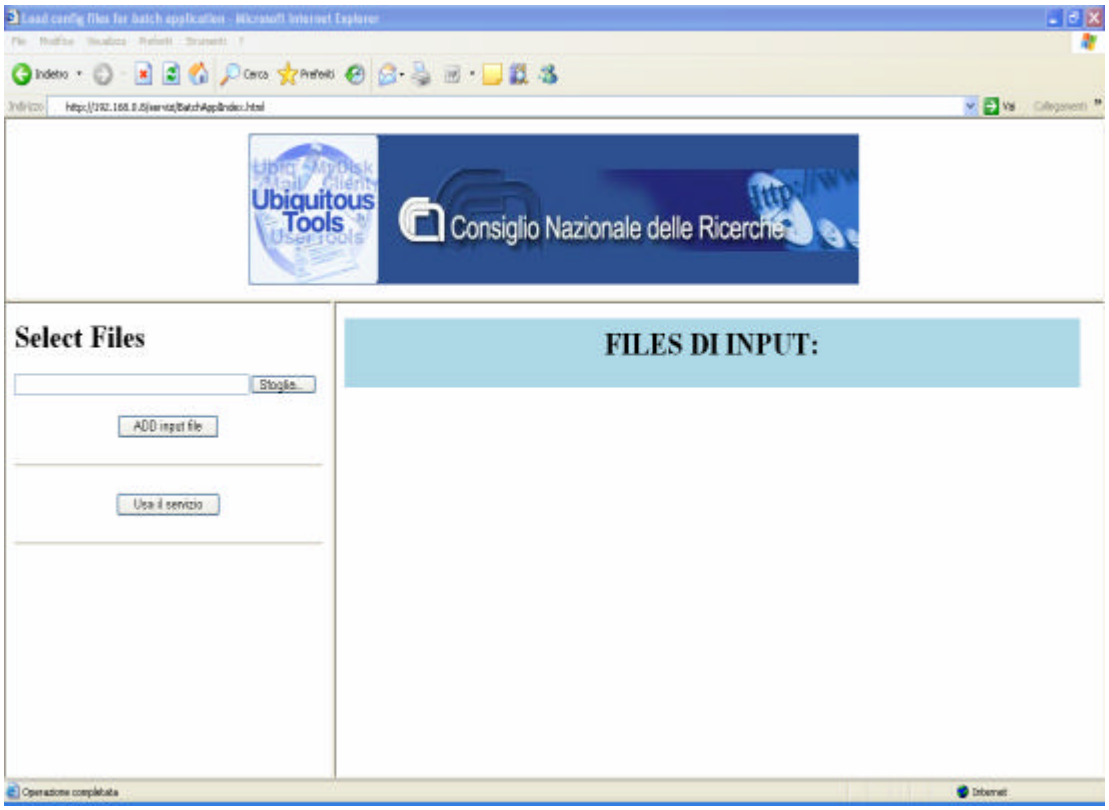


Fig. VI.10 Pagina del servizio di compressione file

L'utente mediante il pulsante *Sfoglia* potrà selezionare i file contenuti nel proprio dispositivo che desidera inoltrare al servizio batch;nel nostro esempio saranno i file che si desidera far comprimere;di essi verrà eseguito l'upload sul nodo in cui si trova il servizio (figura VI.11).

