POLITECNICO DI TORINO

III Facoltà di Ingegneria
Corso di Laurea in Ingegneria delle Telecomunicazioni

Tesi di Laurea

Compilatore CPL per reti SIP con tecnologie JAVA avanzate

Relatore:
Maurizio Morisio

Candidato:
Marco De Nittis

Settembre 2003

Indice

I Introduzione teorica **1**

1 Introduzione **2**

1.1 Contesto del lavoro della tesi . . . 2

1.2 Obiettivi . . . 3

1.3 Organizzazione del testo . . . 5

2 Next Generation Networks **8**

2.1 Introduzione . . . 8

2.1.1 OSA Parlay . . . 9

2.1.2 JAIN API . . . 11

2.2 Tecnologie attualmente esistenti per la personalizzazione dei servizi . 13

2.2.1 CPL (Call Processing Language) . . . 13

2.2.2 SCML (Service Creation Markup Language) . . . 14

2.2.3 XTML (eXtensible Telephony Markup Language) . . . 15

2.3 SIP (Session Initiation Protocol) . . . 16

2.3.1 Caratteristiche del protocollo e Architettura di funzionamento 16

2.3.2 Comandi SIP . . . 20

2.3.3 Informazioni sul protocollo . . . 26

2.3.4 Scenari di funzionamento . . . 28

2.4 Java e SIP . . . 30

2.4.1 JAIN SIP . . . 31

2.4.2 JAIN SIP Lite . . . 32

2.4.3 JCC (Java Call Control) . . . 33

2.4.4 SIP Servlet . . . 34

3 Linguaggio CPL **36**
3.1 Introduzione 36
3.2 Struttura 37
3.2.1 Struttura generica di un elemento CPL 38
3.3 Elementi 40
3.3.1 Nodi di switch 40
3.3.2 Nodi di modifica delle destinazioni 47
3.3.3 Nodi di segnalazione 49
3.3.4 Nodi generici 52
3.3.5 Altri nodi 53
3.3.6 Comportamento predefinito 54
3.4 Esempi 55
3.4.1 Inoltro semplice della chiamata 55
3.4.2 Redirezione della chiamata 55
3.4.3 Inoltro con cambio di indirizzo 55
3.4.4 Cambio di indirizzo in funzione del ora di arrivo 56
3.4.5 Uso del tag *proxy* 57
3.4.6 Uso delle *subaction* 58

II Realizzazione del progetto **60**

4 Architettura del servizio **61**
4.1 Criteri di progettazione 61
4.1.1 Obiettivi 61
4.1.2 Soluzione 62
4.1.3 Motivazioni 63
4.2 Architettura 64
4.2.1 ProfileManager 64
4.2.2 ProfileServerManager 66
4.2.3 ProfileRepository 68
4.3 Gestione del profilo 69
4.3.1 Class Loading dinamico 70
4.3.2 Serializzazione 73
4.3.3 Soluzione adottata 74

4.4 Scenari d'uso 77
4.4.1 Creazione di un nuovo profilo 77
4.4.2 Modifica di un profilo esistente 78
4.4.3 Gestione di una prima chiamata 78
4.4.4 Gestione delle chiamate successive 80

5 Descrizione dell'implementazione 82
5.1 Scelte di design 82
5.2 Struttura dei componenti 83
5.2.1 Package *it.polito.cpl* 83
5.2.2 Package *it.polito.cpl.client* 89
5.2.3 Package *it.polito.cpl.server* 93
5.2.4 Package *it.polito.cpl.parser* 95
5.2.5 Package *it.polito.cpl.classcompiler* 99
5.2.6 Package *it.polito.cpl.classcompiler.parsercompiler* 100
5.2.7 Package *it.polito.cpl.util* 104
5.3 Integrazione con il proxy 105
5.4 Scenari 106
5.5 Prestazioni 108

6 Conclusioni 109
6.1 Risultati ottenuti e problemi riscontrati 109
6.1.1 Incompatibilità da parte di determinati User Agent 110
6.1.2 Errori nella chiusura delle sessioni SIP 111
6.1.3 Gestione incompleta dei timeout del proxy 112
6.1.4 Implementazione incompleta del linguaggio CPL 112
6.1.5 Errori nella creazione guidata dei profili (wizard) 113
6.2 Sviluppi futuri 114

III Appendici 116

A Procedure di configurazione 117

B Struttura del linguaggio CPL 131

C Strumenti utilizzati **140**

D Acronimi **141**

Bibliografia **144**

Ringraziamenti I

Parte I

Introduzione teorica

Capitolo 1

Introduzione

1.1 Contesto del lavoro della tesi

Il lavoro di questa tesi si colloca, all'interno dell'ambito delle telecomunicazioni, in quelle che vengono definite come reti della prossima generazione (NGN - Next Generation Network). Tutto il lavoro di ricerca svolto in questa realtà si muove essenzialmente in due direzioni: la prima è l'innovazione, in cui la ricerca tende a trovare e proporre nuove soluzioni tecnologiche, la seconda è l'ampliamento dei servizi, in cui, basandosi sugli strumenti e le infrastrutture già esistenti, il lavoro punta alla creazione di nuove funzionalità e di *framework* per lo sviluppo più rapido di questi servizi.

Per capire meglio questo tipo di realtà si pensi al contesto delle reti cellulari, in cui da una parte c'è la tendenza ad utilizzare nuove tecnologie, come ad esempio UMTS, e dall'altra parte, verso la maggioranza degli utenti, il mercato si allarga fornendo una vasta gamma di servizi che sfruttano le tecnologie già esistenti e diffuse.

Il lavoro della presente tesi si muove verso questa seconda direzione, in quanto le potenzialità ancora latenti in questo tipo di realtà lascia un ampio margine di libertà ai possibili sviluppi. In questo ambito si parla di reti di telecomunicazioni, che possono essere in primo luogo telefoniche tradizionali (PSTN e ISDN), reti cellulari (GSM, GPRS, UMTS), reti dati (IP, ATM) che possono a loro volta sia supportare i servizi tradizionali di scambio dati, sia occuparsi del trasporto di informazioni multimediali e quindi fondersi nelle realtà di telefonia.

Una delle maggiori tendenze in questo campo è la convergenza: con questo termine si intende l'integrazione di diverse reti in un framework comune che offra interfacce aperte e standard ai servizi; di conseguenza gli stessi servizi diventano più portabili, in quanto slegati dai tipi di reti e dai protocolli sottostanti.

La prospettiva è quella di avere una rete omogenea, in cui è possibile passare da un tipo di sistema trasporto ad un altro in maniera il più possibile trasparente, dal punto di vista dell'utente finale e dello sviluppatore di servizi. In questo modo tutta la complessità del passaggio da una tecnologia ad un'altra viene gestita dall'infrastruttura sottostante. Questo è lo stesso principio su cui si è mossa l'architettura di internet, che è formata da un gran numero di tipi di rete differenti, ma nonostante ciò, la maggioranza delle applicazioni (o utenti) non si deve preoccupare di conoscere l'effettiva struttura sottostante per poter funzionare o lavorare.

1.2 Obiettivi

L'obiettivo di questa tesi è lo sviluppo di un'architettura per la gestione dei profili utente di telefonia all'interno di una rete che si appoggia al protocollo SIP[1], con prestazioni superiori ai prodotti già esistenti, sfruttando tecnologie Java avanzate.

In questo modo ogni utente ha la possibilità di definire il comportamento personalizzato che il sistema deve assumere all'arrivo di una chiamata; così si crea un meccanismo semplice, ed allo stesso tempo flessibile, adatto alla personalizzazione del servizio.

Per la gestione delle chiamate, basata sul protocollo SIP, si è utilizzato un proxy SIP esistente open source che fornisce un'implementazione certificata dello standard JAIN SIP, così da potersi concentrare esclusivamente sulla gestione dei profili. Per la definizione dei profili è stato utilizzato il linguaggio CPL (Call Processing Language) che permette di definire il comportamento che deve venire seguito all'arrivo di una chiamata (telefonica o VoIP) in maniera estremamente versatile e flessibile.

Per raggiungere questi risultati, inizialmente si sono provate diverse soluzioni innovative per la creazione dei profili, basandosi sull'idea di associare ad ogni profilo un oggetto Java, le due idee prese in considerazione sono state: la creazione di oggetti composti da elementi che rappresentavono le operazioni presenti nella definizione dei

[1] Session Initiation Protocol, si veda il capitolo 2

profili, che potevano venir salvati attraverso il meccanismo della serializzazione Java, e la creazione di classi ad hoc per ogni profilo costruite dinamicamente in base alla definizione del profilo. E' stata scelta quest'ultima soluzione per motivi di flessibilità e di prestazioni.

Questa decisione, per quel che riguarda l'utilizzo di queste classi, porta alla necessità di costruire dei meccanismi di caricamento delle classi Java diverso da quello tradizionale, in quanto le implementazioni dei profili non sono presenti sul disco su cui è in esecuzione il componente, ma vengono create a tempo di esecuzione. Quindi sono stati ideati dei meccanismi di caricamento delle classi dinamici, cioè in grado di rendere utilizzabili dal sistema degli oggetti java non disponibili già prima dell'avvio del programma, ma creati anche successivamente. Oltre a questo c'è anche la possibilità di modificare dei profili già esitenti, quindi sostituire l'implementazione Java di un determinato profilo, senza dover riavviare il proxy che l'utilizza.

La fase successiva è stata un'attenta analisi del linguaggio CPL, che essenzialmente è un dialetto XML, in quanto, di fatto, si è resa necessaria la costruzione di un compilatore CPL verso Java che permettesse di tradurre degli script CPL, in codice sorgente Java. Insieme a questa, si è dovuto comprendere l'architettura del proxy utilizzato, dato che molte operazioni CPL sono legate ai meccanismi di segnalazione del protocollo utilizzato (in questo caso SIP) e gestite quindi dal proxy.

Quindi si è iniziato a progettare l'architettura utilizzata dal sistema, dividendo le funzionalità tra due componenti principali: il **ProfileManager** e il **ProfileServerManager**, che si occupano rispettivamente di utilizzare i profili all'arrivo delle chiamate, e di creare i profili a partire dai sorgenti CPL. Si puntato a rendere le parti del sistema il più possibile indipendenti e intercambiabili, così che questo possa venir esteso e modificato con facilità.

Uno dei punti chiave nello sviluppo del sistema sono state le prestazioni, in particolare per la fase di attivazione della chiamata, in quanto il tempo di risposta risulta un parametro critico in un sistema di telefonia.

A questo fine il sistema è stato progettato per effettuare la compilazione del del profilo soltanto quando viene inserita la modifica, salvando l'oggetto così ottenuto. Inoltre la fase di creazione del profilo avviene in maniera slegata rispetto alla ricezione delle chiamate, in questo caso il tempo di creazione del profilo non influenza il tempo di attivazione della chiamata.

Per ridurre ancora i colli di bottiglia, si è creato un meccanismo di cache in modo

che i profili vengano richiesti solo una volta dal proxy, e poi vengano mantenuti memorizzati; in ogni caso è presente un meccanismo che elimina dalla cache un profilo nel caso questo sia stato modificato.

La parte più complessa del lavoro è stata quella di strutturare il compilatore CPL, in quanto le classi Java generate per i profili sono strettamente dipendenti dal'architettura del proxy, e di conseguenza si è trattato di tradurre delle istruzioni generiche (dal linguaggio CPL) in frammenti di codice che interagissero con il proxy e la sua architettura. Che è di tipo asincrono, dato che le chiamate telefoniche e le relative risposte non avvengono istantaneamente o con un ordine rigoroso, ma devono venir elaborate gestendo gli eventi che queste generano.

A questo punto si è passati alla scrittura del codice vera propria, e relativa documentazione, che ha coinvolto, oltre all'utilizzo della piattaforma Java classica, anche altre tecnologie come JSP (Java Servlet Page) e JDBC (Java Database Connectivity), per la creazione di interfaccie utente attraverso browser web e interazione con database su cui vengono memorizzati i profili.

L'ultima fase è stata il test di tutto il sistema, in parte già eseguito singolarmente per alcune componenti durante lo sviluppo; questa fase finale è risultata un'operazione molto delicata in quanto, da una parte non era possibile avere accesso diretto alle parti di codice dei profili che venivano creati durante l'esecuzione, e quindi non era possibile seguire direttamente lo svolgersi delle operazioni che accadevano all'interno di esse; dall'altra parte per la natura asincrona del sistema stesso, dato che lavora sugli eventi generati dai messaggi che arrivano al proxy.

L'ultimo passo è stato quello di curare la distribuzione dei componenti creati, creando degli script automatici che permettano la compilazione e la costruzione dell'applicazione e dei relativi archivi, pronti così per essere utilizzati direttamente.

1.3 Organizzazione del testo

Questo elaborato è diviso in due parti: nella prima viene proposta una panoramica teorica sul contesto e le tecnologie utilizzate, in particolare questa è divisa in tre capitoli.

Nel primo capitolo di introduzione viene fatta una breve presentazione del lavoro svolto a partire dalla sua collocazione all'interno del panorama IT attuale, gli obiettivi perseguiti e il percorso seguito per raggiungerli.

Il secondo capitolo presenta in maniera dettagliata e approfondita il contesto in cui si colloca questo lavoro, le tecnologie e gli standard, attuali e in fase di lavorazione, nel mondo delle infrastrutture per la creazione di servizi, ed inoltre viene fatta un'analisi accurata del protocollo SIP vista la sua importanza all'interno del lavoro svolto in questa tesi.

Nel terzo capitolo viene analizzato in tutte le sue parti il linguaggio CPL, presentando la struttura e le motivazioni che hanno portato alla sua definizione, oltre a questo viene approfondita la specifica di ogni suo comando con i relativi parametri. Il capitolo termina con una serie di esempi di profili scritti in questo linguaggio per poter comprendere meglio il suo funzionamento.

La seconda parte di questo elaborato si occupa di studiare la realizzazione del sistema.

All'interno del quarto capitolo viene presentata l'architettura del sistema realizzato, in particolare si è partiti mostrando il problema, la soluzione e le motivazioni per le scelte prese. Attraverso l'uso di diagrammi UML vengono mostrate i componenti e le interfacce presenti con le relative funzionalità e i rapporti che intercorrono tra loro; inoltre viene presentata un'analisi delle scelte tecnologiche seguite (come il caricamento dinamico delle classi Java), ed infine vengono presentati una serie di scenari d'uso in cui è possibile comprendere il funzionamento complessivo del sistema.

Il quinto capitolo è un'analisi dell'implementazione del sistema, in cui vengono analizzate tutte e le classi e le interfacce, con i relativi metodi principali e le politiche di funzionamento. Questa trattazione viene fatta seguendo la struttura dei *package* Java, in quanto questi, oltre a rappresentare una divisione di collocazione delle classi, corrispondo anche ad un'effettiva divisione logica degli elementi del sistema. Oltre a questo viene spiegato il meccanismo di integrazione con il proxy, sviluppato da NIST, al quale si appoggia il sistema.

Nell'ultimo capitolo vengono presentate le conclusioni di questo lavoro, cioè i risultati ottenuti e i problemi riscontrati nell'implementazione analizzando i sintomi, individuando le motivazioni e ipotizzando una possibile soluzione. Oltre a ciò vengono presentate delle possibili prospettive future per questo lavoro, sia come miglioramenti rispetto allo stato attuale, sia considerando le possibli applicazioni per cui può essere adattato.

Infine nelle appendici vengono mostrate le procedure di configurazione ed installazione, il documento formale in cui viene definita la struttura del linguaggio CPL e una panoramica degli strumenti software utilizzati per la realizzazione di questo lavoro.

Capitolo 2

Next Generation Networks

2.1 Introduzione

Una delle parole chiave che, negli ultimi tempi, ha guidato lo sviluppo tecnologico nel settore IT, per quel che riguarda le telecomunicazioni e le reti, è sicuramente convergenza. Moltissimi sforzi sono stati fatti per la creazione di nuovi standard e tecnologie che fossero mirate alla creazione e alla gestione dei servizi all'interno di queste realtà. Tutto questo prende il nome di Next Geneneration Networks (NGN).

Per la precisione viene definita NGN da [2] la concezione con cui possono venire sviluppate delle reti di telecomunicazione, le quali, a seguito di una divisione formale in strati e l'opportuno uso di interfacce standard di comunicazione, offrono ai gestori, agli sviluppatori e agli operatori di telecomunicazioni, una piattaforma per creare e gestire servizi innovativi, che può evolvere in modo graduale.

Lo scopo per cui è nato questo paradigma è ridurre i costi relativi alla gestione e l'innovazione delle infrastrutture già presenti, quindi semplificare il lavoro e ridurre gli sforzi necessari per il mantenimento di servizi esistenti e lo sviluppo di altri completamente nuovi.

Questo si traduce nella pratica nella definizione di standard di funzionamento per rendere il più possibile uniforme la struttura dell'architettura delle reti, nell'utilizzo di framework aperti per quel che riguarda il lato dello sviluppo, e la scelta di tecnologie adeguate per l'implementazione vera e propria. Tutto questo porta a scrivere ad esempio un unico servizio portabile e riutilizzabile su differenti tipi di rete, invece che seguire l'approccio classico, dove, ad esempio, un servizio scritto per

funzionare su PSTN doveva essere convertito per poter funzionare su VoIP (Voice Over IP) o ISDN.

Il progetto EURESCOM P1109 [3] si è occupato di definire dei criteri di valutazione nell'ambito delle NGN e ha proposto un'architettura di riferimento su cui potersi basare nella classificazione delle diverse soluzioni esistenti.

Le caratteristiche chiave di un'architettura di questo tipo sono [5]:

- la divisione netta dei servizi e delle reti in strati sovrapposti, e la conseguente separazione tra le funzionalità dei servizi e le funzionalità di trasporto
- la possibilità di creare, attivare e gestire qualunque tipo di servizio (utilizzando API mirate alla creazione di servizi sfruttando qualunque tipo di dati multimedialidi tipo audio o video, dati e combinazioni di questi)
- la possibilità di avere il controllo sulle politiche di accesso, sulle sessioni, sulle risorse, sulla gestione dei servizi e sulla sicurezza, che possono essere distribuite in tutta l'infrastruttura comunicando attraverso delle interfacce aperte.
- il supporto sia per terminali (ed in generale l'hardware) già esistenti, sia per nuovi terminali predisposti per interagire con i vari standard delle NGN.

Un altro elemento che necessita di una considerazione particolare è la gestione del QoS (Quality of Service o qualità del servizio che corrispondono a parametri di larghezza di banda, ritardi, perdite, ecc. . .) per quel che riguarda i servizi in real-time, dato che all'interno di una realtà in cui i servizi della rete corrispondono ad un offerta commerciale e quindi un effettivo guadagno, è importante poter controllare la qualità e il tipo prodotto venduto anche dal punto di vista della qualità.

Verrano ora analizzate una serie di realtà esistenti nell'ambito delle infrastrutture per la creazione e la gestione delle NGN, esaminando più nel dettaglio quelle che sono state prese in considerazione nello sviluppo di questa tesi.

2.1.1 OSA Parlay

Il consorzio Parlay [6] si è occupato di definire delle API[1] aperte, slegate da qualunque tecnologia particolare, per lo sviluppo di applicazioni che possono operare in

[1] Application Programming Interface, vengono definite in questo modo le librerie che vengono messe a disposizione degli sviluppatori per utilizzare determinate funzionalità, in questo caso si parla di funzionalità verso i servizi di rete.

differenti ambienti su piattaforme per creazione e le gestione di servizi. Il risultato di questo lavoro è l'Open Service Access (OSA), una serie di API che gestiscono le comunicazione tra le applicazioni IT e il livello relativo alle telecomunicazioni attraverso delle interfaccie standard e aperte.

Le funzionalità delle reti sono descritte come Service Capability Features (SCF) [4] e possono essere sviluppate da terze parti in maniera indipendente, ognuna di queste implementa gruppi di API OSA (ad esempio API per il controllo delle chiamate e la gestione della mobilità dei terminali, ad esempio wireless e cellulari) in questo modo forniscono l'accesso alle caratteristiche della rete che il produttore vuole rendere disponibili attraverso l'interfaccia OSA.

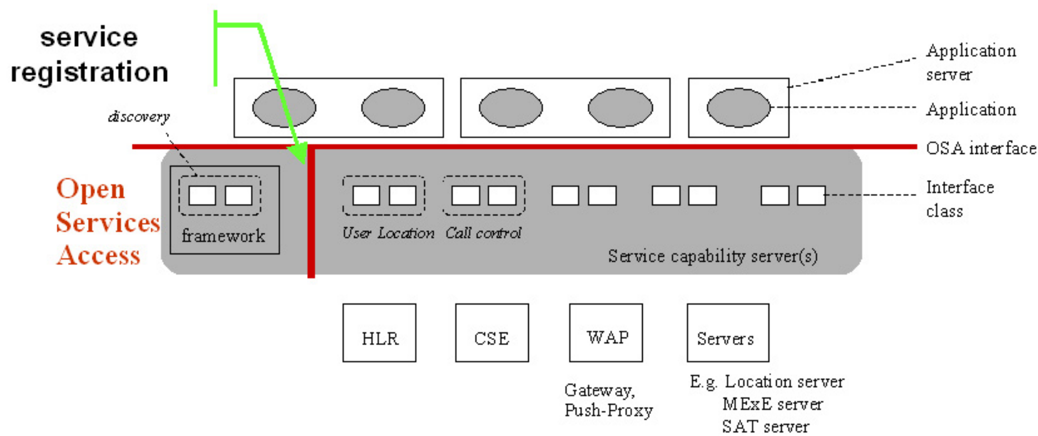

Figura 2.1. Architettura OSA

Il numero di SCF può aumentare gradualmente, perché uno degli obiettivi di Parlay è di fornire un'interfaccia estendibile e scalabile che permetta l'aggiunta di nuove funzionalità nella rete con nuove versione e allo stesso tempo, che questo abbia un impatto minimo nelle applicazioni che usano l'interfaccia OSA.

I punti forti di questa architettura sono:

- Nascondere la complessità della rete, dei protocolli usati e delle implementazioni specifiche dalle applicazioni;
- Essere adatta allo sviluppo di applicazioni utilizzando componenti forniti da terze parti;

- Fornire un accesso sicuro e controllato alle funzionalità di una rete fornite dal gestore o da un fornitore esterno di servizi;
- Rendere disponibili tutte le funzionalità di rete (definite nel punto precedente) fornite dai protocolli ad un maggiore livello di astrazione, ed allo stesso tempo semplificare lo sviluppo dei servizi, ad esempio, combinando servizi già esistenti e integrando alle applicazioni IT.

Questa teconologià ha raggiunto uno stato di maturità notevole, specialmente dopo la fondazione del gruppo di lavoro da parte di grandi enti di standardizzazione come 3GPP CN5, ETSI, Parlay e JAIN con l'obiettivo di far convergere queste tecnologie che sono già simili tuttora.

Sono disponibili diversi prodotti già pronti basati su Parlay/OSA come gateway, framework di sviluppo, application server e applicazioni. La maggior parte delle soluzioni fornite dai produttori sono orientate al funzionamento su reti mobili, molti test sono stati già eseguiti, e diversi operatori stanno pianificando di passare a soluzioni basate su OSA/Parlay già a partire dall'anno corrente.

Le API OSA sono definite come API distribuite, funzionando attraverso meccanismi di elaborazione distribuiti, le versioni CORBA/IDL e Java/RMI sono già tuttora disponibili, mentre le versioni che utilizzano Web Services sono in fase di definizione dal gruppo Parlay-X.

.

2.1.2 JAIN API

JAIN (Java in Advanced Intelligent Networks) è una comunità di aziende creata con lo scopo specifico di definire delle API [9]per raggiungere la portabilità dei servizi, convergenza e accessi sicuri verso reti telefoniche e dati (Internet). Si basa esclusivamente sulla tecnologia Java, ed anche il suo sviluppo segue le specifiche di sviluppo e licenza SUN (JSPA Java Specification Participation Agreement, JCP Java Community Process, SCSL Sun Community Source Code Licensing[2]).

JAIN si occupa di definire due diverse serie di API riguardo le NGN, API verso i protocolli e API verso le applicazioni; le prime specificano le interfaccie verso i

[2]Sono delle procedure aziendali, ideate dalla SUN, che regolano lo sviluppo e la distribuzione di prodotti basati sulla tecnologia Java.

protocolli di reti IP, wireless e telefoniche, la seconda serie di API viene utilizzati per la creazione di servizi fornendo un framework java per l'utilizzo dei protocolli che sono gestiti dal primo gruppi di API.

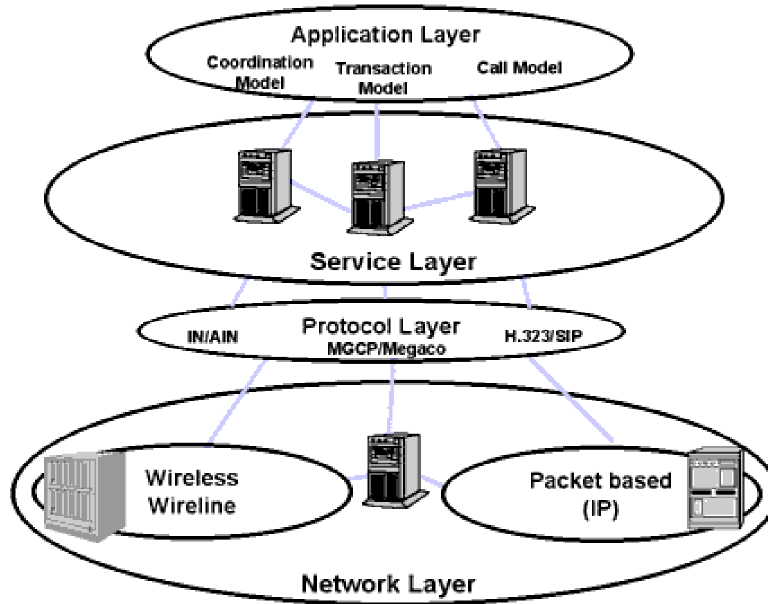

Figura 2.2. Architettura JAIN

Tuttora sono presenti una dozzina di API differenti per la gestione dei protocolli e altrettante a livello applicazione, per quel che riguarda lo scopo di questa tesi è importante citare:

JAIN SIP 1.0 (si veda 2.4.1) fornisce un'interfaccia standard per condividire informazioni tra client e server SIP (Session Initiation Protocol) [2.3], fornendo funzionalità per il la gestione delle chiamate (Call Control) utilizzabili dalle applicazioni. Queste API incapsulano le funzioni delle diverse versioni del protocollo SIP con un'interfaccia standard così da rendere facilemnte utilizzabili le funzionalità di attivazioni delle sessione e trattamento delle chiamate (Call Processing).

JAIN SIP Lite (si veda 2.4.2) definisce un'API più astratta della precedente, di alto livello per accedere allo stack SIP orientato alla creazione rapida di applicazioni. L'API JAIN SIP 1.0 fornisce un'interfaccia di basso livello per

accedere allo stack, quindi necessaria da parte dello sviluppatore una conoscenza abbastanza approfondita del protocollo. Per ovviare a questo è stata definita questa API più semplice che funziona come *wrapper* [3] verso la libreria completa.

Entrambe verranno trattate in maniera più approfondita nel paragrafo 2.4, oltre a queste, come già detto, all'interno di JAIN sono presenti altre tecnologie legate sempre ai servizi delle NGN, ad esempio per la gestione delle presenze, all'instant messaging (messageria istantanea), gestione delle transazioni, dei pagamenti e delle tariffazioni, alla gestione delle applicazioni e dei servizi, alle comunicazioni da e verso reti e terminali wireless appoggiandosi alla piattafoma J2ME (Java 2 Micro Edition).

2.2 Tecnologie attualmente esistenti per la personalizzazione dei servizi

Nell'ambito di questa tesi ci si è focalizzati sulla personalizzazione dei servizi offerti per ogni utente che usufruisce del servizio, per questo motivo è importante avere una panoramica delle realtà attualmente più interessanti su questo orizzonte.

2.2.1 CPL (Call Processing Language)

CPL è un linguaggio di scripting usato per definire come devono venire gestite le chiamate entranti o uscenti in una rete NGN. Questo linguaggio ed è basato su XML [11], ed è stato sviluppato per essere eseguito all'interno di un proxy SIP, o all'interno di un terminale con funzionalità avanzate, per implementare nuovi servizi.

CPL è stato concepito per essere utilizzato da utenti considerati anche non fidati, per inserire i loro servizi sui server SIP. CPL è un linguaggio leggero, efficiente e facile da implementare ed estendibile perché è possibile aggiungere nuove funzionalità personalizzate in modo che, in ogni caso, gli script già esistenti possano continuare a funzionare.

[3] Si definisce wrapper (dall'inglese involucro) un strato software che nasconde al suo interno la complessità delle chiamate ad altri componenti che si occupano dell'eleborazione vera e propria, mentre allo sviluppatore viene fornita un'interfaccia semplice da utilizzare.

CPL permette di lavorare con una delle principali funzionalità di una rete: il Call Control; nell'architettura di riferimento l'interprete CPL può essere ospitato dal Call Server e gli script CPL posso essere caricati dall' Application Server attraverso il protocollo HTTP (GET/POST), oppure possono essere generati dinamicamente (da componenti lato server).

Può essere utilizzato per implementare servizi in differenti scenari: si possono utilizzare script creati direttamente dagli utenti in modo che possano personalizzare la politica della gestione delle proprie chiamate inviando questi al server, oppure gli script possono venir creati dagli amministratori di sistema secondo le preferenze degli utenti, oppure questi possono venir generati da applicazioni web che traducono le richieste in documenti CPL.

Dato che CPL è un linguaggio standard può essere utilizzato da terze parti per creare servizi personalizzati per i clienti, e questi servizi possono venire eseguiti su server propri dei clienti, oppure direttamente dai fornitori di servizi di questi.

CPL è un linguaggio ormai abbastanza maturo ed è completamente specificato [11] benché non sia ancora stata rilasciata la versione definitiva. Esso comunque non permette di generare delle chiamate verso due o più utenti perchè gli script vengono attivati solo ed esclusivamente in seguito ad eventi relativi a chiamate, come la ricezione o l'invio, e non può essere utilizzato per descrivere dei comportamenti complessi, benché il linguaggio in se sia comunque flessibile.

Si possono trovare un buon numero di prodotti commerciali che utilizzano CPL, e probabilmente tutti gli Application Server che lavorano su reti SIP/H323 supportano CPL.

Per un analisi più approfondita si rimanda al capitolo 3 dove viene analizzato nel dettaglio il funzionamento del linguaggio e il suo utilizzo.

2.2.2 SCML (Service Creation Markup Language)

SCML [12] è un linguaggio di scripting anch'esso basato su XML usato per descrivere servizi in rete NGN definito all'interno delle specifiche JAIN. Esso è in grado di di gestire diverse funzionalità (a partire da funzionalità relative presenti nei terminali all'interazione con l'utente, chiamate multiparty[4], multimedia, e meccanismi

[4]si definifisce chiamata multiparty una comunicazione a cui participano più di 2 persone, come ad esempio una conferenza.

di controllo delle chiamate), che possono essere messe a disposizione dai fornitori di servizi e utilizzate dagli sviluppatori. Attualmente il lavoro è stato focalizzato sul lato Call Control, oltre a questo SCML permette di gestire eventi relativi ai messaggi e alle chiamate, in questo modo un'applicazione può registrarsi per ricevere e quindi reagire ad un particolare insieme di eventi. La registrazione continua a rimanere valida finché lo script non si disattiva.

