Università degli Studi di Pisa
Facoltà di Ingegneria
Corso di Laurea Specialistica in Ingegneria Informatica

Anno accademico 2006/07

Tesi di Laurea Specialistica

Un'architettura per la contrattazione e la gestione della qualità del servizio in applicazioni web

Candidato:
Gaetano F. Anastasi

Relatori:
Prof. P. Ancilotti
Prof. G. Lipari

Sommario

Oggetto di questa tesi è la progettazione e lo sviluppo di un'architettura che consenta di contrattare e gestire la qualità del servizio (QoS) fornita da applicazioni web. In particolare, la tesi verrà incentrata sullo studio di applicazioni di tipo soft real-time che offrono i propri servizi tramite un'interfaccia web. Tali applicazioni sono spesso soggette a particolari politiche aziendali, che possono prevedere di differenziare la fornitura del servizio in base al tipo di utenza. Nasce da qui la necessità di contrattazione della QoS tra il cliente ed il fornitore. Il problema verrà affrontato con riferimento ad un modello basato sulla stipula di accordi. Affinché la QoS contrattata possa essere effettivamente garantita dal soggetto fornitore, è necessario che si utilizzi un'opportuna politica di gestione delle risorse. In questa tesi si farà riferimento a tecniche di resource reservation all'interno della struttura concorrente di un server web.

Infine, si verificherà che l'utilizzo di un'architettura di questo tipo permette di fornire garanzie di qualità del servizio difficile da garantire altrimenti, soprattutto in condizioni di carico elevato del sistema.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Scopo della tesi è quello di progettare e realizzare un'architettura software che permetta di effettuare la contrattazione e la gestione della qualità del servizio in tutte quelle applicazioni, basate su Web, dove vi siano tali esigenze. In questo capitolo verrà dunque inizialmente inquadrato il problema, verrà poi definito il contesto in cui si muove questo lavoro ed infine si metterà in risalto, anche attraverso l'analisi di lavori correlati, il contributo innovativo apportato da questa tesi.

1.1 Definizione del problema

Le applicazioni in cui è particolarmente importante la gestione della "qualità del servizio" sono solitamente indicate come ***soft real-time*** ([1]). In generale, un'applicazione real-time è caratterizzata da impliciti vincoli temporali: tali vincoli devono essere rispettati per garantire un corretto funzionamento dell'applicazione. Le comuni applicazioni real-time, cosiddette *hard real-time*, sono utilizzate in sistemi critici, per cui se l'applicazione viola un vincolo temporale ciò può causare eventi catastrofici nel sistema. Le applicazioni soft real-time, invece, non sono utilizzate in sistemi critici, dunque la violazione di un vincolo temporale comporta solamente un degrado nelle prestazioni del sistema, che sarà tanto maggiore quanto maggiore sarà il numero di violazioni dei vincoli temporali dell'applicazione.

Le prestazioni di un sistema determinano la soddisfazione degli utenti che lo utilizzano. Una misura di tale soddisfazione è data dalla cosiddetta **qualità del servizio** dell'applicazione (Quality of Service o QoS). Il termine "qualità del servizio" viene utilizzato in diversi contesti con diversi significati ma in ogni caso è legato all'impatto che il comportamento del sistema ha sugli utilizzatori: dunque maggiore è il grado di soddisfazione dell'utente, maggiore sarà la QoS. Nel caso in cui un sistema sia composto da diverse applicazioni con requisiti temporali, un importante parametro che influenza la soddisfazione dell'utente è il tempo di risposta di ogni sin-

gola applicazione. Minore è il tempo di risposta dell'applicazione, maggiore sarà la soddisfazione dell'utente e dunque maggiore sarà la QoS. Il tempo di risposta di un'applicazione dipende, in assenza di precauzioni particolari, non solo dal comportamento temporale dell'applicazione stessa ma anche da quello delle altre applicazioni. In un contesto soft real-time, il comportamento temporale di un'applicazione è determinato dal numero di violazioni temporali e dall'entità di tali violazioni: per poter fornire una certa qualità del servizio, un sistema soft real-time deve dunque tenere sotto controllo le violazioni dei vincoli temporali delle applicazioni.

Le applicazioni soft real-time sono diffuse in diversi domini applicativi, fra i quali possono essere citati i sistemi multimediali, le reti di comunicazione, la realtà virtuale, i videogames interattivi. In questa tesi ci si concentrerà particolarmente su quelle applicazioni soft real-time che possono essere rese disponibili come servizi tramite il Web. Esempi tipici di applicazioni di questo tipo possono essere le applicazioni di video on demand, le applicazioni distribuite o le applicazioni relative a transazioni finanziarie.

I servizi cui si è fatto riferimento potrebbero essere forniti da un'azienda che, seguendo delle politiche interne, differenzia il servizio proposto in base alle classi di utenza. Inoltre, un utente che usufruisce di tali servizi, potrebbe avere necessità di ricevere precise garanzie sul servizio richiesto. In presenza di questo tipo di necessità, che si verificano soprattutto se il servizio viene fornito a scopo commerciale, si rende indispensabile una contrattazione della qualità del servizio.

1.2 Definizione del dominio

Per quanto detto in fase di definizione del problema, questo lavoro si muove all'interno di tre diversi ambiti, quello delle applicazioni web, quello della contrattazione della qualità del servizio e quello della gestione della qualità del servizio. Per ognuno di essi si indicheranno di seguito le tecnologie di supporto presenti allo stato dell'arte, fra le quali si individueranno quelle che più si adattano al problema in esame.

1.2.1 Applicazioni web

Grazie all'enorme popolarità di Internet e alla diffusione della banda larga, le applicazioni che utilizzano il Web come piattaforma sono innumerevoli.

Una da citare è sicuramente il commercio elettronico, utilizzato sia per le transazioni commerciali che avvengono tra un'azienda ed il cliente finale (transazioni b2c o business-to-customer) sia per le transazioni che avvengono tra due aziende o tra un'impresa e i suoi fornitori (transazioni b2b o business-to-business).

Un'altro esempio di applicazione web è costituito dai programmi per ufficio, come gli elaboratori di testi o i fogli di calcolo, i quali vengono sempre più

spesso resi disponibili dalle aziende produttrici tramite un'interfaccia web e possono dunque essere utilizzati disponendo di un comune browser. Un'azienda che applica tale politica è ad esempio Google ([26]), con il progetto "Google Documenti".

Un'ulteriore esempio di applicazione web può essere quello dei cosiddetti video *on demand* (VoD): numerosissimi sono ormai i portali che permettono all'utente di scegliere i contenuti multimediali preferiti, per poterli comodamente visualizzare in tempo reale senza doverli preventivamente scaricare sul proprio PC.

Per l'utilizzo di questi servizi, in accordo ad una certa politica aziendale, potrebbero esistere degli utenti o delle classi di utenti privilegiate, per le quali è necessario fornire una certa qualità del servizio.

Lo stato dell'arte

Il protocollo HTTP è il protocollo di comunicazione più utilizzato nella rete Internet per la visualizzazione e il trasferimento delle informazioni. Dunque il *server web*, il cui compito primario è ricevere ed elaborare richieste HTTP, può essere considerato come la porta di accesso alle applicazioni che utilizzano il Web come piattaforma. Tuttavia, in una tipica interazione client-server tramite Internet, il server web può da solo svolgere come compito quello di prelevare una pagina web dal disco fisso e presentarla al client a seguito di una sua richiesta. Per poter generare i contenuti dinamici che sono alle base di tutte le applicazioni web presenti tuttora è invece necessario che il server web si appoggi ai cosiddetti *content-generator*, che elaborano le informazioni necessarie per generare una nuova pagina web da presentare al client. Fra i generatori di contenuti ve ne sono alcuni, come i programmi (o script) con interfaccia cgi e gli script in linguaggio php, che eseguono all'interno del server web; altri invece, come le servlet Java, devono essere eseguite, all'esterno del web server, dai cosiddetti *application server*, che sono dei veri e propri server specializzati nell'esecuzione di quei programmi che si avvalgono di complesse infrastrutture di supporto.

Gli application server, inoltre, rivestono un ruolo fondamentale per quello che riguarda i *Web Services*, che sono sicuramente una delle applicazioni web più interessanti emerse negli ultimi tempi. Un web service è un componente software che viene reso disponibile tramite la rete e può dunque essere utilizzato da altre applicazioni nell'ottica del riuso e dell'interoperabilità, nonché in un ottica commerciale. Chi propone i propri servizi tramite questo sistema, li pubblica in indici appropriati, denominati UDDI, e li descrive tramite WSDL (Web Service Description Language) ([15]), un linguaggio di specifica basato su XML. Chi vuole accedere al servizio comunica con il fornitore tramite il protocollo SOAP([16]), attraverso cui avviene uno scambio di messaggi XML incapsulati all'interno di richieste e risposte HTTP. Il protocollo SOAP viene spesso implementato tramite l'uso di linguaggi ad

alto livello ed è per questo che le applicazioni basate su tecnologia web services hanno generalmente bisogno di un application server. Gli application server sono nati come architettura *middleware* (letteralmente che "sta in mezzo"), spesso tuttavia essi si sono evoluti inglobando in loro stessi la logica per gestire anche le richieste http e poter essere così utilizzati in maniera *stand-alone* (Apache Tomcat ne è un esempio).

Per quanto riguarda questo lavoro si è deciso di integrare i meccanismi di supporto alla qualità del servizio all'interno dei server web. Essi infatti costituiscono sicuramente la base da cui partire per lo studio delle applicazioni web, senza contare che a tutt'oggi offrono comunque supporto ad un maggior numero di applicazioni rispetto agli application server.

La scelta di Apache2

Il web server che si utilizzerà nel corso di questo lavoro è **Apache**, che risulta il più adatto ai nostri scopi per due ragioni principali. La prima riguarda la natura *open-source* del software: risulta fondamentale infatti la possibilità di disporre del codice sorgente del server, nell'ottica di studiarne la struttura concorrente per poterlo poi modificare in base alle nostre esigenze. La seconda ragione riguarda invece l'enorme diffusione di Apache: ad Aprile 2007, Apache è infatti il web server più utilizzato, supportando il 58.50% dei siti attivi sulla rete Internet [1], ovvero quasi il 70% in più di quelli supportati tramite i web server proprietari forniti da Microsoft (quelli della famiglia IIS). Tale enorme diffusione, unita alla sua caratteristica di essere open source, ne fanno un software molto affidabile (può essere infatti costantemente esaminato e corretto), particolarmente adatto dunque all'uso in sistemi di produzione reali.

Il web server Apache esiste ed è utilizzato in due diverse versioni, una è quella 1.3, l'altra viene invece identificata genericamente come **Apache2**. Per questo lavoro si utilizzerà Apache2, perché, dopo aver analizzato le differenze tra le due famiglie, ci si è resi conto che meglio rispondeva alle nostre esigenze. Le motivazioni a riguardo sono: (a) migliore leggibilità del codice, dovuta alla riorganizzazione in chiave modulare dell'architettura; (b) introduzione di nuove API, con le quali aumentano le funzionalità che è possibile garantire con i moduli; (c) supporto ai thread Unix, con i quali si può migliorare la scalabilità del web server.

1.2.2 Contrattazione della qualità del servizio

I contesti in cui vi è un'interazione tra un soggetto consumatore ed un soggetto fornitore di servizi possono essere molteplici ed i servizi offerti possono essere i più vari: la gestione di una macchina server o l'assistenza su un prodotto software o una richiesta di pagamento inviata tramite Web. In tutti

[1] fonte Netcraft (www.netcraft.com), Aprile 2007 Web Server Survey

questi casi, molto diversi fra loro, se il cliente richiede una certa qualità del servizio (QoS), affinché questa possa essere garantita dal fornitore vi è comunque la necessità tra le parti di accordarsi sulle modalità di fruizione del servizio.

Un accordo del genere viene, molto spesso, specificato tramite l'uso dei *Service Level Agreement* (SLA), veri e propri "contratti" tramite i quali il fornitore da un lato descrive i servizi erogati e i vincoli che egli garantisce di rispettare nell'esecuzione di un servizio, dall'altro definisce anche gli eventuali costi associati al servizio e le penalità che si impegna a corrispondere in caso di mancato rispetto dei vincoli.

Lo stato dell'arte

In passato, la stipula di un SLA, prevedeva che vi fosse un contatto o un incontro fisico tra le organizzazioni che incarnano i due soggetti: tali organizzazioni davano vita ad un accordo statico e generalmente di lunga durata, per il quale i tempi di un'eventuale modifica non erano certo rapidi. Questo metodo per la stipulazione può essere ancora accettabile in contesti in cui si erogano servizi a basso livello, la cui fornitura si prevede debba essere prolungata nel tempo.

Di certo, invece, non risulta adeguato nei casi in cui i servizi siano offerti tramite il Web, un contesto per natura molto dinamico. E' proprio per questo che negli ultimi anni la comunità scientifica si è prodigata per individuare un metodo per stipulare i contratti in maniera automatica, dinamica e flessibile. Le soluzione ottenute, inoltre, sebbene ricercate per poter sfruttare appieno le potenzialità del Web e delle interazioni elettroniche, possono essere comunque proficuamente applicate in tutti gli altri campi, perché permettono di automatizzare le interazioni e di rendere gli accordi fra le parti più accurati, con la valutazione in tempo reale delle risorse da dedicare all'esecuzione di un servizio.

Le architetture emerse per creare e monitorare Service Level Agreement sono sostanzialmente due: **WSLA** ([9]), architettura sviluppata da IBM e **WS-Agreement** ([8]), sviluppata da Open Grid Forum. Tali architetture presentano approcci in alcuni punti molto simili. Innanzitutto entrambe hanno come obiettivo la flessibilità, questo perché, come già evidenziato, gli ambienti di potenziale utilizzo sono molto eterogenei fra loro. In secondo luogo esse prevedono che gli accordi specifichino sia i servizi erogati dal fornitore sia i vincoli che egli si impegna ad onorare. Tali vincoli, negoziabili, dipendono fortemente dal servizio associato ma sono spesso dei parametri relativi alla qualità del servizio, come ad esempio il tempo di risposta del fornitore. Per questo motivo entrambe le architetture prevedono che un accordo definisca anche come i parametri della qualità del servizio verranno misurati: in tal modo il soggetto consumatore può effettivamente verificare il rispetto delle garanzie che gli sono state fornite. Questa operazione, definita

di “monitoraggio”, viene adeguatamente supportata da appositi meccanismi che ne permettono la realizzazione.

La scelta di WS-Agreement

Nonostante le similitudini negli approcci alle problematiche di fondo, esistono alcune differenze fra le WSLA e WS-Agreement. Una analisi di tali differenze, basata su [10], ha fatto propendere, nell'ambito di questo lavoro, per l'utilizzo di WS-Agreement.

La caratteristica maggiormente distintiva di WS-Agreement è che gli accordi sono definiti indipendentemente da come il fornitore rende pubblici i servizi forniti: questo ha come vantaggio quello di poter definire degli accordi su servizi già esistenti, senza modificarne l'implementazione. In quest'ottica, WS-Agreement si limita a definire solo le interfacce degli accordi, lasciando completa libertà riguardo all'implementazione dei servizi. WSLA, invece, interviene in maniera a volte invasiva: in tale architettura, ad esempio, viene definita la struttura delle metriche per la misurazione dei parametri di QoS.

La nostra scelta, effettuata sulla base delle motivazioni fornite, viene inoltre rafforzata dalla decisione degli stessi sviluppatori di WSLA, i quali nella prospettiva del raggiungimento di uno standard condiviso, hanno abbandonato il proprio progetto per supportare invece WS-Agreement.

1.2.3 Gestione della qualità del servizio

Una certa qualità del servizio può essere garantita da un fornitore solamente se questo gestisce adeguatamente le proprie risorse interne. E' necessario dunque che il fornitore si avvalga di meccanismi di basso livello che supportino la gestione della QoS, permettendo di dedicare determinate risorse esclusivamente all'esecuzione di un servizio per cui si sono offerte delle garanzie. Il tipo ed il numero di queste risorse dipenderanno rispettivamente dal servizio che si deve fornire e dalla qualità che si vuole assicurare. In questa sede, analizzeremo i meccanismi di supporto alla gestione della QoS che possono essere utilizzati quando la fornitura di un servizio è realizzata dall'esecuzione di un task.

Lo stato dell'arte

Una delle risorse principali da tenere in considerazione quando si deve eseguire un task è il processore, visto che il tempo di esecuzione di una particolare applicazione è sicuramente legato alla qualità del servizio fornito. Molte delle applicazioni rese disponibili come servizi, sono pensate per essere eseguite con sistemi operativi “general purpose”, in cui è però impossibile avere delle garanzie relative ai tempi di esecuzione. Questo aspetto è da non sottovalutare, soprattutto nel caso in cui si debbano eseguire applicazioni soft

real-time, in cui le violazioni degli impliciti requisiti temporali sono causa di malfunzionamento e dunque di cattiva erogazione del servizio. Per poter eseguire questo tipo di applicazioni è necessario dunque che un sistema operativo general purpose sia supportato da un meccanismo di *protezione temporale*, che garantisca che il comportamento temporale di un task non sarà influenzato da quello di tutti gli altri presenti nel sistema.

Nei sistemi operativi general purpose, i task sono attivati dinamicamente ed il loro comportamento non può essere preventivamente determinato, ragion per cui, in essi, uno dei modi migliori per fornire isolamento temporale è quello di utilizzare tecniche di *Resource Reservation* ([2]): con tali tecniche si fornisce isolamento ad un task semplicemente riservandogli una certa frazione del tempo di esecuzione del processore. Infatti si può pensare che, con l'uso di questa tecnica, un task venga eseguito da un processore virtuale dedicato, la cui velocità è una frazione di quella del processore reale: da un lato, dunque, le eventuali anomalie di esecuzione del task vengono confinate all'interno di quel tempo di esecuzione, dall'altro il task non è influenzato dal resto del sistema poiché gli viene garantito l'uso esclusivo della CPU in quella frazione di tempo.

Vista la popolarità, la disponibilità per varie piattaforme e la possibilità di disporre del codice sorgente, Linux è particolarmente adatto come base per l'implementazione dei meccanismi di Resource Reservation ed effettivamente molte sono le implementazioni basate su questo kernel. Nel corso di questo lavoro si farà riferimento all'implementazione proposta e descritta in [3], che ha la caratteristica, distintiva rispetto alle altre, di non soffrire i problemi relativi alle anomalie di schedulazione che si possono verificare in conseguenza di attivazione aperiodiche dei task. Questo pregio è una conseguenza del particolare algoritmo di schedulazione utilizzato in [3], come ivi evidenziato. Tale implementazione può essere agevolmente utilizzata perché presente nel framework di AQuoSA ([22]), di cui si farà uso nel corso di questo lavoro.

Da notare, infine, che un altro modo possibile per garantire protezione temporale ai task è quello di utilizzare tecniche di Dynamic Soft Real-Time ([14]), come viene fatto all'interno del progetto GARA ([13]), che realizza in tal modo la gestione della QoS. Tali tecniche però possono soffrire di problemi legati a lunghe latenze tra l'istante di arrivo di un task e la sua esecuzione. Ciò è dovuto al particolare modello di schedulazione, che prevede che i task da eseguire in real-time inizino la loro esecuzione a normale priorità (tale priorità verrà poi cambiata in priorità real-time da un demone, il quale esegue anch'esso a normale priorità dinamica). Il problema delle possibili lunghe latenze è facilmente riconducibile all'impossibilità di garantire immediatezza nei tempi di risposta di un servizio, dunque tali tecniche risultano poco adatte al nostro problema.

La scelta di AQuoSA

L'architettura di AQuoSA ([22]) è stata utilizzata in questo lavoro per fornire supporto alla gestione della qualità del servizio. Tale scelta è dettata principalmente dal fatto che AQuoSA permette di utilizzare facilmente, tramite API, robusti meccanismi di schedulazione basati su *reservation* (vedi sezione precedente).

AQuoSA ha inoltre diverse caratteristiche rimarcabili, che vengono di seguito descritte:

- *Admission control*: garantisce che l'immissione nel sistema di una nuova applicazione con garanzia temporali non interferisca con le garanzie delle altre applicazioni dello stesso tipo.
- *Security policies*: vi è la possibilità di attuare particolari politiche di sicurezza, fornendo all'amministratore di sistema la facoltà di assegnare delle garanzie temporali specifiche per alcuni utenti o gruppi.
- *Feedback control layer*: è un livello di controllo che permette di adattare l'allocazione della CPU per un task, in accordo alle sue attuali esigenze.

1.3 Contributi di questa tesi

In questo lavoro, la qualità del servizio contrattata da un soggetto consumatore tramite SLA, dipende dalla gestione delle risorse fatta all'interno dell'architettura. Anche diverse altre architetture fanno uso di contrattazione tramite SLA per la gestione delle risorse (si veda [11]). Tali architetture, fra le quali una delle più rimarcabili è il VIOLA MetaScheduling Service ([12]), sono generalmente utilizzate in ambienti GRID ed hanno dunque come obiettivo principale quello di gestire risorse appartenenti a più nodi, per poter così eseguire applicazioni distribuite. Queste applicazioni sono spesso eseguite in *batch mode* e solitamente richiedono un'elevata capacità computazionale: in tale contesto dunque è importante garantire che il servizio venga eseguito e non è invece prioritario che esso venga eseguito immediatamente. Per la loro natura, dunque, architetture come il VIOLA MetaScheduling Service si differenziano dalla nostra, in quanto non affrontano il problema di fornire risposta immediata del servizio, cosa che invece è di fondamentale importanza in applicazioni *web oriented*.

Per poter fornire in breve tempo risposta del servizio è necessario che ad una richiesta segua un'immediata esecuzione del task che realizza il servizio, senza che questo possa essere in alcun modo influenzato dalla concorrente esecuzione di altri servizi. Si è visto come un modo per realizzare questo comportamento consista nel riservare una percentuale della CPU all'esecuzione di quel servizio: la nostra architettura dunque gestisce il processore

come risorsa principale. Quello che viene proposto in questa architettura è una gestione del processore fatta attraverso le tecniche di *resource reservation*, con le quali si riesce ad avere brevi tempi di latenza tra l'arrivo di un task e la sua esecuzione ed inoltre si ottengono le garanzie di isolamento temporale tra i task necessarie al controllo delle prestazioni del sistema. Il contributo principale di questa tesi consiste dunque nell'integrazione di tali tecniche di gestione della QoS all'interno della struttura concorrente di un server web. In questo modo, qualsiasi applicazione che viene resa disponibile tramite interfaccia web può essere fornita ad un soggetto consumatore, che può fruirla con una certa qualità del servizio. Tale integrazione potrebbe aprire un nuovo scenario nell'ambito della fornitura di servizi web, soprattutto in quelli in cui la contrattazione della QoS viene fatta a scopi commerciali. Infatti, se si rispettano i vincoli di garanzia richiesti dalla schedulazione real-time sottostante, si riesce a far sì che un servizio, richiesto con la stessa QoS da più soggetti, venga eseguito in un intervallo di tempo costante. In questo modo si elimina l'impredicibilità legata al tempo di esecuzione del servizio che, se unita ad analoghe tecniche applicate ad esempio alla rete o alla gestione del disco fisso, porterebbe ad una completa conoscenza a priori di tutte le garanzie che è effettivamente possibile fornire. Si eliminerebbe così la possibilità, per un fornitore di servizi, di incorrere in penali dovute al mancato rispetto dei vincoli di QoS garantiti in fase di contrattazione.

1.4 Struttura della tesi

Nel proseguo di questa tesi verranno inizialmente esposti i concetti su cui si basa il lavoro realizzato. In particolare, nel capitolo 2 verrà trattato il problema della contrattazione della QoS tramite accordi, con particolare riferimento alle soluzioni proposte dal framework WS-Agreement. Nel capitolo 3 si analizzerà invece il problema di come fornire supporto alla gestione della QoS, con riferimento alle tecniche di resource reservation di AQuoSA. Nel capitolo 4 si studierà la struttura concorrente ed il funzionamento di un server web, per mezzo del quale verranno eseguite le richieste di servizio. Il server web preso in esame è Apache2, al quale si farà riferimento anche nel capitolo 5, in cui si illusteranno i metodi e le tecniche per estenderne le funzionalità. Il capitolo 6 riguarda invece in maniera specifica l'architettura realizzata in questa tesi: in tale capitolo i concetti introdotti in precedenza verranno utilizzati per il progetto e lo sviluppo dell'architettura in esame. Nel capitolo 7 verranno condotti alcuni esperimenti, per poter analizzare qual è il comportamento dell'architettura, in relazione all'esecuzione di servizi per cui si devono fornire garanzie di QoS. Infine, nel capitolo 8 si tireranno alcune conclusioni e si indicheranno delle possibili strade da percorrere, in futuro, sul sentiero tracciato da questa tesi.

Capitolo 2

WS-Agreement

WS-Agreement è un'architettura nata con lo scopo di fornire un metodo per la definizione e la gestione di *Service Level Agreement*. Come ampiamente discusso nella sezione 1.2.2, un Service Level Agreement è un "contratto", tramite il quale due soggetti, generalmente un consumatore di servizi ed un fornitore di servizi, stipulano degli accordi da rispettare reciprocamente.

WS-Agreement è una proposta dello *Open Grid Forum* ed in particolare di uno dei suoi gruppi di lavoro, il *GRAAP WG* (Grid Resource Allocation and Agreement Protocol Working Group). Il GRAAP Working Group ha curato le specifiche dell'architettura ([8]), prefiggendosi il raggiungimento dei tre seguenti obiettivi: (a) stabilire un linguaggio per definire accordi con cui descrivere le potenzialità di un fornitore di servizi; (b) definire un protocollo per la creazione di accordi sulla base di modelli; (c) specificare un'interfaccia per verificare il rispetto di un accordo a tempo di esecuzione.

In questo capitolo si analizzeranno appunto tali elementi, non prima di aver presentato però il modello concettuale su cui si basa WS-Agreement.

2.1 Modello concettuale

Il modello concettuale per l'architettura di un sistema che si basa su WS-Agreement, prevede una suddivisione in due livelli:

- livello di accordo (*agreement layer*), tramite cui si fornisce una interfaccia, basata su *Web services*, con la quale creare, rappresentare e monitorare gli Agreement;

- livello di servizio (*service layer*), con il quale si indica invece il livello in cui il servizio descritto viene eseguito.

Tale suddivisione viene evidenziata dalla figura 2.1. In essa si fa riferimento al caso in cui è il soggetto consumatore ad iniziare l'interazione

per la creazione dell'accordo: si dice in questo caso che il consumatore assume il ruolo dell'*Agreement Initiator*, mentre il fornitore assume il ruolo dell'*Agreement Responder*.