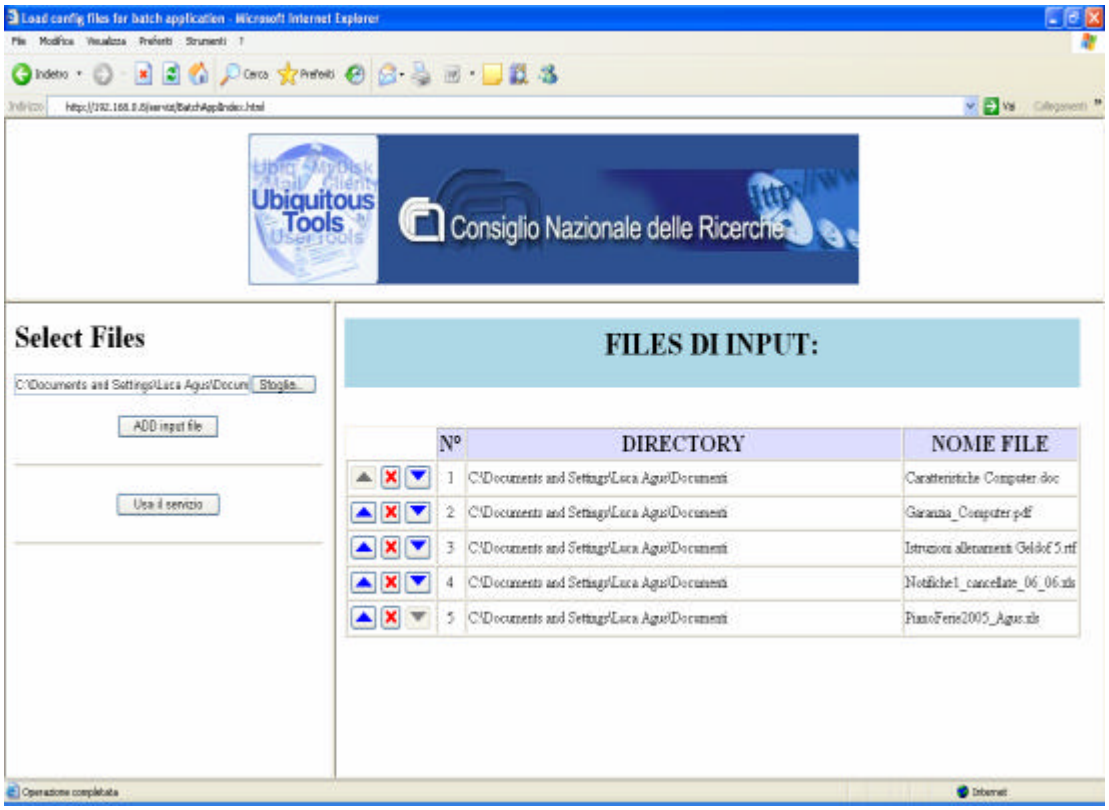


Fig. VI.11 Caricamento dei file da comprimere

Dopo si può avviare l'esecuzione mediante il pulsante *"Usa il servizio"*.

Questa azione porterà alla pagina successiva che comunica all'utente il corretto avvio dell'esecuzione (figura VI.12). Da questo momento l'utente può tornare alla propria pagina personale per utilizzare altri servizi,per effettuare il logout per uscire dall'ambiente,oppure può scegliere di utilizzare nuovamente lo stesso servizio batch,magari con altri file di input.

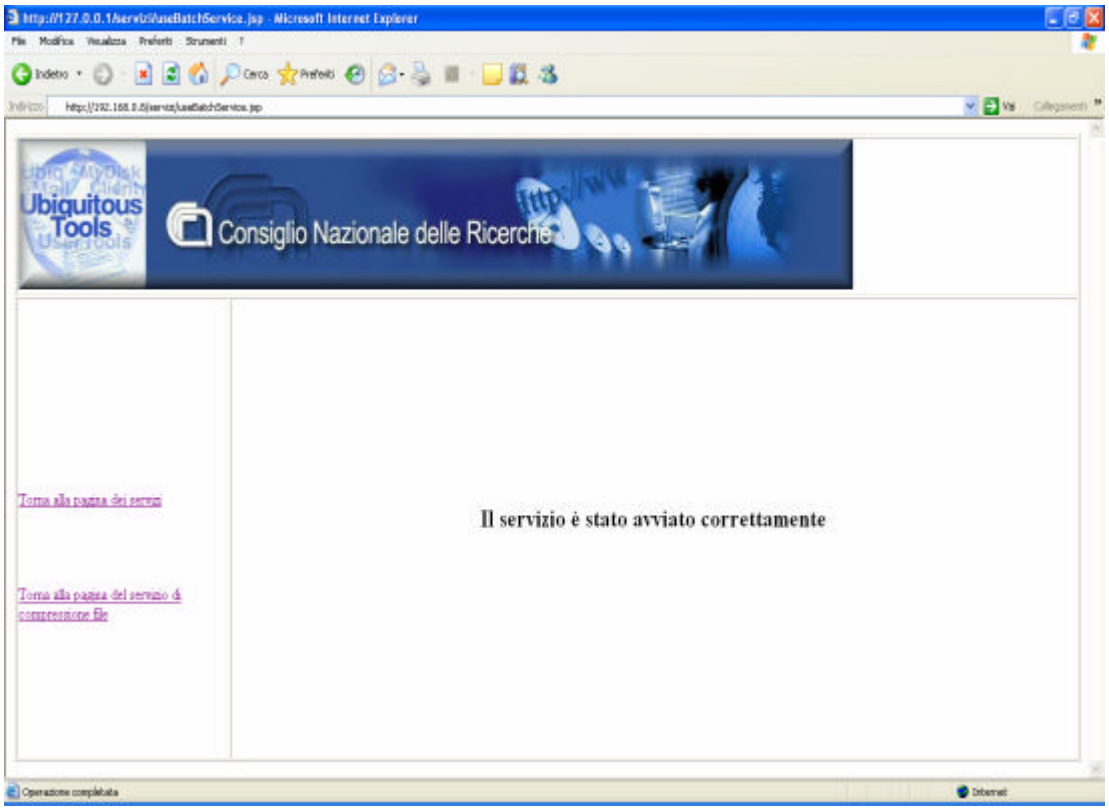


Fig. VI.122 Notifica di avvio del servizio di compressione

Nel caso in cui si scelga di tornare alla pagina del servizio,si possono ,come spiegato prima ,controllare le applicazioni batch che sono in esecuzione per conto dell'utente. All'interno di questa pagina si può anche scegliere di fermare l'esecuzione di un servizio avviato,ad esempio perché non si è più interessati ad avere un risultato.Ciò è evidenziato dalla figura VI.13 successiva.