SCML possono venire eseguiti su un server di chiamate, su un application server oppure su un terminale intelligente, uno script può controllare la logica di funzionamento di un servizio e quindi le chiamate utilizzando tutti le informazioni messe a disposizione dalle interfacce che può utilizzare (es. JAIN) e da quelle fornite dalle impostazioni definite dall'utente.

Nella fase di creazione del servizio attraverso un ambiente di creazione, lo script può venire implementato con degli strumenti XML, editor di testo o convertendo programmi JAVA o C++. L'ambiente di sviluppo si occupa anche della fase d'installazione, dove lo scirpt SCML viene validato e memorizzato in un repository, questo poi verrà caricato dall'application server per l'esecuzione. Un interprete XML si occuperà di eseguire (processore SCML) lo script nell'application server e di convertire le istruzioni SCML nel linguaggio utilizzato dal server.

Lo SCML è stato creato per semplificare lo sviluppo da parte dei programmatori, in particolare coloro che hanno dimestichezza con XML e la programmazione web. Per quanto riguarda l'usabilità da parte di fornitori di servizi di terze parti, SCML non supporta esplicitamente dei meccanismi per gestire script non fidati. In ogni caso, si può appoggiare per gestire problemi di sicurezza di questo tipo, alle funzionalità che mette a disposizione il framework OSA/Parlay. In questo modo gli script SCML posso essere eseguiti facilmente in ambienti fidati.

Il linguaggio SCML è abbastanza semplice da usare, ha una difficoltà paragonabile a quella di CPL, comunque, rispetto a questo, risulta molto più flessibile e potente, e sopratutto questo rimane solo a livello di proposta di standard, e non sono presenti implementazioni mature.

2.2.3 XTML (eXtensible Telephony Markup Language)

XTML è un'altro linguaggio basato su XML che è stato sviluppato per fornire un framework per lo sviluppo di applicazioni di telefonia o multimediali senza doversi

appoggiare ad un protocollo specifico. XTML è basato sugli eventi: un'applicazione è composta da un set di gestori di eventi, questi possono essere indipendenti dal protocollo (come ad esempio eventi di timer, avvio di sessioni) oppure posso dipendere da questo (arrivo di un messaggio SIP). Un gestore di eventi è composto da un set di azioni che sono collegate insieme per formare il flusso di esecuzione di un'applicazione. E' possibile collegare alle azioni degli script scritti in Javascript, e questi possono e possono venir eseguiti in concomitanza con l'esecuzione dell'azione.

Il supporto per gli aspetti dipendenti dal protocollo sono gestiti attraverso un meccanismo di plug-in per definire tutte le azioni che cambiano al variare del protocollo (ad esempio l'invio di un messaggio SIP).

XTML è utilizzato da tecnologie proprietaria , ed è stato proposto come standard W3C. Le specifiche XTML sono pubbliche e aperte così qualunque azienda può sviluppare le proprie prodotti basati su XTML, anche se alcune parti dello standard sono legate alla piattaforma propietaria.

2.3 SIP (Session Initiation Protocol)

SIP[5] è un protocollo di segnalazione definito[6] dal IETF (Internet Engineering Task Force) per stabilire delle connessioni real-time e conferenze su reti IP. Ogni sessione può includere diversi tipi di flussi dati come audio e video, sebbene ora la maggior parte delle estensioni SIP riguardino le comunicazioni audio [16]. E' un protocollo di tipo testuale e si avvicina molto ad HTTP (hypertext transfer protocol) come struttura.

SIP è indipendente dal tipo di rete su cui si appoggia, è stato strutturato per essere uno standard aperto e scalabile, in modo che possa essere utilizzato per diversi scopi.

2.3.1 Caratteristiche del protocollo e Architettura di funzionamento

Come già detto, SIP è un protocollo di tipo testuale [18] ispirato ai protocolli HTTP e SMTP (Simple Mail Transfer Protocol), dei quali è simile sia la sintassi sia il sistema

[5]motivare il cambio di argomento
[6]L'IETF difinisce i protocolli, li pubblica o definisce gli standard?

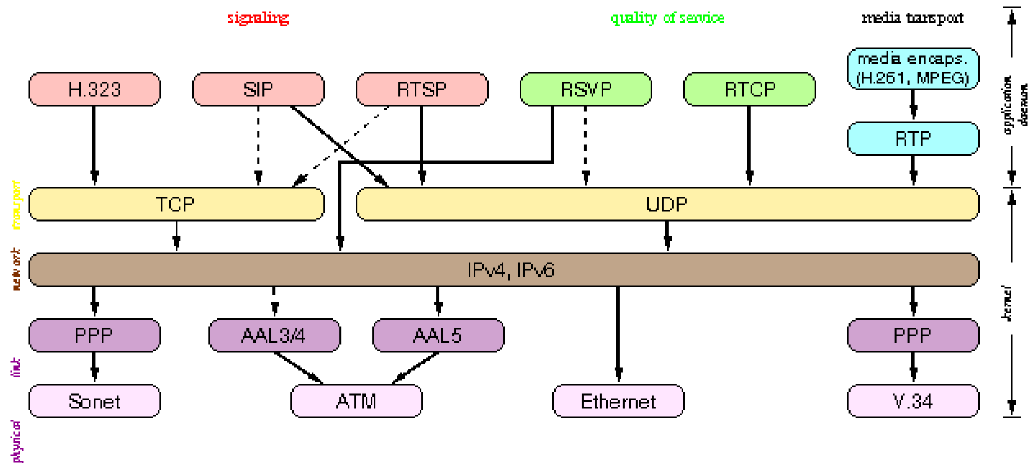

Figura 2.3. Stack di protocolli utilizzabili insieme a SIP

di funzionamento, e anche'esso lavora con un'architettura client/server.

Lo scopo di SIP è la gestione delle chiamate, l'attivazione delle sessioni e la chiusura delle stesse, SIP non si occupa del trasporto dei dati (audio/video) ma solo della segnalazione, la comunicazione vera e propria avviene attraverso altri protocolli come RTP (Real Time Protocol). Nella fase di setup della chiamata vengono definite le caratteristiche della connessione da stabilire, queste informazioni si trovano all'interno del payload[7] del pacchetto.

Le entità principali che interagiscono attraverso SIP sono:

SIP User Agent (UA) Chiamato anche terminale, rappresenta un'estremità di una comunicazione, è un dispositivo hardware o software che è in grado di iniziare e ricevere una chiamata utilizzando SIP come protocollo di segnalazione. Esistono programmi che possono simulare un terminale SIP in un PC, questi vengo anche chiamti softphone, oppure possono essere dei dipositivi simili a telefoni tradizionali, che invece di appoggiarsi ad una rete telefonica tradizionale (PSTN o ISDN), utilizzano una connessione IP. Gli UA possono funzionare sia come server, sia come client a seconda se ricevono o iniziano una chiamata.

[7]Rappresenta la parte dei dati del pacchetto. Ogni pacchetto è diviso in due parti: l'intestazione (header) che contiene tutti i campi utilizzati per l'instradamento della chiamata e le informazioni di servizio del protocollo, e la parte dati (payload) che contiene le informazioni trasportate.

Figura 2.4. Esempi di terminali SIP hardware(User Agent)

SIP Proxy Server Si occupa di gestire le chiamate per una determinata area, inoltra, smista e rielabora le chiamate che gli arrivano secondo le politiche definite. Normalmente gestisce tutti gli utenti / UA di un dominio SIP, e smista le chiamate in arrivo sui relativi UA, o in maniera più complessa se viene definito un criterio per la gestione personalizzata dei profili per ogni utente. Se non è in grado di servire la chiamata, questa viene inoltrata al proxy successivo o verso il dominio specifico, in questo modo il proxy funziona sia da client sia da server. La gestione delle chiamate può essere fatta tenendo conto dello stato delle stesse, quindi il proxy memorizza queste informazioni per tutta la durata della sessione della chiamata.

SIP Redirect Server Reindirizza le chiamate SIP verso altre destinazioni, viene utilizzato per associare serie di indirizzi differenti, la differenza sostanziale rispetto al proxy è che questo non si occupa di inoltrare la chiamata all'indirizzo specificato, ma notifica allo UA (o ad un proxy precedente) l'indirizzo da chiamare e effettivamente, e il client si occupa di rifare la chiamata a questo nuovo indirizzo.

SIP Registrar Mantiene un database delle registrazioni degli utenti, questo elemento viene interrogato da un proxy o da un redirect server per conoscere informazioni riguardo agli utenti chiamati. Viene utilizzato ad esempio per

notificare all'infrastruttura SIP l'attuale locazione dell'utente (ad esempio se questo si trova in ufficio o è rintracciabile tramite cellulare), così che si possa instradare la chiamata verso il terminale attualmente attivo. Può essere integrato direttamente all'interno di un proxy[8].

Il protocollo SIP utilizza come indirizzi degli URI (Uniform Resource Identifier) standard , come ad esempio:

```
sip:marco@polito.it
```

Questo è il caso più semplice, è possible aggiungere delle informazioni agguntive utilizzando la sintassi di passaggio dei parametri del protocollo HTTP

```
sip:voicemail@iptel.org?subject=callme
```

Inoltre è possibile aggiungere ulteriori informazioni in un altro formato come ad esempio la locazione geografica, che può essere utilizzato per defire un indirizzo in funzione della sua posizione spaziale

```
sip:sales@hotel.xy; geo.position:=48.54_-123.84_120
```

per questo caso si immagini uno scenario in cui l'utente da raggiungere sia un dotato di un terminale wireless o satellitare.

Può capitare che all'interno di un'infrastruttura SIP vengano utilizzate altre tecnologie permettendo la convergenza di altre reti di telecomunicazioni, in questo modo, oltre ad utilizzare indirizzi SIP è possibile che vengano usati altri tipo di inidrizzi, ad esempio quelli di tipo telefonico classico (questo nel caso la rete che utilizza SIP abbia un router che instradi le chiamate su rete PSTN o ISDN). Quindi è possibile trovare richieste SIP verso indirizzi del tipo:

```
tel:442887735
```

Normalmente in questi casi le richieste vengono mappate su una serie di indirizzi SIP tradizionali così da permettere l'interoperabilità tra i due tipi di rete.

In realtà è possibile utilizzare qualunque tipo di URL (Uniform Resource Locator) classico all'interno di una chiamata SIP come "*mailto:*" o "*http:*", in questo modo l'User Agent (o il proxy) dovrebbe gestire la gestione corretta dell'indirizzo, quindi mandando una mail o aprendo un browser a seconda della situazione.

[8] Il proxy utilizzato all'interno di questa tesi ha il Registrar integrato direttamente al suo interno

2.3.2 Comandi SIP

Come già detto, SIP è un protocollo simile ad HTTP, ma a differenza di quest'ultimo, SIP è un protocollo *stateful*[9], questo si rispecchia oltre che nella struttura del protocollo, anche nelle architetture dei client e dei proxy.

Le comunicazioni SIP avvengono con un meccanismo di richiesta - risposta, le richieste possono essere di 6 tipi a seconda del tipo di operazione:

1.	*INVITE*
2.	*ACK*
3.	*BYE*
4.	*CANCEL*
5.	*OPTION*
6.	*REGISTER*

Tabella 2.1. Richieste SIP

In particolare vengono troviamo:

INVITE Richiede l'inizio di una sessione di comunicazione, in particolare viene specificato l'indirizzo SIP del destinatario che deve ricevere la richiesta. In questo tipo di messaggio è presente un payload al cui interno vengono specificate le informazioni per l'instaurazione della comunicazione, quindi il tipo di protocollo utilizzato per la sessione vera e proprio e i relativi parametri, come il tipo di codifica e le informazioni di rete (come ad esempio la porta di comunicazione) necessarie.

ACK Conferma l'instaurazione di una sessione, viene utilizzato soltanto insieme ad INVITE, e avvisa lo UA che riceve la chiamata che la sessione è stata aperta con successo, benché possa sembrare un messaggio di risposta in realtà non lo è in quanto rappresenta la conferma che entrambi i terminali sono a conoscenza dell'attuale stato raggiunto. Questo è l'unico messaggio che non necessita di un messaggio di risposta.

[9] Si definisce *stateful* un protocollo in cui le entità che partecipano alla comunicazione devono tener traccia dello stato in cui si trova la comunicazione, in quanto la risposta è dipendente dal tipo di eventi e dal proprio stato, che è determinato dalla sequenza di eventi precedenti. Al contrario, un protocollo *stateless* indica un tipo di comunicazione in cui ogni richiesta è indipendente dalle altre e quindi chi partecipa alla comunicazione non ha bisogno di ricordare la situazione in cui si trova, cioé la risposta è dipendente solo dall'ultimo evento accaduto, ma non dagli eventi precedenti.

BYE Richiede la chiusura di una sessione di comunicazione, una volta che la comunicazione è stata instaurata, questa richiesta avvisa la controparte che la sessione è terminata.

CANCEL Annulla la creazione di una sessione, a differenza del comando precendente questo termina non la sessione di comunicazione, ma la fase di instaurazione annullando il precedente INVITE da parte dello UA che ha generato la richiesta di INVITE. Se lo UA che riceve la chiamata, volesse terminare la fase di *handshaking*[10] non utilizzerebbe questo comando, ma risponderebbe con l'opportuno codice di risposta all'INVITE specificando il rifiuto della chiamata.

OPTION Viene utilizzato per interrogare un UA riguardo le sue funzionalità, in questo modo un UA chiamante può decidere che tipo di comunicazione instaurare, in base alle informazioni che riceve, così che si possa ottimizzare il tipo di sessione instaurata in base alle capacità dei terminali utilizzati. Quando un proxy riceve questo tipo di richiesta, questa viene semplicemente inoltrata allo UA destinazione.

REGISTER Viene utilizzato per informare un registrar della propria presenza. Tipicamente viene utilizzato da un UA quando questo viene attivato, in questo modo viene notificata la presenza di un utente, il suo indirizzo (oppure i suoi indirizzi) attuale. Così che quando arriva una chiamata verso un relativo indirizzo SIP, il proxy interroga il registrar per conoscere i dati di un determinato utente e quindi inoltrare la chiamata: lo stesso discorso può essere fatto per un redirect server. Le registrazioni normalmente hanno una scadenza, al termine della quale un terminale deve rinnovare la richiesta di registrazione per un altro periodo di tempo. Questo meccanismo viene utilizzato per evitare che rimangano informazioni di utenti non più raggiungibili o con terminale disattivato.

Come già detto, ad ogni richiesta effettuata, il terminale SIP si aspetta una risposta, ad ognuna è associato un codice numerico di tre cifre che ricalca la struttura dei codici del protocollo HTTP. In particolare esistono 6 famiglie di codici che

[10]Si definisce *handshaking* (dall'inglese stretta di mano) di un protocollo, la fase preliminare in cui si definisco,tra le due parti, i parametri di configurazione necessari alla successiva instaurazione della comunicazione.

raggruppano risposte di tipo simile:

1xx	Informational
2xx	Success
3xx	Redirect
4xx	Server error
5xx	Server internal error
6xx	Global failure

Tabella 2.2. Famiglie di codici di ritorno SIP

1xx Risposta di tipo informativo, cioè viene utilizzata per informare che un'operazione è in corso, in generale viene mandata in attesa di conoscere una risposta definitiva che è dipendente da una terza entità, questa può essere una UA a cui viene inoltrata la chiamata, o per un utente che fisicamente deve rispondere.

All'interno di questa famiglia si possono trovare i seguenti casi:

100 (Trying) Operazione in esecuzione.

180 (Ringing) Chiamata arrivata a destinazione in attesa di risposta da parte dell'utente.

181 (Call is being forwarded) La chiamta è stata rediretta in attesa di risposta da parte del server (inteso come UA o proxy successivo).

182 (Queued) La richiesta è stata messa in coda, appena il destinatario diverrà raggiungibile la richiesta verrà inoltrata.

2xx Indica un'operazione avvenuta con successo, l'unico codice di questa famiglia è: **200 (OK)** che viene utilizzato quando un'operazione è andata a buon fine da parte del server (colui che riceve la richiesta), come ad esempio quando una richiesta di INVITE viene accettata, quindi la comunicazione è pronta ad essere instaurata.

3xx Viene trasmesso quando viene attivata una redirezione della chiamata, in questo caso è il client, che può essere lo UA che ha iniziato la chiamata, o il proxy che l'ha inoltrata, che si deve occupare di richiamare all'indirizzo specificato all'interno della risposta. In questa famiglia sono presenti cinque casi distinti:

300 (Multiple choices) Sono presenti più indirizzi verso cui reindirizzare la chiamata.

301 (Moved permanently) L'indirizzo destinazione è stato spostato definitivamente.

302 (Moved temporarily) L'indirizzo viene rediretto temporaneamente.

305 (Use proxy) L'indirizzo deve essere raggiunto attraverso un proxy.

380 (Alternative Service) la chiamata non ha avuto successo, ma esistono dei servizi alternativi.

4xx Indica un errore legato al protocollo SIP, da parte di un determinato server, che può essere un UA oppure un proxy che riceve la chiamata, in questo caso abbiamo una gran varietà di risposte possibili:

400 (Bad request) La richiesta non è stata inoltrata nella forma corretta.

401 (Unauthorized) La richiesta richiede un'autenticazione

402 (Payment Required) Attualmente questo codice non è implementato, ma dovrebbe richiedere il pagamanto di una tariffa.

403 (Forbidden) La richiesta fatta è vietata dal server.

404 (Not found) L'indirizzo richiesto non è stato trovato.

405 (Method not allowed) La richiesta fatta non è permessa.

406 (Not acceptable) L'indirizzo cercato non è in grado di rispondere a questa particolare richiesta.

407 (Proxy authentication required) Viene richiesta l'autenticazione verso il proxy utilizzato.

408 (Request timeout) Non vi è stata risposta in tempo utile alla richiesta fatta.

410 (Gone) L'indirizzo richiesto non è più disponibile e non se ne conosce uno alternativo.

413 (Request entity too large) Il payload della richiesta è troppo grosso.

414 (Request-URI too long) La riga di richiesta nell'intestazione è troppo lunga.

415 (Unsupported media type) Il tipo di payload non è utilizzabile dal server.

416 (Unsupported URI scheme) Il tipo di inidirizzo non è supportato dal server.

420 (Bad extension) Il server non è in grado di utilizzare l'estensione richiesta.

421 (Extension required) Un'estensione necessaria non è supportata dal client.

423 (Interval too brief) La scadenza specificata è troppo breve (ad esempio nel caso di una registration)

480 (Temporarily unavailable) Il destinatario è temporaneamente non raggiungibile.

481 (Invalid Call-ID) Identificativo di chiamata non valido, ad ogni sessione viene associato un identificativo di chiamata che in questo caso non corrisponde a nessuno di quelli presenti.

482 (Loop detected) Il server ha scoperto un percorso circolare tra gli indirizzi per cui passa la richiesta, ad esempio quando questa viene inoltrata ad un proxy in cui è già passata.

483 (Too many hops) La richiesta è passata da troppi proxy.

484 (Address incomplete) L'indirizzo specificato è incompleto.

485 (Ambiguous) L'indirizzo specificato è ambiguo.

486 (Busy here) L'utente chiamato, in questo momento non può far fronte a questa chiamata su questo terminale.

487 (Request terminated) La sessione è già terminata, viene utilizzato quando vengono fatte delle richieste e la comunicazione era già conclusa in precedenza.

488 (Not acceptable here) La comunicazione non è accettabile in questo UA, ma potrebbe esserlo in un altro.

491 (Request pending) C'è già una richiesta in corso tra gli stessi UA.

493 (Undecipherable) La richiesta contiene una parte criptata non decifrabile.

5xx Corrispondono ad errori interni al server, e a differenza di quelli della famiglia 4xx, non sono strettamente legati al protocollo SIP, ma piuttosto all'implementazione software dell'applicazione (a volte si può parlare di implementazione hardware). In questa famiglia possiamo trovare:

500 (Server internal failure) Errore interno del server.

501 (Not implemented) Il server non supporta la funzionalità richiesta.

502 (Bad gateway) Il proxy ha ricevuto una risposta non valida da parte del proxy successivo.

503 (Service unavailable) Il server non è temporaneamente in grado di far fronte a questa richiesta.

504 (Gateway timeout) Il proxy non ha ricevuto una risposta in tempo utile da parte del proxy successivo.

505 (Version not supported) Versione del protocollo non supportata.

513 (Message too large) Lunghezza del messaggio eccessiva.

6xx Rappresentano ad errori globali relativi ad un determinato utente. Alcuni di questi ricalcano errori della famiglia 4xx, la differenza può essere delineata dal fatto che questi errori vengono specificati nel caso il sistema abbia una conoscenza globale della situazione di un determinato utente, mentre i 4xx vengono generati da parte di uno specifico UA, e quindi hanno validità parziale. Nella famiglia 6xx possiamo trovare:

600 (Busy) L'utente non può far fronte alla chiamata su qualunque suo indirizzo.

601 (Decline) L'utente rifiuta la chiamata.

604 (Does not exists) L'utente non esiste.

606 (Not acceptable) Il terminale dell'utente non è in grado di instaurare questo tipo di comunicazione(simile al 488).

Si può notare la notazione non uniforme nella numerazione dei codici di ritorno, in particolare spesso si vede che da numerazioni basse si passa a valori con decine pari a 80, questo per mantenere la compatibilità verso i codici di errore del protocollo HTTP. Infatti paragonando le due serie di valori [15] si può notare come i codici con decine e unità piccole siano un sottoinsieme del codici HTTP; i codici dei messaggi specifici del protocollo SIP iniziano da numerazioni alte per non rischiare sovrapposizioni.

2.3.3 Informazioni sul protocollo

Un generico pacchetto di richiesta SIP può avere questa forma:

```
REGISTER sip:130.192.31.53:2000 SIP/2.0
Via: SIP/2.0/UDP 130.192.31.18:5060;branch=fake.1
Call-ID: 1057571362953@130.192.31.18
CSeq: 1463037875 REGISTER
From: ”Mr Harpo” <sip:harpo@polito.it>;tag=1057573575937
To: ”Mr Harpo” <sip:harpo@polito.it>
Contact: ”Mr Harpo” <sip:harpo@130.192.31.18:5060>
Expires: 3600
Content-Length: 0
```

Questo pacchetto è di tipo REGISTER: lo si può notare nella prima riga dell'intestazione in cui viene specificato il comando; subito dopo sono evidenziati l'indirizzo del registrar a cui viene inoltrata la richiesta, e il numero di versione del protocollo (SIP/2.0). Dopo la prima riga sono presenti tutte le varie voci dell'header ognuna delle quali ha una funzione ben definita (una trattazione più approfondita di questa esula dallo scopo di questo elaborato, in questo caso si rimanda a [14]).

Il pacchetto di risposta ha una forma simile:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 130.192.31.18:5060;branch=fake.1
Call-ID: 1057571362953@130.192.31.18
CSeq: 1463037875 REGISTER
From: ”Mr Harpo” <sip:harpo@polito.it>;tag=1057573575937
To: ”Mr Harpo” <sip:harpo@polito.it>
Expires: 3600
Contact: ”Mr Harpo” <sip:harpo@130.192.31.18:5060>;expires=3600
Content-Length: 0
```

In questo caso si può notare, come al solito, il codice di risposta sempre nella prima riga e la presenza degli altri campi dell'intestazione.

Nei due esempi precedenti i pacchetti SIP non erano dotati di una parte dati (payload) ma erano semplicemente costituiti dall'intestazione. Gli unici tipi di pacchetti che portano con se un campo dati sono INVITE e OPTION, nei quali vengono spedite le informazioni e i parametri riguardanti la sessione audio (ma non solo) di comunicazione da instaurare.

Un esempio di pacchetto INVITE è il seguente:

```
INVITE sip:harpo@130.192.31.18:5060 SIP/2.0
Via: SIP/2.0/UDP 130.192.31.53:2000;branch=z9hG4bKd964aa3645a
Via: SIP/2.0/UDP 130.192.31.53:5060;branch=fake.1
Contact: "Marco MDN" <sip:mdn@130.192.31.53:5060>
Call-ID: 1057572933895@130.192.31.53
CSeq: 1709578720 INVITE
From: "Marco MDN" <sip:mdn@polito.it>;tag=1057572933895
To: <sip:harpo@polito.it>
Priority: non-urgent
User-Agent: SipClient
Content-Type: application/sdp
Max-Forwards: 70
Record-Route: <sip:130.192.31.53:2000>
Content-Length: 164