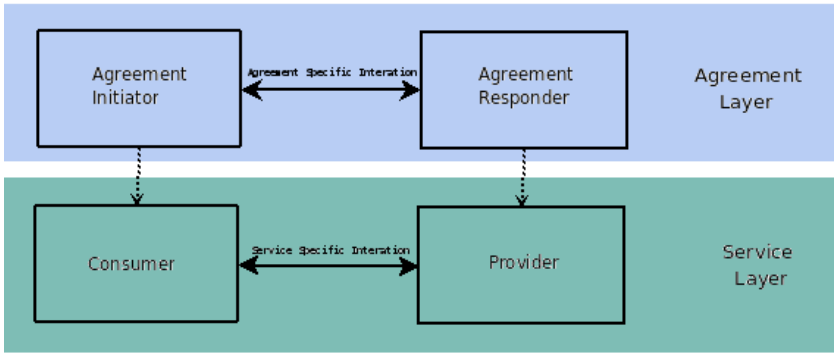

Figura 2.1: Modello concettuale WS-Agreement

Il modello a due livelli permette di rendere WS-Agreement adatto a qualsiasi dominio applicativo, poiché si colloca ad un livello superiore a quello in cui il servizio viene fornito. Inoltre, il livello di servizio potrebbe essere relativo ad una applicazione già esistente: in questo caso la separazione dei livelli permette una successiva aggiunta del livello di accordo senza che si comprometta quello che si trova più in basso.

2.2 Modello di un accordo

Un accordo tra un cliente ed il fornitore di un servizio può avvenire solamente se essi condividono un linguaggio comune su cui intendersi. L'architettura WS-Agreement definisce tale linguaggio utilizzando una grammatica XML con cui specificare un accordo (*Agreement*) o un prototipo di accordo (*Agreement Template*). In questa sezione verrà analizzata appunto la struttura di un Agreement e la struttura di un Agreement Template.

2.2.1 Struttura di un Agreement

Un Agreement è un documento XML, per cui sarà composto da vari elementi, ciascuno dei quali può avere diversi attributi. Nel descrivere la struttura di un **Agreement** (figura 2.2) si utilizzerà la notazione XPath([19]), attraverso cui si possono indirizzare parti di un documento XML. Nel seguito verranno descritte in maniera discorsiva le caratteristiche principali degli elementi costitutivi di un Agreement, per una esposizione esaustiva si rimanda invece a [8].

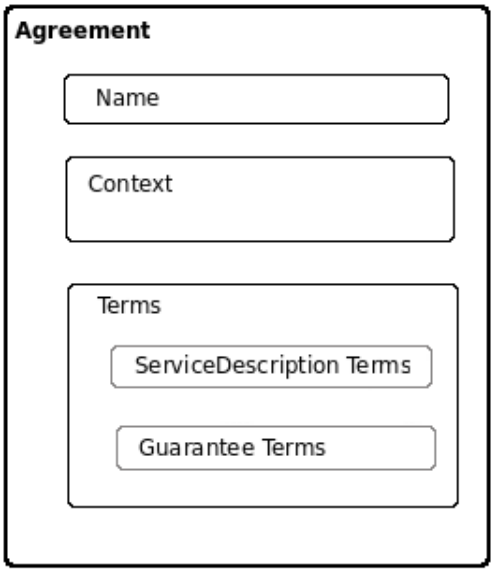

Figura 2.2: Struttura di un Agreement

Agreement/@AgreementId Attributo, di carattere obbligatorio, che costituisce un identificatore univoco dell'accordo.

Agreement/Name Campo opzionale che specifica il nome che può essere dato ad un agreement. Il nome non può essere considerato come un identificatore univoco dell'agreement.

Agreement/Context Contiene varie informazioni che riguardano l'accordo, come i partecipanti coinvolti, il tempo di validità, il prototipo da cui l'accordo è stato generato.

Agreement/Terms/ServiceDescriptionTerms In questa sezione vengono specificati i servizi associati all'agreement. Ciascun Service Description Term (SDT) può descrivere interamente o parzialmente uno dei servizi forniti. Un SDT consiste di tre parti: un nome per il SDT; un nome per il servizio che viene descritto in questa sezione dell'accordo; la descrizione del servizio in un linguaggio specifico del dominio applicativo.

Agreement/Terms/GuaranteeTerms In questa sezione si specificano soprattutto quali sono le garanzie che si vogliono assicurare nell'esecuzione di un servizio definito all'interno di un SDT. Si possono specificare inoltre quali sono le condizioni che devono essere rispettate affinchè un servizio possa essere fornito, nonchè dei valori aziendali che esprimono l'importanza di raggiungere un determinato obiettivo descritto nell'accordo.

2.2.2 Struttura di un Agreement Template

Un Agreement viene generato a partire da un prototipo, un **Agreement Template**, che costituisce dunque una guida di riferimento su cui basarsi. Un prototipo può essere pensato come una sorta di "modello prestampato" in cui alcuni campi sono lasciati in bianco affinchè possano essere successivamente riempiti. Fuor di metafora, un Agreement ed un il rispettivo template hanno, come evidenziato in figura 2.3, la stessa struttura, eccetto che per una sezione aggiuntiva che viene di seguito descritta.

Template/CreationConstraints Questa sezione ha lo scopo di indicare quali sono i campi configurabili di un accordo e quali sono i valori accettabili per riempire tali campi. Se i vincoli indicati non sono rispettati nella richiesta per la creazione di un accordo allora tale richiesta sarà giudicata non conforme con il prototipo di riferimento.

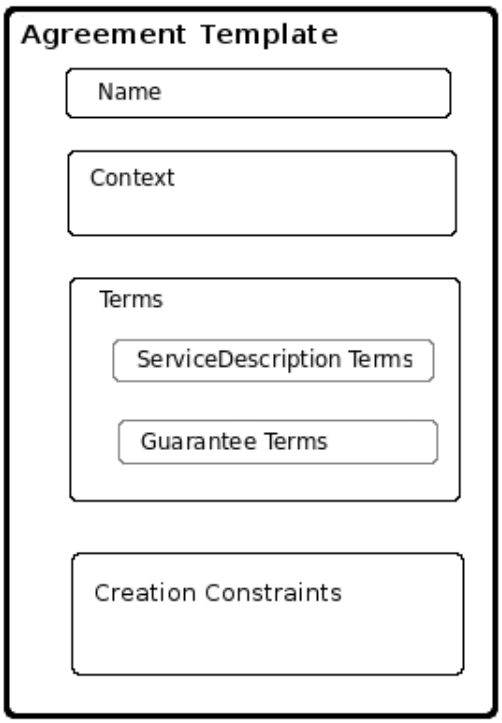

Figura 2.3: Struttura di un Agreement Template

Vi è inoltre da notare che ogni Template è contraddistinto da un identificatore univoco (`TemplateId`) che viene utilizzato per associare ogni accordo al prototipo di riferimento dal quale è stato generato.

2.3 Modello di interazione

WS-Agreement specifica il modello astratto tramite cui interagiscono le due parti coinvolte nella creazione e nella gestione di accordi. I protocolli di trattative definiti da WS-Agreement riguardano le modalità di creazione di un accordo e le modalità con cui un accordo viene monitorato per verificarne il rispetto. Nel seguito tali protocolli verranno analizzati in maniera approfondita.

2.3.1 Protocollo di creazione

WS-Agreement definisce un protocollo di trattative da effettuare per giungere alla creazione di un nuovo accordo. Le due parti coinvolte sono indicate come *Agreement Initiator* e *Agreement Responder*: il primo è l'iniziatore del processo di trattative, mentre il secondo risponde alle richieste di accordo ed è colui che fornisce i prototipi. E' da notare che questi due ruoli, relativi al livello di accordo, sono indipendenti dai ruoli a livello di servizio: ciò avviene in maniera conforme al principio di separazione dei livelli espresso dal modello concettuale dell'architettura. Se ci si restringe all'ambito delle applicazioni web, però, si avrà generalmente che l'iniziatore dell'accordo sarà il consumatore del servizio, mentre chi risponde è il fornitore del servizio stesso.

Facendo riferimento a questa identificazione dei ruoli, si avrà che il fornitore del servizio pubblicherà uno o più prototipi, in cui si decrivono servizi e vincoli offerti. Quando il consumatore vorrà creare un contratto inizierà le trattative prelevando un template ed imposterà poi, in base alle proprie esigenze, i campi da compilare. Essi, verosimilmente, corrisponderanno a parametri di qualità del servizio. Il consumatore a questo punto invia la sua offerta al fornitore, il quale valuta la proposta ricevuta e decide se accettarla o meno: in caso positivo viene creato un nuovo Agreement ed il fornitore da quel momento si impegna a rispettarlo; in caso negativo viene invece inviata all'utente una notifica di rifiuto.

Questo modello di interazione client-server è di tipo sincrono e viene realizzato da un particolare componente dell'Agreement Responder. Tale componente, denominato *Agreement Factory*, espone, tramite interfaccia Web Service, le due operazioni *createAgreement(A)* e *getTemplates()*, che devono essere invocate dall'iniziatore di un accordo. Come mostrato in figura 2.4, l'iniziatore recupera i prototipi di accordo con la getTemplates(), sceglie un prototipo dalla lista, ne riempie i campi ed invia la sua offerta al fornitore con la createAgreement(A). Se il fornitore accetta, viene creato un nuovo Agreement ed il servizio inizia ad essere erogato come concordato tra le parti.

E' da notare che, oltre al modello sincrono presentato, le specifiche WS-Agreement prevedono anche la possibilità di utilizzare un modello di inte-

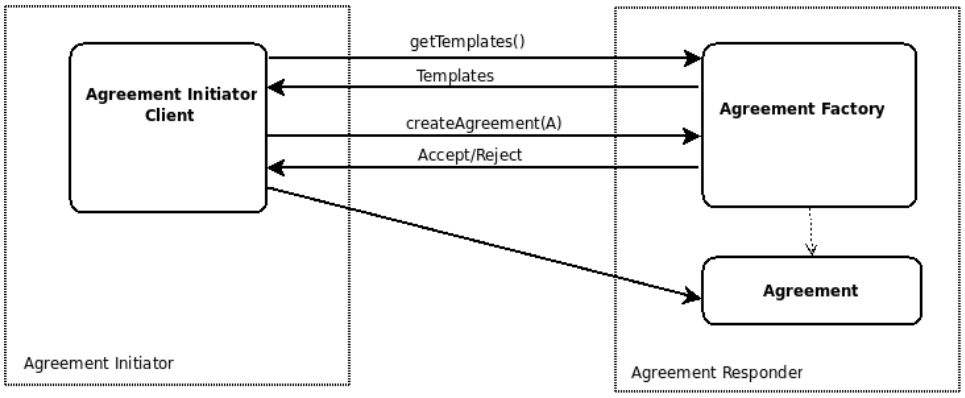

Figura 2.4: Interazioni nella creazione di un Agreement

razione asincrono, particolarmente utile quando il processo di decisione del Responder è molto lungo. Il modello asincrono viene realizzato da un altro componente del Responder, denominato *PendingAgreementFactory*, il quale, a seguito di un'operazione *CreatePendingAgreement* (analoga della CreateAgreement()), ritorna subito al chiamante senza aver preso una decisione sull'accettazione dell'accordo. L'iniziatore dell'accordo potrà poi conoscere la decisione in due modi: il primo è quello che fa uso di attesa attiva, il secondo prevede invece che l'iniziatore stesso debba esporre due operazioni, *Accept()* e *Reject()*, che verrano invocate dal Responder al termine del processo di decisione, per indicare se l'accordo è stato o meno accettato.

Se si fa riferimento alle specifiche WS-Agreement ([8]), si potrà notare come i componenti AgreementFactory e PendingAgreementFactory vengano indicati come *port type*. Questo è dovuto al fatto che, come già accennato, WS-Agreement utilizza un'interfaccia basata su Web Services, dove ogni servizio viene descritto tramite il linguaggio WSDL. Nella terminologia WSDL 1.1 ([15]), infatti, un port type corrisponde pressappoco ad una libreria di metodi. Ogni descrizione dei port type presenti nelle specifiche WS-Agreement, contiene infatti sia la definizione di alcuni metodi, indicati con il termine *operation*, sia la definizione di alcune ResourceProperty, a ciascuna delle quali è associato un metodo di tipo *getResourceProperty()* con lo stesso nome. Le ResourceProperty specificano quei valori dello stato di una risorsa WS-Resource ([17]) che possono essere visualizzati dal richiedente del servizio: ad esempio, un Template è una risorsa che può essere acceduta in lettura tramite l'esposizione del metodo getTemplates() da parte del AgreementFactory (che è un port type).

2.3.2 Protocollo di monitoraggio

In questa sezione verrà analizzato il protocollo di monitoraggio definito da WS-Agreement. Tale protocollo permette soprattutto di verificare, a tempo

di esecuzione del servizio, il rispetto delle garanzie fornite con l'accordo. Inoltre, permette di controllare lo stato in cui si trova l'accordo e lo stato di esecuzione dei servizi associati a tale accordo. I metodi attraverso cui è possibile fare ciò sono esposti da un componente dell'Agreement Responder. Tale componente è denominato *Agreement*, ad indicare che esso incarna un accordo che è già stato creato: ne deriva che esisterà un componente Agreement per ogni accordo creato.

Il componente Agreement espone, oltre ai metodi connessi al monitoraggio, altri metodi, che illustreremo brevemente. Tramite alcuni di essi è possibile recuperare informazioni sull'accordo creato: ciascuno di questi metodi (*getName*, *getAgreementId*, *getContext*, *getTerms*) restituisce infatti le parti dell'accordo di cui porta il nome. Poi, un ulteriore metodo che può essere invocato riguardo ad un accordo, è il metodo *terminate*, che consente all'iniziatore di interrompere l'accordo in qualsiasi istante.

I metodi connessi alle operazioni di monitoraggio verranno invece esaminati con maggiore grado di approfondimento: essi sono i metodi attraverso cui si possono monitorare lo stato di un accordo, di un servizio o delle garanzie.

Lo **stato di un Agreement** può essere monitorato tramite il metodo *getAgreementState()*, che restituisce i seguenti valori:

- `Pending` - Indica che è stata fatta un'offerta ma questa non è ancora stata valutata;
- `PendingAndTerminating` - Indica che è stata fatta un'offerta e che questa non era ancora stata valutata quando è arrivata una richiesta di terminazione da parte dell'Initiator;
- `Observed` - Indica che un'offerta è stata ricevuta ed accettata;
- `ObservedAndTerminating` - Indica che è stata fatta un'offerta, che questa è stata accettata e che è arrivata una richiesta di terminazione da parte dell'Initiator (richiesta che deve ancora essere valutata dal Responder);
- `Rejected` - Indica che l'offerta ricevuta non è stata accettata;
- `Complete` - Indica che un'offerta è stata ricevuta, accettata e tutte le attività che la riguardano sono state completate;
- `Terminated` - Indica che l'accordo è stato terminato esplicitamente dall'Initiator.

Gli stati attraverso cui può passare un Agreement durante il suo ciclo di vita possono essere visualizzati attraverso il diagramma di figura 2.5. In esso si evince come `Pending`, `Observed` e `Rejected` sono gli unici possibili stati iniziali di un Agreement, a cui si arriva dallo stato di transizione

`OfferReceived`, non esposto. Si nota inoltre, come dagli stati PendingAndTerminating e ObservedAndTerminating si può ritornare nei rispettivi stati di partenza, se la richiesta di terminazione fatta dall'iniziatore dell'accordo viene respinta.

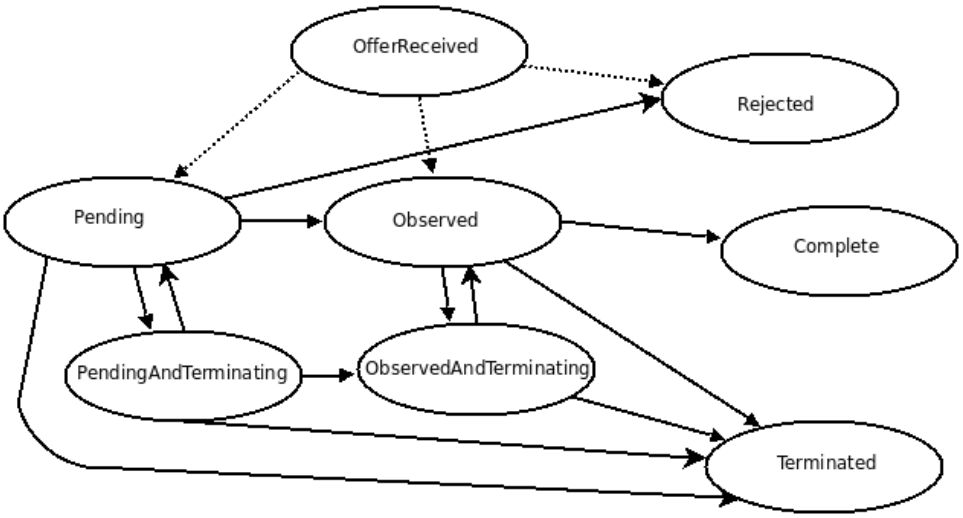

Figura 2.5: Diagramma degli stati di un Agreement

Un'altra operazione di monitoraggio che può essere compiuta è quella dello **stato del servizio**, effettuabile tramite il metodo *getServiceTermState()*. Tale metodo, anche in questo caso esposto dal fornitore del servizio, restituisce i seguenti valori:

- `Not Ready` - E' lo stato iniziale del servizio, indica che la sua esecuzione non è stata ancora iniziata;
- `Ready` - Indica che il servizio è genericamente pronto. Tale stato ha due sottostati che lo specificano maggiormente: `Processing` indica che il servizio è in esecuzione; `Idle` indica che il servizio non è correntemente in uso o in esecuzione;
- `Completed` - Indica che l'interazione con il servizio è stata completata.

Lo stato di un servizio è strettamente connesso a quello delle garanzie ad esso associate. Infatti se un servizio si trova negli stati `Not Ready` o `Ready`, lo stato delle sue garanzie non è determinato.

Il monitoraggio dello **stato delle garanzie** può essere effettuato tramite il metodo *getGuaranteeTermState()*, il quale restituisce i seguenti valori:

- `Fulfilled` - Indica che una garanzia è attualmente soddisfatta;
- `Violated` - Indica che una garanzia è attualmente disattesa;

- `NotDetermined` - E' lo stato iniziale di una garanzia, nel qual caso indica che nessuna attività riguardo alla garanzia si è ancora verificata. Durante l'esecuzione del servizio, indica invece che il rispetto della garanzia è in corso di verifica.

Si nota, dunque, come attraverso il metodo getGuaranteeTermState() si può verificare l'effettivo adempimento di un contratto nel corso dell'esecuzione del corrispondente servizio.

Capitolo 3

AQuoSA

AQuoSA (Adaptive Quality of Service Architecture)([22]) è un progetto open-source sviluppato all'interno del Real Time Systems Laboratory ([23]) della Scuola Superiore Sant'Anna di Pisa. Esso è nato con lo scopo di fornire, all'interno di Linux, meccanismi di schedulazione a supporto della gestione della qualità del servizio per applicazioni *soft real-time*.

AQuoSA permette di supportare le necessità delle applicazioni soft real-time introducendo, in un sistema operativo general-purpose come Linux, diverse funzionalità, fra le quali l'isolamento temporale e la schedulazione adattativa.

In questo capitolo si discuterà inizialmente dell'approccio alla gestione della QoS utilizzato da AQuoSA, successivamente verrà analizzata l'architettura del framework ed infine si focalizzerà l'attenzione su uno dei componenti dell'architettura, scelto per il particolare rilievo che ha nell'ambito di questo lavoro.

3.1 Approccio alla gestione della QoS

Nelle applicazioni soft real-time, il mancato rispetto dei vincoli temporali intrinseci nell'applicazione, non è critico per il funzionamento del sistema, tuttavia ne influenza le prestazioni: maggiore sarà il numero di violazioni dei vincoli temporali, minori saranno le prestazioni del sistema. Le prestazioni di un sistema sono legate alla soddisfazione degli utenti che lo utilizzano, quindi minori saranno le prestazioni minore sarà la qualità del servizio (QoS), intesa appunto come misura del grado di soddisfazione dell'utente. L'obiettivo primario di un sistema real-time è dunque quello di tenere sotto controllo le violazioni dei vincoli temporali delle applicazioni, per poter così assicurare una certa qualità del servizio.

Affinchè si possano supportare applicazioni soft real-time in un sistema operativo general purpose è necessario che vi siano meccanismi che consentono di poter effettuare una schedulazione predicibile, come timer ad alta

risoluzione e bassi tempi di latenza del kernel. Tali meccanismi, presenti nel kernel Linux, non sono però sufficienti. Linux, infatti, allo stato attuale, supporta la schedulazione dei task per attività real-time solamente con algoritmi a priorità fisse. Tali scheduler soffrono di problemi relativi all'equità (*fairness*) e alla sicurezza. Se, ad esempio, un utente non privilegiato potesse accedere allo scheduler a priorità fissa, la sua applicazione, attivata alla massima priorità, bloccherebbe il sistema semplicemente con l'esecuzione di un ciclo infinito. D'altra parte, se solamente gli utenti privilegiati potessero accedere alle funzionalità di schedulazione real-time, sarebbe difficile fornire garanzie di esecuzione agli utenti non privilegiati. Per evitare tali problematiche è necessario garantire dunque anche *isolamento temporale* tra i vari task del sistema, per far sì che il comportamento (dal punto di vista temporale) di un task non possa influenzare quello degli altri. In particolare, la proprietà di isolamento temporale può essere definita come segue.

Definizione 1 *Un algoritmo di schedulazione si dice che garantisce isolamento temporale se la capacità di ogni task τ_i con $(i = 1, ..., n)$ di rispettare i suoi vincoli temporali dipende solamente dall'evoluzione del carico di lavoro del task.*

Un metodo efficace, utilizzato anche all'interno di AQuoSA, per fornire isolamento temporale ai task del sistema, è quello che fa uso di algoritmi di schedulazione appartenenti alla classe degli algoritmi Reservation Based (RB)([2]). Essi si basano sul concetto di Resource Reservation: ad ogni task viene "riservata" una certa frazione di una risorsa condivisa, che in questo caso è la CPU.

Consideriamo un insieme di task indipendenti $\tau_1, ..., \tau_n$ che condividono il processore. In una schedulazione RB una *reservation* è rappresentata da una coppia (Q, P): per cui se ad un task τ_i è associata una reservation (Q_i, P_i), quel task sarà eseguito per un certo tempo Q_i ogni periodo P_i. Il periodo di tempo P_i si definisce come periodo di reservation, mentre il rapporto $B_i = Q_i/P_i$ viene definito *banda*. La schedulazione RB garantisce l'isolamento temporale tra i task, per cui si può assumere che un task, a cui è associata una reservation, esegua su una CPU virtuale dedicata, la cui velocità è pari alla frazione B_i della CPU reale. In tal modo si garantisce dunque che la QoS di un'applicazione non viene influenzata dal comportamento temporale delle altre applicazioni presenti nel sistema.

3.2 Descrizione dell'architettura

L'architettura di AQuoSA è stata progettata tramite una suddivisione in livelli che potesse permettere di ottenere un sistema efficiente, sicuro, compatibile, flessibile e portabile. Un sistema che utilizza AQuoSA, infatti, si propone di essere: (a) efficiente poiché l'overhead introdotto dal framework

risulta trascurabile; (b) sicuro poiché è possibile definire delle quote di utilizzo della CPU per utente o per gruppi, evitando possibili attacchi volontari basati sui meccanismi introdotti; (c) compatibile perché permette alle preesistenti applicazioni non real-time di eseguire senza modifiche; (d) flessibile poiché nuovi algoritmi di controllo e predizione possono essere facilmente inseriti; (e) portabile perché il supporto ad un nuovo kernel può essere introdotto con poche modifiche.

L'obiettivo principale di AQuoSA è, come detto, quello di fornire tecniche di resource reservation della CPU all'interno del kernel Linux. Tale obiettivo viene realizzato tramite diversi componenti dell'architettura di AQuoSA (vedi figura 3.1). Di seguito ciascun componente verrà analizzato e descritto, mettendo in luce le particolari funzionalità che realizza.

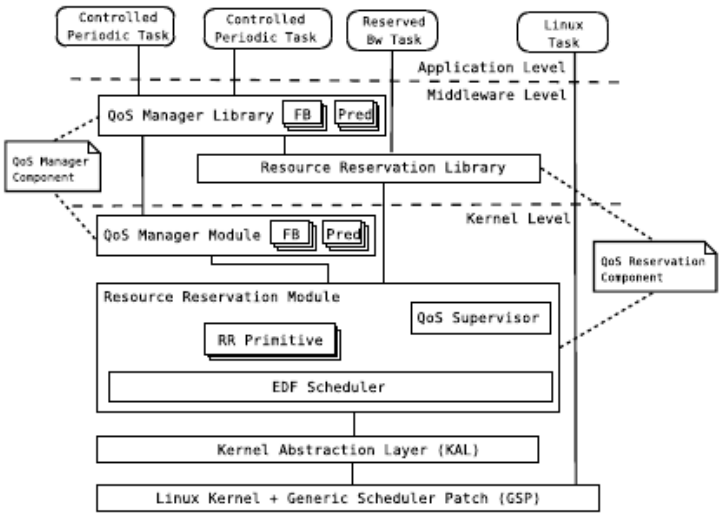

Figura 3.1: Architettura di AQuoSA

3.2.1 Generic Scheduler Patch

Questo componente rappresenta il livello più basso dell'architettura ed è l'unico dipendente dal kernel. E' costituito da una piccola patch del kernel, denominata Generic Scheduler Patch (GSP), che permette di estendere lo scheduler Linux. La GSP infatti intercetta gli eventi di schedulazione ed invoca gli appropriati metodi presenti all'interno di un modulo caricato dinamicamente nel kernel (il Resource Reservation Module). In questo modo vengono forzate le decisioni di schedulazioni del kernel Linux senza sostituire il suo scheduler, che può così continuare a schedulare i processi non real-time, i quali eseguono in background rispetto a quelli soft real-time. Il kernel "patchato" esporta un insieme di puntatori a funzione (chiamati *hook*), che

possono essere impostati da un modulo per implementare una appropriata strategia di schedulazione. Gli eventi principali per i quali sono stati introdotti degli hook sono la creazione e la terminazione di un task, il bloccaggio e il risveglio nell'accesso a risorse condivise, il ricevimento dei segnali SIGSTOP e SIGCONT. Per ognuno di questi eventi i corrispondenti hook sono: `fork_hook`, `cleanup_hook`, `block_hook`, `unblock_hook`, `stop_hook`, `continue_hook`. La GSP patch introduce, inoltre, un campo `private_data` nella struttura `task_struct`: tale campo permette di associare ad ogni task dei dati esterni. Un suo utilizzo può essere fatto, ad esempio, per associare ad ogni task i dati che rappresentano una reservation.

Al di sopra del kernel patchato vi è poi un livello, il Kernel Abstraction Layer (KAL), che permette di astrarre, tramite un insieme di macro scritte in C, tutte le funzionalità aggiuntive del kernel che è necessario utilizzare, come l'abilità di misurare il tempo e di impostare timer.

3.2.2 QoS Reservation Component

Questo componente è costituito da un modulo del kernel e da una libreria di metodi che comunicano fra di loro tramite un Linux *virtual device*.

Il modulo del kernel è il "Resource Reservation Module": tale modulo realizza l'algoritmo di Resource Reservation. Il modulo permette di associare una reservation ad uno o più task: in questo modo ad esempio si possono fornire garanzie temporali a tutti i thread di un'applicazione con una sola reservation. I task del kernel inoltre hanno per default una reservation a loro associata, per evitare che i task real-time possano far attendere indefinitamente i servizi del sistema. Il modulo contiene inoltre il QoS supervisor, che opera (al momento di creazione di un task e durante l'esecuzione del task stesso) per accettare, ritardare, rimodellare o respingere le richieste di banda fatte da un'applicazione. Tali scelte dipendono dal carico a cui è sottoposto il sistema e permettono, ad esempio in caso di sovraccarico, di soddisfare le richieste in base a livelli di priorità.