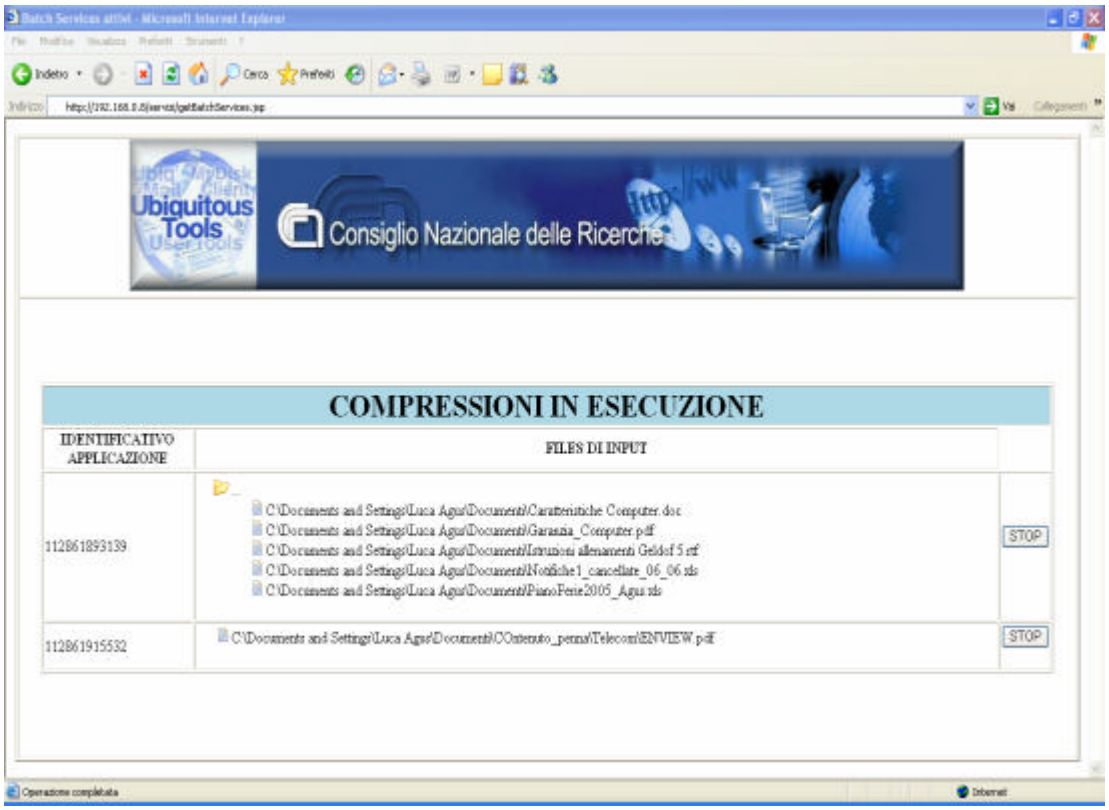


Fig. VI.13 Pagina di controllo delle esecuzioni del servizio in corso

Nel caso in cui l'utente sceglie di abbandonare l'ambiente ,lasciando però in attività i servizi batch che ha precedentemente avviato,la sua sessione viene posta in stato di sleeping,per poter essere risvegliata in un momento successivo all'atto di una nuova autenticazione (figure VI.14 e VI.15).