v=0
o=mdn 1057572395080 0 IN IP4 130.192.31.53
s=Generic sdp session
c=IN IP4 130.192.31.53
t=0 0
m=audio 10626 RTP/AVP 0
a=rtpmap:0 PCMU/8000/1
a=sendrecv
```

Si può notare al di sotto dell'intestazione la sezione dati del pacchetto, in cui si possono notare le informazioni utilizzate per instaurare una comunicazione audio tramite RTP. Tutte queste informazioni sono codificate secondo lo standard SDP

(Session Definition Protocol) che più che essere un protocollo è un formato in cui vengono scritti questi dati.

SDP in particolare si occupa di specificare:

- Il tipo di media da utilizzare come il codec[11] e la frequenza di campionamento.
- Il destinatario del flusso identificata da un indirizzo IP e da una porta.
- Il nome della sessione e lo scopo (come nel campo *subject* nella posta elettronica).
- Informazioni temporali sull'attivazione della sessione.
- I dati sul contatto (cioè il nome utente ed il relativo URI SIP).

Confrontando il formato del protocollo SIP con il protocollo HTTP, come è stato sottolineato diverse volte, si notano numerose analogie, a partire dalla configurazione dei pacchetti fino a codici di ritorno, questo potrebbe far pensare ad una possibile compatibilità o interoperabilità tra i due protocolli. In realtà questi rappresentano due entità distinte e indipendenti che in nessun caso possono lavorare insieme, sia per problemi riguardanti alla logica di funzionamento (si ricorda che HTTP è un protocollo stateless, mentre SIP non lo è), sia per problemi riguardanti alla struttura vera e propria. Questo viene ancora più enfatizzato analizzando i contesti in cui operano, SIP nel controllo e la gestione delle chiamate e HTTP per il trasferimento di risorse, contesti che sono completamente distinti e disaccoppiati.

La somiglianza è dovuta alla semplicità intrinseca dell'utilizzo di un protocollo testuale di questo tipo, che è stata provata dalla lunga storia di HTTP ed è stata ripresa all'interno del contesto SIP per sfruttare tutti i suoi vantaggi e l'esperienza maturata in tutti questi anni.

2.3.4 Scenari di funzionamento

Una volta compresa la struttura di SIP è importante capire come funziona nell'utilizzo reale, in particolare possiamo riconoscere tre fasi prinicipale all'interno delle sessioni SIP:

[11] Codec è abbreviazione di COdificatore DECodificatore, rappresenta il formato in cui vengono codificati i dati, in questo caso dati audio.

Registrazione (Registration) viene fatta da parte di ogni UA verso un registrar appena questo inizia a funzionare attraverso una richiesta di tipo REGISTER, in questo modo il terminale notifica la sua presenza e il suo indirizzo al sistema.

Attivazione della chiamata (Call Setup) effettuata ogni volta che un UA vuole eseguire una chiamata verso un altro UA, normalmente la richiesta (di tipo INVITE) viene mandata ad un proxy che si occupa di inoltrarla al destinatario corretto, in maniera analoga ad un centralino tradizionale. La sessione di comunicazione vera e propria inizia non appena il terminale destinazione notifica la sua disponibilità ad instaurare la comunicazione attraverso una risposta di tipo OK (codice 200) il terminale chiamante conferma con un messagio ACK. In tutti questi passaggi il proxy si occupa di fare da tramite tenendo aggiornato lo stato della sessione da entrambi i lati della transazione.

Termine della chiamata (Teardown) per terminare la comunicazione un terminale manda una richiesta di tipo BYE alla controparte che risponde con un OK per notificare la ricezione del messaggio. Qualunque terminale, sia quello che ha attivato la chiamata, sia il destinatario, può eseguire una richiesta di chiusura di connessione.

Si possono vedere queste operazioni nella figura 2.5, in cui troviamo una situazione con due UA e un proxy che funziona anche come registrar, in questo modo è possible notare come tutti i messaggi vengono indirizzati al proxy e questo li smista ai relativi destinatari.

In questo modo, è facile capire che, il proxy deve mantenere lo stato di ogni UA coinvolto nelle comunicazioni attualmente in corso: in questo caso si parla di proxy statefull. In scenari di funzionamento molto estesi, come all'interno di società estremamente grandi, il proxy necessita di una grande quantità di risorse, si dal punto di vista computazionale, sia dal punto di vista della memoria.

Per evitare questi problemi di scalabiltà è possibile far funzionare il proxy in modalità stateless, in cui non viene memorizzato lo stato delle transazioni, e il proxy si occupa soltanto di inoltrare le richieste di INVITE che riceve ai relativi destinatari, tutti le altre comunicazioni avvengono direttamente tra le UA interessate senza passare più dal proxy.

In questo modo le risorse di cui ha bisogno il proxy non dipendono dal numero di chiamate che questo deve elaborare. Benché si guadagni dal punto di vista della

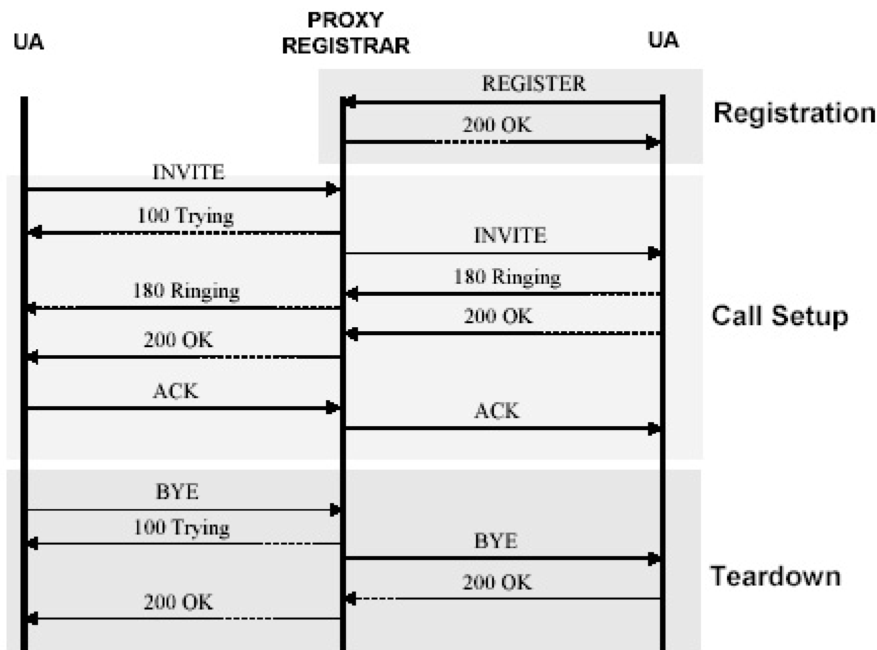

Figura 2.5. Sessione SIP

scalabilità, con questo tipo di soluzione, ci sono una serie di svantaggi a cui si va incontro, come ad esempio alla perdita di un controllo centralizzato sulle comunicazioni. Cioè, se comunque il proxy si trova al centro di tutte le sessioni SIP, è molto semplice tener traccia di tutte le chiamate, quindi conoscere informazioni tipo la durata e il numero di chiamate di un terminale per così poter eseguire controlli, statistiche e tariffazioni.

2.4 Java e SIP

All'interno del contesto delle NGN, Java trova una posizione privilegiata, in quanto dà la possibilità di sfruttare componenti già pronti senza avere troppi vincoli sul tipo di piattaforma da utilizzare.

In questa sezione verrà fatta una panoramica sulle tecnologie più interessanti in questo ambito, usate per definire una API più astratta per le applicazioni che

vogliono utilizzare SIP senza dover lavorare direttamente con lo *stack* del protocollo.

2.4.1 JAIN SIP

L'API JAIN SIP fornisce un'interfaccia Java standard e portabile per condividere informazioni tra client SIP e server SIP, mettendo a disposizione funzionalità per il controllo delle chiamate e quindi fornendo un'infrastruttura per lo sviluppo di applicazioni su reti convergenti.

Questa API permette la creazione rapida e l'attivazione di servizi dinamici di telefonia all'interno di piattaforme Java. Le applicazioni di telefonia [10] hanno bisogno di risorse costose per il loro sviluppo, le prove a la messa in funzione. Un componente JAIN SIP può essere creato rapidamente, provato e integrato in una gran varietà di piattaforme avendo a già disposizione un buon numero di strumenti e utility. Una soluzione inter piattaforma basata su JAIN offre ai gestori telefonici, ai fornitori di servizi e di componenti di rete, un ambiente di sviluppo aperto e consistente dove sviluppare e integrare servizi di telefonia altamente riutilizzabili.

Un'applicazione JAIN SIP può essere scritta come programma, applet, servlet o bean; le API mettono a disposizione le potenzialità presenti nelle diverse versioni di SIP in un'interfaccia Java comune.

JAIN SIP non fornisce l'implementazione dello stack SIP, ma si occupa soltanto di fornire un 'interfaccia standard; ogni implementazione del protocollo può aderire a questa specifica, in questo caso viene fornito un TCK (Test Compatibility Kit) che si occupa di verificare la compatibilità dell'implementazione allo standard definito. In questo caso si può ottenere la vera interoperabilità tra le varie implementazioni, così le applicazioni diventano indipendenti dalla piattaforma sulla quale vengono eseguite, e con piattaforma non si intendeno più solo le coppie hardware/sistema operativo, ma si considera anche l'infrastruttura SIP utilizzata.

Nella figura 2.6 è visibile la struttura di una generica applicazione che si basa su questa API, in particolare vengono gestiti tutti gli aspetti relativi alla comunicazione attraverso SIP, come ad esempio la risposta alle richieste che arrivano al programma, attraverso opportune interfacce che gestiscono gli eventi relativi ai vari eventi SIP (come l'arrivo di richieste o di risposte, o la scadenza di timeout), o la creazione e gestione degli indirizzi, messaggi ed elementi del pacchetto SIP.

Nell'ambito di questa tesi è stata utilizzata l'implementazione di riferimento

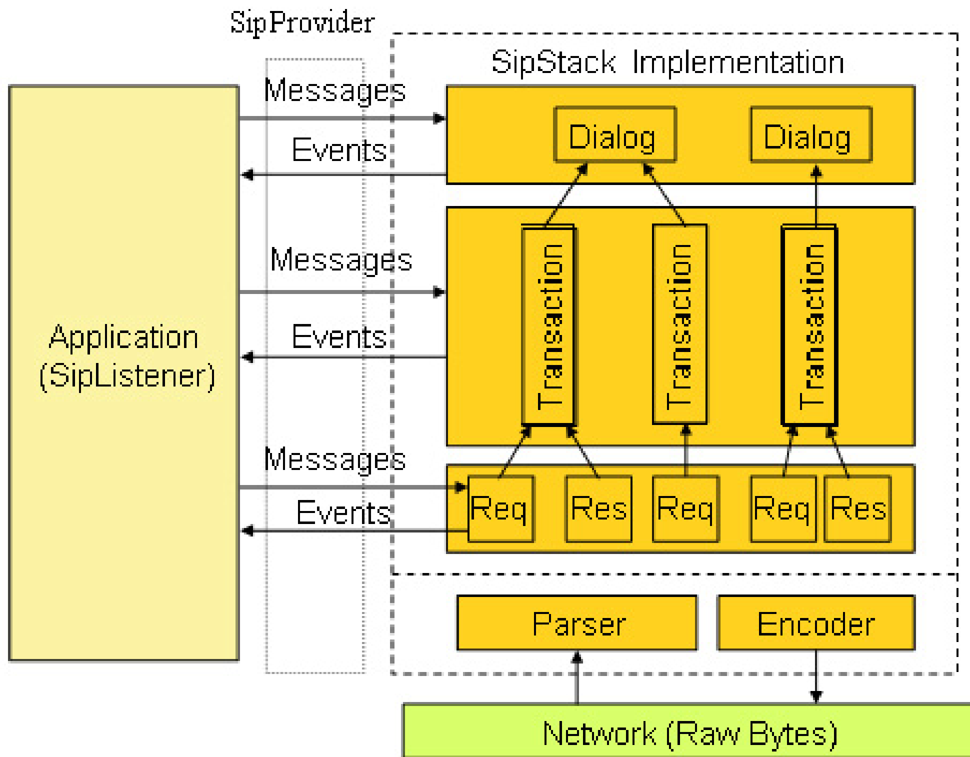

Figura 2.6. Schema di un'applicazione JAIN SIP

definita dal NIST (National Institute of Standards and Technology) [17], ed in particolare il proxy sviluppato nell'ambito del questo progetto.

2.4.2 JAIN SIP Lite

L'API JAIN SIP Lite è una libreria con interfaccia Java mirata soltanto allo sviluppo di applicazioni SIP di tipo User Agent, cioè terminali software che si affacciano a reti SIP; questo definisce chiaramente che tipo di funzionalità di rete essa offre. Lo scopo di queste API è di aumentare le potenzialità offerte dall'intefaccia JAIN SIP e dalle SIP Servlet: queste possone essere viste come degli involucri tramite i queli si può accedere alle funzionalità del protocollo SIP, ma richiedono un programmatore esperto per essere utilizzate correttamente.

JAIN SIP Lite offre un'API di alto livello, la cui implementazione è in grado di gestire diversi aspetti puramente tecnici relativi a SIP, che in questo divengono

trasparenti allo sviluppatore che le usa. La differenza principale verso le SIP Servlet è che questa interfaccia non vincola il programmatore all'uso di un'Application Server e non è strettamente legato ad un proxy SIP per funzionare (a differenza della tecnologia SIP Servlet).

Il vantaggio di utilizzare un'API standard, è che risulta molto semplice far sviluppare i servizi da parte di terzi e poi poterli utilizzare all'interno di qualunque sistema.

Le specifiche JAIN SIP Lite sono in fase di standardizzazione da parte del JCP; attualmente sono ancora in fase di revisione e quindi non è ancora presente un release definitiva, anche se è presente una prima release da parte del NIST.

2.4.3 JCC (Java Call Control)

JCC è una API Java che fornisce un livello di astrazione per le funzionalità di controllo delle chiamate, questo è definito come standard all'interno dell'abito JAIN. JCC permette di costruire applicazioni che possono venir attivate o ricevere delle notifiche durante la fase di avvio delle chiamate, all'interno di uno scenario di servizi Intelligent Network (IN) o Advanced Intelligent Network (AIN). JCC quindi permette di sviluppare applicazioni che possono essere eseguite su qualunque piattaforma che supporta queste API. In questo modo permette ai fornitori di servizi di offrire nuove funzionalità agli utenti finali in maniera rapida ed efficiente potendo sviluppare autonomamente, attraverso fonti esterne, comprando servizi già pronti o combinando queste soluzioni.

Per quel che riguarda le funzionalità di rete esposte, JCC include elementi per l'osservazione, l'attivazione, la risposta, il processamento o la manipolazione delle chiamate, come anche l'attivazione e l'invocazione di applicazioni e il relativo utilizzo dei risultati ottenuti durante l'elaborazione della chiamata. Il modello di chiamata specifico (incluse le macchine a stati specifiche per un particolare protocollo) per un particolare protocollo è defìnto all'interno dell'API.

Per JCC, quando si parla di chiamate, si includono (ma non solo) sessioni di tipo multimediale o multiparty effuettuate al di sopra di reti integrate (PSTN, a pacchetto e/o wireless).

Questa libreria offre le funzionalità sufficienti per implementare la maggior parte, anche se non tutti, i servizi di base e avanzati offerti dai gestori; per poter gestire

tutte le capacità offerte dalle reti, gli sviluppatori hanno bisogno di far riferimento alle API JCAT (Java Coordination e Transaction), che integrano le API JCC per la gestione delle NGN.

Questa API non è stata studiata per poter utilizzare l'infrastruttura di segnalazione delle reti di telecomunicazione da parte degli utenti. Piuttosto, l'approccio è di fornire la possibilità di sviluppare servizi da parte di programmatori indipendenti e di poterli integrare all'interno della rete senza compromettere l'affidabilità e la sicurezza della rete. In quest'ottica è stata fornita una implementazione di riferimento e una suite di test di compatibilità per poter sviluppare nell'assoluta sicurezza del rispetto delle specifiche dello standard.

2.4.4 SIP Servlet

Le SIP Servlet sono un set di librerie usate per creare servizi in reti basate su SIP. Le API SIP Servlet [19] sono API Java basate sulle già esistenti servlet. Le SIP servlet funzionano in maniera analoga a queste ultime, infatti l'applicazione impersonata dalla servlet non funziona in maniera autonoma, ma viene ospitata da un'infrastruttura software definita Servlet Container. Le specifiche SIP servlet hanno anche l'obbiettivo di rendere standard diversi aspetti del container: le regole utilizzare per associare le richieste SIP ad una specifica servlet, il modello di sicurezza, il Servlet Deployment Descriptor (documento che accompagna la servlet che contiene le informazioni per il container su come deve essere gestita la servlet, normalmente è in formato XML), il formato del file di distribuzione analogo al JAR Java (simile al format WAR utilizzato dalle servlet HTTP).

Le API SIP Servlet permettono alle applicazioni di iniziare e di rispondere a richieste SIP. Queste forniscono una interfaccia semplice alle funzionalità offerte da SIP (sia come User Agent, sia come proxy) all'applicazione, mentre tutte le questioni di basso livello legate al protocollo sono gestite in modo trasparente dal SIP Servlet Container.

Gli aspetti di cui si occupa di gestire il container sono: ritrasmissioni di messaggi (nel caso di funzionamento come proxy), scelta della risposta migliore (ad esempio quando arrivano più risposte a seguito di chiamate concorrenti), generazione di numeri di sequenza e degli identificativi di chiamata e la gestione degli header.

Attualmente lo standard SIP Servlet ha raggiunto la sua prima release definitiva

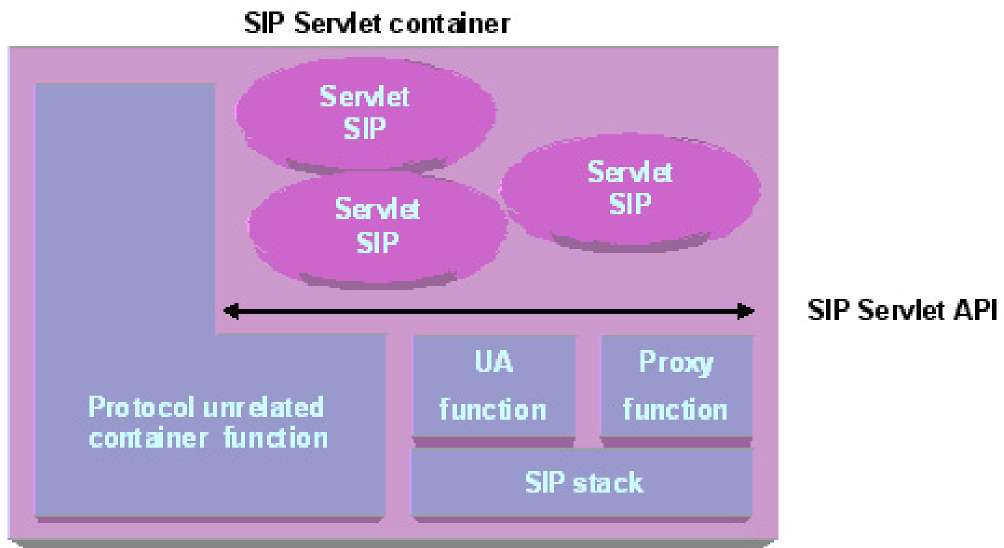

Figura 2.7. Architettura SIP Servlet

all'interno del JCP all'inizio del 2002.

Capitolo 3

Linguaggio CPL

3.1 Introduzione

CPL o Call Processing Language è un linguaggio che può essere utilizzato per descrivere e controllare servizi di telefonia su rete (Voice Over IP). E' stato progettato per essere utilizzato sia su server di rete o su terminali avanzati. E' stato pensato per essere semplice, estendibile, facilmente maneggiabile da programmi attraverso interfacce grafiche, slegato da qualunque piattaforma o dal protocollo di segnalazione utilizzato (che normalmente si tratta di SIP o H.323). Oltre a ciò si è tenuto conto delle problematiche di sicurezza che possono nascere in questi contesti, e dato che gli utenti possono creare liberamente i propri profili CPL sotto forma di script e inviarli ad un server che li esegue, è stato fatto in modo che questi script non possano eseguire programmi arbitrariamente, inoltre, non esiste il concetto di variabile o di ciclo, quindi vengono automaticamente esclusi tutti i problemi legati a situazioni di *overflow* e *denial of service*[1].

Come già detto CPL è basato su XML [20], quindi i programmi sono costituiti da documenti di testo e gli elementi del linguaggio sono costituiti da *tag*, cioè identificatori di testo racchiusi tra parentesi angolari; questi elementi sono strutturati

[1] *Overflow* e *Denial of Service* (DOS) sono tipi di problemi legati alla sicurezza delle applicazioni e dei servizi. In particolare il primo nasce quando è possibile impostare valori di variabili, o parametri di funzioni dall'esterno, e non c'è alcun controllo sulla quantità di dati inserti. In questo modo è possibile bloccare un'applicazione o addirittura far eseguire del codice arbitrario inserendo una quantità di dati maggiore rispetto a quanto il sistema sia in grado di accettare. Il secondo è una possibile conseguenza del primo, cioè si definisce un attacco DOS quando questo è mirato a disabilitare o rendere inacessibile un servizio (negazione del servizio).

in maniera gerarchica all'interno del documento. Il vantaggio di questa scelta è che l'elaborazione di un documento di questo tipo è estremamente semplice, in quanto esistono un gran numero di librerie in grado di leggere e analizzare documenti XML. Un altro grosso vantaggio nell'utilizzo di XML è la sua capacità di poter definire una schema di documento, al quale tutti i file che appartengono a quel particolare dialetto (in questo caso CPL) devono aderire completamente. Il processo di controllo si definisce *validazione*, e un documento che passa questa fase viene tecnicamente definito valido.

Tutta questa elaborazione viene eseguita automaticamente, quindi è estemamente facile controllare se uno script CPL segue effetivamente le specifiche del linguaggio o meno. Infatti è molto importante poter controllare che un programma CPL non contenga degli errori in fase di caricamento dello script nel server, piuttosto che in fase di esecuzione, cosa che altrimenti si potrebbe tradurre in un malfunzionamento del servizio[2]

E' importante sottolineare che il linguaggio CPL serve per eseguire determinate azioni all'arrivo, o all'uscita, di un chiamata; nel momento in cui viene scatenato uno di questi eventi vengono lanciate le istruzioni contenute all'interno dello script. Quindi è importante sottolineare che CPL lavora soltanto all'interno della fase di INVITE (paragrafo 2.3.4) nell'attivazione della chiamata, permettendo ad esempio di decidere dove inoltrare la chiamata.

3.2 Struttura

La struttura gerarchica di XML viene rispecchiata in CPL: tutto lo script è racchiuso all'interno del tag radice <*cpl*>, all'interno di questo vengono inserite le azioni di controllo delle chiamate, che possono essere di tre tipi: <*subaction*>,<*incoming*> e <*outgoing*>, questi vengono definiti tag di azione.

Gli elementi di tipo *subaction* definiscono quelle che in un generico linguaggio di programmazione vengono considerate funzioni, cioè frammenti di codice che possono venir chiamati da qualunque altra parte del programma. La chiamata ad una *subaction* in CPL viene fatta attraverso il tag <*sub*>. A differenza della maggior parte

[2]Ovviamente il malfunzionamento sarebbe isolato al particolare utente a cui lo script è stato associato, mentre il server continuerebbe a funzionare normalmente per tutto il resto dell'utenza.

dei linguaggi di programmazione non è possibile richiamare più volte la stessa funzione all'interno di una stessa esecuzione, quindi non è possibile creare programmi ricorsivi. Questa che può sembrare una limitazione, è dovuta sopratutto a motivi di sicurezza, perché in questo modo è impossibile creare dei cicli di lunghezza indeterminata, quindi così si evitano possibili situazioni di *stack overflow*[3]

I tag di tipo *incoming* e *outgoing* definisco le parti di script che vengono chiamate quando arriva in ingresso una chiamata (*incoming*) o quando sta uscendo una chiamata dal sistema *outgoing*. In generale è normale che sia presente uno solo di questi elementi all'interno di uno stesso script, questo perché viene utilizzato uno o l'altro a seconda del contesto in cui questo viene eseguito. Nel caso CPL venga utilizzato per modificare il comportamento di un UA avanzato (come un telefono VoIP) possono venire utilizzati entrambi gli elementi per modellare sia il traffico in entrata che quello in uscita dal terminale. Se invece gli script CPL si trovano ad essere eseguiti all'interno di un proxy viene normalmente utilizzato solo il tag *incoming* per definire come si deve comportare il sistema in caso dell'arrivo di una chiamata per un particolare utente.

Nell'ambito di questa tesi, in funzione del tipo di archittura utilizzata [cap. 4], si è utilizzato CPL nel secondo scenario descritto in quanto viene utilizzato per decidere il comportamento del proxy riguardo le chiamate in arrivo.

3.2.1 Struttura generica di un elemento CPL

Con script CPL si considera una sequenza di istruzioni CPL che possono trovarsi all'interno di un tag di azione (i tre nominati nella sezione precedente) che descrivono una serie comportamenti che il sistema deve eseguire. Ogni script è formato da una serie di nodi che corrispondono ad una serie di istruzioni CPL, ogni nodo può essere rappresentato, all'interno del linguaggio CPL, da uno o più tag XML. In particolare esistono diverse famiglie di nodi:

Nodi di *Switch* Questi permettono di effettuare delle scelte all'interno dell'esecuzione di uno script. Sono equivalenti al costrutto *if ... then ... else* presente in qualunque linguaggio di programmazione. Le condizioni che possono essere

[3] Si definisce *stack overflow* la situazione di errore causata quando all'interno dell'esecuzione di un processo la memoria viene saturata a fronte della chiamata di troppe funzioni contemporaneamente, che quindi vengono a riempire completamente la memoria. Questo è un problema comune nella scrittura di funzioni ricorsive, cioè quelle funzione che richiamano se stesse.

specificate riguardano, a seconda del nodo, condizioni temporali, controlli sugli indirizzi di origine e di destinazione, parametri riguardanti la lingua, elementi dell'header del pacchetto di richiesta, alla priorità e ad altri aspetti della chiamata. Tutti questi nodi sono formati da più elementi XML, in particolare c'è un tag principale che specifica che tipo di switch si sta definendo, e al suo interno si trovano quelli che sono deifiniti *output* che rappresentano le varie condizioni che si possono presentare all'interno di quel tipo di scelta. All'interno di ognuno di questi output vengono inserite le parti di codice da eseguire nei vari casi.

Nodi di modifica delle destinazioni Questi nodi vengono utilizzari per modificare la lista delle destinazioni che si trovano nella richiesta arrivata al gestore CPL: in questo modo è possibile cambiare la destinazione di una chiamata, cioè effettuare la redirezione verso un'altro indirizzo. Nella maggior parte dei casi è presente un solo destinatario, ma è possibile trovare casi in cui la richiesta di chiamata viene diretta verso più indirizzi.

Nodi di operazioni di segnalazione Sono il punto di contatto tra il linguaggio CPL e il protocollo di segnalazione sottostante, vengono utilizzate per inoltrare le chiamate, eseguire delle redirezioni e rispondere alle UA che ha generano le chiamate. Alcune di queste usano un meccanismo analogo a quello descritto nei nodi di switch per intraprendere azioni a seconda del risultato dell'operazione eseguita.

Nodi di operazioni generiche Corrispondo ad azioni che non ricadono negli altri casi, come ad esempio l'esecuzione di log, o invio di mail.

Un esempio di script CPL può essere il seguente:

```
<cpl>
  <incoming>
    <time-switch tzid="America/New_York"
            tzurl="http://zones.example.com/tz/America/New_York">
      <time dtstart="20000703T090000" duration="PT8H">
        <proxy />
      </time>
```

```
      <time dtstart="20030303T090000"
            duration="PT8H" until="20030303T090000">
        <log message="Received call" />
      </time>
      <otherwise>
        <location url="sip:jones@voicemail.example.com">
          <proxy />
        </location>
      </otherwise>
    </time-switch>
  </incoming>
</cpl>
```

E' importante notare che l'esecuzione di un programma CPL avviene in modo ricorsivo anziché sequenziale, cioè l'operazione successiva da eseguire viene cercata all'interno del tag corrente e non nel tag successivo, come invece accade nella stragrande maggioranza dei linguaggi. Quindi la struttura gerarchica del documento XML viene esplorata di padre in figlio durante l'esecuzione, e non capita in nessun caso il contrario. L'unica eccezione avviene all'interno dei costrutti in cui sono presenti degli output (nodi di scelta e simili) in cui ci si sposta tra tag *sibling* (letteralmente fratelli, si intendono elementi che sono figli di uno stesso elemento genitore) per decidere quale condizione sia effettivamente verificata, ma una volta che viene selezionato un output (cioè viene eseguita la scelta) non c'è la possibilità di ritornare indietro nella gerarchia.

3.3 Elementi

Per semplicità i nodi sono stati divisi per categoria a seconda della loro funzione. I parametri che possono prendere i vari elementi vengono specificati attraverso attributi dei tag XML.

3.3.1 Nodi di switch

Vengono utilizzati per eseguire determinate azioni subordinate all'avverarsi di un determinata condizione, quindi a seconda della situazione in cui è stata fatta una

chiamata vengono effettuate delle operazioni piuttosto che altre. Ad esempio il seguente frammento di codice:

```
...
    <address-switch field="origin">
        <address is="sip:info@vari.com">
            ... ramo 1 ...
        </address>
        <address subdomain-of="lavoro.com">
            ... ramo 2 ...
        </address>
        <otherwise>
            ... ramo 3 ...
        </otherwise>
    </time-switch>
 ...
```

Discrimina a seconda dell'indirizzo chiamante il comportamento del proxy, il nodo è di tipo <*address-switch*> che si occupa di elaborare l'indirizzo di una chiamata, in particolare in questo caso viene utilizzato l'indirizzo dir origine (specificato nel suo attributo *field="origin"*). Una volta specificato su cosa verterà la scelta vengono presentati i vari casi, ognuno dei quali corrisponderà ad un output. Dato che stiamo utilizzando un <*address-switch*>, gli output saranno di tipo <*address*>, la condizione effettiva viene specificata negli attributi di ogni output. In particolare nel primo *address* viene controllato che l'indirizzo corrisponda a *sip:info@vari.com*, nel qual caso verrebbe eseguito il codice presente nel ramo 1; la seconda condizione viene verificata (ovviamente solo nel caso in cui la prima non sia vera) se il chiamante appartiene al domino *lavoro.com*, nel cui caso l'esecuzione passa alle istruzioni presenti nel ramo 2.

Esistono degli output che sono presenti all'interno di ogni nodo di switch, uno di questi corrisponde al tag <*otherwise*> che viene eseguito quando nessuno degli altri output risulta verificato. Nei linguaggi di programmazione classici corrisponde all'elemento *else* del costruttto *if then else*.

Un altro output particolare è il <*not-present*> che viene scelto quando il parametro presente nel tag di switch non è presente all'interno della richesta della chiamata. Ovviamente non è applicabile per tutti i casi, perché alcune informazioni, come ad esempio l'ora di arrivo della chiamata, sono sempre presenti.

Address Switch

Viene utilizzato per prendere decisioni in funzione degli indirizzi presenti nella richiesta di chiamata.

Il tag specifico di questo nodo è:

Tag:	<*address-switch*>
Output:	*address*,*otherwise*,*not-present*
Parametri:	*field*,*subfield*

Attraverso i parametri *field* e *subfield* si può specificare quale indirizzo utilizzare per eseguire i controlli, in particolare nel primo parametro, che è obbligatorio, si possono specificare i valori: *origin*, *destination* e *original-destination*, che corrispondono rispettivamente all'indirizzo di chi richiede la chiamata, l'indirizzo di chi riceve la chiamata e l'indirizzo originale del destinatario presente nella richiesta di INVITE nel caso si siano modificati gli indirizzi di destinazione.

Nel parametro *subfield* si può decidere se analizzare solo una parte dell'indirizzo, in particolare i valori validi possono essere: *address-type*, *user*, *host*, *port*, *tel*, *display*. In particolare corrispondono a:

address-type tipo di indirizzo, in particolare può assumeri i valori: *sip*, *tel* (nel caso sia un numero di telefono vero e proprio) o *h323* (se il protocollo di segnalazione non fosse SIP ma H.323). In realtà potrebbe assumere altri valori per quanto già visto nel paragrafo 2.3.1 sugli indirizzi utilizzabili con SIP.

user nome utente

host nome della macchina presente nell'indirizzo

port numero di porta specificata nell'indirizzo

tel il numero telefonico, nel caso sia un indirizzo di tipo telefonico

display corrisponde alla descrizione testuale che appare nell'indirizzo

Esistono anche altri valori specifici ad un determinato protocollo che è possibile utilizzare all'interno del campo *subfield*; per questi si veda [11].

Ogni output di questo switch specifica una determinata condizione applicabile all'elemento dell'indirizzo scelto, in particolare ad ogni tag *address* presente all'interno di questo nodo specifica un condizione ben precisa. In particolare viene utilizzato un attributo per specificare il tipo di condizione. Gli attributi a disposizione utilizzabili all'interno di ogni output *address* sono:

is viene utilizzato per specificare una corrispondenza esatta del valore specificato all'interno di questo attributo con l'indirizzo selezione nell'*address-switch*

contains viene verificato quando il valore specificato è contenuto nell'indirizzo (è possibile utilizzarlo soltanto con il *subfield display*)

subdomain-of può essere utilizzato per controllare se un determito indirizzo appartiene ad un determinato dominio

Oltre all'output *address* è possibile trovare anche *otherwise* e *not-present* con i significati visti nel paragrafo 3.3.

String Switch

Viene utilizzato per fare scelte su informazioni di tipo testuale presenti in vari campi dell'intestazione del pacchetto INVITE, come l'oggetto del messaggio o il tipo di UA utilizzato.

Il tag utilizzato è:

Tag:	*<string-switch>*
Output:	*string,otherwise,not-present*
Parametri:	*field*

Il parametro *field* contiene una stringa con la descrizione del campo dell'intestazione da utilizzare per i confronti, in particolare è possibilire inserire un valore tra: *subject*, *organization*, *user-agent* e *display*.

Nel dettaglio questi campi corrispondono a:

subject definisce il campo dell'oggetto della comunicazione.

organization corrisponde all'organizzazione di chi sta eseguendo la chiamata

user-agent è il nome del UA che viene utilizzato da chi effettua la chiamata

display corrisponde alla descrizione testuale che appare nell'indirizzo

L'output principale utilizzato in questo switch è *string*, che in base ai parametri specificati controlla il contenuto del campo specificato, in particolare può contenere gli attributi:

is cerca una corrispondenza esatta del valore specificato in questo attributo con il campo selezionato

contain controlla se il valore specificato nell'attributo è contenuto nel campo

Oltre all'output *string* è possibile utilizzare, come quanto già visto per lo switch precedente, anche gli output *otherwise* e *not-present* con lo stesso significato.

Language Switch

Questo switch esegue i controlli in base alla lingua utilizzata dal chiamante, lo switch utilizzato è:

Tag:	*<language-switch>*
Output:	*language*,*otherwise*,*not-present*

In realtà quando si parla di lingua utilizzata, ci si può riferire alla lingua che il chiamante potrebbe preferire per la comunicazione. Questo nodo non necessita di alcun parametro.

L'output utilizzato è *language*, in cui viene specificato il tipo di lingua in base al contenuto del parametro *matches*.

Il formato utilizzato per specificare la lingua è definito nel RFC 3066 [21], è possibile definire anche dei gruppi di lingue, in modo da far verificare per un solo output diversi linguaggi contemporaneamente.

Oltre all'output *language* è possiblie utilizzare, come già visto, anche gli output *otherwise* e *not-present* con i medesimi significati.

Time Switch

Attraverso questo switch è possible specificare delle condizioni basate sul momento di arrivo di chiamata, potendo definire degli intervalli di tempo in modo estremamente flessibile potente.

Il tag che definisce questo switch è:

Tag: <*time-switch*>
Output: *time*,*otherwise*
Parametri: *tzid*, *tzurl*

Con questo switch è possibile creare degli script che si comportino in modo diverso a seconda dell'ora o del giorno in cui arriva la chiamata. E' possibile specificare intervalli di tempo utilizzando lo standard del *Internet Calendar and Sceduling Core Object Specification (iCalendar COS)* definito all'interno del RFC 2445 [22], in cui vengono definiti due formati, uno per specificare un istante di tempo o un'indicazione temporale (un dato giorno o mese, per esempio), mentre il secondo viene utilizzato per rappresentare delle durate. Utilizzando combinazioni di questi due elementi è possibile definire una serie di intervalli di tempo, anche ricorrenti, in modo estremamente flessibile.

All'interno del tag *time-switch* è possibile definire due attributi: *tzid* e *tzurl*, che possono contenere rispettivamente un *Time Zone Identifier* e un *Time Zone URL*, definiti entrambi nel RFC 2445 [22], vengono usati per specificare a quale zona oraria appartengono le indicazioni di tempo gestite da questo switch. Non specificando questi attributi si dà per scontato che tutte le ore specificate vengono calcolate in base alle impostazioni locali del sistema nel quale vengono utilizzati i profili.

Gli output di questo switch sono definiti dal tag *time*, per ognuno di questi output è possible definire una serie di intervalli di tempo. Il modo più semplice per utilizzare questo tag è specificare un singolo intervallo di tempo, questo può essere fatto in due modi, indicando l'istante di partenza e l'istante finale (usando gli attributi *dtstart* e *dtend*), oppure l'inizio e la durata dell'intervallo (con gli attributi *dtstart* e *duration*).

Indicazioni più complesse possono essere costruite utilizzando degli intervalli ricorrenti, questo viene fatto in primo luogo specificando l'attributo *freq* in cui viene specificata la frequenza di ripetizione. I valori utilizzabili possono essere: *secondly*, *minutely*, *hourly*, *daily*, *weekly*, *monthly* o *yearly*, in questo modo si dichiara che l'intervallo di tempo specificato viene considerato ripetuto rispettivamente ogni secondo, ogni minuto, ogni ora, ogni giorno, ogni settimana ogni mese ed ogni anno.

E' importante fare attenzione a non definire intervalli di durata tale che si sovrappongano durante le ripetizioni, cioè se si vuole che un intervallo venga ripetuto

ogni ora, questo non deve avere durata superiore all'ora.

E' possibile specificare ogni quanto tempo spesso l'intervallo viene ripetuto attraverso il parametro *interval*, cioè se vogliamo definire un intervallo che venga ripeuto ogni 2 settimane dovremmo inserire gli attributi:

```
... freq="weekly" interval="2" ...
```

mettendo il suo valore pari a 3 questo viene ripetuto ogni 3 settimane.

Se viene omesso l'attributo *freq*, ma vengono impostati degli altri attributi relativi alla ricorrenza degli intervalli, viene data per scontata una frequenza di tipo giornaliero.

Se si vuole dare un termine alla ripetizione degli intervalli si può utilizzare il parametro *until*, oppure *count*; il primo permette di specificare l'ultimo istante in cui viene considerato valido un intervallo, mentre il secondo permette di definire il numero di volte che questo deve venire ripetuto.

Esiste una serie di attributi con cui è possibile specificare delle liste di valori di tempo (ad esempio di giorni o di ore) in cui l'intervallo è valido. In particolare, si possono utilizzare i seguenti elementi:

bysecond si possono definire una serie di secondi, all'interno di un minuto, in cui la regola viene considerata valida, i secondi sono compresi tra 0 e 59

byminute contiene la lista dei minuti all'interno dell'ora, anch'essi compresi tra 0 e 59

byhour indica la lista di ore della giornata, valori inseribili sono compresi tra 0 e 23

byday indica la lista di giorni della settimana in cui la regola viene considerata valida, i valori possibili possono essere tra *MO, TU, WE,TH, FR, SA* e *SU* (che corrispondono rispettivamente a lunedì, martedì, mercoledì, giovedì, venerdì, sabato e domenica)

bymonthday indica la lista giorni del mese in cui la regola è verificata

byyearday indica la lista di giorni dell'anno, i valori sono compresi da 1 a 366

byweekno indica la lista di quali settimane all'interno dell'anno i valori vanno da 1 a 53

bymonth indica la lista di mesi all'interno dell'anno in cui la regola viene considerata valida

Quando si inseriscono più valori all'interno di uno di questi campi, questi vanno separati da virgole. E' possibile inserire delle combinazioni degli elementi precedenti per ottenere delle regole estremamente complesse e articolate.

Oltre agli attributi già visti ne esistono altri come *wkst* o *bysetpos*, per questi si rimanda al RFC di riferimento [11].

E' possibile utilizzare l'output *otherwise* nel caso il momento di arrivo della chiamata non coincida con alcun intervallo di tempo presente all'interno dello switch.

Priority Switch

Vengono utilizzati per permettere agli script CPL di prendere decisioni in funzione della priorità specificata dal UA chiamante.

Lo switch è definito dal tag:

Tag: *<priority-switch>*
Output: *priority,otherwise*

L'output utilizzato è *priority*, e può utilizzare uno tra questi parametri: *greater*, *less* e *equal*, che rendono verificata la condizione rispettivamente se la priorità specificata è maggiore, minore o uguale al valore espresso.

I valori possibili che possono essere utilizzati sono: *emergency*, *urgent*, *normal* e *non-urgent*, che corrispondono a priorità decrescenti. Nel caso non sia specificata alcuna priorità all'interno della chiamata, questa viene considerata di tipo *normal.*

3.3.2 Nodi di modifica delle destinazioni

Vengono utilizzata per modificare l'indirizzo originale a cui è indirizzata la chiamata, sono usati per reindirizzare le chiamate all'interno del dominio SIP (e non solo) per poter, ad esempio, avere all'interno del dominio degli indirizzi differenti da quelli visti dall'esterno. Può essere utilizzato per redirezionare le chiamate, ad esempio, a seconda dell'ora del giorno, ad un centralino o ad una casella vocale.

Funzionano in maniera diversa dai nodi switch in quanto (a parte un caso), non venendo effettuate delle scelte non sono presenti output, e quindi una volta eseguito il tag l'esecuzione passa direttamente al nodo sucessivo (cioè il nodo interno).

Location

Viene utilizzatore per aggiungere esplicitamente un indirizzo alla lista attuale degli indirizzi per la chiamata.

Tag:	*<location>*
Parametri:	*url*, *priority*,*clear*

L'indirizzo da aggiungere viene specificato all'interno del paramentro *url* secondo le regole definite dal protocollo sottostante. Un solo indirizzo può essere inserito in ogni nodo di tipo *location*, se si volessero agguingere più indirizzi è sufficiente inserire più nodi di questo tipo in cascata.

Il parametro *priority* specifica un valore numerico (decimale) che va da 0.0 a 1.0, dove 1.0 è la massima priorità. Viene utilizzato dal server per decidere l'ordine in cui eseguire le chiamate nel caso della presenza di più indirizzi. Nel caso non venga specificato viene preso in considerazione il valore 1.0.

Il parametro *clear* specifica se la lista dagli indirizzi già presente debba essere svuotata prima di aggiungere questo URL, i valori possibili sono *yes* o *no*. Specificando questo attributo automaticamente viene considerato questo indirizzo come la nuova destinazione della chiamata.

Lookup

Questo nodo viene utilizzato per modificare la lista delle destinazioni da una fonte esterna che depende, non tanto dal protocollo utilizzato, ma dal sistema software che gestisce lo script CPL.

Questo tag ha le seguenti caratteristiche:

Tag:	*<lookup>*
Output:	*success*, *notfound*, *failure*
Parametri:	*source*, *timeout*, *use*, *ignore*, *clear*

E' presente un solo parametro obbligatorio, *source*, nel quale viene specificato l'URL che deve essere interrogato per ricevere la lista degli indirizzi da aggiungere. In maniera simile al tag *location* è possibile specificare l'attributo *clear* per eliminare gli URL già presenti nella lista delle destinazioni.

Gli altri attributi sono specifici della fase di interrogazione, in particolare si ha *timeout* che specifica il numero di secondi che deve rimanere in attesa di una risposta,

scaduti i quali si considera la ricerca infruttuosa. Per gli altri attributi si rimanda al relativo RFC [11].

Dato che non è garantito che la richiesta abbia esito positivo, all'interno di questo tag sono presenti tre output che vengono scelti a seconda del risultato dell'operazione:

success se l'operazione è andata a buon fine e sono stati aggiunti degli indirizzi alla lista attuale

notfound se l'operazione è andata a buon fine, ma non sono stati ritornati indirizzi da aggiungere alla lista attuale

failure se ci sono stati errori nell'operazione di interrogazione, o se il timeout specificato è scaduto

Location Removal

Viene utilizzato per rimuovere indirizzi specifici dalla lista degli indirizzi, l'uso di questo nodo dipende fortemente dal protocollo sottostante e dall'applicazione che lo gestisce.

In particolare è descritto dal seguente tag:

Tag:	*<remove-location>*
Parametri:	*location*, *param*, *value*

L'argomento *location* specifica l'indirizzo o il pattern di indirizzi da rimuovere, nel caso non venga specificato vengono utilizzati gli altri parametri per specificare il comportamente di questo nodo.

Gli altri due parametri, *param* e *value*, specificano, rispettivamente, una serie di nomi di parametri e della corrispondente serie di valori, separati da virgole, che vengono passati al sistema che gestisce lo script CPL per specificare le modalità di eliminazione degli indirizzi. I parametri dipendono dal sistema usato e dal protocollo sottostante.

3.3.3 Nodi di segnalazione

Servono per effetuare delle operazioni relative al protocollo sottostante, è possibile inoltrare le chiamate, ridirezionarle o mandare risposte arbitrarie al chamante.

Proxy

Questo nodo inoltra la chiamata verso la lista delle destinazioni attualmente presente, il nodo ha le seguenti caratteristiche:

Tag:	*<proxy>*
Output:	*busy*, *noanswer*, *redirection*, *failure*, *default*
Parametri:	*timeout*, *recurse*, *ordering*

L'azione specifica intrapresa dal sistema quando deve eseguire questo nodo, dipende dal protocollo di segnalazione utilizzatto, nel caso di SIP viene mandato un pacchetto di INVITE a tutti gli indirizzi presenti all'interno della lista delle destinazioni, e si rimane in attesa di una risposta.

Dopo che l'operazione di *proxy* termina, il server CPL sceglie la risposta "migliore" tra quelle ricevute, utilizzando i criteri definiti dal server o dall'amministratratore del dominio.

Se la chiamata avviene con successo lo script CPL termina, altrimenti viene eseguito un output a seconda della situazione che si viene a creare. In particolare gli output possibili sono:

busy l'indirizzo a cui viene inoltrata la chiamata non è in grado di rispondere

noanswer non c'è stata alcuna risposta in tempo utile

redirection la chiamata è stata ridiretta ad un altro indirizzo

failure è stato ricevuto un messaggio di errore

default viene selezionato quando si verifica una situazione per cui non è stato specificato l'output

Se non viene specificato un output di tipo *default* e si verifica una situazione che non è gestita dagli altri output presenti, lo script termina.

All'interno del tag *proxy* è possibile specificare il tempo, passato il quale, la chiamata viene considerata senza risposta attraverso il l'attributo *timeout*; se questo non viene specificato, il valore di base deve essere considerato di 20 secondi.

Il parametro *recurse* (che può assumere i valori *yes* o *no*) serve per specificare se il server deve occuparsi automaticamente di effetture un successivo inoltro della chiamata, nel caso che la risposta ricevuta dal destinatario sia di redirezione: in

questo caso non verrà mai scelto l'output *redirection* in quanto viene gestito automaticamente dall'impostazione *recurse*.

Il parametro *ordering* specifica come verranno eseguite le chiamate nel caso ci sia più di un indirizzo nella lista delle destinazioni. I valori possibili che può assumere sono:

parallel vengono eseguite tutte le chiamate contemporaneamente

sequential le chiamate vengono eseguite una dopo l'altra in sequenza seguendo il valore di priorità che è stato loro assegnato

first-only viene eseguita solamente la chiamata verso l'indirizzo con maggior priorità

Il valore predefinito nel caso questo parametro non venga specificato è *parallel*.

Redirect

Questo nodo istruisce il server CPL di rispondere al chiamante di redirezionare la chiamata trasmettendo la lista delle destinazioni attuale. Le caratteristiche di questo nodo sono:

Tag:	<*redirect*>
Parametri:	*permanent*

Appena viene eseguito, questo tag, termina l'esecuzione dello script CPL, di conseguenza questo è un nodo terminale, in quanto non può avere né output né nodi successivi. Il parametro *permanent*, che può assumere i valori *yes* o *no*, indica se la risposta al chiamante deve indicare se la redirezione è permanente o meno.

Reject

Quando viene incontrato questo nodo viene spedito un messaggio di rifiuto della chiamata a chi l'ha generata. Le caratteristiche di questo nodo sono:

Tag:	<*reject*>
Parametri:	*status,reason*

Anche questo nodo, appena viene eseguito, termina l'esecuzione dello script. E' necessario specificare il codice del motivo del rifiuto attraverso il parametro *status*, i valori possibili sono:

busy il destinatario è occupato

notfound il destinatario non è stato trovato

reject la chiamata è stata rifiutata

error è stato generato un errore

E' possibile inserire come valore un codice specifico di un protocollo, in particolare se il server CPL si appoggia a SIP è possibile inserire un codice numerico tra quelli descritti nel paragrafo 2.3.2.

Oltre al codice dell'errore è possibile inviare anche una descrizione testuale del motivo attraverso il parametro *reason*.

3.3.4 Nodi generici

Oltre ai tipi di nodi già visti esistono altri nodi che effettuano altri tipi di operazioni.

Mail

Questo nodo notifica al server di informare l'utente dello stato dello script CPL attraverso una email. Le sue caratteristiche sono:

Tag:	<*mail*>
Parametri:	*url*

L'unico attributo che viene utilizzato è *url*, in cui va indicato l'indirizzo di posta elettronica destinazione come URL (specificando il protocollo *mailto*). E' possibile specificare altre informazioni, come l'oggetto della email, attraverso i parametri dell'URL.

All'interno della email deve essere descritto lo stato attuale dello script CPL e della chiamata che l'ha generata.

Log

Questo nodo richiede al server di memorizzare delle informazioni sulla chiamata all'interno di un registro, le sue caratteristiche sono:

Tag:	*<log>*
Parametri:	*name,comment*

Questo nodo accetta due argomenti, entrambi opzionali: *name* in cui viene specificato il nome del registro, e *comment* in cui viene inserito un commento. Il server dovrebbe inoltre includere altre informazioni riguardo alla chiamata.

Sub

Questo nodo esegue un determinata procedura definita attraverso un tag *subaction* all'interno dello stesso script. Il tag è definito nel seguente modo:

Tag:	*<sub>*
Parametri:	*ref*

L'esecuzione viene passata alla *subaction* identificata con il nome che viene specificato all'interno del parametro *ref*. Questo nodo non può avere altri elementi CPL al suo interno, in quanto le operazioni che seguono si trovano all'interno della relativa *subaction*. Si veda per altre informazioni il paragrafo 3.3.5.

3.3.5 Altri nodi

Oltre ai tag principali *incoming* e *outgoing*, è possibili avere direttamente nell'elemento radice (*cpl*) dello script altri due tag, il primo *subaction* viene utilizzato per definire delle procedure personalizzate, il secondo *ancillary* viene utilizzato per definire delle estensioni del linguaggio CPL.

Subaction

Come già visto in precedenza esiste la possibilità di definire delle procedure che possono venir chiamate da qualunque punto dello script. Le caratterische di questo tag sono:

Tag:	*<subaction>*
Parametri:	*id*

Questi procedure hanno una limitazione: non possono essere richiamate più di una volta all'interno della stessa esecuzione dello script, questo per evitare la possibilità di avere cicli di lunghezza indeterminata. Quindi è necessario, nel momento in cui lo script viene caricato nel server, che venga controllato che questo vincolo sia rispettato, ed in caso contrario rifiutare lo script.

Il parametro *id* viene utilizzato per definire il nome della procedura, che viene poi utilizzato all'interno del parametro *ref* del nodo *sub*, i nomi sono *case-sensitive* [4].

Ancillary

Questo nodo è stato definito per poter aggiugere delle estensioni al linguaggio CPL, attualmente non è stato ancora utilizzato.

3.3.6 Comportamento predefinito

Quando all'interno di un tag non sono presenti dei nodi successivi, o non sono presenti degli output, automaticamente lo script termina e viene utilizzato un comportamento predefinito, che in realtà cambia a seconda del contesto dello script.

In particolare possono succedere le seguenti cose:

- Se non sono stati modificati gli indirizzi di destinazione, non è stata eseguita nessuna operazione di segnalazione e la lista delle destinazioni è vuota: allora viene passato il controllo al proxy che si occupa di utilizzare la strategia predefinita per trattare le richieste in arrivo.

- Se ci troviamo nella situazione precedente ed in più la lista delle destinazioni non è vuota viene inoltrata la chiamata verso gli URL presenti.

- Se sono state effettuate delle modifiche agli indirizzi, ma nessuna operazione di segnalazione si inoltra la chiamata, se invece la lista è vuota viene rimandato il messaggio di tipo *notfound* al chiamante.

Esistono altri casi particolari, per i quali si rimanda al relativo RFC [11].

[4] Si definisce *case-sensitive* un campo di tipo stringa in cui è rilevante la differenza tra le lettere minuscole e le maiuscole

3.4 Esempi

Per comprendere meglio il funzionamento di questo linguaggio è utile analizzare qualche esempio.

3.4.1 Inoltro semplice della chiamata

Lo script più semplice che si occupa semplicemente di inoltrare la chiamata all'indirizzo specificato può avere questa forma:

```
<cpl>
  <incoming>
    <proxy />
  </incoming>
</cpl>
```

L'unico nodo è *proxy* che inoltra la chiamata.

3.4.2 Redirezione della chiamata

Se invece di inoltrare direttamente la chiamata si vuole rispondere con un messaggio di redirezione si può fare in questo modo:

```
<cpl>
  <incoming>
    <redirect />
  </incoming>
</cpl>
```

3.4.3 Inoltro con cambio di indirizzo

Per mappare un indirizzo fittizio nell'indirizzo reale è necessario modificare la lista degli indirizzi utilizzando i nodi per la modifica delle destinazioni, in particolare il nodo *location*.

```
<cpl>
  <incoming>
```

```
    <location url="sip:indir.reale@company.com" clear="yes">
      <proxy />
    </location>
  </incoming>
</cpl>
```

In questo modo la richiesta fatta verso l'indirizzo a cui è registrato questo script viene inoltrata all'indirizzo *sip:indir.reale@company.com*. Il parametro *clear* serve per eliminare la destinazione precedente dalla lista.

3.4.4 Cambio di indirizzo in funzione del ora di arrivo

Se si volesse eseguire operazioni diverse a seconda del momento di arrivo della chiamata è necessario utilizzare un *time-switch*.

Nel seguente esempio a partire dal primo giugno 2003, tutti i giorni dalle 8 del mattino, per 8 ore, le chiamate elaborate da questo script vengono inoltrate all'indirizzo specificato all'interno del tag *location*. Tutte le altre chiamate, benché non vi sia nulla di espressamente specificato, vengono inoltrate all'indirizzo originale, questo per quanto detto nel paragrafo 3.3.6 sul comportamento predefinito.

```
<cpl>
  <incoming>
    <time-switch>
      <time dtstart="20030601T080000" duration="PT8H" freq="daily">
        <location url="sip:indir.reale@company.com" clear="yes">
          <proxy />
        </location>
      </time>
    </time-switch>
  </incoming>
</cpl>
```

Se si volesse modellare il comportamento con maggiore precisione si potrebbero utilizzare più output di tipo *time* all'interno dello switch:

```
<cpl>
```

```
  <incoming>
    <time-switch>
      <time dtstart="20030601T080000" duration="PT8H"
                      freq="daily" byday="MO,TU,WE,TH,FR">
        <location url="sip:indir.reale@company.com" clear="yes">
          <proxy />
        </location>
      </time>
      <time dtstart="20030601T000000" duration="PT24H"
                      freq="daily" byday="SA,SU">
        <reject status="reject" reason="In vacanza" />
      </time>
    </time-switch>
  </incoming>
</cpl>
```

Nell'output *time* originale è stato aggiunto l'attributo *byday* che lo rende valido nei giorni dal lunedì al venerdì, mentre il secondo *time* viene verificato tutti i sabati e le domeniche, in questo caso non inoltrando più la chiamata ma rifiutandola direttamente (attraverso il nodo *reject*). Negli altri caso viene applicato il comportamento predefinito del server.

3.4.5 Uso del tag *proxy*

Finora non ci si è preoccupati di intraprendere altre azioni nel caso in cui la chiamata non andasse a buon fine. Questo è fattibile gestendo gli output del tag *proxy*.

```
<cpl>
 <incoming>
   <proxy>
      <busy>
        <location clear="yes" url="sip:me@forwarded.address">
          <proxy />
        </location>
      </busy>
```

```
    </proxy>
  </incoming>
</cpl>
```

Il primo *proxy* inoltra la chiamata, nel caso sia occupato (output *busy*) si prova ad inoltrare la chiamata ad un altro indirizzo. E' possibile gestire anche altri casi utilizzando gli output visti nel paragrafo 3.3.3.

3.4.6 Uso delle *subaction*

Con le *subaction* è possibile definire delle parti di codice richiamabili da diverse punti dello script, in particolare in questo esempio è stata definita l'azione *voicemail*, che presumibilmente redirige la chiamata verso una casella vocale.

```
<cpl>
  <subaction id="voicemail">
    <location url="sip:jones@voicemail.example.com" clear="yes">
      <redirect />
    </location>
  </subaction>

