Università degli Studi di Pisa
Dipartimento di Informatica
Corso di Laurea Specialistica in Tecnologie Informatiche

Tesi di Laurea

Monitoraggio Adattivo del livello di qualità in Architetture Orientate ai Servizi

Candidato:
Francesco Cogoni

Relatore :
Ing. Antonia Bertolino
Co-relatore :
Guglielmo De Angelis

Controrelatore :
Antonio Brogi

Anno Accademico 2006-2007

Indice

Capitolo 1

Introduzione

Il recente affermarsi di architetture SOA e di sistemi basati sui Web Service ha determinato lo sviluppo di strumenti tecnologici e metodologie atte al monitoraggio e alla verfica della qualità dei servizi forniti. La verifica a tempo di esecuzione dei parametri relativi la qualità del servizio espressi negli SLA (*run time SLA checking*) è una delle attività fondamentali dell'SLA management. L'SLA checking deve essere preciso e continuo al fine di rilevare, tempestivamente, ogni possibile violazione dei SLA.
Le tecniche di SLA checking attualmente disponibili, tuttavia, non sono scalabili negli scenari caratterizzati da SLA complessi e da un numero elevato ed eterogeneo di client, i cui dispositivi hanno capacità di banda e memorizzazione limitate. Queste tecniche trascurano, infatti, l'overhead dello stesso meccanismo di SLA checking, responsabile del consumo di preziose risorser server e della conseguente riduzione della capacità del service provider. Congiuntamente alla crescita del carico di lavoro del servizio, l'overhead può essere causa indiretta di ulteriori violazioni degli SLA.

Questa tesi elabora un nuovo approccio basato su un meccanismo di SLA checking adattivo (Monitoraggio adattivo di SLA) che offre maggiori

opportunità di scalare negli scenari caratterizzati dalla combinazione di Web Service e ambienti pervasivi mobili. Il monitoraggio e la verifica degli SLA sono opportunamente adattati alle condizioni di funzionamento e di carico di lavoro del servizio. Quando è possibile l'SLA checking è svolto in maniera più leggera, con un minor numero di controlli sugli SLA, al fine di risparmiare preziose risorse server da un inutile utilizzo. Al contrario l'SLA checking intensifica i controlli nelle situazioni in cui è più probabile il verificarsi di una violazione di SLA.

Il prossimo capitolo espone i principi architetturali che caratterizzano le architetture (SOA): l'indipendenza dei servizi da uno specifico linguaggio di programmazione, dalla piattaforma e dal sistema operativo nel quale essi sono in esecuzione, attraverso il paradigma di collaborazione del "publish, find, bind and invoke" che permette ad una applicazione/servizio di poter ricercare dinamicamente e invocare le funzionalità offerte da un'altro servizio. Successivamente viene presentata la tecnologia Web Service, considerata la più valida implementazione del paradigma SOA, attraverso la trattazione dei principali standard tecnologici: SOAP, WSDL, UDDI ed il loro legame con il paradigma di collaborazione "publish, find, bind and invoke".

Nello stesso capitolo sono inoltre presentati: alcuni esempi di tecnologie per la definizione degli aspetti extra funzionali dei Web Service come i livelli di qualità del servizio (WSLA, WSA) ed infine il framework Axis, utilizzato per la realizzazione di questo lavoro. Il capitolo 3 fornisce lo studio teorico dell'approccio e presenta le motivazioni che giustificano un meccanismo adattivo di questo tipo. Il capitolo 4 presenta, invece, gli aspetti tecnologici riguardanti l'implementazione del meccanismo di SLA checking sull'architettura concreta del framework di Apache Axis. Il capitolo 5 presenta, infine, un particolare caso di studio condotto sull'architettura presentata nei capi-

toli 3 e 4, congiuntamente all'analisi e alla discussione dei risultati ottenuti. Il capitolo 6 propone le considerazioni conclusive di questo lavoro di tesi e prospetta alcuni esempi di possibili sviluppi futuri dello studio presentato.

Capitolo 2

Contesto tecnologico

2.1 Introduzione

La graduale ma costante evoluzione del Web ha determinato negli ultimi decenni un cambiamento radicale nel modo in cui gli utenti e i sistemi software interagiscono e comunicano, offrendo loro nuove opportunità d'integrazione in un ambiente sempre più dinamico e globalizzato.
Inizialmente, il Web era costituito da applicazioni "statiche", sviluppate con la tecnologia HTML, che permettevano all'utente la navigazione e la sola consultazione delle informazioni.
Successivamente l'esigenza dell'utente di interagire con i sistemi residenti sul Web ha portato alla nascita di tecnologie quali ASP, JSP, Php e XML. Le prime tre tecnologie hanno reso "dinamici" la modifica e l'aggiornamento delle pagine HTML direttamente via Web, mentre la tecnologia XML ha fornito una modalità comune per la rappresentazione di informazioni strutturate e facilmente trasferibili sulla rete.

Attualmente si assiste ad un'ulteriore trasformazione del Web: internet non ha solo l'obiettivo di garantire l'interazione degli utenti con i sistemi e le

applicazioni situati in rete, ma anche quello di permettere l'interazione e la cooperazione fra loro delle medesime applicazioni. Da semplice contenitore ed archivio di informazioni consultabili dal navigatore, il Web si evolve in un sistema costituito da applicazioni distribuite che cooperano e integrano le proprie funzionalità offrendo ai propri fruitori, siano essi semplici utenti o a loro volta applicazioni, nuovi servizi e opportunità.

Quest'ultima fase dell'evoluzione del Web vede, inoltre, l'utilizzo delle architetture orientate ai servizi (SOA) basate sulla tecnologia Web Service: un modello architetturale per la realizzazione di sistemi software distribuiti, organizzati secondo il concetto di "servizio". SOA e i Web Service forniscono, quindi, una modalità standard ed interoperabile di interfacciamento tra applicazioni che esportano e integrano le proprie funzionalità incapsulando la logica applicativa all'interno dei servizi (vedi sezione 2.2).

L'affermarsi del paradigma SOA in ambito aziendale, congiuntamente all'innovazione delle tecnologie di comunicazione, ha determinato l'emergere di una nuova classe di applicazioni software sempre più complesse il cui sviluppo è il frutto dell'integrazione, attraverso il Web, di sistemi software etereogenei tra loro ed appartententi a differenti organizzazioni.
Queste innovazioni tecnologiche rappresentano un'importante opportunità di soddisfare le esigenze degli utenti fornendo loro delle funzionalità e dei servizi nuovi: si consideri ad esempio le applicazioni che forniscono servizi di streaming audio e video in ambienti mobili di ultima generazione.
I servizi forniti sono spesso caratterizzati da requisiti di qualità del servizio (QoS) sempre più stringenti da parte degli utenti: mantenere accettabile la qualità del servizio erogato è una necessità di vitale importanza per rendere realmente fruibile la stesso servizio.
Tuttavia, offrire agli utenti determinate funzionalità e garantire una qualità

del servizio accettabile nelle applicazioni SOA è una sfida che presenta diverse difficoltà poiché il controllo sull'elaborazione del processo applicativo non appartiene più a una singola organizzazione/azienda. Le caratteristiche architetturali di SOA impongono, quindi, che la qualità del servizio sia stabilita attraverso la negoziazione e la conseguente creazione di un "Service Level Agreement" (SLA): un contratto stipulato tra il fornitore e il fruitore del servizio che definisce obblighi e doveri tra le parti contraenti. Il recente affermarsi di architetture SOA e di sistemi basati sui Web Service ha determinato, inoltre, lo sviluppo di strumenti tecnologici e metodologie atte al monitoraggio e alla verifica della qualità dei servizi forniti.

Le prossime sezioni presenteranno i principi architetturali delle applicazioni orientate ai servizi (SOA) e la tecnologia Web Service, attraverso la trattazione dei loro principali standard tecnologici di riferimento.

2.2 Architetture Orientate ai Servizi

Con il termine *Service-Oriented Architecture* (*SOA*) si indica un modello architetturale per la realizzazione di sistemi software distribuiti.
Le unità fondamentali di tale modello sono i servizi, applicazioni indipendenti tra loro che interagiscono all'interno di una rete di comunicazione.
Ogni servizio può utilizzare le funzionalità rese disponibili dagli altri servizi e mettere a disposizione le proprie. In genere, un servizio deve poter essere *ricercato* e *recuperato* dinamicamente attraverso la definizione della sua *interfaccia* che lo renda *accessibile* in modo trasparente rispetto alla sua allocazione. La pubblicazione dell'interfaccia nella rete rende note, infatti, le modalità di accesso al servizio.

Ogni servizio è definito nei termini di ciò che fa, in base alla definizione della propria interfaccia, astrazione attraverso la quale si nascondono le

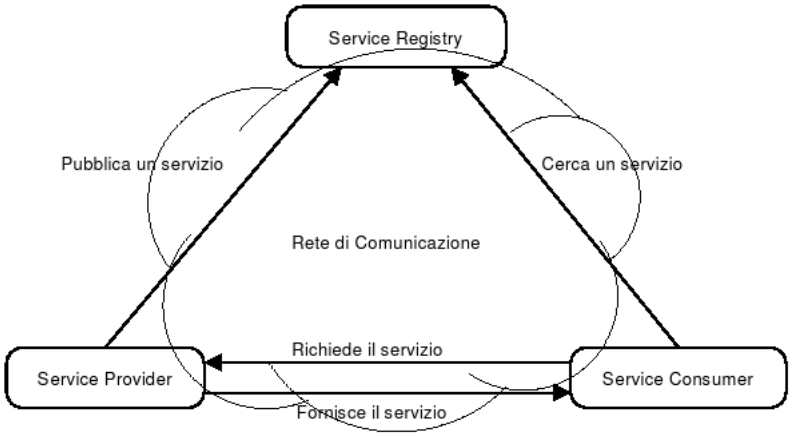

Figura 2.1: Attori nelle architetture SOA

tecnologie utilizzate per la sua implementazione.
Il servizio è indipendente dal linguaggio di programmazione utilizzato per realizzarlo (cross-language) e sopratutto dalla piattaforma e dalle caratteristiche del sistema operativo nel quale è in esecuzione (cross-platform).

All'interno dell'architettura SOA si individuano tre attori: il *Service Provider*, il *Service Consumer* e il *Service Registry* [13].
Essi assumono differenti ruoli che permettono di descrivere l'interazione tra le componenti del sistema. Il *Service Provider* è il fornitore del servizio: quest'entità *network-addressable* accetta ed esegue le richieste degli utenti, conformi alle specifiche di interfaccia.
Il *Service Consumer* è, invece, l'entità che richiede il servizio: essa può consistere in un'applicazione, in un modulo software o in un'altro servizio.
Il *Service Registry* è, infine, l'entità che contiene il registro dei servizi disponibili: permette la ricerca dei Service Provider e fornisce le relative interfacce ai Service Consumer [13].
La figura 2.1 mostra le interazioni che coinvolgono le tre entità, disposte all'interno di una rete di comunicazione, secondo il paradigma di collabora-

zione “publish find, bind and invoke” .
Il Service Provider pubblica sul service Registry URL e modalità d’accesso del servizio reso disponibile (publish). Il Service Consumer consulta invece il Service Registry (find), dal quale ottiene le informazioni necessarie a comunicare con il provider e ad accedere alle funzionalità offerte dal servizio cercato (bind and invoke).

Le caratteristiche architetturali finora esposte , si riferiscono ad interazioni di tipo *point to point* tra servizi . SOA, tuttavia, è in grado di descrivere scenari ben più complessi di collaborazione e composizione di servizi, con l’ausilio di due differenti paradigmi, denominati *orchestration* e *choreography.*
Orchestration pone all’interno dell’architettura una singola entità come *responsabile* e *coordinatrice* della corretta evoluzione della logica applicativa; al contrario *Choreography* prevede che ogni singola componente sia individualmente responsabile del processo applicativo.