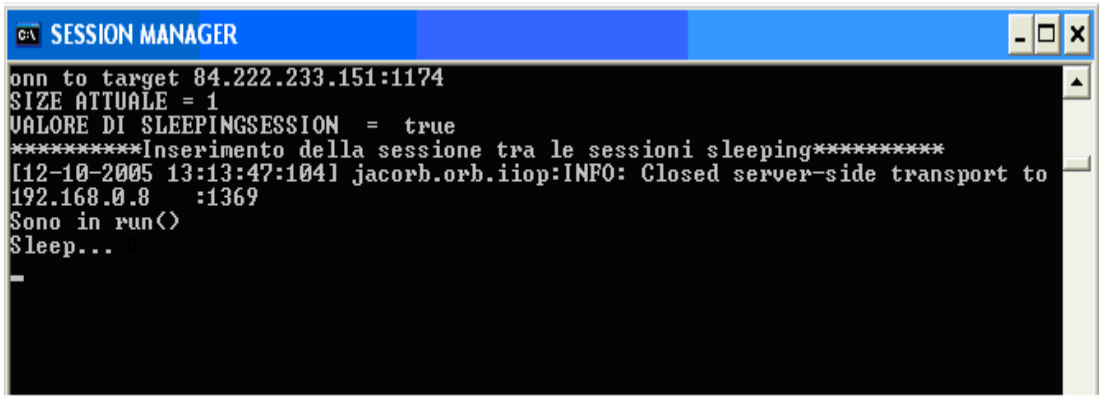


Fig. VI.14 Inserimento della sessione tra quelle in sleeping

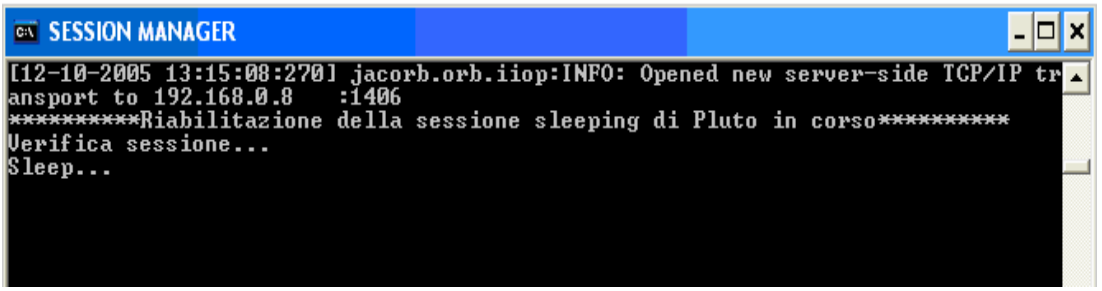


Fig. VI.15 Riattivazione della sessione precedentemente in sleepi ng

VI.2.3 IL SERVIZIO DI MESSAGGISTICA

Nel momento in cui il servizio batch,nel nostro caso il compressore di file,termina la sua esecuzione,viene generato un messaggio di notifica per l'utente.La costruzione del messaggio avverrà indipendentemente dalla presenza o meno dell'utente nell'ambiente:di conseguenza la presenza di nuovi messaggi sarà visualizzata immediatamente nella pagina privata dell'utente,se quest'ultimo in quel momento si trova collegato ad *UbiSystem*.In caso contrario la visualizzazione avverrà non appena l'utente rientra e si autentica (figura VI.16).

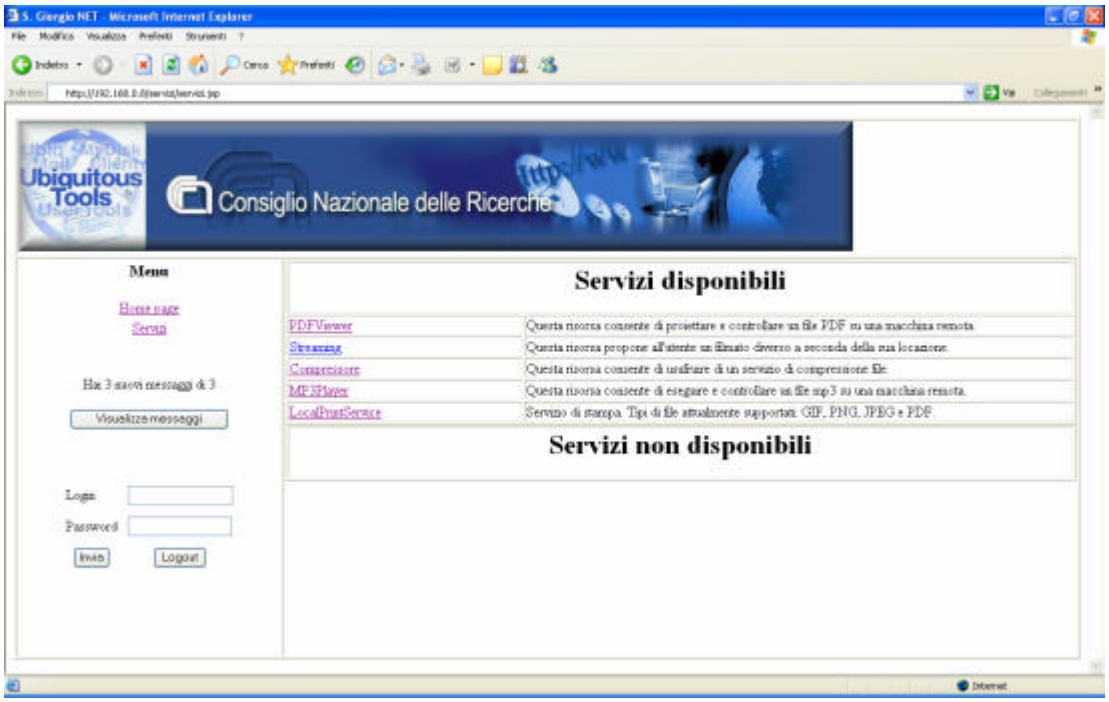


Fig. VI.16 Notifica delle presenza di nuovi messaggi

Scegliendo di visualizzare i messaggi si accede alla pagina in cui viene mostrato quali sono i servizi che hanno terminato correttamente l' esecuzione.I servizi sono identificati non solo dal loro nome ma anche dai file di input utilizzati dall'utente,per consentire a quest'ultimo di distinguere le varie esecuzioni che ha avviato di uno stesso servizio.Trattandosi di un prototipo si è preferito mostrare anche un identificativo univoco per ogni esecuzione.Nel caso del nostro esempio di compressore file tale identificativo può apparire poco utile.Si potrebbe rivelare invece di maggiore utilità pratica nel caso di applicazioni che richiedono in ingresso un numero molto elevato di file:può allora rivelarsi appropriato fornire questo identificativo all'atto dell'avvio del servizio,in maniera tale che l'utente possa poi utilizzarlo per orientarsi nel distinguere le esecuzioni terminate. Ovviamente in ogni messaggio è fondamentale anche il campo che fornisce un link al risultato dell'elaborazione,tramite il quale è possibile direttamente prelevare i file di interesse (figura VI.17).