  <incoming>
    <address-switch field="origin" subfield="host">
      <address subdomain-of="example.com">
        <proxy timeout="10">
          <busy> <sub ref="voicemail" /> </busy>
          <noanswer> <sub ref="voicemail" /> </noanswer>
          <failure> <sub ref="voicemail" /> </failure>
        </proxy>
      </address>
      <otherwise>
        <sub ref="voicemail" />
      </otherwise>
    </address-switch>
  </incoming>
</cpl>
```

In questo caso, se il chiamante appartiene al dominio *example.com*, viene inoltrata la chiamata all'indirizzo destinazione, in caso di insuccesso la chiamata viene rediretta alla casella vocale. Nel caso la chiamata non arrivi da quel dominio viene subito girata verso la casella vocale.

Parte II

Realizzazione del progetto

Capitolo 4

Architettura del servizio

4.1 Criteri di progettazione

4.1.1 Obiettivi

L'obiettivo che ci si è posti in questa tesi è di creare un meccanismo di gestione di profili per un proxy SIP basato sul linguaggio CPL secondo le seguenti linee guida:

- Velocità di esecuzione, intesa come rapidità nella risposta alla chiamata,in quanto la gestione del profilo avviene quasi esclusivamente in questa fase, dato che il proxy deve decidere come comportarsi di fronte della richiesta di avvio di una comunicazione.

- Estendibilità, il sistema deve poter essere modificato facilmente, sia per poter aggiungere nuove funzionalità, in quanto il linguaggio CPL gestisce la possibilità di utilizzare delle estensioni, sia per poter essere adattato in altri contesti.

- Interoperabilità, il sistema realizzato deve poter essere facilmente integrato all'interno di scenari NGN potendo comunicare efficacemente con gli altri componenti seguendo meccanismi standard.

Tutto questo appoggiandosi a funzionalità avanzate del linguaggio Java, come il caricamento dinamico delle classi.

4.1.2 Soluzione

La soluzione che si è scelto di adottare si basa un proxy già esistente, in particolare l'implementazione effettuata da NIST [17] dello standard JAIN SIP (paragrafo 2.4.1). Per la gestione dei profili sono stati sviluppati due componenti indipendenti:

ProfileServerManager è un componente isolato dal proxy che si occupa di ricevere i profili scritti in linguaggio CPL, questi vengono convertiti dinamicamente in oggetti Java e vengono memorizzati all'interno di un database.

ProfileManager è integrato all'interno del proxy SIP e si occupa di gestire gli oggetti profilo (che vengono creati dal **ProfileServerManager**), gestendo la richiesta e l'immagazzinamento temporaneo di questi in funzione delle richieste pervenute al proxy.

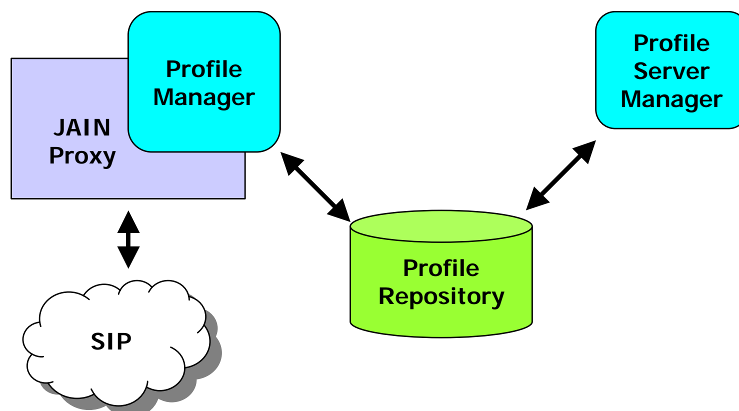

Figura 4.1. Macro architettura del sistema

La gestione della conversione dei file CPL in oggetti Java viene fatta creando ad hoc una classe Java per ogni documento CPL inserito. Questo significa che il **ProfileServerManager** analizza il profilo e in base a questo crea una classe Java, la compila e la memorizza. Mentre il **ProfileManager** è in grado di caricare dinamicamente (a runtime[1]) una la classe e di istanziarne un oggetto.

[1] A tempo di esecuzione, cioè mentre il programma è in esecuzione, senza la necessità di riavviarlo.

I profili creati dal **ProfileServerManager** vengono memorizzati in un repository[2], e da questo caricati dal **ProfileManager** quando il proxy lo richiede.

4.1.3 Motivazioni

Attualmente i server CPL presenti sul mercato (ad esempio DynamicSoft o StarSIP) funzionano seguendo due strategie diverse:

- ogni volta che arriva una chiamata viene letto lo script CPL e poi interpretato direttamente
- viene eseguita una chiamata remota ad un server che gestisce effettivamente i profili

Entrambe le soluzioni rallentano significativamente il tempo di attivazione della chiamata: questo è dovuto al fatto di dover interpretare ogni volta il file CPL, operazione lenta in quanto ogni volta è necessario rielaboare la struttura XML del documento, nell'altra soluzione la necessità di creare una chiamata a procedura remota è un'operazione anch'essa costosa in termini di tempo.

Per evitare questi tipi di colli di bottiglia si è scelto di creare, a partire da un profilo in formato CPL, un oggetto Java puro, del tutto equivalente al profilo specificato (ovviamente dal punto di vista semantico) che sia in grado di essere utilizzato dal proxy per gestire direttamente l'attivazione della chiamata.

In questo modo la parte più esosa dal punto di vista computazionale, cioè l'interpretazione del documento CPL, avviene nel momento in cui il profilo viene modificato, e solamente in quel momento. Il **ProfileServerManager** ha al suo interno un vero e proprio compilatore da CPL verso Java, che genera un classe Java direttamente utilizzabile: questa operazione è sicuramente più lunga della sola interpretazione del profilo CPL, però il vantaggio è che viene eseguita un'unica volta in maniera asincrona rispetto alle chiamate del proxy.

Quindi il proxy, appena riceve una chiamata, deve soltanto caricare il profilo dal repository e istanziare l'oggetto profilo che si occuperà di gestire la chiamata, questo soltanto la prima volta, in quanto se l'oggetto è già stato creato viene utilizzato direttamente per tutte le chiamate successive, senza più accedere al repository.

[2] Un repository si occupa di immagazzinare e conservare in modo centralizzato una serie di entità, in questo caso le classi dei profili.

Appare chiaro come questo tipo di soluzione garantisca un velocità di attivazione della chiamata di gran lunga maggiore rispetto alle soluzioni tradizionali, in quanto le operazioni più costose vengono eseguite non all'arrivo della chiamata, ma alla definizione del nuovo profilo.

4.2 Architettura

Verrà ora analizzata l'architettura del sistema nel dettaglio, in particolare è stata fatta molta attenzione per rendere tutto il sistema il più possibile modulare così da facilitare il riuso e l'aggiornamento dei vari componenti.

Come detto in precedenza è possibile riconoscere due componenti principali che si occupano, in maniera indipendente, di affrontare i due aspetti del problema: la gesitone della chiamata e la gestione dei profili.

4.2.1 ProfileManager

Il **ProfileManager** è parte integrante del **JAIN-Proxy** e si occupa di recuperare l'oggetto che gestisce un determinato profilo e lo restituisce alla sezione del **JAIN-Proxy** che si occupa di elaborare chiamate in ingresso.

Internamente il **ProfileManager** è formato da diverse componenti:

- il collegamento al repository dei profili;
- il decodificatore di profili;
- la cache locale che memorizza i profili utilizzati;

Nel momento in cui arriva la chiamata viene controllato prima di tutto se nella cache locale è presente il profilo cercato, in caso contrario la richiesta viene fatta al repository utilizzato che restituisce il profilo codificato.

Prima di poterlo utilizzare questo viene decodificato dal relativo componente (per una spiegazine approfondita su come vengono gestiti e memorizzati i profili si veda il paragrafo 4.3), quindi memorizzato nella cache e poi restituito al proxy.

Se il profilo è già presente nella cache, viene utilizzato direttamente senza inutili accessi al repository.

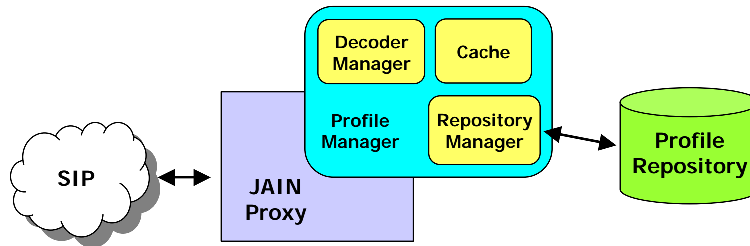

Figura 4.2. Architettura del **ProfileManager**

La gestione della cache è importante in quanto da questa dipende l'occupazione della memoria e la velocità di risposta del sistema, la politica di base utilizzata si basa sui criteri che controllano la quantità di profili memorizzati e la data del loro ultimo utilizzo. In particolare quando si supera una determinata soglia di profili memorizzati la cache attiva una procedura che si occupa della sua pulizia: questa procedura inizia a cercare i profili che hanno le date di utilizzo più vecchie e si occupa di eliminarli. In questo modo la memoria viene liberata nel momento in cui il *Garbage Collector* della *JVM* (Java Virtual Machine) lo ritiene necessario.

La modalità di gestione della cache è modificabile in quanto è possibile specificare la classe che si occupa di gestire la pulizia, in questo modo è possibile modificare i criteri descritti per avere una gestione personalizzata in base alle esigenze dell'utilizzo. Siccome le prestazioni (sopratutto dal punto di vista dell' occupazione di memoria) cambiano a seconda del tipo di traffico e di utenza di cui è interessato il proxy, in questo modo si può adattare il comportamento al contesto incui viene fatto il *deployment*.

L'integrazione con il proxy è uno dei punti critici di questo sistema in quanto si è puntato ad eseguire modifiche mirate e facilmente replicabili, per fare in modo che sia possibile adattarle facilmente nel caso vengano rilasciate altre versioni del proxy. E' facile immaginare come i componenti sviluppati siano molto dipendenti dall'architettura del proxy, in quanto questo non è stato progettato per accogliere moduli esterni, e di conseguenza per evitare di fare troppe modifiche nell'implementazione fornita dal NIST le attività più complesse sono state spostate nei componenti aggiunti.

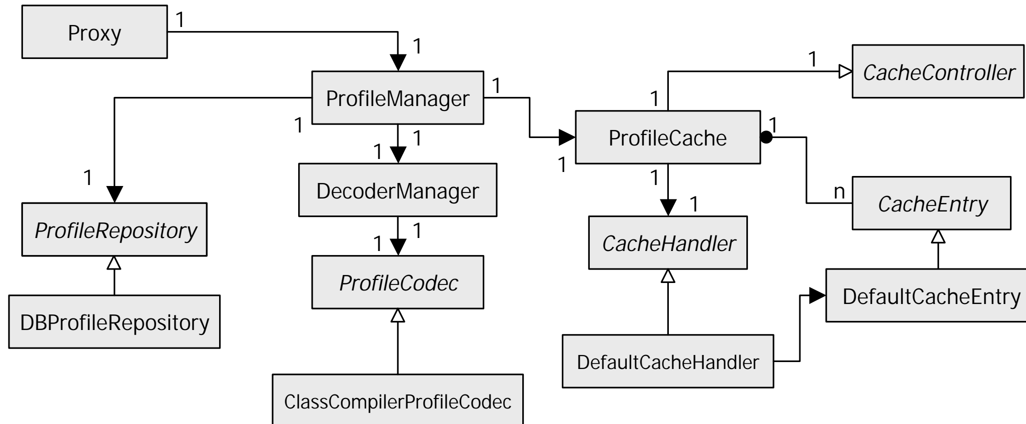

Figura 4.3. Diagramma della classi del **ProfileManager**.

4.2.2 ProfileServerManager

Il **ProfileServerManager** è il componente che si occupa di amministrare i profili, viene eseguito all'interno di un web server che supporta le JSP (Java Server Pages), in quanto tutta l'interfaccia di gestione fornita all'amministratore del sistema viene gestita via web.

Il **ProfileServerManager** è composto dalle seguenti parti:

- l'interfaccia web
- il compilatore CPL
- il codificatore di profili
- il gestore del repository

L'interfaccia permette all'amministratore di creare nuovi profili, mostrare quelli già registrati, modificarli e cancellarli. Per la creazione e la modifica viene utilizzato un form in cui si modifica lo script CPL, è previsto anche un aiuto grafico per la creazione del profilo basato su un applet, anche se questo, per via della flessibilità del linguagguo CPL, sa gestire solo una quantità limitata di casistiche.

Tutta le funzioni di accesso ai profili esistenti vengono eseguite attraverso il gestore del repository, che si occupa di leggere, modificare e memorizzare tutte le informazioni che dovranno poi essere utilizzate anche dal **ProfileManager**. All'interno

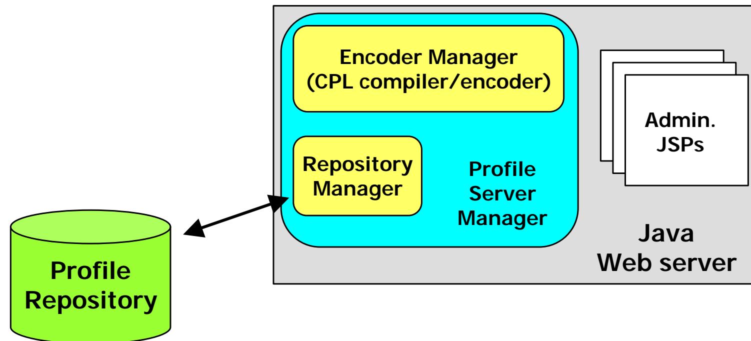

Figura 4.4. Architettura del **ProfileServerManager**

del repository vengono salvati tutti i dati relativi ai profili, come il nome del profilo, la data di ultima modifica, il profilo compilato e il codice sorgente CPL. Ovviamente a seconda del componente verranno utilizzate soltanto alcune informazioni.

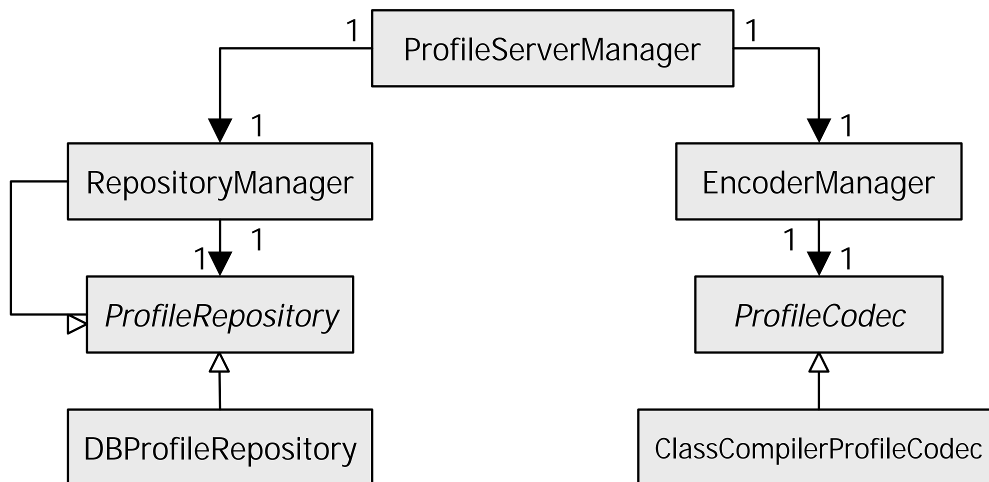

Figura 4.5. Diagramma della classi del **ProfileServerManager**.

Lo scopo principale del **ProfileServerManager** è di trasformare gli script CPL in classi Java utilizzabili dal proxy, per fare questo è presente un vero e proprio compilatore da CPL a Java che trasforma i documenti XML con i profili in bytecode

Java. Questo procedimento avviene in due fase, la prima in cui avviene il *parsing*[3] del CPL e la seconda in cui viene creato il profilo vero e proprio(per un'analisi approfondita si veda il paragrafo 4.3).

4.2.3 ProfileRepository

Il repository dei profili è gestito in modo generico attraverso un'interfaccia (*ProfileRepository*) che deve essere implementata da un componente vero e proprio. Le funzionalità che mette a disposizione permettono di:

- Inserire un nuovo profilo ed il relativo CPL nel repository
- Recuperare un profilo
- Recuperare un documento CPL
- Cancellare un profilo
- Ottenere la lista dei profili memorizzati

In questo modo è possibile creare diversi tipi di repository a seconda delle specifiche esigenze. In particolare è stato implementato il **DBProfileRepository**, che attraverso l'architettura JDBC (Java DataBase Connectivity) memorizza i profili all'interno di un generico database.

Con questo tipo di architettura, il sistema è slegato da un particolare meccanismo di immagazzinamento dei profili, quindi è possibile adattare questo sistema integrandolo alle tecnologie di una realtà già esistente. E' possibile ad esempio creare un'implementazione che si appoggi a LDAP (Lightweight Directory Access Protocol) per sfruttare un meccanismo di directory per gestire i profili, oppure un'implementazione che utilizzi il protocollo HTTP per richiedere i profili ad un servlet, nel caso i due componenti si trovino in sezioni diverse di una rete separate da un firewall.

[3] Con il termine *parsing* si intende la fase dell'analisi di un documento o programma, in cui viene effetuata la scomposizione della sua struttura nei sui elementi base seguendo le sue regole sintattiche.

4.3 Gestione del profilo

I profili, dal punto di vista del NIST proxy, risultano degli oggetti che implementano l'intefaccia *Profile*, il come vengono implementati questi oggetti dipende esclusivamente dal tipo di *codec*[4] utilizzato.

Questi codec implementano l'interfaccia *ProfileCodec*, e sono in grado, nella fase di codifica, di trasformare uno script CPL[5] in un oggetto di tipo **ProfileObject**, che viene utilizzato come formato intermedio comune a tutti i codec, per contenere il profilo. Questo tipo di oggetto viene utilizzato per la gestione e la memorizzazione dei profili all'interno del repository.

La fase di decodifica avviene all'interno del **ProfileManager** in cui il **ProfileObject** ottenuto dal repository viene convertito dal *ProfileCodec* utilizzato nella relativa istanza dell'interfaccia *Profile*, pronto per essere utilizzato dal proxy.

Figura 4.6. Fasi della gestione dei profili.

Con questo tipo di architettura la gestione dei profili è assolutamente trasparente rispetto al tipo di strategia utilizzata per creare e codificare i profili, anche il tipo di codec utilizzato viene specificato all'interno di un file di configurazione e quindi è possibile, senza dovere ricompilare il sistema, cambiare il tipo di formato utilizzato.

Dato che all'interno sia del **ProfileManager** che del **ProfileServerManager** i *ProfileCodec* non vengono utilizzati direttamente ma rispettivamente all'interno di un **DecoderManager** e un **EncoderManager**, esiste la possibilità di utilizzare più di un codec contemporaneamente[6], questo perché all'interno del **ProfileObject** che viene utilizzato come contenitore per i profili codificati, viene memorizzzata anche

[4] In generale si definisce un codec (abbreviazione di COder DECoder) un componente in grado di codificare e succesivamente decodificare in un particolare formato un tipo di dato, in questo caso si parla di oggetti Java.

[5] Lo script CPL viene memorizzato come testo all'interno di un oggetto di tipo **CPLProfile**.

[6] Attualmente è necessaria una serie di piccole modifiche, ma che in ogni caso non cambiano l'architettura del sistema.

l'informazione sul codec usato. In questo modo è impossibile tentare di decodificare un profilo usando un *ProfileCodec* differente da quello utilizzato per codificarlo (evenienza che può essere causata soltanto da un'incongruenza delle configurazioni del **ProfileManager** e del **ProfileServerManager**).

In fase di progetto sono state prese in considerazione due possibili soluzioni:

- Creazione dinamica di classi Java runtime in funzione del profilo CPL inserito, caricate dal **ProfileManager** attraverso un meccanismo di *Class Loading* dinamico.
- Creazione di oggetti composti che simulano il comportamento dello script CPL, e che rimangono persistenti all'interno del sistema attraverso il meccanismo della serializzazione.

Verranno ora trattate nel dettaglio entrambe le soluzioni per analizzare gli effettivi vantaggi e svantaggi.

4.3.1 Class Loading dinamico

In questa soluzione quando viene richiesta la codifica di un documento CPL viene creata una classe Java ad hoc per questo specifico profilo, durante la codifica viene istanziato un oggetto di questa classe che risulta essere il profilo vero e proprio. Ovviamente affinché funzioni è necessario che la classe creata implementi l'interfaccia *Profile*.

La fase di codifica segue una serie di passi:

1. Viene analizzato il documento CPL sorgente e di questo viene creato in memoria il relativo *AST*[7] (Abstract Syntax Tree).
2. Si verifica la correttezza semantica del programma.
3. Viene creato un file sorgente Java al cui interno vengono tradotte (in linguaggio Java) le istruzioni presenti nel file CPL.

[7] L'AST di un programma è la rappresentazione gerarchica degli elementi del linguaggio presenti nel programma, in questo modo viene messa in evidenza la sua struttura sintattica. Nella sua costruzione è possibile verificare la correttezza sintattica del documento e costruendo opportunamente gli oggetti che costituiscono i suoi nodi si può controllare anche la validità semantica.

4. Il file sorgente viene compilato in bytecode utilizzando un compilatore Java standard.

5. Il file contente il bytecode viene letto e il contenuto memorizzato all'interno di un **ProfileObject**.

Alla fine del processo il **ProfileObject** creato contiene in formato binario la classe Java memorizzata nel relativo file *.class*. Il criterio utilizzato per dare il nome alla classe corrisponde all'identificativo SIP dell'utente a cui appartiene il profilo più una sequenza numerica pseudo casuale (viene utilizzata una porzione della sequenza numerica generata dall'orologio del sistema indicante l'istante di tempo attuale), questo per non correre il rischio di creare due classi differenti con lo stesso nome (si veda più avanti sul caricamento delle classi).

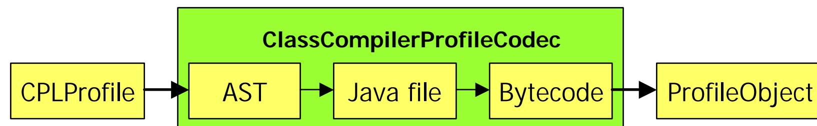

Figura 4.7. Fasi di un profilo durante la codifica attraverso la compilazione dinamica.

L'istanziazione del oggetto *Profile* utilizzato dal proxy viene eseguito nella fase di decodifica, in particolare in questa fase vengono eseguite le seguenti operazioni:

1. Viene istanziato un *classloader* modificato opportunamente (**CodecClassLoader**) per lavorare con i **ProfileObject**.

2. Viene richiesto il caricamento della classe relativo al profilo attraverso il classloader modificato.

3. Dalla classe creata, attraverso meccanismi di *reflection* Java viene istanziato un oggetto *Profile*.

Nell'architettura Java, il *classloader* è un componente implemenentato come classe Java, che si occupa di caricare in memoria le classi normalmente presenti all'interno di un disco sotto forma di file bytecode (file *.class*) e di renderli disponibili alla JVM per poi essere utilizzate nell'esecuzione di un programma. Esistono classloader

modificati per caricare le classi attraveso una connessione di rete, che vengono usati ad esempio negli applet, dato che i binari Java sono pubblicati all'interno di un server web.

In generale è possibile scrivere un classloader personalizzato che si occupi di caricare le classi in modo arbitrario e fare in modo che un programma lo utilizzi in determinati frangenti. Per quel che riguarda la decodifica del profilo si utilizza il **CodecClassLoader** che è in grado di leggere il bytecode delle classi direttamente all'interno di un **ProfileObject**, in questo modo è possibile aggiungere delle classi alla JVM durante l'esecuzione del programma, classi che possono venir create e compilate anche mentre il programma è già in corso.

Ci sono comunque dei vincoli da rispettare anche utilizzando un classloader modificato, ad esempio non è lecito caricare all'interno di uno stesso classloader due classi con lo stesso nome (o esaminato da un altro punto di vista due implementazioni diverse della stessa classe), questo per evitare ambiguità, o evitare che il codice di una classe cambi mentre esistono già delle istanze all'interno della JVM. Per questo motivo è necessario, nel momento in cui viene modificato il profilo di un utente, cambiare il nome della classe che lo implementa, in questo modo è impossibile che ci siano conflitti all'interno dello spazio dei nomi del classloader.

Il conflitto sui nomi è possibile solo all'interno di uno stesso classloader, infatti è possibile caricare due classi con lo stesso nome con classloader differenti, ma dato che nel meccanismo di decodifica viene utilizzato un solo classloader per tutti i profili, è necessario che le rispettive classi abbiano nomi univoci. La scelta di utilizzare un classloader comune è dovuta a considerazioni sull'occupazione della memoria, cioé se per ogni profilo utilizzato fosse caricato un classloader privato questo significherebbe che in scenari con centinaia di utenti la quantità di memoria occupata aumenterebbe senza vantaggi significativi.

Il vantaggio principale di questo tipo di codec è la significativa velocità di esecuzione del profilo, in quanto è implementato direttamente in Java e quindi non è necessaria alcuna interpretazione aggiuntiva. Inoltre in questo modo è possibile modificare anche drasticamente la struttura dei profili generati senza che questo si debba rispecchiare in alcun modo nel proxy che li utilizza.

Dal lato opposto, la quantità di memoria occupata risulta maggiore che in altre soluzioni, in quanto per ogni profilo viene caricata dalla JVM una classe diversa, in scenari in cui i profili cambino di frequente sarebbe necessaria una politica di

gestione diversa del classloader per poter liberare la memoria dagli oggetti e dalle classi non più utilizzate.

4.3.2 Serializzazione

La seconda seconda soluzione presa in considerazione si basa sulla creazione di una serie di classi che corrispondano agli elementi del linguaggio CPL, in questo modo l'oggetto che implementa l'interfaccia *Profile* contiene un assemblamento di questi oggetti organizzati in maniera analoga alla struttura dello script CPL a cui corrisponde.

In questo modo ogni profilo si creerebbe una specie di metalinguaggio fatto di oggetti Java e durante l'esecuzione della gestione della chiamata il controllo passerebbe, di volta in volta, ad ognuno di questi, in modo tale che ognuno esegua l'equivalente dell'istruzione CPL a cui corrisponde.

In questo modo durante la codifica vengono eseguiti i seguenti passi:

1. Viene analizzato il documento CPL e viene creato in memoria il relativo AST con un procedimento del tutto analogo alla soluzione precedente.

2. Viene eseguito il controllo semantico.

3. Si assembla il profilo così come verrebbe utilizzato nel proxy ricalcando la struttura del AST.

4. L'oggetto ottenuto viene serializzato e memorizzato all'interno del **ProfileObject**.

La struttura dell'oggetto costrutito è del tutto analoga a quella del relativo AST, in quanto questo rispecchia la struttura sintattica del programma originale; è possibile costruendo opportunamente gli oggetti che vengono utilizzati per l'AST, che ognuno di questi si occupi a generare il relativo equivalente utilizzato per la costruzione del profilo vero e proprio.

In questo modo la fase di codifica risulterebbe estremamente più semplice rispetto alla soluzione con la creazione dinamica delle classi, in quanto una volta effettuato il parsing e costruito l'AST si può costruire direttamente la struttura del profilo senza altre elaborazioni complesse.

Anche la fase di decodifica sarebbe estremamente semplice in quanto si tratta semplicemente di deserializzare l'oggetto immagazzinato nel relativo **ProfileObject** e restituirlo al **ProfileManager**.

Lo svantaggio di questo tipo di soluzione, rispetto a quella precedente, è che l'elaborazione del gestione della chiamata per il profilo non avverrebbe direttamente attraverso una esecuzione diretta di codice Java, ma indirettamente navigando la struttura sintattica di oggetti che costituisce il profilo, in questo modo le prestazioni risulterebbero penalizzate.

Come vantaggio, a parte la semplicità nella fase di codifica e di decodifica dal punto di vista dello sviluppo, si avrebbe un parziale risparmio di memoria rispetto alla soluzione precedente, in quanto non viene caricata una classe per ogni profilo, ma il numero di classi caricate nella JVM risulta costante: un risparmio di memoria parziale perché comunque, per ogni profilo, non verrebbe istanziato un solo oggetto, ma tutta una serie a seconda della complessità dello script CPL a cui corrisponde.

4.3.3 Soluzione adottata

Tra le due soluzioni si è scelto di implentare la prima, cioè di creare dinamicamente le classi contenti i profili. La scelta è stata fatta seguendo i criteri presentati nel paragrafo 4.1.1, in particolare riguardo alla velocità di esecuzione nell'attivazione di una chiamata, in più un meccanismo di questo tipo può risultare più flessibile nel caso di cambiamenti delle implementazioni dei profili ed inoltre può essere più facilmente utilizzato anche in ambiti diversi rispetto alla creazione di profili per proxy SIP.

Architettura del codec

Il codec è diviso in due parti ben distinte, sia dal punto di vista logico che da quello funzionale, una si occupa della codifica del profilo e l'altra della decodifica.

Per quel che riguarda la parte di codifica, il codec utilizza internamente una serie di componenti che si occupano di affrontare le varie fasi dell'eleborazione, in particolare:

- Un parser CPL che si occupa di leggere il documento XML, controllarne la validità e costruire il relativo AST.

- L'assemblatore di profili che, a partire dall'AST, crea il file sorgente Java con le funzionalità del profilo

- Il compilatore Java che si occupa di convertire il sorgente Java nel relativo bytecode

Il parser CPL (**CPLParser**) analizza il documento XML prima di tutto facendo eseguire la validazione attraverso il relativo DTD[8] (si veda l'appendice B), questo passaggio semplifica notevolmente il controllo sintattico dello script, oltre a questo vengono comunque eseguiti altri controlli come sulla validità dei parametri o sull'assenza di cicli (si veda il capitolo su CPL 3).

Per quanto riguarda la creazione dell'AST il *CPLParser* si appoggia al **ClassCompilerNodeBuilder** che si occupa di generare i nodi dell'AST utilizzati dal codec, in generale è possibile scrivere un differente *CPLNodeBuilder* e un differente set di nodi in modo da poter utilizzare lo stesso parser anche su implementazioni differenti, così che se si volesse implementare un altro tipo di codec è possibile riutilizzare lo stesso parser.

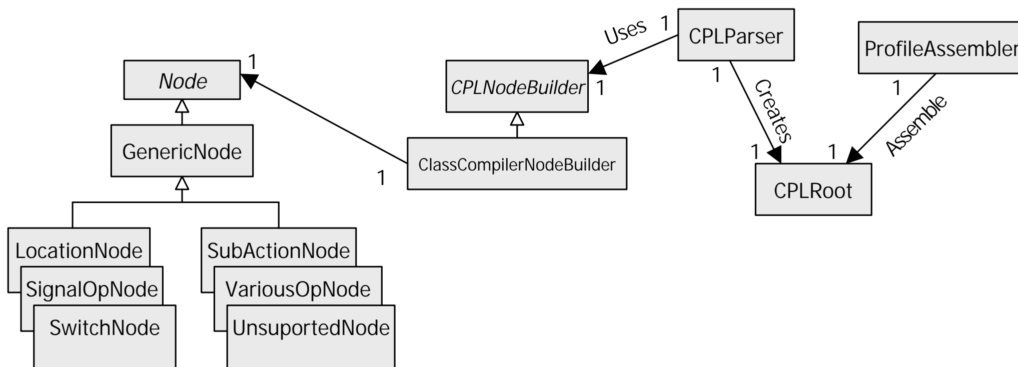

Figura 4.8. Diagramma della classi del compilatore CPL.

L'AST generato viene poi utilizzato dal **ProfileAssembler** per creare il file sorgente Java relativo al profilo, in particolare questa classe esplora ricorsivamente

[8]DTD, o Data Type Definition, è un documento in cui si definisce la struttura di un dialetto XML in cui vengono specificati tutti i vincoli sui tag e gli attributi, è possibile attraverso questo convalidare automaticamente un documento XML.

l'albero dei nodi generando per ogni nodo il relativo codice Java. In realtà ogni nodo si occupa di generare la propria porzione di codice e il **ProfileAssembler** si occupa di assemblare tutto, organizzare i dettagli della struttura del file generato, e salvare il sorgente ottenuto fondendo i dati raccolti dall'AST insieme ad un file modello che contiene lo scheletro base del profilo.

Durante questa fase è possibile trovare ancora altri errori dello script originale, che vengono notificati annullando la compilazione.

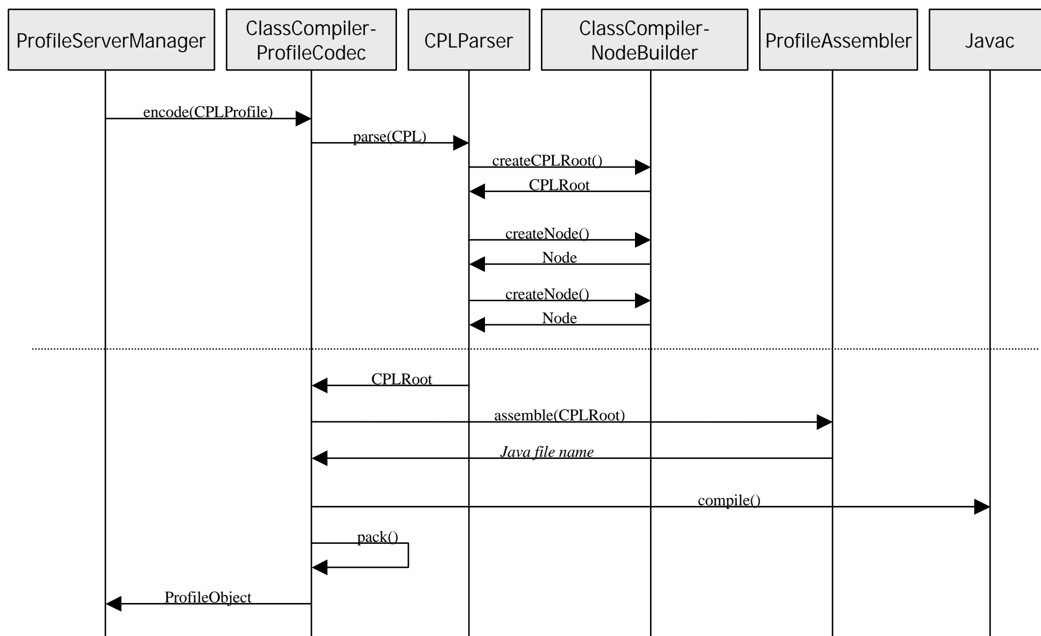

Figura 4.9. Diagramma della eventi del compilatore CPL.

Una volta creato il file sorgente questo viene trasformato in bytecode (file *.class*) da un compilatore Java esterno: questo viene chiamato non come processo esterno, ma si accede direttamente alla classe Java che lo implementa. Durante il funzionamento normale non devono trovarsi errori di compilazione, questo sarebbe sintomo di errore nella generazione del file Java e quindi di un errore nella traduzione da CPL a Java.

Successivamente alla compilazione, il bytecode viene letto e immagazzinato all'interno di un **ProfileObject** che risulta il prodotto finale della codifica.

Per quel che riguarda la fase di decodifica, che avviene all'interno del proxy (o più

precisamente all'interno del **ProfileManager**) il meccanismo è stato già analizzato approfonditamente nel paragrafo 4.3.1

4.4 Scenari d'uso

4.4.1 Creazione di un nuovo profilo

In questo caso d'uso l'unico componente che viene utilizzato è il **ProfileServer-Manager**. In particolare la creazione di un nuovo profilo avviene attraverso un'interfaccia web in cui l'amministratore può inserire lo script direttamente come testo o inviando il file XML presente sul computer locale (figura 4.10), è anche possibile generare il sorgente CPL graficamente attraverso un applet (figura 4.11). Quindi tutta l'operazione verrà effettuata attraverso un browser.

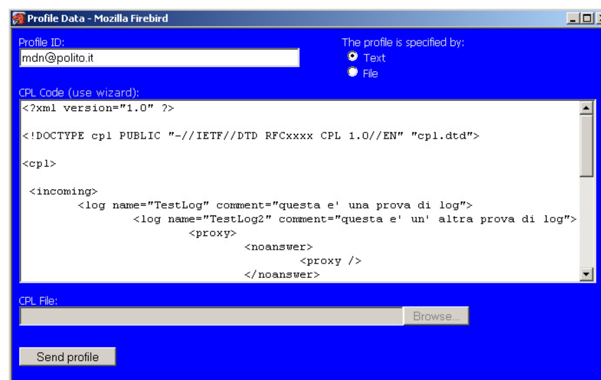

Figura 4.10. Creazione di un nuovo profilo.

Appena definito il profilo, il file CPL finale viene mandato al **ProfileServer-Manager** che si occuperà di attivare il *ProfileCodec* utilizzato, il quale interpreta e compila il file CPL (si veda il paragrafo 4.3.3 per un approfondimento sulle modalità di compilazione). Il *ProfileCodec* restituisce il **ProfileObject** che viene poi mandato al *ProfileRepository* che si occupa di memorizzarlo nel database con gli altri profili. Nel repository viene anche memorizzato il documento CPL che potrà poi essere utilizzato nel caso si voglia modificare successivamente questo profilo.

Se tutto il procedimento arriva senza alcun errore fino a questo punto, il **ProfileServerManager** si occupa di notificare l'inserimento di un nuovo profilo al

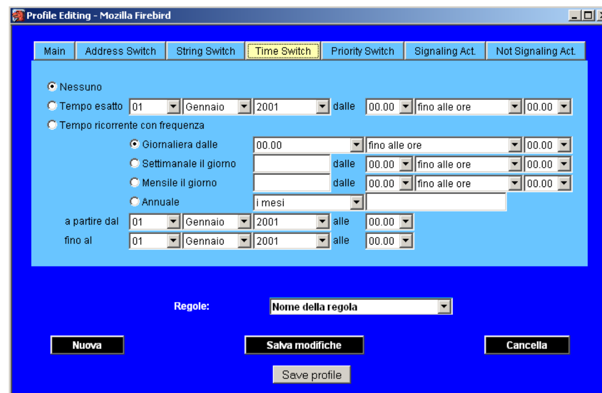

Figura 4.11. Creazione di un profilo tramite wizard.

ProfileManager, questo viene fatto attraverso una chiamata remota fatta tramite RMI (Remote Method Invocation) verso la cache del **ProfileManager** che esporta l'interfaccia *CacheController*. In questo modo se è già presente un altro profilo memorizzato nella cache per un dato utente, questo viene eliminato, forzando così la richiesta al *ProfileRepository* nel caso arrivi una chiamata per quel dato utente.

Questa operazione può sembrare inutile nel caso della creazione di un nuovo profilo, ma in realtà è indispensabile in quanto nella cache vengono inseriti dei profili segnaposto per gli utenti per i quali non hanno un profilo definito. Questo meccanismo serve per evitare inutili ripetute chiamate al *ProfileRepository* una volta che è stato appurato che non è presente un profilo per quel determinato utente.

4.4.2 Modifica di un profilo esistente

L'operazione di modifica di un profilo esistente è del tutto simile alla precedente, con la differenza che prima di venir fornita la pagina di modifica del profilo (figura 4.14) che è analoga a quella di creazione, viene caricato il documento CPL memorizzato nel repository durante la sua creazione. Successivamente le operazioni si ripetono come nel caso precedente.

4.4.3 Gestione di una prima chiamata

L'arrivo della prima chiamata coinvolge il **ProfileManager**. Tutto inizia dal **JAIN-Proxy**, che riceve una richiesta SIP di inizio chiamata e allora viene richiesto al

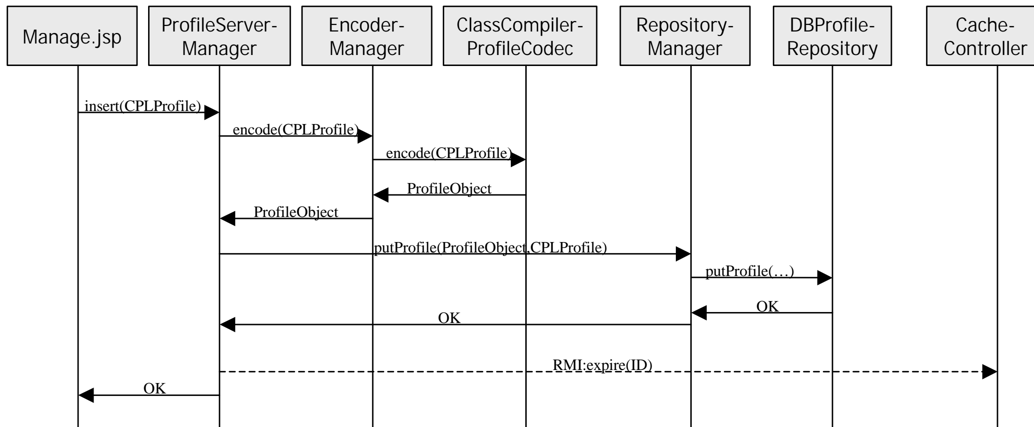

Figura 4.12. Diagramma degli eventi per la creazione di un nuovo profilo.

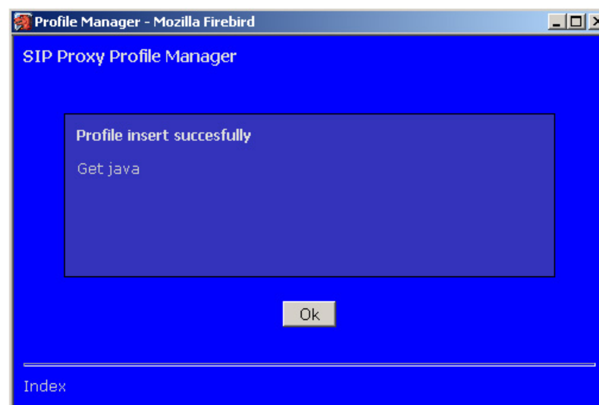

Figura 4.13. Messagio finale di creazione del profilo.

ProfileManager il profilo relativo all'utente chiamato.

Il **ProfileManager** controlla all'interno della propria cache se il questo è già presente. Essendo la prima chiamata, non viene trovato, allora la richiesta viene estesa al *ProfileRepository* configurato. Nel caso venga utilizzato il **DBProfileRepository** la ricerca viene eseguita all'interno di un database. Il *ProfileRepository* restituirà il **ProfileObject** contente il profilo. Questo viene quindi decodificato dal relativo *ProfileCodec* che fornisce l'oggetto che implementa l'interfaccia *Profile*, che, dopo essere memorizzato nella cache locale, a sua volta viene restituito indietro al **JAIN-Proxy** per elaborare la chiamata.

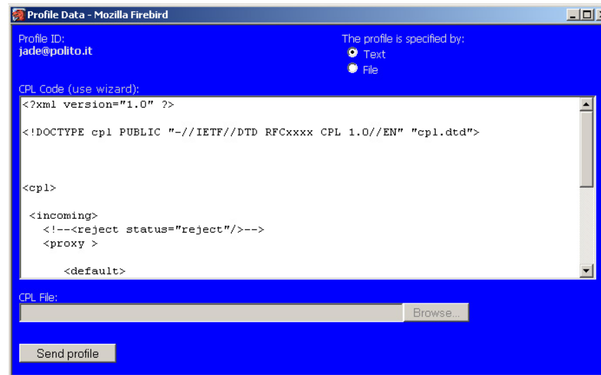

Figura 4.14. Modifica di un profilo esistente.

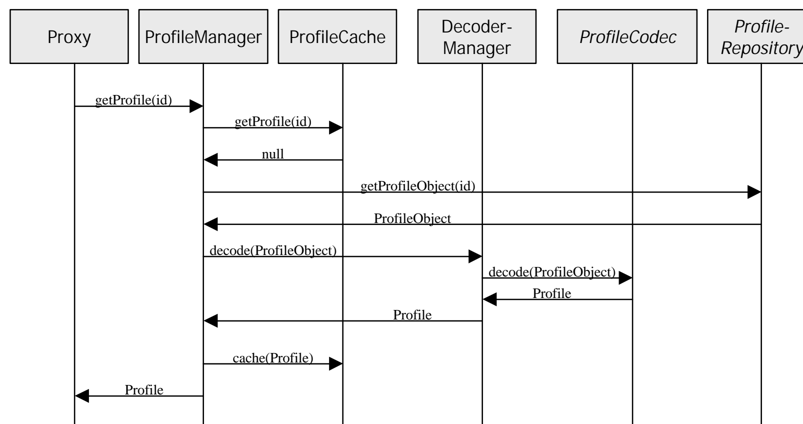

Figura 4.15. Diagramma degli eventi per la prima chiamata.

Nel caso in cui il profilo non sia presente all'interno del *ProfileRepository*, nella cache viene inserito un segnaposto per evitare che vengano fatte altre richieste per il profilo non esistente. In questo caso il proxy si limita ad eseguire le operazioni predefinite per la gestione delle chiamate.

4.4.4 Gestione delle chiamate successive

In questo scenario è stata già ricevuta una precedente chiamata per questo utente, quindi il relativo profilo è già stato richiesto al repository ed è già stato inserito nella cache locale. Quindi non viene effettuata la richiesta al repository remoto, infatti quando viene controllata la cache locale e viene trovato il profilo, questo viene

restituito subito al **JAIN-Proxy**, con il conseguente aumento delle prestazioni.

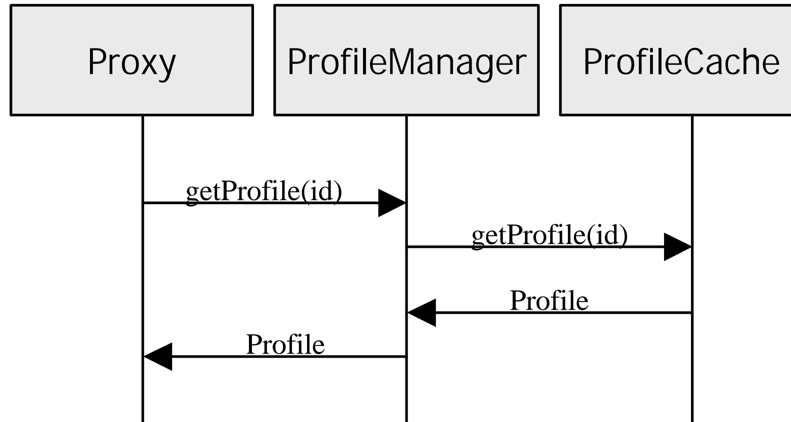

Figura 4.16. Diagramma degli eventi per le chiamate successive.

Nel caso in cui nel frattempo sia stato creato un nuovo profilo, il **ProfileServerManager** avrà già allora notificato alla cache, attraverso RMI, la presenza di una nuova versione, in questo caso la cache si è occupata di cancellare il profilo precedentemente memorizzato. In questo modo è come se si tornasse allo scenario precedente perché, non essendo presente nulla nella cache per questo utente, è come se fosse la prima chiamata che il **ProfileManager** riceve per questo.

Profile List - Mozilla Firebird

Profiles stored (11)

	Profile ID	Last modification	
1	test@id	19-lug-2003 16.26.20	edit - delete
2	testone@id	25-lug-2003 16.45.51	edit - delete
3	isa@polito.it	27-lug-2003 14.43.45	edit - delete
4	jade@polito.it	1-ago-2003 17.43.51	edit - delete
5	pippo@polito.it	28-lug-2003 15.08.20	edit - delete
6	test@polito.it	27-lug-2003 23.48.22	edit - delete
7	harpo@polito.it	28-lug-2003 15.02.51	edit - delete
8	addressTest@polito.it	28-lug-2003 14.45.52	edit - delete
9	complex@polito.it	27-lug-2003 18.12.43	edit - delete
10	proxyredir@polito.it	28-lug-2003 10.32.01	edit - delete
11	test1@polito.it	28-lug-2003 14.21.32	edit - delete

Index

Figura 4.17. Lista dei profili memorizzati nel repository.

Capitolo 5

Descrizione dell'implementazione

In questo capitolo verranno esaminati i dettagli implementativi relativi ai componenti realizzati e le motivazioni che hanno determinato le scelte fatte durante lo sviluppo del sistema.

5.1 Scelte di design

Durante la stesura del codice sono state prese in considerazione le linee guida presentate nel paragrafo 4.1.1, inoltre si è cercato di mantenere il più possibile consistente con la specifica il funzionamento dei vari elementi.

Ad esempio ogni componente che poteva avere diverse implementazioni a seconda del tipo di funzionalità che si desidera aggiungere (si pensi ad esempio alla gestione della cache), viene utilizzato attraverso un'interfaccia, e l'implementazione da utilizzare viene specificata attraverso un parametro di configurazione e instanziata attraverso il meccanismo di *reflection* del linguaggio Java.

Un altro esempio è il meccanismo di configurazione, per cui in tutti i componenti principali è presente un metodo