Oltre le suddette caratteristiche di *ricerca* e *invocazione dinamica* dei servizi, l’architettura SOA presenta un basso grado di accoppiamento tra le sue componenti (*loosely coupled*) [23]. Le dipendenze tecnologiche e funzionali tra i servizi sono infatti, di numero limitato. Tale caratteristica rende il sistema facilmente modificabile e flessibile.

SOA stabilisce, in altri termini, i principi architetturali, indipendenti da una particolare tecnologia, in grado di definire applicazioni realmente riusabili e facilmente integrabili in ambiente eterogeneo [23].

2.3 Tecnologie per Web Service

I Web Service sono una tecnologia considerata la più valida implementazione del paradigma SOA [13]. Questa tecnologia fornisce un'approccio distribuito per l'integrazione di applicazioni eterogenee che comunicano sopra l'infrastruttura internet. Il riutilizzo dell'infrastruttura di comunicazione già esistente, internet, è un elemento del successo della tecnologia Web Service. Il *Web Services Architecture Working Group* [2] (del W3C) definisce un Web Service come:

> " un'applicazione software identificata da un URI (Uniform Resource Identifier), le cui interfacce pubbliche e collegamenti sono definiti e descritti come documenti XML, in un formato comprensibile alla macchina (specificatamente WSDL). La sua definizione può essere ricercata da altri agenti software situati su una rete, che possono interagire direttamente con il Web Service, mediante le modalità specificate nella sua definizione. Le entità coinvolte nella comunicazione scambiano tra loro messaggi (SOAP), utilizzando protocolli Internet (tipicamente HTTP)."

I principali standard tecnologici su cui si fondano i Web service sono: l'eXtensible Markup Language (XML), il Simple Object Access Protocol (SOAP) [26], l'Universal Description, Discovery and Integration (UDDI) [17] e il Web Services Description Language (WSDL) [25].

Tali tecnologie permettono di incapsulare la logica applicativa nei servizi, che quindi possono essere *pubblicati*, *ricercati* e *invocati* sul Web [13]. Un servizio web è infatti un'applicazione *self-describing* e *modulare* poiché le entità coinvolte nella comunicazione non sono a conoscenza del formato e del contenuto dei messaggi scambiati . La definizione del formato viaggia

con lo stesso messaggio e nessuna informazione esterna è necessaria per lo scambio dei messaggi [23].

L'interazione tra il service provider e il service consumer è interamente *indipendente dal linguaggio* e dalla *piattaforma* in quanto la comunicazione tra le due entità viene regolata da un documento WSDL che descrive il servizio e ne definisce l'interfaccia. WSDL astrae tutti i dettagli di implementazione e di piattaforma, garantendo così l'*interoperabilità* tra applicazioni [23]. Attraverso la tecnologia UDDI, i Web Service forniscono uno strumento per rendere dinamica sia la ricerca che l'invocazione dei servizi stessi. La figura 2.2 illustra le interazioni che coinvolgono alcuni servizi web di un'applicazione SOA, secondo il paradigma "publish find, bind and invoke".

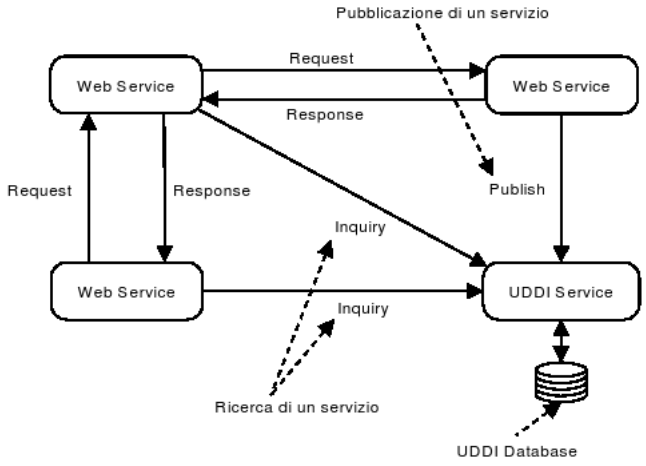

Figura 2.2: Interazioni tra Web Service in un'architettura SOA.

Gli aspetti tecnologici citati, saranno argomento e oggetto di approfondimento delle sezioni seguenti, le quali presenteranno inoltre esempi di tecnologie per la definizione degli aspetti extra-funzionali dei Web Service come i livelli di qualità del servizio (QoS).

2.3.1 WSDL

La tecnologia *Web Services Description Language* (WSDL) definisce un linguaggio XML, utilizzato per descrivere un Web Service. WSDL considera un servizio come un insieme di endpoint che interoperano attraverso lo scambio di messaggi. Quando necessario, le operazioni invocabili sul servizio possono essere istanziate come concreti endpoint attraverso la definizione di informazioni implementative quali il protocollo utilizzato (ad esempio SOAP, HTTP GET, MIME) e gli indirizzi presso i quali sono accessibili [25]. Un documento WSDL fornisce quindi le sole informazioni sintattiche necessarie per la corretta invocazione del servizio, astrae dalla descrizione tutti i dettagli implementativi [1]. La struttura di un documento WSDL (figura 2.3) è com-

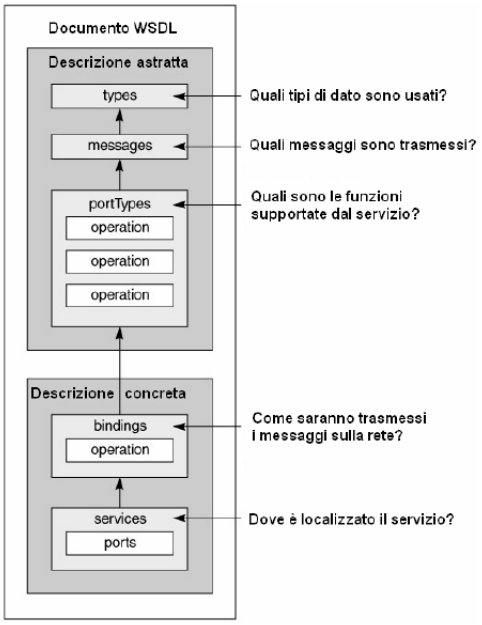

Figura 2.3: Struttura di un documento WSDL

posta di una descrizione *astratta*, che stabilisce ciò che è offerto dal servizio, e di una *concreta*, che determina la propria collocazione e la modalità di comunicazione. All'interno della parte *astratta* sono presenti diversi elementi

tag, ognuno dei quali identifica una distinta sezione del documento.
Gli elementi *Types* definiscono i tipi di dato che sono coinvolti nello scambio di messaggi. WSDL utilizza il sistema standard per la tipizzazione, definito per gli schemi XML (*XSD, XML Schema Definition*), ed inoltre permette la definizione di altri tipi di dato all'interno del documento [25].
La sezione relativa ai *message* definisce l'*input* e l'*output* dei servizi. Ogni elemento *message* racchiude le informazioni relative ad uno specifico messaggio, tra i quali i parametri e i rispettivi tipi.
L'elemento *PortType* contiene invece, i dettagli relativi alle operazioni che un Web Service rende utilizzabili. Ogni operazione viene descritta per mezzo di un ulteriore elemento chiamato *operation*. Al suo interno vi sono i messagi di input e/o di output accettati dal servizio e l'ordine da seguire per il passaggio dei parametri.
La parte *concreta* del documento è composta dalle sezioni degli elementi *Binding* e *Service* che identificano rispettivamente, il protocollo di comunicazione e la relativa collocazione del servizio.
Il tag *binding* associa gli *endpoint* e il protocollo utilizzato per lo scambio di messaggi alle operazioni offerte dal Web Service. Il protocollo di comunicazione può essere HTTP GET/POST, MIME o SOAP; quest'ultimo il più utilizzato. L'elemento *binding* ospita e definisce tanti elementi *operation* quante sono le operazioni supportate dal Web Service. Il *binding* inoltre specifica lo stile del collegamento, che a sua volta può essere di tipo *rpc* o *document*. La stile del collegamento influisce sulla struttura del corpo del messaggio SOAP (elemento *Body*). Lo stile *document*, infatti, prevede che il contenuto dell'elemento *Body* del messaggio SOAP sia costituito da un documento. Quando si utilizza lo stile (*rpc*), l'elemento *Body* contiene l'invocazione ad una procedura remota detta *RPC*.

L'elemento *service*, relativo alla localizzazione del servizio, raccoglie l'elenco di tutti le funzioni offerte dal servizio, ed ognuna di esse è definita da un elemento *port* che identifica il collegamento utilizzato e il relativo indirizzo URL (endpoint) associato.

2.3.2 SOAP

SOAP (Simple Object Access Protocol) è un protocollo di comunicazione semplice ed estendibile, progettato per lo scambio di messaggi in ambiente distribuito [26] . Il formato dei messaggi scambiati è basato su XML. Sebbene originariamente, tale tecnologia sia stata concepita per appianare le differenze tra le diverse piattaforme di comunicazione basate su RPC, lo standard SOAP è divenuto il protocollo per lo scambio di messaggi, più usato dai Web Service [23]. Talvolta con l'acronimo SOAP ci si riferisce a "Service Oriented Architecture Protocol" anziché all'originario "Simple Object Access Protocol" [23]. Le specifiche SOAP definiscono un formato dei messaggi standard che consiste in un documento capace di ospitare infomazioni sia di tipo RPC che di tipo document-centric: quest'ultimo è il più comune.

Lo standard SOAP fornisce quindi un'infrastruttura che consente di descrivere il contenuto dei messaggi, e un insieme di regole di codifica (encoding rule) per rappresentare i tipi di dato definiti. SOAP è un protocollo di alto livello completamente indipendente dal protocollo di trasmissione sottostante. Quest'ultimo infatti può essere indifferentemente un protocollo JMS (Java Message Service), SMTP (Simple Mail Transfer Protocol), MIME (Multipurpose Internet Message Encapsulation) o altri. HTTP è il più utilizzato.

Il protocollo HTTP permette ai messaggi SOAP di attraversare particola-

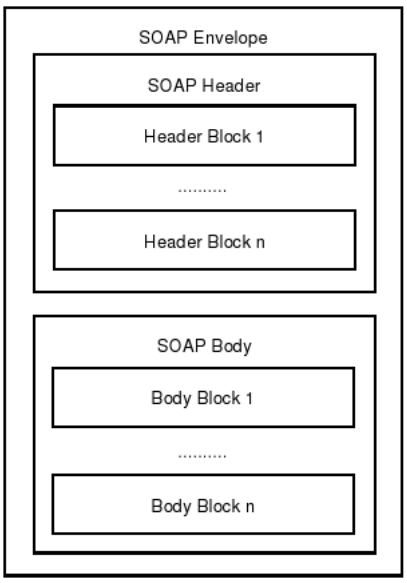

Figura 2.4: Rappresentazione di un messaggio SOAP

ri sistemi di filtraggio del traffico, chiamati *firewall*. Insieme alla specifica WSDL, SOAP definisce un formato standard di descrizione degli endpoint che contiene le principali caratteristiche che un'applicazione service oriented deve avere: è indipendente dalla piattaforma e non legato a nessun specifico linguaggio e dalla relativa tecnologia.

La struttura di un messaggio SOAP, mostrata in figura 2.4, è costituita da un contenitore chiamato *envelope*, all'interno del quale vengono distinte due sezioni principali denominate *Header* e *Body*. Esse contengono rispettivamente l'intestazione e il corpo del messaggio.
La sezione *Header* è *opzionale* e strutturata in blocchi: specifica le informazioni riguardanti l'instradamento del messaggio sulla rete. La sezione *Body* contiene invece l'informazione principale di un messaggio SOAP sia sottoforma di semplici dati che di chiamate a procedura di tipo RPC. In caso di chiamata RPC, tali informazioni sono l'indirizzo del destinatario e il nome del metodo invocato con i relativi parametri di *input* e di *output*.