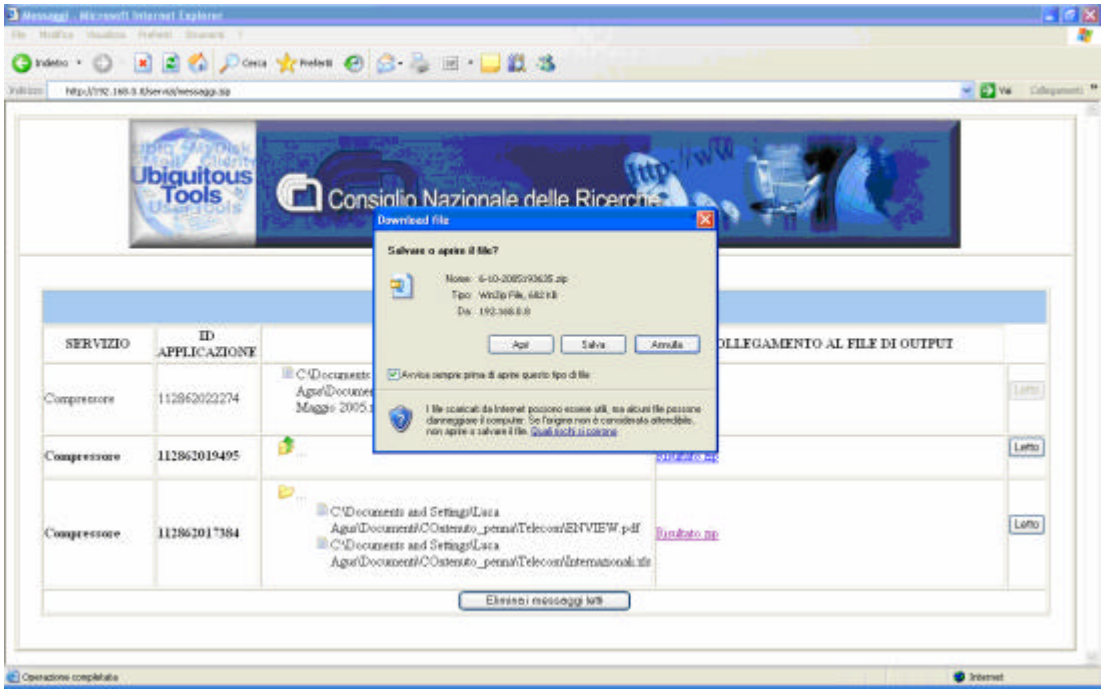


Fig. VI.17 Pagina dei messaggi:l'utente può prelevare il risultato dell'elaborazione

E' inoltre possibile gestire i propri messaggi: nel momento in cui l'utente accede al risultato della propria elaborazione,ma vuole ancora avere memoria della notifica può scegliere di cliccare sul pulsante *"Letto"*, che andrà a catalogare il messaggio in questione come messaggio vecchio. Successivamente tale messaggio sarà ancora riproposto,fin quando l'utente non sceglie di eliminare i messaggi già letti,scegliendo nella pagina l'opzione opportuna (figura VI.18).

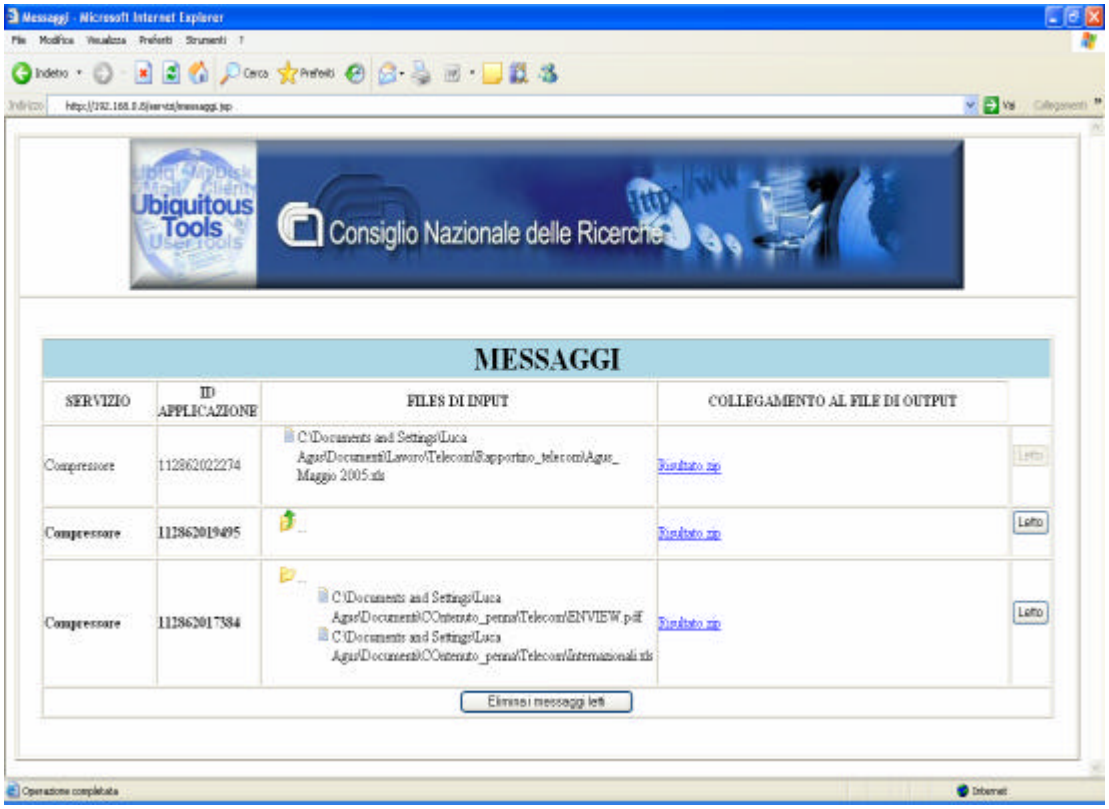


Fig. VI.18 Gestione dei messaggi da parte dell'utente

Ovviamente si può scegliere la politica più opportuna con cui gestire la conservazione dei messaggi. Si può ad esempio scegliere di conservarli secondo le scelte dell'utente entro un certo tempo stabilito e di provvedere a sollecitare alla cancellazione quando l'utente dovesse avere una casella di messaggi troppo piena.