```
public boolean init(Properties props)
```

in cui i parametri vengono specificati attraverso un oggetto di tipo **Properties**, che contiene la lista dei parametri letti da un file di configurazione o definiti come variabili d'ambiente Java. In questo modo ogni componente interno (come i codec) può ricevere i suoi parametri facendo propagare lo stesso oggetto di proprietà. Se la configurazione è avvenuta con successo il metodo ritorna un valore *true*.

5.2 Struttura dei componenti

Per analizzare la struttura del sistema verrà seguita la struttura dei *package* Java, in quanto essi seguono le divisioni logiche degli elementi del sistema: per ognuno di questi verranno analizzate le classi contenute, la loro struttura e i relativi metodi più rilevanti.

5.2.1 Package *it.polito.cpl*

All'interno di questo package sono contenute le classi utilizzate da tutti i componenti (**ProfileManager** e **ProfileServerManager**) e le interfacce base su cui si appoggia il sistema.

Interfaccia *Profile*

Questa è l'interfaccia base che viene implementata da tutti i profili generati dal sistema: il proxy riceve un oggetto di questa interfaccia dal **ProfileManager** e gli passa il controllo della chiamata attraverso uno dei metodi esposti. In particolare sono presenti i metodi:

```
public boolean processInvite(RequestEvent requestEvent,
      Vector targetURIlist,ServerTransaction serverTransaction)
      throws ParseException,SipException;

public boolean processResponse(ResponseEvent responseEvent);

public boolean processTimeout(TimeoutEvent timeoutEvent);
```

che vengono chiamati rispettivamente quando arriva un messaggio per il profilo, comunemente un messaggio di tipo INVITE (si veda il paragrafo 2.3.2 sui comandi SIP), quando arriva una risposta ad una precedente chiamata, e quando il proxy genera un *timeout*[1] in quanto una risposta è tardata ad arrivare.

[1]Si definisce *timeout* lo scadere di un determinato intervallo di tempo, ad esempio il tempo predefinito di attesa di una risposta; in questo contesto la scadenza di un timeout genera un evento che indica che l'utente chiamato non risponde o che un terminale non è raggiungibile.

Il *ProfileCodec* utilizzato si occupa di generare un oggetto che sia in grado di gestire opportunamente le chiamate a questi metodi per soddisfare la definizione di profilo inserita attraverso il documento CPL.

Classe ProfileObject

E' il tipo di oggetto che viene utilizzato per memorizzare il profilo in maniera indipendente dal codec utilizzato, in particolare vengono salvati all'interno dei campi dell'oggetto:

- L'identificativo del profilo come stringa
- Il tempo di creazione del profilo (come *timestamp*[2])
- L'identificativo del codec con cui è stato codificato il profilo come stringa
- Il profilo codificato, all'interno di un oggetto di tipo *Serializable*[3]
- Possibili dati aggiuntivi all'interno di un oggetto di tipo *Serializable*

La presenza dei dati aggiuntivi può essere utile nel caso che un determinato *ProfileCodec* abbia la necessità di memorizzare delle informazioni utili per la decodifica del profilo. In particolare il **ClassCompilerProfileCodec** memorizza all'interno di questo campo il nome della classe che è stata creata, a cui non sarebbe possibile risalire direttamente dal nome del profilo.

Tutto l'oggetto è anch'esso *Serializable* in quanto può venir salvato dal *ProfileRepository* serializzando direttamente l'oggetto (in particolare il **DBProfileRepository** serializza tutto il **ProfileObject** all'interno di un campo di un database).

Classe CPLProfile

Questa classe contiene il documento CPL così com'è stato inserito pronto per essere elaborato dal *ProfileCodec*. In particolare il file CPL viene memorizzato all'interno di un vettore di *byte* come è stato ricevuto attraverso l'interfaccia web. Oltre

[2] Il *timestamp* è un indicazione di tempo che può essere interpretato come un timbro postale, in questo caso serve per marcare l'istante di creazione del profilo.

[3] Un oggetto Java di tipo che è *Serializable* può essere salvato automaticamente come file binario all'interno di qualunque tipo di flusso dati, come può essere un file su disco, una connessione di rete o un record di un database.

a questo vengono memorizzati l'identificativo del profilo e la data di inserimento (timestamp).

Interfaccia ***ProfileCodec***

E' l'interfaccia che devono implementare tutti i codec per poter esser utilizzati dal sistema. In particolare è stata studiata per poter separare nettamente il lato di codifica da quello di decodifica, questo perché i due meccanismi spesso sono completamente differenti l'uno dall'altro e hanno bisogno di operazioni specifiche e indipendenti.

All'interno del sistema infatti, l'uso dei due meccanismi è completamente separato, infatti l'istanza del codec utilizzata dal **ProfileManager** si occupa esclusivamente di decodificare i profili, mentre quella utilizzata dal **ProfileServerManager** viene usata per codificarli. Per altre considerazioni in proposito si veda la descrizione della struttura del **ClassCompilerProfileCodec**.

Nel caso il codec venga utilizzato per decodificare, i metodi che vengono utilizzati sono:

- `public boolean initDecoder(Properties props)`
  Configura il codec attraverso i parametri presenti all'interno dell'oggetto **Properties**, i parametri sono specifici dell'implementazione quindi cambiano a seconda del codec. Il metodo restituisce un valore *true* se l'inizializzazione ha avuto buon fine, *false* altrimenti.

- `public Profile decode(ProfileObject profileObject)`
  Decodifica il profilo, cioè trasforma l'oggetto **ProfileObject** in un profilo vero e proprio (si veda il paragrafo 4.3). Nel caso qualcosa non funzioni nel modo opportuno viene generata un'eccezione di tipo **CPLCodecException**.

Invece nel caso si utilizzino le funzionalità di codifica i metodi di riferimento sono:

- `public boolean initEncoder(Properties props)`
  Ha la funzionalità analoga al relativo metodo di decodifica, i parametri specificati in generale possono essere differenti tra le versioni.

- `public ProfileObject encode(String id, CPLProfile cpl)`
  Si occupa di codificare il profilo dal sorgente CPL in un **ProfileObject**. Anche

questo metodo genera un'eccezione di tipo **CPLCodecException** in caso di errore.

Come già detto, quello che accade nella varie fasi dipende a seconda del tipo di codec utilizzato.

Classe CPLCodecException

E' l'eccezione che viene generata nel caso qualcosa non funzioni nella fase di codifica o decodifica: viene utilizzato per notificare gli errori all'interno del sorgente CPL all'interfaccia web.

Interfaccia *ProfileRepository*

E' la generica interfaccia che deve essere implementata da un repository affinché possa essere utilizzato all'interno del sistema. Anche per questa è stata adottata un politica simile analoga a quella del *ProfileCodec*, cioé è possibile individuare anche per questo componente due diversi tipo di funzionamento: un funzionamento in sola lettura, cioé il semplice recupero dei profili memorizzati, utilizzato dal **ProfileManager**, e in lettura/scrittura, in cui è possibile inserire, cancellare, ottenere la lista dei profili, tipo di funzionamento utilizzato dal **ProfileServerManager**.

E' possibile così costruire dei repository che implementino soltanto la parte di sola lettura ed utilizzarla soltanto all'interno del **ProfileManager**, si provi ad immaginare, ad esempio, un *ProfileRepository* che scarichi i profili attraverso il protocollo HTTP da un server remoto, in questo modo è possibile attraversare con facilità dei firewall che separano le sezioni della rete in un cui sono in esecuzione i due componenti.

Queste due tipi di repository vengono definiti: *read-only repository* nel caso sia possibile soltanto recuperare i profili o *storing repository* nel caso siano presenti anche le funzionalità di inserimento.

Per verificare che tipo di *ProfileRepository* si sta utilizzando esiste il metodo

```
public boolean isReadOnly();
```

che restituisce un valore vero nel caso l'implementazione si di tipo *read-only*, falso nel caso sia uno *storing repository*.

I repository di tipo *read-only* devono implementare il metodo:

```
public ProfileObject getProfileObject(String id)
```

che restituisce il **ProfileObject** del profilo con l'identificativo specificato, se questo non è presente ritorna un valore *null*. Nel caso venga generato qualche errore viene lanciata un'eccezione di tipo **RepositoryException**.

Se invece l'implementazione appartiene ad uno *storing repository* devono venire implementati anche i seguenti metodi:

- `public boolean putProfile(ProfileObject profile,CPLProfile cpl)`
  Si occupa di inserire un nuovo profilo (ed il relativo codice sorgente CPL) all'interno del repository.

- `public CPLProfile getCPL(String id)`
  Viene utilizzato per recuperare il codice CPL di un relativo profilo, viene utilizzato nella fase di modifica.

- `public boolean deleteProfile(String id)`
  Si occupa di cancellare un profilo dal repository.

- `ProfileInfo [] getStoredInfoProfiles()`
  Restituisce la lista dei profili memorizzati all'interno del repository sotto forma di vettore di **ProfileInfo**.

Per impostare i parametri di inizializzazione del repository è presente il metodo:

```
public boolean init(Properties props)
```

Che riceve come argomento la lista dei parametri, come già visto per il *ProfileCodec*, i nomi e i valori di questi parametri sono strettamente dipendenti dall'implementazione utilizzata.

Classe RepositoryException

E' l'eccezione che viene generata in tutte le operazioni di accesso al repository nel caso qualcosa non abbia funzionato correttamente.

Classe ProfileInfo

Questa è una classe che viene utilizzata per ritornare le informazioni riguardo ai profili memorizzati, in particolare contiene solo due proprietà che permettono di memorizzare l'identificativo del profile memorizzato e il suo timestamp.

Classe DBProfileRepository

E' l'implementazione dell'interfaccia *ProfileRepository* che permette di memorizzare i profili all'interno di un database attraverso l'architettura JDBC su qualunque tipo di database che abbia un driver compatibile. Questa classe funziona come *storing repository* infatti permette sia di memorizzare, sia di recuperare i profili memorizzati.

I metodi presenti sono quelli definiti dall'interfaccia *ProfileRepository*, in più è presente un membro privato:

```
private boolean testDB()
```

che viene chiamato durante la fase di inizializzane del repository che controlla la connessione al database eseguendo un'inserimento e un cancellazione, permettendo in questo modo di controllare anche i permessi che il processo possiede verso il database e quindi individuare e notificare tutti i possibili errori già in fase di avvio del sistema e non durante il funzionamento normale.

I dati sulla connessione al database vengono specificati attraverso quattro parametri: il nome della classe del driver JDBC, la stringa di connessione (che è specifica del driver), il nome utente e la password per l'accesso al database.

Interfaccia *CacheController*

Questa interfaccia viene utilizzata per esportare le funzioni di controllo della cache attraverso il RMI, quindi questi metodi vengono implementati dalla cache del **ProfileManager** (classe **ProfileCache**). In particolare i metodi presenti sono:

- `public boolean expireProfile(String id)`
  Notifica alla cache di far espirare il profilo selezionato se questo è presente, restituisce il valore vero se è presente, falso altrimenti.

- `public void expireAll()`
  Richiede alla cache di far espirare tutti i profili presenti. Questo metodo allo

stato attuale non è mai utilizzato dal sistema ed è stato inserito per possibili usi futuri.

5.2.2 Package *it.polito.cpl.client*

All'interno di questo package sono presenti tutte le classi utilizzate esclusivamente dal **ProfileManager**.

Classe ProfileManager

E' la classe principale di questo package, e viene utilizzata direttamente dal proxy per richiedere i profili all'arrivo delle chiamate. Si occupa di gestire tutti le parti del componente **ProfileManager** come la cache, il repository e il gestore dei codec.

Possiede due metodi di inizializzazione:

```
public boolean init(String confFile)
public boolean init(Properties props)
```

Il primo può venir chiamato specificando il nome di un file di configurazione da cui leggere i parametri di funzionamento e impostazioni di configurazione, dopo aver letto il file viene chiamato il secondo metodo che si occupa di eseguire le inizializzazioni vere e proprie. Oltre alle proprietà lette da file vengono anche aggiunte quelle presenti tra le variabili d'ambiente della JVM.

Questo metodo, nell'ordine, crea l'istanza del *ProfileRepository* specificato nel file di configurazione ed esegue la sua inizializzazione, attiva e configura la cache e il **DecoderManager** chiamando i metodi *init(...)* dei vari elementi. Oltre a questo attiva il meccanismo di traduzione degli indirizzi (si veda più avanti).

L'inizializzazione viene considerata eseguita con successo se tutti questi passi avvengono senza errori, in caso contrario viene restituito il valore falso.

Il metodo principale del **ProfileManager** è:

```
public Profile getProfile(String id)
```

che viene chiamato dal proxy per ottenere un determinato profilo, questo si occupa di chiedere alla cache l'oggetto profilo relativo ad un determinato utente o di inoltrare la richiesta al **ProfileRepository** e poi farla decodificare dal codec opportuno.

Nell'esecuzione di queste operazioni viene utilizzato l'identificativo dell'utente (il suo indirizzo SIP) sotto forma di stringa; per poter funzionare correttamente questo indirizzo viene innanzitutto normalizzato, eliminando la parte di protocollo e poi, questo viene utilizzato per cercare all'interno della tabella delle registrazioni del proxy (classe **examples.proxy.registrar.RegistrationTable**) tutti gli alias con cui questo utente viene identificato. Queste informazioni vengono memorizzate in una *Hashtable* in modo tale da associare tutti gli indirizzi di uno stesso utente allo stesso profilo.

Le operazioni di gestione degli alias vengono eseguite nei metodi:

- `private void checkAliases(String id)`
  Controlla le registrazioni all'interno del proxy e le memorizza nella *Hashtable*.

- `private String getAlias(String id)`
  Cerca all'interno dell'*Hashtable* la presenza di alias.

- `public void addAlias(String id,String alias)`
  Aggiunge un'alias nella *Hashtable*.

Un'altra funzionalità del **ProfileManager** aggiunta successivamente è il meccanismo di traduzione degli indirizzi, cioé è possibile specificare un pattern di indirizzi (attraverso un'espressione regolare[4]) che possono venir poi tradotti in un'altro, o in un'altra serie di indirizzi.

E' possibile cioé utilizzare parti dell'indirizzo di partenza per costruire l'indirizzo destinazione: per una spiegazione più approfondita sul meccanismo di funzionamento si veda l'appendice A.

Classe DecoderManager

E' il gestore dei codec utilizzato dal **ProfileManager**, è stato costruito per dar la possibilità di utilizzare più codec contemporaneamente per sviluppi futuri, anche nell'attuale implemetazione gestisce un'unico codec. Questa classe ha tre metodi:

[4]E' un linguaggio progettato appositamente per descrivere pattern di testo, diventato famoso attraverso il linguaggio PERL e ormai presente in tutti i linguaggi più recenti. Per utilizzarlo all'interno del sistema è stata utilizzata un libreria distribuita (con licenza GPL) dal progetto Jakarta di Apache Foundation [23].

- `public boolean init(Properties props)`
  Viene chiamato dal metodo omonimo del **ProfileManager** e si occupa di creare (attraverso i meccanismi di *reflection* Java) e inizializzare il codec vero e proprio, il cui nome della classe viene specificato attraverso un parametro di configurazione.

- `public Profile decode(ProfileObject profileObject)`
  Richiede al codec di decodificare il **ProfileObject** richiesto, in pratica funziona da tramite verso il codec vero e proprio. Nel caso sussista un errore nella decodifica, e quindi venga generata un'eccezione di tipo **CPLCodecException** da parte del *ProfileCodec* vero e proprio, questa viene rilanciata al metodo chiamante.

- `public String registerCodec(String codecClass)`
  Attualmente non è implementato (restituisce *null* in ogni caso), verrà utilizzato per poter registrare altri codec in fase esecuzione.

Classe ProfileCache

E' la classe che gestisce la cache del **ProfileManager**, viene configurata attraverso il metodo

```
public boolean init(Properties props)
```

Con le stesse modalità delle classi già viste in precedenza. I profili sono memorizzati all'interno di una *Hashtable* utilizzando come campo indice l'identificativo del profilo. Per inserire un profilo viene utilizzato il metodo:

```
public void cache(Profile profile)
```

In realtà non viene memorizzato direttamente il profilo all'interno della cache, ma questo viene contenuto all'interno di oggetto di *CacheEntry*, questa specie di contenitore viene utilizzato per poter tener traccia dell'utilizzo che viene fatto del profilo inserito nella cache: in questo modo, è possibile utilizzare politiche complesse per la gestione di alcuni aspetti della cache, come ad esempio la pulizia.

L'oggetto *CacheEntry* viene creato da un *CacheHandler* che si occupa anche di gestire la pulizia vera e propria. Questo meccanismo viene descritto in maniera più approfondita successivamente.

Quando arriva una chiamata per un utente a cui non è stato creato alcun profilo (cioé all'interno del repository non è presente alcun profilo associato a quel identificativo), il proxy tratta la chiamata con la sua politica predefinita, in ogni caso viene fatta ogni volta una richiesta al *ProfileRepository*, se arrivassero molte chiamate di questo tipo si avrebbero un gran numero di richieste inutili al repository, con di conseguenza una riduzione delle prestazioni. Per evitare questo tipo di inconveniente è stato aggiunta la presenza di quelli che vengono chiamati **DummyProfile**[5]. Quando un profilo non viene trovato all'interno del repository, nella cache viene comunque inserito un segnaposto vuoto per quel utente, in questo modo ad ogni chiamata successiva, la cache restituisce il profilo fasullo al **ProfileManager** che a questo punto non inoltra la richiesta al *ProfileRepository*, ma notifica direttamente al proxy la mancanza del oggetto *Profile* per quel utente. Il metodo che viene utilizzato per inserire un **DummyProfile** è:

```
public void cacheDummy(String id)
```

Il **ProfileManager** richiede i profili inseriti nella cache attraverso il metodo:

```
public Profile getProfile(String id)
```

che si occupa di cercare all'interno della *Hashtable* la relativa *CacheEntry*, e poi estrae da questa il profilo e lo restituisce al chiamante.

La classe **ProfileCache** implementa inoltre l'interfaccia *CacheController* che viene utilizzata per esportare attraverso RMI i metodi per eliminare degli elementi dall'interno della cache (per la lista di questi metodi si veda il paragrafo relativo).

Interfaccia *CacheHandler*

Questa interfaccia viene utilizzata per permettere l'utilizzo di politiche personalizzate nella gestione degli elementi nella cache. Il **ProfileCache** si appoggia ad una implementazione di questa interfaccia per creare le *CacheEntry* da inserire nella cache. I metodi presenti sono:

- `public CacheEntry newEntry(Profile profile)`
  Viene utilizzato per istanziare gli oggetti che implementano *CacheEntry* (pattern *Factory*), in questo modo la cache non si deve preoccupare dell'implementazione attuale utilizzata.

[5]Letteralmente profili fasulli.

- `public void cleanCache(Hashtable cache,int numElemsToRemove)`
  Esegue la pulizia della cache secondo la politica definita dall'implementazione.

Interfaccia *CacheEntry*

E' la generica interfaccia del contenitore usato per memorizzare i profili nella cache, possiede due metodi:

- `public void newRequest()`
  Viene chiamato ogni volta che viene richiesto questo profilo dalla cache all'interno del metodo *getProfile(...)* della **ProfileCache**, viene utilizzato, ad esempio per tenere il conto di quante volte questo profilo è stato utilizzato, in realtà il suo utilizzo dipende dall'implementazione, o ancora meglio dalle politiche che si vogliono seguire per la gestione dei profili nella cache.

- `public Profile getProfile()`
  Restituisce l'oggetto profilo che contiene all'interno della *CacheEntry*.

Classe DefaultCacheHandler

E' l'implementazione predefinita dell'interfaccia *CacheHandler*; ogni volta che viene richiesta la pulizia della cache vengono cancellati gli elementi in base alla data di ultimo utilizzo, quindi vengono scelti i più vecchi.

5.2.3 Package *it.polito.cpl.server*

Questo package contiene le classi base utilizzate dal **ProfileServerManager**.

Classe ProfileServerManager

La classe principale del componente server, viene utilizzato dalle pagine JSP per le operazioni di gestione dei profili. I suoi metodi principali rispecchiano queste finalità:

- `public boolean insertProfile(CPLProfile cplProfile)`
  Inserisce un nuovo profilo all'interno del repository a partire dal suo **CPLProfile**, cioè dal documento CPL. Questo metodo si occupa di tutti i passi

necessari per la creazione e la memorizzazione del profilo, in particolare si appoggia al **EncoderManager** per codificare il profilo, e usando il **RepositoryManager** lo memorizza all'interno del repository.

- `public boolean removeProfile(String id)`
  Rimuove dal repository un profilo precedentemente memorizzato.

- `public CPLProfile getCPL(String id)`
  Ottiene dal repository il sorgente CPL (sotto forma di **CPLProfile**) di un profilo precedentemente inserito, questo viene utilizzato nella fase di modifica del profilo in cui è utile avere a disposizione il codice originale.

Sono presenti altri metodi che vengono utilizzati per accedere alle informazioni sullo stato della creazione dei profili dalle pagine JSP, in particolare:

- `public String getLastError()`
  Restituisce l'ultimo errore incontrato nella creazione di un profilo, questo viene visualizzato sul browser per mostrare la descrizione del problema nel caso l'operazione non sia andata a buon fine.

- `public String getLastProfileName()`
  Restituisce il nome dell'ultimo sorgente Java creato nella generazione di un profilo, è utile in fase di test, ma non è necessario durante il funzionamento normale.

E' possibile accedere direttamente al repository utilizzato dal **ProfileServerManager** attraverso il metodo:

```
public ProfileRepository getCurrentProfileRepository()
```

Per quanto riguarda l'inizializzazione del componente sono presenti due metodi:

```
public boolean init(String confFile)
public boolean init(Properties props)
```

che funzionano in maniera identica rispetto a quelli del **ProfileManager**. In questo caso chi si occupa di creare ed eseguire l'inizializzazione sono le pagine JSP: al primo accesso viene creato un **ProfileServerManager** che viene mantenuto tra tutte le pagine dell'applicazione web attraverso il *ServletContext*.

Un'ultima nota si può fare per il metodo:

```
private void notifyViaRMI(String id)
```

Che si occupa di notificare, attraverso il meccanismo RMI, alla cache del **ProfileManager**, che un determinato profilo è stato modificato e quindi di eliminare la presenza della vecchia versione, se questa è presente, dalla cache.

Ogni volta che viene chiamato questo metodo viene rieseguita la connessione, in questo modo, benche si abbia una leggere diminuzione delle prestazioni, si evitano tutti i problemi legati a momenti di temporanea irraggiungibilità del **ProfileManager**.

Classe EncoderManager

Si occupa di gestire l'interazione con il codec, attualemente si occupa di inizializzare il codec in base ai parametri di configurazione e di passargli la richiesta di compilazione dei profili.

E' stata creata per permettere facilmente l'estensione del sistema nell'uso dei codec, come ad esempio la gestione di codec multipli.

Classe RepositoryManager

Funziona in maniera analoga alla classe precedente, si occupa semplicemente di fare da tramite e gestire il *RepositoryManager* vero e proprio.

5.2.4 Package *it.polito.cpl.parser*

In questo package sono contenute tutte le classi che costituiscono lo scheletro del parser CPL. E' stato creato un package a parte invece che inserirlo all'interno della gerarchia *it.polito.cpl.classcompiler* perché è possibile utilizzare queste classi per costruire differenti codec. Lo scopo di queste classi non è di creare il profilo oggetto, ma è di costruire l'AST relativo al sorgente CPL, questo AST viene poi trattato in maniera differente a seconda del codec che lo deve utilizzare.

Classe CPLParser

E' la classe principale che si occupa dell'interpretazione del documento CPL, questo può venire specificato sia come percorso di file, sia come *InputStream* a seconda del costruttore usato.

