Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Specialistica in Informatica

Un sistema di monitoring SLA a livello applicativo

Tesi di Laurea in Sistemi Middleware

Relatore:
Chiar.mo Prof.
FABIO PANZIERI

Presentata da:
ALESSANDRO CANCEMI

Correlatore:
Dr.
NICOLA MEZZETTI

III Sessione
Anno Accademico 2009-2010

La scienza può solo accertare ciò che è,
ma non ciò che dovrebbe essere,
ed al di fuori del suo ambito restano
necessari i giudizi di valore di ogni genere.
Albert Einstein

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

I Service Level Agreement (SLA) sono elementi contrattuali che definiscono formalmente i parametri e i livelli di qualità che un servizio deve rispettare, nella specifica erogazione verso un determinato cliente. Oltre alla definizione dei livelli di qualità del servizio, un SLA può abilitare la definizione dei diritti e degli obblighi di entrambe le parti, i requisiti di sicurezza del servizio, nonché le caratteristiche delle tecniche di monitoring o di adattamento da attuare per garantire la corretta implementazione del SLA stesso.

Appropriati linguaggi formali basati su XML sono stati progettati per consentire la definizione di SLA in tutti i suoi elementi. Durante la sua implementazione, un SLA può collezionare evidenze circa il rispetto dei requisiti che definisce, individuare possibili disservizi e scatenare logiche di notifica o adattamento al fine di garantire i requisiti contrattati, oppure evidenziare le deficienze del servizio per consentire al cliente la riscossione delle penali che possono essere previste nel contratto. Per garantire la più corretta implementazione rispetto alle esigenze del cliente e ridurre al minimo le possibilità di generare contenziosi, un SLA dovrebbe specificare i vincoli attraverso parametri definiti allo stesso livello di astrazione del servizio oggetto della fornitura e dovrebbe essere direttamente interpretabile. Per esempio, parametri di qualità come banda o tempo di inoltro dovrebbero essere impiegati per definire contratti di fornitura di servizi infrastrutturali mentre parametri come

throughput o tempo di risposta dovrebbero essere utilizzati nella definizione di contratti di erogazione di servizi applicativi. Vale la pena notare che vi sono parametri, come la disponibilità, che trovano senso a diversi livelli di astrazione.

In questo lavoro si vuole studiare il problema del monitoring di quei parametri del SLA che riguardano aspetti definiti a livello applicativo, nei contesti dell'erogazione di servizi business-to-business.

Al momento, la maggior parte dei lavori sul monitoring e l'imposizione di SLA, anche a livello applicativo, prevedono la specifica dei requisiti di qualità in termini di parametri infrastrutturali, come bandwidth e cpu time, e tentano di imporre i livelli di qualità contrattati attraverso tecniche di adattabilità basate su tali parametri. A nostro avviso, il punto debole di questo approccio risiede nel processo di traduzione dei vincoli di qualità tra la specifica del cliente, formalizzata secondo dimensioni a livello applicativo, e una specifica adatta ad essere interpretata dal servizio di implementazione di SLA. Al momento non ci risulta l'esistenza di alcuna metodologia formale per tradurre un vincolo espresso secondo una dimensione di livello applicativo nelle corrispondenti dimensioni di livello infrastrutturale. Tuttavia, anche assumendo l'esistenza di una funzione esatta per risolvere questo problema, questa traduzione perderebbe le informazioni circa la correlazione tra queste ultime dimensioni e lo specifico servizio di riferimento.

Il primo risultato che ci proponiamo di ottenere attraverso questo lavoro di tesi mira a risolvere questo problema attraverso la progettazione e lo sviluppo di un'infrastruttura che sia capace di controllare l'esecuzione di un numero arbitrario di applicazioni, monitorandone la capacitĹ di soddisfare i vincoli di livello applicativo specificati dai service level agreements definiti tra il fornitore dei servizi e i clienti di tali applicazioni; a questo scopo, abbiamo individuato i seguenti parametri:

- Tempo di risposta
- Throughput

- Disponibilità

Un secondo risultato che intendiamo ottenere con questo lavoro riguarda la definizione e l'implementazione di quella porzione di SLA che definisce i diritti e gli obblighi delle parti. L'idea alla base di questo contributo nasce dall'osservazione secondo la quale parlare di monitoraggio o imposizione di requisiti di qualità su operazioni applicative potrebbe perdere di significato qualora queste operazioni non fossero opportunamente utilizzate. Basti pensare ad un servizio web, che implementa diverse operazioni, che eroga correttamente il proprio servizio quando le operazioni sono invocate seguendo un ordine ben preciso. Definiamo un pattern di interazione legittimo come una sequenza, non necessariamente finita, di invocazione delle operazioni di un servizio coerente con la semantica di uso del servizio. Tutte le altre sequenze di esecuzione definiscono l'insieme dei pattern non legittimi.

Noi riteniamo che sia legittimo assumere che l'utilizzatore di un servizio sia a conoscenza della semantica delle operazioni messe a disposizione del servizio e della modalità secondo la quale queste devono essere invocate; sulla base di questa assunzione riteniamo che l'utilizzatore di un servizio abbia il diritto di pretendere il mantenimento dei livelli di qualità contrattati solamente quando questi utilizza il servizio secondo pattern di interazione legittimi. Assumiamo inoltre che, in corrispondenza di pattern di interazione non legittimi, l'utilizzatore del servizio non debba avere diritto di reclamo a fronte di violazioni di SLA.

Il nostro proposito è quindi quello di definire, progettare, e implementare un servizio che sia capace di eseguire il monitoring dei livelli di qualità del servizio erogato congiuntamente al monitoring dei pattern di interazione, riconoscendo le violazioni dei livelli di qualità erogati tenendo in considerazione tutte e sole le invocazioni che corrispondono a pattern di interazione legittimi. A tal fine, abbiamo previsto le seguenti attività:

1. Studio delle possibili relazioni utili a descrivere i pattern di interazione legittimi per le invocazioni delle operazioni di un servizio;

2. Studio dello stato dell'arte per l'individuazione di un linguaggio di specifica di SLA congruo rispetto alle nostre esigenze ed eventuale arricchimento;

3. Progettazione dell'architettura di monitoring come un servizio interno a JBoss, che:

   - implementi la logica di controllo descritta in precedenza;
   - sia indipendente dalle applicazioni monitorate; pertanto si possa impiegare per controllare l'esecuzione di una qualunque applicazione senza richiedere variazioni al codice dell'applicazione stessa;
   - sia in grado di eseguire contemporaneamente molteplici SLA per monitorare uno o più servizi. Uno stesso servizio deve poter essere erogato verso diversi clienti e uno stesso cliente deve poter accedere a molteplici servizi, schierati sulla stessa istanza di application server. Ogni coppia (cliente, servizio) sarà caratterizzata da un propria istanza di SLA;
   - sia robusto e tollerante ai guasti;
   - sia in grado di astrarre dallo specifico linguaggio di descrizione di SLA attraverso un approccio di ingegnerizzazione che prevede componenti dedicati all'interpretazione e all'implementazione di uno specifico linguaggio di descrizione del SLA.

Con questa tesi, ci proponiamo di dimostrare la fattibilità di questo progetto e di darne un'implementazione prototipale che tenga conto degli aspetti prestazionali in modo che sia immediatamente utilizzabile in contesti industriali.

Si noti che l'obiettivo di questa tesi non è lo studio di una mappatura comune a tutti i linguaggi per la descrizione di service level agreements ma, piuttosto, l'implementazione di un servizio che soddisfi i requisiti espressi precedentemente; a tal fine, prenderemo in considerazione la gestione di un

solo linguaggio di descrizione del SLA, lasciando la gestione dei rimanenti linguaggi come obiettivo di estensioni future.

Capitolo 2

Stato dell'arte

2.1 Introduzione

Un aspetto importante per il successo di un'impresa che offre servizi web è realizzare un adeguato livello di performance e di disponibilità descritti attraverso un SLA. Infatti il fornitore, indipendentemente dal tipo di servizio offerto, dovrebbe tentare di soddisfare i vincoli imposti da un SLA al meglio delle sue capacità poiché in caso contrario si potrebbero avere forti ripercursioni monetarie.

Un passo importante nel processo per il supporto di SLA è la creazione e la presentazione di relazioni periodiche al cliente sulle performance monitorate e rese disponibili per l'analisi. Le questioni fondamentali relative al monitoraggio includono un preciso linguaggio per descrivere in modo formale un SLA, il calcolo di metriche flessibili ed infine dei servizi per la rilevazione delle violazioni.

2.1.1 Il linguaggio per SLA

Il primo obiettivo di un linguaggio che definisce SLA è di fornire la capacità di esprimere, con il massimo grado di precisione, le caratteristiche qualitative e quantitative di un servizio. Attraverso questo linguaggio, le parti coinvolte nel contratto possono formulare rapidamente e con precisione

il livello di un particolare servizio ed offrirlo al tempo stesso. Altri risultati rilevanti sono la possibilità di far riferimento ad uno standard, che tutti sono in grado di comprendere ed utilizzare, realizzare in maniera semplice il confronto fra le diverse offerte, pubblicizzare e recuperare informazioni sulle diverse offerte, ragionare sulle proposte del servizio, in modo da capire cosa si possa offrire e ricevere ed infine controllare facilmente le garanzie della qualità del servizio.

I principali requisiti per raggiungere tali obiettivi sono:

- *Parametrizzazione*: ogni SLA include un insieme di parametri per fornire una descrizione qualitativa di un servizio;
- *Composizionalità*: in un ambiente multi-dominio, un servizio potrebbe essere il risultato di una cooperazione tra le diverse entità del dominio. I servizi possono essere dissociati o aggregati, quindi, i fornitori del servizio dovranno essere capaci di comporre SLA in modo da formulare nuove offerte per i clienti;
- *Validazione*: Prima di iniziare un SLA, i contraenti devono essere in grado di validarlo, verificando la sua sintassi e la sua coerenza;
- *Monitoraggio*: Idealmente, le parti dovrebbero essere capaci di controllare automaticamente se i livelli del servizio stabiliti in un accordo vengono effettivamente forniti dai suoi fornitori ed analogamente se i livelli del servizio forniti ai propri clienti sono stati soddisfatti;
- *Imposizione*: Una volta che vengono concordati i livelli del servizio, possono essere configurati router di rete, sistemi di gestione dei database, middleware e Web Server per far rispettare SLA in modo automatico utilizzando tecniche come cache, replica, clustering e farming.

In [KLD+02] viene presentato *Web Service Level Agreement* (WSLA), un linguaggio flessibile ed estendibile basato su XML Schema. Il linguaggio formale di WSLA permette di definire qualsiasi tipo di accordo indipendentemente dai parametri utilizzati per specificare i requisiti di qualità del servizio

(QoS), che possono variare in base al contesto. WSLA comprende un insieme di estensioni standard che consentono di definire accordi completi che si riferiscono ai web service e che includono garanzie relative a tempi di risposta, throughput ed altre metriche comuni.

In [TPP+03] viene presentato, invece, *Web Service Offerings Language* (WSOL), un linguaggio basato su XML che permette il monitoraggio e la valutazione della qualità del servizio tra i fornitori e i richiedenti di web service. Ogni offerta del servizio che descrive un insieme di qualità del servizio, vincoli funzionali e diritti di accesso è dinamica e può essere manipolata a runtime. WSOL presenta un SLA basandosi su misurazioni effettuate in ontologie esterne che, in modo strutturato, descrivono in linguaggio naturale le misurazioni, compresi i consigli su come dovrebbero essere adottate e le loro interdipendenze. Anche WSOL come WSLA definisce opportune azioni di gestione, comprese le notifiche in caso di violazioni dei vincoli imposti da un SLA. In WSLA, però un SLA contiene maggiori dettagli per i vincoli della QoS rispetto ai servizi offerti da WSOL. Ad esempio, WSLA contiene la descrizione dettagliata delle garanzie di un'azione di gestione, mentre WSOL descrive solo le sanzioni monetarie da apportare. Inoltre, a differenza di WSLA, i servizi offerti da WSOL possono contenere specifiche formali dei vincoli non funzionali e dei diritti di accesso, che non sono specificati nel SLA.

In [SLE04] viene invece introdotto SLAng, un linguaggio per SLA che, a differenza dei precedenti che si concentrano esclusivamente sui web service, definisce un vocabolario per un ampio spettro di servizi Internet. Il linguaggio è modellato in UML; gli autori non hanno munito il linguaggio di uno schema xml ritenendo che questo potesse limitarne l'estendibilità. SLAng consente la descrizione dei vincoli che devono essere soddisfatti attraverso la specifica di vincoli espressi mediante il linguaggio *Object Constraint Language* (OCL) e accompagnati da descrizioni in linguaggio naturale che ne definiscono la semantica. I benefici della semantica formale comprendono sia la riduzione dell'ambiguità nel significato del linguaggio che i mezzi per

controllare l'assenza di incongruenze e di lacune.

2.1.2 Approcci generali

Solitamente per supportare e gestire SLA vengono utilizzati tre approcci. Il primo, chiamato *Insurance* ed attualmente il più diffuso, viene descritto attraverso i seguenti punti:

1. individua gli obiettivi del servizio offerto (performance, disponibilità, tempi di risposta);
2. monitora gli obiettivi del servizio concordato;
3. rilascia relazioni SLA, possibilmente includendo incontri periodici con i clienti per discutere lo stato di conformità dei SLA;
4. rilascia dei crediti appropriati ai clienti se non sono stati garantiti i livelli del servizio;
5. modifica periodicamente gli obiettivi del livello del servizio in modo che la probabilità di una loro violazione ed il conseguente impatto finanziario siano accettabili.

Una descrizione dell'architettura di rete che controlla il rispetto dei vincoli imposti da un SLA può essere trovata in [KSH00]

Il secondo approccio, chiamato *Provisioning*, a differenza del primo utilizza tecniche di configurazione per supportare SLA all'interno della rete. In particolare:

1. individua gli obiettivi del servizio (performance, disponibilità, tempi di risposta) da fornire a ciascun cliente;
2. **determina la giusta configurazione del sistema da utilizzare per ciascuno dei clienti**;
3. monitora gli obiettivi del servizio concordato;

4. rilascia relazioni SLA, possibilmente includendo incontri periodici con i clienti per discutere lo stato di conformità dei SLA;

5. rilascia dei crediti appropriati ai clienti se non sono stati garantiti i livelli del servizio.

Il terzo approccio, chiamato *Adaptive*, aggiunge un altro aspetto della configurazione che si adatta all'approccio *Provisioning*. In particolare:

1. individua gli obiettivi di servizio (performance, disponibilità, tempi di risposta) da fornire a ciascun cliente;

2. determina la giusta configurazione del sistema da utilizzare per ciascuno dei clienti;

3. monitora gli obiettivi del servizio concordato;

4. **se il monitoraggio indica una possibile violazione degli obiettivi, riadatta la configurazione del cliente per garantire al meglio gli obiettivi del servizio**;

5. rilascia relazioni SLA, possibilmente includendo incontri periodici con i clienti per discutere lo stato di conformità dei SLA;

6. rilascia dei crediti appropriati ai clienti se non sono stati garantiti i livelli del servizio.

2.2 Sistemi di monitoring e imposizione di SLA

Nonostante negli ultimi anni si siano compiute molte ricerche per adottare nuove e sofisticate tecniche di monitoraggio di SLA, molti fornitori di servizi utilizzano metodi propri per realizzarlo.

Per gli attributi non deterministici (quelli sconosciuti al momento dell'invocazione del servizio come ad esempio il tempo di risposta o la disponibilità)

possono essere usati diversi approcci di monitoraggio per misurare continuamente i valori attuali della qualità del servizio. Concettualmente ci sono due approcci per il monitoraggio: server-side e client-side. Il primo è solitamente preciso ma richiede l'accesso all'attuale implementazione del servizio che non è sempre possibile. Il secondo è invece indipendente dall'implementazione del servizio ma i valori misurati potrebbero non essere sempre aggiornati dato che solitamente viene realizzato inviando richieste di prova.

In genere misurazioni come latenza, throughput, tasso di trasferimento, tempo di risposta dei messaggi, disponibilitá e affidabilitá possono essere compiute sia attraverso un monitoraggio client che server side mentre misurazioni come scalabilitá, robustezza, capacitá, gestione delle eccezioni, stabilitá vengono compiute solo attraverso un monitoraggio server side.

In [TK05], gli autori discutono ed analizzano tre diverse tecniche di monitoraggio client e server side per gli attributi relativi della qualità del servizio: sniffing a basso-livello, proxy e modifiche della libreria SOAP Engine. Per lo sniffing a basso livello, che si occupa di catturare pacchetti SOAP in entrata ed in uscita, il maggiore vantaggio è dato dall'indipendenza dal codice client, tuttavia le limitazioni riguardano la dipendenza dall'hardware per il riconoscimento dei protocolli di rete e la possibilità di decifrare e decomprimere messaggi SOAP cifrati o compressi. L'approccio proxy invece, usato come un mediatore tra il client e il fornitore, ha il vantaggio di essere indipendente dalla piattaforma e dall'hardware e ha meno overhead di CPU rispetto all'approccio di sniffing. Gli svantaggi sono dati dal bisogno di configurare il codice client e dall'incapacità di risolvere la trasformazione dei messaggi SOAP direttamente, quindi di appesantire il codice del client. Infine per quanto riguarda l'approccio di modifica della libreria SOAP Engine per registrare le informazioni necessarie per le misurazioni delle performance, i maggiori vantaggi sono dovuti alla capacitá di trasformare direttamente i messaggi SOAP, dal poco overhead di CPU rispetto all'approccio di sniffing e dall'indipendenza dal codice dalla parte del client. Tuttavia le limitazioni di questo approccio sono dovute alla dipendenza dall'implementazione e dalla

piattaforma.

Sia il monitoraggio client side che server side può avvenire offline oppure online. Nel monitoraggio offline vengono registrate e memorizzate in sicurezza dal monitor tutte le interazioni, in genere dal lato client, che vengono analizzati in determinati intervalli di tempo per verificare se è stata violata la qualità del servizio. Diversamente avviene nel monitoraggio online in cui i dati vengono analizzati quando ancora il servizio viene fornito, e gli avvisi generati non appena viene rilevata una violazione della qualità del servizio. Il monitoraggio online comporta test periodici se le condizioni del contratto vengono rispettate da tutte le parti interessate, con intervalli che possono variare, a seconda dell'accordo (in generale dell'ordine dei secondi). Una proprietà come il bandwidth della rete, per esempio, deve essere monitorata continuamente se si vuole garantire che gli attributi riguardanti la qualità del servizio non vengano violati. Il monitoraggio online è difficile da implementare in modo efficiente per la continua interazione fra tutte le parti coinvolte. Il monitoraggio offline, invece, presenta svantaggi dovuti dalla necessità di immagazzinare grandi volumi di dati e, cosa più importante, dall'incertezza nel determinare se la violazione abbia avuto luogo.

Agli approcci descritti in precedenza si aggiungono altri tre metodi di monitoraggio della rete: attivo, passivo e attraverso l'uso di agenti *Simple Network Management Protocol* (SNMP). Il monitoraggio attivo ottiene lo stato attuale della rete configurando la macchina di test nel punto che si vuole misurare e successivamente inviando, per un periodo specifico, traffico extra da una macchina all'altra. Le performance possono essere misurate con strumenti semplici e facili, come il ping e il traceroute, inoltre la quantità di traffico generato ed analizzato è ridotto rispetto al monitoraggio passivo. Come conseguenza i pacchetti di prova possono perdersi ed il traffico per il test potrebbe appesantire la rete. In generale, smarrimenti, ritardi e connettività sono metriche che possono essere misurate mediante un monitoraggio attivo. Un lavoro connesso a questo tipo di monitoraggio è RIPE NCC Test Traffic Measurement [ACKO02].

Il monitoraggio passivo, invece, ottiene lo stato attuale della rete catturando il pacchetto senza creare ulteriore traffico. Tuttavia, rispetto al metodo di monitoraggio attivo possono essere misurate facilmente solo alcune metriche di performance, come l'utilizzazione e il throughput. Un lavoro connesso a questo tipo di monitoraggio è descritto in [HKJH02].

Anche l'ultimo metodo che utilizza agenti SNMP, per esempio in [Wal95], permette di misurare lo stato del dispositivo di rete, monitorando le informazioni sul traffico. Tuttavia, nonostante sia semplice e scalabile questo metodo permette di misurare solo il throughput e la funzionalità della rete.

Di seguito vengono descritti i principali lavori che si focalizzano su un particolare approccio di monitoraggio: server side o entrambi.