Allo stesso modo si dovrà conservare per un certo tempo il risultato delle elaborazioni:trascorso tale periodo i risultati non saranno più disponibili e per riottenerli si dovrà rieseguire l'applicazione. Le politiche con cui gestire queste eventualità saranno fortemente dipendenti dal tipo di servizio batch che si intende integrare in *UbiSystem*.

VI.2.4 MESSAGGI D'ERRORE

Se scade la sessione o in caso di altri problemi verrà visualizzata una generica pagina d'errore (figura VI.19).

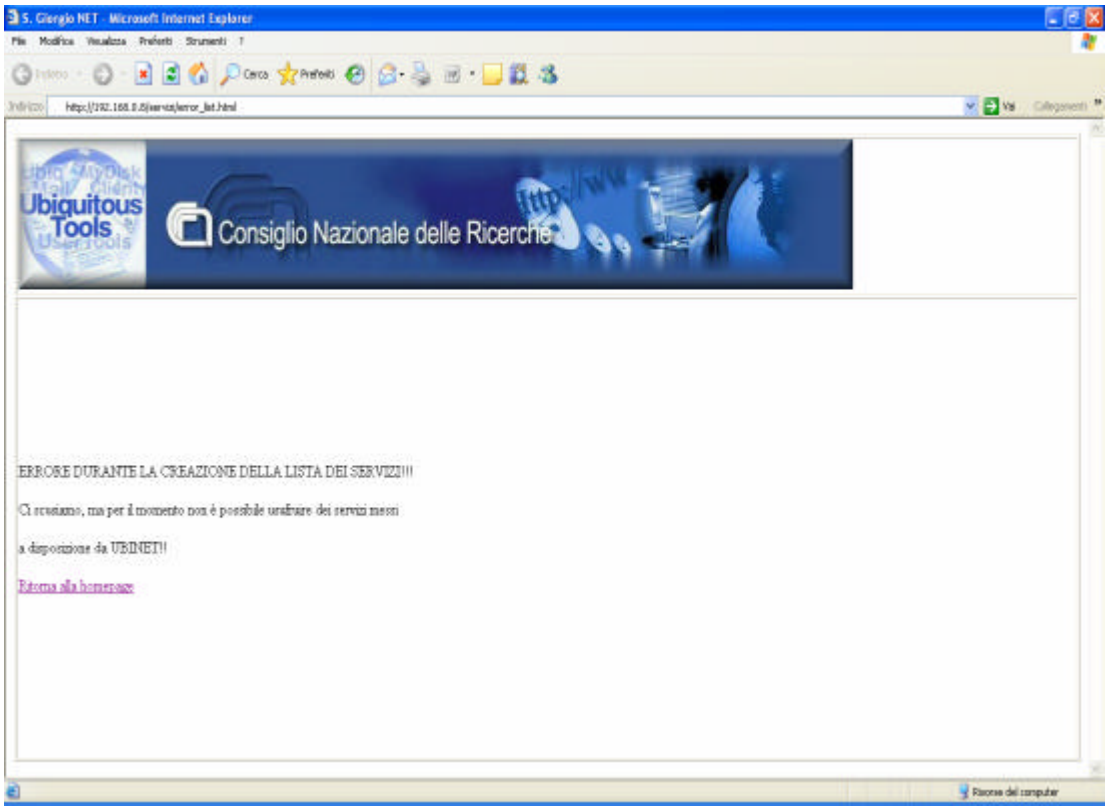


Fig. VI.139 Pagina d'errore

CONCLUSIONI E SVILUPPI FUTURI

Il prototipo implementato adotta tecnologie web-based: l'interazione con l'ambiente e la ricerca delle risorse disponibili al suo interno avviene per mezzo di un semplice browser.

Per la realizzazione dei servizi sono state utilizzate le emergenti tecnologie dei Web Services, il cui punto di forza è di utilizzare un set base di protocolli disponibili ovunque, garantendo l'interoperabilità tra piattaforme diverse e mantenendo, comunque, la possibilità di utilizzare protocolli più avanzati e specializzati per effettuare compiti specifici.

Inoltre, l'approccio seguito ha visto l'impiego di elementi del Web Semantico. In particolar modo, si è fatto ricorso all'uso di ontologie per garantire l'interoperabilità sintattica e semantica fra le diverse entità presenti nell'ambiente e per fornire dei meccanismi avanzati di condivisione delle informazioni e di discovery dei servizi.