La libreria di metodi è invece la "Resource Reservation Library": essa permette di esportare le funzionalità del modulo alle applicazioni utenti, che ne possono usufruire tramite l'utilizzo delle appropriate API.

3.2.3 QoS Manager Component

Questo componente è responsabile di fornire un insieme di tecniche di controllo e predizione che possono essere utilizzate da un task per poter adattare dinamicamente i suoi requisiti di banda. Il QoS Manager è composto da un modulo del kernel, una libreria di metodi ed un insieme di sottocomponenti di predizione e retroazione che possono essere indifferentemente posti all'interno del modulo o della libreria. Le applicazioni interagiscono unica-

mente con la libreria, che, a seconda della configurazione, redirige le proprie chiamate al codice in user-space o a quello in kernel-space.

La principale differenza tra le due configurazioni riguarda il numero di volte che l'applicazione commuta da user-space a kernel-space: in una implementazione user-space sono necessari due commutazioni, mentre con una implementazione kernel-space ne è sufficiente una. La possibilità di compilare un algoritmo di controllo in kernel-space permette dunque di ridurre riduce l'overhead dell'infrastruttura di gestione della QoS.

3.3 QoS Reservation Component

In questa sezione verrà analizzato con particolare dettaglio il QoS Reservation Component, cui si è fatto cenno in fase di descrizione dell'architettura. Si concentrerà l'attenzione su tale componente di AQuoSA poiché è quello che permette di realizzare ed utilizzare le resource reservation. In particolare, verrà prima descritto l'algoritmo di schedulazione, come implementato all'interno del Resource Reservation Module ed in seguito verrà fatta una panoramica sulla Resource Reservation Library, che permette alle applicazioni utente di sfruttare le funzionalità fornite dal corrispondente modulo del kernel.

3.3.1 L'algoritmo di schedulazione

In questa sezione verrà trattato il problema dell'implementazione degli algoritmi di schedulazione Reservation Based, con particolare riferimento all'implementazione effettuata all'interno del Resource Reservation Module di AQuoSA. I concetti di base degli algoritmi di schedulazione RB sono già stati affrontati nella sezione 3.1, in cui si è evidenziata soprattutto la loro importanza nel fornire isolamento temporale.

Gli algoritmi di schedulazione Reservation Based prevedono generalmente, nella loro implementazione, l'uso di una entità RSV_i con cui rappresentare una reservation per il generico task τ_i. Ogni reservation è caratterizzata dai parametri Q_i e P_i, che indicano che il task associato eseguirà per un tempo Q_i all'interno del periodo P_i. Ad ogni reservation viene poi associato un budget q_i. Il budget q_i viene decrementato quando τ_i esegue e quando arriva a 0 si dice che la reservation è stata svuotata. All'inizio del successivo periodo di reservation, q_i viene riempito fino a Q_i e la reservation RSV_i diventa eleggibile. Se RSV_i è vuota il task τ_i non può essere schedulato, se invece è nello stato eleggibile allora τ_i può essere schedulato in accordo ad un criterio di assegnamento della priorità tipico dei sistemi real-time, come RM (Rate Monotonic) o EDF (Earliest Deadline First).

Una relazione di consistenza necessaria che uno scheduler RB deve veri-

ficare è appunto la seguente

$$\sum_i B_i \leq U^{lub}$$

con $U^{lub} \leq 1$ e dipendente dal particolare criterio di assegnamento di priorità usato (nel caso si utilizzi EDF si ha $U^{lub} = 1$).

Il concetto di reservation è particolarmente efficace ed utile, tuttavia come osservato in [3], alcune sue implementazioni non sono capaci di fornire piena protezione temporale in determinati casi, particolarmente quelli legati ad attivazioni aperiodiche dei task.

Tenendo conto di ciò, in AQuoSA è stato utilizzato un algoritmo di schedulazione basato su CBS (Constant Bandwith Server), che riesce ad affrontare correttamente questo tipo di anomalie di schedulazione e risulta inoltre particolarmente adatto per la schedulazione RB. Dalla seguente breve descrizione delle regole dell'algoritmo CBS si potranno notare immediatamente, infatti, delle analogie con le regole generali di una schedulazione RB.

- Vi sono diversi server S_i caratterizzati da una coppia (Q_i, T_i^s), in cui Q_i è il massimo budget e T_i^s è il periodo del server. Il rapporto $B_i = Q_i/T_i^s$ è invece la banda del server. Il server mantiene due variabili interne, il budget corrente q_i e la deadline dinamica d_i^s.

- Se un task τ_i è servito da S_i, allora a ciascun job di τ_i sarà associata una deadline uguale a quella del server d_i^s

- Il job con la deadline più vicina viene selezionato per essere eseguito, in accordo ad una assegnazione di priorità basata su EDF.

- Ogni volta che un job esegue, il budget q_i del suo server viene decrementato di una quantità pari al tempo di esecuzione del job.

- Quando $q_i = 0$ il budget corrente viene ricaricato al valore massimo Q_i e la deadline del server viene spostata di un valore pari a T_i^s.

- Quando arriva, ad un certo istante r, un nuovo job del task τ_i, se $q_i \geq (d_i^s - r)\frac{Q_i}{T_i^s}$ allora una nuova deadline $d_i^s = r + T_i^s$ viene generata e q_i viene ricaricata al suo valore massimo; altrimenti il job viene servito con i parametri correnti del server (deadline e budget con valori pari a quelli dell'arrivo del job).

Il CBS riesce a far fronte alle anomalie di schedulazione che potrebbero portare ad una violazione dell'isolamento temporale: questo perché la deadline di un job di un task viene di volta in volta decisa in base al tempo di arrivo del job e alla capacità attuale del server. In tal modo, ai task

con un cattivo comportamento viene abbassata la priorità (posponendo la loro deadline) e si permette così ai task serviti da altri server di eseguire correttamente.

3.3.2 La libreria utente

La "Resource Reservation Library" permette di utilizzare le funzionalità del modulo di Resource Reservation all'interno delle applicazioni utente. Si è visto come, utilizzando tale modulo, si possa riservare ad un task una CPU virtuale dedicata, che abbia velocità pari ad una frazione di quella della CPU reale. La libreria dunque consente principalmente di effettuare le seguenti operazioni:

- creare una nuova risorsa virtuale assegnandogli una certa banda;
- cambiare e recuperare i parametri della risorsa dopo la sua creazione;
- spostare l'esecuzione di un task tra le varie risorse virtuali.

Per quanto detto in 3.3.1, ad un task viene riservata una risorsa virtuale semplicemente "attaccandolo" ad un server CBS, che verrà utilizzato per servire le richieste del task. Fra i parametri associati ad un server, saranno di particolare importanza il budget ed il periodo, dato che il loro rapporto determina la banda della CPU virtuale dedicata al task servito.

In particolare, i parametri associati a ciascun server sono identificati, all'interno della libreria, con il tipo di dato `qres_params_t`, che viene definito nel seguente modo:

```
typedef struct qres_params_t {
    qres_time_t Q_min;
    qres_time_t Q;
    qres_time_t P;
    unsigned int flags;
    qres_time_t timeout;
} qres_params_t;
```

Listing 3.1: Parametri associati ad un server

I campi della struttura identificano i parametri di ciascun server. Il significato di tali parametri viene spiegato di seguito:

- `Q_min`: è il budget garantito al server;
- `Q`: è il budget richiesto dal server;
- `P`: è il periodo del server;
- `flags`: è una combinazione di flag con i quali possono essere impostati determinati comportamenti del server;
- `timeout`: è il tempo massimo di auto distruzione del server.

E' da notare che tutti i tempi vengono espressi in microsecondi.

Come anticipato, i parametri del server vengono specificati al momento della sua creazione, operazione che è resa disponibile da un'apposito metodo. Una breve panoramica di questa funzione di libreria e di tutte le altre significative verrà data di seguito. Per una trattazione dettagliata si rimanda invece alla documentazione di riferimento.

- *qres_init* - Permette di inizializzare la libreria, verificando preventivamente che il kernel supporti la gestione della QoS (ciò implica la presenza del modulo di Resource Reservation all'interno del kernel);
- *qres_cleanup* - Effettua la pulizia delle risorse associate alla libreria, per cui dopo l'invocazione di questo metodo risulta impossibile effettuare altre invocazioni dei metodi della libreria;
- *qres_create_server* - Permette la creazione di un server con i parametri che vengono specificati (ad ogni server del sistema viene associato un identificatore univoco);
- *qres_attach_thread* - Permette di associare un task (sia esso processo o thread) ad un server che è già stato creato;
- *qres_detach_thread* - Permette di "staccare" un task da un server (in base ai parametri del server, questo può significare anche l'eliminazione del server se nessun altro task vi è attaccato);
- *qres_destroy_server* - Permette di distruggere un server il cui identificatore viene passato come parametro (se il server ha dei task ad esso attaccati essi vengono preventivamente staccati);
- *qres_get_sid* - Permette di conoscere l'identificatore del server a cui è attaccato il task specificato come parametro (metodo utile soprattutto per verificare se un task è effettivamente attaccato ad un server);
- *qres_get_bandwith* - Permette di conoscere la banda del server specificato come parametro;
- *qres_get_params* - Permette di recuperare i parametri associati al server specificato;
- *qres_set_bandwith* - Permette di modificare la banda del server specificato come parametro;
- *qres_set_params* - Permette di modificare i parametri del server specificato.

Un semplice esempio di come utilizzare tali funzioni di libreria può essere rappresentato dal listato 3.2.

```
qres_params_t params = {
    .Q_min = 0,
    .Q = 50000L,
    .P = 100000L,
    .flags = 0,
};
qres_sid_t sid;

qres_init();
qres_create_server(&params, &sid));
qres_attach_thread(sid, 0, 0);

/* elaborazioni dell'applicazione */

qres_destroy_server(sid));
qres_cleanup();
```

Listing 3.2: Semplice esempio di utilizzo della libreria RR

In tale semplice esempio, dove fra l'altro non si verifica la corretta esecuzione dei metodi tramite il loro valore di ritorno, si mostra come un task possa creare una risorsa virtuale con banda pari al 50% ed utilizzarla nel corso della sua esecuzione. Si fa notare infine come, nel caso specifico, il task attacca se stesso al server appena creato, passando il valore 0 come secondo e terzo parametro della *qres_attach_thread* (in generale tali parametri rappresentano rispettivamente il process id e il thread id del task da attaccare).

Capitolo 4

Apache2

Il web server Apache è un progetto sviluppato dalla Apache Software Foundation, il cui scopo è quello di fornire un server HTTP che sia sicuro, efficiente ed estensibile. Il progetto è attivo da lungo tempo: all'anno 1995, infatti, risale la prima versione del server. Nel 2000 è stata poi introdotta la versione 2.0 del server, che presenta una completa revisione architetturale; mentre nel 2007 è stata rilasciata la versione 2.2 di Apache, ultima stabile. Le versioni della famiglia 2.x hanno tutte in comune lo stesso progetto architetturale, dunque ci si riferisce ad esse con il termine Apache2. Nel corso di questo lavoro si farà riferimento unicamente ad Apache2, anche quando non espressamente indicato per brevità.

In questo capitolo si affronterà innanzitutto la struttura concorrente di Apache, vero cuore pulsante del server; successivamente si analizzerà in maniera completa il funzionamento interno ed infine si presenteranno alcune fra le caratteristiche tecniche di maggior rilievo utilizzate per lo sviluppo del server.

4.1 Struttura concorrente

Apache è un web server concorrente, capace cioè di gestire contemporaneamente richieste provenienti da più client: tali tipi di server vengono generalmente identificati come *multitasking web server*, poiché la concorrenza viene gestita internamente dal server tramite un'architettura che prevede l'uso di più task.

4.1.1 Considerazioni sulle architetture concorrenti

In generale, il progetto di un'architettura concorrente (o architettura multitasking) di un qualsiasi server deve tenere in considerazione due aspetti principali: il sistema operativo sottostante e l'uso a cui è destinato il server.

In primo luogo analizziamo perché è importante conoscere il sistema operativo su cui si vuole far girare il server. Tipicamente i sistemi operativi si distinguono per diverse scelte strategiche, come il modo in cui viene implementato un task (ad esempio con processi o con thread), il modo in cui avviene la gestione della comunicazione tra i task stessi, il modo in cui viene gestita la concorrenza. La conoscenza delle particolari caratteristiche di un sistema operativo, dunque, permette di sfruttarne appieno le potenzialità, sia in fase di progetto che in fase di sviluppo del server, con conseguente miglioramento in termini di prestazioni.

In secondo luogo, si deve considerare quale sia il comportamento che si vuole abbia il server: infatti a seconda del numero di operazioni che si deve gestire o di quale dipendenza ci sia tra due richieste, utilizzare un certo tipo di architettura può essere o meno conveniente. Per chiarire questo concetto si può analizzare l'architettura multitasking del server *inetd*: in essa vi è un solo task, il cosidetto *master server*, che ha il compito di attendere le richieste; nel momento in cui una di essa arriva, il master server crea un'altro processo che gestisce la richiesta e poi termina. Tale architettura sarebbe certamente adatta per un server in cui si ricevono poche richieste e ciascuna di esse ha una certa durata o si deve avvalere di informazioni di stato; risulterebbe invece inefficiente in un web server, visto che il processo master dovrebbe creare un nuovo task per ogni connessione HTTP, senza potere nel frattempo accettare richieste in ingresso.

4.1.2 Gestione della concorrenza

L'architettura di Apache prende in considerazione entrambi gli aspetti visti nella sezione precedente.

Se ci si riferisce al primo aspetto, notiamo infatti che in Apache sono presenti diverse architetture multitasking, il cui scopo principale è quello di supportare i diversi sistemi operativi su cui Apache può essere installato: Unix, Windows, Netware, BeOS, OS/2. Ciascuna di queste architetture utilizza le API native dei vari sistemi operativi, per poterne sfruttare caratteristiche e funzionalità.

Se invece si prende in esame il secondo aspetto, si può notare come l'architettura sia stata progettata appositamente per non incorrere in inefficienze tipiche, come quelle cui si è fatto riferimento nel caso del server inetd (vedi 4.1.1). Tutte le architetture multitasking di Apache si basano infatti su un *task pool*: esiste sempre in esecuzione un certo numero di task che si occupano di ricevere le richieste. Quando arriva una richiesta, dunque, non viene creato alcun task per il suo servizio: la richiesta sarà immediatamente servita se in quel momento almeno uno dei task del pool è inattivo, altrimenti il servizio avverrà non appena uno dei task termina il processamento di una richiesta precedente. I task del pool vengono creati in fase di inizializzazione del server web e durante la loro esecuzione vengono costantemente control-

lati da un altro componente, il *master server*. Il master server, dunque, non si occupa di ricevere le richieste, ha invece il compito di controllare e gestire i task del pool in accordo ai parametri con cui è configurato il web server: il numero di task creati all'avvio o il massimo numero di task inattivi, sono, ad esempio, fra i parametri che è possibile configurare.

Si è detto che per ognuna delle piattaforme supportate da Apache esiste almeno una corrispondente architettura multitasking: ognuna di esse viene incapsulata in uno dei cosidetti **Multi-Processing Modules** (MPM), tramite cui si può scegliere facilmente quale architettura utilizzare per il proprio server. Tale scelta va fatta in fase di compilazione, tenendo conto che può essere presente nel server un solo MPM per volta. Gli MPM sono intercambiabili, per cui il server ha le stesse funzionalità qualunque sia l'architettura usata, ciò che cambia è il modo in cui tali funzionalità vengono realizzate.

Possiamo concludere dicendo che gli MPM, una particolarità di Apache2, risultano una notevole miglioria a livello progettuale e la loro introduzione permette di sfruttare, in maniera semplice e veloce, quelli che abbiamo visto essere i vantaggi derivanti dall'utilizzare diverse architetture multitasking in diversi contesti: ovvero una maggiore efficienza nelll'utilizzo dei costrutti nativi del sistema operativo e una maggiore adattabilità nei diversi contesti applicativi.

4.1.3 Architetture concorrenti su Unix

Le architetture concorrenti di Apache per sistemi operativi Unix sono essenzialmente due: esse corrispondono ai moduli MPM Preforking e Working. Il modulo Preforking presenta la stessa architettura multitasking delle primissime versioni di Apache mentre l'architettura del modulo Working è stata una delle novità introdotte con la versione 2 del server. Nel seguito si descriverà in dettaglio le due architetture, analizzandone la struttura ed individuandone gli appropriati contesti d'uso.

Preforking MPM

Il Preforking MPM costituisce probabilmente la più importante architettura multitasking presente in Apache 2. Deve la sua importanza non solo a ragioni puramente storiche ma anche al fatto che è tutt'ora l'architettura predefinita per le versioni Unix del server, visto che è comunque quella che garantisce maggiore stabilità. Infatti, in tale architettura, i task del pool destinati a gestire le richieste sono dei **processi** e il fatto che essi abbiano spazi di indirizzamento separati garantisce che, in caso di eventuali problemi, solamente il processo che li ha subiti vada in crash e non l'intero server.

I processi che fanno parte del pool sono creati dal processo padre, il master server, *prima* che venga ricevuta una qualsiasi richiesta. Questa

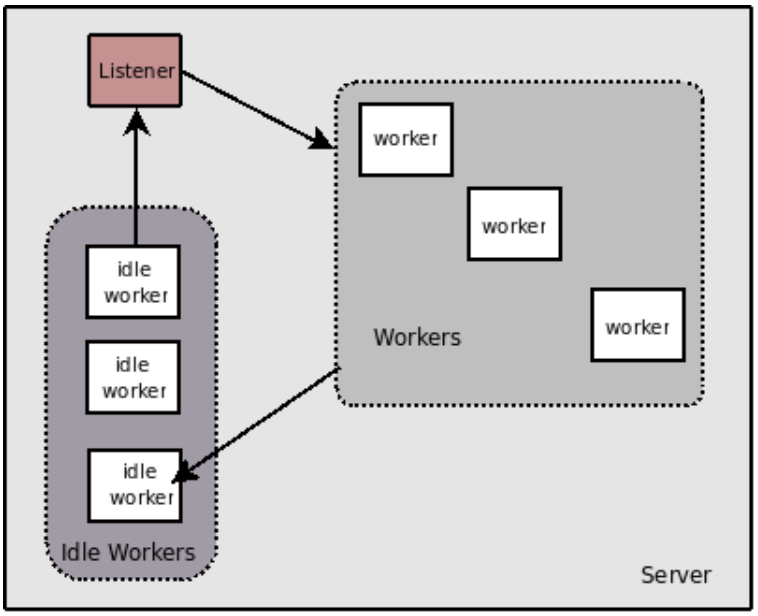

Figura 4.1: Ruoli di un processo figlio nell'architettura preforking

caratteristica risulta talmente distintiva da poter spiegare, unitamente alla considerazione che i processi figlio sono creati con la chiamata di sistema `fork()`, il perché l'architettura sia stata denominata *pre*forking.

In accordo a quanto detto sugli aspetti comuni dei vari MPM, solamente i processi figlio si occupano di ricevere delle richieste. Ciascuno di essi, in questa architettura, può trovarsi di volta in volta a ricoprire uno di questi tre ruoli (vedi figura 4.1):

- `listener`
- `worker`
- `idle worker`

In ogni istante, all'interno del task pool, vi può essere un solo processo `listener`, il quale attende di stabilire una connessione con un client che effettua una richiesta.

Non appena arriva la richiesta, il listener cambia il suo ruolo in `worker`, ovvero si occupa di gestire la richiesta appena ricevuta.

Quando poi il worker ha terminato di processare la sua richiesta, esso chiude la connessione instaurata con il client e diventa un `idle worker`, ovvero si inserisce nella coda dei processi che aspettano di diventare listener. Infatti, un listener che diventa worker viene sostituito dal primo processo presente nella coda degli idle worker. Tale paradigma di comportamento può facilmente essere visto come un'applicazione del cosidetto *leader-followers pattern*: il listener è il leader, gli idle worker sono i followers.

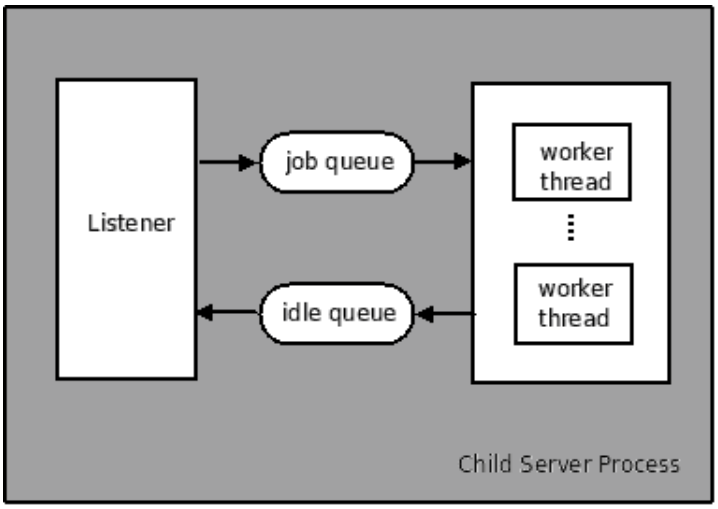

Figura 4.2: Organizzazione dei thread in un processo figlio nell'architettura working

Si ribadisce, infine, che il master server non si occupa di attendere l'arrivo di richieste in ingresso, bensì ha il compito di aggiustare il numero di processi figlio in accordo ai parametri forniti.

Worker MPM

Il Worker MPM presenta una struttura concorrente particolarmente interessante, poiché prevede l'utilizzo di un'**architettura ibrida**, costituita da un numero variabile di processi che includono, al proprio interno, un numero fisso di thread. Con l'uso dei thread si riesce ad avere, rispetto ad un'architettura multiprocesso, un server che risponde ad un maggior numero di richieste con minor uso di risorse: questo MPM è dunque ideale per essere impiegato in server che hanno come principale requisito una grande scalabilità, anche a scapito di una seppur minima perdita di stabilità. C'è da sottolineare infatti che il modello utilizzato nel Worker MPM cerca di ottenere un compromesso tra stabilità e prestazioni. Infatti con un modello multithread puro si avrebbero le maggiori prestazioni ma pensiamo a cosa avverrebbe se un thread andasse in crash: l'intero server sarebbe compromesso, a differenza del modello ibrido dove solamente il processo contenente quel thread sarebbe affetto dal problema.

Analizziamo ora il funzionamento del modulo Worker, la cui architettura è dunque sia multiprocesso che multithread: a livello di processo essa è organizzata in maniera analoga al Preforking MPM, mentre a livello di thread utilizza il cosidetto modello *job queue*.

Tale modello verrà ora descritto, con riferimento alla figura 4.2. Ogni processo figlio include un unico thread *listener* e un insieme di thread *wor-*

ker. Il listener ha come unico compito quello di attendere una connessione sulla porta di ascolto del server. I worker invece hanno il compito di servire le richieste e comunicano con il listener tramite due code, la coda di lavoro (*job queue*) e quella di attesa (*idle queue*). Se vi è almeno un worker in attesa, condizione indicata dalla presenza di almeno un token nella idle queue, il listener aspetta di ricevere una connessione e, non appena la riceve, mette nella job queue un item: in questo modo vi è la certezza che la richiesta in ingresso verrà servita immediatamente da un worker. Dopo aver completato l'esecuzione della richiesta, il thread worker si registrerà poi come idle (inserendo un token nella idle queue) e aspetterà un nuovo item nella job queue.

4.2 Funzionamento interno

In questa sezione si analizzerà il funzionamento interno di Apache2. Inizialmente verrà analizzato il comportamento generale del server, descrivendone le varie fasi di esecuzione. In seguito, verrà evidenziato quali sono le fasi in cui il funzionamento interno di Apache2 dipende dall'architettura concorrente che si utilizza. Infine, si descriverà in dettaglio il funzionamento del server, prendendo come riferimento l'architettura concorrente di tipo preforking.

4.2.1 Comportamento generale

Il comportamento generale di Apache può essere analizzato indipendentemente dalla sottostante architettura concorrente, poiché consiste nell'eseguire in successione una sequenza di fasi. Quello che contraddistingue le varie architetture sarà poi il modo in cui alcune di queste fasi vengono eseguite. Possono essere identificate, in accordo alla nomenclatura utilizzata in [5], le seguenti fasi di esecuzione (vedi figura 4.3).

- **First-time initialization** - Fase di inizializzazione eseguita immediatamente all'avvio;
- **Restart loop** - Ciclo principale del server, eseguito ad ogni riavvio;
- **Master server loop** - Ciclo contenuto nel restart loop, in cui il master server controlla i task del pool;
- **Request-response loop** - Ciclo principale eseguito dai task del pool;
- **Keep-alive loop** - Ciclo contenuto nel request-response loop, in cui i task del pool processano le richieste HTTP
- **Clean-up** - Fase di pulizia delle risorse, eseguita sia dal server principale che dai task del pool.

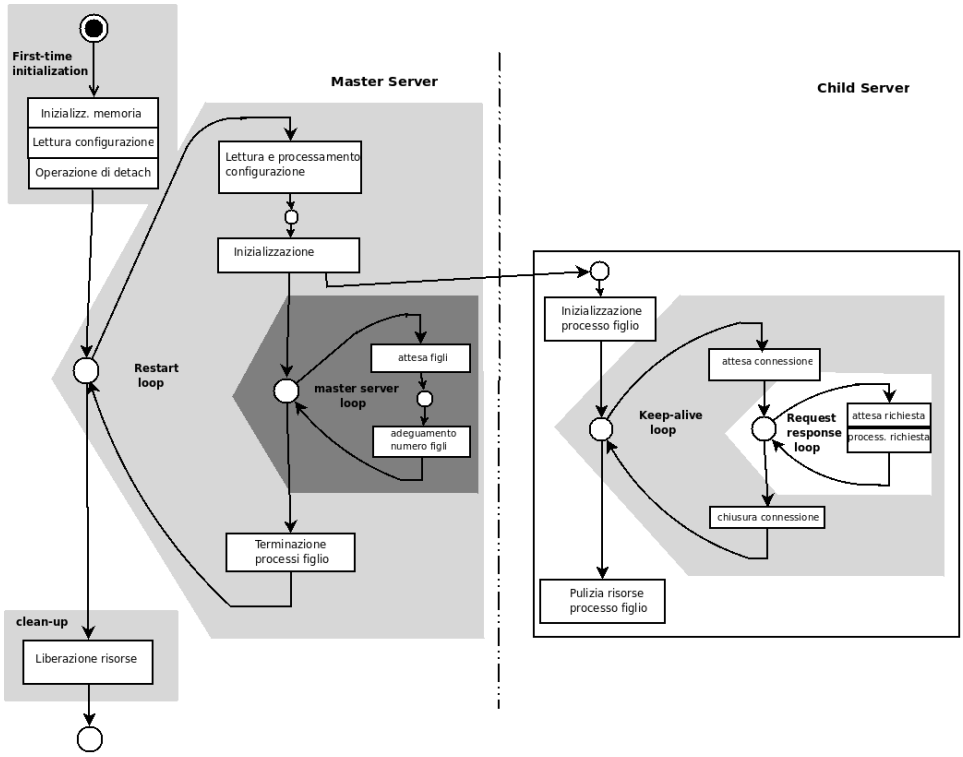

Figura 4.3: Funzionamento interno di Apache

4.2.2 Ambiti specifici

Le fasi di esecuzione in cui un'architettura concorrente si distingue da un'altra possono essere facilmente individuate se si tiene conto delle principali differenze tra i vari moduli MPM. Essi, per quanto detto in 4.1.2, si differenziano dal punto di vista interno per tre aspetti principali:

- per il modo in cui vengono creati e gestiti i processi figlio che rispondono alle richieste;
- per il modo in cui vengono accettate le richieste;
- per il modo in cui viene effettuato il collegamento (binding) con le porte su cui il server ascolta.