2.2.1 Monitoring Server side

SLA Monitoring

Ci sono numerosi lavori che realizzano le diverse tecniche di monitoraggio mostrate. In [BAFP09] viene ad esempio presentato un framework di validazione PLASTIC che può combinare diverse tecniche per la verifica di proprietà funzionali o non-funzionali, spaziando da fasi testing offline ed online. Le proprietà SLA sono descritte in SLAng e in WS-Agreement. L'implementazione, tuttavia, presenta alcune limitazioni tra cui l'integrazione dei soli sensori per il tempo di risposta nonché l'affidabilità e l'assenza di punti di integrazione per la gestione dei SLA e l'accesso ai dati di monitoraggio.

In [KL03], gli stessi autori del linguaggio WSLA forniscono un framework per integrarlo facilmente nei sistemi e-commerce. Seppure il framework possa lavorare con qualsiasi tipo di servizio, si suppone che il SLA sia definito per un web service descritto da un documento WSDL. L'architettura prevede diverse fasi per realizzare un sistema di monitoraggio completo. La fase di negoziazione consente di concordare il relativo SLA, la fase di sviluppo di verificare la validità dell'accordo e di distribuirlo ai diversi componenti coinvolti. Le fasi di misurazione e valutazione si occupano invece della raccolta

delle misurazioni provenienti dalle risorse e della verifica del rispetto degli accordi definiti nel SLA. Il servizio di valutazione delle condizioni, una volta rilevata la violazione di un qualche obiettivo, notifica con un messaggio il servizio di gestione, che applicherà le eventuali azioni correttive descritte nel SLA. Nel framework inoltre è possible definire alcune condizioni che provocano la terminazione di un contratto, come ad esempio il mancato rispetto di una clausola da parte di una delle due parti o il raggiungimento di un certo timeout.

Anche in [TMPE04], come [KL03] fornisce l'infrastruttura per il relativo linguaggio implementato. WSOI è fondamentalmente un parser XML e un estensione di SOAP Engine, che fornisce un elaborazione sugli artefatti WSOI. Il parser XML fornisce mezzi per convalidare file WSOL e di trasformare le offerte di servizi WSOL e i file (ontologie, WSDL) in alberi XML e tabelle di simboli particolari contenenti i dati necessari per la validazione semantica del WSOL estratto dalle ontologie. I gestori WSOI sono in pratica handler Axis di Apache. Questi forniscono i mezzi per le richieste SOAP, le risposte, l'intercettazione e la manipolazione di messaggi di errore in Axis. Attraverso questo il WSOI è in grado sia di fornire le offerte del servizio che di monitorare le sue richieste e risposte. Nel lavoro non si spiega se tutti i fornitori dei servizi web in realtà dovrebbero includere l'estensione del SOAP Engine, o se quest'ultimo dovrebbe essere immesso in un SOAP intermediario aziendale (router SOAP). Inolte, le descrizioni del meccanismo di pagamento sono supportati dal linguaggio WSOL ma non sono in alcun modo integrati automaticamente alla soluzione.

Anche in [LKHL02] viene presentata una generica architettura, composta da tre componenti, di un sistema di monitoraggio di SLA per servizi di rete NSP e ISP che può essere utilizzata per i vari servizi offerti dalla rete. Il primo componente, chiamato *Parameter Mapper*, si occupa di decidere per ogni parametro relativo alla qualità del servizio le relative metriche di performance della rete da correlare, la locazione e il periodo del monitoraggio, nonché il tipo di gestori da utilizzare. Il secondo componente, chiamato *Customer*

SLA, si occupa di memorizzare le informazioni di ogni contratto stipulato col cliente, mentre l'ultimo componente, chiamato *Monitoring*, si occupa di raccogliere per ogni gestore SLA (uno per ogni metrica di performance della rete) i dati misurati, inviandoli all'analizzatore delle performance. Quest'ultimo si occupa di validare per ogni cliente la qualità di un particolare servizio usando sia una funzione di valutazione che le informazioni del contratto stipulato col cliente, entrambi memorizzati in specifici database. Le informazioni analizzate vengono quindi memorizzate in un data store per un certo periodo di tempo e, se viene individuato un problema come una violazione di SLA, l'analizzatore delle performance invia un messaggio di avvertimento ad un particolare gestore che avverte l'amministratore o il sistema di manutenzione SLA.

Oggi molti lavori si occupano della creazione di un gruppo di server condivisi e la fornitura di server liberi per la riconfigurazione automatica della rete. Oceano [AFF+01], ad esempio, è un prototipo scalabile con un'infrastruttura gestibile per realizzare un utility computing per e-business. Oceano gestisce le risorse dell'utility computing in maniera dinamica ed automatica riassegnando risorse ad applicazioni distribuite per soddisfare specifici SLA scritti attraverso un generico linguaggio. Oceano include parecchie funzionalità tra cui il monitoraggio guidato di SLA, la correlazione di eventi, la scoperta di topologie di rete, e la riconfigurazione automatica della rete.

Gli autori di [LPRT07], a differenza del prototipo Oceano presentano un lavoro che si applica specificatamente nel contesto Application Server. In particolare, viene presentato un middleware che gestisce la qualità del servizio (non web service) dentro un cluster di Application Server. Questo approccio propone tre moduli, implementati come componenti, per assicurare l'adozione della qualità del servizio. Il primo è il modulo di monitoraggio, che controlla periodicamente la configurazione del cluster di appartenenza per rilevare se i nodi devono aderirvi o abbandonarlo quando si hanno fallimenti o altri problemi. Inoltre, controlla diversi dati del cluster come il tempo di risposta, il tasso di richiesta del cliente, e violazioni di SLA per rilevare se la

qualità del servizio del cluster consegnato si discosta da ciò che è richiesto e specificato nel SLA. Il secondo componente è il modulo della configurazione che è responsabile di assicurare la disponibilità del servizio ricostruendo il cluster al fine di soddisfare le specifiche SLA. Il terzo è il modulo di bilanciamento del carico che intercetta le richieste del cliente per bilanciarle tra i diversi nodi del cluster. Per definire un SLA, invece di standard esistenti, usano un XML ispirato da SLAng.

Un approccio simile viene adottato in [ZK02] dove viene presentata un'architettura per l'enforcement coordinato di SLA tra applicazioni usando risorse clusterizzate. Tuttavia quest'architettura invece che realizzata a livello middleware come in [LPRT07], viene implementata su due differenti livelli di rete nel contesto web service: redirezione HTTP a livello applicazione e redirezione di pacchetti a livello trasporto.

Gran SLAM [SH09] è un sistema basato su Java EE capace di monitorare proprietà non funzionali e l'aderenza dei contratti SLA negoziati. L'architettura di Gran SLAM, basata sul framework OSGi, è composta principalmente da cinque componenti. Il più importante è un nucleo che gestisce e controlla gli altri moduli ma soprattutto è responsabile della supervisione dei contratti che sono memorizzati nel database locale. Gli autori hanno progettato una libreria di astrazione SLA che permette di estrarre obiettivi, date di scadenza e altri dati importanti dai linguaggi WS-Agreemnt e WSAG. Ogni volta che un nuovo contratto diventa attivo, Grand SLAM ispeziona i relativi SLA estraendo i parametri che devono essere garantiti e, successivamente, attiva tutti i componenti di misurazione non ancora avviati per fornire i relativi risultati al nucleo. Attraverso il pattern observer, il nucleo verrà notificato da eventuali nuove misurazioni. Grand SLAM divide le misurazioni in due gruppi. Il primo, il monitoraggio locale, include misurazioni specifiche del server come l'utilizzo della CPU. Il secondo gruppo, il monitoraggio remoto realizzato attraverso un proxy esterno, compie misurazioni specifiche dei web service come il tempo di avvio o il tempo di risposta. Entrambi i monitoraggi vengono effettuati online in modo attivo cioè inviando traffico extra in

determinati intervalli di tempo definiti dall'utente.

In [MPMJS05] questo articolo gli autori descrivono un'implementazione di monitoraggio SLA che può generare metriche dei dati raccolti da software direttamente da macchine di lettura SLA. L'obiettivo del lavoro proposto è di fornire servizi di monitoraggio generici che possono essere adattati per l'uso in ambienti eterogenei supponendo di avere un ente specializzato nel monitoraggio SLA e una valutazione che potrebbe non essere legata a nessuna particolare piattaforma middleware e/o linguaggio SLA. Per la descrizione

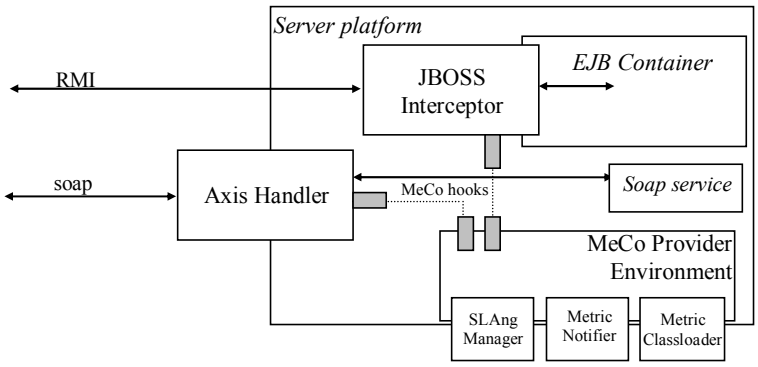

Figura 2.1: Architettura di una generica piattaforma di monitoring SLA

del linguaggio SLA nell'implementazione dell'architettura di monitoraggio gli autori si sono avvalsi di SLAng. In figura 2.1 viene mostrata l'architettura della piattaforma con le interazioni tra in vari componenti. In particolare il componete MeCo è responsabile della raccolta di metriche dei dati e la propagazione di tali dati ai servizi di misurazione e valutazione. MeCo è disponibile per un utilizzo arbitrario di piattaforme middleware e protocolli associati. I gestori SOAP Engine Axis, di Apache Software Foundation forniscono un'opportunità appropriata per reindirizzare i messaggi SOAP al servizio MeCo (attraverso JBoss interceptors per invocazioni Java RMI) al fine di raccogliere le relative metriche. *SLAng Manager* esamina un SLA per produrre la metrica del dato che MeCO osserva. *Metric Notifier*, invece, assume la responsabilità di gestire il messaggio appropriato che passa tra MeCo e il servizio di misurazione. *Metric Classloader*, infine, carica la

metrica per implementare il monitoraggio dei dati richiesti come indicato da *SLAng Manager*. Da notare che per ogni linguaggio SLA si avrà un diverso componente *SLAng Manager* e *Metric Classloader*.

QoS Monitoring

In [PSS+05], gli autori presentano una piattaforma, implementata come un'estensione dell'Application Server Web-Sphere il cui obiettivo è di gestire i tempi di risposta di diverse applicazioni web. All'interno di un ambiente di hosting, il sistema reagisce principalmente al collocamento delle repliche, al numero di repliche da utilizzare per ciascuna delle applicazioni web e all'assegnazione dei server adatti per l'esecuzione di tali repliche. Per raggiungere questi obiettivi, il sistema fa uso di un livello proxy che divide e controlla il flusso delle richieste, classificandole secondo la politica definita dall'utente. Successivamente uno scheduler si occupa di inoltrare la richiesta ad un meccanismo di bilanciamento del carico che seleziona il nodo idoneo per elaborarla. Attraverso questa architettura possono essere schierate, su nodi diversi di un cluster WebSphere, differenti repliche di diverse applicazioni web. A differenza del lavoro proposto in [LPRT07] il sistema mappa le performance osservate ad una funzione di utilità, e costantemente adatta l'allocazione delle risorse per ottimizzare l'intero sistema, ma non offre un meccanismo di gestione delle risorse dinamico guidato da SLA.

In [WLLM07] viene presentato un monitoraggio online con il relativo approccio di analisi per i seguenti eventi: messaggi di risposta, esecuzione dell'applicazione, cambiamenti di stato delle risorse, richieste del client e gestione delle operazioni. Il framework è composto da sonde distribuite, agenti, un analizzatore centrale ed una rappresentazione ad alto livello. Sia le sonde che gli agenti vengono utilizzati per estrarre eventi interessanti. Alcune sonde possono compiere anche l'analisi per quei vincoli che sono facili da controllare (ad esempio il valore dei parametri e tempi di risposta). A differenza delle sonde l'agente si mantiene indipendente dalla specifica applicazione e viene eseguito come un processo proprio. Sono responsabili ad esempio dell'uti-

lizzo medio della CPU o della memoria negli ultimi minuti. L'analizzatore centrale viene utilizzato per elaborare i diversi tipi di eventi, in particolare per individuare le violazioni di un vincolo (un messaggio sbagliato, tempo di risposta che supera un certo limite), prevedere il rischio (carico in eccesso, perdita di memoria, deadlock), determinare il problema (considerare la ragione della violazione del comportamento del servizio). Per dare all'amministratore del servizio una visione ad alto livello della qualità del sistema, il framework prevede la rappresentazione visiva dello stato del web service.

2.2.2 Monitoring Client e Sever side

Molti lavori si focalizzano su tecniche efficienti di monitoraggio online che sfruttano entrambi gli approcci client che server side.

SLA Monitoring

In [MRLD09], viene presentato un framework, realizzato in Vienna Runtime Environment for Service-oriented Computing (VRESCo), che integrando i due approcci usa un'elaborazione di eventi che permette facilmente di gestire gli obblighi SLA, descritti attraverso *Esper Processing Language* (EPL), in modo da reagire tempestivamente in caso di violazioni. Il monitoraggio client side utilizza un tool chiamato QUATSCH che è in grado di misurare accuratamente gli attributi server side a tempo di esecuzione usando l'analisi e lo sniffing di pachetti TCP a basso livello. In particolare sfruttando l'handshake TCP per distinguere i diversi tempi di invocazione del servizio. Il monitoraggio server side basato invece sul Windows Performance Counter (WPC) è ristretto ai servizi implementati attraverso il framework .NET essendone parte integrante. WPC supporta un ricco insieme di contatori che possono essere misurati a runtime (per esempio il tempo di esecuzione, il numero di chiamate al secondo, ecc.). In entrambi gli approcci il monitoraggio viene effettuato online in modo attivo cioè inviando traffico extra in determinati intervalli di tempo definiti dall'utente. Da notare che pur essendo indipendenti l'uno dall'altro, alcuni attributi relativi alla qualità del servizio

possono essere misurati solamente da uno dei due approcci (ad esempio la latenza e il tempo di risposta sono misurati client side).

In [RSE08] viene descritto, invece, un sistema di monitoraggio online che traduce vincoli temporali, di affidabilità e di throughput espressi in SLA per web service in automi temporizzati. Viene quindi presentata un'implementazione di questa tecnica utilizzando un plug-in per Eclipse e dei gestori SOAP Engine Axis, di Apache Software Foundation. Il plug-in per Eclipse che è stato implementato per creare, modificare e verificare la correttezza di diversi SLA scritti in SLAng, produce il rispettivo gestore SOAP che viene usato per esplorare la catena di messaggi in entrata ed in uscita. In particolare analizza e spedisce i messaggi SOAP ai rispettivi verificatori che implementano le metodologie di verifica attraverso gli automi temporizzati. Ogni gestore, con i rispettivi verificatori possono risiedere sia sul lato client che su quello server. Il vantaggio principale del lavoro presentato consiste nella separazione delle performance per la verifica dei parametri (monitorati attraverso i verificatori) dal numero totale dei messaggi nel sistema. Ogni verificatore SLA, inoltre è molto leggero (pochi Kb). Attraverso questa tecnica di verifica gli autori mostrano come gestire poche centinaia di eventi per secondo ottenendo una media di verifica al di sotto un milli secondo.

QoS Monitoring

In [HDJ08] viene presentato un middleware QOSH (QoS-Oriented Self-Healing) con auto risanamento per applicazioni basate su web service orientato alla qualità del servizio. Un modulo di monitoraggio prevede l'osservazione e la conservazione di valori rilevanti dei parametri QoS sia client-side (RSM) che server-side (PSM). Ai messaggi SOAP sincroni ed asincroni osservati vengono aggiunti dei parametri della qualità del servizio definiti come metadati che estendono gli header dei messaggi SOAP. Il modulo di analisi valuta la salute di un determinato servizio e non una specifica interazione all'interno di una conversazione. L'approccio osserva l'evoluzione dei valori QoS calcolati a runtime e mira a rilevare il degrado QoS che è considerato come il sintomo

di un deficit futuro o imminente. Il middleware supporta politiche di risanamento predicenti e reattive implementate attraverso i moduli di Diagnosi e Prognosi. Il modulo di decisione si basa sulle azioni di riconfigurazione architetturale che fornisce sostituzioni singole e composte per i web service all'origine della violazione della qualità del servizio per la prognosi e di azioni di riparazione per la diagnosi. Molti componenti con auto-risanamento possono essere impiegati da una terza parte in modo da non sovraccaricare il lato del fornitore o del richiedente. La terza parte è una persona di fiducia tra il prestatore e il richiedente.

In [TGN+03] viene presentata una architettura che permette la definizione dinamica, la pubblicazione e il matching sia delle offerte di un web service che i requisiti riguardanti le prestazioni del server e della rete, la sicurezza, le transazioni e i prezzi sia a run-time che a tempo di implementazione. L'architettura consente inoltre agli utenti di ottenere informazioni in tempo reale sulle prestazioni del server al fine di monitorare il conseguimento dei servizi assicurati in modo da dare all'utente un feedback immediato sul QoS. Sul lato client, un proxy QoS risiede tra il client del servizio Web e l'interfaccia di rete. Il proxy osserva il traffico su una porta specifica, attraverso il quale il client del web service invia le sue richieste al server. Il proxy mappa le richieste del cliente sulla rete dopo aver rilevato i parametri di trasporto QoS configurati dall'applicazione del cliente. Sul lato server, un proxy o un bilanciatore di carico, che si trova tra l'interfaccia di rete e il web service, quando il fornitore del servizio invia le risposte al client definisce i parametri di QoS in base ai suoi requisiti. Le informazioni QoS relative alle prestazioni di rete indicate dal client vengono poste nel header SOAP, che verrà analizzato dal proxy QoS sia sul lato client che server. L'utente definisce le informazioni QoS che gli interessano riguardanti le prestazioni del server mentre un'interfaccia utente grafica (GUI) sul lato client mostra le prestazioni del server e della rete.

Capitolo 3

Progettazione

Questo capitolo presenta il modello del prototipo che abbiamo prodotto durante lo svolgimento di questo lavoro di tesi. Prima di affrontare nel dettaglio le caratteristiche del progetto che abbiamo elaborato, vale la pena ricordare che il nostro lavoro si propone di sviluppare un servizio interno a JBoss che sia in grado di:

- implementare una logica di controllo tale da individuare violazioni sui requisiti di disponibilità, tempo di servizio, e throughput, compatibilmente con i vincoli imposti dalla descrizione del buon uso di un servizio;

- essere configurabile e componibile ad ogni applicazione che espone web services senza richiedere variazioni al codice dell'applicazione;

- eseguire contemporaneamente molteplici SLA per monitorare uno o più servizi. Uno stesso servizio deve poter essere erogato verso diversi clienti e uno stesso cliente deve poter accedere a molteplici servizi, schierati sulla stessa istanza di application server. Ogni coppia (cliente, servizio) sarà caratterizzata da un propria istanza di SLA;

- essere robusto ai guasti (la perdita dei dati deve essere minima all'occorrenza di guasti di tipo crash);

- mantenere una banca dati contenente le informazioni relative alle invocazioni effettuate dai clienti ai servizi in modo da rendere riproducibili i controlli effettuati;
- imporre un overhead trascurabile sulle performance dell'application server.

3.1 Prerequisiti

Questa sezione ha lo scopo di chiarire tutti i prerequisiti che sono fondamentali alla comprensione del contenuto di questo capitolo.

Il primo aspetto che è necessario aver chiaro è la possibilità di osservare le comunicazioni tra un client web e un server che risponde a messaggi SOAP. Al fine di abilitare una programmazione orientata agli aspetti relativamente a web service, gli application server che implementano il framework J2EE mettono a disposizione gli handler, che sono astrazioni utili ad intercettare i messaggi SOAP consumati e prodotti da un web service, per incapsulare logiche relative alla gestione di aspetti non funzionali. L'idea è quindi quella di sfruttare l'astrazione di handler per implementare una sonda in grado di monitorare i messaggi SOAP scambiati tra clienti web ed un web service, estraendo tutte le informazioni che sono funzionali all'osservazione del transito dei messaggi e al monitoraggio di Service Level Agreement.