Il nuovo prototipo di UbiSystem include, ora, dei componenti che si occupano di applicare delle politiche e meccanismi di gestione dei servizi batch. Un utente, munito di dispositivo mobile, può muoversi nell'ambiente ed usufruire di questo nuovo tipo di servizi integrati in maniera facile ed intuitiva.Come per gli altri tipi di applicazione,anche per queste la disponibilità di accesso dipenderà dalle caratteristiche del device con cui ci si collega e dai diritti d'accesso.

Per fare in modo di testare le nuove funzionalità introdotte è stato necessario apportare delle modifiche ai servizi di sistema esistenti.

Ci sono problemi a cui il sistema implementato ancora non fornisce una soluzione.

Ad esempio non è stato ancora risolto il problema di come un dispositivo possa fornire la sua descrizione in maniera automatica e dettagliata, quando entra a far parte dell'ambiente. Tale descrizione è infatti indispensabile per determinare il set di servizi compatibili con il dispositivo che l'ambiente può offrire oppure per determinare i servizi che il dispositivo stesso può offrire nel momento in cui entra a far parte dell'ambiente, considerazione, quest'ultima, di rilievo se si pensa anche al caso in cui è l'utente che entra nell'ambiente e vuole esporre un proprio servizio.

L'attuale soluzione prevede il prelievo delle informazioni contenute nell'header HTTP nel momento in cui viene effettuata una richiesta e la generazione di una descrizione di massima. Su questa strada si sta muovendo il World Wide Web Consortium che sta già affrontando questo genere di problematiche.

In questo lavoro di tesi non sono state affrontate questioni riguardanti la sicurezza e la privacy delle informazioni scambiate: aspetti, che nei contesti di pervasive computing, diventano di fondamentale importanza.

Studi futuri potranno inoltre essere incentrati ad individuare quali siano le più valide soluzioni per garantire la permanenza dei dati relativi al sistema e agli utenti,aspetto da cui la realizzazione del prototipo ancora prescinde. A questa problematica seguirà quella di attuare procedure di recovery nel caso di indisponibilità o guasto di uno o più componenti di *UbiSystem*.

Concludendo, l'architettura proposta rappresenta soltanto un ulteriore passo avanti verso la realizzazione di un ambiente pervasivo completo; molti sono gli sforzi che ancora si devono compiere e numerosi restano gli ambiti in cui la ricerca dovrà muoversi affinché l'*Ubiquitous Computing* diventi una realtà.

BIBLIOGRAFIA

[1] Anind K.D., *"Providing Architectural Support for Building Context-Aware Application"*, PhD thesis, Georgia Institute of Technology, 2000.

[2] Apache Axis - Web Services, *http://ws.apache.org/axis/*.

[3] Apache Jakarta Project, *http://jakarta.apache.org/*.

[4] Banavar G., *"Challenges in Design and Software Infrastructure for Ubiquitous Computing Application"*, IBM TJ Watson Research Center, University of Zurich.

[5] Barbeau M., *"Mobile, Distributed and Pervasive Computing"*, Handbook of Wireless Networks and Mobile Computing, John Wiley and Sons, Inc., Febbraio 2002.

[6] Barbieri L., *"Web Services: Protocolli e Strumenti, Interoperabilità ed evoluzioni future"*, 2003.

[7] Berners-Lee T., *"Sematic Web road map"*, http://www.w3.org/DesignIssues/Semantic.html.

[8] Finin T., Chen H., *"An Ontology for Context Aware Pervasive Computing Environments"*, University of Mariland, Baltimore County, USA, 2003.

[9] Chen H., Finin T., *"Semantic Web in the Context Broker Architecture"*, University of Mariland, Baltimore County, USA, 2004.

[10] Cooltown Homepage, *http://www.cooltown.com*.

[11] DAML-S, http://www.daml.org/services

[12] DARPA Agent Markup Language Homepage, *http://www.daml.org/*

[13] Gellersen H.W., Davies N., *"Beyond Prototypes: Challenges in Deploying Ubiquitous Systems"*, IEEE Pervasive Computing, Gennaio-Marzo 2002.

[14] GAIA Homepage, *http://choices.cs.uiuc.edu/gaia/*.

[15] Grimm R. et al., *"System Directions for Pervasive Computing"*, University of Washington, 2000.

[16] Harms D., *"JSP Servlet e MySql"*, Mc Graw-Hill Editore, Milano, 2002.

[17] Rakotonirainy A., Henricksen K., Indulska J., *"Infrastructure for Pervasive Computing: Challenges"*, School of Computer Science and Electrical Engineering, The University of Queensland, 2000.

[18] IEEE Pervasive Computing Homepage, *http://www.computer.org/pervasive/*.

[19] Joshi A., Kagal L., Korolev V., Avancha S., Finin T., Yesha Y., *"Centaurus: An Infrastructure for Service Management in Ubiquitous Computing Environments"*, Dept. of Computer Science and Electrical Engineering, University of Maryland Baltimore Conty, USA, 2002.

[20] Barton J., Kindberg T., *"A web-based nomadic computing system"*, White paper, HP Labs, cooltown.hp.com, 2001.

[21] Fox A., Kindberg T., *"System Software for Ubiquitous Computing"*, IEEE Pervasive Computing, Gennaio-Marzo 2002.

[22] Maffioletti F., *"Requirements for an Ubiquitous Computing Infrastructure"*, Cultura Narodov Prichernomoria Journal vol.3. Simferopol, Ukraine, Settembre 2001.