La creazione e la gestione dei task del pool avvengono ad opera del master server, rispettivamente nelle fasi di restart loop e di master server loop.

Gli aspetti relativi all'accettazione delle richieste ed il modo in cui esse vengono ascoltate, riguardano invece i vari task del pool, per cui avvengono nella fase di request-response loop.

Le operazioni eseguite nelle tre fasi di restart loop, master server loop e request-response loop risultano dunque influenzate dall'architettura multitasking inclusa in quel momento nel server (si veda la figura 4.4, in cui gli ambiti specifici ad un MPM sono racchiusi in rosso).

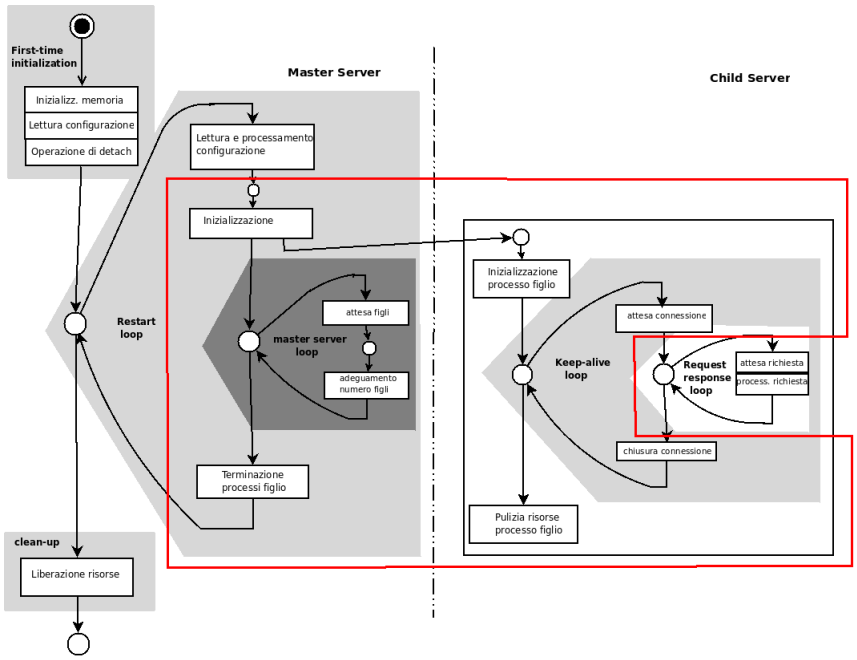

Figura 4.4: Ambito di un MPM

Dalla figura si evidenzia inoltre che le altre fasi prevedono invece l'esecuzione delle stesse operazioni qualunque sia l'architettura sottostante.

4.2.3 Descrizione dettagliata

Una descrizione dettagliata delle fasi di esecuzione individuate può essere effettuata prendendo come riferimento l'architettura preforking (vedi 4.1.3). Ovviamente la descrizione delle fasi indipendenti dall'architettura concorrente rimane valida anche nel caso di altre architetture.

First-time initialization

In questa prima fase il server inizializza innanzitutto le proprie aree di memoria. In seguito, poiché il server può essere esteso tramite l'uso dei moduli, si registrano le informazioni riguardanti i vari moduli inclusi staticamente. Da notare che Apache2 è un'architettura fortemente modulare, per cui molti dei componenti fondamentali (vedi gli MPM) sono costituiti da moduli. In questa fase vengono poi valutati eventuali parametri trasmessi tramite riga di comando, vengono letti i file di configurazione ed infine viene effettua la cosiddetta operazione di *detach*.

L'eseguibile di Apache si trova infatti nel file binario *httpd*, che generalmente viene lanciato tramite shell con il comando `apachectl start`. In questo caso, il processo padre di Apache è proprio la shell, per cui il server deve staccarsi da essa per poter continuare la sua esecuzione anche dopo che il processo shell viene terminato. E' da notare che, dopo che il server ha eseguito la detach, nessun messaggio di errore verrà più stampato nella shell, per cui a scopo di debug potrebbe risultare utile evitare che il server si stacchi: questo comportamento può essere ottenuto avviando il server con l'opzione `-X`. Dopo la fase di detach ha senso identificare il processo server principale con il master server.

Restart loop

Tale ciclo viene così chiamato perché Apache vi entra non solo dopo la prima fase di inizializzazione ma anche ad ogni riavvio. In questa fase vengono per prima cosa riletti i file di configurazione, per fare in modo che il server sia aggiornato rispetto ad un loro eventuale cambiamento (il server deve dunque essere riavviato ad ogni modifica della configurazione). In seguito vengono inizializzati i socket di rete per ascoltare le richieste e poi vengono creati i task del pool.

Nel caso dell'architettura preforking si parla di processi figlio, poiché i task del pool vengono creati utilizzando la chiamata di sistema `fork()`. Come conseguenza, tutti i processi figlio hanno una copia esatta delle risorse del padre (il master server), dunque essi possiedono anche i dati di configu-

razione e quelli per l'accesso ai socket. Dopo la creazione dei figli, il master server entra nella fase successiva, quella del master server loop.

Master server loop

In questa fase il master server effettua essenzialmente due operazioni. La prima di queste consiste nel controllare il numero di figli inattivi (rappresentato dalla variabile `idle_count`) verificando che essi siano all'interno dei limiti forniti tramite configurazione. Questi limiti sono espressi dalle variabili `ap_daemons_max_free` e `ap_daemons_min_free`. Se il numero di figli inattivi è minore di `ap_daemons_min_free` verranno creati nuovi figli, tanti quanti ne servono. Se invece tale numero è maggiore di `ap_daemons_max_free` verrà ucciso un solo processo figlio, quello registrato inattivo come ultimo. L'uccisione dei processi sovrannumerari viene effettuata dal master tramite l'uso della cosiddetta *Pipe of Death* (PoD), una pipe condivisa da tutti i figli appartenenti alla stessa generazione.

Il secondo compito che il master server deve svolgere è relativo alla funzionalità cosiddetta di *graceful restart*: in caso di riavvio del server, questa funzione permette di non uccidere immediatamente i processi figlio che stanno servendo delle richieste, permettendo loro di espletarle. Al contrario i figli inattivi devono essere subito uccisi, operazione di cui si occupa appunto il master server, utilizzando la PoD. E' da notare infine che i figli che stanno gestendo delle richieste provvederanno da soli ad "uccidersi": una volta terminata la gestione della richiesta in corso durante il riavvio, essi infatti verificano di non appartenere alla generazione precedente, in caso contrario terminano la loro esecuzione.

Request-response loop

Tale fase viene eseguita solamente dai processi figlio: essi, dopo una necessaria inizializzazione, entrano in questo ciclo dove aspettano di divenire worker per poter entrare poi nella fase successiva, dove la richiesta in ingresso viene effettivamente servita. Per poter iniziare a servire una richiesta, essa deve prima essere accettata. Nell'architettura preforking ciascun processo figlio si preoccupa di ricevere la richiesta che andrà poi a servire, dunque prima di divenire worker deve passare dallo stato di listener, in cui ascolta sulla porta del server per instaurare la connessione TCP su cui verranno instradate le richieste HTTP.

Per poter distribuire le connessioni TCP in ingresso, viene utilizzato un semaforo di mutua esclusione, denominato *accept mutex*, che ogni processo deve acquisire prima di poter effettuare la chiamata di sistema `accept()` (con cui si stabilisce una connessione TCP). Le procedure a protezione della mutua esclusione sono la `accept_mutex_on()` per l'acquisizione e la `accept_mutex_off()` per il rilascio. L'uso del mutex garantisce che vi sia in

ogni istante un solo processo listener: questo permette, ad esempio, nel caso in cui il server web sia in ascolto su più porte, di evitare che tutti i processi figlio siano bloccati sulla stessa porta.

Dopo che viene creata la connessione con il client che ha effettuato la richiesta, il processo figlio entra nel keep-alive loop, dove serve la richiesta. Al termine di questo ciclo interno viene chiusa la connessione precedentemente aperta e si ritorna all'inizio del ciclo request-response.

Infine notiamo che, nel caso di architettura ibrida (vedi 4.1.3), i child server, per poter divenire worker devono solamente attendere un item nella coda di lavoro: questo poiché vi è un unico thread listener in ascolto.

Keep-alive loop

Il keep-alive loop riguarda il processamento vero e proprio di una richiesta HTTP e viene eseguito dai serventi figli (child server) dopo l'instaurazione della connessione TCP avvenuta nel request-response loop. Questo ciclo viene ripetuto n volte, dove n è il numero di richieste HTTP instradate su una stessa connessione TCP.

Il nome di questa fase deriva, appunto, da uno dei valori che può assumere il campo `Connection` dello header HTTP. Se tale header contiene `Connection: KeepAlive` allora verrà riutilizzata la stessa connessione TCP per più richieste HTTP fatte allo stesso Host (per ogni risorsa presente in una pagina web è necessario infatti effettuare una distinta richiesta). Da notare che tale comportamento è quello di default per la versione 1.1 del protocollo.

Il processo figlio uscirà da tale ciclo se al termine di tutte le operazioni di processamento di una singola richiesta sarà verificata una delle seguenti condizioni:

- la connessione è stata persa;
- la connessione è andata in time-out;
- la connessione è non persistente (ovvero valore KeepAlive assente);
- c'è stata una segnalazione di graceful restart.

Il processamento di una richiesta HTTP, il cui codice è contenuto nel file `request.c`, consiste, molto sinteticamente, nell'associare ad un URI (Uniform Resource Identificator) una risorsa locale, autenticare e autorizzare il client, determinare il tipo MIME della risorsa e, conseguentemente, selezionare il gestore (content handler) che deve generare la risposta. Per una descrizione maggiormente dettagliata della fase di processamento di una richiesta si rimanda a [5].

Clean-up

La fase di pulizia viene eseguita prima della terminazione di ciascun task all'interno di Apache, dunque sia dal master server che dai child server. Essa consiste principalmente nel liberare le risorse occupate all'interno del sistema, ovvero la memoria, le porte su cui il server ascolta e i file gestiti.

4.3 Caratteristiche tecniche

In questa sezione verranno discusse due delle caratteristiche tecniche di maggior rilievo presenti in Apache2, la cui trattazione risulta utile per comprendere a fondo come certe funzionalità vengono realizzate all'interno del server. Questa comprensione risulta di particolare importanza soprattutto nel momento in cui si volessero estendere tali funzionalità. Inizialmente verrà descritto il modo in cui viene gestita la configurazione, in seguito verrà analizzato il metodo di gestione delle risorse (file, memoria, ecc.) utilizzate nel corso del funzionamento del server.

4.3.1 Gestione della configurazione

Il web server Apache utilizza come metodo principale di configurazione quello basato su file di testo. All'interno di un file di configurazione sono contenute varie direttive, ciascuna posizionata su una diversa linea. Ogni direttiva modifica il comportamento di un determinato aspetto del server tramite l'uso di un certo numero di parametri (tale numero, che può essere anche pari a zero, dipende da come la direttiva viene definita).

Un tipico esempio di direttiva è il seguente:

```
Listen 80
```

Se questa linea fosse presente nella configurazione specificherebbe che il server attende di ricevere richieste dalla porta 80.

Nel proseguo verranno analizzati prima i diversi tipi di file di configurazione che è possibile utilizzare, per poi descrivere i vari contesti in cui è possibile inserire le varie direttive. Tutto questo risulta di fondamentale importanza per capire come Apache gestisce internamente la configurazione.

File di configurazione

Il file di configurazione più importante è sicuramente il file **httpd.conf**, tramite il quale si possono modificare le più importanti caratteristiche del server. Tale file non è però l'unico che è possibile utilizzare. Infatti, i file di configurazione tramite i quali personalizzare Apache possono essere suddivisi in due diversi **tipi**, di cui daremo ora descrizione.

file di configurazione globale Con essi si intendono il file *httpd.conf* e tutti i file che, tramite esso è possibile caricare con la direttiva `Include`. Tali file contengono direttive che si applicano all'intero processo Apache, al server principale e ai vari virtual host. I virtual host (si veda [7]) possono essere considerati dei server "virtuali", dato che permettono di avere differenti siti web, differenziati per nome o per indirizzo ip, sulla stessa macchina fisica, il tutto senza che questo sia visibile all'utente finale. Per utilizzare delle impostazioni specifiche per un virtual host è sufficiente posizionare le appropriate direttive all'interno di una sezione `<VirtualHost>`.

I file di configurazione globale possono contenere anche delle impostazioni che si applicano solamente a parti del server, in questo caso però esse devono essere inserite all'interno delle apposite sezioni `<Directory>`, `<File>` o `<Location>`, che si riferiscono rispettivamente a particolari directory, file o indirizzi.

E' da notare che, i cambiamenti ai file di configurazione globale non hanno effetto immediato se vengono effettuati durante l'esecuzione del server: affinchè Apache li riconosca è necessario che venga riavviato.

file di configurazione locale Con essi si intendono invece i file *.htaccess*, i quali permettono ai webmaster di utilizzare configurazioni personalizzate per i loro siti (nei limiti concessi loro dall'amministratore del server). Tali file vengono generalmente posizionati nella cartella radice di un sito web, dato che si applicano anche a tutte le sottocartelle. Essi sono letti per ogni richiesta effettuata a quel sito, i cambiamenti a tali file hanno dunque effetto immediato.

Contesti per le direttive

Ogni direttiva è caratterizzata da un contesto, che indica in quali file e sezioni essa può legalmente essere utilizzata. Se una direttiva si trova in un contesto non proprio, si verificherà un errore del server in fase di lettura della configurazione. In accordo a quanto riportato nella documentazione ufficiale di Apache ([7, directive-dict.html]), si possono identificare quattro diversi contesti:

- *server config*: indica che la direttiva può essere usata in un file globale ma non in sezioni `<VirtualHost>` or `<Directory>`. Inoltre non può essere utilizzata in file .htaccess.
- *virtual host*: se una direttiva è valida in questo contesto essa può apparire solamente nei file di configurazione globali all'interno di una sezione `<VirtualHost>`.
- *directory*: indica che la direttiva può essere usata solamente in sezioni identificate dalle direttive `<Directory>`, `<File>` o `<Location>`.

- *.htaccess*: se una direttiva è valida in questo contesto, può essere usata solamente nei file .htaccess. Può tuttavia non essere processata, in base alle impostazioni di sovrascrittura attivate.

La definizione dei contesti in cui sono permesse le direttive risulta di notevole importanza, perché si riflette all'interno di Apache nell'utilizzo di due diverse strutture dati per gestire e memorizzare la configurazione. Tali strutture dati sono:

- **per-directory config**: è costituita dalle direttive che hanno un contesto di tipo *directory* o *.htaccess*.
- **per-server config**: è costituita da tutte quelle direttive caratterizzate da un contesto di tipo *server config* o *virtual host*.

Tale distinzione va presa particolarmente in considerazione nel caso in cui si vogliano estendere le funzionalità di Apache e si vuole che tali funzionalità possano essere configurate.

4.3.2 Gestione delle risorse

Il meccanismo tramite cui in Apache si gestiscono tutte le risorse utilizzate dal server è quello dei *pool*. Essi costituiscono un componente fondamentale di Apache, poiché quasi tutte le sue risorse interne risiedono nei pool.

Descrizione del meccanismo

Un *pool* è un insieme di risorse di diverso tipo, quali ad esempio file, memoria, programmi figlio, socket o pipe. Le risorse allocate in un pool, sebbene possano essere eterogenee, hanno la caratteristica comune di avere lo stesso tempo di vita: esse infatti saranno rilasciate ogni qual volta un pool viene distrutto.

I pool possono essere visti come un metodo di gestione delle risorse che si pone a metà strada tra un meccanismo ad alto livello come la *garbage collection*, in cui è il linguaggio (Java ne è un esempio) a preoccuparsi di liberare le risorse, ed un meccanismo a basso livello (come quello usato nel C) che prevede un'esplicita liberazione delle risorse. E' da notare che il non avere l'incombenza di dover liberare una risorsa permette di assicurare un corretto comportamento del programma, soprattutto nel caso in cui si devono gestire situazioni di errore, quando prevedere espliciti rilasci per tutti i possibili percorsi di esecuzione risulta un compito non banale. Tutto questo però si ottiene a scapito di un certo overhead e della possibilità di fornire al programmatore delle scelte consapevoli di ottimizzazione.

L'utilizzo dei pool permette di avere un approccio misto: da un lato c'è la possibilità per il programmatore di liberarsi dalla preoccupazione di dover esplicitamente liberare una risorsa allocata e, allo stesso tempo, viene fornita una certa libertà di scelta sul tempo di vita di una risorsa.

Tempo di vita di una risorsa

Il tempo di vita di una risorsa associata ad un pool coincide con il tempo di vita del pool. In Apache sono presenti quattro tipi predefiniti di pool, ciascuno con un diverso tempo di vita, specificato nella descrizione data di seguito.

Request pool Ha il tempo di vita di una richiesta HTTP, viene acceduto tramite il campo `pool` di una variabile di tipo `request_rec` ed è adatto per risorse temporanee usate per processare una singola richiesta.

Connection pool Ha il tempo di vita di una connessione TCP, viene acceduto tramite il campo `pool` di una variabile di tipo `conn_rec` ed è utile soprattutto per risorse temporanee che non possono essere associate con una richiesta, come ad esempio il caso di filtraggio a livello di connessione.

Process pool Ha il tempo di vita di un processo server, viene acceduto tramite il campo `pool` di una variabile di tipo `process_rec` ed è adatto per risorse che devono avere una lunga durata, come quelle che devono essere riutilizzate più richieste.

Configuration pool E' anch'esso associato con un processo ma differisce dal *process pool* perché viene nuovamente inizializzato ogni qual volta Apache legge la configurazione; viene acceduto tramite il campo `pconf` di una variabile di tipo `process_rec`; è adatto per risorse legate alla configurazione, come le strutture che rappresentano la configurazione di un modulo.

Si noti che si è fatto riferimento a dei tipi di dato (`request_rec`, `conn_rec`, `process_rec`) fondamentali in Apache2, poiché rappresentano le informazioni chiave che il server associa ad ogni richiesta, connessione o processo. Per maggiori informazioni su tali tipi (ad esempio quali sono i campi delle strutture con cui sono definiti) si rimanda alla documentazione interna di Apache2, generabile dai sorgenti tramite il comando `make dox`.

Quando vi è la necessità di memorizzare dei dati, tutto quello che deve fare uno sviluppatore è quello di associare tali dati con il pool che ha lo stesso tempo di vita (o il minore fra quelli accettabili) che si vuole abbiano i dati. Nel caso in cui i pool predefiniti non siano poi adeguati, è possibile creare, utilizzando le Apache API, dei pool personalizzati (cosiddetti *sub pool*), derivati direttamente da quelli predefiniti.

Allocazione di una risorsa

Per allocare una risorsa, ad esempio una variabile, all'interno di un pool, si utilizza la funzione *`apr_pcalloc`*, il cui prototipo è

```
void* apr_pcalloc (apr_pool_t* p, apr_size_t size)
```

Tale funzione alloca lo spazio di memoria specificato da `size` nel pool `p` ed inizializza a zero la memoria allocata, che viene restituita come puntatore a `void`. Utilizzando tale funzione, la variabile allocata sarà dunque associata al pool, che provvederà automaticamente e al momento opportuno a liberare lo spazio ad essa associato.

Il meccanismo dei pool può essere utilizzato anche quando non vi è la possibilità di allocare variabili tramite la *apr_pcalloc* perché, ad esempio, si ha la necessità di utilizzare delle librerie di terze parti. E' possibile utilizzare, infatti, in queste situazioni la funzione *apr_pool_cleanup_register*, la quale ha il seguente prototipo:

```
void
apr_pool_cleanup_register(apr_pool_t* p,
                          const void* data,
                          apr_status_t(*)(void*)plain_cleanup,
                          apr_status_t(*)(void*)child_cleanup
                          )
```

Tramite la *apr_pool_cleanup_register*, si può associare ad un pool `p` una funzione di nome *plain_cleanup*, in modo tale che essa venga invocata ogni qual volta il pool debba essere distrutto. In questo modo, si possono associare ad un pool anche dati per i quali vi è la necessità di utilizzare funzioni di allocazione native, visto che le apposite funzioni di "pulizia" saranno invocate al termine della vita del pool. Per completezza vi è inoltre da dire che, relativamente al prototipo della *apr_pool_cleanup_register*, il parametro `data` rappresenta i dati da passare alla funzione da registrare, mentre `child_cleanup` è la funzione che verrà invocata poco prima della terminazione di un processo figlio.

Capitolo 5

Estendere Apache2

Le funzionalità di Apache possono essere facilmente estese o modificate tramite l'utilizzo di **moduli**. I moduli sono dei pezzi di codice sorgente tramite i quali è possibile determinare il comportamento che il server assume nelle diverse fasi della sua esecuzione. Apache2 ha una struttura spiccatamente modulare, molte delle sue funzionalità sono infatti realizzate da moduli. Una istanza eseguibile del server sarà costituita dunque dal nucleo (che contiene solamente le funzionalità di base) e dai vari moduli che vengono inclusi.

Un modulo può essere incluso nel server in maniera statica o in maniera dinamica. Nel primo caso il codice del modulo deve essere aggiunto ai sorgenti del server affinchè esso possa poi essere compilato con l'intera distribuzione (si veda [6]). Nel secondo caso, invece, si ha che il modulo viene caricato come una libreria condivisa durante l'inizializzazione o il riavvio del server: questo compito viene svolto dal modulo `mod_so`, che deve essere incluso staticamente, compilando Apache con l'opzione `--enable-so`. E' da notare che la possibilità di usufruire dell'inclusione dinamica risulta particolarmente vantaggiosa, soprattutto in fase di sviluppo di un modulo: questo perché le verifiche successive ad ogni modifica del modulo possono essere effettuate senza la necessità di dover ricompilare l'intero server.

Nel corso del capitolo verrà prima descritta l'interfaccia fornita da Apache per permettere la comunicazione tra un modulo ed il nucleo ed infine si illusteranno alcune tecniche tipiche da utilizzare nella realizzazione di un modulo.

5.1 Descrizione dell'interfaccia

I moduli interagiscono con il nucleo del server tramite una comune interfaccia. Il nucleo fornisce la possibilità ad un modulo di agganciarsi in varie fasi dell'esecuzione, tramite l'uso degli *hook*: se un modulo definisce un metodo che realizza un hook (un *hook handler*), esso verrà invocato dal nucleo al momento opportuno.

Gli hook non sono l'unico modo per realizzare l'interfacciamento ma di certo costituiscono il meccanismo più importante ed utilizzato, per cui si farà riferimento esclusivamente ad essi. Per altri meccanismi, come *optional function*, *optional hook* o *filter* il lettore interessato può far riferimento a [6].

Nel proseguo verrà chiarito il meccanismo degli hook e verrà fornita una panoramica sui principali hook che possono essere utilizzati dai moduli per inserirsi nelle varie fasi di esecuzione del server.

5.1.1 Il meccanismo degli hook

Un **hook** è un punto in cui il nucleo permette ad un modulo di inserirsi nel flusso di esecuzione. Per cui, un hook specifica semplicemente un prototipo di funzione: tale funzione dovrà essere effettivamente realizzata da un modulo che è interessato ad inserirsi in quel punto.

Per definire un hook, il nucleo di Apache effettua quattro operazioni ad esso relative:

1. dichiarazione;
2. inserimento in una apposita struttura dati;
3. implementazione;
4. chiamata all'implementazione.

Ad esempio, se si vuole definire un hook per un metodo del tipo

```
int myhook(int, char*);
```

allora le operazioni che dovranno essere compiute dal nucleo saranno innanzitutto le seguenti:

```
AP_DECLARE_HOOK(int, myhook, (int,char*))

APR_HOOK_STRUCT(
    APR_HOOK_LINK(myhook)
    APR_HOOK_LINK(other_hook)
)
```

Tali operazioni permettono dunque di dichiarare un hook e di inserirlo nell'apposita struttura che contiene i link a tutti gli hook definiti.

L'implementazione degli hook viene invece effettuata tramite l'uso di alcune macro (AP_IMPLEMENT_HOOK_VOID, AP_IMPLEMENT_HOOK_RUN_ALL, AP_IMPLEMENT_HOOK_RUN_FIRST) che si differenziano perché generano diversi tipi di hook, come sarà chiarito in seguito. In ogni caso, l'utilizzo di una qualsiasi di queste macro determina l'implementazione di una funzione che si chiamerà *ap_run_myhook*, dove `myhook` è appunto il nome dello hook.

L'operazione successiva all'implementazione di uno hook è la sua chiamata: tramite l'utilizzo della *ap_run_myhook* in un certo punto del codice, si permetterà che in quel punto un modulo si inserisca nel flusso di esecuzione.

Per completare il discorso rimane da descrivere come un modulo può trarre profitto da un hook registrando un metodo per esso. La registrazione avviene tramite una funzione del tipo *ap_hook_name*, dove `name` è il particolare nome dello hook. Proseguendo con l'esempio fatto in precedenza, una registrazione sarà del tipo

```
ap_hook_myhook(hook_function, NULL, NULL, APR_HOOK_MIDDLE);
```

Il primo argomento di questa funzione, in questo caso *hook_function*, è il più importante: è il nome della funzione (presente nel modulo) che sarà invocata dal nucleo quando arriverà alla *ap_run_myhook()*.

I successivi parametri permettono di specificare invece delle particolari modalità di ordinamento tra i moduli. Infatti, più di un modulo può definire un handler hook e la loro invocazione da parte del nucleo avviene generalmente in base all'ordine in cui essi sono stati registrati: l'unico modo per modificare questo comportamento è quello di utilizzare tali parametri.

Notiamo inoltre che un handler hook può anche non essere invocato dal nucleo. Per chiarire il concetto introduciamo i valori che può ritornare un hook. Essi sono:

- OK - il metodo che realizza lo hook ha completato le sue operazioni;
- DECLINED - il metodo che realizza lo hook non è interessato ad eseguire le sue operazioni;
- un codice di risposta HTTP - tipicamente indicante la condizione di errore verificatasi.

Solitamente il nucleo invoca tutti gli hook handler che sono registrati per un hook, a meno che un modulo non restituisca un codice di errore: in questo caso avremo che la procedura di invocazione verrà interrotta a quel modulo. Tale comportamento si verifica sia per un hook implementato con la macro AP_IMPLEMENT_HOOK_RUN_ALL, sia per uno implementato con la AP_IMPLEMENT_HOOK_RUN_FIRST (le uniche due macro che implementano un hook per cui è previsto valore di ritorno). Inoltre, se un hook è stato implementato con la RUN_FIRST, si avrà che la procedura di invocazione terminerà non solo in caso di errore ma anche non appena un handler hook restituisce `OK`. Ciò equivale a dire che un hook di questo tipo è un hook per il quale solo ad un modulo ne è permessa la gestione.