Più precisamente, SOAP-RPC stabilisce sia dove inserire e sia come strutturare queste informazioni all'interno del corpo di un messaggio SOAP.
La struttura relativa alla richiesta ha lo stesso nome del metodo invocato, e contiene tanti elementi quanti sono i parametri di input o input/output. Questi elementi hanno lo stesso nome e lo stesso ordine dei parametri del metodo specificato nel documento WSDL. Nel medesimo modo viene modellata anche la risposta: l'unica differenza in questo caso però è che il nome associato alla struttura è per convenzione il nome di tale metodo seguito da "Response". All'interno della risposta si trova un elemento contenente il valore di ritorno del metodo e tanti elementi quanti sono i parametri di output e input/output.

2.3.3 UDDI

Una componente fondamentale dell'architettura orientata ai servizi è il meccanismo di ricerca e binding dinamico dei Web Service Description in cui sono coinvolti i potenziali clienti di un servizio. Questa parte dell'infrastruttura Web Service, è rappresentata da una directory centrale che ospita la descrizione dei servizi web.
Attualmente il principale standard tecnologico per la ricerca dei servizi web è l'*Universal Description, Discovery and Integration* (UDDI). Esso fornisce uno strumento di intermediazione tra i service consumer e i service provider: basato su XML, utilizza SOAP per le comunicazioni da e verso l'esterno, definendo un meccanismo comune per pubblicare e trovare informazioni sui Web Service tramite le loro descrizioni WSDL. UDDI mette a disposizione un registro nel quale possano essere ricercate informazioni "Human Readable", comprensibili all'utente: indirizzi, contatti o altri dati relativi ad una azienda. Lo stesso registro fornisce inoltre informazioni tecniche "Machine

Readable", interpretabili ed utilizzabili dalla macchina, che vengono utilizzate per localizzare un servizio al quale potersi connettere. Un registro UDDI è costituito in realtà da un database distribuito, cioè da più registri dislocati sulla rete e tra loro connesssi, ognuno dei quali si trova sul server di una azienda che contribuisce allo sviluppo di questo archivio pubblico. Il sistema mantiene una centralizzazione virtuale, in altri termini l'informazione che è stata registrata su uno dei registri o nodi del sistema viene successivamente propagata e resa disponibile su tutti gli altri tramite la loro sicronizzazione. Questo aspetto tecnologico, oltre ad alleggerire il carico di lavoro di ogni singolo nodo protegge il sistema da possibili situazioni di failure del database, grazie alla ridondanza dei dati. UDDI può essere immaginato come una sorta di "pagine gialle", in cui si cercano informazioni sulle aziende. La registrazione di un servizio all'interno di un UDDI Registry infatti è costituita di tre parti: Yellow Pages, White Pages, Green Pages. Con le *Yellow Pages* le aziende ed i loro servizi vengono catalogate sotto differenti categorie e classificazioni. Nelle *White Pages* si trovano varie informazioni tra le quali gli indirizzi di una azienda e contatti ad essa relativi. Infine nelle *Green Pages*, si registrano informazioni tecniche relative ai servizi e proprio grazie ad esse questi ultimi possono essere invocati. Le interazioni con un registro UDDI sono suddivisibili in due categorie. Esistono infatti interazioni finalizzate alla ricerca di dati e interazioni volte all'inserimento e la modifica dei dati stessi. Questa distinzione è rappresentata da due API (Application Programming Interface) tramite le quali si accede ai contenuti di UDDI: *Inquiry API* e *Publish API*.

2.3.4 Linguaggi per i Service Level Agreement: Web Service Level Agreement e WS-Agreement

Le architetture orientate ai servizi, sono caratterizzate da meccanismi che definiscono la qualità del servizio (QoS) in base a specifici accordi detti *Service Level Agreement* (*SLA*). Attraverso un SLA viene stabilito un contratto che individua obblighi e doveri delle due parti, il service provider e il service consumer.

In genere, un SLA contiene una descrizione tecnica della qualità del servizio con associata una metrica: queste informazioni sono chiamate Service Level Specifications (SLS).

Con l'utilizzo di SLA nelle architetture SOA, ogni invocazione di servizio protrebbe essere governata da uno specifico accordo; in altri termini un client potrebbe richiedere a un *service registry* un particolare servizio corrispondente alle sue esigenze di QoS. Il service registry potrebbe a sua volta negoziare gli SLA con i service provider riguardo la capacità di servizio richiesta. In uno scenario di questo tipo, il service registry potrebbe ad esempio decidere di accordare un servizio in base ad una serie di parametri quali: il carico di lavoro del server (workload) , le garanzie di qualità che il servizio offre ed infine il prezzo che è necessario pagare per usufruire del servizio.

In un'architettura *agreement-driven*, sia l'applicazione client che gli stessi servizi da essa utilizzati, possono essere equipaggiati di meccanismi per il monitoraggio degli accordi SLA. Allo stesso modo, anche il provider e il cliente del servizio potrebbero utilizzare sistemi di misurazione e gestione della qualità del servizio.

Tale scenario permette di misurare sia la qualità del servizio garantita dal server e sia la qualità del servizio riscontrata dal client. In ambienti distri-

buiti su larga scala, i client e i provider potrebbero scegliere di delegare la gestione e la verifica degli SLA a una terza parte.

Di seguito sono riportati due esempi di linguaggi tra i più significativi per la specifica di un SLA.

Web Service Level Agreement (WSLA) è una specifica XML che determina: i vincoli o clausole prestazionali associati alle condizioni di un Web Service. In particolare, questo linguaggio permette di specificare gli accordi (contratti) tra fornitore del servizio e cliente definendo gli obblighi delle parti coinvolte. Un documento WSLA consta di tre parti principali: la sezione *Parties*, la sezione *Service Definition* e la sezione *Obbligations*. La sezione *Parties* definisce gli attori coinvolti nell'accordo.

Nello specifico, questa sezione contiene i dati sia il fornitore del servizio che del cliente, congiuntamente a quelli di una terza parte coinvolta nel monitoraggio del servizio. Il listato 2.1 mostra un esempio di definizione dei tipi WSLA.

```
<xsd:complexType name="WSLAType">
  <xsd:sequence>
    <xsd:element name="Parties" type="wsla:PartiesType"/>
    <xsd:element name="ServiceDefinition"
        type="wsla:ServiceDefinitionType" maxOccurs="unbounded"/>
    <xsd:element name="Obligations" type="wsla:ObligationsType"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="SLA" type="wsla:WSLAType"/>
```

Listing 2.1: Definizione tipi WSLA

La sezione *Service Definition* del documento racchiude invece: i riferimenti alle operazione del servizio e la definizione dei parametri osservati e delle metriche. I parametri possono essere utilizzati nelle asserzioni al fine di garantire gli accordi.

Le metriche specificano come misurare o calcolare i valori dei parametri. La descrizione di una metrica include infatti anche l'entità incaricata della

misurazione. WSLA fornisce costrutti per la composizione delle metriche, cosiché metriche ad un diverso livello di astrazione possano essere espresse in un medesimo accordo.
La sezione *Obbligations* rappresenta infine il fulcro di una specifica WSLA; in essa si definisce l'impegno a mantenere un particolare stato del servizio per un dato periodo. Con "stato del servizio", viene inteso l'indicazione di un valore ai parametri del servizio. La presenza di una azione all'interno di una obbligazione esprime un impegno a compiere una particolare attività se una data precondizione è soddisfatta. In particolare, l'esecuzione di tale azione può essere delegata a terzi.

Definito dal *Global Grid Forum* (GGF), il linguaggio WS-Agreement è considerato un'evoluzione di WSLA. Il suo obiettivo è fornire uno standard nella costruzione di architetture orientate a scenari *agreement-driven*. I principali punti di forza di questo linguaggio riguardano la specifica di elementi *domain-indipendent* di un semplice processo di contrattazione. Più precisamente, definizioni generiche possono essere potenziate con concetti *domain-specific*.

Il listato 2.2 mostra la struttura ad alto livello di un WS-Agreement espressa per mezzo di un documento XML che contiene oltre le indicazioni descrittive, le informazioni dell'accordo e del contesto al quale si riferisce, e infine la definizione degli elementi facenti parte dell'accordo.

L' elemento *Context* è utilizzato per descrivere le parti coinvolte ed altri aspetti dell'accordo che non rappresentino obblighi tra le parti, così come la data di scadenza. Un accordo può essere definito per uno o più *Context*.
Gli elementi *Term* esprimono nel WS-Agreement consensi ed obblighi di una delle parti. Elementi sepciali come ad esempio gli operatori AND/O-

```
<wsag:Agreement AgreementId="xsd:string">
  <wsag:Name>
    xs:NCName
  </wsag:Name>
  <wsag:AgreementContext>
    wsag:AgreementContextType
  </wsag:AgreementContext>
  <wsag:Terms>
    wsag:TermCompositorType
  </wsag:Terms>
</wsag:Agreement>
```

Listing 2.2: Struttura WSA

R/XOR, possono essere usati per combinare i *term*. I *Term* riguardanti gli obblighi sono organizzati logicamente in due sezioni differenti. La prima sezione, individua i servizi coinvolti per mezzo degli elementi chiamati *Service Description Terms*.

Queste elementi descrivono essenzialmente gli aspetti funzionali del servizio fornito con il relativo accordo. Un *Term* contiene il proprio nome e il nome del servizio al quale si riferisce. Uno speciale tipo di *Service Description Terms* è il *Service Reference*, che punta a una descrizione del servizio distinta dalla descrizione presente nello stesso accordo.

La seconda sezione della definizione di un elemento *Term*, specifica le garanzie misurabili e associate agli altri *term* dell'accordo, che possono essere soddisfatte o violate.

Nel *Guarantee Term* sono contenute le informazioni sulle parti obbligate (il fornitore e il cliente del servizio) e la lista dei servizi coperti da garanzia *(Service Scope)*; sono inoltre presenti un espressione booleana che specifica le condizioni entro le quali si riconosce tale garanzia *(Qualifying Condition)*, ed infine l'asserzione reale che deve essere garantita sul servizio (*Service Level Objective*) con la relativa lista di valori business-related *(Business Value List)* dell'accordo.

In generale, le informazioni contenute nei campi di un *Guarantee Term* sono

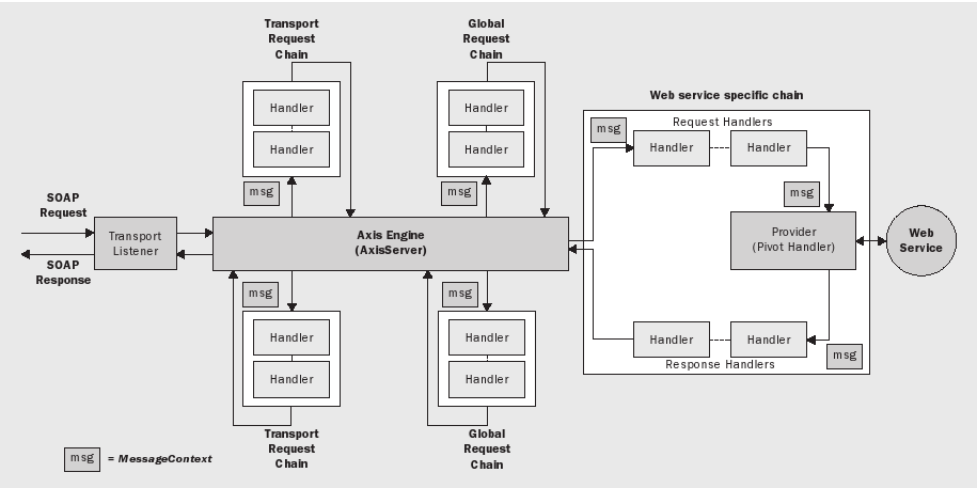

Figura 2.5: Architettura di Axis

espresse per mezzo di specifici linguaggi *domain-specific*.