[23] Magnanini P., *"Sistemi di discovery: La soluzione Jini"*, Università degli Studi di Bologna, 2001.

[24] Capra L., Mascolo C., Emmerich W., *"Mobile Computing Middleware"*, Dept. of Computer Science, University College London, 2002.

[25] Sturm P., Mattern F., *"From Distributed Systems to Ubiquitous Computing"*, Department of Computer Science, Switzerland - Germany, 2002.

[26] McGrath R.E., Ranganathan A., Campbell R.H., Mickus M.D., *"Use of Ontologies in Pervasive Computing Environments"*, Dept. of Computer Science, University of Illinois, Urbana-Champaign, USA, Aprile 2003.

[27] Project Aura Homepage, http://www-2.cs.cmu.edu/~aura/.

[28] Ranganathan A., Campbell R.H., *"A Middleware for Context-Aware Agents in Ubiquitous Computing Environments"*, Department of Computer Science, University of Illinois at Urbana-Champaign, USA, 2002.

[29] Hess C., Roman M., Cerqueira R., Ranganathan A., Campbell R.H., Nahrsted K., *"A Middleware Infrastructure for Active Spaces"*, IEEE Pervasive Computing, Ottobre-Dicembre 2002.

[30] Mukherjee A., Saha D., *"Pervasive Computing: A Paradigm for the 21st Century"*, IEEE Computer Society, Marzo 2003.

[31] Saracco R., *"Ubiquitous Computing"*, Mondo digitale, n.3, Settembre 2003.

[32] Satyanarayanan M., *"Pervasive Computing: Vision and Challenges"*, IEEE Personal Communication, Agosto 2001.

[33] UDDI.org, *Universal Description, Discovery and Integration of Web Services*, http://uddi.org/.

[34] Baldoni R., Virgillito A., *"Una introduzione alla architettura CORBA con elementi avanzati di interazione client/server in Java"*, Dip. di Informatica e Sistemistica, Università di Roma "La Sapienza".

[35] Wallbank N., *"A Requirements Analysis of Infrastructure for Ubiquitous Computing Environments"*, Computing Department, Lancaster University, 2002.

[36] Weiser M., *"The Computer for the 21st Century"*, Scientific Am., Settembre 1991; riedito dall'IEEE Pervasive Computing, Gennaio-Marzo 2002.

[37] World Wide Web Consortium, *Extensible Markup Language (XML) 1.0 (Third Edition)*, http://www.w3.org/TR/2004/REC-xml-20040204/, Febbraio 2004.

[38] World Wide Web Consortium, *OWL Web Ontology Language Semantics and Abstract Syntax*, http://www.w3.org/TR/2004/REC-owl-semantics-20040210/, Febbraio 2004.

[39] World Wide Web Consortium, *Resource Description Framework*, http://www.w3.org/RDF/.

[40] World Wide Web Consortium, *SOAP Version 1.2 Part 1: Messaging Framework*, http://www.w3.org/TR/soap12-part1/, Giugno 2004.

[41] World Wide Web Consortium, *Web Services Architecture*, *http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/*, Febbraio 2004.

[42] World Wide Web Consortium, *Web Services Description Language (WSDL) 1.1*, http://www.w3.org/TR/wsdl, Marzo 2001.

[43] Zhu F., Mutka M., Ni L., *"Classification of Service Discovery in Pervasive Computing Environments"*, Michigan State University, East Lansing, 2002.

[44] Stefano Russo, Carlo Savy, Domenico Cotroneo, Antonio Sergio, *"Introduzione a CORBA"*, McGraw-Hill, 2002.

[45] Pradeep Gore, Ron Cytron Douglas Schmidt, Carlos O'Ryan, *"Designing and Optimizing a Scalable CORBA Notification Service"*.

[46] Horrocks, Ian and Sattler, "CORBA-FaCT", http://www.cs.man.ac.uk/~horrocks/FaCT/CORBAFsCT.html

[47] David Garlan, Bradley Schmerl, *"Component-Based Software Engineering in Pervasive Computing Environments"*, Carnegie Mellon University.

[48] JacORB, http://www.jacorb.org

[49] João Pedro Sousa, David Garlan, *"Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments"*, School of Computer Science Carnegie Mellon University, August 2002.

[50] Raffaella Laraia, Tesi di Laurea *"Metodologie e politiche di gestione dei servizi in UbiSystem"*,Corso di laurea in Ingegneria Informatica,Università "Federico II" Napoli, anno accademico 2003/2004.

[51] Francesca Tullio, Tesi di Laurea *"Politiche e servizi di gestione di utenti mobili nell'ambiente UbiSystem"*,Corso di laurea in Ingegneria Informatica,Università "Federico II" Napoli, anno accademico 2003/2004.

[52] Marco Avellino, Tesi di Laurea *"I servizi di location e di context all'interno dell'ambiente UbiSystem"*, Corso di laurea in Ingegneria Informatica,Università "Federico II" Napoli, anno accademico 2003/2004.

[53] Massimo Esposito,Tesi di Laurea *"Analisi e caratterizzazione di sistemi di Pervasive Computing: un approccio ontology-based per la risoluzione di problemi semantici*, Corso di laurea in Ingegneria Informatica ,Università "Federico II" Napoli,anno accademico 2002/2003.