Un'ultima annotazione riguarda il fatto che, sebbene la definizione di un hook avvenga generalmente nel nucleo, anche un modulo può definire degli hook in maniera analoga a quanto riportato. Così un modulo permette ad altri moduli di inserirsi nel proprio flusso di esecuzione.

5.1.2 Hook principali

In una qualsiasi distribuzione di Apache2 che includa i sorgenti del server, gli hook definiti tramite il meccanismo appena analizzato possono essere elencati utilizzando lo script `support/list_hooks.pl`. Essi possono essere suddivisi in due gruppi, quelli relativi all'avvio o riavvio del server e quelli relativi alla ricezione e al processamento di una richiesta HTTP. Di seguito verranno analizzati gli hook principali di Apache2 secondo tale suddivisione.

Hook per l'avvio del server

Gli hook di seguito descritti sono relativi alle fasi di *First-time initialization* e *Restart loop* (vedi 4.2.3). Per ognuno di essi il nome riportato è lo stesso che i moduli devono utilizzare in fase di registrazione degli hook.

pre_config Questo hook viene attivato dopo che la configurazione del server è stata letta ma prima che essa venga processata nelle due fasi di *First-time initialization* e *Restart loop*.

open_logs Questo hook può essere utilizzato se un modulo ha la necessità di aprire file di log o iniziare un processo di logging.

post_config Questo hook viene attivato dopo che la configurazione è stata letta ed elaborata nelle due fasi di *First-time initialization* e *Restart loop*. E' importante notare che tale hook, così come il `pre_config`, viene attivato due volte, dunque particolari attenzioni vanno prese nelle elaborazioni fatte al suo interno.

child_init Ogni handler registrato per questo hook viene invocato una volta per ogni processo figlio, non appena esso viene creato. Tale hook risulta dunque appropriato per operazioni di inizializzazione che devono essere effettuate in un figlio.

Hook per ricevere e processare le richieste

Gli hook di seguito descritti sono relativi alla fase di *Request-response loop* e alla sua fase interna, il *Keep-Alive loop* (vedi 4.2.3). Da notare che verrano presi in considerazione solamente gli hook più importanti per lo sviluppo di un modulo (gli hook interni non verranno dunque trattati).

pre_connection Questo hook, relativo alla fase di *Request-response loop*, permette di eseguire operazioni che è necessario effettuare prima del processamento di una richiesta ma dopo aver accettato la connessione TCP.

post_read_request Viene attivato immediatamente dopo che lo header di una richiesta è stato letto. Può dunque essere usato per operazioni che si basano unicamente su informazioni desumibili da un header HTTP.

translate_name Questo hook può essere sfruttato per effettuare particolari mapping tra un indirizzo URL contenuto in una richiesta ed un nome di file.

access_checker Questo hook può essere utilizzato per verificare se la macchina client può accedere alla risorsa richiesta. In genere la verifica viene fatta su gruppi di indirizzi IP o in base allo user-agent utilizzato.

check_user_id E' un hook relativo ad una eventuale procedura di autenticazione, utilizzato per verificare dunque se le credenziali fornite da un client sono valide. Tipicamente si tratta di confrontare i dati di autenticazione forniti con quelli presenti in locale e memorizzati in un file o in un database.

auth_checker E' attivato successivamente allo hook check_user_id ed è relativo alla procedura di autorizzazione di un client la cui identità è stata provata. Permette di verifica dunque se quel determinato client ha i permessi per accedere alla risorsa che ha richiesto.

type_checker Questo hook permette ad un handler di determinare o di impostare il tipo MIME della risorsa richiesta. La determinazione del tipo MIME stabilisce quale sarà il gestore di contenuti (content handler) che dovrà generare la risposta.

fixups Tale hook viene attivato poco prima che avvenga la gestione della richiesta, per cui rappresenta l'ultima chance per un modulo di eseguire delle operazioni prima che la richiesta venga elaborata.

handler Questo hook attiva il cosiddetto *content handler*, tramite il quale ciascun modulo può elaborare la richiesta e generare la risposta da inviare al client.

log_transaction Con questo hook si permette ad un modulo di registrare, tramite dei log, l'avvenuto processamento della richiesta. Quando questo hook viene attivato, infatti, il processamento è già stato completato.

5.2 Tecniche comuni

In questa sezione verranno descritte le tecniche comuni da utilizzare nello sviluppo di un modulo. Innanzitutto un modulo deve comunicare al nucleo in quali punti del flusso di esecuzione vuole essere coinvolto, per cui nella prima sottosezione verrà analizzata la struttura dati che permette di registrare handler per gli hook: tale struttura incarna il punto di contatto tra

un modulo ed il nucleo. In seguito, vista l'importanza di fornire agli utilizzatori di un modulo la possibilità di configurarne le funzionalità, verranno affrontati i metodi da utilizzare per la gestione della configurazione.

Nel corso della trattazione si farà riferimento implicito alle API di Apache, tramite le quali i moduli recuperano dal nucleo le opportune informazioni. Generalmente esse possono essere facilmente individuate perché presentano il prefisso *ap*. Fra le API di Apache possono essere annoverate anche quelle facenti parte della libreria Apache Portable Runtime ([24]). Esse sono contraddistinte dal prefisso *apr*. La libreria Apache Portable Runtime (APR) fornisce funzionalità indipendenti dal sistema operativo e può essere utilizzata anche per progetti esterni ad Apache.

5.2.1 Informazioni da fornire al nucleo

Il punto di contatto tra il nucleo di Apache ed un modulo è costituito da una struttura, denominata `module`, che ciascun modulo deve contenere: tale struttura fornisce infatti al nucleo informazioni sulle funzioni implementate dal modulo, in modo tale che esse possano essere opportunamente invocate. In Apache 2, la definizione di una struttura di tipo `module` ha la seguente forma:

```
module AP_MODULE_DECLARE_DATA mymodule = {

  STANDARD20_MODULE_STUFF,
  my_create_dir_config,     /*create per-directory config struct*/
  NULL,                     /*merge per-directory config structs*/
  my_create_srv_config,     /*create per-server config structure*/
  NULL,                     /*merge per-server config structures*/
  commands,                 /*command apr_table_t*/
  register_hooks            /*register hooks*/
};
```

Listing 5.1: Definizione di una struttura di tipo module

Nella definizione sono presenti diverse entrate, fra le quali la prima ha sempre lo stesso valore, `STANDARD20_MODULE_STUFF`: tale entrata distingue un modulo per Apache2 da quelli per versioni precedenti e serve ad inizializzare alcuni elementi che il nucleo utilizza nella gestione dei moduli. Le successive entrate invece sono relative alle funzioni di aggancio predefinite offerte dal nucleo: se il modulo implementa una certa funzione allora la corrispondente entrata conterrà il nome della funzione definita dal modulo, viceversa conterrà `NULL`. Le funzioni presenti nell'esempio saranno utilizzate nel corso della trattazione; riportiamo ora, comunque, una breve descrizione di tutte le entrate, nell'ordine in cui compaiono nella struttura `module`.

Creazione struttura di configurazione per-directory Questa entrata viene utilizzata nel caso in cui il modulo necessita di utilizzare dati

di configurazione che abbiano un contesto per-directory (vedi sezione 4.3.1). La funzione corrispondente viene invocata dal nucleo una prima volta per il server principale ed ogni volta per ogni direttiva specifica per quel modulo.

Unione strutture di configurazione per-directory Tale aggancio può essere utilizzato se il modulo necessita di stabilire una configurazione assoluta valida per una specifica richiesta: tale opportunità è utile nel caso di incongruenze fra i diversi file di configurazione che vengono presi in considerazione per una richiesta.

Creazione struttura di configurazione per-server Tale aggancio viene utilizzato se il modulo necessita di utilizzare dati di configurazione che abbiano un contesto per-server (vedi sezione 4.3.1). La funzione corrispondente viene invocata dal nucleo una volta per il server principale ed una volta per ciascun virtual host.

Unione strutture di configurazione per-directory Tale aggancio può essere utilizzato se il modulo ha la necessità di ereditare, dal server principale, eventuali opzioni che non sono state impostate all'interno di una sezione `<VirtualHost>`. La funzione corrispondente viene invocata dal nucleo una volta per ciascun virtual host.

Gestore di comandi Questo aggancio viene utilizzato se il modulo necessita di definire delle direttive di configurazione: in questa entrata viene fornito un riferimento ad tabella contenente le associazioni fra le direttive e le corrispondenti funzioni che devono gestire i parametri da fornire.

Registrazione di agganci In corrispondenza di questa entrata viene fornito il riferimento ad una funzione (`register_hooks` nell'esempio precedente) che permette di registrare funzioni per tutti gli altri agganci forniti dal nucleo, ovvero quelli a cui si è fatto riferimento in 5.1.2. L'utilizzo di questa "redirezione" permette di inserire nuovi hook all'interno del nucleo senza alterare l'interfaccia della module structure.

5.2.2 Gestione della configurazione in un modulo

Affinchè un modulo possa fornire piene funzionalità è fondamentale che esso permetta di configurare alcuni aspetti del suo funzionamento. L'importanza della configurazione per un modulo è sottolineata dal fatto che ben cinque entrate di una struttura di tipo `module` (listato 5.1) riguardano aspetti di configurazione.

In ciascun modulo, infatti, è usuale definire una configurazione specifica, identificata generalmente da una apposita struttura dati. La configurazione deve poi essere allocata, inizializzata e devono essere fornite eventuali disposizioni per le operazioni di *merging*: lo sviluppatore di un modulo dovrà dunque scrivere le funzioni che compiono tali operazioni ed inserirle all'interno delle apposite entrate in una struttura di tipo module.

Da notare che quest'ultima operazione è influenzata dalla scelta effettuata sul contesto cui si vuole venga applicata la configurazione: generalmente infatti si utilizza una configurazione che si applichi o al contesto per-directory o a quello per-server (vedi sezione 4.3.1).

Con riferimento alla definizione della struttura `mymodule` effettuata nel listato 5.1, verranno ora forniti esempi di come utilizzare tre fra i possibili agganci disponibili: si descriverà come creare una configurazione per-server, come creare una configurazione per-directory e come implementare direttive di configurazione. Inoltre verrà anche mostrato come accedere alla configurazione all'interno delle varie funzioni del modulo. Non verranno invece discussi ulteriormente gli agganci relativi alle funzioni di *merging*, dato che si tratta di operazioni la cui implementazione è spesso non necessaria.

Creazione configurazione per-server

Per poter utilizzare in un modulo una configurazione per-server, si dovrebbe utilizzare questo tipo di codice:

```
typedef struct {
/* data variables */
} my_srv_config;

static void* my_create_srv_config(apr_pool_t* pool,
                                  server_rec* srv
                                 )
{
    my_srv_config* srv_cfg;
    srv_cfg = apr_pcalloc(pool, sizeof(my_srv_config) );
    /* Setup default values for data variables */
    return srv_cfg;
}
```

Listing 5.2: Creazione per-server config

Si osserva che prima di tutto è stata effettuata la definizione della struttura destinata a contenere i parametri di configurazione: essa è nell'esempio di tipo `my_srv_config`. Nella funzione *my_create_srv_config*, il cui nome deve comparire nell'apposita entrata della struttura `module`, si ha invece l'allocazione all'interno del pool e l'inizializzazione della variabile `srv_cfg`: essa rappresenterà appunto la configurazione del modulo.

Creazione configurazione per-directory

Per poter invece usufruire di una configurazione per-directory il codice è analogo, eccettuato il prototipo della funzione da utilizzare per l'aggancio. Si veda a tal proposito il listato 5.3.

```
typedef struct {
/* data variables */
} my_dir_config;

static void* my_create_dir_config(apr_pool_t* pool, char* x){
    my_dir_config* dir_cfg;
    dir_cfg = apr_pcalloc(pool, sizeof(my_dir_config) );
    /* Setup default values for data variables */
    return dir_cfg;
}
```

Listing 5.3: Creazione per-directory config

Accesso alla configurazione

Tipicamente si ha la necessità di accedere ai parametri di configurazione del modulo all'interno degli handler hook definiti dal modulo stesso. Questo può essere fatto grazie a due strutture dati fornite dal nucleo, `server_rec` e `request_rec`, entrambe definite nel file `httpd.h`.

Se si ha a disposizione la prima di queste strutture è possibile accedere solo alla propria configurazione per-server, tramite il campo `module_config`, come indicato di seguito:

```
my_srv_config* srv;
srv = ap_get_module_config(s->module_config, &mymodule);
```

dove `s` è appunto una variabile di tipo `server_rec`, `&mymodule` è invece l'indirizzo alla struttura module definita nel proprio modulo.

Se si ha a disposizione una variabile di tipo `request_rec`, `r` nell'esempio seguente, si può invece accedere anche alla configurazione per-directory del modulo, usando il campo `per_dir_config`:

```
my_dir_config* dir;
dir = ap_get_module_config(r->per_dir_config, &mymodule);
```

Si nota come, in entrambi i casi, per "recuperare" la configurazione si fa uso della funzione `ap_get_module_config`.

Ottenuto il puntatore alla struttura contenente la configurazione del modulo (`srv` o `dir` negli esempi precedenti) sarà possibile accedere ai singoli parametri di configurazione, in genere rappresentati da campi della struttura di configurazione.

Implementazione direttive di configurazione

I parametri di configurazione specifici del modulo possono essere impostati dall'utente tramite l'utilizzo di apposite direttive, che devono essere definite dal modulo stesso. Tale definizione deve essere effettuata utilizzando un array di tipo `cmd_rec`, come viene esplicitato dal seguente listato, in cui si specifica che il modulo può gestire una direttiva di nome `ExampleDirective`.

```
static const cmd_rec commands [] = {
    AP_INIT_FLAG("ExampleDirective", set_ex_directive, NULL,
                 RSRC_CONF, "Help about example directive"
                ),
    /* other directives if needed */
    {NULL}
};
```

Listing 5.4: Implementazione direttive

La definizione della direttiva viene fatta nell'esempio tramite l'uso della macro `AP_INIT_FLAG`, che è contenuta assieme a diverse altre nel file `http_config.h`: tramite tali macro si può specificare, per ogni particolare direttiva, il tipo e il numero dei parametri da poter impostare. A scopo esemplificativo si possono citare le seguenti:

- `AP_INIT_NO_ARGS`: specifica che la direttiva non può accettare argomenti;
- `AP_INIT_FLAG`: specifica che la direttiva può accettare un solo argomento di tipo On/Off;
- `AP_INIT_TAKE1`: specifica che la direttiva può accettare un solo argomento di tipo stringa.

Tutte queste macro hanno lo stesso prototipo, che sarà chiarito analizzando il significato dei parametri attuali utilizzati nel listato 5.4.

1. `ExampleDirective`: il nome della direttiva;
2. `set_ex_directive`: la funzione che implementa la direttiva, tipicamente in essa si recupera la configurazione e si settano i relativi parametri, come mostrato di seguito.

```
static const char* set_ex_directive(cmd_parms* cmd,
                                    void* mconfig,
                                    const int arg){
    my_server_conf* s_cfg;
    s_cfg=ap_get_module_config(cmd->server->module_config,
                               &mymodule);
    /* set parameters */
    return NULL;
}
```

3. `NULL`: un puntatore a dati;

4. `RSRC_CONF`: indica dove la direttiva è permessa ed in questo caso specifica un contesto per-server. Altri valori potrebbero essere `OR_ALL`, ad indicare che la direttiva può essere usata in tutti i file di configurazione, oppure `ACCESS_CONF`, ad indicare che può essere in un contesto per-directory all'interno del file di configurazione globale;

5. `Help about example directive`: indica un breve messaggio di aiuto che viene visualizzato in caso di errore nell'impostare la direttiva.

Nel caso dell'esempio considerato si avrà dunque che il modulo sarà capace di interpretare una direttiva del tipo

```
ExampleDirective On
```

presente in uno dei file di configurazione del server.

Capitolo 6

Architettura

L'architettura che è stata realizzata nel corso di questo lavoro ha come scopo principale quello di fornire dei servizi tramite Web garantendo una certa qualità del servizio (QoS), che può essere contratta con il soggetto consumatore che intende fruire di tali servizi. In particolare, viene previsto che le richieste di servizio siano ricevute ed eseguite da un server web, il quale, per poter eseguire tali servizi con garanzie di QoS, viene esteso con un opportuno meccanismo di gestione delle risorse. La QoS che deve essere garantita viene precedentemente contrattata con il consumatore, tramite un protocollo basato sulla stipula di accordi: l'architettura dunque prevede l'interconnessione tra il sottosistema che effettua la contrattazione e quello che eroga il servizio.

Nella nostra architettura il server web utilizzato è Apache2, il supporto alla gestione della qualità del servizio avviene tramite AQuoSA, il framework di contrattazione utilizzato è invece WS-Agreement. I concetti di base relativi a tali tecnologie sono stati ampiamente discussi nei capitoli precedenti ed in questa sede si farà ampio riferimento ad essi. Anche le motivazioni relative all'utilizzo di tali tecnologie sono già state discusse (vedi sezione 1.1) e dunque non verranno ripetute.

In questo capitolo verranno, invece, prima analizzati e specificati i requisiti del sistema, in seguito si descriverà l'architettura realizzata ed infine verranno esposti alcuni dettagli implementativi relativi allo sviluppo dell'architettura.

6.1 Analisi e specifica dei requisiti

Per poter capire e descrivere in maniera precisa e completa che cosa si vuole far fare al sistema, verrà trattata in questa sezione la fase di analisi e specifica dei requisiti. In particolare, inizialmente ci si rivolgerà alla comprensione del problema dell'utente (analisi dei requisiti dell'utente) ed in seguito verranno

descritte le caratteristiche del sistema software (specifica dei requisiti del software).

6.1.1 Analisi dei requisiti dell'utente

L'analisi dei requisiti dell'utente può essere suddivisa in due parti, una di definizione ed una di specifica. La definizione dei requisiti risulta meno dettagliata e permette una migliore comprensione generale del problema, la specifica invece affronta l'analisi con un maggior grado di dettaglio.

Definizione dei requisiti dell'utente

1. Fruizione di un servizio web con una certa QoS.
2. Contrattazione della QoS con cui si vuole fruire uno o più servizi.

Specifica dei requisiti dell'utente

1. Fruizione del servizio web
   - (1.1) Fruizione di uno o più servizi accessibili dal Web.
   - (1.2) Rispetto, durante la fruizione del servizio, delle garanzie di QoS che sono state assicurate.
   - (1.3) Informazioni sulle caratteristiche del sistema che fornisce i servizi.
2. Contrattazione della QoS
   - (2.1) Richiesta di una certa QoS per la fruizione di uno o più servizi.
   - (2.2) Informazioni sull'accettazione della richiesta e dunque sulla conseguente stipula del contratto di fornitura.
   - (2.3) Terminazione dell'accordo.

6.1.2 Specifica dei requisiti del sistema

Si descriveranno ora le caratteristiche che deve avere il sistema per poter soddisfare le esigenze individuate nella sottosezione precedente.

1. Fruizione del servizio web
   - (1.1) Il sistema deve rendere visibile all'esterno un server web, che permetta di ricevere ed elaborare le richieste di servizio effettuate dai client.
   - (1.2) Il sistema deve far sì che il server web esegua il servizio con la QoS promessa, tramite l'utilizzo di un meccanismo che fornisca garanzie temporali in fase di esecuzione.

(1.3) Il sistema deve descrivere le proprie caratteristiche fondamentali: sistema operativo usato, numero e tipo di CPU, tipo del server web.

2. Contrattazione della QoS

(2.1) Il sistema deve permettere all'utente di specificare la QoS desiderata tramite un parametro che indica la percentuale di CPU che l'utente vuole sia utilizzata nell'esecuzione dei servizi richiesti.

(2.2) Il sistema deve fornire all'utente notifica del risultato del proprio processo decisionale riguardante l'accettazione delle richieste di QoS che gli sono state fatte. Nel caso di accettazione, il sistema si impegna a fornire i propri servizi con la QoS specificata.

(2.3) Il sistema deve permettere all'utente di terminare esplicitamente un accordo quando egli non ha più bisogno di fruire i servizi forniti dal sistema.

6.2 Progetto

In questa sezione verrà documentata la fase di progetto. Inizialmente verrà effettuata una descrizione generale dell'architettura, in seguito si esamineranno con maggiore dettaglio i vari componenti che la costituiscono.

6.2.1 Descrizione generale dell'architettura

La nostra architettura deve permettere ad un consumatore di contrattare la qualità del servizio desiderata nella fruizione di un certo numero di servizi.

La contrattazione della QoS tra un consumatore ed il nostro sistema, che è il fornitore di servizi, viene effettuata con un modello basato sulla stipula di accordi. In particolare, tra i due attori la contrattazione avviene per mezzo di uno scambio di messaggi che seguono il protocollo sincrono di WS-Agreement (si veda 2.3.1). Così come avviene per tutti i sistemi basati su WS-Agreement (vedi sezione 2.1), anche la nostra architettura sarà suddivisa in due livelli: il livello di accordo o *agreement layer* ed il livello di servizio o *service layer*. Tale suddivisione consente di trattare in maniera distinta l'interfaccia per la contrattazione da quella per l'erogazione del servizio.

Nel caso specifico del nostro dominio applicativo, il consumatore del servizio coincide con il ruolo dell'Agreement Initiator ed ha il compito di iniziare la contrattazione. Il fornitore, invece, all'interno del quale si inserisce la nostra architettura, coincide con l'Agreement Responder ed ha il compito di fornire i prototipi di accordo. Sulla base di questi prototipi il consumatore invierà la propria proposta di accordo, contenente la QoS richiesta. Se la contrattazione andrà a buon fine verrà creato un accordo (Agreement). Dopo

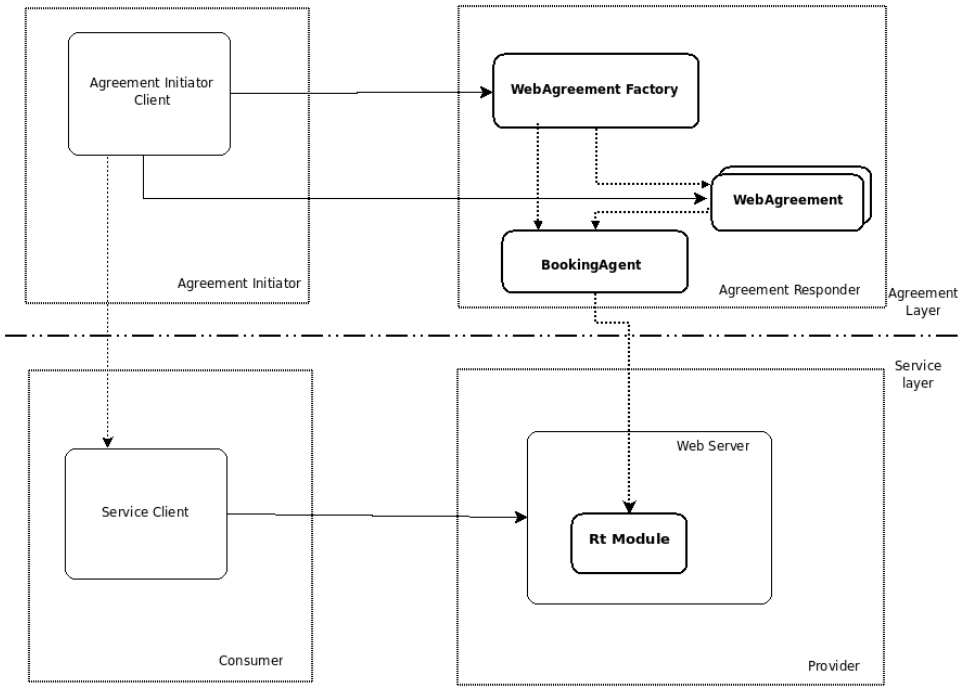

Figura 6.1: Modello concettuale dell'architettura realizzata

che un accordo è stato creato, il nostro sistema si deve impegnare a garantire la QoS contrattata durante la fornitura dei successivi servizi richiesti da quel consumatore. Quando il consumatore ha terminato la fruizione dei servizi, egli richiederà in maniera esplicita la terminazione dell'accordo.

L'adattamento del tipico modello di interazione WS-Agreement al nostro sistema viene raffigurato in figura 6.1.

Con riferimento a tale figura possono anche essere individuati i componenti principali della nostra architettura, di cui si darà in questa fase una breve descrizione.

- *WebAgreementFactory* - E' il componente si preoccupa di contrattare con il consumatore la qualità del servizio. In particolare interagisce con il consumatore nel processo di creazione di un accordo ed eventualmente crea l'accordo;

- *WebAgreement* - E' il componente che svolge tutte quelle operazioni che sono relative ad un Agreement, ovvero fornisce al client informazioni in merito ad un particolare Agreement e permette la terminazione di un Agreement. Tale componente entra in gioco solamente nel caso in cui la procedura di creazione di un Agreement, effettuata dal WebAgreementFactory, sia andata a buon fine;

- *BookingAgent* - Si preoccupa di decidere se la QoS richiesta dal con-

sumatore può essere garantita ed eventualmente prenota le risorse da destinare all'esecuzione dei servizi per quel client. Si occupa anche di "cancellare" le prenotazioni, ovvero di liberare le risorse utilizzate nella fornitura dei servizi associati ad un particolare client;

- *Rt Module* - Permette di eseguire il servizio richiesto dal consumatore con la QoS desiderata. Tale componente si trova all'interno del Web Server, che può essere considerato come il servizio principale esposto dall'architettura: tramite un accordo, infatti, si contratta l'uso del web server per l'esecuzione dei servizi richiesti. Ciò implica che il consumatore potrà usufruire di tutti i servizi disponibili tramite l'accesso al web server.

I componenti individuati verranno di seguito analizzati in maniera approfondita.

6.2.2 WebAgreementFactory

Il componente WebAgreementFactory interagisce con il consumatore nel processo di creazione di un accordo: fornisce i prototipi, riceve le proposte di accordo e comunica al consumatore se esse sono state accettate. Realizza dunque i requisiti 1.3 e 2.1, inoltre collabora con il BookingAgent e il WebAgreement per realizzare il requisito 2.2.

Se si considera la trattazione effettuata riguardo al protocollo di creazione di un Agreement (sezione 2.3.1), tale componente ha lo stesso ruolo dell'AgreementFactory. In termini di relazioni tra le classi potremmo infatti considerare il WebAgreementFactory come una realizzazione dell'interfaccia AgreementFactory (vedi figura 6.2).

I metodi implementati dal WebAgreementFactory, di cui in ogni momento esiste un'unica istanza all'interno del sistema, sono di seguito analizzati.