Un altro concetto che è importante definire è quello di *evento*, che sarà richiamato diverse volte nel proseguo di questo documento. Definiamo un evento come un'unità atomica di informazione che descrive un'interazione, significativa al monitoring del SLA, tra cliente e applicazione e che incapsula tutte le informazioni necessarie all'individuazione del consumatore del servizio, all'individuazione del SLA di riferimento, nonché alla valutazione del soddisfacimento dei requisiti imposti dal SLA individuato.

Pertanto, riteniamo che un evento debba comprendere le seguenti informazioni:

- l'identificativo della sessione HTTP che trasporta il messaggio SOAP;

- il numero di sequenza associato al messaggio SOAP;
- il namespace[1] che identifica il wsdl associato al servizio web;
- le informazioni necessarie ad identificare il cliente del servizio web (possibilmente attraverso autenticazione);
- la qualificazione del tipo di messaggio (richiesta o risposta);
- il timestamp relativo all'occorrenza dell'evento;
- i valori dei parametri dell'operazione[2].

Al fine di poter disporre di tutte le informazioni necessarie alla corretta descrizione degli eventi, abbiamo dovuto inserire nel header SOAP dei messaggi in transito le seguenti informazioni, non presenti di default:

username : il nome utente associato al cliente che invoca il servizio;

password : la parola d'ordine associata al cliente che invoca il servizio;

sequence number il numero di sequenza che individua univovamente un messaggio all'interno di una data sessione.

3.2 La modellazione del buon uso di un web service

Il contributo innovativo di questa tesi riguarda la modellazione del buon uso di un servizio web e la logica con cui l'esito dei controlli su questo aspetto si applica al monitoraggio degli altri parametri definiti nel SLA.

La considerazione che ha stimolato il concepimento di contributo è semplice ed intuitiva. L'imposizione di un SLA ha senso quando il servizio è

[1]Identificativo univocamente assegnato al descrittore di servizio web, definito con la sintassi propria degli URI.

[2]Al momento non sono impiegati da nessuna logica implementata, tuttavia essi possono essere utili all'implementare di logiche più complesse di controllo sul buon uso dei servizi

propriamente invocato; la ragione di questa affermazione risiede nel grado di impredicibilità che un uso improprio di un servizio introduce nella modellazione degli aspetti prestazionali e di disponibilità del servizio stesso. Riteniamo inoltre che un cliente che utilizza le operazioni messe a disposizione di un servizio secondo una semantica diversa da quella con cui esso è stato concepito non debba poter avanzare pretese sulle performance del servizio stesso, quando lo stesso uso improprio non si trasforma in un azione che inibisce le funzionalità del servizio stesso (denial of service).

Pertanto, al fine di dare una prima implementazione a questo tipo di controlli, abbiamo elaborato le seguenti ipotesi:

- *Le operazioni esposte da un web service possono essere fra loro in relazione di dipendenza causale*: Questa condizione può essere vera per servizi di business come, ad esempio, servizi per la prenotazione di posti o di biglietti, che consentono l'acquisto di un biglietto solamente dopo che questo è stato riservato, per evitare molteplici vendite dello stesso biglietto con il conseguente effetto di overbooking. Il prototipo sviluppato con questa tesi estende il linguaggio SLAng [SLE04] per incorporare la descrizione delle relazioni di causalità tra le operazioni di un dato web service; la specifica di tale relazione è formalizzata attraverso la descrizione di un automa a stati finiti deterministico costruito e interpretato come descritto in dettaglio nella Sezione 3.2.1;

- *Una corretta invocazione di un'operazione esposta da un web service richiede l'istanziazione di tutti i parametri formali definiti dall'operazione stessa*: questa ipotesi implica l'imposizione di un controllo sull'insieme di parametri passati durante l'invocazione dell'operazione. Il prototipo sviluppato in questa tesi verifica la coincidenza dei parametri per nome, ordine e numero.

Il nostro sistema di monitoring SLA interpreta le violazioni a queste regole di buon uso inibendo i controlli sul soddisfacimento degli altri vincoli

(e.g. throughput, tempo di servizio, disponibilità) espressi nel SLA per quel servizio , in particolare:

- quando vi è una violazione delle dipendenze causali, i controlli sui vincoli di SLA vengono sospesi fintanto che l'esecuzione dell'automa delle dipendenze causali non ritorna su uno stato "correttońń;
- quando vi è una violazione sull'istanziazione dei parametri formali, i controlli sui vincoli di SLA vengono sospesi fino all'invocazione dell'operazione successiva.

3.2.1 La descrizione delle dipendenze causali tra le operazioni di un servizio

Questa sezione presenta l'estensione progettata per arricchire un linguaggio di definizione di Service Level Agreements con la descrizione del buon uso del servizio; nel nostro caso, il linguaggio esteso è SLAng.

Prima di introdurre il formalismo che abbiamo impiegato, partiamo dalla definizione di un servizio web che si presta alla definizioni di dipendenze causali tra le invocazioni alle operazioni implementate. In particolare, abbiamo individuato il contesto di un'agenzia di prenotazione voli e abbiamo definito un web service che fornisse le seguenti operazioni, tipiche di un servizio di questo tipo:

`list`: servizio per la ricerca di voli;

`getDetails`: servizio per la visione dei dettagli del volo;

`reserve`: servizio per riservare temporaneamente un posto sul volo;

`confirm`: servizio per confermare l'acquisto del volo;

`printTicket`: servizio per ottenere i dettagli del biglietto acquistato.

Assumiamo che le dipendenze causali definite nella semantica del servizio siano le seguenti:

1. in ogni momento è corretto invocare l'operazione `list`;
2. in ogni momento è corretto invocare l'operazione `getDetails`;
3. in ogni momento è corretto invocare l'operazione `reserve`;
4. è corretto invocare l'operazione `confirm` quando l'operazione precedentemente invocata è `reserve`;
5. è corretto invocare l'operazione `printTicket` quando l'operazione precedentemente invocata è `confirm`.

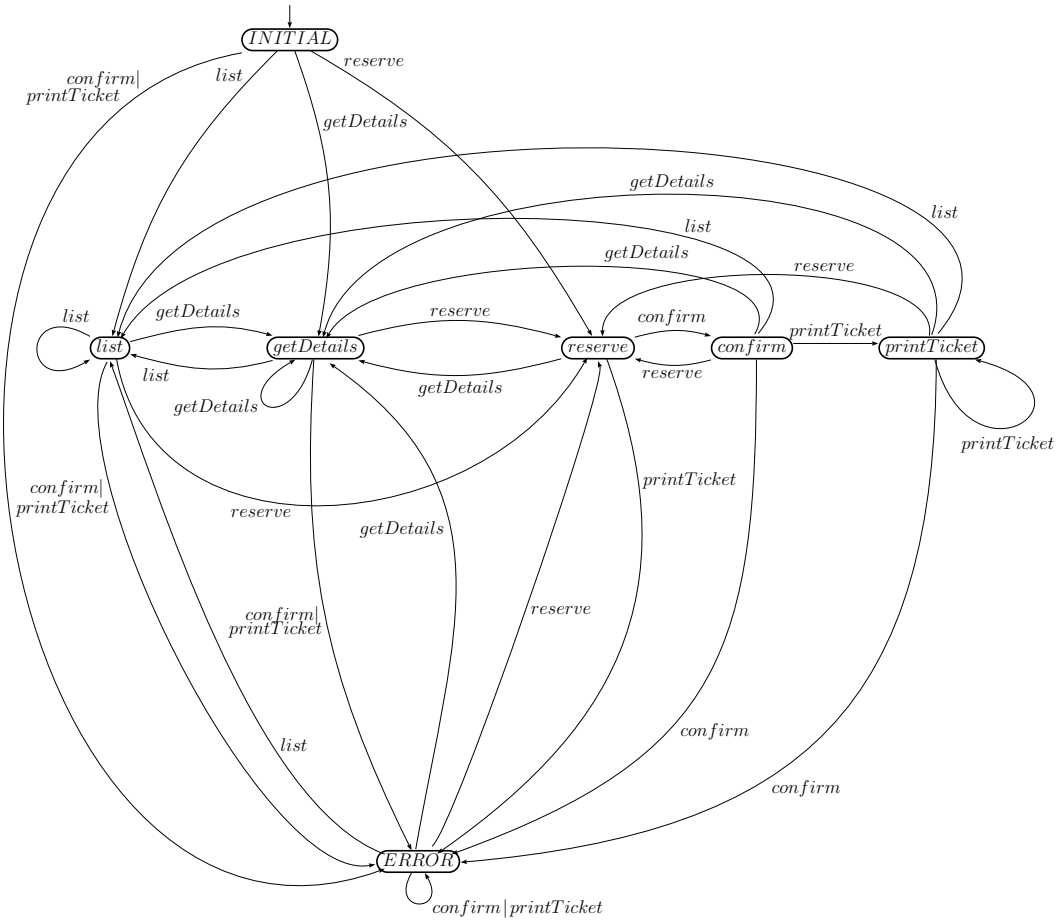

Figura 3.1: Automa delle dipendenze causali per il web service di esempio.

Utilizzando queste regole di dipendenza causale, è possibile trascrivere l'automa in figura 3.1. Questo automa ha un insieme di stati definito come descritto di seguito. Gli stati "corretti" sono definiti sulla base della relazione descritte sopra e sono collegati da archi etichettati con i nomi delle operazioni che consentono di transitare da uno stato ad un altro. Tutte le invocazioni che violano le dipendenze descritte sopra, conducono a stati "di errore" che evidenziano la violazione al buon uso del servizio. Dagli stati di errore si può transitare verso stati "corretti" soltanto attraverso l'invocazione di tutte e sole quelle operazioni che possono essere invocate in ogni momento.

Abbiamo quindi definito il linguaggio xml incluso in appendice A per trasformare la rappresentazione grafica dell'automa in una rappresentazione xml che possa essere incapsulata in un SLA e quindi interpretata dal nostro sistema di monitoring. Adottando tale formalismo, l'automa descritto sopra si traduce nel codice riportato nel riquadro 3.1.

Listing 3.1: Rappresentazione xml dell'automa in figura 3.1.

```
<tns:usageDescription xmlns:tns="http://www.eng.it/ws/SLA/usageAutomata"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <tns:state label="initial" type="initial">
    <tns:stateLink label="list">list</tns:stateLink>
    <tns:stateLink label="getDetails">getDetails</tns:stateLink>
    <tns:stateLink label="reserve">reserve</tns:stateLink>

    <tns:stateLink label="confirm">error</tns:stateLink>
    <tns:stateLink label="printTicket">error</tns:stateLink>
  </tns:state>
  <tns:state label="list">
    <tns:stateLink label="list">list</tns:stateLink>
    <tns:stateLink label="getDetails">getDetails</tns:stateLink>
    <tns:stateLink label="reserve">reserve</tns:stateLink>

    <tns:stateLink label="confirm">error</tns:stateLink>
    <tns:stateLink label="printTicket">error</tns:stateLink>
  </tns:state>
  <tns:state label="getDetails">
    <tns:stateLink label="list">list</tns:stateLink>
    <tns:stateLink label="getDetails">getDetails</tns:stateLink>
    <tns:stateLink label="reserve">reserve</tns:stateLink>

    <tns:stateLink label="confirm">error</tns:stateLink>
```

```
    <tns:stateLink label="printTicket">error</tns:stateLink>
  </tns:state>
  <tns:state label="reserve">
    <tns:stateLink label="list">list</tns:stateLink>
    <tns:stateLink label="getDetails">getDetails</tns:stateLink>
    <tns:stateLink label="reserve">reserve</tns:stateLink>
    <tns:stateLink label="confirm">confirm</tns:stateLink>

    <tns:stateLink label="printTicket">error</tns:stateLink>
  </tns:state>
  <tns:state label="confirm">
    <tns:stateLink label="list">list</tns:stateLink>
    <tns:stateLink label="getDetails">getDetails</tns:stateLink>
    <tns:stateLink label="reserve">reserve</tns:stateLink>
    <tns:stateLink label="printTicket">printTicket</tns:stateLink>

    <tns:stateLink label="confirm">error</tns:stateLink>
  </tns:state>
  <tns:state label="printTicket">
    <tns:stateLink label="list">list</tns:stateLink>
    <tns:stateLink label="getDetails">getDetails</tns:stateLink>
    <tns:stateLink label="reserve">reserve</tns:stateLink>
    <tns:stateLink label="printTicket">printTicket</tns:stateLink>

    <tns:stateLink label="confirm">error</tns:stateLink>
  </tns:state>
  <tns:state label="error" type="error">
    <tns:stateLink label="list">list</tns:stateLink>
    <tns:stateLink label="getDetails">getDetails</tns:stateLink>
    <tns:stateLink label="reserve">reserve</tns:stateLink>

    <tns:stateLink label="confirm">error</tns:stateLink>
    <tns:stateLink label="printTicket">error</tns:stateLink>
  </tns:state>
</tns:usageDescription>
```

3.3 L'architettura e i componenti

Questa sezione descrive la struttura dell'architettura che abbiamo individuato per soddisfare i requisiti presentati precedentemente.

L'immagine in figura 3.2 descrive l'architettura progettata e come questa si colloca rispetto alla comunicazione tra un attore (rappresentato nella figura

da un omino stilizzato) e un'applicazione (rappresentata dall'interfaccia a web service nella parte in alto a destra).

La nuvoletta attraversata dai messaggi scambiati tra attore e applicazione rappresenta la sonda degli eventi che è stata realizzata attraverso l'implementazione di un handler SOAP. Questo componente risiede sull'application server e il suo ruolo è quello di intercettare ogni messaggio SOAP, trasmesso da o verso l'applicazione, di estrarre da ogni messaggio i dati rilevanti alle attività di monitoring, e di comunicare tali dati al servizio di monitoring. Questo è l'unico componente dell'architettura che si pone a livello applicativo e interagisce in maniera sincrona con i servizi messi a disposizione dall'applicazione; pertanto, al fine di soddisfare i requisiti sulle performance che ci siamo inizialmente posti, questo componente ha la responsabilità di imporre la minima perdita di performance ai servizi applicativi. Per questa ragione, abbiamo deciso di sfruttare un canale di comunicazione basato sul protocollo UDP per consentire alla sonda degli eventi di comunicare efficientemente e in maniera asincrona gli eventi rilevati al servizio di monitoring. Nonostante, in un contesto generico, un protocollo di comunicazione a datagram non garantisca l'ordinamento dei messaggi inviati e non garantisca una e una sola consegna per ogni messaggio trasmesso, in un contesto locale il protocollo UDP riesce ad essere affidabile quanto il protocollo TCP senza imporre gli stessi vincoli prestazionali e di sincronia.

Il servizio di monitoring si sviluppa attraverso l'interazione di due componenti: il collezionista di eventi e il componente di monitoring degli SLA.

Il gestore di eventi è un componente capace di interpretare i messaggi UDP prodotti dalla sonda degli eventi e il cui funzionamento ha il duplice scopo di disaccoppiare l'arrivo degli eventi dalla loro elaborazione da parte del componente di monitoring e di implementare le funzionalità di tolleranza ai guasti tali da ridurre al minimo la probabilità di perdita di eventi a fronte di un guasto di tipo crash dell'application server. Entrambe queste funzionalità sono state realizzate attraverso una struttura a coda che ha un backup persistente che viene precaricato ad ogni riavvio del servizio.

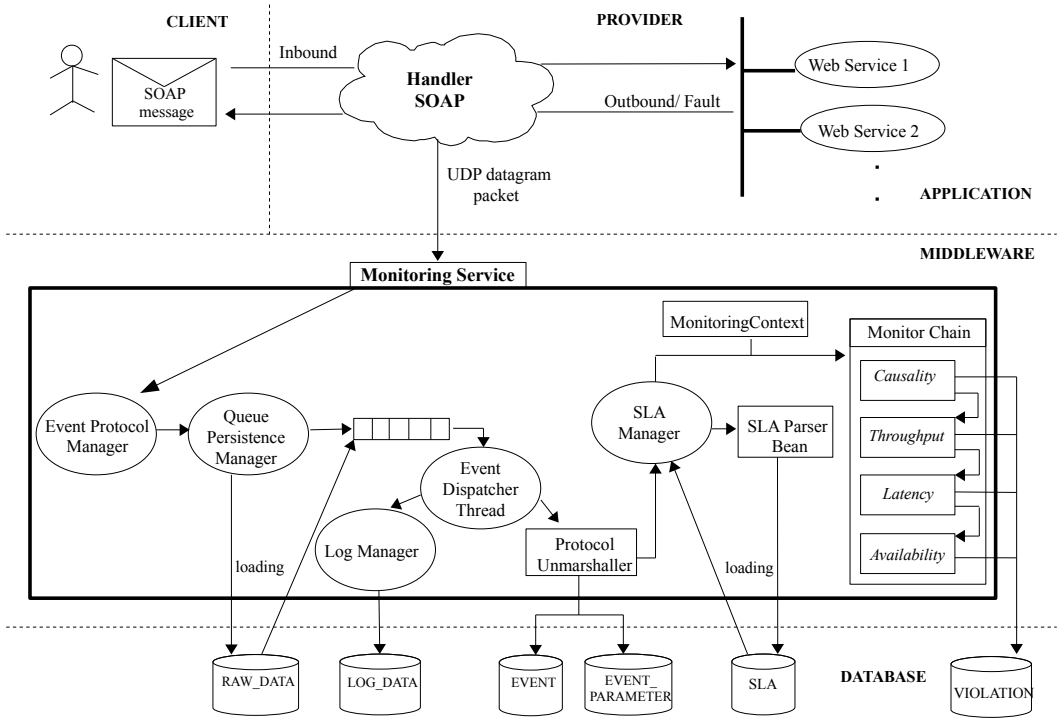

Figura 3.2: Architettura del sistema

Infine, il componente di monitoring degli SLA si preoccupa di implementare, nell'ordine, le seguenti funzioni che sono fondamentali per l'implementazione del monitoring del SLA:

- prelevare ogni evento, in ordine di arrivo, dall'interfaccia messa a disposizione dal collezionista di eventi;
- controllare che l'evento soddisfi i requisiti che ne determinano la validità ai fini delle attività di monitoraggio;
- inserire l'evento nella banca dati;
- valutare l'impatto del nuovo evento sullo stato di esecuzione della relativa istanza di SLA, evidenziando eventuali violazioni.

L'ultimo aspetto che introduciamo in questa sezione, non ultimo per importanza, riguarda la scelta del linguaggio che abbiamo deciso di adottare per la descrizione dei Service Level Agreements. Dall'analisi dello stato dell'arte in merito, abbiamo deciso di utilizzare la notazione SLAng perché offre maggiori garanzie di estendibilità e quindi si presta meglio all'estensione con la specifica dei requisiti sul buon uso nonché alle possibili estensioni, attraverso lavori futuri, sul numero e sulla tipologia dei parametri analizzabili dal sistema di monitoring.

Le prossime sezioni descriveranno in dettaglio ciascuno di questi componenti, partendo dalla banca dati che offre il supporto per la persistenza delle informazioni.

3.4 La banca dati

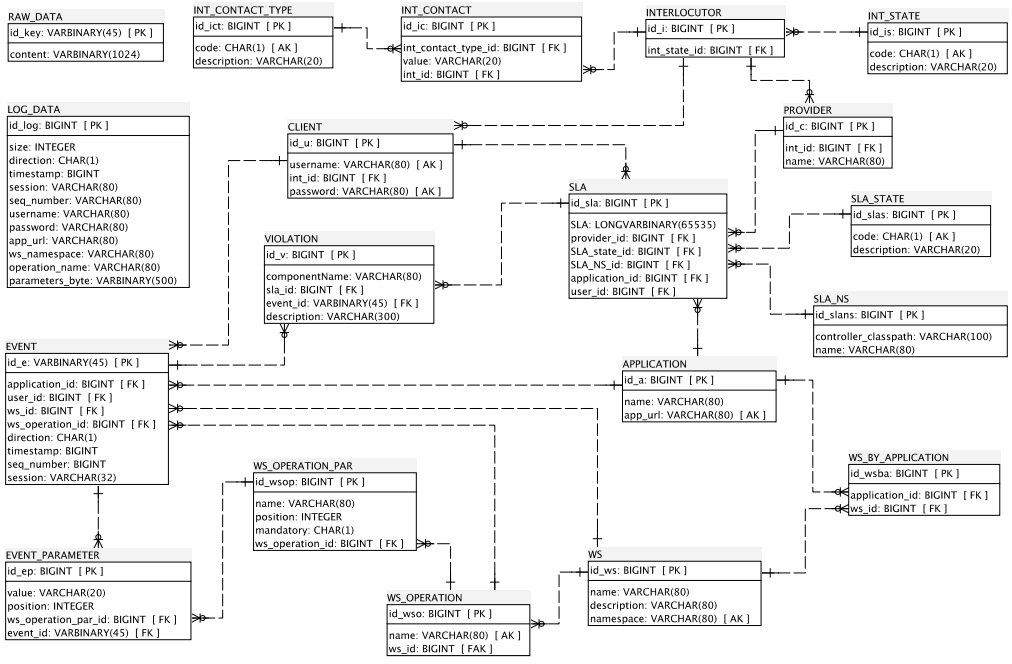

Figura 3.3: Rappresentazione del modello E-R

L'immagine in figura 3.3 mostra il modello entità-relazione adottato nel nostro sistema. Per una lettura semplificata lo schema viene di seguito riassunto in quattro gruppi.