2.4 Apache Axis

Axis (Apache eXtendable Interction System) è una piattaforma Java, completamente Open Source, per la creazione e l'utilizzo di applicazioni che si basano sui Web Service.

Questa tecnologia è un'infrastruttura di supporto per il deployment dei servizi e l'utilizzo dei diversi protocolli di trasporto utilizzati da SOAP: non è quindi solo un SOAPEngine in grado di processare messaggi e permettere di sviluppare client, server e gateway SOAP.

La figura 2.5 mostra le principali entità che compongono l'infrastruttura di Axis. La struttura portante dell'architettura di Axis è costituita dagli *handlers*, oggetti capaci di processare messaggi SOAP. Essi sono responsabili delle elaborazioni associate al flusso di input, output e di fault del servizio.

Gli *handler* consentono allo sviluppatore di implementare specifiche funzionalità ed estendere il comportamento del framework. In generale, gli handler

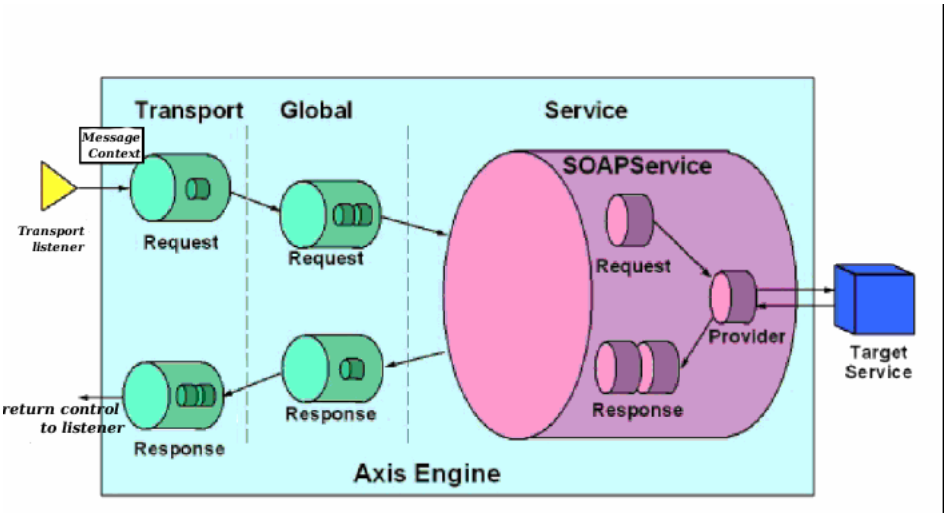

Figura 2.6: Il cammino del Message sul Server.

appartengono a tre diverse famiglie: *transport-specific*, *service-specific* o *global*. Gli handlers di ciascuna famiglia sono a loro volta combinati in Chain suddivise per tipo: *transport*, *global* e *service*. Più precisamente le Chain sono una collezione ordinata di handler che vengono invocati in sequenza.

L'entità centrale dell'architettura è l'*Axis Engine*. L'AxisEngine o semplicemente Engine è l'handler che coordina l'attività di tutte le componenti del framework, richiamando durante le diverse fasi di elaborazione di un messaggio, gli handler o eventualmente le chain specificate. L'elaborazione di un messaggio SOAP consiste nel passaggio di un *context message* ad ogni componente dell'architettura di Axis. Un *context message* è una struttura che contiene un *request message*, un *response message* e una lista di proprietà (attributi dell'header SOAP). Il flusso delle informazioni che attraversa l'infrastruttura di Axis può essere analizzato sotto due diversi profili; il primo considera l'elaborazione dei messaggi quando Axis è invocato come *server*, l'altro quando invocato come *client*.

La figura 2.6 mostra il cammino di un messaggio sul server attraverso le

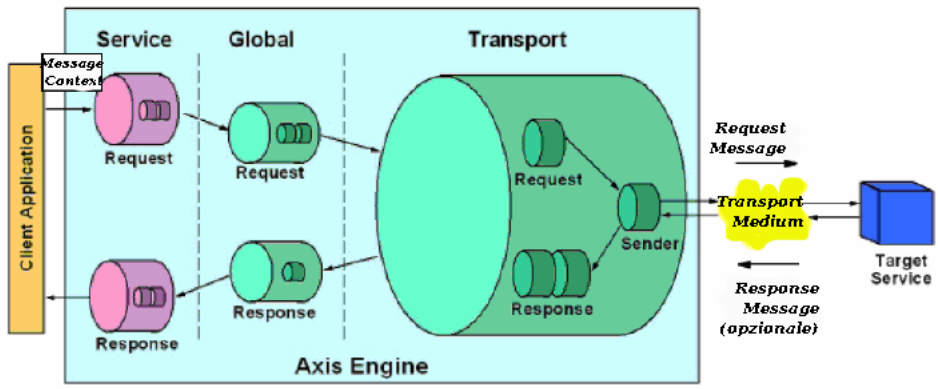

Figura 2.7: Il cammino del Message sul Client.

componenti del framework: in particolare i cilindri piccoli identificano gli handler, mentre quelli grandi, le chain. Il componente *transport listener* converte il messaggio in entrata in un oggetto *Message* e lo incapsula in un oggetto *MessageContext*. Successivamente il *transport listener* carica varie proprietà specificate come attributi SOAP e imposta il tipo di trasporto come proprietà sul *MessageContext*.

Una volta che il *MessagceContext* è stato inizializzato viene trasferito all'*AxisEngine*. Successivamente, l'*AxisEngine*, dopo aver individuato il tipo di trasporto, invoca la *request Chain* del Transport se questa esiste, passandogli il *MessageContext*. Infine, lo stesso *Engine*, localizza una *global request Chain* e invoca gli handlers specificati. Dopodichè alcuni handlers impostano il campo *serviceHandler* del *MessageContext* e lo stesso campo *serviceHandler* indicherà l'handler da invocare per eseguire il servizio richiesto.

La chain Service oltre a contenere un *request* e un *response* hanlder, possiede anche il *Provider* handler, l'entità responsabile dell'implementazione della logica di back end del servizio.

La figura 2.7 mostra invece il cammino del Message sul client.

Nello strato Service non compare il provider poiché il servizio è fornito da un nodo remoto, ma vi sono comunque la *request Chain*e la *response Chain* che processano le *service-specific* della richiesta e della risposta.
Successivamente viene invocato il *Global request Chain* e infine il *Transport*. L'handler *Transport Sender* invia la richiesta SOAP al Server Axis ed ottiene, eventualmente, una risposta.
La risposta viene incapsulata nel campo *responseMessage* del *MessageContext* che viene propagato attraverso le *response Chains* del transport, del global e del service.

Il *deployment* è un'operazione che deve essere effettuata su un Web Service affinché quest'ultimo possa essere utilizzato. Attraverso questa operazione si notifica a Axis la presenza di un nuovo servizio, specificando il suo nome e la sua locazione.
Axis fornisce due diversi meccanismi di deployment caratterizzati da differenti possibilità di configurazione.
L'*Istant Deployment* permette la pubblicazione dinamica di un servizio come Java Web Service (JWS). Tale soluzione può essere utililizzata in fase di debug di un'applicazione o di test della piattaforma in quanto semplice da implementare.
La scelta dell'Istant deployment comporta però, delle forti limitazioni poiché la pubblicazione dinamica di un servizio Web necessità del codice sorgente e non permette di specificare quali metodi sono esposti dal servizio. Essa inoltre non consente di definire un mapping dei tipi SOAP/Java.
Il *Custom Deployment* è al contrario un meccanismo di pubblicazione statico, in grado di superare tutte le limitazione illustrate sopra: permette infatti di specificare gli handler associati al servizio Web e le relative fasi di attivazioni.

Le informazioni di deployment sono espresse attraverso apposti file di configurazione in formato XML, detti *Web Services Deployment Descriptor* (WSDD).