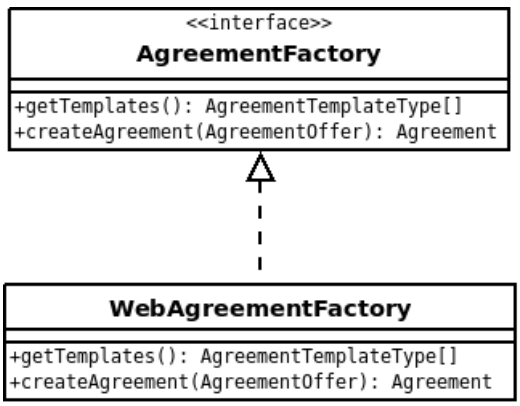

Figura 6.2: Relazione tra WebAgreementFactory e AgreementFactory

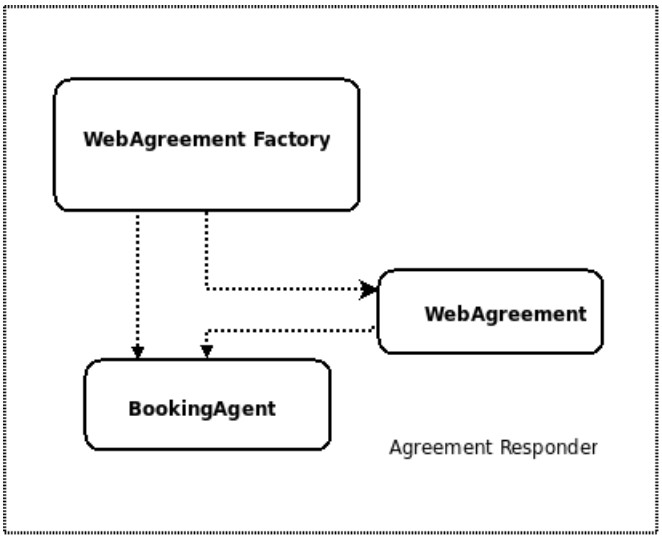

Figura 6.3: Architettura a livello di accordo

- *getTemplates* - Questo metodo restituisce un array di prototipi di tipo `AgreementTemplateType`. Viene utilizzato dal consumatore per prelevare i prototipi di accordo forniti dal componente. Tali prototipi vengono generati dinamicamente a seguito di una invocazione del metodo e descrivono principalmente il contesto in cui verrà eseguito il servizio (es. sistema operativo, velocità del processore, etc.), il tipo di servizio offerto e i parametri di QoS contrattabili. In particolare esiste un solo parametro contrattabile, ovvero la percentuale di CPU con cui si vuole venga eseguito il servizio.

- *createAgreement* - Questo metodo prende come parametro una proposta di accordo di tipo `AgreementOffer` e restituisce un accordo di tipo `Agreement` nel caso in cui il componente decide di accettare la proposta fatta dal consumatore. Se invece questa proposta non viene accettata il metodo lancia delle eccezioni di tipo `AgreementFactoryException`, che informano il consumatore sulle ragioni del rifiuto. Tale rifiuto infatti può avvenire sia perché i termini contenuti nella proposta di accordo non sono correttamente definiti, sia perché il sistema non è in grado di garantire la QoS desiderata dal consumatore.

Il componente WebAgreementFactory, durante il processo di creazione, interagisce, oltre che con il consumatore, anche con gli altri due componenti dell'architettura che si trovano a livello di accordo, ovvero il WebAgreement e il BookingAgent (figura 6.3).

L'interazione con il BookingAgent avviene quando si deve decidere se la QoS richiesta dal consumatore può essere accordata. Al BookingAgent è infatti delegato il processo decisionale di accettazione dell'Agreement in base

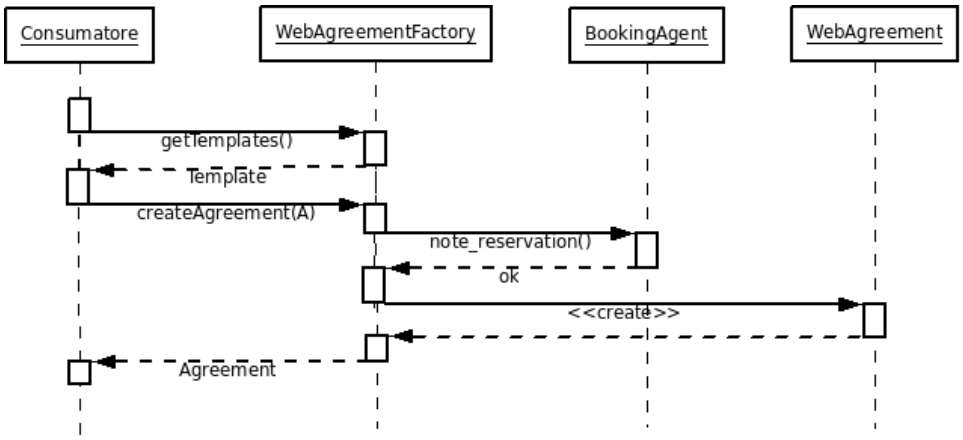

Figura 6.4: Diagramma di sequenza: operazione di creazione di Agreement avvenuta con successo

alle risorse interne del sistema. Se il BookingAgent risponde negativamente, allora il WebAgreementFactory comunicherà al consumatore il rifiuto della proposta di accordo. Altrimenti il WebAgreementFactory dovrà interagire con il componente WebAgreement per la creazione vera e propria dell'Agreement.

Le interazioni che avvengono nel caso relativo ad un'operazione di creazione di un Agreement completata con successo possono essere rappresentate, in maniera formale, con il diagramma di sequenza di figura 6.4.

Nel diagramma si nota come il WebAgreementFactory verifica la possibilità di garantire la QoS desiderata dal consumatore attraverso l'invocazione del metodo *note_reservation*. Se questo metodo restituisce un valore positivo allora verrà propagata al WebAgreement la richiesta di creazione dell'accordo e verrà poi chiuso il colloquio con il consumatore.

E' da notare che il caso in cui una proposta di accordo viene rifiutata, a causa dell'impossibilità di garantire la QoS desiderata, differisce da quello presentato in figura 6.4 solamente per il fatto che nessuna interazione avviene con il WebAgreement. Infatti, a seguito di un valore di ritorno negativo della *note_reservation*, viene immediatamente lanciata dal WebAgreementFactory una eccezione che comunica al consumatore il rifiuto della proposta.

6.2.3 WebAgreement

Il componente WebAgreement entra in gioco solamente dopo che si è concluso positivamente un processo di creazione con il consumatore. Questo componente, infatti, incarna un accordo già stipulato, per cui realizza le funzionalità che permettono le comuni operazioni informative e di monitoraggio di un Agreement, secondo la tipica interfaccia che è stata esposta

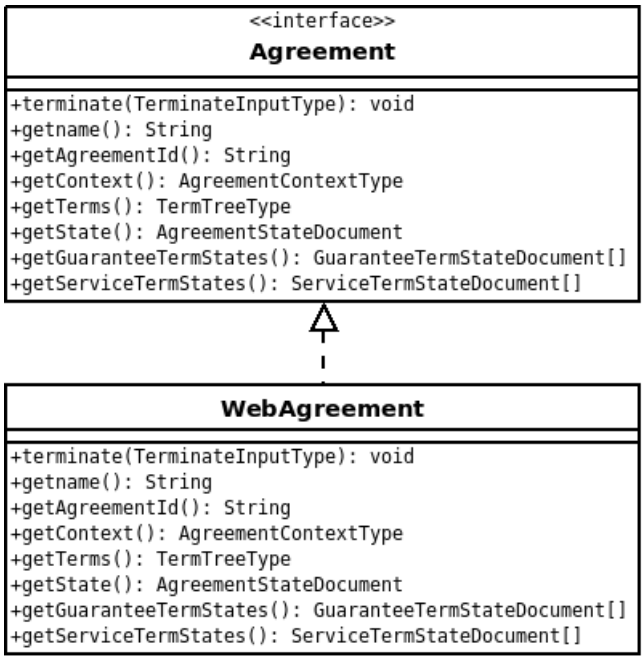

Figura 6.5: Relazione tra WebAgreement e Agreement

nella sezione 2.3.2. Tale componente realizza il requisito 2.3 e collabora con il WebAgreementFactory per la realizzazione del requisito 2.2.

In un sistema basato su WS-Agreement, le operazioni informative e di monitoraggio sono definite dal componente denominato Agreement: in termini di relazioni tra classi dunque si avrà che WebAgreement è una realizzazione dell'interfaccia Agreement, la quale definisce i metodi che dovranno essere implementati dal componente.

Con riferimento alla figura 6.5, le operazioni fornite dal componente WebAgreement sono di seguito elencate. Nella loro descrizione si farà riferimento ai termini contenuti nella struttura di un accordo, per la quale si rimanda alla sezione 2.2.

- *terminate(reason)* - Permette al consumatore di terminare l'accordo, specificando le ragioni di terminazione. In tal modo si può comunicare al fornitore che non si ha più bisogno di usufruire dei servizi per i quali è stata contrattata la QoS. A seguito dell'invocazione di questo metodo ogni obbligo tra il consumatore e il fornitore cessa di esistere.

- *getName()* - Restituisce un valore di tipo `String` che rappresenta il nome dell'accordo. Può essere restituita una stringa vuota nel caso in cui nessun nome è stato definito in fase di offerta.

- *getAgreementId()* - Restituisce un valore di tipo `String` che rappre-

senta l'identificatore dell'accordo. Tale valore deve essere definito nell'accordo e deve essere univoco.

- *getContext()* - Restituisce un valore di tipo `AgreementContextType` che rappresenta la sezione Context contenuta nell'accordo. Tale sezione contiene metadati riguardanti l'accordo.

- *getTerms()* - Restituisce un valore di tipo `TermTreeType` che costituisce la sezione Terms contenuta nell'accordo.

- *getState()* - Restituisce un valore di tipo `AgreementStateDocument` che rappresenta lo stato di un accordo. I valori di stato che può assumere un accordo sono gli stessi definiti nella sezione 2.3.2.

- *getGuaranteeTermStates()* - Restituisce un array di valori che sono di tipo `GuaranteeTermStateDocument`, che rappresentano lo stato di ciascuno dei termini di garanzia contenuti nell'accordo. I valori dello stato di una garanzia sono gli stessi definiti nella sezione 2.3.2.

- *getAgreementServiceTermStates()* - Restituisce un array di valori di tipo `ServiceTermStateDocument`, che rappresentano lo stato di ciascuno dei termini di servizio contenuti nell'accordo. I valori dello stato di un servizio sono gli stessi definiti nella sezione 2.3.2.

Analizziamo ora le interazioni del componente in esame con gli altri componenti del livello di accordo. Innanzitutto, osserviamo che il WebAgreement viene invocato dal WebAgreementFactory al momento della effettiva creazione di un accordo: ogni accordo stipulato è infatti rappresentato da un'istanza della classe WebAgreement. Il WebAgreementFactory, infatti, al termine del processo di creazione invia al consumatore il riferimento dell'istanza WebAgreement che incarna l'accordo. Tale riferimento viene successivamente utilizzato dal consumatore, per poter invocare tutte le operazioni informative e di monitoraggio messe a disposizione dal componente. Un riferimento ad una risorsa viene indicato come EPR (Endpoint Reference) ([20]) nella terminologia legata ai Web Services: diremo dunque che il consumatore riceve l'EPR di un Agreement come prova dell'avvenuta accettazione della proposta di accordo.

Il WebAgreement, come già detto, espone anche l'operazione di *terminate*, con cui un consumatore può decidere di terminare l'accordo. Tale operazione può essere invocata dal consumatore sia nel caso in cui egli non sia soddisfatto del servizio fornito (ad esempio perché le garanzie sono state ripetutamente violate) sia per indicare una corretta terminazione del rapporto dovuta al completo espletamento dei servizi richiesti. In ogni caso, tale operazione si traduce nel liberare tutte le risorse associate all'esecuzione dei servizi per quel consumatore: per far questo dunque il WebAgreement deve interagire con il BookingAgent, comunicandogli di rimuovere le

risorse riservate a quel consumatore. Il metodo invocato dal WebAgreement sarà il *delete_reservation(c_id)*, che prende come parametro un valore di tipo `ClientId` con il quale si identifica in maniera univoca all'interno del sistema un particolare consumatore. L'interazione tra WebAgreement e BookingAgent può essere rappresentata con un diagramma di sequenza relativo ad un'operazione di terminazione di accordo (figura 6.6).

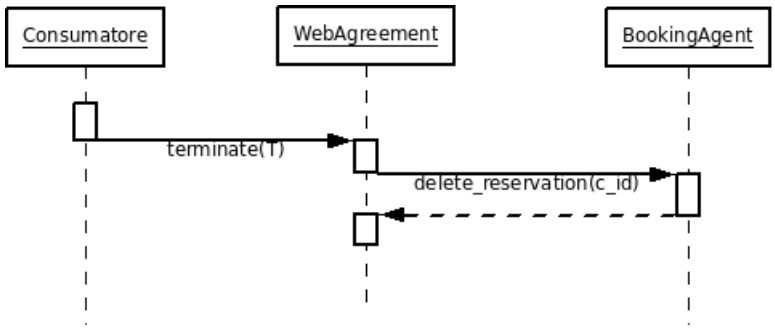

Figura 6.6: Diagramma di sequenza che rappresenta un operazione di terminazione di un Agreement

Dalla figura si nota come nessuna conferma venga data al consumatore come valore di ritorno del metodo *terminate()*. Tuttavia il consumatore può verificare l'effettiva terminazione dell'accordo tramite il metodo *getState()*. Se l'operazione ha avuto successo, infatti, lo stato dell'Agreement avrà il valore `Terminated`.

6.2.4 BookingAgent

Il BookingAgent è il componente che si occupa di gestire la prenotazione delle risorse necessarie a garantire la QoS richiesta dal consumatore. La risorsa considerata dal BookingAgent è la banda (frazione di utilizzazione) di processore da riservare all'esecuzione dei servizi per quel consumatore: tale parametro di QoS è infatti l'unico contrattabile.

I compiti del BookingAgent sono riassumibili come segue:

- verificare la disponibilità della banda di processore richiesta come parametro di QoS;
- prenotare la banda di processore richiesta;
- annullare una prenotazione effettuata.

Il diagramma della classe che realizza tale componente è mostrato, nei suoi dettagli maggiormente significativi, in figura 6.7. Una sola istanza della classe sarà presente nel sistema in ogni istante: in questo modo si garantisce coerenza e correttezza nelle decisioni riguardanti la disponibilità delle risorse.

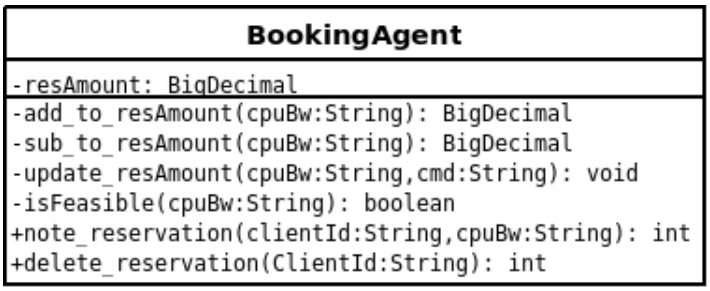

Figura 6.7: Diagramma della classe BookingAgent

In questo modo, infatti l'ammontare della banda di CPU allocata in totale è rappresentato in ogni istante dalla variabile `resAmount` e le operazioni eseguite su tale variabile saranno sicuramente consistenti. La variabile `resAmount` è di tipo `BigDecimal` poiché espressa come numero frazionario (es. 0.75). I metodi della classe che operano su tale variabile vengono di seguito analizzati.

- *add_to_resAmount(cpuBw)* - Metodo a visibilità privata che restituisce un valore di tipo `BigDecimal`. Tale valore rappresenta il numero che si ottiene sommando a `resAmount` il valore espresso da `cpuBw` .
- *sub_to_resAmount(cpuBw)* - Metodo a visibilità privata che restituisce un valore di tipo `BigDecimal`. Tale valore rappresenta il numero che si ottiene sottraendo a `resAmount` il valore espresso da `cpuBw` .
- *update_resAmount(cpuBw, cmd)* - Metodo a visibilità privata che permette di aggiornare il valore di `resAmount` aggiungendogli o sottraendogli (a seconda del valore di `cmd`) il valore espresso da `cpuBw`.
- *isFeasible(cpuBw)* - Metodo a visibilità privata che restituisce un booleano che indica se è possibile o meno aggiornare il valore di `resAmount` aggiungendogli quello espresso da `cpuBw`. Tale metodo permette di verificare se si può allocare una certa banda ad un consumatore senza pregiudicare la QoS dei servizi che si sta già fornendo ad altri consumatori. A tale scopo fa uso al suo interno del metodo *add_to_resAmount*.
- *note_reservation(clientId,cpuBw)* - Metodo a visibilità pubblica che restituisce un intero positivo nel caso in cui è possibile riservare a quel `clientId` la banda `cpuBw`. Altrimenti può restituire due codici di errore: uno che indica che la prenotazione non è possibile perché pregiudica la QoS degli altri consumatori; l'altro che indica che esiste già una prenotazione per quel consumatore. Per verificare la possibilità di effettuare la prenotazione senza pregiudicare la QoS fa uso del metodo *isFeasible*. Nel caso in cui la prenotazione è fattibile utilizza il metodo *update_resAmount* per aggiornare il valore di `resAmount`.

- *delete_reservation(clientId)* - Metodo a visibilità pubblica che cancella la prenotazione relativa al `clientId` specificato come parametro. Tale metodo libera la banda di CPU riservata al consumatore identificato dall'id e la rende disponibile. Fa uso del metodo *update_resAmount* per rendere consistente il valore di `resAmount`.

Se si fa riferimento alla figura 6.1, si nota come il BookingAgent interagisce sia con gli altri componenti a livello di accordo, sia con i componenti del livello di servizio dell'architettura.

A livello di accordo, le azioni corrispondenti ai compiti del BookingAgent vengono attivate a seguito dell'invocazione dei metodi *note_reservation* e *delete_reservation* da parte di WebAgreementFactory e WebAgreement, come indicato nelle sezioni relative a tali componenti.

Per quanto riguarda invece le interazioni a livello di servizio, i compiti del BookingAgent consistono nel:

- comunicare al RtModule la QoS che deve essere garantita nell'esecuzione dei servizi che da quel momento dovranno essere eseguiti per un particolare consumatore;
- comunicare al RtModule che non esistono più obblighi di qualità del servizio per un particolare consumatore.

Tale comunicazione avviene per mezzo di memoria condivisa, in cui ogni prenotazione effettuata dal BookingAgent viene rappresentata tramite una terna di valori (`ClientId`, `CpuBw`, `RsvId`). In tal modo si associa ad ogni consumatore identificato da `ClientId` una banda di CPU specificata da `CpuBw`. Il campo `RsvId` rappresenta invece l'identificatore della risorsa virtuale con cui verranno eseguiti i servizi di quel consumatore: tale risorsa virtuale avrà appunto banda `CpuBw`. La presenza del campo `RsvId` ci indica come il BookingAgent sia conoscenza di quello che avviene a livello di servizio: ciò tuttavia non pregiudica l'indipendenza tra interfaccia a livello di accordo ed interfaccia a livello di servizio, poiché il BookingAgent è un componente interno dell'architettura.

Per ogni consumatore, identificato dal `ClientId`, esisterà in ogni istante al più una entrata di questo tipo in memoria condivisa. In particolare, se vi sono attualmente obblighi con quel consumatore a seguito di un accordo, allora esisterà una entrata corrispondente, altrimenti non esisterà alcuna entrata associata a quel consumatore. Ogni entrata di questo tipo identificherà dunque un "*contratto valido*".

6.2.5 Rt Module

Il Rt Module è il componente che si preoccupa di ricevere ed elaborare le richieste di servizio ricevute da un consumatore, garantendo il rispetto della QoS che è stata contrattata. Realizza dunque i requisiti 1.1 ed 1.2.

Per poter ricevere ed elaborare le richieste di servizio, il Rt Module sfrutta le funzionalità del web server, di cui può essere considerato una estensione. Per poter invece garantire la QoS contrattata dal consumatore, utilizza le funzionalità messe a disposizione dal livello di supporto alla gestione. I parametri di QoS contrattati dal consumatore vengono resi disponibili dal livello superiore della nostra architettura (il livello di accordo).

Il componente, per quanto detto, può essere suddiviso in due parti principali, descritte brevemente di seguito.

- *WebServer Interface* - Permette la ricezione e l'elaborazione delle richieste di servizio, nonché la configurazione di alcune caratteristiche del componente. Tutto ciò viene realizzato sfruttando le funzionalità del web server, con cui ci si interfaccia. Vengono inoltre utilizzati i dati provenienti dal livello di accordo, necessari per determinare quali richieste sono da servire con una certa QoS.

- *Reservation Manager* - Permette di gestire la QoS dei servizi da eseguire con tali requisiti. Viene invocato nel momento in cui devono essere servite richieste con una certa QoS e si interfaccia verso il livello di supporto per poterne effettuare la gestione.

Tali sottocomponenti verrano di seguito analizzati in maniera approfondita.

WebServer Interface

Uno dei compiti di questo sottocomponente è quello di permettere la **configurazione** del *Rt Module*. Per permettere la configurazione del modulo si utilizzeranno le seguenti direttive:

- `RtModule`
- `RtUseAgreement`
- `RtCpuBwReservation`
- `RtSetHandler`

Tali direttive, per il loro particolare significato, andrebbero utilizzate nell'ordine suggerito. Infatti, la prima direttiva permette di abilitare o disabilitare il modulo: prende dunque un parametro che può assumere i valori `On` o `Off`. Se il modulo viene disabilitato con l'uso del valore `Off` allora le altre direttive non avranno alcuna validità.

Anche la seconda direttiva prende come parametro o un valore `On` o un valore `Off`, permettendo di specificare se il modulo viene utilizzato o meno in una architettura che fa uso di contrattazione della QoS. Questa direttiva è presente poiché il progetto del Rt Module prevede che esso possa essere

utilizzato anche al di fuori della nostra architettura. Nel nostro caso la direttiva sarà sempre impostata ad `On`. Da notare che se viene abilitata la contrattazione basata su Agreement, le successive due direttive non avranno validità.

La terza e la quarta direttiva permettono di utilizzare il modulo anche in una architettura in cui non si fa uso di contrattazione della QoS. La terza direttiva prende infatti come parametro un valore che specifica la banda di CPU con cui si vuole eseguire un certo servizio. La quarta direttiva permette di indicare tale servizio tramite la specifica del tipo di gestore che eseguirà un certo servizio. Ad esempio, se si vuole eseguire con una certa QoS un programma CGI, è sufficiente specificare il valore `cgi-script` come parametro della direttiva. Il valore da specificare è dunque quello con cui Apache identifica il gestore delle richieste di un certo tipo MIME. Si fa notare come l'utilizzo del modulo in architetture senza contrattazione non verrà ulteriormente approfondito e si ribadisce che le ultime due direttive non hanno validità nell'architettura in esame.

Sebbene la possibilità di configurazione sia di assoluto rilievo, il compito più importante di tale sottocomponente è tuttavia quello di **intercettare le richieste di servizio** fatte al server web, per poter stabilire se queste devono essere servite con una particolare QoS. Le richieste da servire con garanzie di QoS sono quelle che provengono da soggetti consumatori per i quali esistono dei contratti validi. Al livello di servizio della nostra architettura, un contratto valido è un contratto per cui esiste una terna (`ClientId`, `CpuBw`, `RsvId`) nella struttura dati condivisa con il livello di accordo. Per poter stabilire dunque se una richiesta deve essere servita con una particolare QoS, è sufficiente ottenere l'identificativo del consumatore (a partire dalla richiesta stessa) e verificare se esiste una entrata con lo stesso valore per il campo `ClientId`. Se la richiesta è soggetta a garanzie relative alla qualità del servizio allora viene chiamato in causa il ReservationManager.

Una richiesta di servizio viene effettuata da un consumatore tramite il protocollo HTTP: la richiesta di un servizio si materializza dunque nell'invio di una o più richieste HTTP. Il sottocomponente in esame non deve intercettare tutte le richieste HTTP di un consumatore per poter verificare la necessità di fornire garanzie di QoS a quel consumatore: è sufficiente a tale scopo intercettare solo alcune ben determinate richieste HTTP. Questo perché il protocollo HTTP, che si appoggia al TCP come protocollo di trasporto, permette che più richieste HTTP possano viaggiare su una stessa connessione TCP (in questo caso una connessione si dice di tipo Keep-Alive). Le richieste multiple che vengono effettuate su una stessa connessione vengono poi servite da uno stesso task all'interno della struttura concorrente del web server (si veda ad esempio il capitolo 4). Vi è dunque la necessità di intercettare solamente la prima richiesta instradata su una nuova connessione: se si verifica che per essa si devono fornire garanzie di QoS allora si dovranno fornire le stesse garanzie a tutte le successive richieste HTTP di

quella connessione, visto che provengono dallo stesso soggetto consumatore.

Il processo di verifica della necessità di fornire QoS dipende dalla disponibilità dell'identificativo del consumatore. Dato che questo è disponibile subito dopo la creazione della connessione, il modulo *Rt* può, in realtà, inserirsi nel flusso di esecuzione del server immediatamente prima che venga inviata la prima richiesta sulla connessione.

Le operazioni effettuate a quel punto per verificare se si deve fornire QoS a quel consumatore possono essere riassunte con il diagramma di flusso di figura 6.8. Notiamo che tali operazioni vengono fatte solamente una volta per ogni connessione, cioè dopo che una connessione è stata creata ma prima che venga ricevuta una richiesta. Se si verifica che la connessione proviene da un consumatore per il quale è stata contrattata una certa QoS, allora tutte le richieste HTTP provenienti da quel consumatore saranno servite con quella QoS.

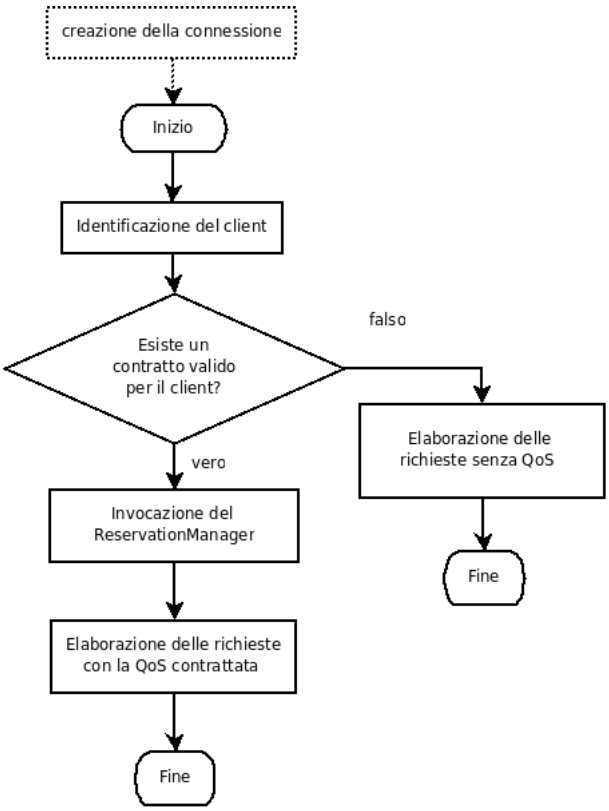

Figura 6.8: Diagramma di flusso: verifica della necessità di fornire QoS

Reservation Manager

Il Reservation Manager è invocato nel momento in cui viene intercettata una richiesta di servizio per cui deve essere garantita una certa QoS. I dati che devono essere utilizzati dal Reservation Manager, affinchè possa svolgere il suo compito di gestione, sono quelli relativi all'entrata (`ClientId`, `CpuBw`, `RsvId`) con cui si identifica un contratto valido. Infatti, il Reservation Manager utilizza il livello di supporto per associare a ciascun consumatore una risorsa virtuale: in questo caso tale risorsa è la CPU, per cui a ciascun consumatore sarà associata una CPU virtuale, identificata da `RsvId`, con banda pari al valore specificato da `CpuBw`.