```
public CPLParser(String cplFileName)
public CPLParser(InputStream cplInputStream)
```

Per analizzare il file XML il parser si appoggia alle API JAXP (Java API for XML Processing) utilizzando Xerces (fornito da Apache Foundation) come implementazione, anche se è possibile cambiare implementazione senza dover modificare il codice in alcun modo.

Un vantaggio di utilizzare questo tipo di libreria è la possibilità di effettuare la validazione del documento XML, e quindi di poter fare un controllo sulla sintassi del sorgente CPL in modo automatico. Per ottenere questo viene inizializzato il *DocumentBuilder* nel metodo

```
private DocumentBuilder initDocumentBuilder()
```

dove viene configurato per notificare gli errori di validazione attraverso la classe interna **ParserChecker**. Se questa operazione non viene eseguita correttamente viene generata un'eccezione di tipo *ParserConfigurationException*.

Questa classe non costruisce direttamente i nodi dell'AST, ma questi vengono generati da un *CPLNodeBuilder* che cambia a seconda dell'utilizzo dei nodi che si vuol fare, ad esempio il **ClassCompilerProfileCodec** ha un suo *CPLNodeBuilder* (**ClassCompilerNodeBuilder**) che genera i nodi che formeranno l'albero sintattico secondo le sue esigenze. E' necessario impostare il *CPLNodeBuilder* da utilizzare prima di far partire l'analisi del documento attraverso il metodo:

```
public void setNodeBuilder(CPLNodeBuilder cplNodeFactory)
```

Il metodo che si occupa effettivamente di costruire l'AST è:

```
public CPL parse()
```

che prima di tutto effettua la validazione del documento CPL fornito, poi inizia l'analisi vera e proprio creando il nodo radice (di tipo *CPL*) e delegandogli il resto dell'elaborazione: infatti l'esplorazione dell'albero XML e la costruzione dell'AST avvengono in maniera completamente ricorsiva. Una volta costruito l'AST viene eseguito un ultimo controllo di consistenza attraverso il metodo:

```
private void consistencyCheck(CPL cpl)
```

che verifica l'assenza di costrutti ciclici nel programma CPL (si veda il paragrafo 3.2).

Interfaccia ***CPLNodeBuilder***

Definisce le funzionalità delle classi che costruiscono i nodi da utilizzare nella formazione dell'AST (pattern *Factory*), i metodi forniti sono:

- `public Node createNode(Element nodeElement)`
  Crea un oggetto di tipo *Node* a partire dall'elemento XML a cui deve corrispondere.

- `public CPL createCPL()`
  Istanzia l'implementazione della radice dell'AST su cui verranno inseriti tutti i nodi.

Interfaccia ***CPL***

Costituisce la radice dell'AST di un programma CPL, l'implementazione viene fornita da un particolare *CPLNodeBuilder*. E' formata da quattro metodi importanti:

- `void setIncomingNode(Node incomingNode)`
  Imposta il nodo iniziale dell'AST.

- `Node getIncomingNode()`
  Restituisce il nodo iniziale dell'AST.

- `void addSubAction(SubAction subAction)`
  Aggiunge un nuovo *subaction* alla lista.

- `public SubAction getSubActionNode(String ref)`
  Restituisce una subaction in base al suo nome tra quelle memorizzate.

Interfaccia ***Node***

E' il prototipo di tutti i nodi dell'AST, contiene tutta una serie di costanti utilizzate per la creazione dell'albero e una serie di metodi come:

- `public String getTypeName()`
  Restituisce il tipo di nodo sotto forma di stringa; per tipo di nodo si intende il nome del nodo CPL che rappresenta (si veda il paragrafo 3.2).

- `public int getType()`
  Restituisce il tipo di nodo sotto forma di costante numerica intera.

- `public boolean hasOutputs()`
  Ritorna *true* se questo nodo possiede degli output.

- `void setNextNode(Node node)`
  Imposta il nodo successivo da eleborare nello script CPL.

- `public Node getNextNode()`
  Restituisce il nodo successivo nell'esecuzione dello script.

- `void checkLoops(CPL cpl,Hashtable subactions)`
  Viene chiamato in modo ricorsivo per controllare se sono presenti dei cicli all'interno della struttura dello script.

Classe Output

Gestisce gli output di un determinato nodo, memorizza inoltre i possibili attributi presenti, in modo che possano venire utilizzati dai nodi nel momento in cui l'AST viene esplorato per costruire il profilo effettivo.

Interfaccia *SubAction*

Estende l'interfaccia *Node* per definire i nodi contenenti delle subaction. In particolare viene aggiunto il metodo:

```
public String getName()
```

che restituisce il nome di riferimento dell'azione.

Classe SubNodePlaceHolder

E' una classe di comodo che viene utilizzata all'interno dell'AST come riferimento ad una subaction.

Classe CPLParsingException

Viene generata quando ci sono degli errori nella costruzione del AST del profilo CPL.

5.2.5 Package *it.polito.cpl.classcompiler*

Questo package contiene le classi che vengono utilizzate dal **ClassCompilerProfileCodec**. All'interno di questo è presente un'altro package che contiene le classi usate dal compilatore CPL, che è analizzato nella sezione successiva.

Classe ClassCompilerProfileCodec

E' la classe che implementa il codec attualmente utilizzato nel sistema, il suo funzionamento è descritto nella sezione 4.3.3. Questa classe implementa le funzioni dichiarate nell'interfaccia *ProfileCodec*, in particolare per quel che riguarda i meccanismi di inizializzazione sono presenti i due metodi:

```
public boolean initDecoder(Properties props)
public boolean initEncoder(Properties props)
```

il primo non esegue alcuna operazione in quanto il meccanismo di decodifica non necessita di parametri particolari, il secondo invece si occupa di configurare tutti i percorsi necessari e il compilatore Java utilizzato per creare i file oggetto relativi ai profili.

Il compilatore Java viene utilizzato direttamente come classe Java e non lanciato dalla linea di comando, in questo modo è possibile migliorare le prestazioni di compilazione. In ogni caso, nel codice, non si fa mai accesso alle sue classi direttamente, ma si accede soltanto attraverso meccanismi di reflection attraverso il metodo

```
private boolean compile(String sourceName)
```

E' stata presa questa scelta per non dover essere legati ad un determinato compilatore nella fase di funzionamento, ed inoltre per non rendere necessario il dover disporre del compilatore quando il codec viene utilizzato solo come decodificatore (nel **ProfileManager**).

Tutta la fase di codifica avviene all'interno del metodo:

```
public ProfileObject encode(String id, CPLProfile cpl)
```

in cui avvengono tutti i passi descritti nel paragrafo 4.3.3.

Per quello che riguarda la codifica viene utilizzato il metodo

```
public Profile decode(ProfileObject profileObject)
```

In cui viene caricata la classe presente nel **ProfileObject** all'interno del **CodecClassLoader** presente nel codec, e successivamente istanziato il profilo vero e proprio.

Classe CodecClassLoader

E' il classloader modificato a cui è stato aggiunto il metodo:

```
public Class loadClass(ProfileObject po)
```

che permette di caricare una classe non in base al nome, ma in base ad un **ProfileObject**.

L'unico metodo che è stato ridefinito dalla classe originale è:

```
public Class findClass(String name)
```

che si occupa di caricare il bytecode della classe sotto forma di vettore di *byte*, che legge questi dati all'interno di una tabella in cui precedentemente il metodo *loadClass(...)* ha inserito il codice che era memorizzato all'interno del **ProfileObject**.

Tutte le altre classi vengono caricate dal classloader di base del sistema.

5.2.6 Package *it.polito.cpl.classcompiler.parsercompiler*

Questo package contiene tutte le classe utilizzate dal codec per eseguire l'analisi del documento cpl e la costruzione dell'AST sfruttando i componenti del package *it.polito.cpl.parser*.

Classe ClassCompilerNodeBuilder

E' l'implementazione dell'interfaccia *CPLNodeBuilder* che si occupa di istanziare i vari nodi le cui implementazioni sono costuite dalle classi:

- **SwitchNode**
- **LocationNode**
- **SignalOpNode**
- **SubActionNode**

- **VariousOpNode**
- **UnsupportedNode**

presenti all'interno di questo package; ogni classe costituisce un tipo di nodo CPL e si occupa di generare il codice Java relativo.

La radice dell'AST è costituita dalla classe **CPLRoot** che implementa l'interfaccia *CPL*. I metodi presenti sono quelli già visti per l'interfaccia *CPLNodeBuilder*.

Classe GenericNode

E' una classe astratta che racchiude tutte le funzionalità comuni che tutti i nodi usano, implementa l'interfaccia *Node* e viene estesa da tutti le classi relative ai nodi viste nel paragrafo precedente.

Oltre all'implementazione dei metodi presenti nell'interfaccia *Node*, possiede una serie di metodi di utilità che vengono utilizzati nella fase di analisi del documento CPL:

```
boolean getBooleanAttribute(Element el,String attrName)
float getFloatAttribute(Element el,String attrName)
int getIntAttribute(Element el,String attrName)
```

che vengono utilizzati per facilitare l'estrazione delle informazioni numeriche (e booleane) dagli attributi del documento XML che costituisce lo script CPL.

Oltre a queste sono presenti altre due funzioni che vengono utilizzate per la fase di assemblamento del sorgente Java che costituisce il profilo. La prima è:

```
abstract public void assemble(ProfileAssembler assembler,
             StringBuffer currentBuffer, int depth)
```

che viene implementata in ogni nodo che si occupa di inserire all'interno dello *StringBuffer* il codice Java specifico del nodo, questo metodo viene chiamato recursivamente esplorando tutto l'albero e quindi facendo in modo che ogni nodo inserisca la sua parte nel posto giusto.

In realtà all'interno di ogni implementazione del metodo *assemble(...)* non viene richiamato lo stesso metodo per la recursione, ma viene richiamato:

```
void recurse(Node next,ProfileAssembler asm,StringBuffer buf, int dep)
```

che si occupa di chiamare il metodo *assemble(...)* del metodo successivo, o se il metodo successivo è di tipo *sub* viene aggiunto il codice relativo alla chiamata della relativa funzione (legata alla corrispondente *subaction*).

Classe ProfileAssembler

E' la classe che si occupa di assemblare il profilo Java a partire dall'AST, per fare questo si basa su un file Java che fa da modello, in cui è presente lo scheletro del profilo che viene poi riempito nel modo opportuno con tutto il codice relativo al profilo.

Per fare questo sono state definite alcune sezioni da riempire all'interno del modello a cui corrispondono altrettanti buffer all'interno di questa classe. Ogni sezione rappresenta una parte del sorgente in cui è possibile aggiungere un tipo di codice, ad esempio esiste una sezione per gli *import* in cui ogni nodo deve aggiungere le istruzioni di *import* per le classi che utilizza, o una sezione per le inizializzazioni (che all'interno del modello si trova nel costruttore della classe) in cui ogni nodo inserisce le inizializzazioni di cui ha bisogno.

Quando ogni nodo ha inserito all'interno dei buffer il proprio codice, questi vengono fusi insieme al modello ed in questo modo viene generato il sorgente Java effettivo.

L'assemblamento viene eseguito nel metodo:

```
public boolean assemble()
```

in cui viene esplorato l'AST attraverso il metodo *assemble(...)* visto nel paragrafo precedente.

Una volta terminato il processo di generazione del codice nei buffer, avviene la fase di fusione con il modello nel metodo:

```
boolean createJavaSource()
```

in cui viene creato il file Java all'interno di una directory temporanea.

Per inserire i frammenti di codice all'interno dei vari buffer sono presenti tutta una serie di funzioni:

```
public void addImport(Object packageName)
public void addDefinition(Object snippet)
```

```
public void addDefinition(Object type,Object propName)
public void addInitialization(Object snippet)
public void addInviteInit(Object snippet)
public void addResponseFunction(Function func)
public void addTimeoutFunction(Function func)
public void addFunction(Function func)
```

Ognuna di questa rappresenta una sezione di codice diversa, e il codice, inserito come stringa, viene memorizzato in una struttura come un *Vector* o una *Hashtable* e poi successivamente inserito nel corpo del file Java.

Classe Function

Per la definizione delle funzioni utilizzate nella fase di assemblamento del profilo Java è presente una classe apposta in cui è possibile memorizzare tutte le informazioni sulla funzione.

In particolare per la gestione delle risposte SIP con il profilo è necessario utilizzare un'architettura asincrona, in quanto non c'è alcuna garanzia sul tempo d'attesa, le considerazioni sono analoghe anche per le mancanze delle risposte (timeout); quindi la gestione degli output dei nodi di segnalazione (*proxy*), vengono gestite attraverso delle funzioni che vengono chiamate in base all'ultimo nodo di questo tipo che è stato elaborato.

Questo meccanismo funziona attraverso un valore numerico intero (status) che è differente per ogni elemento *proxy* all'interno del profilo, quindi quando viene eseguita la chiamata viene memorizzato questo valore all'interno del profilo, nel momento in cui arriva la risposta viene chiamata la funzione associata a quel determinato valore. In questo modo è possibile avere più di un'operazione *proxy* all'interno di uno stesso script.

Quindi oltre al corpo vero e proprio della funzione viene memorizzato, all'interno di questa classe, anche lo stato a cui è associato, e nella fase di composizione del sorgente Java, queste informazioni vengono utilizzare per avere dei costrutti ad hoc per quel determinato tipo di profilo.

5.2.7 Package *it.polito.cpl.util*

In questo package sono presenti due classi che vengono utilizzate all'interno del profilo generato per eseguire delle operazioni ripetitive che è inutile replicare in modo del tutto identico all'interno di ogni profilo generato.

Classe TimeSwitchElement

Questa classe viene utilizzata per gestire un output di tipo *time* in quanto i tipi di intervalli di tempo possono essere definiti in maniera estremamemte complessa e risulterebbe molto scomodo generare ogni volta il codice personalizzato.

Il suo utilizzo è molto semplice: è necessario impostare tutti gli attributi presenti all'interno del tag attraverso il metodo

```
public boolean set(String fieldName,String val)
```

che si occupa di memorizzare tutti i dati che verranno poi utilizzati durante la fase di controllo. Succesivamente attraverso il metodo:

```
public boolean init()
```

viene controllata la consistenza delle informazioni inserite e precalcolati alcuni parametri interni.

Attraverso il metodo:

```
public boolean checkTime(Calendar time)
```

viene controllato se l'indicazione di tempo contenuta nell'oggetto *Calendar* si trova all'interno degli intervalli definiti restituendo un valore vero in questo caso, falso altrimenti.

Classe Utils

Questa classe contiene funzioni statiche utilizzate principalmente dalle classi relative ai profili, in particolare sono presenti i metodi:

```
public static boolean findString(Iterator iter, String str,
                                                         boolean equals)
```

Si occupa di cercare una stringa all'interno di un oggetto di tipo *Iterator*; viene usato nel codice generato da un nodo di tipo *string-switch*.

```
public static boolean addressCheck(HeaderAddress hAddress,
               int type, String secondValue, int checkType)
```

Viene utilizzato per verificare le condizioni sugli indirizzi all'interno degli *address-switch*.

```
static public void proxyDefaultAction(Proxy proxy,
          RequestEvent requestEvent, Vector targetURIList)
```

Effettua le operazioni di chiamata predefinite del proxy.

```
static public void unsuccesfullResponse(Proxy proxy,
                           RequestEvent requestEvent)
```

Notifica al chiamante un insuccesso nella chiamata e termina la sessione SIP.

```
static public URI getAlias(Proxy proxy,String id)
```

Ricerca all'interno della tabella delle registrazioni del proxy l'URI di un particolare utente; viene utilizzato dal **ProfileManager**.

5.3 Integrazione con il proxy

Per poter utilizzare questa architettura è necessario adattare il proxy NIST in modo tale che permetta al **ProfileManager** di gestire (attraverso gli oggetti *Profile*) i messaggi che arrivano allo stack SIP. Per fare questo è necessario comprendere l'architettura di funzionamento di questo e delle API JAIN SIP.

In generale per poter lavorare attraverso JAIN SIP è necessario creare lo stack SIP che si occupa di quello che riguarda le comunicazioni e il protocollo, in modo tale che questo si occupi di tutti i dettagli come ad esempio la gestione delle sessioni e delle transazioni.

Per poter elaborare i messaggi SIP in arrivo dallo stack, il proxy deve implementare l'interfaccia *javax.sip.SipListener*, che pubblica i seguenti metodi:

- `public void processRequest(RequestEvent requestEvent)`
  Viene chiamato all'arrivo di un comando SIP.

- `public void processResponse(ResponseEvent responseEvent)`
  Viene chiamato all'arrivo di una risposta ad un comando.

- `public void processTimeout(TimeoutEvent timeoutEvent)`
  Viene chiamato allo scadere di un timeout.

Ogni applicazione SIP che ascolta i messaggi in arrivo deve implementare questa interfaccia e registrarsi nello stack SIP.

In particolare nel caso del proxy, la classe principale (**examples.proxy.Proxy**) implementa direttamente questi metodi; per poter gestire correttamente il comportamento dei profili è necessario fare in modo che gli oggetti *Profile* elaborino tutti e tre gli eventi.

Quindi è necessario aggiungere del codice al proxy in questi tre metodi, prima che avvenga l'elaborazione predefinita, che richieda al **ProfileManager** i profili relativi agli utenti chiamati e passi a loro il controllo.

Un'altra modifica viene fatta nel costruttore della classe **Proxy** in cui viene effettuata l'inizializzazione del **ProfileManager**.

Durante lo sviluppo sono state eseguite altre piccole modifiche anche in altre parti del proxy, che però non sono necessarie durante il funzionamento normale del sistema, ma sono state aggiunte per scopi di debug.

5.4 Scenari

Il sistema è stato provato sia su piattaforma Windows sia su piattaforma Linux ottenendo i medesimi risultati, per funzionare ha bisogno di JDK (Java Development Kit) versione 1.4 o superiore, questo è dovuto a vincoli delle librerie NIST scritte utilizzando le funzionalità di questa versione.

La maggior parte dei test operativi è stata eseguita su sitemi Windows 2000 utilizzando come UA SipClient, un *softphone* sviluppato implementato in Java.

I test sono stati eseguiti utilizzando due PC su uno dei quali era in esecuzioni il proxy ed un client, e sull'altro uno o più client.

Per l'esecuzione del **ProfileServerManager** sono stati provati due differenti application server:

Tomcat fornito da Apache Foundation

Lite Web Server (LWS) sviluppato da Gefion Software

Ottenendo con entrambi ottimi risultati.

Come motori di database sono stati provati due differenti prodotti:

MySQL probabilmente il più diffuso DBMS open source disponibile praticamente per ogni piattaforma esistente sul mercato, estremamente leggero e funzionale.

HSqlDB DBMS sviluppato in java in grado sia di funzionare in modalità client/server sia direttamente all'interno di un'applicazione; nei test è stato usato nella prima modalità in quanto è necessario accedere al database attraverso due applicazioni separate (proxy/**ProfileManager** e web server/**ProfileServerManager**).

I test effettuati hanno considerato casi di:

1. Chiamate semplici.
   Sono costituite da profili in cui veniva eseguito direttamente l'inoltro della chiamata attraverso il tag *proxy*, e profili in cui veniva fatto seguire il comportamento predefinito, come descritto nel paragrafo 3.3.6.

2. Esecuzione di scelte.
   All'interno dei profili erano presenti elementi switch di tipo *address*, *string* e *time*, in cui, in base alle caratteristiche della chiamata venivano controllati i risultati attraverso la generazione di messaggi attraverso il nodo *log*. Sono stati anche eseguiti test con switch nidificati per verificare l'effettiva robustezza del meccanismo.

3. Redirezione.
   Sono state eseguite prove di reindirizzamento delle chiamate sia attravero il tag *redirect*, sia attraverso l'uso combinato di nodi *location* e *proxy*. E' stato anche provato l'uso combinato di switch insieme alla redirezione.

4. Redirezione multipla e condizionale.
   Si è provato a far redirigere la chiamata da un utente ad un altro nel caso il primo non fosse disponibile, utilizzando gli output del tag *proxy*.

I test presi in considerazione fino a questo punto riguardano principalmente il **ProfileManager**, oltre a questi è stata eseguita una serie di prove significativamente ampia per controllare il funzionamento del **ProfileServerManager**, inserendo un gran quantità di profili CPL di varia complessità per controllare che la creazione non generasse errori e producesse del codice Java valido.

5.5 Prestazioni

Uno degli obiettivi principali di questo sistema, come visto nel paragrafo 4.1.1, è quello di ottenere tempi di risposta brevi alle chiamate, in cui non andasse a pesare l'aggiunta della gestione dei profili.

Questi obiettivi sono stati raggiunti in quanto i tempi di attivazione delle chiamate sono analoghi a quelli del proxy senza modifiche, in quanto non sono eseguite operazioni costose in termini di tempo, soprattutto nella chiamate succesive alla prima in quanto i profili sono già contenuti all'interno della cache. Un altro incremento delle prestazioni è stato ottenuto attraverso la gestione razionale della cache facendo in modo di eliminare le chiamate inutili al repository tenendo riferimento anche degli utenti per cui non è presente il profilo.

Per quello che riguarda la compilazione dei profili il velocità è irrilevante, per quel che riguarda le prestazioni del sistema, in quanto avviene in maniera asincrona rispetto alle chiamate. La durata della creazione e memorizzazione di un profilo si aggira sui 3 - 4 secondi a seconda della complessità (e ovviamente dal tipo di macchina su cui è in esecuzione), in cui la maggior parte del tempo è utilizzato dal compilatore Java per tradurre il sorgente generato in bytecode.

La durata delle altre fasi è almeno un ordine di grandezza inferiore a quest'ultima. Ma in ogni caso la velocità di questa fase non influenza le prestazioni nell'esecuzione delle parti più sensibili da questo punto di vista, inotre questa operazione avviene sporadicamente in quanto la modifica di un profilo è un'operazione che può avvenire anche con intervalli di settimane se non mesi.

Capitolo 6

Conclusioni

In questo capitolo verranno presentati i risultati ottenuti in questo lavoro nelle fasi di sviluppo e di test, i problemi riscontrati, i possibili miglioramenti, e i nuovi campi di applicazione per cui può essere utilizzato.

6.1 Risultati ottenuti e problemi riscontrati

La fase di test e di debug di questo sistema è risultata particolarmente impegnativa, data la sua natura, in cui i vari moduli sono altamente interdipendenti gli uni dagli altri. In particolare la verifica e lo sviluppo del compilatore di profili, in cui tutte le operazioni di modifica avvenivano in maniera indiretta, in quanto ciò che veniva modificato non era codice che poteva essere compilato direttamente, ma elementi di programma che venivano compilati durante la fase di inserimento del profilo dal **ProfileServerManager**, e potevano essere controllati (in caso di compilazione positiva) soltanto dal **ProfileManager** effettuando una chiamata da un UA e analizzando i risultati dei log.

Risultava impossibile anche l'esecuzione passo passo in quanto le operazioni avvengono in maniera distribuita e asincrona (creazione del profilo, esecuzione di una chiamata, risposta di un client . . .), e soprattutto perché il codice del profilo veniva creato ed eseguito runtime, quindi non era disponibili all'IDE[1] utilizzato .

Di conseguenza gran parte delle operazioni di debug è stata effettuata attraverso i

[1]Integrated Development Enviroment cioè ambiente di sviluppo integrato.

registri[2] dei componenti in esecuzione. E' risultato estremamente utile l'analizzatore di protocollo, fornito insieme al proxy NIST, in quanto ha permesso di controllare l'effettivo traffico di messaggi SIP da e verso lo stack SIP del proxy.

Tutti i test effettuati hanno avuto esito positivo, in quanto il comportamento assunto dal sistema è risultato aderente alle specifiche. Sono stati comunque incontrati dei problemi, alcuni di tipo oggettivo, dovuti a limiti del sistema e incompatibiltà di versioni, altri invece risolvibili attraverso modifiche e aggiunte.

In particolare sono stati riscontrati:

- Problemi di incompatibilità verso determinati UA.
- Errata gestione del termine delle comunicazioni.
- Mancata gestione del timeout delle risposte SIP nel proxy.
- Lacune nell'implementazione del linguaggio CPL.
- Problemi nella creazione guidata dei profili.

Questi verranno analizzati nel dettaglio per esaminare con precisione i sintomi, le cause e le possibili soluzioni.

6.1.1 Incompatibilità da parte di determinati User Agent

Uno dei problemi incontrati all'inizio della fase test è stato quello di avere un'infrastruttura di prova affidabile, in quanto, utilizzando i software a disposizione, è stato difficile avere una situazione in cui le chiamata andavano buon fine anche senza utilizzare il sitema modificato.

Sono stati provati diversi UA come:

- HearMe sviluppato dalla società omonima, versione dimostrativa a tempo, versione nativa Win32[3]

[2]Per registri si intende listati di log.

[3]Per versione nativa Win32 si intende un programma eseguibile che funziona su piattaforme Windows a 32 bit, cioè tutte le versioni windows a partire dalla 95 che implementano le API a 32 bit.

- SipClient, sviluppato in Java dotato di tutte le funzionalità più avanzate di un softphone, per la gesione dei flussi audio viene utilizzata l'estensione Java JMF[4].

- JSPhone, fornito insieme alla distribuzione NIST del proxy e delle API JAIN SIP, scritto in Java, è ancora in uno stato di *work in progress*, quindi risulta tutt'altro che completo e semplice da usare.

Purtroppo tutti i client di una certa complessità non sono attualmente disponibili con licenza d'uso gratuita o dimostrativa, infatti i primi due utilizzati sono stati scritti e distribuiti prima del 2002 e non possibile trovarne una versione aggiornata liberamente utilizzabile. Il terzo non è una versione definitiva quindi non è risultato affidabile per il test di un altro sistema.

Inoltre, durante le prove, è risultato che nella maggior parte dei casi (anche senza l'utilizzo del proxy) i client non sono risultati completamente compatibili tra di loro, al punto che non si è riusciti ad effettuare una chiamata diretta (tra client differenti) con successo.

HearMe ha mostrato diversi problemi nel far passare le chiamate attraverso il proxy, quindi non è stato possibile utilizzarlo nei test. La soluzione ottimale, sopratutto in termini di affidabilità, è stata di utilizzare come client SipClient, sia come terminale chiamante sia come terminale chiamato, anche se anch'esso ha mostrato alcuni problemi nella fase di chiusura delle sessioni.

6.1.2 Errori nella chiusura delle sessioni SIP

In tutte le prove con l'uso del proxy, sia prima che dopo le modifiche con il **ProfileManager**, si è riscontrata un'anomalia nel comportamento nella fase di chiusura delle sessioni, in quanto l'ultimo messaggio di conferma di chiusura non riesce ad attraversare il proxy.

Si è tentato di risolvere il problema, ma dato che risulta congenito nel comportamento del proxy, cioè è presente anche senza utilizzare il **ProfileManager**, è stata considerata un'anomalia sistemica, dovuta probabilmente a lievi incompatibilità sul formato dei pacchetti SIP da parte dello stack NIST e dei client utilizzati.

[4]Java Media Framework, un'estensione Java per poter gestire tipi di dati multimediali.

Nella maggioranza dei casi questo malfunzionamento non porta ad alcun problema nelle funzionalità del sistema ma può venir trattato automaticamente dal meccanismo di gestione dei timeout dei client.

6.1.3 Gestione incompleta dei timeout del proxy

Alcuni comandi CPL permettono di definire un timeout per le operazioni di segnalazione, allo scadere del quale è possibile effettuare altre operazioni, ad esempio per gestire i casi di mancata risposta ad una chiamata.

Questo evento deve venir gestito dal proxy sotto forma di scadenza di un timeout che genera l'invocazione, da parte della stack SIP, del metodo *processTimeout(...)* dell'interfaccia *javax.sip.SipListener* (si veda il paragrafo 5.3) implementato nella classe principale del proxy (**examples.proxy.Proxy**). Infatti, come già visto nel capitolo precedente, all'interno di questo metodo viene passato il controllo al profilo per gestire anche queste situazioni.

Il problema riscontrato è che non si è riuscito a trovare il modo di impostare la durata del timeout da parte del profilo; quindi non è possibile in alcun modo specificare una durata precisa, operazione necesaria per alcuni comandi CPL (come ad esempio *proxy*) in è cui possibile inserire un tempo di attesa come parametro.

6.1.4 Implementazione incompleta del linguaggio CPL

Non tutti i comandi CPL visti nel capitolo 3 sono stati implementati nella gestione dei profili nel **ClassCompilerProfileCodec**, in particolare ci si è concentrati sugli elementi di utilizzo più comune dando la precedenza all'ottenere qualcosa di funzionante nella grande maggioranza delle applicazioni, e lasciando la possibilità di implementare le funzionalità mancanti in un secondo tempo, avendo in ogni caso un sistema funzionante.

In particolare gli elementi attualmente non implementati sono[5]:

- Il tag LOOKUP, che viene utilizzato per la risoluzione degli indirizzi, non è stato implementato in quanto il proxy utilizzato non prevede alcun meccanismo di questo tipo.

[5] Per le descrizioni complete si rimanda al relativo capitolo.

- Il tag MAIL non è stato implementato per dare la precedenza ad elementi più strettamente legati alle realtà di telefonia, anche se la filosofia NGN tende a non fare questo tipo di distinzioni. Inoltre si è voluto evitare la necessità di aggiungere le API Java Mail a tutte quelle già utilizzate per il funzionamento del proxy.

- Gli switch sulla priorità e su linguaggio non sono stati inseriti in quanto si è data la precedenza agli altri più comunemente utilizzati, in ogni caso la loro implementazione risulta estremamente semplice e lineare in quanto ricalcano gli altri switch presenti.

- Alcuni modalità del *time-switch* non sono state implementate in quanto di utilizzo limitato rispetto alle altre; in particolare non sono stati gestiti gli attributi: *count*, *wkst* e *bysetpos* con i relativi meccanismi. Sempre all'interno di questo switch non vengono gestite le informazioni sul fuso orario (definite attraverso gli attributi *tzid* e *tzurl*) quindi tutti gli orari inseriti sono relativi alle impostazioni locali della macchina su cui è in funzione il proxy.

6.1.5 Errori nella creazione guidata dei profili (wizard)

Si è notato, durante la fase di test, che i documenti CPL generati dal wizard (si veda il paragrafo 4.4) non sono sempre conformi al linguaggio CPL, in particolare quando viene inserita più di una regola contemporaneamente.

Inoltre la flessibilità nella creazione dei profili è estremamente limitata, in quanto è possibile specificare una sola condizione alternativa alla semplice chiamata in cui si può definire una condizione per ogni tipo di switch.

Questo è dovuto all'implementazione estremamente semplicistica del linguaggio CPL; non si è tentato di correggere questi comportamenti in quanto questo applet è il risultato di lavoro di terzi, e la creazione di un'interfaccia grafica per la creazione dei profili, con un'adeguata flessibilità conforme al linguaggio CPL, esula dagli obiettivi di questo lavoro.

6.2 Sviluppi futuri

Questo lavoro, benché completo e funzionante, non può essere considerato terminato, in quanto man mano che lo sviluppo procedeva si delineavano una serie miglioramenti, frutto dell'esperienza maturata e dei problemi incontrati, che potrebbero aumentare sensibilmente la qualità del sistema, sia in termini di affidabilita, sia di prestazioni. Questo processo, che risulta naturale nel normale ciclo di sviluppo di un software, ha portato alle seguenti idee:

- Gesione migliorata della cache: ad esempio sarebbe possibile gestire lo scadere dei profili attraverso dei *weak reference* o *soft reference* Java per permettere lo svuotamento automatico della memoria; inoltre sarebbe utile studiare diverse politiche di gestione della cancellazione dei profili (sempre dalla cache) in base ad esempio al controllo della frequenza delle chiamate, o di altre informazioni statistiche di utilizzo.

- Gestione delle chiamate concorrenti: attualmente ogni profilo può gestire l'arrivo di una sola chiamata contemporaneamente, dato che i profili non risultano *thread-safe*; di conseguenza sarebbe utile per una gestione più completa delle chiamate. Questo limite è nato dal fatto che il lavoro di integrazione sul proxy è iniziato su una versione precedente del proxy NIST che non gestiva più chiamate concorrenti (durante la fase di sviluppo è stata rilasciata dal NIST una nuova implementazione sia del proxy, sia dello stack JAIN SIP, e quindi tutto il lavoro è stato riadattato alla nuova architettura).

- Creazione di un strato di astrazione per i profili: sarebbe possibli riprogettare l'archittettura dei profili in cui viene slegato il codice generato del compilatore CPL dal proxy utilizzato, in questo modo verrebbe creata una mini API (che andrebbe implementata per ogni proxy usato) in cui sono presenti funzionalità di alto livello relative alle generiche funzionalità del proxy, che viene utilizzata dai profili generati. In questo modo il codice generato per i profili è indipendente dal proxy, e di conseguenza risulterebbe estremamente semplice adattare il **ProfileManager** per essere utilizzato da altri software.

Da un altro punto di vista sarebbe possibile adattare l'architettura del **ProfileManager** e del **ProfileServerManager** per funzionare in un contesto completamente diverso, in cui è necessario gestire dei profili non necessariamente di telefonia;

in questo caso sarebbe sufficiente sostituire il codec in modo tale che non traduca dal linguaggio CPL, ma dal linguaggio usato in quella determinata applicazione.

Sia il **ProfileManager**, sia il **ProfileServerManager** dovrebbero essere ritoccati solo in minima parte, esclusivamente nelle parti specifiche agli identificatori dei profili in cui è necessario rimuovere le parti relative agli indirizzi del protocollo SIP, mentre il resto risulterebbe del tutto riutilizzabile così com'è attualmente.

Parte III

Appendici

Appendice A

Procedure di configurazione

Viene qui di seguito riportato integralmente il file di documentazione presente nella distribuizione del progetto come riferimento per la configurazione e l'installazione del sistema.

ProfileManager and ProfileServerManager
by MDN

Author: Marco De Nittis (denittis@sinet.it)

Summary:
1. Directory structure
2. Compiling and packaging
3. Libraries used
4. Configuring and running the ProfileManager
5. Configuring and running the ProfileServerManager
6. Database configuration
7. Features
8. References

Notes:
In this note will be used some abbreviation:
PM - ProfileManager
PSM - ProfileServerManager
MDN - me
JCP - Java Comunity Process
NIST - National Institute of Standards and Tecnologies

---

1. Directory structure