Il listato 2.3 è un esempio di file wsdd. All'interno dell'elemento de-

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
 xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
 <handler name="NomeHandler" type="java:path.myHandler">
 <parameter name="myHandler.parName" value="..."/>
  </handler>
<service name="NomeServizio" provider="java:RPC">
  <parameter name="className" value="it.cnr.isti.NomeClasse" />
  <parameter name="allowedMethods" value="*" />
    <requestFlow>
       <handler type="NomeHandler"/>
    </requestFlow>
    <responseFlow>
       <handler type="NomeHandler"/>
     </responseFlow>
  </service>
</deployment>
```

Listing 2.3: Web Services Deployment Descriptor

ployment, che riporta i namespace a cui si fa riferimento, l'elemento *service* specifica le informazioni relative al servizio. Nel tag service sono presenti due attributi *name* e *provider* che determinano rispettivamente il nome del servizio e lo stile dell'applicazione. All'interno del costrutto service vi sono due elementi parameter: il primo specifica il nome della classe che implementa il servizio ed il relativo package, mentre il secondo riporta i metodi pubblicati.

Il tag handler definisce il nome dell'handler associato al servizio e il path della classe che lo implementa mentre la fase di attivazione viene specificata attraverso i tag *requestFlow* e *responseFlow* .

Capitolo 3

Monitoraggio adattivo di SLA

3.1 Introduzione

Originariamente, i sistemi software venivano realizzati su un'architettura predefinita, monolitica, e centralizzata. Questa limitazione era dovuta sia alla tecnologia disponibile, sia alla necessità di rendere il sistema risultante più facilmente trattabile e controllabile. Le componenti di una grande applicazione erano infatti sotto il controllo di una singola organizzazione, che a sua volta era responsabile, della loro progettazione, del loro sviluppo, della loro verifica e del deployment.

La successiva evoluzione delle architetture software, si è caratterizzata da una crescita costante del grado di dinamismo e decentralizzazione delle proprie componenti. Questo sviluppo è stato determinato dalla crescente esigenza delle applicazioni di evolversi insieme all'ambiente in cui operavano. Il mondo reale richiede infatti che continui e rapidi cambiamenti debbano essere seguiti da reazioni altrettanto veloci.

Le tradizionali strategie di risposta alle richieste di cambiamento, implicavano una riprogettazione e una rimplementazione *off-line* delle applicazioni, non adatta ad un contesto dinamico. Al contrario, le modifiche sarebbero dovute essere dinamiche ed eseguite a run-time. SOA è un esempio (vedi 2.2) di architettura software in cui le componenti ed i loro collegamenti possono cambiare dinamicamente. La ricerca dinamica dei servizi infatti, prevede che il binding possa essere creato e modificato a run-time. Questo alto grado di dinamismo comporta inoltre un forte impatto negativo sul controllo della correttezza del sistema e sul modo di condurre tale verifica. Tradizionalmente, la verifica veniva eseguita a design-time e si basava sulla conoscenza delle componenti del sistema.
Nel caso di SOA, invece, un'applicazione è costituita di componenti che non sono sotto il pieno controllo di una singola organizzazione: il binding tra i servizi, più precisamente, può cambiare in maniera non prevedibile dopo il deployment. La correttezza del sistema non può quindi essere accertata staticamente: essa richiede infatti il monitoraggio e la verifica a *run-time* degli attributi del sistema.

L'esigenza di monitorare a run-time le applicazioni SOA ha ispirato numerosi progetti di ricerca, sia accademici che industriali. Possiamo distinguere tra approcci mirati al monitoraggio delle proprietà funzionali di un singolo servizio o di un servizio composito, ad approcci orientati alla misurazione dei più importanti attributi non funzionali della qualità del servizio. Le diverse tecniche di monitoraggio funzionale proposte, si basano sulle asserzioni (pre-post condition) [4] [10], sull'Aspect-Oriented Programming [24], sulla specifica delle proprietà [22], e infine sull'intercettazione dei messaggi scambiati [11].

Il monitoraggio non funzionale dei servizi Web considera invece i diver-

si aspetti della qualità del servizio (QoS). Tra questi aspetti, i più comuni riguardano le seguenti proprietà (attributi) di un sistema: l'*Availability*, la *Performance*, la *Reliability* e la *Capacity*. L'*Availability* rappresenta la probabilità che il sistema sia attivo ed il servizio sia disponibile: l'availability, in altre parole, misura il rapporto tra il tempo totale in cui il servizio è disponibile e la lunghezza dell'intervallo di tempo osservato. Questa proprietà è una misura correlata all'affidabilità del servizio (reliability) ed può essere misurata attraverso il Time-to-Repair (TTR) (misura che invece, rappresenta il tempo impiegato a riparare il servizio [9]).

La *Reliability* è la capacità del servizio di garantire le proprie funzionalità in un determinato intervallo temporale. Uno dei modi di misurarla e' il numero di fallimenti nell'unità di tempo. La Reliability misura inoltre, l'affidabilità nella consegna dei messaggi trasmessi tra i clienti e i fornitori del servizio. Nel caso dei servizi web, la Reliability è il rapporto tra il numero di messaggi ricevuti correttamente e il totale dei messaggi trasmessi [7] .

La *Performance* valuta, invece, le prestazioni del servizio considerando la velocità con cui una richiesta viene completata. Tale proprietà è misurata attraverso: la banda di elaborazione (throughput) e il tempo di risposta (response time). Il *throughput* misura il numero di richieste servite in un dato intervallo di tempo mentre il response time è il tempo necessario a completare una richiesta. In termini generali, quindi, un servizio efficiente dovrebbe avere un alto throughput e un breve tempo di risposta [18].

La *Capacity* è infine la misura del numero massimo di richieste che il sistema accetta simultaneamente [21]. La *Capacity* può essere misurata nel rispetto degli obblighi di performance pattuiti negli *SLA*(accordi tra clienti e fornitori del servizio). Ad esempio, il numero di richieste supportate simultaneamente deve rispettare il valore del vincolo del tempo di risposta

stabilito nell'SLA.

Questo lavoro presenta un approccio al monitoraggio e alla verifica a run-time degli attributi della qualità del servizio. Nello specifico, è riferito al monitoraggio e alla verifica dei Service Level Agreement (SLA checking).I servizi forniti sono infatti controllati a tempo di esecuzione per verificare la conformità delle loro prestazioni di QoS rispetto agli accordi negoziati. L'SLA checking fornisce informazioni utilizzabili in qualità di prove in caso di potenziali dispute tra le parti obbligate dal contratto. Le informazioni raccolte possono infine essere utilizzate dal fornitore del servizio per predire il verificarsi di violazioni, e per ridistribuire le risorse che contribuiscono a evitare queste violazioni.

Con il termine SLA management ci si riferisce alla definizione, al monitoraggio automatizzato, al miglioramento e all'ottimizzazione di SLA tra Web Service [20]. La ricerca per elevare l'efficienza e la capacità dell'SLA management è molto attiva. Sono state proposte differenti infrastrutture per l'automazione del monitoraggio di SLA dalle più importanti aziende informatiche: un gruppo di ricercatori, appartenente ai laboratori HP, ha sperimentato un motore distribuito che raccoglie dati sia lato client che lato provider e segue un protocollo di misurazione degli scambi per assicurare che i dati raccolti siano aggregati correttamente e tempestivamente [20]. I ricercatori IBM hanno invece sviluppato Cremona [12], un framework che implementa le specifiche WS Agreement e facilita la creazione di accordi tra client e provider.

In generale , il monitoraggio di SLA dovrebbe essere supportato in ambienti eterogenei, come sostenuto negli studi condotti da Morgan e coautori [15]. Gli sforzi compiuti in questo senso, tuttavia, hanno trascurato la possibilità di fornire servizi web in ambienti mobili e pervasivi: la scalabilità del

meccanismo di SLA checking. Le tecniche e i meccanismi di SLA checking citati non sono tuttavia scalabili, specialmente in caso di SLA complessi, applicati su diversi e sempre più grandi gruppi di client, e forniti da dispositivi con limitate capacità di banda e memoria. Il costo della verifica degli SLA diventa, infatti, più oneroso in ambienti con risorse limitate. Gli attuali strumenti standard di SLA checking sono, infatti, appena sufficienti a verificare le prestazioni su un numero ristretto di client; ed il tentativo di far scalare questi approcci su un numero molto elevato di client, con diversi SLA riguardanti più proprietà dei servizi web, può fallire miseramente. La questione si complica ulteriormente quando è richiesto che tra l'istante in cui si verifica una violazione e l'istante nel quale questa viene segnalata intercorre un intervallo di tempo breve [5]. L'overhead derivato dall'attività di checking di SLA comporta una notevole degradazione delle prestazioni: i consumi di risorse server riducono infatti la *capacity* del servizio e possono causare, inoltre, nuove potenziali violazioni.

L'approccio presentato nella sezione 3.3 stabilisce un meccanismo di SLA checking adattivo per ambienti pervasivi mobili [19]. Questo studio ha il duplice obiettivo di rendere scalabile l'SLA checking negli scenari con un numero elevato di client e di SLA differenti, e di mantenere inalterata (o quasi) la capacità di rilevare le violazioni degli SLA. La prossima sezione motiverà l'esigenza di utilizzare un meccanismo di SLA checking adattivo, attraverso un esempio di scenario applicativo.

3.2 Scenario applicativo

L'azienda *Fraud Detection Service* (**FDS**) si pone l'obiettivo di supportare la vendita online attraverso il rilevamento delle frodi nelle transazioni e nei pagamenti online. FDS fornisce servizi che verificano la consistenza e validità

delle informazioni fornite dai potenziali acquirenti.
Dato un nome e un numero di telefono, ad esempio, FDS verifica, insieme alla compagnia telefonica, la reperibilità dell'indirizzo telefonico; dato un nome e un numero di carta di credito, FDS verifica che la carta di credito appartenga alla persona identificata dal nome e che vi siano sufficienti disponibilità finanziarie per l'acquisto; dato un indirizzo IP e il codice di avviamento postale, FDS utilizza un servizio di localizzazione geografica per rilevare inconsistenze sulla veridicità dei dati dei clienti al fine di valutare l'alto rischio della transazione.

Le aziende interessate a questi servizi stabiliscono con FDS un SLA; un'azienda che vende i propri prodotti online, ad esempio, potrebbe richiedere che le richieste delle carta di credito ricevano una risposta entro un tempo limite di 5 secondi, con una media del tempo di risposta di 2 sec e con un downtime (tempo in cui il servizio non è disponibile) inferiore alla soglia dello 0.01%.
Altri clienti potrebbero stabilire accordi più complessi e differenziare le varie tipologie di richiesta. Un cliente, ad esempio, potrebbe accordarsi con FDS affinchè il tempo di risposta della verifica di validità della carta di credito sia inferiore a 4 sec per acquisti non superiori a 50 euro, ed accordarsi per un tempo superiore ma con controlli più approfonditi, per acquisti oltre l'importo massimo di 200 euro.