Concettualmente, dunque, il compito principale del Reservation Manager è quello di creare una risorsa virtuale ed associare a tale risorsa il task della struttura concorrente del server web che sta servendo la richiesta di servizio. Poiché una risorsa virtuale viene generalmente distrutta quando non vi sono dei task ad essa associati, avremo che ciascuna risorsa virtuale ha il tempo di vita di una connessione: questo perché ogni qual volta viene instaurata una nuova connessione deve essere accertata l'identità del consumatore.

Il compito del Reservation Manager non si esaurisce nella creazione delle risorse virtuali, poiché nell'eseguire le richieste di servizio per un consumatore, possono presentarsi diversi casi a cui dover far fronte. In particolare ne sono stati identificati tre, di cui il primo è il caso cui si è già fatto riferimento, ovvero l'instaurazione della prima connessione tra client e server: in questo caso il Reservation Manager deve limitarsi a creare la risorsa virtuale e associarvi il task servente.

Il secondo caso è relativo al caso in cui più connessioni in contemporanea vengano instaurate con uno stesso soggetto consumatore: nel caso di connessioni successive alla prima, dunque, quello che deve fare il Reservation Manager è associare il task servente alla risorsa virtuale già creata e che contemporaneamente è in uso dagli altri serventi.

Il terzo caso prevede che l'unica connessione tra il consumatore ed il fornitore si andata in timeout mentre si è ancora in obblighi di garanzia del servizio. In questo caso, la risorsa virtuale, poiché ha il tempo di vita di una connessione, non esiste più e dunque deve essere nuovamente creata.

E' necessario dunque che il Reservation Manager tenga traccia, in ogni istante, della risorsa virtuale a cui sono attualmente associati i servizi relativi ad un particolare consumatore. Per poter far questo si utilizzerà il campo `RsvId`, per poter associare ad ogni consumatore una risorsa virtuale identificata univocamente all'interno del sistema. Al momento in cui viene servita la prima richiesta in assoluto di un consumatore avremo RsvId$=-1$, ad indicare che nessuna reservation è stata finora creata per quel consumatore. In questo caso diremo che non esiste una "nota della reservation". Viceversa, quando il valore di RsvId!$=-1$ si dirà che esiste una "nota della reservation".

In particolare, le operazioni che devono essere effettuate dal Reservation Manager, successivamente alla sua invocazione, vengono riassunte nel diagramma di flusso di figura 6.9. Riguardo a tale diagramma è interessante notare che l'operazione di verifica dell'esistenza della reservation viene effettuata utilizzando un `ClientId` che è già stato verificato (si ricorda che il `RsvId` è presente all'interno di un contratto valido). Si fa notare inoltre come possa esistere una nota della reservation in cui il `RsvId` non sia valido: ciò non costituisce un errore ma semplicemente corrisponde al caso, cui si è fatto riferimento in precedenza, in cui l'ultima reservation che è stata utilizzata per servire quel client non esiste più nel sistema.

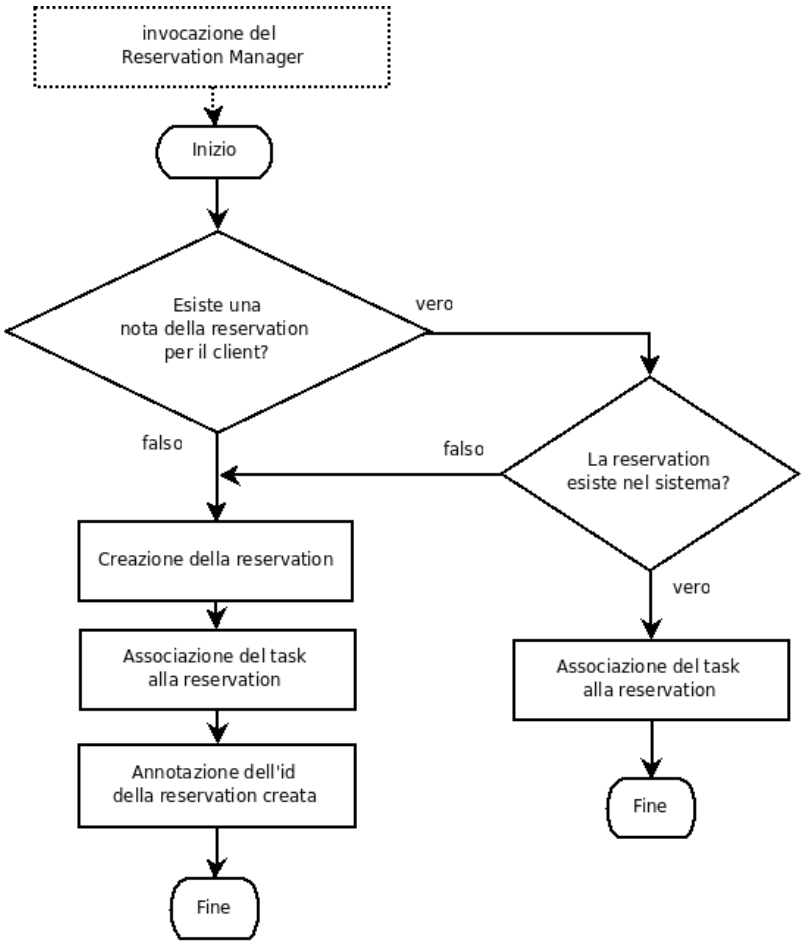

Figura 6.9: Diagramma di flusso operazioni Reservation Manager

6.3 Dettagli di sviluppo

In questa sezione verranno descritti i dettagli più significativi relativi allo sviluppo dell'architettura in esame. Inizialmente si analizzerà il modello dei prototipi di accordo, tramite cui si descrivono i servizi forniti e si permette ad un consumatore di contrattare la QoS desiderata. In seguito verrà descritto come si è realizzata la comunicazione tra il livello di accordo ed il livello di servizio del sistema ed infine verranno esposti alcuni dettagli sulla realizzazione del livello di servizio.

6.3.1 Prototipi di accordo forniti

Si è visto come il processo di creazione di un accordo prevede che il componente WebAgreementFactory debba fornire i prototipi di accordo da cui verranno generate le proposte di accordo effettuate dal consumatore. I prototipi di accordo forniti sono generati secondo le specifiche WS-Agreement, dunque nell'esporre il loro modello si farà riferimento alla struttura di Agreement e Agreement Template, discusse nella sezione 2.2.

L'utilizzo di un prototipo permette di realizzare due delle specifiche del sistema. Tramite un prototipo si descriveranno infatti le caratteristiche del sistema che fornisce il servizio e si dovrà permettere al consumatore di specificare la banda di CPU come parametro di QoS contrattatabile. Per poter descrivere le caratteristiche del sistema fornitore si utilizza il linguaggio JSDL([18]) all'interno della sezione Terms di un prototipo. Di seguito viene riportato il frammento XML relativo.

```
<wsag:ServiceDescriptionTerm wsag:Name="server_description"
wsag:ServiceName="use_of_web_server">
    <jsdl:JobDescription
    xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl">
        <jsdl:Application>
            <jsdl:ApplicationName>
                Apache
            </jsdl:ApplicationName>
            <jsdl:ApplicationVersion>
                2.2
            </jsdl:ApplicationVersion>
        </jsdl:Application>
        <jsdl:Resources>
            <jsdl:OperatingSystem>
                <jsdl:OperatingSystemType>
                    <jsdl:OperatingSystemName>
                        LINUX
                    </jsdl:OperatingSystemName>
                </jsdl:OperatingSystemType>
            </jsdl:OperatingSystem>
            <jsdl:IndividualCPUSpeed>
                    <jsdl:Exact>1200</jsdl:Exact>
            </jsdl:IndividualCPUSpeed>
```

```
        </jsdl:Resources>
    </jsdl:JobDescription>
</wsag:ServiceDescriptionTerm>
```

Listing 6.1: Descrizione del sistema all'interno di un prototipo

Per poter invece permettere di specificare la banda del processore che ogni consumatore può contrattare, si utilizza invece uno schema XML da noi definito. In particolare, viene definito un elemento `ServerParameters`, al cui interno può essere inserito un elemento `CpuBandwith` che indica la banda desiderata. Nel prototipo tali elementi compaiono anch'essi nella sezione Terms, come indicato di seguito.

```
<wsag:ServiceDescriptionTerm wsag:Name="server_parameters"
wsag:ServiceName="use_of_web_server">
   <ret:ServerParameters xmlns:ret="schemas.retis">
      <ret:CpuBandwith> </ret:CpuBandwith>
   </ret:ServerParameters>
</wsag:ServiceDescriptionTerm>
```

Listing 6.2: Descrizione del servizio all'interno di un prototipo

Il fatto che tale parametro di QoS sia contrattabile è indicato nella sezione `CreationConstraints` del prototipo, come indicato di seguito.

```
<wsag:CreationConstraints>
  <wsag:Item wsag:Name="CpuBandwithItem">
    <wsag:Location>
    //wsag:ServiceDescriptionTerm/ret:ServerParameters/ret:CpuBandwith
    </wsag:Location>
    <wsag:ItemConstraint>
      <xs:simpleType xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:restriction base="xs:decimal">
          <xs:maxInclusive value="1"/>
        </xs:restriction>
      </xs:simpleType>
    </wsag:ItemConstraint>
  </wsag:Item>
</wsag:CreationConstraints>
```

Listing 6.3: Vincoli di creazione in un prototipo

I prototipi di accordo vengono generati, sulla base del modello di cui abbiamo appena esposto le sezioni più significative, dal WebAgreementFactory ogni volta che viene invocato il metodo *getTemplates()*. La realizzazione del WebAgreementFactory, così come del WebAgreement, è stata effettuata tramite l'utilizzo delle API fornite dal WSAG4J Framework ([21]): tale framework fornisce infatti una implementazione in Java di gran parte delle specifiche WS-Agreement, aggiornate all'ultima versione disponibile.

6.3.2 Comunicazione tra i livelli di accordo e di servizio

La comunicazione tra il livello di accordo ed il livello di servizio avviene per mezzo di memoria condivisa in cui vengono registrati i contratti validi. In

questo modo il livello di servizio può accedere in lettura ai contratti validi, che vengono invece creati e distrutti ad opera del BookingAgent (vedi sezione 6.2.4).

La struttura dati che contiene i contratti è contenuta in un file che non risiede sul disco fisso, bensì in una zona di memoria condivisa. Ciò è reso possibile dall'utilizzo di un file system di tipo *tmpfs*, il quale viene messo a disposizione dal kernel Linux (si veda, all'interno dell'albero dei sorgenti del kernel, il file `Documentation/filesystems/tmpfs.txt`). Il file condiviso è un file di database di tipo SQLite3 (www.sqlite.org): tale libreria software è stata utilizzata poiché permette di sfruttare le potenzialità di un database relazionale senza utilizzare un'architettura client-server. Un database di questo tipo è contenuto dunque in un unico file e può essere acceduto senza che sia necessario attendere lunghi tempi per la connessione. Tale soluzione risulta dunque adatta al nostro caso, poiché permette di usare un database SQL senza un eccessivo sovraccarico per funzionalità di cui non si necessita. Nell'architettura realizzata, l'uso di un database SQL permette di risolvere facilmente il problema della ricerca di un valore all'interno dell'insieme dei contratti validi. Tali ricerche vengono fatte dai componenti di entrambi i livelli, sul valore del `ClientId`. A livello di accordo, esse devono essere fatte per assicurarsi che non si stia contrattando con un consumatore che ha già stipulato un contratto; a livello di servizio, vengono effettuate durante la fase di intercettamento delle richieste (si veda la sezione 6.2.5) per verificare che il richiedente del servizio abbia un contratto valido. Inoltre, l'uso di un database SQL permette di risolvere facilmente l'operazione di cancellazione di un contratto valido, operazione che deve essere svolta dal BookingAgent.

6.3.3 Realizzazione del livello di servizio

Si è detto come il Rt Module sfrutti le funzionalità del web server per poter ricevere ed elaborare le richieste di servizio. In particolare, il web server utilizzato in questo lavoro è Apache2: il Rt Module è dunque stato realizzato come un modulo di Apache2. All'interno di tale modulo si utilizzeranno poi le funzionalità messe a disposizione da AQuoSA per garantire il rispetto della QoS contrattata con il consumatore.

Nel descrivere la realizzazione del Rt Module si faranno dunque numerosi riferimenti al funzionamento, alla struttura e alle caratteristiche di Apache2 (descritte nel capitolo 4). Si farà altresì riferimento alla trattazione effettuata nel capitolo 5 sull'estensione di tale server web, nonché alla trattazione del capitolo 3 su AQuoSA.

Configurazione del Rt Module

Si è detto in fase di progetto che una delle funzionalità del `Rt Module` è quella di fornire possibilità di configurazione: tale funzionalità può essere

realizzata utilizzando le tecniche di cui si è discusso nella sezione 5.2.2.

Per rappresentare la configurazione si è utilizzata la struttura dati che riportiamo di seguito.

```
typedef struct{
    int active;
    int use_ag;
    char* max_cpu_bw;
    apr_array_header_t* handlers_list;
    sem_t* s_write;
}rt_server_conf;
```

Listing 6.4: Struttura dati che rappresenta la configurazione

Visto che la configurazione influenza il comportamento dell'intero server, essa avrà un contesto per-server e la corrispondente funzione di creazione sarà registrata utilizzando l'apposita entrata "creazione della struttura per-server" nella `module structure`. Tale funzione viene indicata di seguito.

```
static void* create_rt_config(apr_pool_t* p, server_rec* s){
  rt_server_conf* c;
  c=(rt_server_conf*) apr_pcalloc(p, sizeof(rt_server_conf));
  c->active = 0;
  c->use_ag = 0;
  c->max_cpu_bw = DEFAULT_BW;
  c->handlers_list = apr_array_make(p,INIT_ELEM,sizeof(char*));
  c->s_write=NULL;
  return c;
}
```

Listing 6.5: Creazione della struttura per-server

I comandi che il modulo deve riconoscere vengono invece definiti dalla seguente funzione, che dovrà essere registrata anch'essa nella `module structure`, utilizzando l'entrata "gestore di comandi". I metodi che vengono invocati per ogni comando non sono riportati per brevità, essi comunque utilizzano i parametri di ogni direttiva in accordo a quanto riportato nella sezione 6.2.5.

```
static const command_rec mod_rt_cmds[] = {
   AP_INIT_FLAG("RtModule", modrt_active, NULL, RSRC_CONF,
     "set to On/Off to activate or deactive the module"),

   AP_INIT_FLAG("RtUseAgreement", modrt_use_ag, NULL, RSRC_CONF,
     "set to On/Off to use WS-Agreement for negotiation"),

   AP_INIT_TAKE1("RtCpuBwReservation",modrt_set_bw,NULL,OR_ALL,
     "the max cpu bandwith, using a number in the range [0-1])",

   AP_INIT_ITERATE("RtSetHandler",modrt_set_handler,NULL,RSRC_CONF,
     "handlers to execute in rt (es.cgi-script)"),
   {NULL}
};
```

Listing 6.6: Riconoscimento delle direttive

Intercettazione richieste e gestione QoS

Il codice eseguibile del *Rt Module*, come quello di qualsiasi modulo di Apache, viene incluso non solo all'interno del master server, ma anche all'interno dei child server. Ogni child server esegue dunque la fase di intercettazione delle richieste e l'eventuale gestione delle risorse virtuali con le quali si garantisce la QoS. Nell'architettura preforking, utilizzata in questo lavoro per le sue garanzie di stabilità, ogni child server si occupa di ricevere le richieste che andrà a gestire: dunque se un child server si accorge di dover servire richieste con QoS, egli stesso dovrà provvedere (utilizzando le funzionalità del Reservation Manager) a creare la risorsa virtuale da utilizzare. Per quanto si è detto in fase di progetto (sezione 6.2.5), lo hook più appropriato in cui effettuare l'intercettazione della richiesta è lo hook `pre_connection`, per il quale è stata registrata la funzione *rt_service_per_client*. La fase di intercettazione, effettuata appunto in tale funzione, viene brevemente esposta nel listato 6.7. In esso si può notare l'utilizzo di un semaforo di mutua esclusione, condiviso tra il BookingAgent e i child server del RtModule, per garantire la consistenza della base di dati in cui vengono memorizzati i contratti validi.

```
static int rt_service_per_client(conn_rec* c, void* csd){
    ...

    char* client_id = idf_discover_client(c);
    sem_wait(s_cfg->s_write);
    sqlite3* db;
    int rc = sqlite3_open(DB_PATH, &db);
    if(rc) db_open_error(db);
    char* cpu_bw = retrieve_cpubw(client_id, db);

    //do reservation only if we find a contract for the client
    if (cpu_bw == NULL)
        return DECLINED;

    ...
```

Listing 6.7: Intercettazione delle richieste

Successivamente il controllo passa al Reservation Manager, che realizza le sue operazioni (si veda figura 6.9) nella seguente maniera.

```
    ...
    sid = mng_retrieve_rsvid(client_id, db);
    if (sid == -1){
        sid = mng_create_rsv_and_attach(cpu_bw);
        mng_trace_srvid(client_id, sid, db);
    }
    else{
        rv = qres_attach_thread(sid, 0, 0);
        if (rv != QOS_OK){
            sid = mng_create_rsv_and_attach(cpu_bw);
            mng_trace_rsvid(client_id, sid, db);
         }
```

```
        else{
            //we are attached
        }
    }
    db_disconnect(db);
    sem_post(s_cfg->s_write);
    ...
```

Listing 6.8: Operazioni del ReservationManager

E' significativo, inoltre, osservare, tramite il listato 6.9, come viene realizzato il metodo *mng_create_srv_and_attach*, poiché è quello tramite il quale ciascun child server crea il server CBS che rappresenta la reservation e vi si attacca (vedi anche sezione 3.3.2).

```
qres_sid_t mng_create_srv_and_attach(char* cpu_bw){
    qres_sid_t sid;
    double bw = atof(cpu_bw);
    qos_bw_t s_bw = d2bw(bw);
    qres_time_t s_Q = bw2Q(s_bw, QRES_P);
    qres_params_t params = {
        .Q_min = 0,
        .Q = s_Q,
        .P = QRES_P,
        .flags = 0,
    };
    qres_create_server(&params, &sid);
    qres_attach_thread(sid, 0, 0);
    return sid;
}
```

Listing 6.9: Creazione delle risorse virtuali

Si ribadisce inoltre che il tempo di vita dell'associazione tra un task ed una reservation deve avere la stessa durata di una connessione. Per garantire tale comportamento è sufficiente che la funzione *qres_detach_thread* venga associata al `connection pool` tramite il metodo *apr_pool_cleanup_register* (si veda la sezione 4.3.2 sulla gestione delle risorse in Apache).

Infine si deve notare come, per poter utilizzare le funzioni di libreria di AQuoSA, è necessario provvedere preventivamente all'inizializzazione della libreria tramite la *qres_init*. Inoltre, quando non è più necessario invocare metodi della libreria si deve provvedere alla chiamata della *qres_cleanup*. Tali operazioni possono essere effettuate registrando un metodo, che chiameremo *rt_init*, per lo hook `post_config`: all'interno di tale metodo si dovrà provvedere all'invocazione della qres_init. Poiché il metodo *rt_init* viene eseguito ogni volta che viene letta la configurazione del server, risulta naturale associare il metodo di pulizia qres_cleanup con il `pconf pool`, in modo tale che venga invocato lo stesso numero di volte della qres_init.

Capitolo 7

Esperimenti e risultati

L'architettura realizzata in questo lavoro deve permettere al soggetto fornitore di erogare i propri servizi rispettando le garanzie che sono state promesse in fase di contrattazione. Risulta dunque particolarmente importante verificare, con degli esperimenti, che la nostra architettura permette di raggiungere questo obiettivo. In particolare siamo interessati a verificare il comportamento del livello di servizio dell'architettura, poiché è il livello in cui avviene la gestione della QoS.

In questo capitolo, dunque, verranno descritti gli esperimenti effettuati in tal senso ed in seguito verranno analizzati i risultati che sono stati ottenuti. Si dimostrerà in tal modo che è possibile garantire una certa qualità del servizio, soprattutto in condizioni di elevato carico del sistema, solamente se si utilizzano delle opportune tecniche di gestione della QoS, come è stato fatto nella nostra architettura.

7.1 Descrizione degli esperimenti

In questa sezione verranno descritti gli esperimenti che sono stati effettuati. In particolare verrà prima esposto quale è la configurazione dell'ambiente che è stato utilizzato per effettuare gli esperimenti, in seguito si descriverà il tipo degli esperimenti ed infine si individueranno i parametri che saranno oggetto di variazioni nel corso della sperimentazione.

7.1.1 Ambiente utilizzato

Gli esperimenti sono stati condotti tramite l'utilizzo di due macchine, una client e l'altra server, connesse in maniera diretta tramite un cavo di rete Ethernet a 100Mbps. Si è usato una macchina server equipaggiata con un processore a 1600Mhz, una memoria RAM di 740MB ed un sistema operativo Debian GNU/Linux 4.0; la macchina client è stata invece equipaggiata con

un processore a 1700Mhz, una memoria RAM di 512 MB ed un sistema operativo Debian GNU/Linux 4.0.

Nella macchina server è stata installata la nostra architettura. In particolare a livello di servizio si è inserito il componente `Rt Module`, opportunamente configurato, all'interno del server web Apache 2.2. L'inclusione del modulo nel web server è stata fatta in maniera dinamica. Apache 2.2 è stato configurato in maniera che i task serventi di partenza fossero 10, per il resto la configurazione che è stata usata è quella di default.

Nella macchina client si è invece installato un programma per effettuare il benchmark di un server web. I programmi di questo tipo inoltrano un certo numero di richieste HTTP e tramite la misura dei tempi di risposta del server riescono a dare una valutazione delle sue prestazioni. Nei nostri esperimenti si è utilizzato **Apache ab**([25]), uno strumento per il benchmark dal semplice utilizzo, che permette di usufruire di funzionalità essenziali in questo tipo di test: simulazione della concorrenza, utilizzo di connessioni keep-alive, salvataggio di report. In confronto ad altri tool di benchmark (segnaliamo fra tutti Jmeter), *ab* manca di caratteristiche avanzate, come quelle che permettono di costruire complessi scenari di utilizzo del web server. Tuttavia, se si considerano i report sulle prestazioni misurate, si nota come *ab* regge molto bene il confronto con gli altri tool poiché fornisce una notevole quantità di informazioni significative. Inoltre, tali informazioni sono riportate in maniera da poter essere importate in diversi strumenti di analisi: in questo modo *ab* raggiunge, a nostro avviso, una flessibilità di utilizzo maggiore rispetto a quella permessa da altri tool. Nel nostro caso, poiché lo schema delle richieste di servizio da inoltrare al server è abbastanza lineare, *ab* risulta dunque un'ottima scelta.

L'utilizzo di due macchine per effettuare gli esperimenti viene dettato dalla considerazione che, per valutare in maniera accurata le prestazioni del server, si deve "liberare" la macchina su cui il server viene eseguito dal peso computazionale relativo all'invio delle richieste. Infatti, in base al tipo di simulazione effettuata, tale peso è non trascurabile e deve comunque essere tenuto in considerazione, per evitare che le prestazioni misurate siano quelle del tool di benchmark e non quelle del server.

7.1.2 Tipo di esperimenti

Nel corso di questi esperimenti si è interessati a valutare il comportamento del server web soprattutto quando ci si trova in condizioni di elevato carico nel sistema. Tale carico viene generato sinteticamente tramite un programma denominato `test_load` (vedi listato 7.1). Il programma `test_load`, che tende a consumare tutto il tempo di CPU che gli viene concesso, viene lasciato in esecuzione durante la durata di ogni singolo esperimento.

```
struct timeval tv1, tv2;
while (1) {
```

```
    for (i = 0; i < N; i++) {
    x = x * 1773754;
    }
    gettimeofday(&tv1, NULL);
    long delay = (long) (usec * rand() / (RAND_MAX + 1.0));
    usleep(delay);
    gettimeofday(&tv2, NULL);
    unsigned long usec2;
    usec2=(tv2.tv_sec-tv1.tv_sec)*1000000+tv2.tv_usec-tv1.tv_usec;
}
```

Listing 7.1: Programma di generazione del carico

Per ogni esperimento, lo scenario che si vuole simulare è quello in cui il web server riceve contemporaneamente richieste da 10 diversi client. Ogni client invia un numero totale di 100 richieste al server web: una richiesta viene inviata dopo che è arrivata la risposta relativa alla precedente. Tale comportamento viene reso possibile utilizzando *ab* con le seguenti opzioni:

```
ab -c 10 -n 100
```

La concorrenza viene effettuata da *ab* semplicemente instaurando con il server un numero di connessioni TCP contemporanee che è pari al numero di client da simulare.

La richiesta di servizio effettuata da tutti i client simulati nel corso degli esperimenti, consiste nel richiedere l'esecuzione di un programma con interfaccia CGI. Si è scelto un programma che fosse *CPU bound*, visto che il livello di supporto alla gestione della QoS ci offre garanzie, sui tempi di esecuzione, solo per quanto riguarda l'uso del processore. Il programma usato per i test effettua particolari elaborazioni matematiche in memoria, simulando un'operazione di rotazione di 20° di un'immagine le cui dimensioni sono di 4 Mpixel. Il cuore di tale elaborazione è rappresentato dalla formula seguente.

$$\begin{cases} x = x_p \cos(\theta) - y_p \sin(\theta) \\ y = y_p \cos(\theta) + x_p \sin(\theta) \end{cases} \tag{7.1}$$

Tale formula, che permette di calcolare per ogni pixel dell'immagine finale da costruire le coordinate dell'immagine iniziale di partenza, deve essere iterata per ogni pixel dell'immagine finale (di dimensioni superiori a quelle dell'immagine iniziale). Il programma CGI che ne risulta ha un tempo di esecuzione di circa pari ad 1.4 secondi. L'esecuzione di una richiesta di servizio di questo tipo si traduce in un'unica richiesta HTTP di tipo GET inoltrata dal client al server, in cui l'URI richiesta è proprio il nome del programma CGI, che abbiamo chiamato `simulaRotazione`.

La definizione del tipo di esperimenti da effettuare, apre alcune considerazioni riguardo alla compatibilità del modulo Rt con gli esperimenti stessi.