```
- / -- src (a)
   | - lib (b)
   | - jsp (c)
   | - conf (d)
   | - classes (e) *
   | - dist (e) *
   | - temp (f)
   | - log (g) *
```

* Automatically created during compiling/packaging/running
a) Contains all the java sources, it has different branches:

```
- src -- it           contains package it.polito.cpl.**, it is all
      |               the main code of this project  (by MDN)
      |- awtesina     contains the classes of the wizard applet
      |- examples     contains all the classes of the proxy *
      |- tools        contains the classes of some tools used by
      |               the proxy (es traceviewer) (by NIST)
      |- gov          contains the NIST implementation of JAIN-SIP
      |               and JAIN-SDP standards (by NIST) **
      |- javax        contains the JAIN-SIP e JAIN-SDP standard
                      classes (by JCP) **
```

* some little modification on examples.proxy.Proxy by MDN
* those packages are not necessary for the build of this project
b) Contains all the jar libraries, it also has different directory:

```
- lib -  contains the libraries used by the PM and the proxy
      |- server     contains the libraries used by the PSM
      |             (and also the PM and proxy) *
      |- xerces     contains the xerces implementation
                    (present in the original NIST distribution)
```

* This directory is present because all the jars contained in are packed in the jsp war for the web app distribution.

c) Contains all the jsp and PSM elements: the jsp and html pages, PSM configuration file, web.xml (the web application configuration file) and the profile java template used by the ClassCompilerProfileCodec.

d) Contains the configuration files for the proxy and the PM.
In particular there is the xml conf file for the proxy (actually polito.xml) and a property conf file for the PM (deafult is profilemanager.conf)

e) Here are present all the compiled class file of the project

f) Here normally are created temporary java and class files generate by the class compiler (it can be reconfigured in the PSM conf file)

g) This directory is automatically created by the NIST proxy for storing its log (the location can be modified in the xml proxy conf file)

---

2. Compiling and packaging

For compiling and packaging it's present and ant(2) script (build.xml) for managing all the phases, there are present different targets:

clean deletes all the object files and dirs

polito compiles the PM and PSM components

proxy compiles the NIST-proxy components

build compiles the PM, the PSM and the proxy (default)

distPM packages all the PM classes in a single archive in the dist directory

distPSM packages all the PM classes in a single archive in the dist directory

distProxy packages all the NIST proxy classes in a single archive in the dist directory

jsp prepares the PSM distribution under the jsp directory with all the needed libs

war packages the war webapp distribution file (PSM) in the dist directory

dist packages all

cleanDist deletes all packaged files

nistLib compiles ths JAIN-SIP implementation (not necessary if the its library jar it's present in the classpath)

distNIST packages all the JAIN-SIP NIST implementation classes in a single archive in the dist directory

buildAll compiles and packages everything

All the target are chained to have the complete update.

For compiling the components:
>*ant*

For packing the PSM war file:

>*ant war*

For packing all the components:
>*ant dist*

Notes:
- It's important for compiling with ant to have the javac compiler in the ant classpath, normally it's stored in:*JAVA_HOME/lib/tools.jar*

- It's necessary to use the JDK 1.4 or greater for compiling the NIST - proxy.

---

3. Libraries used

During compilation are necessaries these libraries:

For the NIST-proxy and the PM:

nist-sip.jar containing the JAIN SIP (and JAIN SDP) NIST implementation *

mdnLogger.jar MDN logger classes

jakarta-regexp-xxx.jar regular expression engine from Apache jakarta (3)

For the PSM:

mdnLogger.jar MDN logger classes

jakarta-regexp-xxx.jar regular expression engine from Apache jakarta

commons-fileupload.jar Apache jakarta Mime-multipart upload library, used by the jsp pages

During the execution are necessary also other libraries, runtime loaded:

- JDBC driver for the DBProfileRepository according to the DB used, in the distribution are presents the mysql driver and the hsqldb standalone dbms.

For the PSM are also necessary in the classpath the proxy and the JAIN-SIP libraries, these are used only during the profile compiling phase.

---

4. Configuring and running the ProfileManager

Configuring

Since the PM is a part of the NIST proxy, it's necessary to configure also the proxy, for that I refer back to its documentation.
I assume that the correct proxy configuration file is placed in the conf directory (named for instance polito.xml).

The PM default configuration file is 'conf/profilemanager.conf', it's possibile to specify a different one using the enviroment variable 'it.polito.cpl.client.conf'.
An typical conf file can be:

```
# Repository
it.polito.cpl.repository=it.polito.cpl.DBProfileRepository

# Decoder
it.polito.cpl.profilecodec=it.pol...piler.ClassCompilerProfileCodec

# ProfileManager RMI
it.polito.cpl.client.cache.rmienabled=true
it.polito.cpl.client.cache.rmiport=1099
it.polito.cpl.client.cache.rmihost=localhost

# DBProfileRepository
```

```
it.polito.cpl.repository.db.driver=com.mysql.jdbc.Driver
it.polito.cpl.repository.db.url=jdbc:mysql://localhost/mdn
it.polito.cpl.repository.db.user=tesi
it.polito.cpl.repository.db.passwd=tesi

#Translations
it.polito.cpl.addressmatch=0(\\d\\d\\d\\d)@polito.it
it.polito.cpl.addresstranslate=sip:0{1}@sip.polito.it
```

* it.polito.cpl.repository
It's the classname of the repository used, for DBProfileRepository is
it.polito.cpl.DBProfileRepository

it.polito.cpl.profilecodec
The class name of the profile codec to install, for ClassCompilerProfile codec is it.polito.cpl.classcompiler.ClassCompilerProfileCodec

it.polito.cpl.client.cache.rmienabled Select to activate the cache rmi interface to update remotely (from the PSM) the cache contents

it.polito.cpl.client.cache.rmiport
Select the port where the RMI registry for the cache is running
it.polito.cpl.client.cache.rmihost
Select the host where the RMI registry for the cache is running

it.polito.cpl.addressmatch
It's used for setting up the address translation feature, contains the regular expression for address matching

it.polito.cpl.addresstranslate
Contains the pattern to translate the address if the match is succesfull

DBProfileRepository specific:

it.polito.cpl.repository.db.driver

Class name of the jdbc driver to use
* it.polito.cpl.repository.db.url
Connect string to use for establishing connection

it.polito.cpl.repository.db.user
it.polito.cpl.repository.db.passwd
DB username and password

* Running

If the cache RMI publishing is activate it's necessary to run also the RMIRegistry before running the proxy.
The RMI registry can be used also only by the proxy for registration features (refer to the NIST proxy documentation).

The proxy can be execute in two ways:
- bare proxy
- GUI with traceviewer feature

To execute the bare proxy the main class is examples.proxy.Proxy, for the GUI is examples.proxy.gui.ProxyLauncher, in both case is necessary to specify the proxy conf file, so for launching the proxy:

>java examples.proxy.Proxy -cf conf/polito.xml

for the GUI:

>java examples.proxy.gui.ProxyLauncher -cf conf/polito.xml

Anyway it's necessary to set up the classpath, it can be done automatically using the SmartLoader in this way:

>java -jar SmartLoader.jar examples.proxy.gui.ProxyLauncher -cf conf/polito.xml

The SmartLoader starts the application adding to the classpath all the jars present in the current dir and in 'lib' (and also its subdirs) and all the classes present in the current and in the 'classes' directories.

---

5. Configuring and running the ProfileServerManager

Configuring

The PSM it's not a standalone component, it lives inside an jsp enabled web server (from now called 'appserv') , it can be done including in the appserv the 'jsp' directory, or deploying its packed war archive.

The configuration file to use is in the 'jsp' directory and normally it's called 'profileservermanager.conf' (the name is configured inside the web.xml file), it has the same syntax of the PM one.
One possible conf file can be:

```
# ProfileServerManager configuration
it.polito.cpl.repository=it.polito.cpl.DBProfileRepository
it.polito.cpl.profilecodec=
              it.polito.cpl.classcompiler.ClassCompilerProfileCodec

# DBProfileRepository
it.polito.cpl.repository.db.driver=com.mysql.jdbc.Driver
it.polito.cpl.repository.db.url=jdbc:mysql://localhost/mdn
it.polito.cpl.repository.db.user=tesi
it.polito.cpl.repository.db.passwd=tesi

# ClassCompileProfileCodec
it.polito.cpl.codec.classcompiler.outputpath=d:/tesi/work/temp
it.polito.cpl.codec.classcompiler.compiler=com.sun.tools.javac.Main
it.polito.cpl.codec.classcompiler.preservesources=true
```

```
it.polito.cpl.codec.classcompiler.template=/ProfileTemplate.java

# Remote cache
it.polito.cpl.client.cache.rmienabled=true
it.polito.cpl.client.cache.rmiport=1099
it.polito.cpl.client.cache.rmihost=localhost
```

The most part is identical to the PM one, with the same meaning for the parameters. It can be seen an another part referring to the codec,in particular:

it.polito.cpl.codec.classcompiler.outputpath
The directory where are stored the temporary files created during the compilation phase, this directory MUST exist.

it.polito.cpl.codec.classcompiler.compiler
The class name of the compiler used for creating profiles, actually it has been configured to work with the SUN jdk compiler (class com.sun.tools.javac.Main tested with jdk 1.4.1) and the Eclipse (4) compiler (class org.eclipse.jdt.internal.compiler.batch.Main test with 2.1 version).

* it.polito.cpl.codec.classcompiler.preservesources
Can be true or false, it select if the sources and class files shuold be preserved or deleted (the default behaviour is to delete).

it.polito.cpl.codec.classcompiler.template
Contains the path to the profile template file starting from the PSM webapp root.

* Running

To run the PSM it's necessary to deploy the PSM web app to the appserv, the procedure depends on the server used, basically it can be done in 2 ways, the first is to configure the appserv to use also the jsp directory as web app root.

With Tomcat (ver 4.1.24) it can be done adding near the other Context nodes something like this:

```
<Context className="org.apache.catalina.core.StandardContext"
   crossContext="false" reloadable="true"
   mapperClass="org.apache.catalina.core.StandardContextMapper"
   useNaming="true" debug="16" swallowOutput="false" privileged="true"
   displayName="Tesi WebAPP" cachingAllowed="true"
   wrapperClass="org.apache.catalina.core.StandardWrapper"
   docBase="../../../tesi/work/jsp" cookies="true" path="/PSM"
   charsetMapperClass="org.apache.catalina.util.CharsetMapper">
           <Logger className="org.apache.catalina.logger.FileLogger"
             debug="0" verbosity="1" prefix="localhost_PSM_log."
             directory="logs" timestamp="true" suffix=".txt"/>
    </Context>
```

Modifing according to the local settings the attributes:
- displayName (description of the web app)
- docBase (local path to the 'jsp' dir)
- path (appserv virtual directory to reach the web app from the browser)

The second way (and the simpler one) is to use the war packed file with the appserv deploying tool, with tomcat it can be done with the manager webapp via the browser.
Note that if something has been modified, also only the configuration file, the web app must be repacked and redeployed again.

To access the the ProfileServerManager you have to open a web browser to:

http://hostname:8080/PSM/

Where hostname is the name of the host where the appserv is running, with the correct port (8080 is the default for tomcat), to the virtual dir specified during the deploy phase (in tomcat it's visible in the manager page).

** IMPORTANT NOTES **
1. All the library files used by the PSM are inside the war (or in the 'WEB-INF/lib'

dir if you use the unpacked version), but for compiling profiles it's absolutely necessary to add the classpath of the appserv (note: the appserv, NOT the web app) the necessary jars, because of some application server classloader isolation reasons.
With all the appserv tested this problem cannot be avoided with some workarounds.
The libraries used for compiling profiles are:
- nist-sip.jar (present in 'lib')
- compileLib.jar (present in 'dist')
- mdnLogger.jar (present in 'lib')

2. For validating CPL file correctly it's necessary to put in the appserv main directory the CPL DTD (cpl.dtd) present in the root directory of the distribution.

---

6. Database configuration

For using the DBProfileRepository as repository class, it's necessary to properly configure the DBMS used.

First the table that contains the profile has to be created, it contains 4 fields:
- Id as string of at least 50 characters it contains the profile id, if possible it should be the primary key.
- Timestamp as bigint, it will contains a java long value (the timestamp)
- ProfileObject as blob, it will contain the binary rappresentation of the profile
- Cpl as text blob, it will contain the cpl source of the profile

Here are present the DDLs relatives to MySQL and HsqlDB DBMS.

MySQL DDL

```
CREATE TABLE profiles (
  Id varchar(50) NOT NULL default '',
  Timestamp bigint(11) default NULL,
  ProfileObject blob,
  Cpl text,
```

```
  PRIMARY KEY (Id),
  UNIQUE KEY Id(Id)
) TYPE=MyISAM;
```

HsqlDB DDL

```
CREATE TABLE Profiles (
Id CHAR(50) NOT NULL ,
Timestamp BIGINT,
ProfileObject LONGVARBINARY,
Cpl LONGVARCHAR,
PRIMARY KEY(Id)
)
```

---

7. Features

Address translation
There is a built-in feature that allows to translate a set of addresses directly to different set using regular expressions (it's embedded in the ProfileManager).
The configuration is divided in two part: the matching pattern and the translation pattern.
The matching pattern is a standard regular expression to which the address of every call is compared, if the matching is succesfull this address is substituted with the translation pattern.
For example:
matching pattern =¿ sip:.+@sales.company.com (.+ means any string with at least 1 character, so any address in that domain)
translation pattern =¿ sip:sales@company.com

In this case every call directed to any address in the domain 'sales.company.com' is redirected to the sales@company.com address.

It's possible to place in the translated address some elements presents in the original address, this can be done using parenthesis in matching pattern and placeholders in the translation string.

A place holder is an element like that 'x' where x is the number of the brace group in the matching string, so '1' is substituted with the contents of the first brackets pair matched, '2' with the second and so on...
For example:
matching pattern => sip:.+@(.+).company.com
translation pattern => sip:1@company.com

That results in:
sip:foo@IT.company.com => sip:IT@company.com
sip:ceo@marketing.company.com => sip:marketing@company.com

I.e. everything that matched the '(.+)' pattern is taken and is placed in the translated address in the position of '1'

It's possible to use up to 10 parenthesis grouping, moreover the placeholder '0' correspond to the whole original address.

8. References

1. NIST - National Institute of Standards and Tecnology
http://snad.ncsl.nist.gov/proj/iptel/
2. Apache Ant project - http://ant.apache.org/
3. Apache Jackarta project - http://jakarta.apache.org/
4. Eclipse project - http://www.eclipse.org/

Appendice B

Struttura del linguaggio CPL

Il linguaggio XML ha il grande vantaggio di poter definire degli schemi precisi a cui devono aderire i documenti XML per poter essere considerati validi. Per il dialetto CPL è stato definito uno schema ben preciso attraverso un DTD (Document Type Definition)[1] , che descrive il linguaggio CPL, o meglio descrive la sua sintassi e vincoli che hanno i vari elementi del linguaggio.

Qui di seguito è possibile vedere il DTD che descrive il linguaggio CPL:

```
<!-- Nodes. -->
  <!-- Switch nodes -->
  <!ENTITY % Switch 'address-switch|string-switch|language-switch|
                    time-switch|priority-switch' >

  <!-- Location nodes -->
  <!ENTITY % Location 'location|lookup|remove-location' >

  <!-- Signalling action nodes -->
  <!ENTITY % SignallingAction 'proxy|redirect|reject' >

  <!-- Other actions -->
```

[1]Esistono due modi diversi per definire un tipo di documento XML, il primo è attraverso un DTD, il secondo è attravero un *XML Schema* che è anch'esso un documento XML che descrive i vincoli di un altro documento XML. Quest'ultimo sistema è stato introdotto più di recente ed è di gran lunga più potente e flessibile dei DTD, ma allo stesso tempo estremamente più complesso da utilizzare.

```
<!ENTITY % OtherAction 'mail|log' >

<!-- Links to subactions -->
<!ENTITY % Sub 'sub' >

<!-- Nodes are one of the above four categories, or a subaction.
     This entity (macro) describes the contents of an output.
     Note that a node can be empty, implying default action. -->
<!ENTITY % Node     '(%Location;|%Switch;|%SignallingAction;|
                     %OtherAction;|%Sub;)?' >


<!-- Switches: choices a CPL script can make. -->

<!-- All switches can have an 'otherwise' output. -->
<!ELEMENT otherwise ( %Node; ) >

<!-- All switches can have a 'not-present' output. -->
<!ELEMENT not-present ( %Node; ) >

<!-- Address-switch makes choices based on addresses. -->
<!ELEMENT address-switch ( address*, (not-present, address*)?,
                           otherwise? ) >
<!-- <not-present> must appear at most once -->
<!ATTLIST address-switch
   field         CDATA    #REQUIRED
   subfield      CDATA    #IMPLIED
>

<!ELEMENT address ( %Node; ) >

<!ATTLIST address
   is            CDATA    #IMPLIED
   contains      CDATA    #IMPLIED
```

```
   subdomain-of  CDATA    #IMPLIED
> <!-- Exactly one of these three attributes must appear -->


<!-- String-switch makes choices based on strings. -->

<!ELEMENT string-switch ( string*, (not-present, string*)?,
                          otherwise? ) >
<!-- <not-present> must appear at most once -->
<!ATTLIST string-switch
   field         CDATA    #REQUIRED
>

<!ELEMENT string ( %Node; ) >
<!ATTLIST string
   is            CDATA    #IMPLIED
   contains      CDATA    #IMPLIED
>  <!-- Exactly one of these two attributes must appear -->


<!-- Language-switch makes choices based on the originator's preferred
     languages. -->

<!ELEMENT language-switch ( language*, (not-present, language*)?,
                            otherwise? ) >
<!-- <not-present> must appear at most once -->

<!ELEMENT language ( %Node; ) >
<!ATTLIST language
   matches      CDATA     #REQUIRED
>


<!-- Time-switch makes choices based on the current time. -->
```

```
<!ELEMENT time-switch ( time*, (not-present, time*)?, otherwise? ) >
<!ATTLIST time-switch
   tzid          CDATA    #IMPLIED
   tzurl         CDATA    #IMPLIED
>

<!ELEMENT time ( %Node; ) >

<!-- Exactly one of the two attributes "dtend" and "duration"
     must occur. -->
<!-- The value of "freq" is (daily|weekly|monthly|yearly).  It is
        case-insensitive, so it is not given as a DTD switch. -->
<!-- None of the attributes following freq are meaningful unless freq
         appears. -->
<!-- The value of "wkst" is (MO|TU|WE|TH|FR|SA|SU).  It is
        case-insensitive, so it is not given as a DTD switch. -->
<!ATTLIST time
   dtstart       CDATA  #REQUIRED
   dtend         CDATA  #IMPLIED
   duration      CDATA  #IMPLIED
   freq          CDATA  #IMPLIED
   until         CDATA  #IMPLIED
   count         CDATA  #IMPLIED
   interval      CDATA  "1"
   bysecond      CDATA  #IMPLIED
   byminute      CDATA  #IMPLIED
   byhour        CDATA  #IMPLIED
   byday         CDATA  #IMPLIED
   bymonthday    CDATA  #IMPLIED
   byyearday     CDATA  #IMPLIED
   byweekno      CDATA  #IMPLIED
   bymonth       CDATA  #IMPLIED
   wkst          CDATA  "MO"
```

```
   bysetpos       CDATA  #IMPLIED
>


<!-- Priority-switch makes choices based on message priority. -->

<!ELEMENT priority-switch ( priority*, (not-present, priority*)?,
                            otherwise? ) >
<!-- <not-present> must appear at most once -->

<!ENTITY % PriorityVal '(emergency|urgent|normal|non-urgent)' >

<!ELEMENT priority ( %Node; ) >

<!-- Exactly one of these three attributes must appear -->
<!ATTLIST priority
   less           %PriorityVal;  #IMPLIED
   greater        %PriorityVal;  #IMPLIED
   equal          CDATA          #IMPLIED
>


<!-- Locations: ways to specify the location a subsequent action
     (proxy, redirect) will attempt to contact. -->

<!ENTITY % Clear  'clear (yes|no) "no"' >
<!ELEMENT location ( %Node; ) >
<!ATTLIST location
   url            CDATA    #REQUIRED
   priority       CDATA    #IMPLIED
   %Clear;
>
<!-- priority is in the range  0.0 - 1.0.  Its default value SHOULD
      be 1.0 -->
```

```
<!ELEMENT lookup ( success?,notfound?,failure? ) >
<!ATTLIST lookup
  source        CDATA    #REQUIRED
  timeout       CDATA    "30"
  use           CDATA    #IMPLIED
  ignore        CDATA    #IMPLIED
  %Clear;
>

<!ELEMENT success  ( %Node; ) >
<!ELEMENT notfound ( %Node; ) >
<!ELEMENT failure ( %Node; ) >

<!ELEMENT remove-location ( %Node; ) >
<!ATTLIST remove-location
   param        CDATA   #IMPLIED
   value        CDATA   #IMPLIED
   location     CDATA   #IMPLIED
>


<!-- Signalling Actions: call-signalling actions the script can
     take. -->

<!ELEMENT proxy ( busy?,noanswer?,redirection?,failure?,default? ) >

<!-- The default value of timeout is "20" if the <noanswer> output
     exists. -->
<!ATTLIST proxy
   timeout      CDATA   #IMPLIED
   recurse      (yes|no) "yes"
   ordering     (parallel|sequential|first-only) "parallel"
>
```

```
<!ELEMENT busy ( %Node; ) >
<!ELEMENT noanswer ( %Node; ) >
<!ELEMENT redirection ( %Node; ) >
<!-- "failure" repeats from lookup, above. -->
<!ELEMENT default ( %Node; ) >
<!ELEMENT redirect EMPTY >
<!ATTLIST redirect
   permanent     (yes|no) "no"
>


<!-- Statuses we can return -->

<!ELEMENT reject EMPTY >
<!-- The value of "status" is (busy|notfound|reject|error), or a SIP
     4xx-6xx status. -->
<!ATTLIST reject
   status        CDATA    #REQUIRED
   reason        CDATA    #IMPLIED
>

<!-- Non-signalling actions: actions that don't affect the call -->

<!ELEMENT mail ( %Node; ) >
<!ATTLIST mail
   url           CDATA    #REQUIRED
>

<!ELEMENT log ( %Node; ) >
<!ATTLIST log
   name          CDATA    #IMPLIED
   comment       CDATA    #IMPLIED
>
```

```
<!-- Calls to subactions. -->

<!ELEMENT sub EMPTY >
<!ATTLIST sub
   ref           IDREF    #REQUIRED
>


<!-- Ancillary data -->

<!ENTITY % Ancillary 'ancillary?' >

<!ELEMENT ancillary EMPTY >


<!-- Subactions -->

<!ENTITY % Subactions 'subaction*' >
<!ELEMENT subaction ( %Node; )>
<!ATTLIST subaction
   id            ID       #REQUIRED
>


<!-- Top-level actions -->

<!ENTITY % TopLevelActions 'outgoing?,incoming?' >

<!ELEMENT outgoing ( %Node; )>

<!ELEMENT incoming ( %Node; )>
```

```
<!-- The top-level element of the script. -->

<!ELEMENT cpl  ( %Ancillary;,%Subactions;,%TopLevelActions; ) >
```

Appendice C

Strumenti utilizzati

Tutti gli strumenti, i componenti e le librerie utilizzati per l'ideazione, lo sviluppo, l'utilizzo, il test, la documentazione, la stesura e l'impaginazione di questo lavoro sono tutti di tipo open source (con licenza GPL o a questa assimilibile) o freeware, in particolare:

Eclipse project come ambiente integrato di sviluppo, da IBM corporation, sotto licenza CPL, `www.eclipse.org`

TexNicCenter usato per la stesura di questo elaborato.

LaTeX per l'impaginazione.

UmLet per la creazione di diagramm UML

Omondo UML plug-in per Eclipse per UML, licenze miste

MySQL database di test

hSQLDB database di test

TomCat web server e container JSP

LWS web server e container JSP

KeyNote

Jakarta ANT

Appendice D

Acronimi

API Application Programming Interface

AST Abstract Syntax Tree

B2B Business to Businness

B2C Business to Consumer

CORBA Common Object Request Broker Architecture

CPL Call Processing Language

DTD Document Type Definition

HTTP HyperText Transport Protocol

IDE Integrated Development Enviroment

IETF Internet Engineering Task Force

ISDN Integrated Service Digital Network

J2ME Java 2 Micro Edition

JAIN Java APIs for Integrated Network

JAR Java ARchive

JAXP Java Api for XML Processing

JCAT Java Coordination e Transaction

JCC Java Call Control

JCP Java Community Process

JDBC Java DataBase Connectivity

JDK Java Development Kit

JRE Java Runtime Enviroment

JSP Java Server Pages

JVM Java Virtual Machine

LDAP Lightweight Directory Access Protocol

NGN Next Generation Network

NIST National Institute of Standards and Teconology

OSA Open Service Access

PSTN Public Switched Telephone Network (rete telefonica tradizionale)

QoS Quality of Service

RMI Remote Method Invocation

RTP Real Time Protocol

SCS Service Capability Service

SCF Service Capability Features

SCML Service Creation Markup Language

SDP Session Definition Protocol

SIP Session Initiation Protocol

SMTP Simple Mail Transfer Protocol

SOAP Simple Object Access Protocol

SPA Service Provider API

SPI Service Provider Interface

UA User Agent

UDDI Universal Discovery, Description e Integration

URI Uniform Resource Identifier

URL Uniform Resource Locator

XML eXtensible Markup Language

WAR Web application ARchive

WSDL Web Services Description Language

WSFL Web Services Flow Language

XAML Transaction Authority Markup Language

XTML eXtensible Telephony Markup Language

VoIP Voice over IP

Bibliografia

[1] NGN Initiative Roadmap 2002 http://www.ngni.org

[2] ACIF NGN Framework Option Group http://www.acif.org.au/ngn_fog

[3] Eurescom Project P1109: "Next Generation Networks: The services offering standpoint", http://www.eurescom.de/secure/projects/P1100-series/P1109/P1109.htm

[4] P.Falcarin, C.Licciardi: "Analysis of NGN service creation technologies", pubblicato su "IEC Annual review of communications", volume 56, aprile 2003

[5] European Telecommunications Standards Institute http://www.etsi.org

[6] The Parlay Group www.parlay.com

[7] Staish Thatte: "XLANG, Web Services for Business Process Design" Microsoft, 2001

[8] Transaction Authority Markup Language http://www.xaml.org

[9] JAIN Homepage http://java.sun.com/products/jain/overview.html

[10] JAIN SIP Java Community Process http://jcp.org/en/jsr/detail?id=32 http://java.sun.com/products/jain/overview.html

[11] RFC 2824[1] "Call Processing Language Framework and Requirements" http://www.ietf.org/rfc/rfc2824.txt

[12] J.L. Bakker ,R. Jain : "Next Generation Service Creation Using XML Scripting Languages", 2002, www.argreenhouse.com/papers/jlbakker/bakker-icc2002.pdf

[13] Session Initation Protocol Homepage, Columbia University http://www.cs.columbia.edu/sip/

[14] RFC 3261 "SIP Session Initiation Protocol" http://www.ietf.org/rfc/rfc3261.txt

[1] Come si cita un RFC?

[15] RFC 2616 "Hypertext Transfer Protocol – HTTP/1.1" http://www.ietf.org/rfc/rfc2616.txt

[16] SIP RADVision white paper http://www.radvision.com/papers/C1_What_is_SIP.html

[17] National Institute of Standards and Teconology, JAIN SIP RI http://snad.ncsl.nist.gov/proj/iptel/

[18] J. Kuthan, D. Sisalem: "Session Initiation Protocol Tutorial ", iptel.org , http://iptel.org/sip/

[19] SIP Servlets specification. On-line at http://www.jcp.org/en/jsr/detail?id=116

[20] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, "Extensible markup language (XML) 1.0" W3C Recommendation REC-xml- 20001006, World Wide Web Consortium (W3C), Oct. 2000. http://www.w3.org/XML/.

[21] RFC 3066 "Tags for the Identification of Languages" http://www.ietf.org/rfc/rfc3066.txt

[22] RFC 2445 "Internet Calendar and Sceduling Core Object Specification (iCalendar COS)" http://www.ietf.org/rfc/rfc2445.txt

[23] Jakarta Project - Apache Foundation http://jakarta.apache.org/

Ringraziamenti

Ringrazio con tutto il cuore le persone che mi sono state vicine in questi anni, in particolare la mia famiglia, per tutto l'affetto, la pazienza, il supporto e il calore che mi hanno sempre fatto sentire, con la consapevolezza che senza la loro presenza tutto questo non sarebbe stato possibile.

Desidero ringraziare inoltre tutte le persone che hanno vissuto insieme a me quest'avventura, in particolare Isabella per le giornate condivise e per aver reso più piacevoli anche i momenti di difficoltà. Un grazie anche a Francesca che mi ha sempre spinto a dare di più sopratutto all'inizio di questo viaggio, e a Paolo Falcarin per la sua disponibilità.