I potenziali compratori online d'altra parte, potrebbero sottoscrivere servizi di FDS che li tutelino da venditori sospetti attraverso la verifica della loro autenticità e dei loro precedenti movimenti: filtrando così le offerte di vendita di dubbia provenienza. Questi clienti, operando in contesti differenti e con aspettative diverse, potrebbero adattare l'utilizzo di questi servizi alle caratteristiche dei loro dispositivi. Ad esempio, i clienti potrebbero adattare

diversi livelli di verifica alla larghezza di banda di cui dispongono.
Si noti che questi clienti possono spesso ereditare SLA distinti come parte di contratti stabiliti attraverso il loro piano telefonico mobile o il loro programma di prevenzione dalle frodi di carta di credito.

FDS, mentre fornisce i propri servizi, verifica e tiene traccia delle proprie prestazioni. Esistono tre motivazioni che spingono FDS a raccogliere tali informazioni: esigenze legali, prove per la risoluzione di dispute, formulazioni di strategie per la ripartizione dinamica delle risorse. La verifica delle clausole sottoscritte nei Service Level Agreement, tuttavia, può rivelarsi un'attività molto dispendiosa. Le aziende come FDS, hanno solitamente un numero molto elevato di clienti, alcuni dei quali hanno sottoscritto un SLA che permette di compiere migliaia di richieste al giorno. Inoltre, considerato l'andamento di crescità della complessità degli SLA, è probabile che questo problema peggiori e diventi la causa della degradazione delle prestazioni del servizio.

Al fine di ridurre il costo della verifica degli SLA, FDS potrebbe decidere di compiere meno controlli, ma questa scelta comporterebbe il rischio aggiuntivo di perdere preziose informazioni. In alternativa, FDS potrebbe acquisire nuove risorse per gestire il maggior carico determinato dalle attività di verifica. Queste risorse, tuttavia, dovrebbero essere utilizzate per gestire situazioni infrequenti di alto carico di lavoro. In ultima analisi, FDS potrebbe decidere di delegare l'attività di verifica a una terza parte: ma in questo modo avrebbe un minor controllo sulla verifica e un maggior costo risultando meno competitivo.

La prossima sezione individua un approccio alternativo ed efficace a questa sfida. La premessa di questo approccio è di definire un meccanismo di SLA checking più leggero che renda meno dispendiosa l'attività di monito-

ring e minimizzi la perdita di violazioni. Tale meccanismo, detto *Monitoring adattivo di SLA*, adatta dinamicamente (run time) l'attività di SLA checking alle condizioni di funzionamento del servizio, sfruttando a proprio vantaggio la diversità dei comportamenti dei client in scenari generati dalla combinazioni di Web Service e di ambienti pervasivi mobili.

In letteratura, sono presenti diversi lavori di ricerca che convergono verso gli obiettivi del *Monitoraggio adattivo di SLA*. Questi approcci, attraverso un osservazione continua degli attributi di qualità del servizio, definiscono strategie di riconfigurazione del sistema a design-time o a run-time. Menasce [14] ha proposto un controller automatico di QoS che utilizza i tradizionali *performance predictors* per meglio adattare il carico di richieste a design-time. Altri studi più recenti hanno proposto, invece, di rendere il sistema capace di auto adattarsi a run-time: Abrahao e coautori, in riferimento a un modello di costi di Service Level Agreement (in termini di penalità e di ricompense), hanno proposto differenti livelli di operazioni per i ottimizzare i ritorni finanziari [6].

La possibilità di adattare dinamicamente la qualità del servizio, apre la strada ad un modello nuovo di architettura: l'utility computing architecture [16]. L'utility computing architecture, fonisce differenti livelli di servizio tramite un SLA-management automatizzato che regola dinamicamente l'allocazione delle risorse assegnate.

Mentre il soggetto del monitoraggio dei precedenti approcci è stato lo stesso carico di lavoro (workload), che veniva adattato opportunamente all'osservazione e alla verifica degli attributi della qualità del servizio, il *Monitoraggio adattivo di SLA* adatta l'osservazione e la verifica al carico di lavoro rendendo l'SLA-Management più flessibile negli scenari di tipo grid-enviroment [8].

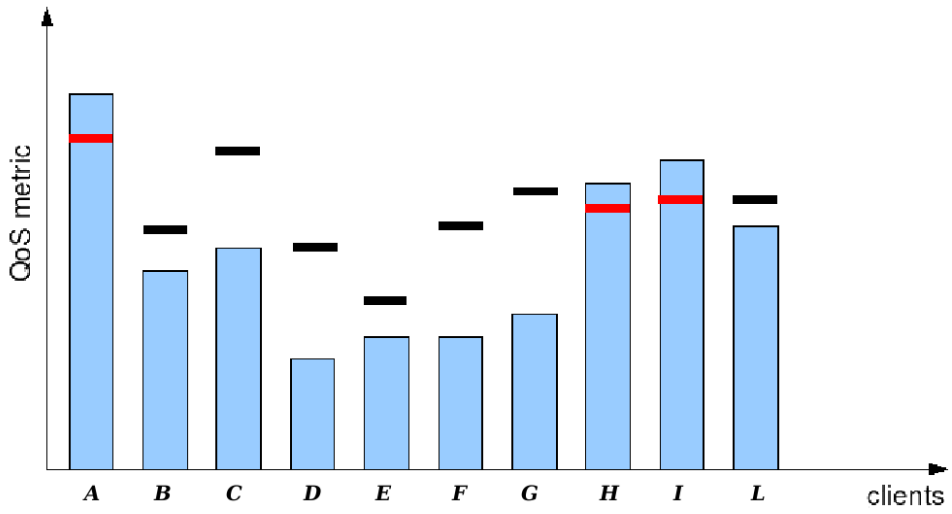

Figura 3.1: Differenti client con differenti distanze dalla violazione

3.3 Descrizione dell'approccio

La combinazione di sistemi software pervasivi con i Web Service dà luogo ad una infinità di potenziali scenari, ognuno con caratteristiche proprie. La diversità interessa differenti aspetti: la tipologia dei servizi, il profilo d'utilizzo dei servizi da parte dei clienti, gli SLA che regolano l'utilizzo di questi servizi e infine le caratteristiche dei dispositivi che ospitano o richiedono i servizi.

Questa diversificazione comporta che la probabilità di violare l' SLA ad ogni istante possa variare tra i clienti in maniera significativa. La figura 3.1 evidenzia il comportamento di alcuni client in uno scenario di questo tipo. Le barre di colore celeste rappesentano la misura delle prestazioni di alcuni client, misurata con una metrica di QoS, mentre i segmenti orizzontali rappresentano i valori di QoS dei vincoli espressi dai rispettivi SLA.

La figura mostra la diversità delle prestazioni tra i client (barre celesti), osservate in uno stesso istante: alcuni client si mantengono infatti all'interno

dei vincoli (segmenti neri), mentre altri superano tali soglie (segmenti rossi). Tali discrepanze sono imputabili sia alla diversità dei vincoli di qualità del servizio degli SLA, sia alla diversità del carico di lavoro di ogni singolo client (il picco di lavoro causato dai clienti Europei di un servizio ad esempio, non coincide con quello dei clienti Americani, ed infine le richieste provenienti da dispositivi mobili sono generalmente più leggere di quelle provenienti da utenti che dispongono di una larghezza di banda maggiore).

L'intuizione alla base della formulazione dell'approccio *Monitoraggio adattivo di SLA* è di incrementare l'efficacia dell'SLA checking attraverso un meccanismo di verifica che adatti dinamicamente se stesso alle condizioni del servizio e al profilo di utilizzo del servizio da parte dei client. Il *Monitoraggio adattivo di SLA* è quindi un meccanismo che *adatta la frequenza di checking del Service Level Agreement alla stima della distanza dalla violazione dei client.*
Come risultato, i client più vicini ad una violazione saranno monitorati con una frequenza maggiore, al fine di accrescere la probabilità di rilevare violazioni, mentre l'attività di verifica sarà meno stringente per i client che ad un dato istante hanno bassa probabilità di violare l'SLA. In questa maniera le risorse utilizzate per il monitoring sono ripartite secondo necessità.

Questo approccio, tuttavia, parte dall'assunto che la diversità tra i potenziali scenari o ambienti comporti una forte variabilità della probabilità dei vari client di violare l'SLA ad un preciso istante. In altri termini, un meccanismo opportunistico che dinamicamente adatti la frequenza di verifica sui singoli client è in grado di rendere disponibili le risorse dove sono più necessarie. La forte differenza tra le prestazioni dei client mostrate in figura 3.1, rappresenta per un approccio adattivo, un'opportunità di fornire miglioramenti spostando le risorse verso i client prossimi a situazioni di

violazione.

La figura 3.3 illustra l'evoluzione dell'attività *SLA-checking* di due client (A e B), attraverso l'utilizzo delle due componenti logiche sopra descritte. Come si vede dalla figura, il comportamento del *meccanismo di campionamento* varia a seconda della distanza dalla violazione: le aree di colore più chiaro rappresentano gli intervalli temporali nei quali la frequenza di campionamento è più bassa mentre le aree di colore scuro rappresentano gli intervalli in cui tale frequenza cresce. L'esempio in figura distingue tre diversi intervalli (aree) con altrettante frequenze di campionamento;
La frequenza dell'analisi aumenta progressivamente nelle aree più scure. Essa passa dall'analisi di un 1 evento ogni quattro monitorati, ad un'analisi completa di tutti gli eventi negli intervalli di tempo in cui il client è prossimo alla violazione. Nella figura si evidenzia la diversità dell'attività di checking negli istanti t_1 e t_2 per i due client: nell'istante t_1 il client A si trova in una situazione di non violazione in cui il servizo è ben al si sotto del vincolo di SLA , mentre il client B si trova in una situazione di violazione. All'istante t_2 la situazione dei due client risulta essere inversa.

Il *Monitoraggio adattivo di SLA* consiste di due componenti logiche fondamentali: il *meccanismo di campionamento* e lo *stimatore della probabilità di violazione.*
La figura 3.2 rappresenta le due unità logiche: il *meccanismo di campionamento* è rappresentato dalla funzione g all'interno dell'ovale. Il meccanismo di campionamento riceve in input un valore s che indica una stima della distanza dalla violazione e restituisce il valore p che, invece, rappresenta l'inverso della frequenza di campionamento, ovvero il periodo.
Il rettangolo celeste rappresenta invece lo *stimatore della probabilità di violazione* e contiene le funzioni h e f; la funzione h calcola la media pesata

dei valori v_i ricevuti in ingresso, con $i = 1..k$ (numero delle precedenti osservazioni).

La funzione h è una aggregazione delle osservazioni del passato: la media pesata è solo una tra le diverse possibilità di rappresentare il *passato*. Il valore ottenuto dalla funzione h , insieme al valore attualmente osservato e al valore del vincolo dell'SLA, costituisce l'input della funzione f.

La funzione f determina, il *peso* da attribuire al valore delle osservazioni del passato rispetto al valore dell'osservazione corrente. Essa rappresenta la stima della *distanza dalla violazione*.

Più formalmente, il meccanismo di campionamento è definito dalla funzione $g : S \rightarrow R$, che mappa i valori di score $s \in S = \{s : s \in [0, 1] \subset \mathbf{R}\}$ nei corrispondenti periodi $p \in P \subset N$.

Ogni valore s è calcolato ad ogni interazione *client-server*(stima attuale della distanza dalla violazione). La stima prodotta dalla funzione g raccoglie in input il valore ottenuto dalla funzione $f : V \times V \rightarrow S$.

I valori $v \in V = \{v : v \in [0, 1] \in \mathbf{R}\}$ rappresentano invece la differenza percentuale tra il valore misurato e il valore del vincolo espresso nel *Service Level Agreement*.

Il valore s è ottenuto combinando i valori di v_c e di $v_h \in V$. v_c rappresenta la distanza dalla violazione corrente, ed è una misurazione puntuale per ogni evento che viene monitorato dal sistema; la misura di v_h esprime la media pesata della distanza dalla violazione, calcolata sulle misurazioni precedenti. Il valore è ottenuto considerando una finestra temporale di dimensione k prefissata.

Viene presentato di seguito una possibile istanziazione dell'approccio proposto precedente. Il listato 3.1 mostra un possibile algoritmo in pseudocodice, che calcola il valore s di score.

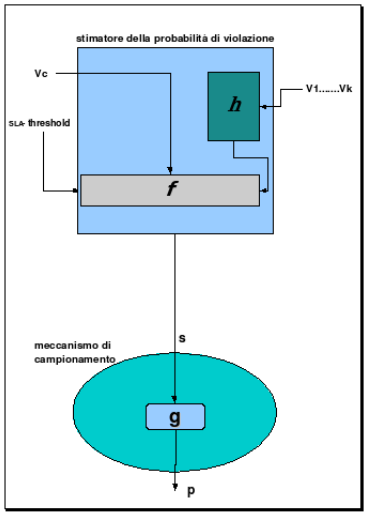

Figura 3.2: Componenti base