Il primo comprende tutte le entità che vengono popolate alla ricezione dei pacchetti UDP inviati dalla sonda degli eventi, quindi per ogni messaggio scambiato tra cliente e fornitore del servizio:

RAW_DATA rappresenta l'insieme di tutti i pacchetti UDP ricevuti dal socket del servizio. Gli attributi id_key e content indicano rispettivamente l'intestazione e il contenuto del pacchetto come descritto nel protocollo di codifica in sezione 3.6. L'entità viene utilizzata allo scopo di mantenere temporaneamente in memoria tutti i pacchetti UDP in modo da consentire, qualora l'application server interrompesse la sua regolare attività, un ri-caricamento in memoria principale dei pacchetti al riavvio del sistema;

LOG_DATA rappresenta l'insieme dei pacchetti ricevuti dal servizio di monitoring con tutte le informazioni introdotte dalla sonda in modo da facilitarne la lettura per un'eventuale analisi dei dati;

EVENT e EVENT_PARAMETER rappresentano rispettivamente l'entità degli eventi da monitorare e i valori dei parametri dell'operazione del web service invocata. EVENT_PARAMETER è in relazione con l'entità WS_OPERATION_PAR perché rappresenta l'insieme dei valori dei parametri dell'operazione di uno specifico web service. EVENT, invece, è in relazione con tutte quelle entità necessarie ai fini del monitoraggio: CLIENT, APPLICATION, WS, WS_OPERATION. Gli attributi direction, timestamp, seq_number e session indicano rispettivamente la direzione del messaggio SOAP intercettato (inbound, outbound o fault), il timestamp, il numero di sequenza e la sessione HTTP. Le due entità sono tra loro in relazione uno a molti poiché un evento potrebbe trasportare più valori dei parametri indicati nell'operazione invocata;

VIOLATION rappresenta l'insieme delle violazioni che si sono verificate durante la fase di monitoring. Ogni violazione è in relazione con le entità SLA e EVENT in modo da individuare immediatamente l'evento che ha scatenato l'infrazione con lo specifico documento violato. L'attributo componentName e description indicano rispettivamente il relativo componente del monitoring che ha infranto una specifica qualità del servizio e la descrizione della causa che ha scatenato la violazione;

Il secondo gruppo comprende tutte le entità che riguardano i soggetti coinvolti nel monitoraggio:

INTERLOCUTOR rappresenta il soggetto coinvolto nel monitoraggio di uno SLA. Nel nostro caso i soggetti sono rappresentati da un cliente CLIENT o un fornitore del servizio PROVIDER;

INT_CONTACT descrive il contatto utilizzato per notificare una possibile violazione dello SLA associato all'interlocutore. L'entità è in relazione con **INT_CONTACT_TYPE** che ne definisce il tipo attraverso il valore indicato nell'attributo code. Utile nel caso si volessero notificare i soggetti coinvolti nello SLA violato;

INT_STATE descrive lo stato di un interlocutore. Nel nostro caso forniamo due diversi stati: Attivo e Sospeso, espressi rispettivamente col il valore 0 e 1 nell'attributo code. Quando è in stato sospeso si vuole sospendere momentaneamente il servizio di monitoring ad un determinato interlocutore;

CLIENT rappresenta il cliente coinvolto in uno specifico SLA;

PROVIDER rappresenta il fornitore del servizio web a cui è associato uno specifico SLA;

Il terzo gruppo riguarda le entità che descrivono gli SLA:

SLA rappresenta il Service Level Aggrement stipulato tra cliente e fornitore del web service. Il valore dell'attributo SLA contiene il contratto

stipulato espresso attraverso uno specifico linguaggio. L'entità è in relazione con CLIENT e PROVIDER che ne rappresentano le parti coinvolte e APPLICATION che descrive l'applicazione associata per quello specifico Service Level Agreement. Uno SLA ha inoltre uno stato SLA_STATE e uno specifico linguaggio SLA_NS con i quali è relazionato;

SLA_STATE descrive lo stato di uno SLA. Nel nostro caso forniamo due diversi stati: Attivo e Sospeso, espressi rispettivamente col il valore 0 e 1 nell'attributo code. Quando è in stato sospeso si vuole sospendere momentaneamente il servizio di monitoring per un determinato SLA;

SLA_NS rappresenta il linguaggio SLA adottato. L'attributo controller_-classpath descrive la classe Java utilizzata per la lettura e recupero dei parametri da monitorare dall'XML contenuto nell'entità SLA. Nel nostro caso il linguaggio adottato è SLAng di cui abbiamo implementato l'apposito parser per il recupero dei parametri da monitorare;

Il quarto e ultimo gruppo comprende tutte le entità che riguardano il web service:

APPLICATION descrive l'applicazione sviluppata dal fornitore che realizza i web service. L'entità è relazionata con **WS_BY_APPLICATION** per permettere un'associazione di tipo molti a molti in quanto un'applicazione può avere più web service e lo stesso web service essere realizzato su più applicazioni;

WS descrive il web service sviluppato e disponibile, caratterizzato da un namespace univoco, un nome e una descrizione;

WS_OPERATION rappresenta l'operazione del web service che può essere invocata dal cliente. L'entità è in relazione uno a molti con **WS_OPERATION_PAR** che ne rappresenta il parametro;

3.5 La sonda degli eventi

Come anticipato nella sezione 3.1, un handler è un'astrazione capace di intercettare i messaggi SOAP in transito da e verso un web service. Pertanto, abbiamo impiegato questa astrazione come base per l'implementazione della sonda degli eventi.

Abbiamo quindi progettato la sonda degli eventi come un handler capace di estrarre dai messaggi SOAP in transito le informazioni necessarie a caratterizzare gli eventi associati ad ogni singolo messaggio, diretto al o prodotto dal servizio, e capace di inviare tali informazioni al gestore degli eventi sfruttando un protocollo appositamente progettato per avere un impatto ridotto sulle performance dell'applicazione monitorata. In particolare, l'uso di un protocollo appositamente progettato su UDP garantisce la comunicazione asincrona delle informazioni relative agli eventi nonché la riduzione del tempo di trasmissione di tali informazioni.

Al fine di individuare i problemi di disponibilità, sono state integrate nella sonda logiche aggiuntive per individuare e censire occorrenze di fault SOAP o di status code HTTP corrispondenti ad errori.

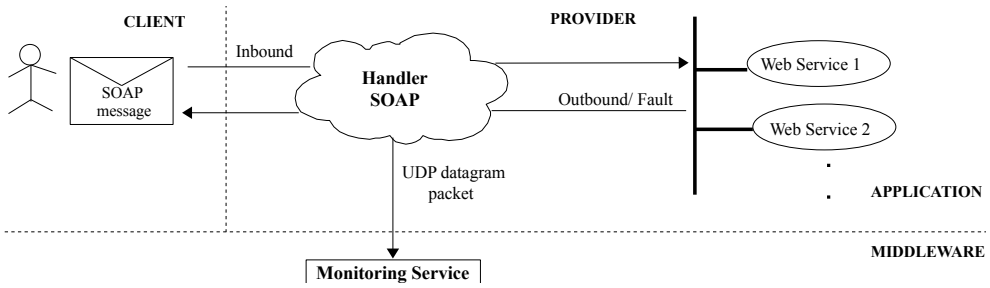

Figura 3.4: La sonda degli eventi

Una volta raccolte tutte le informazioni necessarie alla caratterizzazione dell'evento, viene generato un datagram UDP contenente tali informazioni, descritte secondo il protocollo presentato nella sezione 3.6.

3.6 Il protocollo di codifica e trasmissione degli eventi

Questa sezione descrive in dettaglio il protocollo che abbiamo progettato per la trasmissione via UDP delle informazioni sugli eventi. Ogni messaggio di protocollo è composto da un'intestazione e da un contenuto; l'intestazione contiene un'identificatore univoco dell'evento mentre il contenuto contiene tutte le informazioni elencate al 3.1.

Il protocollo presentato in questa sezione ha l'obiettivo di trasportare al servizio di monitoring le informazioni che descrivono gli eventi intercettati dalla sonda.

Abbiamo deciso di dividere il messaggio di protocollo in due parti: una, l'intestazione, che comprende quelle informazioni che identificano univocamente l'evento catturato e l'altra, il corpo, che comprende tutte quelle informazioni a supporto che sono funzionali alle operazioni del servizio di monitoring.

Il protocollo prevede due diversi tipi di messaggio, uno associato ai messaggi in ingresso al servizio e uno associato ai messaggi in uscita. Il primo byte di ogni messaggio qualifica la direzione associata all'evento e, di conseguenza, identifica la struttura del messaggio.

L'intestazione di ogni messaggio è lunga 49 bytes e contiene la chiave dell'evento, che comprende le seguenti informazioni:

- direzione del messaggio (1 byte);
- il numero di sessione HTTP (32 bytes);
- il numero di sequenza associato al messaggio SOAP (8 bytes).
- il timestamp associato alla rilevazione dell'evento (8 bytes);

La struttura del contenuto varia a seconda della tipologia di evento, se in ingresso al web service o in uscita.

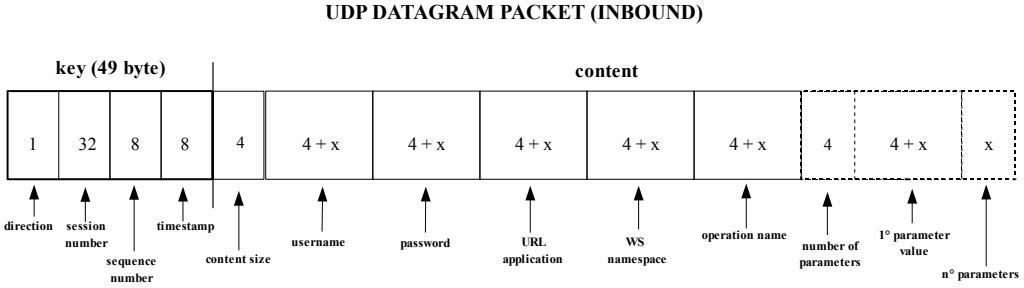

Figura 3.5: Pacchetti UDP: inbound

Un contenuto che descrive un evento associato ad un messaggio in ingresso è strutturato come segue (vedi fig. 3.5 per la descrizione grafica):

- dimensione totale del contenuto in bytes (4 bytes);
- username (4 + x bytes);
- password (4 + x bytes);
- application url (4 + x bytes);
- web service namespace (4 + x bytes);
- operation name (4 + x bytes);
- parametro 1 (4 + x bytes);
- parametro i (4 + x bytes);
- parametro n (4 + x bytes).

Come si può osservare dalla rappresentazione grafica dei protocolli, i parametri che sono dimensionati con la notazione $4 + x$ sono tutti quei parametri che non hanno una dimensione massima prefissata e che sono quindi preceduti da 4 byte per indicare il numero di byte successivi contenenti l'informazione.

Un contenuto che descrive un evento associato ad un messaggio fault è strutturato come segue (vedi fig. 3.6 per la descrizione grafica):

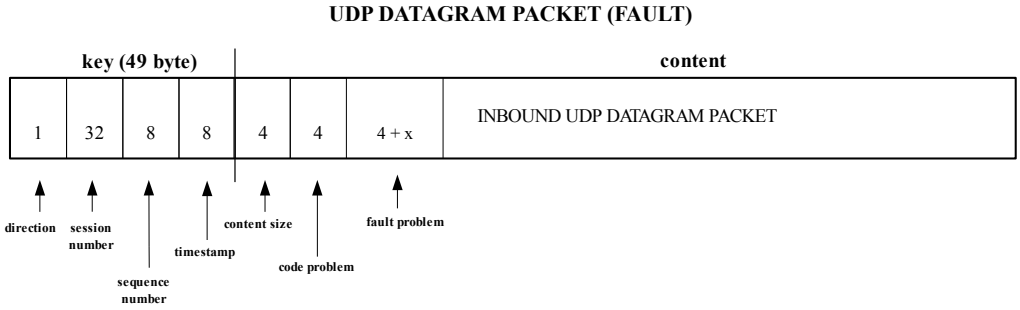

Figura 3.6: Pacchetti UDP: fault

- dimensione totale del contenuto in bytes (4 bytes);
- codice del problema (4 bytes);
- descrizione del problema (4 + x bytes).

Per gli eventi associati a messaggi in uscita, la struttura del contenuto è modellata in maniera simile a quella degli eventi in ingresso.

3.7 Il gestore della coda di eventi

Il gestore degli eventi è quel componente dell'architettura che:

- implementa il lato server del protocollo che abbiamo appena descritto per gestire la ricezione degli eventi prodotti dalla sonda;
- interpreta i messaggi di protocollo istanziando, scartando i messaggi non validi e trasformando quelli validi in oggetti di tipo Evento che vengono poi passati al componente di monitoring degli SLA;
- implementa meccanismi di tolleranza ai guasti per scongiurare che un guasto all'application server possa comportare la perdita di eventi.

L'immagine in figura 3.7, descrive la logica e il flusso dei dati di questo componente, implementati attraverso la combinazione delle funzionalità dei seguenti elementi:

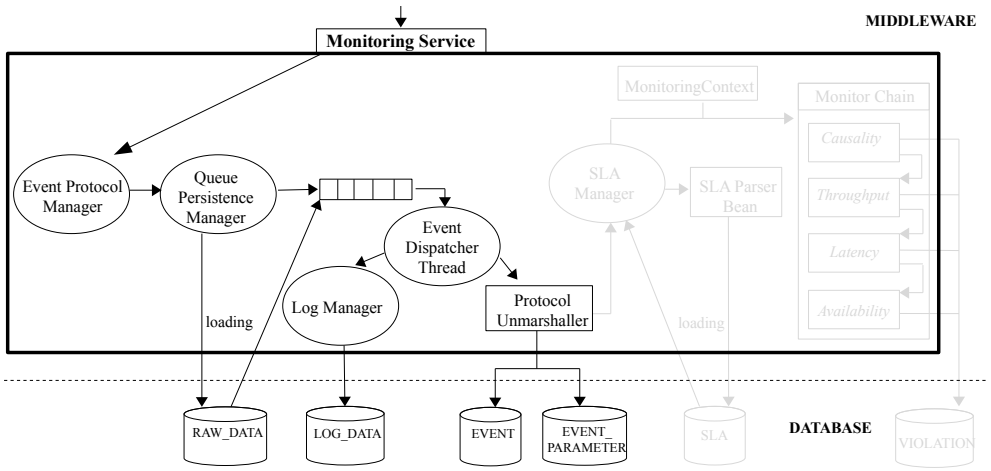

Figura 3.7: Il gestore della ricezione degli eventi

- **Event Protocol Manager:** Elemento che implementa il lato server del protocollo descritto nella sezione 3.6;
- **Queue Persistence Manager:** Elemento che implementa la gestione della persistenza associata al buffer degli eventi non ancora elaborati dal componente di monitoring SLA e che inserisce i messaggi nell'apposita struttura dati a coda;
- **Event Dispatcher Thread:** Thread che rimane in attesa di nuovi messaggi di evento sulla coda dei messaggi e responsabile di invocare i servizi del Protocol Unmarshaller e di passare gli eventi così prodotti al componente di monitoring di SLA;
- **Protocol Unmarshaller:** Elemento responsabile dell'interpretazione dei messaggi di protocollo ricevuti e della creazione delle relative istanze del tipo Evento.

All'arrivo di un evento, il gestore della coda degli eventi implementa la seguente logica:

1. Il Event Protocol Manager riceve un messaggio di protocollo, ne esamina la corretta costruzione e lo passa al Queue Persistence Manager;

2. Il Queue Persistence Manager, alla ricezione del messaggio del protocollo degli eventi, si occupa di inserire l'evento nella banca dati, in una tabella dedicata al mantenimento dei messaggi Event Protocol ricevuti, e di inserire il messaggio nella coda degli eventi;

3. Il Event Dispatcher Thread è in attesa di nuovi eventi sulla coda degli eventi. All'inserimento di un nuovo evento questo thread, se è in attesa, viene attivato. Questo thread preleva l'evento dalla coda e richiama i servizi del Protocol Unmarshaller per ottenere un'istanza dell'oggetto Evento da passare al componente di monitoring SLA in caso dell'eseucuzione con successo dei test di coerenza sul contenuto.

4. Il Protocol Unmarshaller, richiamato dal Event Dispatcher Thread, interpreta il messaggio di evento e produce un'istanza di oggetto Evento correttamente popolata.

I test di coerenza implementati sul contenuto delle istanze degli oggetti Evento, e che ne abilitano il passaggio al componente di SLA Monitoring, sono i seguenti:

- **Analisi dei tipi:** verifica la correttezza dei tipi e delle dimensioni di ogni valore presente nel messaggio
  - il numero di sequenza e il timestamp sono rappresentati come numeri;
  - tutti i valori presenti non sono vuoti o nulli;
- **Controllo dei vincoli di integrità referenziale:** verifica la compatibilità dei dati riportati dall'evento con quelli gestiti dalla banca dati
  - sui valori riportati nel campo direzione (inbound, outbound o fault);
  - sul cliente identificato attraverso l'autenticazione;

  - sull'applicazione associata all'url riferito;
  - sul web service associato al namespace ed all'applicazione individuata;
  - sull'operazione associata al nome e al web service individuato;
  - sui parametri associati all'operazione individuata;

- **Analisi sull'imponibilità dei requisiti di SLA:** esegue i seguenti controlli

  - che lo stato del cliente sia attivo;
  - che che lo stato del SLA sia attivo;
  - che l'evento non sia ancora stato analizzato dal monitoring SLA;
  - che l'evento per una stessa sessione, web service e cliente abbia un timestamp e un numero di sequenza maggiore rispetto a quello dell'ultimo evento monitorato;
  - che l'arrivo di un evento outbound, o di un evento fault, succeda quello del corrispondente evento inbound.

Superato il controllo di coerenza, gli eventi vengono passati al componente di SLA monitoring. Si noti che il controllo sui parametri verifica solamente la presenza dei parametri richiesti dal metodo senza entrare in merito del contenuto di questi; sarà materia di lavori futuri l'estensione di questo controllo con l'abilitazione di verifiche sintattiche sul formato dei valori dei parametri specificati nelle invocazioni.

Al riavvio del server dopo un fallimento, il Queue Persistence Manager recupera dalla banca dati lo stato di elaborazione degli eventi e ripopola lo stato della coda degli eventi in modo da riprendere l'esecuzione delle funzioni di monitoring dall'ultimo stato coerente prima dell'occorrenza del guasto; questa operazione avviene durante lo startup dell'application server e quindi prima dell'avvio dei contesti applicativi.

3.8 Il componente di monitoring del SLA

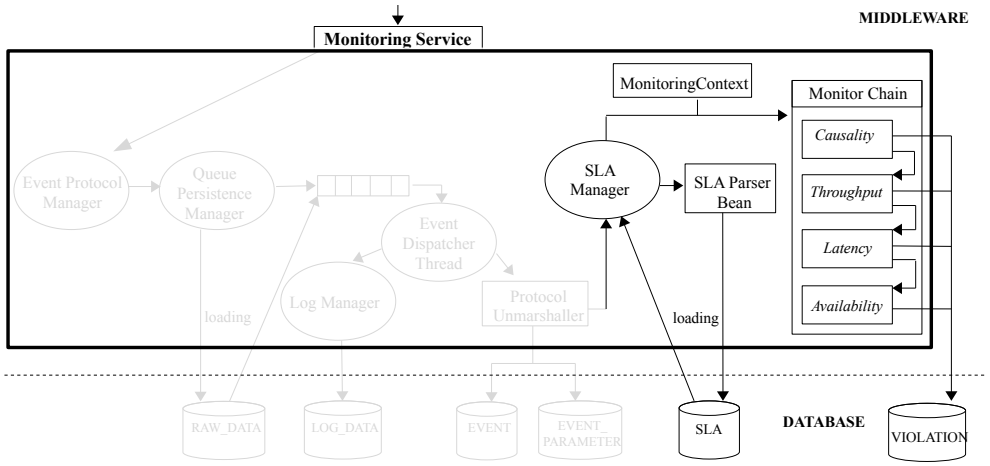

Figura 3.8: Il componente di monitoring del SLA

Questo componente è il cuore di questa tesi e si occupa di implementare tutti i controlli per la verifica della corretta esecuzione degli SLA configurati attraverso la banca dati. In figura 3.8 si descrive in dettaglio la struttra di questo componente e la logica che esso implementa.