Il fatto che i diversi client vengano simulati da parte del programma di benchmark è solitamente trasparente al server web e non viene generalmente preso in considerazione nei normali test (in cui solitamente si è interessati al *throughput* del server). Questo è effettivamente quello che si farà quando si condurranno gli esperimenti sul server web originale. Nel caso in cui si utilizza la nostra architettura, si deve però tenere in considerazione che il modulo Rt si basa sull'identificazione dei diversi client, per poter fornire loro la qualità del servizio che hanno contrattato. Tale identificazione, visto che le richieste sono tutte inviate tramite il programma di benchmark, risulta dunque impossibile da effettuare nello stesso modo definito in fase di progetto. Per poter simulare l'identificazione dei client si è dunque ricorso, solamente nel corso degli esperimenti, ad una leggera modifica del *Rt Module*, la quale prevede che l'instaurazione di ogni nuova connessione venga riconosciuta come effettuata da un diverso client, in maniera consistente al funzionamento di *ab*. Tale modifica, sebbene porti ad un cambiamento nel comportamento del *Rt Module*, non inficia invece sulla bontà degli esperimenti per come essi sono stati definiti: negli esperimenti siamo infatti interessati a misurare le prestazioni del server per ogni singola richiesta di servizio piuttosto che per client. Il significato che la simulazione della concorrenza assume è dunque quello di aggiungere complessità nelle elaborazioni che deve effettuare il server. Inoltre, poiché siamo interessati solamente alle prestazioni del livello di servizio dell'architettura, si assume che, al momento di una richiesta di servizio, la contrattazione sia già stata avvenuta e dunque i dati sui contratti validi sono già a disposizione del *Rt Module*. Durante un esperimento, ad ogni client simulato sarà associato un contratto, il cui valore per il campo `CpuBw` (vedi sezione 6.2.4) è lo stesso per tutti i client.

7.1.3 Parametri di test

Nel corso degli esperimenti si vuole innanzitutto testare la differenza di comportamento tra il server web originale ed il server web con il *Rt Module* inserito. Un insieme di esperimenti riguarda dunque il caso in cui le richieste vengono servite dal server web originale. Un'altro insieme di esperimenti è invece relativo al caso in cui le richieste vengono servite con gestione di qualità del servizio. I due insiemi devono essere indipendenti, dunque solo un server è attivo per ogni insieme di esperimenti.

Per quanto riguarda gli esperimenti in cui viene effettuata la gestione della QoS da parte del Rt Module, essi dipendono da diversi parametri, al variare dei quali siamo interessati a verificare qual è il comportamento del server. I parametri considerati riguardano le *reservation* (si veda la sezione 3.3.1), su cui si basa la gestione della QoS fatta dal *Rt Module* per mezzo di AQuoSA.
Innanzitutto consideriamo il periodo P della reservation. La valutazione di tale parametro è interessante poiché la schedulazione RB, sottostante

al Rt Module, tende ad approssimare un'allocazione tanto più fluida della CPU quanto minore è il periodo della reservation. In pratica però, si ha che il sovraccarico dovuto alle commutazioni di contesto dei task diventa maggiormente rilevante quanto più P è piccolo. Vengono dunque effettuate due serie di esperimenti: una in cui $P = 10ms$, l'altra in cui $P = 100ms$. Inoltre ricordiamo che la schedulazione RB può essere eseguita correttamente solo se si verifica la relazione di consistenza relativa all'algoritmo utilizzato. Nel caso specifico dell'algoritmo utilizzato da AQuoSA tale relazione assume la seguente forma

$$\sum_i B_i \leq 1 \tag{7.2}$$

dove B_i è la banda di ogni CPU virtuale. E' dunque interessante variare come parametro la banda contrattata da ogni consumatore. In particolare, siamo interessati a verificare la differenza di comportamento tra il caso in cui la relazione di consistenza è verificata ed il caso in cui non lo è. Poiché nel corso di un esperimento vengono simulati 10 client concorrenti ed ognuno di essi ha associata la stessa banda, viene effettuata una serie di esperimenti in cui si verifica la relazione di consistenza utilizzando $B_i = 0.09$ ed un'altra serie in cui invece si utilizza un valore di $B_i = 0.4$ che non verifica la relazione.

7.2 Analisi dei risultati

I dati forniti da *Apache ab* come risultato delle operazioni di benchmark sono di diverso tipo. Quelli che più ci interessano riguardano la misurazione del tempo di vita di una connessione: essi misurano, per ciascuna connessione, il tempo che intercorre tra l'apertura di una connessione e la sua chiusura a seguito della risposta del server. Poiché *ab* si trova sulla macchina client, i tempi misurati sono quelli che verrebbero percepiti da un soggetto consumatore. Con l'utilizzo di *ab* è possibile eliminare dalla valutazione i tempi necessari ad instaurare una connessione e considerare solamente il tempo in cui una connessione viene processata. Ciò è importante perché i test sono stati condotti in maniera tale che ad ogni richiesta di servizio corrispondesse l'instaurazione di una connessione TCP tra client e server: i dati forniti da *ab*, riguardo ai tempi di processamento di una connessione, coincidono dunque con i tempi di esecuzione del servizio. Si ribadisce che tale tempo è quello in cui un soggetto consumatore vede eseguito il servizio richiesto.

Per ognuno dei tipi di esperimenti individuati nella sezione precedente, sono state effettuate 20 ripetizioni di ogni singolo esperimento. Si sono individuati come parametri di rilevazione da analizzare i tempi di processamento minimi, massimi e medi, oltre che la deviazione standard, utile per verificare di quanto oscillano, rispetto al tempo medio di processamento, i tempi di tutte le richieste. Per ognuno di tali parametri si è calcolato il valor

medio relativo al totale delle 20 ripetizioni ed è stata verificata la bontà di tale risultato, calcolando l'intervallo di confidenza al 90%: in questo modo possiamo verificare qual è l'intervallo in cui ricade il 90% dei risultati ottenuti.

7.2.1 Rilevazioni per il server originale

Si inizierà l'analisi dei risultati partendo dalle rilevazioni effettuate per le richieste inviate al web server non modificato. Identificheremo tale insieme di esperimenti come caso 1. I risultati ottenuti sono riportati nella tabella 7.1.

Tabella 7.1: Tempi di processamento rilevati per il web server originale

Tempo di processamento	Valore medio (s)	Intervallo di confidenza al 90% (s)
min	0.726	[0.712, 0.741]
medio	12.001	[10.122, 13.881]
max	66.057	[58.162, 73.952]
dev.std	17.585	[15.144 , 20.026]

L'osservazione dei tempi minimi, medi e massimi suggerisce una forte variabilità nei tempi di processamento ottenuti. Infatti, si va da un tempo minimo di servizio inferiore al secondo ad un tempo massimo di servizio di 66 secondi. Tale oscillazione nel range di valori non è occasionale, poiché vi è un valore molto alto per la deviazione standard. Il fatto che essa sia pari a circa 17 secondi ci indica che i tempi di processamento differiscono mediamente di tale valore rispetto al tempo di processamento medio. Per poter visualizzare tale concetto si possono rappresentare con un grafico i tempi di processamento relativi ad un singolo esperimento (vedi figura 7.1). Il grafico mostra in ascisse le richieste, ordinate per tempo di inizio e nelle ordinate i tempi di processamento di ogni richiesta. L'esperimento cui si è fatto riferimento nel grafico risulta particolarmente significativo poiché i suoi tempi di processamento sono molto vicini a quelli medi e comunque all'interno degli intervalli di confidenza. Tali tempi sono, per quanto riguarda minimo, media, massimo e deviazione standard, rispettivamente pari 0.709s, 12.230s, 68.986s, 19.058s.

L'analisi dei risultati ottenuti per questo insieme di esperimenti ci porta a concludere che con un normale server web è impossibile, in condizioni di carico elevato del sistema, assicurare una certa qualità del servizio: alcuni utenti infatti saranno particolarmente soddisfatti, per i bassi tempi di servizio, altri invece non lo saranno affatto per i tempi eccessivamente elevati.

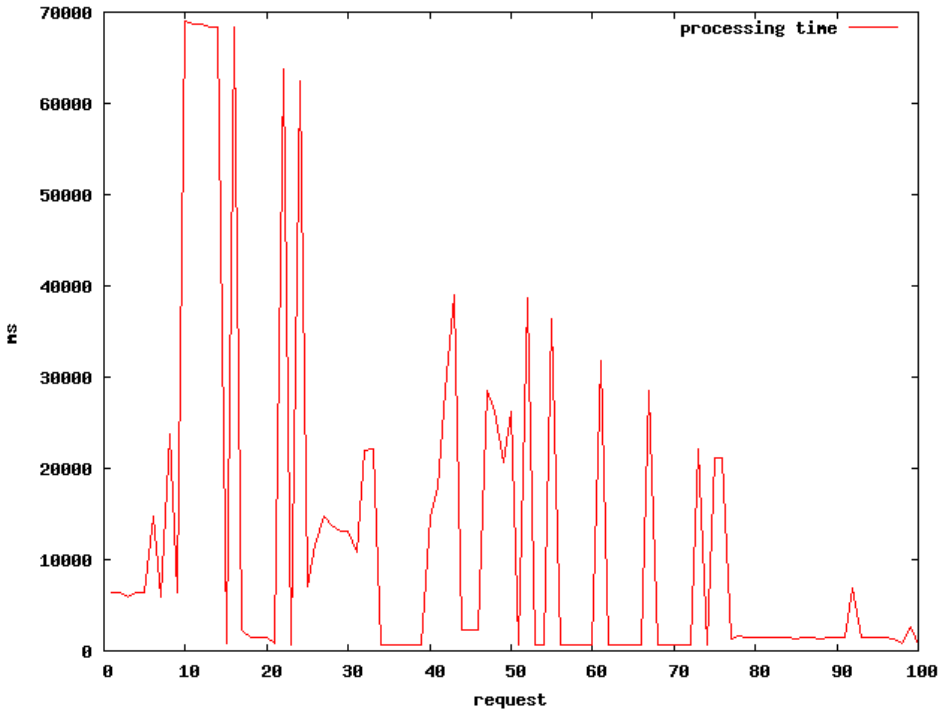

Figura 7.1: Tempi di processamento relativi al server web originale

7.2.2 Rilevazioni per il server con il Rt Module

L'analisi dei risultati viene proseguita con i dati relativi alle richieste in cui vi è gestione della QoS, effettuata tramite il *Rt Module* incluso nel server web.

Inizialmente vengono confrontati i dati ottenuti quando il Rt Module è stato configurato con periodo di reservation pari a 10ms ed i dati ottenuti con un periodo di reservation pari a 100ms (in entrambi i casi la banda di ogni reservation è $B = 0.09$, valore che rispetta la relazione di consistenza di cui sopra). Riportiamo in tabella 7.2 i dati ottenuti, relativi ai valori medi dei parametri rilevati, calcolati sul totale delle 20 ripetizioni per ciascun tipo di esperimento.

Tabella 7.2: Confronto tra i tempi di processamento rilevati per il web server con il modulo Rt (reservation con $B = 0.09$)

	min (s)	medio (s)	max (s)	dev.std (s)
$P = 10ms$	7.150	7.474	9.277	0.492
$P = 100ms$	7.902	7.955	8.292	0.078

Dall'analisi di tali dati si nota come i tempi di risposta minimi e medi siano leggermente più bassi nel caso di reservation con periodo $P = 10ms$. Ciò è dovuto al fatto che un periodo di reservation basso permette ai task che devono servire le richieste di essere maggiormente responsivi. D'altra parte si nota come, sempre nel caso di periodo uguale a 10ms, il tempo massimo e la deviazione standard abbiano valori superiori, il che indica una maggiore oscillazione dei risultati attorno al tempo medio: ciò è dovuto al sovraccarico generato dai numerosi cambi di contesto necessari per simulare una allocazione fluida del processore.

Poiché siamo interessati più che altro ad ottenere tempi di servizio costanti, nel proseguo utilizzeremo il caso di $P = 100ms$ e $B = 0.09$ come base di confronto per gli esperimenti in cui si gestisce la qualità del servizio. Identificheremo tale insieme come caso 2. Analizziamo dunque in maniera approfondita i dati relativi a tale insieme, riportandoli in maniera completa nella tabella 7.3. Da tali dati si nota come i tempi di processamento di una richiesta siano in questo caso praticamente costanti: i valori minimi, medi e massimi sono molto vicini e la deviazione standard ha un valore relativamente molto basso. Si può osservare inoltre l'accuratezza dei risultati ottenuti, dedotta dalla ristrettezza dei vari intervalli di confidenza.

Tabella 7.3: Tempi di processamento per il web server con il modulo Rt (reservation con $P = 100ms$ e $B = 0.09$)

Tempo di processamento	Valore medio (s)	Intervallo di confidenza al 90% (s)
min	7.902	[7.895, 7.908]
medio	7.955	[7.945, 7.965]
max	8.292	[8.155, 8.429]
dev.std	0.078	[0.047, 0.109]

Anche in questo caso si darà rappresentazione grafica del comportamento del server web (vedi figura 7.2) utilizzando un esperimento altamente significativo, i cui valori per i tempi di processamento minimi, medi, massimi e per la deviazione standard sono pari a 7.907, 7.953, 8.119, 0.053.

Per poter confrontare la differenza di comportamento del server tra il caso in cui si gestisce la QoS e il caso in cui non si gestisce, useremo proprio quest'ultimo esperimento (dati di tabella 7.3) in relazione a quello relativo alla tabella 7.1. Il confronto tra i dati di tali tabelle, mette in luce come i tempi di servizio con gestione della QoS siano praticamente costanti, mentre quelli in cui la QoS non viene gestita possono essere, nel caso minimo inferiori di circa 7 secondi, nel caso massimo superiori di circa 60 secondi. Il tempo di servizio medio è invece molto vicino nei due casi ed è superiore nel caso di assenza di gestione solo per circa 4 secondi. La differenza tra i due

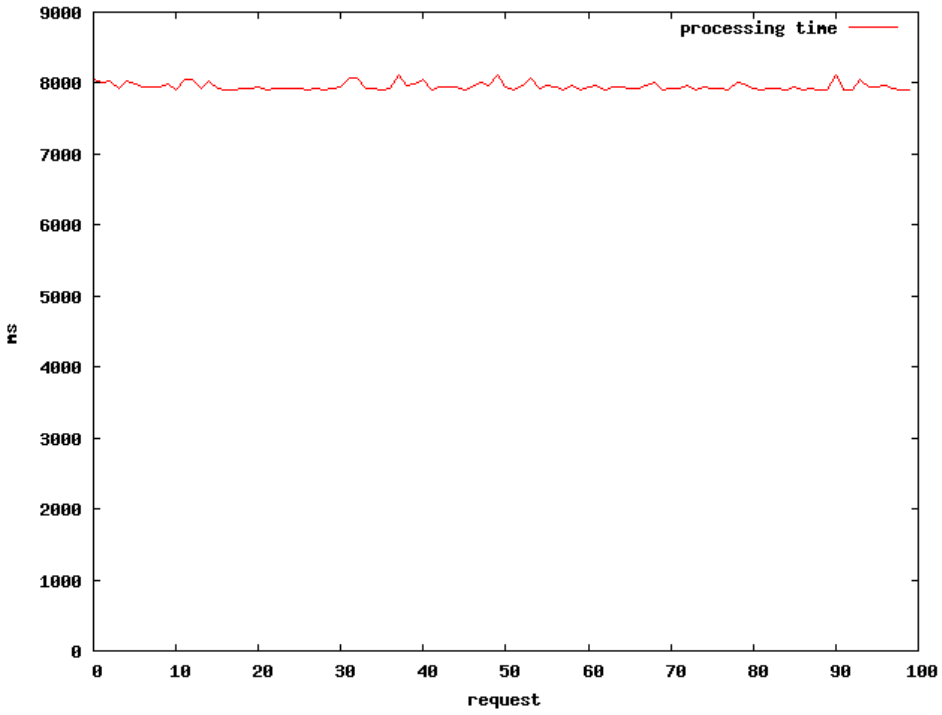

Figura 7.2: Tempi di processamento con il modulo Rt inserito nel server

comportamenti può essere apprezzata in figura 7.3, in cui si riportano, nella stessa scala, i due grafici già analizzati.

Il fatto che il tempo di servizio medio nel caso di assenza di QoS sia molto simile al caso con QoS, non deve trarre in inganno sulla possibilità di fare a meno della gestione della QoS. Infatti si è visto che la forte oscillazione dei tempi di processamento del caso senza QoS determinerà un diverso grado di soddisfazione degli utenti ed inoltre rende praticamente impossibile una stima sulla QoS che il server è in grado di garantire. L'utilizzo di un meccanismo di gestione della QoS permette invece di avere isolamento temporale tra i task, per cui è come se ogni richiesta di servizio fosse eseguita su una CPU dedicata: dato che ognuna di essa ha la stessa velocità si ottengono tempi di processamento costanti. Viceversa, nel caso senza QoS, ogni richiesta di processamento può essere influenzata dai comportamenti temporali degli altri task del sistema e questo spiega le forti impennate nei tempi di servizio visibili dalla figura 7.3.

Un'ulteriore serie di dati da analizzare riguarda il caso in cui si utilizza la gestione della QoS ma la relazione di consistenza 7.2 risulta violata. Si utilizzeranno infatti come parametri della reservation $P = 100ms$ e $B = 0.4$: dato che i client che si connettono in contemporanea sono 10, la relazione di consistenza verrebbe violata se si dovessero rispettare tali parametri nella

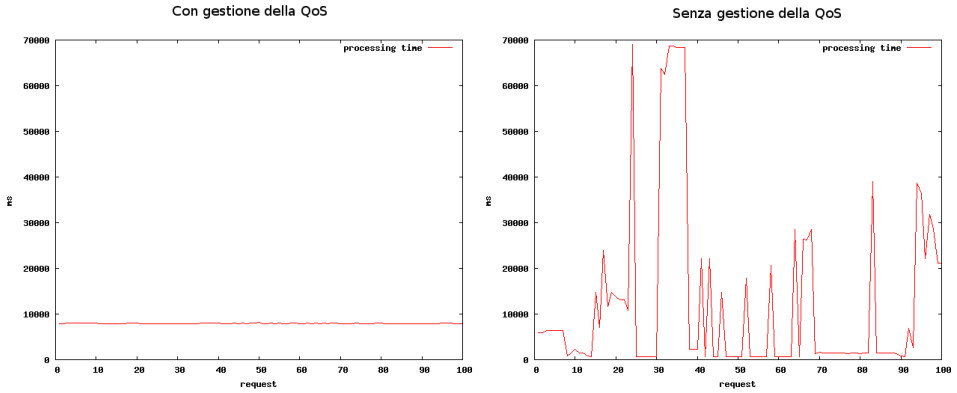

Figura 7.3: Confronto tra server con QoS e server senza QoS

creazione delle reservation. Poiché AQuoSA non permette la creazione di una reservation se essa comporta una violazione della relazione di consistenza, si avrà che alcuni consumatori verranno effettivamente serviti con una reservation ($P = 100ms$, $B = 0.4$), altri consumatori invece verranno serviti senza gestione della QoS. Dall'osservazione dei dati ottenuti, riportati nella tabella 7.4, possiamo notare come i tempi di processamento variano da 2.8s a 15.3s, con una deviazione standard consistente, pari a 2.14s. I tempi di processamento, dunque, non sono costanti come nel caso in cui si rispetta la relazione di consistenza (caso 2, tabella 7.3). Questo avviene poiché il meccanismo di gestione della QoS non può eseguire correttamente e dunque l'isolamento temporale tra i task non viene più garantito in maniera completa. La variabilità in tal caso è comunque maggiormente contenuta rispetto a quella che si ha in completa assenza di gestione della QoS (caso 1, tabella 7.1): ciò dimostra come sia possibile, utilizzando la gestione della QoS, poter fornire delle garanzie (sebbene meno precise) anche nel caso in cui il numero di utenti serviti contemporaneamente porti ad una violazione della relazione di consistenza.

Tabella 7.4: Tempi di processamento per il web server con il modulo Rt (reservation con $P = 100ms$ e $B = 0.4$)

Tempo di processamento	Valore medio (s)	Intervallo di confidenza al 90% (s)
min	2.821	[2.718 , 2.925]
medio	7.804	[7.718 , 7.891]
max	15.315	[14.420 , 16.210]
dev.std	2.142	[2.026 , 2.257]

Infine, si è ritenuto interessante ripetere gli insiemi di esperimenti relativi ai casi 1 e 2 anche quando nel sistema non viene eseguito il programma `testLoad` di generazione di carico. Il sistema dunque è soggetto ad un carico normale, determinato unicamente dalle richieste. I dati ottenuti assumono particolare significato se confrontati con i rispettivi dati ottenuti in condizioni di carico elevato. Tale confronto viene come proposto in tabella 7.5.

Tabella 7.5: Confronto tra i tempi di processamento rilevati

tempi processamento		min (s)	medio (s)	max (s)	dev.std (s)
senza carico	senza QoS	0.714	7.034	45.546	11.676
	con QoS	7.810	7.873	8.104	0.067
con carico	senza QoS	0.726	12.001	66.057	17.585
	con QoS	7.902	7.955	8.292	0.078

Dall'osservazione dei dati si nota come i vari tempi di processamento relativi al caso di gestione della QoS rimangono pressoché costanti, sia con carico che in assenza di carico: ciò è dovuto alla proprietà di isolamento temporale più volte enunciata, che permette al sistema di non essere influenzato dal programma di generazione del carico. I tempi di processamento relativi al caso senza gestione della QoS risultano invece migliori, nel caso di assenza del carico, soprattutto per quanto riguarda i tempi massimi: ciò avviene poiché il sistema beneficia dell'assenza del programma di carico, il quale influenzava, in quel caso, i task responsabili dei servizi. Si noti che la forbice tra i tempi minimi e massimi è comunque molto ampia ed i problemi relativi al diverso grado di soddisfazione dei vari utenti si presentano anche in questa occasione.

Si può concludere, dunque, che l'utilizzo di un sistema che gestisce la QoS è utile soprattutto in caso di carico elevato, ma il suo utilizzo è comunque opportuno ogni qual volta si debbano fornire delle garanzie di qualità del servizio.

Capitolo 8

Conclusioni e sviluppi futuri

In questa tesi si è visto quale sia l'importanza di garantire una certa qualità del servizio (QoS) nella fornitura di servizi tramite il Web, sia nell'ottica di una corretta erogazione del servizio per le applicazioni con vincoli temporali, sia nell'ottica di esigenze legate al rispetto di accordi stipulati tramite contrattazione. Si è visto come sia possibile fornire delle garanzie di QoS solamente tramite opportuni meccanismi di gestione. In particolare, si è individuato come, nelle applicazioni web, una soluzione efficace per fornire tali garanzie sia quella di utilizzare tecniche di *resource reservation* all'interno della struttura concorrente di un server web.

Per quanto riguarda gli sviluppi futuri di questa tesi, si indicheranno in questa sede due possibili direzioni di ricerca. Innanzitutto si può notare come, in questo lavoro, si siano applicate le tecniche di resource reservation solamente in relazione all'uso della CPU: ciò può costituire un problema se si devono fornire garanzie per applicazioni fortemente legate all'I/O. In riferimento a questo tipo di applicazioni, potrebbe dunque essere interessante investigare i benefici apportati dall'utilizzo di tecniche di resource reservation in cui la risorsa considerata è il disco fisso. In secondo luogo si può notare come possa essere particolarmente difficile per un consumatore stabilire, in fase di contrattazione, quale sia la banda di processore necessaria per l'esecuzione dei propri servizi. Potrebbe costituire un miglioramento, in termini di usabilità del sistema, il poter permettere ad un consumatore di specificare, per la QoS desiderata, un parametro che per lui sia maggiormente significativo, come ad esempio il tempo di risposta del servizio. Ciò implicherebbe che debba essere il fornitore a dover stabilire l'entità delle risorse da dover riservare, per eseguire quel servizio con quell'obiettivo. A tal proposito sarebbe dunque interessante valutare quali possano essere, in tal senso, i vantaggi apportati dall'uso di tecniche di *adaptive reservation* ([4]).

Bibliografia

[1] G. Buttazzo, G. Lipari, L. Abeni, M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency*, 2005

[2] C. W. Mercer, S. Savage, H. Tokuda. *Processor capacity reserves for multimedia operating systems*, Technical Report CMU-CS-93-157, Carnegie Mellon University, Pittsburg, Maggio 1993

[3] L. Abeni, G. Lipari. *Implementing resource reservation in linux*, Dicembre 2002

[4] L. Abeni, G. Buttazzo. *Adaptive bandwith reservation for multimedia computing*. In Proc. of the IEEE Real Time Computing Systems and Applications, Hong Kong, Dicembre 1999.

[5] B. Gröne, A. Knöpfel, R. Kugel, O. Schmidt. *The Apache Modeling Project*, 2004

[6] B. Laurie, P. Laurie. *Apache The Definitive Guide, 3rd Edition*, 2003

[7] The Apache Software Foundation. *Apache HTTP Server Version 2.2 Documentation*. http://httpd.apache.org/docs/2.2

[8] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, M. Xu. *Web Service Agreement Specification*, 14 Marzo 2007

[9] H. Ludwig, A. Keller, A. Dan, R. P. King, R. Franck. *Web Service Level Agreements (WSLA) Language Specification*, 2003

[10] A. Raffio, C. Tziviskou. *Service Level Agreement Languages.*

[11] J. Seidel, O. Wäldrich, W. Ziegler, P. Wieder, R. Yahyapour. *Using SLA for resource management and scheduling - a survey*, CoreGRID TR-0096, 30 Agosto 2007.

[12] O. Wäldrich, W. Ziegler, P. Wieder. *A Meta-Scheduling Service for Co-allocating Arbitrary Types of Resources*, CoreGRID TR-0010, 2 Dicembre 2005.

[13] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, A. Roy. *A Distributed Resource Management Architecture that Supports Advance Reservations and Co-allocation*

[14] K. Nahrstedt, H. Chu, S. Narayan. *QoS-aware resource management for distributed multimedia applications*. Journal on High-Speed Networking, IOS Press, Dicembre 1998.

[15] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. *Web Services Description Language 1.1*, 15 Marzo 2001.
Disponibile all'indirizzo http://www.w3.org/TR/wsdl/

[16] Simple Object Access Protocol, SOAP specification version 1.2.
Disponibile all'indirizzo http://www.w3.org/TR/soap12/

[17] S. Graham, A. Karmarkar, J. Mischkinsky, I. Robinson, I. Sedukhin. *Web Services Resource 1.2 (WS-Resource)*, 1 Aprile 2006.

[18] A. Anjomshoaa, F. Brisard, M.Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, A. Savva. *Job Submission Description Language (JSDL) Specification*, GFD-R.056, 7 Novembre 2005.

[19] J. Clark, S. DeRose. *XML Path Language (XPath) Version 1.0.*
W3C Recommendation. November 16, 1999.
Disponibile all'indirizzo http://www.w3.org/TR/xpath

[20] M. Gudgin, M. Hadley. *Web Services Addressing 1.0 - Core*
W3C Working Draft 15 Febbraio 2005

[21] http://packcs-e0.scai.fhg.de/mss-project/wsag4j/index.html - VIOLA MSS Project, WS-Agreement Framework Website.

[22] http://aquosa.sourceforge.net - AQuoSA, Official Website.

[23] http://retis.sssup.it - Real-Time Systems Laboratory of the Scuola Superiore Sant'Anna di Pisa, Official Website

[24] http://apr.apache.org/ - Apache Portable Runtime Project Official Website.

[25] http://httpd.apache.org/docs/2.2/programs/ab.html - Apache HTTP server benchmarking tool, Official Website.

[26] http://www.google.com - Google Official Website.