```
double violationLikelihood(int currentValue, int slaThreshold, int historicalValue){

        double score;
        int historicalValue = computeViolationTrend;
        if( historicalValue > currentValue )
                score = historicalValue / slaThreshold;
        else
                score = currentValue / slaThreshold;
        return score;

}
```

Listing 3.1: Meccanismo di campionamento

La funzione `violationLikelihood` riceve in ingresso i seguenti parametri: il *currentValue*, l'*slaThreshold* e l'*historicalValue*. Il*currentValue* esprime il valore della distanza dalla violazione corrente misurata con la metrica di QoS, mentre l'*slaThreshold* rappresenta il valore del vincolo di QoS espresso nell'SLA.

L' *historicalValue* esprime, infine, la stima della distanza della violazione calcolata sulle osservazione passate.

La funzione `violationLikelihood` coincide con la funzione f (vedi figura 3.2) , mentre la funzione h (vedi figura 3.2) è rappresentata dalla funzione *computeViolationTrend* .

Attraverso il valore di *score* il sistema adatta la frequenza del monitoring e l'analisi dei messaggi che vengono scambiati tra client e server. Un valore di score basso induce il sistema ad adattare la frequenza di monitoring con un valore molto basso. In questa circostanza solamente una parte di tutti gli eventi e i messaggi scambiati tra client e server verranno analizzati. Il periodo di campionamento e conseguentemente il numero di eventi ignorati dall'analisi varia in accordo alla stima della distanza dalla violazione. Se la qualità del servizio monitorata non rispetta i limiti imposti dall'SLA, il valore di *score* sarà massimo. Un valore di score massimo, rende l'analisi e l'attività di checking continua su tutti gli eventi e i messaggi scambiati da client e server. Questa modalità è detta *full-checking* o *FULL-mode* ed impone un periodo di campionamento pari a uno. Il comportamento del sistema è volto ad analizzare meno messaggi possibili fintantoché la qualità del servizio rispetti i vincoli dell'SLA. Solo in caso di violazione o di situazioni affini il sistema passa alla modalità di *full-checking* o FULL-mode.

Il valore di score in un dato istante è determinato tramite l'analisi dell'evoluzione temporale di una parte delle osservazioni raccolte dal sistema fino a quell'istante. Come già accennato precedentemente il sistema stesso fa riferimento ad una finestra temporale di dimensione k fissata. La seguente formula rappresenta, quindi, una delle possibili implementazioni della funzione h :

$$v_H = \sum_{i=1}^{k} w_i v_i + \max_{j \in 1..k} \{v_j\} w_{max}$$

con:

$$\sum_{i=1}^{k} w_i + w_{max} = 1$$

Ogni elemento v_i in qualità di dato storico, contribuisce nel calcolo di v_H secondo il peso w_i. Attraverso la dimensione della finestra temporale si definisce l'importanza attribuita alle precedenti osservazioni. La dimensione della finestra deve essere scelta tenendo conto dell'andamento temporale di v_c. Tale scelta, è di fondamentale importanza dato che costituisce la rappresentazione del passato.

La scelta di una finestra temporale di piccole dimensioni, infatti, si adatta a scenari in cui il valore di v_C ad ogni iterazione, varia lentamente rispetto ai precedenti valori.

Al contrario, la scelta di una finestra temporale più grande risulta adeguata negli scenari in cui l'evoluzione temporale del valore v_C presenti picchi e quindi, un andamento meno predicibile.

Contrariamente al primo scenario in cui l'evoluzione temporale può essere facilmente rappresentata con l'analisi e il contributo di poche osservazioni, il secondo scenario neccessita di un analisi storica più accurata, basata su un numero di osservazioni più elevato.

Nell'implementazione della funzione h, il parametro w_{max} contribuisce a dare un ulteriore peso al valore massimo osservato all'interno della finestra temporale.

Questo parametro consente di rendere il meccanismo di campionamento ancora più conservativo, offrendo la possibilità di garantire un maggiore controllo negli scenari in cui la variazione tra le osservazioni successive sia molto alta.

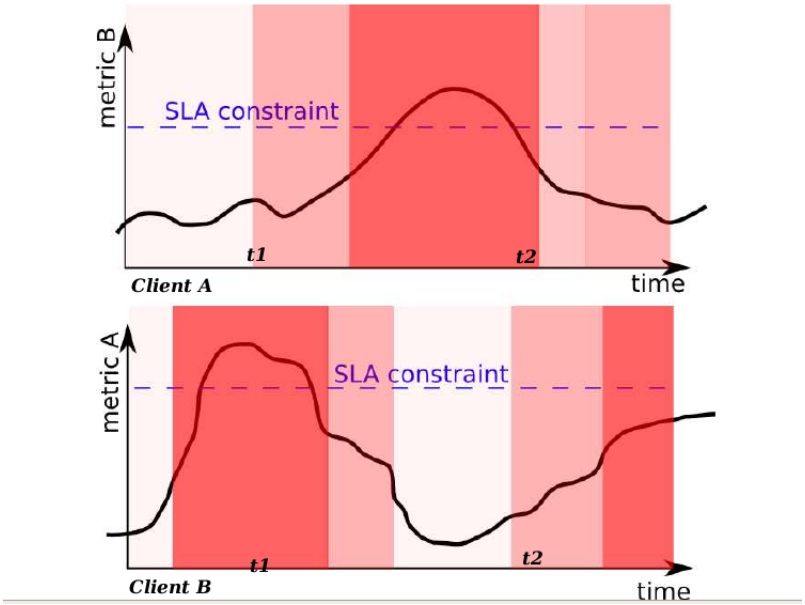

Figura 3.3: Comportamento del monitoraggio adattivo di SLA con due differenti client

L'approccio *Monitoraggio adattivo di SLA*, sopra presentato, fornisce due diversi vantaggi. In primo luogo, riducendo l'attività di checking nelle iterazioni dei client lontani dalla violazione, questo approccio preserva numerose risorse del sistema da un inutile utilizzo. L'obiettivo di questo approccio è la riduzione del numero totale di eventi analizzati, con la conseguenze ripartizione delle risorse tra i client secondo le necessità (come mostrato in figura negli istanti t_1 e t_2).

Il meccanismo di campionamento riduce infatti al minimo il consumo delle risorse nelle iterazioni dove i client sono distanti dalla violazione analizzando un minor numero di eventi per poi aumentare tale numero non appena diminuisce la distanza dalla violazione.

In secondo luogo, come conseguenza immediata del vantaggio della riparti-

zione delle risorse, si ottiene la possibilità che tale meccanismo possa scalare meglio di altri approcci, negli scenari in cui i client esibiscono un differente profilo di utilizzo del servizio nel tempo (vedi figura 3.3).

In ultima analisi, è interessante notare che i benefici suddetti si basano su due condizioni.

La prima condizione riguarda i client che hanno un comportamento eterogeneo tra loro: client diversi hanno differenti distanze dalla violazione in un dato istante (vedi figura 3.1).

La seconda condizione considera invece, la metrica di QoS che si osserva (ad esempio il *tempo di risposta*); essa evolve nel tempo con un'andamento regolare e privo di picchi improvvisi (vedi figura 3.3).

Quest'ultima condizione consente di definire una forma di *"Località temporale"* in cui risulti possibile predire il comportamento futuro dei client in base all'osservazione passata delle loro richieste e risposte.

Capitolo 4

Implementazione

4.1 Architettura su Apache Axis

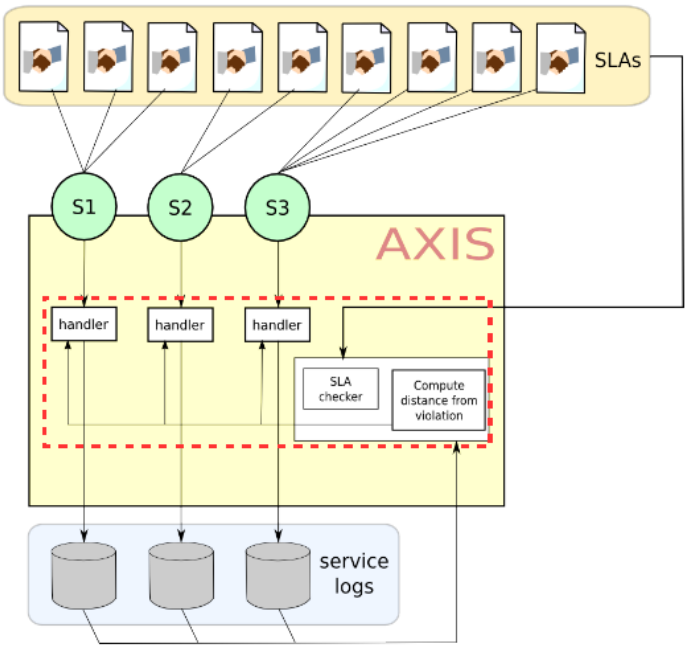

Figura 4.1: Architettura su apache axis

La figura 4.1 fornisce una descrizione dell'architettura astratta dell'approccio *Monitoraggio Adattivo di SLA* (vedi sezione 3.3), sopra l'infrastrut-

tura di Apache Axis.

Nell'architettura del *monitoraggio adattivo di SLA* ogni servizio web monitorato ha associato un handler specifico.

L'handler è attivato dall'AxisEngine (vedi figura 2.5) durante la fase di *request flow* e in quella di *response flow*. La figura 4.2 mostra le due fasi nelle quali la richiesta verso il servizio e la risposta in uscita dal servizio vengono intercettate ed elaborate dall'handler.

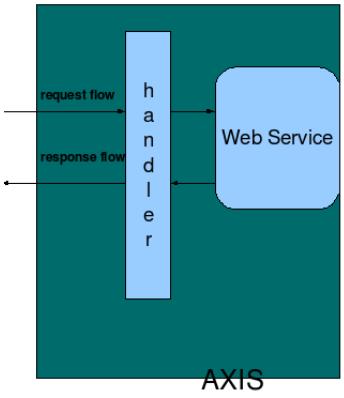

Figura 4.2: Request flow e response flow.

L'architettura prevede, inoltre, che ogni handler si possa connettere a un sistema di log al fine di tenere traccia dell'attività del servizio. Ogni hanlder può quindi essere configurato dinamicamente per consentire l'attivazione o la disattivazione della propria attività di logging.

Subito dopo la rilevazione di uno evento o di una richiesta al servizio, ogni handler consegna le informazione monitorate al sistema di log e parallelamente le inoltra ad un'altro specifico componente denominato *SLA Analyzer*.

L'architettura del sistema (vedi figura 4.1), prevede che l'*SLA Analyzer* (o semplicemente *Analyzer*) appartenga allo stesso *AxisEngine* in cui il Web Service e l'handler associato vengono pubblicati.

L'*Analyzer* è composto da due componenti denominate rispettivamente *SLA Checker* e *Data Collection Controller* (DCC).
L'*SLA Checker* ha il compito di analizzare gli eventi monitorati e catturati dall'handler.
L'attività del *checker* consiste nel confrontare i valori osservati, con i valori dei vincoli espressi nei *Service Level Agreement*. Ad ogni servizio può essere inoltre associato uno o più *Service Level Agreement* come evidenziato in figura 4.1.
Il comportamento dell'Analyzer potrebbe essere esteso prevedendo un insieme di reazioni alle possibili violazioni dei *Service level Agreement*.

Il *Data Collection Controller* è l'entità fondamentale del framework ed implementa la logica del sistema descritta in figura 3.2. Il *DCC* raccoglie le informazioni di output trasferite dall'*SLA checker* tra cui il valore corrente dell'analisi e il valore del vincolo di QoS espresso nell'SLA con la particolare metrica di QoS monitorata.
Attraverso queste informazioni, il *DCC* determina il periodo di campionamento che viene successivamente trasferito all'handler. Il periodo di campionamento verrà poi utilizzato dall'handler per campionare gli eventi osservati. In altri termini, il *DCC* mantiene la giusta corrispondenza tra la *stima della distanza dalla violazione* osservata e il suo *periodo di campionamento*.

La struttura della architettura prevede che la logica applicativa del generico Web Service sotto osservazione sia totalmente indipendente dalla logica del framework.
L'*Analyzer* e il *Data Collection Controller* svolgono infatti, attività completamente slegate da quelle del servizio web poichè i meccanismi di *SLA Checking* e di campionamento sono attività concorrenti alla logica del servizio.

4.2 Estensione di Apache Axis

Questa sezione tratta l'implementazione del sistema considerando le principali componenti dell'architettura e le motivazioni che hanno determinato la scelta di modificare il codice sorgente del framework Apache Axis.

4.2.1 Modifica di Apache Axis

Il funzionamento dell'infrastruttura descritta nella precedente sezione è basata sulla persistenza e la condivisione delle informazioni tra le due principali componenti: l'*Analyzer* e l' *Handler*. Entrambe le componenti, infatti, riferiscono ad ogni interazione del sistema, ad una tabella (hash), contenuta dall'AxisEngine, che associa il *periodo di campionamento* alla relativa classe di QoS per il servizio monitorato (vedi figura 4.12). Nello specifico, l'Analyzer accede in scrittura alle informazioni contenute nella tabella per mezzo del DCC, aggiornando il periodo di campionamento per ogni nuovo valore calcolato dall'SLA Checker. L'handler, invece, accede in lettura alle stesse informazioni aggiornando il valore del proprio periodo di campionamento con il valore letto dalla tabella. Entrambe le componenti accedono a queste informazioni in maniera concorrente. Per l'implementazione di questo meccanismo è stato necessario modificare e ricompilare il codice sorgente dello stesso framework di Axis 1.4. Più precisamente, le modifiche del codice sorgente hanno riguardato l'estensione dell'AxisEngine con l'aggiunta tra i suoi membri di un oggetto della classe java.util.Hashtable che implementa la tabella di corrispondenza suddetta (vedi figura 4.12). L'hashtable è dichiarata statica ed è resa accessibile all'esterno attraverso due metodi, chiamati *getQoSPeriod()* e *setQoSPeriod()*, che rispettivamente leggono e modificano il valore dell'intero period. La scelta di utilizzare la struttura

dati Hashtable è stata determinata dalla caratteristiche di sincronizzazione e di efficienza native di questa classe.

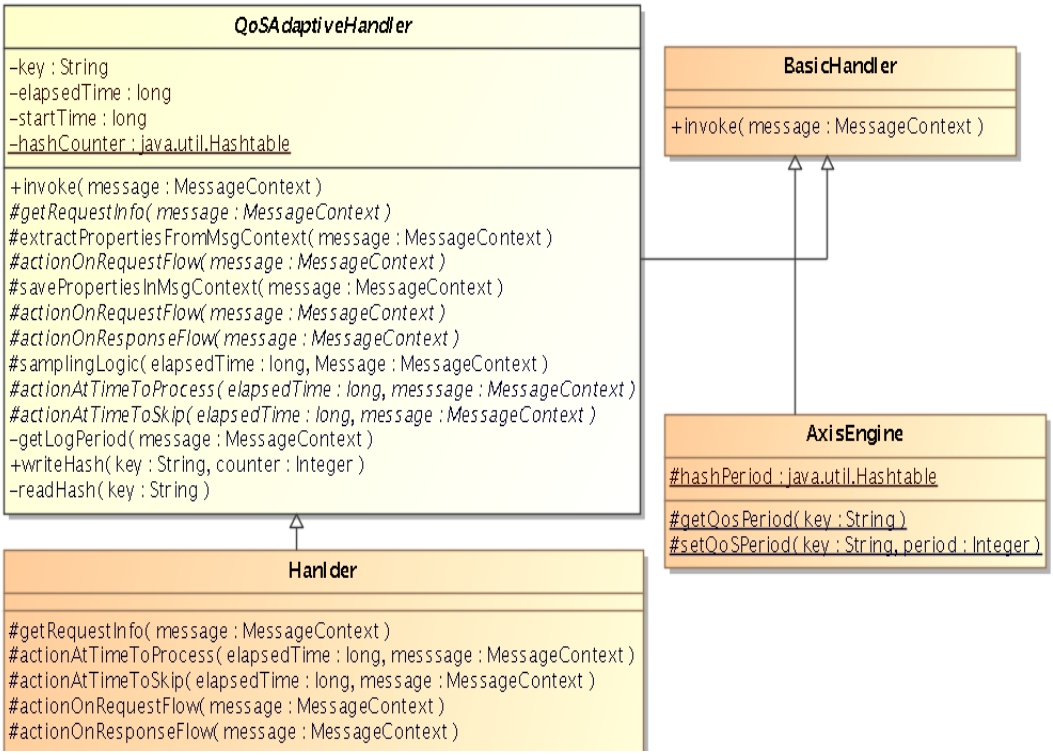

Figura 4.3: Diagramma UML della classe QoSAdaptiveHandler e dell'AxisEngine.

4.2.2 QoSAdaptiveHandler

Gli handler associati ai servizi monitorati ereditano ed estendeno un comportamento comune definito dalla classe astratta *QoSAdaptiveHandler*.

Il *QoSAdaptiveHandler*, a sua volta estende i generici handler offerti dal framework di Axis. Più precisamente, il QoSAdaptiveHandler deriva il comportamento dalla classe padre astratta *org.apache.axis.handlers.BasicHandler* attraverso la ridefinizione del metodo *invoke*. La classe *BasicHandler* implementa a sua volta l'interfaccia *org.apache.axis.Handler* [3]. La figura 4.4

mostra le relazioni che intercorrono tra l'interfaccia e le classi suddette.

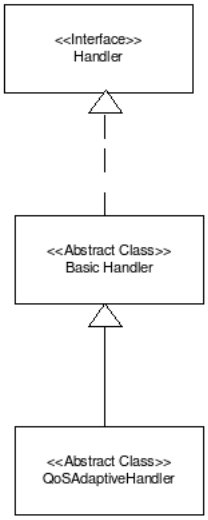

Figura 4.4: Grafico UML della classe QoSAdaptiveHandler

Gestione delle fasi di requestflow e di response flow

Le funzionalità aggiuntive offerte dal *QoSAdaptiveHandler* sono: il *calcolo della qualità del servizio* e l'*implementazione della logica di campionamento*. La figura 4.5 mostra un diagramma delle attività che ogni handler compie ad ogni iterazione del sistema. Tale diagramma illustra, inoltre, la sequenza delle macro operazioni svolte da alcune delle principali componenti del sistema (l'handler, AxisEngine, PivotHanlder, Web Service).

L'AxisEngine attiva, attraverso l'invocazione del metodo *invoke*, l'hanlder associato al servizio monitorato. Il *QoSAdaptiveHandler* accede in questo modo ai messaggi SOAP scambiati tra client e servizio web.
Ogni richiesta verso il servizio web (*request flow*) ed ogni risposta del servizio verso il client (*response flow*) viene infatti intercettata dal *QoSAdap-*

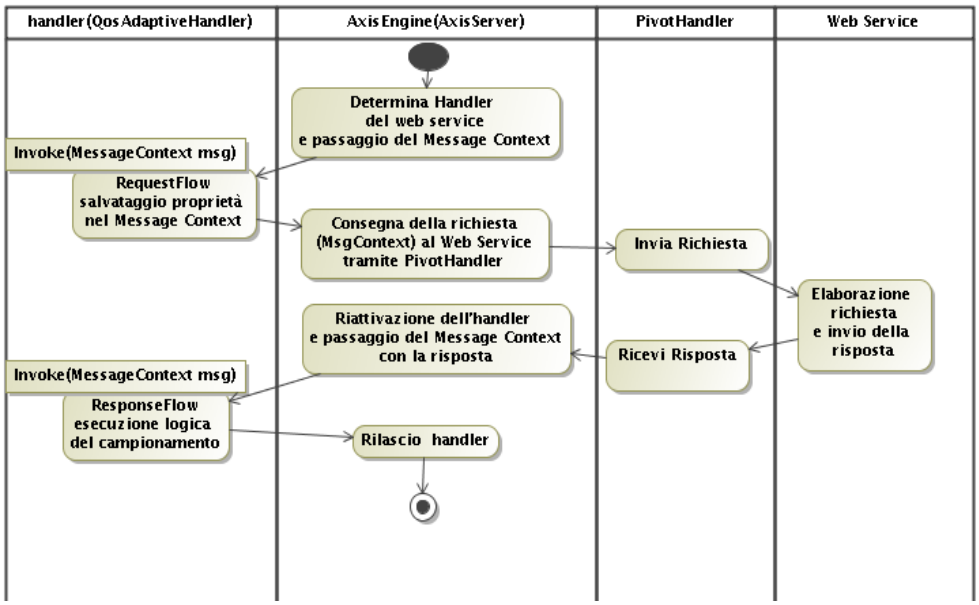

Figura 4.5: Diagramma delle attività handler

tiveHandler. Durante questa fase l'handler inizia la sua elaborazione della richiesta verso il servizio. L'accesso alle informazioni contenute nella richiesta o nella risposta avviene tramite l'oggetto MessageContext, passato dall'AxisEngine come parametro del metodo *invoke* ad ogni sua invocazione (vedi sezione 2.4) [3].

Il *MessageContext* contiene l'oggetto *message* che incapsula un messaggio SOAP e contiene inoltre altre informazioni/attributi quali: il *TargetService*, il *TransportName* (HTTP) e il *PastPivot*.
Il *TargetService* è l'attributo che individua il servizio al quale la richiesta è indirizzata, mentre il *TransportName* indica il protocollo utilizzato per la trasmissione. Il *PastPivot* è infine un particolare attributo che determina il passaggio del messaggio dalla fase di richiesta alla fase di risposta.

L'handler può incapsulare all'interno del *MessageContext*, informazioni calcolate durante la fase di request flow , per mezzo del metodo *setProperty*. Tali attributi potranno essere accessibili e utilizzabili durante la fase di

response flow , attraverso il metodo *getProperty*.
Gli attributi del *MessageContext* forniscono un meccanismo che garantisce la consistenza delle informazioni nel passaggio tra le due diverse fasi di elaborazione.

Terminata la fase di request flow, il flusso di controllo come anche il messaggio contenente la richiesta intercettata dall'hanlder, passa all'AxisEngine che attraverso il PivotHanlder inoltra la richiesta al servizio.

Dopo aver ricevuto la risposta del servizio web, il PivotHandler restituisce il controllo all'AxisEngine che provvede ad riattivare l'hanlder associato al servizio, attraverso la invocazione del metodo *invoke*. In tale fase, detta fase di response flow, l'hanlder estrae dal messageContext le informazioni contenenti la risposta del servizio ed esegue la logica di campionamento dei messaggi intercettati.

Il listato 4.1 mostra una tra le possibili definizioni del metodo *invoke*, il quale implementa e gestisce la logica di controllo della fase di *request flow* ed *response flow*.