Prima di presentarvi in dettaglio questo componente, ci preme spendere qualche parola per descrivere un ragionamento che sta alla base della sua progettazione. Al fine di limitare l'uso della memoria richiesta dal componente, abbiamo deciso di implementare tutti gli elementi dedicati al monitoring, ovvero quei sottocomponenti che realizzano i controlli sull'esecuzione dei requisiti di SLA, come oggetti stateless e di mantenere in opportune strutture dati le informazioni relative allo stato di implementazione degli SLA. Per questa ragione, abbiamo isolato due tipologie di oggetto: *SLARequirementsBean* e il *MonitoringContext*. Il SLARequirementsBean è l'astrazione del SLA nello specifico contesto di monitoring. Questa classe serve per disaccoppiare l'implementazione dei controlli sul mantenimento dei requisiti dal linguaggio mediante il quale tali requisiti sono descritti. Un elemento che sarà meglio descritto in seguito, il SLABean, fornisce l'interfaccia che è in

grado di ottenere un'istanza di SLARequirementsBean dato un SLA descritto attraverso uno specifico linguaggio. Il MonitoringContext è responsabile di mantenere le informazioni sullo stato di esecuzione di un'istanza di SLA. In pratica, questo oggetto viene utilizzato come stato condiviso tra tutti gli elementi di monitoring; essi utilizzano le interfacce di questo oggetto per immagazzinare le informazioni sullo stato attuale dell'esecuzione dei controlli sul SLA.

Gli elementi che costituiscono questo componente sono i seguenti:

- **SLA Manager:** questo elemento implementa l'interfaccia esposta dal componente, fornendo al gestore della coda degli eventi il metodo per la notifica di nuovi eventi. Per ogni evento notificato dal gestore della coda degli eventi, questo componente recupera (i) l'istanza di MonitoringContext associato all'istanza di SLA e (ii) l'istanza di SLARequirementsBean relative all'istanza di SLA associato all'evento in corso di analisi. Una volta recuperati, questo elemento richiama i servizi del Monitor Chain passando l'evento, il contesto di monitoring e il bean contenente i requisiti di SLA che devono essere osservati;

- **SLA Parser Bean:** Questo elemento è presente in tante istanze quanti sono i linguaggi SLA gestiti dal sistema di monitoring. Ogni singolo linguaggio di SLA ha una propria implementazione del bean che abilita l'interpretazione degli SLA scritti nel linguaggio associato e l'estrazione di tutti i vincoli ivi definiti e monitorabili da questo componente;

- **Monitor Chain:** Questo elemento è il contenitore di tutti quei sottocomponenti responsabili del monitoring dei requisiti di SLA. Esso offre al SLA Manager un unico punto di accesso per scatenare, all'arrivo di un evento, l'esecuzione dei i controlli su tutti quei requisiti definiti nel SLA e implementati dalla nostra architettura. Nell'implementazione attuale, questo elemento abilita l'esecuzione dei controlli sul buon uso del servizio, sul throughput, sulla latenza, e sulla disponibilità. Ogni violazione individuata dai sottocomponenti di monitoring

viene memorizzata nella banca dati, nell'apposita tabella relativa alle violazioni.

Durante l'istanziazione del servizio, all'avvio dell'application server, il SLA Manager esegue le seguenti azioni:

- precarica dalla banca dati tutti gli SLA attivi, mantenendone le informazioni in opportune strutture dati;
- fornisce gli SLA caricati all'apposito Parser, denominato SLABean, in base al linguaggio utilizzato per definire lo SLA (SLAng nel nostro caso);
- realizza un insieme di contesti, uno per ogni SLA distinto, contenenti i diversi parametri individuati negli SLA;

Un contesto contiene tutte le informazioni sui parametri da monitorare individuati nei diversi SLA.

I parametri che abbiamo preso in considerazione, seguono le definizioni fornite dall'articolo [RSE08]:

Thoughput il numero di richieste che un client può effettuare entro un limite di tempo;

Latenza la risposta del web service deve seguire la richiesta entro un limite di tempo;

Disponibilità numero di errori che l'applicazione può avere entro un limite di tempo.

Le sezioni che seguono forniscono una descrizione dettagliata di ogni sottocomponente di monitoring realizzato nel nostro sistema.

3.8.1 Causalità

Il primo componente di monitoraggio, chiamato Causality Monitor verifica l'aderenza dell'uso del servizio rispetto alle indicazioni del buon uso specificate nel SLA.

Per un web service, abbiamo individuato la definizione del buon uso sfruttando la relazione di causalità tra le operazioni messe a disposizione; questa relazione ci consente di specificare che ogni operazione può essere causalmente seguita soltanto da uno specifico sottoinsieme delle operazioni messe a disposizione dal servizio stesso. Per implementare questo requisito, il SLA è stato arricchito con una descrizione di tale relazione, realizzata attraverso la descrizione di un automa deterministico a stati finiti che ha tanti stati quante sono le operazioni e che, da ogni stato, ha tante transizioni quante sono le operazioni che causalmente possono seguire. L'etichetta di una transizione coincide con il nome dell'operazione invocata porta l'automa nello stato col nome di tale operazione.

Questo sottocomponente di monitoring prende in esame solamente gli eventi di tipo inbound, ovvero quelli che corrispondono a invocazioni del servizio. Gli eventi outbound superano sempre questo stadio del monitoring senza generare violazioni.

In base all'operazione precedentemente richiesta, memorizzata in un'apposita variabile all'interno del monitoring context, il Causality Monitor verifica se l'evento attualmente in esame corrisponde alle transizioni legittime descritte dall'automa precedentemente descritto, rispetto allo stato attuale riferito dal monitoring context. In caso questo evento descriva l'invocazione di una delle operazioni previste, l'evento passa questo stadio del monitoring senza generare violazioni, altrimenti una violazione viene generata.

Questo è l'unico sottocomponente di monitoring che, all'individuazione delle violazioni, inibisce l'esecuzione dei componenti di monitoring successivi nella catena impostando l'apposito flag definito nel monitoring context.

3.8.2 Throughput

Tale requisito è imposto per evitare l'uso eccessivo di un servizio da un cliente, che potrebbe causare un degrado della qualità del servizio offerto e, di conseguenza, una violazione degli obblighi della latenza e della disponibilità. Come spiegato nell'articolo [RSE08] attraverso l'utilizzo di automi

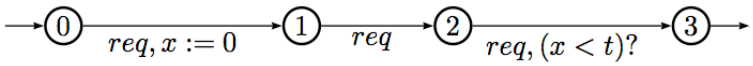

Figura 3.9: Automa temporizzato per le violazioni del throughput

temporizzati è possibile eseguire il monitoraggio in modo efficiente. L'automa temporizzato viene creato in base ai requisiti da monitorare definiti nel SLA associato.

I controlli di questo sottocomponente di monitoring vengono eseguiti nel solo caso in cui il Causality Monitor non abbia rilevato alcuna violazione al buon uso del servizio.

In figura throughput viene rappresentato un automa che accetta tutti gli eventi inbound (etichettati con req) in cui tre richieste si verificano entro un unità di tempo t. Gli eventi che non verificano le condizioni di tempo scatenano una violazione dei requisiti del throughput descritto in precedenza in questa sezione.

3.8.3 Latenza

Questo componente prende in esame l'evento inbound e il relativo evento outbound prodotto per la stessa operazione invocata dal cliente. Attraverso la differenza dei tempi dei due eventi viene individuata l'eventuale violazione della latenza associata al SLA.

I controlli di questo sottocomponente di monitoring vengono eseguiti nel solo caso in cui il Causality Monitor non abbia rilevato alcuna violazione al buon uso del servizio.

3.8.4 Disponibilità

Un'altra serie di requisiti per i web service include vincolare il numero di errori accettabili dall'applicazione. Nel nostro caso questo componente prende in esame tutti i messaggi di fault generati dall'applicazione e utiliz-

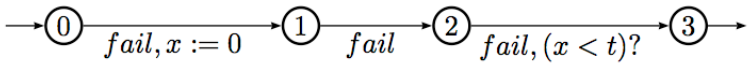

Figura 3.10: Automa temporizzato per le violazioni della disponibilità

zando un automa temporizzato, come nel caso del throughput, individua una possibile violazione del parametro di disponibilità indicato nello SLA.

I controlli di questo sottocomponente di monitoring vengono eseguiti nel solo caso in cui il Causality Monitor non abbia rilevato alcuna violazione al buon uso del servizio.

In figura 3.10 viene rappresentato un automa che accetta tre occorrenze di fallimento (etichettati con fail) entro un unità di tempo t. L'automa in figura corrisponde a un'istanza del pattern di disponibilità descritta in precedenza in questa sezione; da notare come il numero degli stati dell'automa è uguale al numero di eventi da analizzare + 1.

Capitolo 4

Implementazione

In questo capitolo mostreremo gli aspetti fondamentali che riguardano lo sviluppo del progetto. In particolare verrà presentata una sezione per:

- illustrare il framework implementativo;
- descrivere l'organizzazione del progetto;
- fornire la guida per l'utilizzo del servizio realizzato.

4.1 Framework implementativo

L'intero progetto è stato scritto in Java che tra i vantaggi figura la portabilità, la compatibilità con qualsiasi ambiente operativo e l'automatizzazione della gestione di aspetti di sviluppo, come ad esempio l'uso della memoria, le cui gestioni attraverso linguaggi di programmazione meno evoluti sarebbero a carico dello sviluppatore. Inoltre industrialmente, è di fatto il linguaggio maggiormente utilizzato per l'implementazione di applicazioni di tipo enterprise.

Per la realizzazione dei sorgenti abbiamo scelto *Eclipse* Helios 3.6, una tra le più diffuse piattaforme di sviluppo, che attraverso il plug-in Subversion ci ha permesso di mantenere il progetto in un archivio SVN residente sui server del dipartimento di Scienze dell'Informazione. Attraverso il plug-in JBoss

Tools versione 3.2, invece, è stato possibile creare l'ambiente di sviluppo per la realizzazione del nostro servizio di monitoring per l'application server JBoss nella versione 6.

Il linguaggio che abbiamo utilizzato per definire gli SLA è SLAng. Gli autori del linguaggio [SLE04] hanno messo a disposizione un editor, realizzato come un plugin per Eclipse, per abilitare la definizione di service level agreements usando loro linguaggio e sfruttando i vantaggi della comoda interfaccia grafica del IDE.

Per migliorare le caratteristiche di manutenibilità, evolvibilità e portabilità abbiamo deciso di adottare la tecnica di programmazione ORM (Object-Relational Mapping) utilizzando la piattaforma middleware Hibernate che al momento risulta il più diffuso e il più evoluto tra i vari ORM disponibili per il linguaggio Java.

Per consentire, infine, la creazione e manipolazione efficiente di database abbiamo utilizzato il DBMS MySQL nella versione 5.5.

4.2 Struttura del progetto

Il sistema di monitoring presentato in questa tesi è stato sviluppato attraverso due progetti Java. Il primo, chiamato `JBossWSHandler`, ha l'obiettivo di realizzare la sonda degli eventi: il componente dell'architettura che risiede a livello applicativo e descritto in 3.5. Il secondo progetto, chiamato `JBossMonitoring`, ha l'obiettivo di realizzare il componente residente a livello middleware che si occupa della gestione della coda degli eventi e del monitoring di SLA, trattati rispettivamente nelle sezioni 3.7 e 3.8.

4.2.1 La sonda degli eventi

Per realizzare la nostra sonda degli eventi ci siamo avvalsi del framework JBoss WS Native integrato a JBoss che consente lo sviluppo di web service. JBoss WS native si basa sulle nuove specifiche della Sun Microsystems chiamate JAX-WS ed integrate nella piattaforma Java EE.

La nostra sonda è una classe Java, chiamata `FilterHandler` e composta da 378 righe di codice, che consente di pre-elaborare il messaggio SOAP di richiesta prima che raggiunga il servizio invocato e di post-elaborare il messaggio SOAP di risposta prima che venga restituito al client.

Questa logica sincrona di elaborazione del messaggio causa l'introduzione di overhead; la sonda ferma il normale proseguimento del messaggio intercettato fino alla terminazione della sua elaborazione.

I messaggi SOAP intercettati dalla nostra sonda e trasformati in `SOAPMessageContext` vengono gestiti da un apposito metodo che consente di individuare le informazioni rilevanti ai fini del monitoring. In particolare:

`handleInbound` gestisce i messaggi in entrata o inbound;

`handleOutbound` gestisce i messaggi in uscita o outbound;

`handleFault` gestisce i messaggi fault SOAP o di status code HTTP corrispondenti ad errori.

Per ogni direzione, la sonda degli eventi struttura in modo differente il messaggio da trasportare al servizio di monitoring in base al protocollo di codifica descritto in 3.6. In tabella 4.1 indichiamo il valore del byte direzione utilizzato per l'identificazione del messaggio SOAP in modo da permetterne la corretta gestione.

Direzione	
Inbound	0
Outbound	1
Fault	2

Tabella 4.1: Valore del byte relativo alla direzione del messaggio SOAP.

Nel frammento di codice in 4.1 mostriamo le operazioni che vengono compiute all'interno del metodo `handleInbound` in modo da sottolineare la rapidità di elaborazione della sonda per ogni messaggio intercettato.

Le righe di codice dal 24 al 81 mostrano la creazione del messaggio di protocollo di trasmissione degli eventi, composto dalle informazioni contenute in `SOAPMessageContext` con eccezione della direzione, del timestamp e della dimensione del content del pacchetto.

Le righe di codice dal 87 al 91 mostrano l'inserimento della chiave del pacchetto inbound nell'header `HTTPServletResponse` in modo da associarlo al relativo messaggio outbound che verrà successivamente intercettato.

Infine la porzione di codice indicata dalla riga 95 al 97, mostra l'invio del pacchetto in modo asincrono al gestore della coda di eventi descritto in 3.7. Il ritorno del booleano `true` indicato nella riga 124 consente alla sonda di far riprendere il normale proseguimento del messaggio intercettato.

Listing 4.1: Gestione del messaggio SOAP Inbound

```
1  private boolean handleInbound(SOAPMessageContext msg){
2
3      DatagramSocket clientSocket=null;
4      try {
5        /*  recupero del messaggio SOAP */
6        HttpServletRequest req =
               (HttpServletRequest)msg.get(MessageContext.SERVLET_REQUEST);
7        SOAPMessage soapMessage =  msg.getMessage();
8
9        /*  verifica dell'esistenza di un SOAP header e di un SOAP body
10        *  con il recupero degli elementi richiesti nell'header    */
11       SOAPHeader soapHeader = soapMessage.getSOAPHeader();
12       SOAPBody soapBody = soapMessage.getSOAPBody();
13       if (soapBody == null){
14         l.error("handleInbound: Il messaggio non contiene un Body");
15         return false;
16       }
17       if((soapHeader == null) || !findInfoHeader(soapHeader)){
18         l.error("handleInbound: Il messaggio non contiene gli elementi Header
                richiesti");
19         return false;
20       }
21
22       /*  recupero di tutte le informazioni richieste
23        * per il comporre l'evento da monitorare   */
24       String session=req.getSession().getId();
25       long sequence = Long.valueOf(node_name.get(NODE_NAME[0]));
26       long timestamp = System.currentTimeMillis();
27       String username=node_name.get(NODE_NAME[1]);
28       String password = node_name.get(NODE_NAME[2]);
29       String applURL=
30                "http://"              +
31                req.getLocalName()+":"  +
32                req.getLocalPort()      +
33                req.getContextPath()    ;
34       QName op = (QName)msg.get(msg.WSDL_OPERATION);
35       String NSBody= op.getNamespaceURI();
36       String operation=op.getLocalPart();
37       byte[] parB=findParameter(soapBody,op.getLocalPart());
```

```
38
39        /*  trasformazione delle stringhe in byte array */
40        byte[] sessionB=session.getBytes();
41        byte[] usernameB=username.getBytes();
42        byte[] passwordB=password.getBytes();
43        byte[] applURLB=applURL.getBytes();
44        byte[] NSBodyB=NSBody.getBytes();
45        byte[] operationB=operation.getBytes();
46
47        /*  calcolo della dimensione del content, necessaria
48         *  per la corretta scomposizione del pacchetto     */
49        int contentSize =
50              CONTENT_FIX_SIZE    +
51              usernameB.length    +
52              passwordB.length    +
53              applURLB.length     +
54              NSBodyB.length      +
55              operationB.length   ;
56        if(parB!=null)
57          contentSize=contentSize+parB.length;
58
59        ByteBuffer bb = ByteBuffer.allocate(KEY_SIZE+contentSize);
60        /*  creazione della chiave (49 byte)  */
61        bb.put(INB);
62        bb.put(sessionB);
63        bb.putLong(sequence);
64        bb.putLong(timestamp);
65
66        /*  creazione del content */
67        bb.putInt(contentSize);
68        bb.putInt(usernameB.length);
69        bb.put(usernameB);
70        bb.putInt(passwordB.length);
71        bb.put(passwordB);
72        bb.putInt(applURLB.length);
73        bb.put(applURLB);
74        bb.putInt(NSBodyB.length);
75        bb.put(NSBodyB);
76        bb.putInt(operationB.length);
77        bb.put(operationB);
78        if(parB!=null)
79          bb.put(parB);
80        byte[] sendData = new byte[UDP_PACK_SIZE];
81        sendData = bb.array();
82
83        /*  inserimento nell'header HttpServletResponse della chiave
84         *  del pacchetto Inbound in modo da recuperare tutte le
85         *  informazioni necessarie per il monitoring durante la
86         *  gestione del relativo messaggio SOAP Outbound       */
87        String control=INB+session+Long.toString(sequence)+Long.toString(timestamp);
88        HttpServletResponse
              hsr=(HttpServletResponse)msg.get(MessageContext.SERVLET_RESPONSE);
89        hsr.addHeader(FILTERcontrol, control);
90        byteInbound.put(control, sendData);
91        byte_session.put(control, sessionB);
92
93        /*  invio del pacchetto al componente di monitoring
94         *  mediante socket UDP                 */
95        DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
              IPAddress, UDP_PORT);
96        clientSocket = new DatagramSocket();
97        clientSocket.send(sendPacket);
98
99      } catch (SOAPException e) {
```

```
100         l.error("SOAPException "+e.getMessage());
101         return false;
102       }catch(UnsupportedOperationException e){
103         l.error("handleInbound: UnsupportedOperationException "+e.getMessage());
104         return false;
105       }catch(ClassCastException e){
106         l.error("handleInbound: ClassCastException "+e.getMessage());
107         return false;
108       }catch(NullPointerException e){
109         l.error("handleInbound: NullPointerException "+e.getMessage());
110         return false;
111       }catch(IllegalArgumentException e){
112         l.error("handleInbound: IllegalArgumentException "+e.getMessage());
113         return false;
114       }catch(NumberFormatException e){
115         l.error("handleInbound: SequenceNumber "+e.getMessage());
116         return false;
117       }catch (IOException e) {
118         if(clientSocket!=null)
119           if(!clientSocket.isClosed())
120             clientSocket.close();
121         l.error("handleInbound: IOException "+e.getMessage());
122         return false;
123       }
124       return true;
125     }
```

4.2.2 Il servizio di monitoring SLA

Il componente che risiede al livello middleware rappresenta un servizio di JBoss che viene realizzato attraverso l'implementazione di Plain Old Java Object (POJO): classi Java con alcune annotazioni che ne denotano la natura particolare del componente.

Come gli MBean standard, i servizi POJO necessitano di definire il nome dell'oggetto e l'interfaccia corrispondente. Per il nostro servizio di monitoring abbiamo utilizzato le seguenti annotazioni:

```
@Service(objectName = "servicePOJO:service=MonitoringService")
@Management(MonitoringServiceInterface.class)
@Depends ("jboss.jca:name=MySqlDS,service=DataSourceBinding")
```

In questo modo viene definito:

- il nome del servizio: utilizzando il modello *dominio:proprietà=valore*;
- l'interfaccia `MonitoringServiceInterface`: che definisce gli attributi e le operazioni del POJO;

- la dipendenza al servizio `DataSourceBinding`: che attende la disponibilità del servizio che realizza il datasource.

I metodi del ciclo di vita del servizio che abbiamo implementato sono:

`create()` chiamato dal server per la creazione (che avviene successivamente ai servizi dai quali dipende);

`destroy()` chiamato dal server per la rimozione.

L'immagine in figura 4.1 mostra la struttura del progetto in package Java organizzati in modo da essere intuitivamente comprensibili e facilitare un'eventuale espansione del sistema.

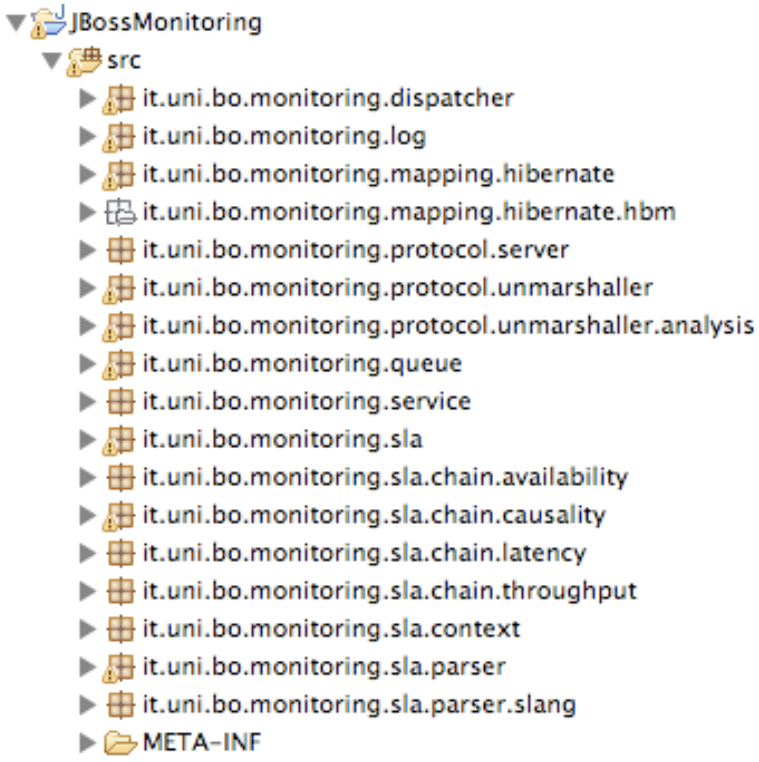

Figura 4.1: Servizio di monitoring SLA: organizzazione dei package

La tabella 4.2 mostra il numero di classi realizzate con il totale di righe di codice prodotte all'interno di ogni package.

Nome package (`it.uni.bo.monitoring.*`)	Numero di classi	Righe di codice
`dispatcher`	1	52
`log`	1	96
`mapping.hibernate`	19	1511
`protocol.server`	3	372
`protocol.unmarshaller`	3	381
`protocol.unmarshaller.analysis`	6	654
`queue`	2	196
`service`	3	170
`sla`	1	123
`sla.chain.availability`	3	175
`sla.chain.causality`	3	109
`sla.chain.latency`	3	126
`sla.chain.throughput`	3	174
`sla.context`	5	301
`sla.parser`	1	45
`sla.parser.slang`	6	442
Totale	63	4927

Tabella 4.2: Servizio di monitoring SLA: contenuto dei package.

Il package `service` contiene l'interfaccia `MonitoringServiceInterface` descritta in precedenza e la relativa implementazione che realizza il servizio POJO per JBoss. All'interno dello stesso pacchetto, abbiamo realizzato la classe `StaticFields` per dichiarare tutte le costanti statiche utilizzate negli altri package nonché tutte le interrogazioni SLQ che vengono compiute durante l'analisi dei dati dall'elemento Protocol Unmarshaller.

Il package `mapping.hibernate` contiene tutte le classi Java necessarie per la gestione completa della banca dati. La libreria Hibernate, ci ha permesso di realizzare la mappatura dell'intera banca dati con oggetti Java in modo da mantenere unÕomogeneità nella gestione gestione delle risorse.

All'interno del servizio di monitoring i messaggi ricevuti dalla sonda degli eventi vengono trasportati da un componente all'altro (realizzato come un thread per permettere una comunicazione asincrona) attraverso l'utilizzo di

code bloccanti. Un coda bloccante provoca il blocco di uno dei componenti quando si rimuove un elemento e la coda è vuota.

Le code bloccanti sono state realizzate attraverso l'oggetto `LinkedBlockingQueue` che offre i seguenti metodi bloccanti:

`put(E elem)` permette di inserire l'oggetto elem in cima alla coda;

`take()` restituisce l'elemento in cima alla coda. Se la coda è vuota, questo metodo mette in attesa in thread corrente, finché non viene inserito almeno un elemento nella coda.

Di seguito descriviamo il contenuto di ogni package per i due componenti che realizzano il sistema di monitoring (il gestore della coda di eventi e il componente di monitoring del SLA) evidenziando gli aspetti implementativi più importanti.

Il gestore della coda di eventi

All'interno del package `protocol.server` viene realizzato l'elemento Event Protocol Manager che implementa il lato server del protocollo descritto nella sezione 3.6. In particolare abbiamo realizzato un thread che utilizzando il socket della libreria `java.net` è in grado ricevere i messaggi UDP prodotti dalla sonda degli eventi. L'indirizzo e la porta associati al socket possono essere configurati opportunamente mediante il file `monitoring.properties` descritto in sezione 4.3 in modo da integrare il servizio di monitoring nel proprio ambiente di sviluppo. Nello stesso package abbiamo realizzato un primo analizzatore di validità dei pacchetti in entrata che ne verifica la dimensione e fornendo una prima scomposizione del pacchetto.

All'interno del package `queue` viene realizzato l'elemento Queue Persistence Manager attraverso le classi `LazyInsertRawData` e `LazyRemoveRawData`. `LazyInsertRawData` si occupa di inserire gli eventi ricevuti dal Event Protocol Manager nella banca dati e successivamente nella coda dei messaggi, chiamata `rawDataQueue`, condivisa con il Dispatcher. La classe `LazyRemoveRawData`, viceversa, si occupa di rimuovere dalla banca dati tutti gli eventi

che sono stati analizzati dal Protocol Unmarshaller. Al fine di implementare le funzionalità di tolleranza ai guasti in modo ridurre al minimo la probabilità di perdita di eventi a fronte di un guasto di tipo crash dell'application server, la classe `LazyInsertRawData` realizza una struttura a coda con un backup persistente che viene precaricata ad ogni riavvio del servizio. Il metodo `loadBackup`, illustrato in 4.2 mostra il caricamento nella coda `rawDataQueue` di tutti gli eventi memorizzati nella banca dati ma non ancora analizzati.

Listing 4.2: Backup della coda di eventi

```
Session session = factory.openSession();
Iterator<RawData> a = session.createQuery(QUERY_ALL_RAWDATA).iterate();
while(a.hasNext()){
  RawData rd = a.next();
  rawDataQueue.add(new DataObject(rd.getIdKey(),rd.getContent()));
}
```

All'interno del package `dispatcher` viene realizzato l'elemento Event Dispatcher Thread, descritto in sezione 3.7, che si occupa di inoltrare gli eventi ricevuti dal Queue Persistence Manager sia al gestore dei log che al componente di monitoring.

Il package `log` contiene il gestore dei log; un thread che in maniera asincrona si occupa di memorizzare in persistenza i messaggi intercettati dalla sonda in modo da consentirne una lettura immediata in caso di anomalie del sistema.

Infine, nel package `protocol.unmarshaller` viene realizzato l'elemento Protocol Unmarshaller: responsabile dell'interpretazione dei messaggi di protocollo ricevuti e della creazione delle relative istanze del tipo Evento. In particolare all'interno del package `protocol.unmarshaller.analysis`, la classe `DataAnalysis` si avvale di tre diversi analizzatori, definiti attraverso l'interfaccia `Analyzers`, che attraverso il metodo `validate` effettuano rispettivamente l'analisi dei tipi, l'imponibilità dei requisiti di SLA e il controllo dei vincoli di integrità referenziale dell'evento ricevuto dal Dispatcher. Nel frammento di codice in 4.3 mostriamo il metodo `analyse` della classe `DataAnalysis` che consente di produrre un'istanza di oggetto Evento e re-

stituirla al componente di SLA Monitoring solo se tutte le analisi compiute dai tre analizzatori sono andate a buon fine

Listing 4.3: Analisi degli eventi

```
public Evento analyse(DataObject dataObj) {
    Evento evento=new Evento(dataObj);

    for(int i=0;i<analyzers.size();i++){
      if(!analyzers.get(i).validate(evento)){
        return null;
      }
    }
    return evento;
  }
```

Il componente di monitoring del SLA

All'interno del package `sla` viene realizzato il SLA Manager, l'elemento che si occupa di richiamare i servizi di monitoring del Monitor Chain. Abbiamo realizzato questo elemento attraverso un thread, chiamato `SLAManager`, che rimane in attesa di ricezione di nuovi eventi inviati dal gestore della coda. Nel frammento di codice in 4.4 mostriamo la ricezione dell'evento inserito nella coda bloccante `eventQueue` condivisa con il gestore della coda degli eventi, il controllo dell'esistenza dell'istanza di SLA associata all'evento ricevuto e, infine, il richiamo dei servizi del Monitor Chain con il passaggio dell'evento da monitorare e la relativa istanza di SLA associata.

Listing 4.4: Monitoring dell'evento

```
Evento evento= eventQueue.take();
if(evento!=null){
  long idSla=evento.getIdSla();
  if((idSla>=0) && (contextMap.containsKey(idSla)))
    monitoringChain.processEvent(ei, contextMap.get(idSla));
}
```

Il package `sla.context` contiene la classe che realizza il Monitoring-Context: l'elemento responsabile di mantenere le informazioni sullo stato

di esecuzione di un'istanza di SLA. Al momento della creazione, questo elemento realizza tutti gli oggetti e le strutture necessari ai vari componenti di monitoring definiti attraverso l'interfaccia `Component`. Il metodo `makeTrans(Evento e)` consente, all'interno dello stesso componente di monitoring, il passaggio da uno stato di esecuzione sui controlli del SLA all'altro. Il frammento di codice in 4.5, mostra il metodo `createAutomatonThroughput` implementato all'interno della classe `MonitoringContext` che crea una lista di `Component` di throughput realizzati come automi temporizzati a stati finiti come descritto in 3.8.2.

Listing 4.5: Monitoring Context: creazione del componente di throughput

```
1  private List<Component> createAutomatonThroughput(List<Throughput> tl,
       SessionFactory factory){
2
3    List<Component> raitList = new ArrayList<Component>();
4      for(int i=0;i<tl.size();i++){
5
6        int window = tl.get(i).getWindow().intValue();
7        int numStates = (int) tl.get(i).getConcurrency() + 2;
8
9        AutomataState Init= new AutomataState("Init");
10       AutomataState l1= new AutomataState("l1");
11       AutomataState Final= new AutomataState("l"+(numStates-1));
12
13       Init.addTransition(new AutomataTransition(l1));
14       for (int j=1; j<numStates-2; j++) {
15         AutomataState as = l1;
16         l1= new AutomataState("l"+(j+1));
17         as.addTransition(new AutomataTransition(l1));
18         if(j==numStates-3)
19           l1.addTransition(new AutomataTransition(Final, window));
20       }
21       raitList.add(new ThroughputManager(numStates, Init, Final,
             factory));
22     }
23     return raitList;
24   }
25 }
```

Il package `sla.chain` contiene tutte le classi che realizzano la logica del monitoring SLA, in paricolare abilita l'esecuzione dei controlli sul buon uso

del servizio, sul throughput, sulla latenza, e sulla disponibilità. I diversi componenti responsabili del monitoring dei requisiti di SLA sono realizzati utilizzando il pattern Chain of Responsability, che permette di separare gli oggetti che invocano richieste dagli oggetti che le gestiscono, in modo da dare ad ognuno la possibilità di gestire queste richieste. Viene utilizzato il termine catena perché di fatto la richiesta viene inviata e segue la catena di oggetti, passando da uno all'altro, finché non trova quello che la gestisce. Attraverso la classe astratta `MonitorChain` vengono esposti i metodi:

`setNext(MonitorChain mc)` per mantenere il riferimento al componente di monitoring successivo;

`processEvent(Evento e, MonitoringContext mc)` utilizzato per gestire la richiesta di monitoring dell'evento in base al contesto di uno specifico SLA.

Ogni componente che estende questa classe procede all'analisi dell'evento e successivamente all'invocazione del componente successivo. Come descritto in 3.8 tutti i componenti di monitoring vengono eseguiti nel solo caso in cui il Causality Monitor (il primo della catena) non abbia rilevato alcuna violazione al buon uso del servizio.

Il package `parser` contiene le classi necessarie allo sviluppo del componente di lettura del SLA realizzato in base ad uno specifico linguaggio. Per consentire la realizzazione dei componenti per l'interpretazione di altri linguaggi oltre a SLAng, utilizzato all'interno del nostro sistema, abbiamo realizzato la classe astratta `SlaBean` che attraverso il metodo `addSlaLanguage` consente di aggiungere i diversi interpreti.

Il package `parser.slang` contiene il nostro interprete `SlangBean`, che estende la classe astratta `SlaBean`, per il linguaggio SLAng. In fase di avvio del servizio, il nostro interprete si occupa di:

- recuperare tutti i documenti SLA memorizzati in banca dati;
- analizzarne la validità;

- estrarre tutte le informazioni relative ai parametri da monitorare;
- creare appositi oggetti contenenti tutte le informazioni estratte.

4.3 Guida al corretto utilizzo del servizio

Per garantire la portabilità del progetto e permettere l'automatizzazione del processo di sviluppo del servizio realizzato, abbiamo utilizzato la libreria JAVA Ant versione 1.8 che attraverso due file `build.xml` consente di generare i relativi file jar da inserire correttamente all'interno dell'application server JBoss.

Il jar realizzato nel progetto `JBossWSHandler` che incapsula la sonda degli eventi dovrà essere incluso nel build path dell'applicazione che realizza il web service da monitorare. Il web service dovrà avere la seguente annotazione J2EE:

```
import javax.jws.HandlerChain;
@HandlerChain(file = ''META-INF/jaxws-handlers-server.xml")
```

che consente di indicare all'interno del file jaxws-handlers-server.xml il riferimento alla classe che realizza la nostra sonda attraverso il seguente frammento xml:

```
<handler>
        <handler-name>FilterHandler</handler-name>
        <handler-class>handler.webService.jbossws.FilterHandler
            </handler-class>
</handler>
```

Il jar realizzato nel progetto `JBossMonitoring` che incapsula il servizio che realizza la coda degli eventi, l'analisi e il monitoring di SLA dovrà essere inserito nella cartella di deploy di JBoss non in fase di esecuzione.

Per la configurazione del datasource è necessario creare un file di configurazione per MySQL chiamato `mysql-ds.xml`, in cui vengono dichiarati il nome del servizio associato al database, l'url per accedervi con il relativo nome utente e password, e copiarlo nella stessa directory di deploy di JBoss.

Per la comunicazione con il componente sonda residente a livello applicativo e con il database abbiamo fornito un file chiamato `monitoring.properties` da inserire in `$JBOSSHOME/conf` che attraverso i campi `eventQueue.input.inetAddress`, `eventQueue.input.udpPort` e `eventQueue.input.datagramSize` consente di configurare rispettivamente l'indirizzo IP e la porta associata al socket nonché la dimensione massima del messaggio di protocollo di trasmissione degli eventi che abbiamo progettato e realizzato. Attraverso il campo `connection.datasource` viene infine fornito il nome del datasource indicato nel file `mysql-ds.xml` descritto in precedenza.

Capitolo 5

Test e Perfomance

In questo capitolo presentiamo il lavoro che è stato svolto per accertare il overhead imposto dall'uso del servizio che abbiamo sviluppato.

5.1 Concezione e preparazione dei test

Lo scopo dei test che ci siamo proposti di effettuare è la misurazione del overhead imposto dall'uso del nostro servizio.

A tal fine, abbiamo concepito una finta applicazione che emulasse il comportamento di un'applicazione reale; per questo scopo abbiamo sviluppato il servizio web che nella sezione 3.2.1 abbiamo portato ad esempio per la descrizione del formalismo adottato per la modellazione dei requisiti di buon uso. Per comodità ricordiamo le operazioni implementate dalla nostra finta applicazione:

list: servizio per la ricerca di voli;

getDetails: servizio per la visione dei dettagli del volo;

reserve: servizio per riservare temporaneamente un posto sul volo;

confirm: servizio per confermare l'acquisto del volo;

printTicket: servizio per ottenere i dettagli del biglietto acquistato.

Abbiamo dato all'interfaccia un'implementazione fasulla in modo che ciascun metodo avesse un tempo di esecuzione realistico rispetto alle performance di un web service di questo tipo. Abbiamo fornito ogni operazione di un'implementazione fasulla, in quanto lo sviluppo di un'applicazione di prenotazione voli sarebbe fuori dalle competenze di questo lavoro di tesi. In particolare, l'implementazione di ogni operazione impone la sospensione del thread di esecuzione per un periodo di tempo realisticamente richiesto da un'implementazione reale dell'operazione. Il tempo di servizio di ogni operazione viene deciso attraverso un generatore di numeri pseudocasuale con distribuzione di Poisson: una distribuzione di probabilità discreta che descrive la modalità in cui gli eventi, fra loro indipendenti e identicamente distribuiti, si verificano successivamente sapendo che mediamente se ne verifica un numero λ. Ogni operazione del web service è quindi associata ad un proprio generatore di numeri pseudocasuali generati secondo la distribuzione di Poisson; la tabella 5.1 dichiara i tempi medi di esecuzione assegnati per ciascuno dei metodi sopra menzionati. Il thread chiamante l'operazione viene sospeso per un numero di millisecondi deciso dal generatore associato all'operazione e il controllo viene restituito al chiamante al termine della sospensione del thread.

Operazione	λ (ms)
list	140
getDetails	25
reserve	200
confirm	200
printTicket	25

Tabella 5.1: Tempo medio di risposta in millisecondi di ogni operazione del web service realizzato.

Si noti che questo tipo di implementazione dei metodi è modesto nell'esigenza di risorse; tuttavia è importante ricordare che l'enfasi di questo

lavoro di test deve porsi sull'analisi del carico di lavoro imposto dal servizio di monitoring di SLA e non tanto dall'aumento delle risorse richieste dell'applicazione, all'aumentare del numero di richieste.

Sempre al fine di emulare un contesto d'uso realistico, abbiamo sviluppato un emulatore configurabile di user agent che potesse fare richieste all'applicazione emulando un numero arbitrario di utenti che agiscono simultaneamente. Per fare questo, abbiamo osservato che il comportamento di uno user agent può essere rappresentato attraverso un automa a stati finiti probabilistico temporizzato avente le seguenti caratteristiche:

- uno stato per ogni azione che il user agent può compiere;
- per ogni stato, un arco uscente associato alla probabilità che l'utente scelga quella direzione e al tempo necessario al user agent per prendere quella scelta;
- la somma delle probabilità degli archi uscenti da uno stato non finale dell'automa è 1;
- dato uno stato qualunque non finale, la distribuzione con cui uno user agent decide la direzione da prendere è uniforme;
- dato un arco uscente da uno stato, la distribuione con cui viene deciso il tempo che uno user agent impiega a percorrere quella transizione è una distribuzione di Poisson.

Abbiamo quindi realizzato un framework programmabile capace di interpretare descrizioni xml di tale tipologie di automa ed emulare, per un periodo di tempo ben definito, l'esecuzione di un numero arbitrario di user agent che interagiscono col servizio programmato seguendo il pattern probabilistico di interazione definito dalla descrizione dall'automa interpretato. In appendice C definiamo il linguaggio che abbiamo sviluppato per la descrizione di user agent e che viene interpretato dal framework che ci ha permesso di sviluppare l'applicazione client utilizzata per lo svolgimento dei test.

Questa applicazione, emula un user agent avente il comportamento descritto dall'automa in figura 5.1 e dalle tabelle 5.2 e 5.3. Affinché il test sia verosimile, ogni user agent emulato comunica con l'application server impiegando per tutte le comunicazioni con il web service una stessa sessione http, diversa da quelle impiegate dagli altri user agent emulati dalla stessa applicazione.

In figura 5.1 è rappresentato l'automa che descrive il comportamento del user agent. Ogni stato dell'automa è etichettato in corrispondenza dell'operazione che viene eseguita quando il user agent l'attraversa mentre ogni arco è etichettato con due numeri che rappresentano, rispettivamente, la probabilità che il user agent ha di seguire quell'arco e il tempo che esso impiega ad effettuare la transizione di stato.

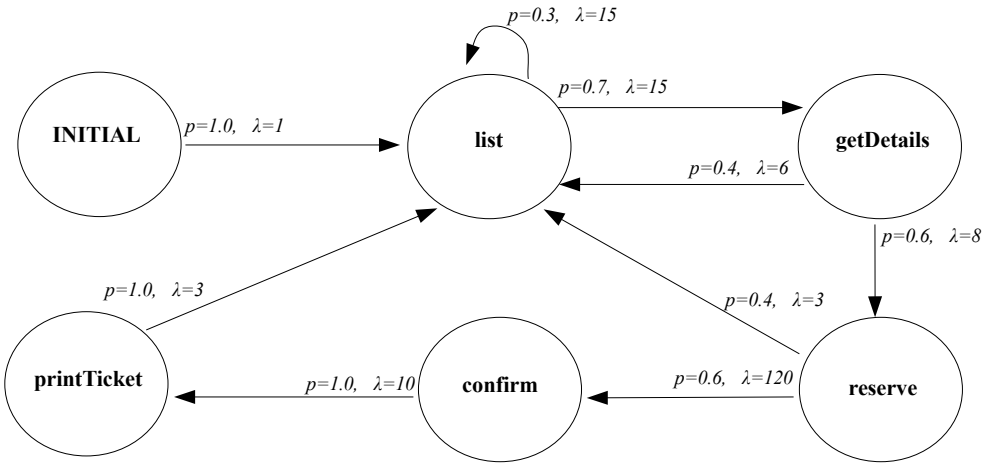

Figura 5.1: Rappresentazione grafica dell'automa che modella il comportamento del user agent.

Le due tabelle 5.2 e 5.3 offrono una rappresentazione alternativa dell'automa; la prima evidenzia le probabilità da un nodo di transitare verso gli altri, mentre la seconda evidenzia il tempo medio di transito da uno nodo verso un altro. Le tabelle si leggono per riga. L'intestazione di riga esprime lo stato di partenza mentre le intestazioni di colonna esprimono lo stato di

arrivo e la cella di intersezione contiene il valore specifico evidenziato dalla tabella.

	list	**getDetails**	**reserve**	**confirm**	**printTicket**
list	0.3	0.7	0	0	0
getDetails	0.4	0	0.6	0	0
reserve	0.4	0	0	0.6	0
confirm	0	0	0	0	1
printTicket	1	0	0	0	0

Tabella 5.2: Emulazione del user agent: probabilità di transizione tra stati.

	list	**getDetails**	**reserve**	**confirm**	**printTicket**
list	15	15			
getDetails	6		8		
reserve	3			120	
confirm					10
printTicket	3				

Tabella 5.3: Emulazione del user agent: tempo medio di transizione tra stati, in secondi.

Per svolgere i test, abbiamo quindi misurato le performance di esecuzione del web service descritto sopra in assenza e in presenza del servizio di monitoring, al variare del numero di user agent emulati. In particolare, ciascun test ha avuto la durata di due ore ed è stato ripetuto, con e senza servizio di monitoring, per un numero di user agent emulati pari a 10, 20, 40, 80, 160 e 320. Le performance ottenute in questo modo sono state confrontate al fine di isolare il overhead introdotto dal nostro prototipo. I risultati dei test sono presentati nella sezione 5.3.

5.2 L'ambiente di test

I test sono stati realizzati su un Fujitsu LIFEBOOK A530 con le seguenti caratteristiche:

Processore	Intel® Core™ i5-560M (2.66 GHz, 3 MB)
RAM	4 GB DDR3, 1066 MHz, PC3-8500, DIMM
Hard disk	SATA, 5400 rpm, 500 GB
Sistema Operativo	Genuine Windows® 7 Professional

Tabella 5.4: Ambiente di test: caratteristiche hardware.

I test sono stati realizzati eseguendo sulla macchina i seguenti software:

JVM (JBoss)	–Xms1152M –Xmx1152M –XX:PermSize=512M
JVM (User agent)	–Xms512M –Xmx512M
MySQL	512 MB

Tabella 5.5: Ambiente di test: caratteristiche software.

5.3 I risultati

In questa sezione presentiamo il risultato dei test che abbiamo svolto seguendo l'approccio definito nella sezione 5.1. Come anticipato precedentemente, abbiamo utilizzato il client, sviluppato attraverso il framework prodotto per l'occasione, per generare le invocazioni al web service della prenotazione di voli, in presenza e in assenza del servizio di monitoring.

La prima serie di test è stata prodotta senza il servizio di monitoring e ha visto il web service sollecitato con sei test della durata di due ore, emulando rispettivamente 10, 20, 40, 80, 160, 320 user agent. La seconda serie di test è identica alla prima, fatta eccezione per la presenza del servizio di monitoring nella configurazione di test. Durante ognuno dei test, gli user agent

emulati hanno continuamente e simultaneamente esibito il comportamento programmato attraverso l'automa descritto precedentemente; si noti infatti che l'automa implementato dagli user agent presenta diversi cicli e non ha uno stato finale. La tabella 5.6 mostra il numero di richieste generate dal client (Reqs) che abbiamo realizzato per ciascun test, con e senza SLA monitoring, e la corrispondente frequenza media di richieste per secondo (RPS) pervenute al server.

	Reqs w SLA	**RPS w SLA**	**Reqs w/o SLA**	**RPS w/o SLA**
10 u.a.	3557	0.494	3555	0.494
20 u.a.	7365	1.023	7298	1.014
40 u.a.	14436	2.005	14453	2.007
80 u.a.	29004	4.014	29041	4.033
160 u.a.	57683	8.012	57650	8.007
320 u.a.	115283	16.011	114962	15.967

Tabella 5.6: Richieste generate dagli esperimenti, con e senza monitoring, e relative frequenze medie di richieste al secondo.

Alla fine di ciascun test, abbiamo calcolato il tempo medio di servizio percepito dal cliente per ciascuno dei test e, componendo questi risultati, abbiamo ottenuto il grafico in figura 5.2. In questa figura possiamo notare che all'aumentare delle istanze di user agent, rappresentate graficamente dalla linea azzurra (misurabile attraverso l'asse sulla destra), il overhead introdotto dal sistema di monitoring rimane costante e uguale a 4 millisecondi. In particolare, l'andamento del tempo di servizio senza monitoring del SLA è descritto dalla linea rossa e mantiene il valor medio sui 144 millisecondi per tutti i test, mentre l'andamento del tempo di servizio con il monitoring del SLA è descritto dalla linea verde che mantiene il valor medio sui 148 millisecondi per tutti i test; entrambe queste linee sono misurabili attraverso l'asse sulla destra. Questo dato dimostra l'efficacia di due degli accorgimenti che

abbiamo adottato nella progettazione di questo servizio di monitoring. Il pri-

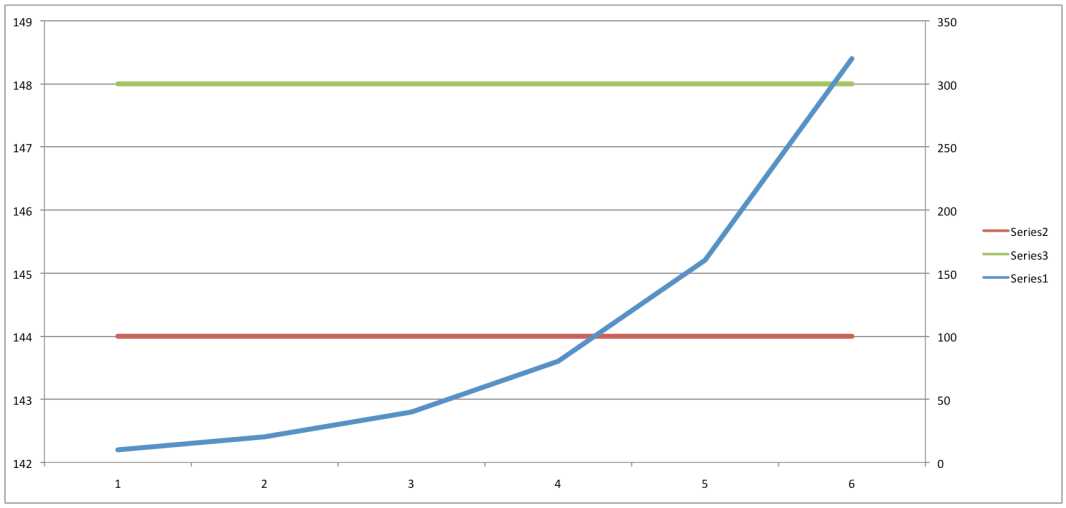

Figura 5.2: Grafico del overhead imposto dal servizio di monitoring al variare del carico di richieste

mo accorgimento riguarda l'efficienza dell'implementazione del handler che, come descritto nel capitolo precedente, si dimostra col basso overhead imposto dall'estrazione dei dati dell'evento dal messaggio SOAP e dallo stream HTTP sottostante. Il secondo riguarda invece il protocollo basato su UDP che abbiamo progettato e implementato per trasmettere le informazioni sugli eventi rilevati al servizio di monitoring. La combinazione di queste due tecniche fa in modo che:

- Il handler riesca a comunicare gli eventi rilevati al servizio di monitoring senza dipendere dai tempi di reazione dell'interfaccia del servizio;
- L'asincronia intrinseca alla progettazione del servizio e resa possibile dal disaccoppiamento della sonda di eventi dal servizio di monitoring attraverso lo studio del gestore della coda degli eventi.

La tabella 5.7 mostra il numero di eventi isolati dalla sonda durante i test effettuati, quelli gestiti durante il tempo del test e quelli accodati e gestiti oltre il termine del test.

	Ev. Totali	**Ev. Analizzati**	**Ev. Rimanenti**	**Tempo**
10 u.a.	7114	7114	0	0
20 u.a.	14730	14730	0	0
40 u.a.	28872	28872	0	0
80 u.a.	58008	58008	0	0
160 u.a.	115366	62199	53167	$2h < t < 3h$
320 u.a.	230566	62637	167929	$8h < t < 9h$

Tabella 5.7: Eventi prodotti dagli esperimenti, eventi analizzati durante il test, eventi analizzati dopo il termine del test e tempo impiegato per l'analisi.

La tabella 5.8 mostra il numero delle violazioni generate durante i test nonché le rilevazioni sui tempi minimi e massimi che sono trascorsi tra il censimento di un evento e la notifica della violazione associata. Da questa tabella si può notare che la notifica delle violazioni si è mantenuta in tempo reale fino all'esperimento eseguito con 80 istanze di user agent. Già a partire da 160 user agent, quando la frequenza di generazione degli eventi ha superato la capacità di analisi del SLA monitor, la garanzia di notifica in tempo reale si è persa.

	Violazioni	t_min **(ms)**	t_max **(ms)**
10 u.a.	2	507 ms	521 ms
20 u.a.	4	513 ms	528 ms
40 u.a.	14	531 ms	2774 ms
80 u.a.	127	513 ms	3562 ms
160 u.a.	502	506 ms	$> 2\ h$
320 u.a.	59608	539 ms	$> 8\ h$

Tabella 5.8: Violazioni del SLA individuate durante i test e tempi minimi e massimi impiegati per l'individuazione e notifica.

La tabella 5.9 mostra il numero delle violazioni generate durante i test nonché la classificazione di tali violazioni in base al tipo.

Per lo scopo di misurare il overhead, non abbiamo considerato il buon uso perché ogni violazione al buon uso esclude tutti i controlli fatti a valle e quindi la richiesta di un numero inferiore di risorse, in termini di computazione e di memoria, riducendo il tempo di analisi di un evento prodotto.

Per tutte le operazioni realizzate, abbiamo fissato la latenza indicata nel nostro SLA a 330 ms, la tabella mostra un incremento delle violazioni di latenza all'aumentare del numero di user agent e quindi delle richieste.

La tabella mostra, inoltre, come nei primi test non vi siano violazioni sul throughput in quanto il rate di richieste mediamente generato dagli user agents non raggiunge la soglia di throughput contrattata dal cliente (fissata, per lo scopo dei test, a 80 richieste per cinque secondi). A partire dal test con 160 user agent, si può notare che iniziano a comparire violazioni al throughput, generate nei momenti di addensamento della generazione delle richieste da parte delle istanze di user agent emulate.

	Violazioni	**Latenza**	**Throughput**	**Disponibilità**	**Buon uso**
10 u.a.	2	2	0	0	0
20 u.a.	4	4	0	0	0
40 u.a.	14	14	0	0	0
80 u.a.	127	127	0	0	0
160 u.a.	502	219	283	0	0
320 u.a.	59608	774	58834	0	0

Tabella 5.9: Classificazione delle violazioni individuate dal monitor SLA durante i test effettuati.

Dall'analisi dei dati delle tabelle 5.7 e 5.8 è possibile fare le seguenti deduzioni:

- il prototipo sviluppato consente di elaborare in tempo reale circa 62000 eventi in due ore, corrispondenti ad una frequenza di richieste al servizio pari a 4.305 richieste al secondo;
- il sistema è robusto allo stress; esso si è dimostrato capace di sopportare per due ore una frequenza di richieste al secondo di quattro volte superiore alla frequenza che garantisce l'elaborazione degli eventi in tempo reale.

In conclusione, dall'analisi dei dati ottenuti attraverso i test risulta che:

- il prototipo sviluppato soddisfa i requisiti che ci siamo posti come obiettivo di questa tesi;
- il comportamento prestazionale del servizio che abbiamo sviluppato scala all'aumentare del numero di utenti dei servizi;
- il servizio che abbiamo realizzato soddisfa il requisito di robustezza quando sollecitato per lungo tempo da un numero di richieste superiore alla sua capacità di analisi;
- il overhead introdotto dal servizio di monitoring è contenuto anche quando sollecitato per lungo tempo da un numero di richieste superiore alla sua capacità di analisi.

Capitolo 6

Conclusioni

Con questa tesi abbiamo realizzato una delle prime implementazioni di un servizio di monitoring di service level agreements (SLA) integrata in JBoss, un'implementazione del framework J2EE disponibile gratuitamente e largamente utilizzato in contesti industriali. In particolare, il servizio che abbiamo realizzato è capace di interpretare istanze di service level agreements opportunamente formalizzate e, limitatamente all'erogazione di servizi in modalità business-to-business, di monitorarne l'esecuzione coerentemente con i vincoli di performance e disponibilità definiti a livello applicativo. Attualmente, il nostro servizio è in grado di individuare violazioni rispetto alla definizione di vincoli di performance relativi a throughput, tempo di risposta, e disponibilità.

Con questa tesi ci siamo anche posti l'obiettivo di dare un contributo allo stato dell'arte relativamente all'introduzione del concetto di buon uso di un servizio e della relazione che questo concetto può avere con l'imposizione di vincoli di performance specificati da service level agreements. Il nostro contributo si basa sulla considerazione che un cliente che utilizza le operazioni messe a disposizione da un servizio secondo una semantica diversa da quella del servizio stesso, non debba poter avanzare pretese sulle performance sperimentate.

Coerentemente con tale considerazione, con questa tesi abbiamo pro-

posto una prima formalizzazione del concetto di buon uso, definendone la relazione con le operazioni di monitoring di SLA, e fornendo una prima implementazione di tali controlli nel servizio che abbiamo presentato.

Riassumendo, con questo lavoro di tesi abbiamo raggiunto il nostro obiettivo iniziale, realizzando un servizio interno a JBoss che:

- Implementa logiche che consentono di monitorare il buon uso di un web service;
- Implementa una logica di controllo tale da individuare violazioni sui requisiti di disponibilità, tempo di servizio, e throughput, compatibilmente con i vincoli imposti dalla formalizzazione del buon uso di un servizio;
- Consente l'esecuzione contemporanea di molteplici istanze di SLA per monitorare uno o più servizi distribuiti su uno o più contesti applicativi;
- Può essere facilmente applicato per monitorare l'esecuzione di qualunque applicazione senza richiedere interventi sul codice applicativo;
- È robusto e tollerante ai guasti;
- Mantiene una banca dati contenente le informazioni relative alle invocazioni effettuate dai clienti ai servizi in modo da rendere riproducibili i controlli effettuati;
- Impone un overhead trascurabile sulle performance dell'application server.

Questo servizio è stato oggetto di test intensivi mirati ad accertarne l'applicabilità in contesti reali, oltre che a verificare l'efficacia delle nostre scelte progettuali e a verificare il soddisfacimento dei requisiti iniziali. I risultati dei test svolti e presentati in questo documento ci permettono di affermare che:

- Il prototipo sviluppato soddisfa i requisiti che ci siamo posti come obiettivo di questa tesi;

- Il comportamento prestazionale del servizio che abbiamo sviluppato scala all'aumentare del numero di utenti dei servizi;
- Il servizio che abbiamo realizzato soddisfa il requisito di robustezza quando sollecitato per lungo tempo da un numero di richieste superiore alla sua capacità di analisi;
- Il overhead introdotto dal servizio di monitoring si mantiene contenuto anche quando sollecitato per lungo tempo da un numero di richieste superiore alla sua capacità di analisi.

Durante l'esecuzione della fase di test abbiamo individuato e corretto alcuni aspetti prestazionali legati al servizio implementato; tuttavia l'opera di ottimizzazione può essere estesa per raggiungere livelli prestazionali e di robustezza superiori a quelli forniti dall'attuale implementazione. Porteremo a termine questa attività nel contesto di future estensioni del lavoro e della sua pubblicazione in licenza open sotto forma di progetto sourceforge.

Un altro aspetto che sarà oggetto di lavori futuri riguarda l'estensione dei parametri analizzati da parte del monitor SLA: per esempio, parallelamente al throughput e al fine di gestire politiche specializzate per l'accounting sull'accesso ai servizi, è importante introdurre la gestione del numero di richieste per unità di tempo (tipicamente il giorno).

Inoltre, un ulteriore estensione futura riguarderà il buon uso e, nella fattispecie, i controlli da effettuare sui valori dei parametri passati per le invocazioni ai servizi. In particolare, le specifiche del buon uso saranno estese al fine di abilitare controlli sintattici sul formato e sul range di valori ammissibili per tutti i parametri in input.

Un ulteriore sviluppo da considerare per gli interventi futuri è la realizzazione dei componenti per l'interpretazione di altri linguaggi, oltre a SLAng, per la definizione di Service Level Agreements.

Appendice A

Linguaggio XML per la descrizione degli automi

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.eng.it/ws/SLA/usageAutomata"
      xmlns:tns="http://www.eng.it/ws/SLA/usageAutomata"
      elementFormDefault="qualified">

  <element name="usageDescription" type="tns:_usageDescription"/>

  <complexType name="_usageDescription">
    <sequence>
      <element name="state" minOccurs="1" maxOccurs="unbounded"
           type="tns:_state"/>
    </sequence>
  </complexType>

  <complexType name="_state">
    <sequence>
      <element name="stateLink" minOccurs="0" maxOccurs="unbounded"
           type="tns:_stateLink"/>
    </sequence>
    <attribute name="label" use="required" type="string"/>
    <attribute name="type">
      <simpleType>
        <restriction base="string">
          <enumeration value="initial"/>
          <enumeration value="final"/>
          <enumeration value="error"/>
        </restriction>
```

```
        </simpleType>
      </attribute>
    </complexType>

    <complexType name="_stateLink">
      <simpleContent>
        <extension base="string">
          <attribute name="label" type="string"/>
        </extension>
      </simpleContent>
    </complexType>

  </schema>
```

Appendice B

Esempio di SLA

```
<?xml version="1.0" encoding="UTF-8"?>
<XMI xmlns:SLAng="file:////slangta.emofxmi" xmi.version="1.2">
  <!--
    This document is in XMI format according to the OMG XML Metadata
    Interchange (XMI) Specification v.1.2, OMG Document
         formal/02-01-01
    (http://www.omg.org/)
  -->
  <XMI.header>
    <XMI.metamodel href="file:////slangta.emofxmi" />
  </XMI.header>

  <XMI.content>
    <SLAng:Duration unit="S" value="5.0" xmi.id="mofid:21830" />
    <SLAng:Duration unit="mS" value="330.0" xmi.id="mofid:21833" />
    <SLAng:PartyDefinition description="fornitore"
      sLA="mofid:21838" xmi.id="mofid:21834" />
    <SLAng:PartyDefinition description="cliente"
      sLA="mofid:21838" xmi.id="mofid:21836" />

    <SLAng:SLA xmi.id="mofid:21838">
      <SLAng:SLA.services>
        <SLAng:ServiceDefinition xmi.idref="mofid:21840" />
      </SLAng:SLA.services>
      <SLAng:SLA.auxiliaryClauses>
        <SLAng:AuxiliaryClause xmi.idref="mofid:21855" />
        <SLAng:AuxiliaryClause xmi.idref="mofid:21862" />
        <SLAng:AuxiliaryClause xmi.idref="mofid:21867" />
        <SLAng:AuxiliaryClause xmi.idref="mofid:21884" />
        <SLAng:AuxiliaryClause xmi.idref="mofid:21889" />
        <SLAng:AuxiliaryClause xmi.idref="mofid:21894" />
```

```
  </SLAng:SLA.auxiliaryClauses>
  <SLAng:SLA.parties>
    <SLAng:PartyDefinition xmi.idref="mofid:21836" />
    <SLAng:PartyDefinition xmi.idref="mofid:21834" />
  </SLAng:SLA.parties>
</SLAng:SLA>

<SLAng:ElectronicServiceDefinition
  client="mofid:21836" description="ServizioAerei"
       provider="mofid:21834"
  sLA="mofid:21838" xmi.id="mofid:21840" />

<SLAng:OperationDefinition description="checkFlights"
  operation="mofid:21878" xmi.id="mofid:21843" />
<SLAng:OperationDefinition description="getFlightDetails"
  operation="mofid:21879" xmi.id="mofid:21845" />
<SLAng:OperationDefinition description="reserveFlight"
  operation="mofid:21881" xmi.id="mofid:21849" />
<SLAng:OperationDefinition description="confirmFlightReservation"
  operation="mofid:21882" xmi.id="mofid:21851" />
<SLAng:OperationDefinition description="getTicketDetails"
  operation="mofid:21883" xmi.id="mofid:21853" />

<SLAng:InputThroughputScheduledSlidingPenaltyClause
  concurrency="80" sLA="mofid:21838" window="mofid:21830"
       xmi.id="mofid:21855" />
<SLAng:FixedLatencyFailureModeDefinition
  maxDuration="mofid:21833" operations="mofid:21843"
       sLA="mofid:21838"
  xmi.id="mofid:21862" />
<SLAng:FixedLatencyFailureModeDefinition
  maxDuration="mofid:21833" operations="mofid:21845"
       sLA="mofid:21838"
  xmi.id="mofid:21867" />
<SLAng:FixedLatencyFailureModeDefinition
  maxDuration="mofid:21833" operations="mofid:21849"
       sLA="mofid:21838"
  xmi.id="mofid:21884" />
<SLAng:FixedLatencyFailureModeDefinition
  maxDuration="mofid:21833" operations="mofid:21851"
       sLA="mofid:21838"
  xmi.id="mofid:21889" />
<SLAng:FixedLatencyFailureModeDefinition
  maxDuration="mofid:21833" operations="mofid:21853"
       sLA="mofid:21838"
  xmi.id="mofid:21894" />
```

```
<SLAng:ElectronicServiceInterface
  xmi.id="mofid:21877">
  <SLAng:ElectronicServiceInterface.operations>
    <SLAng:Operation xmi.idref="mofid:21883" />
    <SLAng:Operation xmi.idref="mofid:21881" />
    <SLAng:Operation xmi.idref="mofid:21882" />
    <SLAng:Operation xmi.idref="mofid:21878" />
    <SLAng:Operation xmi.idref="mofid:21879" />
  </SLAng:ElectronicServiceInterface.operations>
  <tns:usageDescription
       xmlns:tns="http://www.eng.it/ws/SLA/usageAutomata"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <tns:state label="initial" type="initial">
      <tns:stateLink label="list">list</tns:stateLink>
      <tns:stateLink label="getDetails">getDetails</tns:stateLink>
      <tns:stateLink label="reserve">reserve</tns:stateLink>

      <tns:stateLink label="confirm">error</tns:stateLink>
      <tns:stateLink label="printTicket">error</tns:stateLink>
    </tns:state>
    <tns:state label="list">
      <tns:stateLink label="list">list</tns:stateLink>
      <tns:stateLink label="getDetails">getDetails</tns:stateLink>
      <tns:stateLink label="reserve">reserve</tns:stateLink>

      <tns:stateLink label="confirm">error</tns:stateLink>
      <tns:stateLink label="printTicket">error</tns:stateLink>
    </tns:state>
    <tns:state label="getDetails">
      <tns:stateLink label="list">list</tns:stateLink>
      <tns:stateLink label="getDetails">getDetails</tns:stateLink>
      <tns:stateLink label="reserve">reserve</tns:stateLink>

      <tns:stateLink label="confirm">error</tns:stateLink>
      <tns:stateLink label="printTicket">error</tns:stateLink>
    </tns:state>
    <tns:state label="reserve">
      <tns:stateLink label="list">list</tns:stateLink>
      <tns:stateLink label="getDetails">getDetails</tns:stateLink>
      <tns:stateLink label="reserve">reserve</tns:stateLink>
      <tns:stateLink label="confirm">confirm</tns:stateLink>

      <tns:stateLink label="printTicket">error</tns:stateLink>
    </tns:state>
    <tns:state label="confirm">
      <tns:stateLink label="list">list</tns:stateLink>
      <tns:stateLink label="getDetails">getDetails</tns:stateLink>
```

```
      <tns:stateLink label="reserve">reserve</tns:stateLink>
      <tns:stateLink label="printTicket">printTicket</tns:stateLink>

      <tns:stateLink label="confirm">error</tns:stateLink>
    </tns:state>
    <tns:state label="printTicket">
      <tns:stateLink label="list">list</tns:stateLink>
      <tns:stateLink label="getDetails">getDetails</tns:stateLink>
      <tns:stateLink label="reserve">reserve</tns:stateLink>
      <tns:stateLink label="printTicket">printTicket</tns:stateLink>

      <tns:stateLink label="confirm">error</tns:stateLink>
    </tns:state>
    <tns:state label="error" type="error">
      <tns:stateLink label="list">list</tns:stateLink>
      <tns:stateLink label="getDetails">getDetails</tns:stateLink>
      <tns:stateLink label="reserve">reserve</tns:stateLink>

      <tns:stateLink label="confirm">error</tns:stateLink>
      <tns:stateLink label="printTicket">error</tns:stateLink>
    </tns:state>
  </tns:usageDescription>
</SLAng:ElectronicServiceInterface>

<SLAng:Operation interface="mofid:21877" xmi.id="mofid:21878">
  <SLAng:Operation.definitions>
    <SLAng:OperationDefinition xmi.idref="mofid:21843" />
  </SLAng:Operation.definitions>
</SLAng:Operation>

<SLAng:Operation interface="mofid:21877" xmi.id="mofid:21879">
  <SLAng:Operation.definitions>
    <SLAng:OperationDefinition xmi.idref="mofid:21845" />
  </SLAng:Operation.definitions>
</SLAng:Operation>

<SLAng:Operation interface="mofid:21877" xmi.id="mofid:21881">
  <SLAng:Operation.definitions>
    <SLAng:OperationDefinition xmi.idref="mofid:21849" />
  </SLAng:Operation.definitions>
</SLAng:Operation>

<SLAng:Operation interface="mofid:21877" xmi.id="mofid:21882">
  <SLAng:Operation.definitions>
    <SLAng:OperationDefinition xmi.idref="mofid:21851" />
  </SLAng:Operation.definitions>
</SLAng:Operation>
```

```
      <SLAng:Operation interface="mofid:21877" xmi.id="mofid:21883">
        <SLAng:Operation.definitions>
          <SLAng:OperationDefinition xmi.idref="mofid:21853" />
        </SLAng:Operation.definitions>
      </SLAng:Operation>
    </XMI.content>
  </XMI>
```

Appendice C

Modellazione di uno user agent

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.eng.it/ws/test/userAgent"
      xmlns:tns="http://www.eng.it/ws/test/userAgent"
      elementFormDefault="qualified">

  <element name="configuration">
    <complexType>
      <sequence>
        <element name="concurrentUserAgents" type="unsignedInt"/>
        <element name="testDuration" type="unsignedInt"/>
        <element name="userAgent" type="tns:_userAgent"/>
      </sequence>
    </complexType>
  </element>

  <complexType name="_userAgent">
    <sequence>
      <element name="agentParam" minOccurs="0" maxOccurs="unbounded">
        <complexType>
          <simpleContent>
            <extension base="string">
              <attribute name="name" type="string" use="required"/>
              <attribute name="type" type="string" use="optional"/>
            </extension>
          </simpleContent>
        </complexType>
      </element>
      <element name="behavior" type="tns:_behavior"/>
    </sequence>
  </complexType>
```

```
<complexType name="_behavior">
  <sequence>
    <element name="state" minOccurs="1" maxOccurs="unbounded"
         type="tns:_state"/>
  </sequence>
</complexType>

<complexType name="_state">
  <sequence>
    <element name="stateBehavior">
      <complexType>
        <sequence>
          <element name="stateAction" minOccurs="0"
               maxOccurs="unbounded" type="tns:_stateAction"/>
        </sequence>
      </complexType>
    </element>
    <element name="stateTransitions">
      <complexType>
        <sequence>
          <element name="stateLink" minOccurs="0"
               maxOccurs="unbounded" type="tns:_stateLink"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
  <attribute name="label" use="required" type="string"/>
  <attribute name="type">
    <simpleType>
      <restriction base="string">
        <enumeration value="initial"/>
        <enumeration value="final"/>
      </restriction>
    </simpleType>
  </attribute>
</complexType>

<complexType name="_stateAction">
  <sequence>
    <element name="stateActionParam" minOccurs="0"
         maxOccurs="unbounded">
      <complexType>
        <simpleContent>
          <extension base="string">
            <attribute name="name" type="string" use="required"/>
            <attribute name="type" type="string" use="optional"/>
```

```
            </extension>
          </simpleContent>
        </complexType>
      </element>
    </sequence>
    <attribute name="class" type="string"/>
  </complexType>

  <complexType name="_stateLink">
    <sequence>
      <element name="likelyhood" type="float"/>
      <element name="meanTransitionDelay">
        <complexType>
          <simpleContent>
            <extension base="float">
              <attribute name="type">
                <simpleType>
                  <restriction base="string">
                    <enumeration value="fixed"/>
                    <enumeration value="random"/>
                  </restriction>
                </simpleType>
              </attribute>
            </extension>
          </simpleContent>
        </complexType>
      </element>
      <!-- stato d'arrivo -->
      <element name="targetState" type="string"/>
    </sequence>
    <attribute name="label" type="string"/>
  </complexType>

</schema>
```

Appendice D

Automa user agent

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:configuration xmlns:tns="http://www.eng.it/ws/test/userAgent"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.eng.it/ws/test/userAgent
      userAgentSpecs.xsd ">
  <tns:concurrentUserAgents>20</tns:concurrentUserAgents>
  <tns:testDuration>7200000</tns:testDuration>
  <tns:userAgent xmlns="http://www.eng.it/ws/test/userAgent">
    <tns:behavior>
      <tns:state label="initial" type="initial">
        <tns:stateBehavior>
          <tns:stateAction
              class="it.eng.ws.test.stateAction.InitialState" />
        </tns:stateBehavior>
        <tns:stateTransitions>
          <tns:stateLink>
            <tns:likelyhood>1.0</tns:likelyhood>
            <tns:meanTransitionDelay
                type="random">1000</tns:meanTransitionDelay>
            <tns:targetState>checkFlights</tns:targetState>
          </tns:stateLink>
        </tns:stateTransitions>
      </tns:state>

      <tns:state label="checkFlights">
        <tns:stateBehavior>
          <tns:stateAction
              class="it.eng.ws.test.stateAction.CheckFlights" />
        </tns:stateBehavior>
        <tns:stateTransitions>
          <tns:stateLink>
```

```
      <tns:likelyhood>0.3</tns:likelyhood>
      <tns:meanTransitionDelay
          type="random">15000</tns:meanTransitionDelay>
      <tns:targetState>checkFlights</tns:targetState>
    </tns:stateLink>
    <tns:stateLink>
      <tns:likelyhood>0.7</tns:likelyhood>
      <tns:meanTransitionDelay
          type="random">15000</tns:meanTransitionDelay>
      <tns:targetState>getFlightDetails</tns:targetState>
    </tns:stateLink>
  </tns:stateTransitions>
</tns:state>

<tns:state label="getFlightDetails">
  <tns:stateBehavior>
    <tns:stateAction
        class="it.eng.ws.test.stateAction.FlightDetails" />
  </tns:stateBehavior>
  <tns:stateTransitions>
    <tns:stateLink>
      <tns:likelyhood>0.4</tns:likelyhood>
      <tns:meanTransitionDelay
          type="random">6000</tns:meanTransitionDelay>
      <tns:targetState>checkFlights</tns:targetState>
    </tns:stateLink>
    <tns:stateLink>
      <tns:likelyhood>0.6</tns:likelyhood>
      <tns:meanTransitionDelay
          type="random">8000</tns:meanTransitionDelay>
      <tns:targetState>reserveFlight</tns:targetState>
    </tns:stateLink>
  </tns:stateTransitions>
</tns:state>

<tns:state label="reserveFlight">
  <tns:stateBehavior>
    <tns:stateAction
        class="it.eng.ws.test.stateAction.ReserveFlight" />
  </tns:stateBehavior>
  <tns:stateTransitions>
    <tns:stateLink>
      <tns:likelyhood>0.6</tns:likelyhood>
      <tns:meanTransitionDelay
          type="random">120000</tns:meanTransitionDelay>
      <tns:targetState>confirmFlightReservation</tns:targetState>
    </tns:stateLink>
```

```
            <tns:stateLink>
              <tns:likelyhood>0.4</tns:likelyhood>
              <tns:meanTransitionDelay
                    type="random">3000</tns:meanTransitionDelay>
              <tns:targetState>checkFlights</tns:targetState>
            </tns:stateLink>
          </tns:stateTransitions>
        </tns:state>

        <tns:state label="confirmFlightReservation">
          <tns:stateBehavior>
            <tns:stateAction
                  class="it.eng.ws.test.stateAction.ConfirmFlightReservation"
                  />
          </tns:stateBehavior>
          <tns:stateTransitions>
            <tns:stateLink>
              <tns:likelyhood>1</tns:likelyhood>
              <tns:meanTransitionDelay
                    type="random">10000</tns:meanTransitionDelay>
              <tns:targetState>getTicketDetails</tns:targetState>
            </tns:stateLink>
          </tns:stateTransitions>
        </tns:state>

        <tns:state label="getTicketDetails">
          <tns:stateBehavior>
            <tns:stateAction
                  class="it.eng.ws.test.stateAction.TicketDetails" />
          </tns:stateBehavior>
          <tns:stateTransitions>
            <tns:stateLink>
              <tns:likelyhood>1</tns:likelyhood>
              <tns:meanTransitionDelay
                    type="random">3000</tns:meanTransitionDelay>
              <tns:targetState>checkFlights</tns:targetState>
            </tns:stateLink>
          </tns:stateTransitions>
        </tns:state>

      </tns:behavior>
    </tns:userAgent>
  </tns:configuration>
```

Bibliografia

[ACKO02] M. Alves, L. Corsello, D. Karrenberg, and C. Ogut. New measurement with the RIPE NCC test traffic measurement setup. In *In Proceedings of PAM 2002 Workshop*, Fort Collins, CO, Mar. 2002. RIPE-NCC-New Project Group.

[AFF+01] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger. Oceano-SLA based management of a computing utility. In *In Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, pages 855 –868, Seattle, WA, USA, May 2001.

[BAFP09] A. Bertolino, G. Angelis, L. Frantzen, and A. Polini. The PLASTIC Framework and Tools for Testing Service-Oriented Applications. In A. De Lucia and F. Ferrucci, editors, *Proceedings of the International Summer School on Software Engineering – ISSSE 2006-2008*, number 5413, pages 106–139, Berlin, Heidelberg, 2009. Springer.

[HDJ08] R. B. Halima, K. Drira, and M. Jmaiel. A QoS-Oriented Reconfigurable Middleware for Self-Healing Web Services. In *ICWS '08: Proceedings of the 2008 IEEE International Conference on Web Services*, pages 104–111, Washington, DC, USA, 2008. IEEE Computer Society.

[HKJH02] S. H. Han, M. S. Kim, H. T. Ju, and J. W. K. Hong. The Architecture of NG-MON: A Passive Network Monitoring System for High-Speed IP Networks. In *DSOM '02: Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, pages 16–27, London, UK, 2002. Springer-Verlag.

[KL03] A. Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *J. Network Syst. Manage.*, 11(1):57–81, 2003.

[KLD+02] A. Keller, H. Ludwig, A. Dan, R. Franck, and R. P. King. Web Service Level Agreement (WSLA) Language Specification. *IBM Corporation*, Jul. 2002.

[KSH00] E. C. Kim, J. G. Song, and C. S. Hong. An integrated CNM architecture for multi-layer networks with simple SLA monitoring and reporting mechanism. *Network Operations and Management Symposium, 2000. NOMS 2000. 2000 IEEE/IFIP*, pages 993 –994, 2000.

[LKHL02] H. J. Lee, M. S. Kim, J. W. Hong, and G. H. Lee. QoS parameters to network performance metrics mapping for SLA monitoring. volume 5. KNOM Rev., 2002.

[LPRT07] G. Lodi, F. Panzieri, D. Rossi, and E. Turrini. SLA-Driven Clustering of QoS-Aware Application Servers. *Software Engineering, IEEE Transactions on*, 33(3):186–197, 2007.

[MPMJS05] G. Morgan, S. Parkin, C. Molina-Jimenez, and J. Skene. Monitoring Middleware for Service Level Agreements in Heterogeneous Environments. In *In Proceedings of the 5th IFIP Conference on e-Commerce, e-Business, and e-Government (I3E*, volume 189, pages 79–94, Poznan, Poland, Oct. 2005. IFIP.

[MRLD09] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. Comprehensive QoS monitoring of Web services and event-based SLA violation detection. In *MWSOC '09: Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing*, pages 1–6, New York, NY, USA, 2009. ACM.

[PSS+05] G. Pacifici, W. Segmuller, M. Spreitzer, M. Steinder, A. Tantawi, and A. Youssef. Managing the Response Time for Multi-Tiered Web Applications. Technical Report RC 23651, IBM, 2005.

[RSE08] F. Raimondi, J. Skene, and W. Emmerich. Efficient online monitoring of web-service SLAs. In Mary Jean Harrold and Gail C. Murphy, editors, *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 170–180, New York, NY, USA, 2008. ACM.

[SH09] J. Spillner and J. Hoyer. SLA-driven Service Marketplace Monitoring with Grand SLAM. In Boris Shishkov, JosÃÍ Cordeiro, and Alpesh Ranchordas, editors, *ICSOFT (2)*, pages 71–74. INSTICC Press, 2009.

[SLE04] J. Skene, D. D. Lamanna, and W. Emmerich. Precise Service Level Agreements. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 179–188, Washington, DC, USA, 2004. IEEE Computer Society.

[TGN+03] M. Tian, A. Gramm, T. Naumowicz, H. Ritter, and J. Schiller. A Concept for QoS Integration in Web Services. In *In 1st Web Services Quality Workshop (WQW2003) at WISE*, pages 149–155. IEEE Computer Society, 2003.

[TK05] N. Thio and S. Karunasekera. Automatic Measurement of a QoS Metric for Web Service Recommendation. In *ASWEC '05:*

Proceedings of the 2005 Australian conference on Software Engineering, pages 202–211, Washington, DC, USA, 2005. IEEE Computer Society.

[TMPE04] V. Tosic, W. Ma, B. Pagurek, and B. Esfandiari. Web Service Offerings Infrastructure (WSOI) Ð A Management Infrastructure for XML Web Services. In *In Proceedings of NOMS (IEEE/IFIP Network Operations and Management Symposium)*, pages 817–830, Seoul, South Korea, 2004. IEEE.

[TPP+03] V. Tosic, B. Pagurek, K. Patel, B. Esfandiari, and W. Ma. Management applications of the Web Service Offerings Language (WSOL). In *In 15th International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 2681, pages 468–484, Velden, Austria, Jun. 2003. Lecture Notes in Computer Science (LNCS), Springer-Verlag.

[Wal95] S. Waldbusser. Remote Network Monitoring Management Information Base. United States, Feb. 1995. RFC Editor.

[WLLM07] Q. Wang, Y. Liu, M. Li, and H. Mei. An Online Monitoring Approach for Web services. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 335–342, Washington, DC, USA, 2007. IEEE Computer Society.

[ZK02] T. Zhao and V. Karamcheti. Enforcing Resource Sharing Agreements among Distributed Server Clusters. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, volume 1, page 141, Washington, DC, USA, 2002. IEEE Computer Society.