```
public void invoke(MessageContext msgContext) {
      if(!msgContext.getPastPivot()){
           startTime = System.currentTimeMillis();
           getRequestInfo(msgContext);
           String key = serviceTarget + qosClass;
           savePropertiesInMsgContext(msgContext);
           actionOnRequestFlow(msgContext);
       }else{
             extractPropertiesFromMsgContext(msgContext);
             actionOnResponseFlow(msgContext);
             elapsedTime = System.currentTimeMillis() - startTime;
             samplingLogic(elapsedTime, msgContext);
         }
}
```

Listing 4.1: Gestione del request e response flow

L'elaborazione del flusso in entrata, è gestita nel blocco di codice della clausola *if*. In tale fase, il message Context ed anche la richiesta non hanno

ancora raggiunto e superato il *Pivot Handler* (vedi sezione 2.4).
Durante questa fase viene calcolato il tempo d'arrivo della richiesta: l'attributo StartTime contiene tale valore misurato in millisecondi. La fase di *response flow* è invece gestita dal blocco di codice della clausola *else*. In questo esempio, viene calcolato il tempo di risposta del servizio, come misura della qualità del servizio. Tale valore (elapsedTime) è ottenuto per differenza temporale tra l'istante (startTime) in cui viene intercettata la richiesta e l'istante in cui la risposta viene intercettata in uscita. La figura 4.6 mostra il percorso del MessageContext durante la fase di *request-response flow* e l'utilizzo dell'attributo *StartTime*.

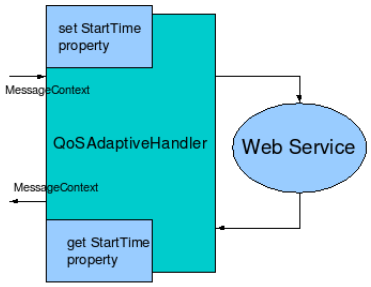

Figura 4.6: MessageContext e relativo percorso.

Il *QoSAdaptiveHandler*, ad ogni iterazione estrae inoltre dal messaggio le informazioni relative alla tipologia della richiesta. Tali informazioni sono la classe di QoS alla quale appartiene il client che ha invocato il servizio ed il nome del servizio invocato determinato dall'attributo TargetService.
Queste informazioni sono contenute all'interno della richiesta SOAP, più precisamente nel corpo (*body*) della busta SOAP, e vengono estratte tramite la funzione *getRequestInfo*. Le informazioni estratte vengono poi inserite nel *MessageContext* durante la fase *request*, con il metodo *savePropertiesInMsgContext* e vengono successivamente estratte in fase di *response* attraverso

l'invocazione del metodo *extractPropertiesFromMsgContext*.

Il listato 4.1 presenta inoltre due metodi astratti *actionOnRequestFlow* e *actionOnResponseFlow*, i quali vengono invocati rispettivamente durante la fase di request e di response.
La definizione di questi metodi offre allo sviluppatore che estende la classe *QoSAdaptiveHandler* un meccanismo per gestire il flusso delle informazioni durante le due fasi di elaborazione. I metodi suddetti, infatti, ricevono come parametro d'ingresso il *MessageContext*, attraverso il quale hanno pieno controllo sui messaggi scambiati tra client e servizio. Una tra le possibili implementazioni di tali metodi potrebbe prevedere il calcolo della banda di elaborazione come metrica della qualità del servizio.

Campionamento dei messaggi

Dopo aver calcolato la qualità del servizio, il *QoSAdaptiveHandler* campiona il traffico di messaggi scambiati tra client e servizio.

Il diagramma rappresentato in figura 4.7, mostra le attività che compongono la logica utilizzata dall'handler per campionare i messaggi scambiati tra client e servizio ad ogni interazione. L'intero *period* definisce il numero di richieste (eventi) che il *QoSAdaptiveHandler* deve scartare prima di trasferire la richiesta alla fase di analisi, mentre il counter è un numero intero che tiene traccia degli eventi scartati fino a quell'istante.
I valore di counter viene incrementato ad ogni iterazione del sistema fino a raggiungere il valore di Period, dopo il quale le richieste non vengono più scartate ma trasferite al sistema di logging, e conseguentemente all'Analyzer. Successivamente, il counter viene azzerato e si itera lo stesso procedimento. Il valore di period è mantenuto consistente, ad ogni iterazione, consideran-

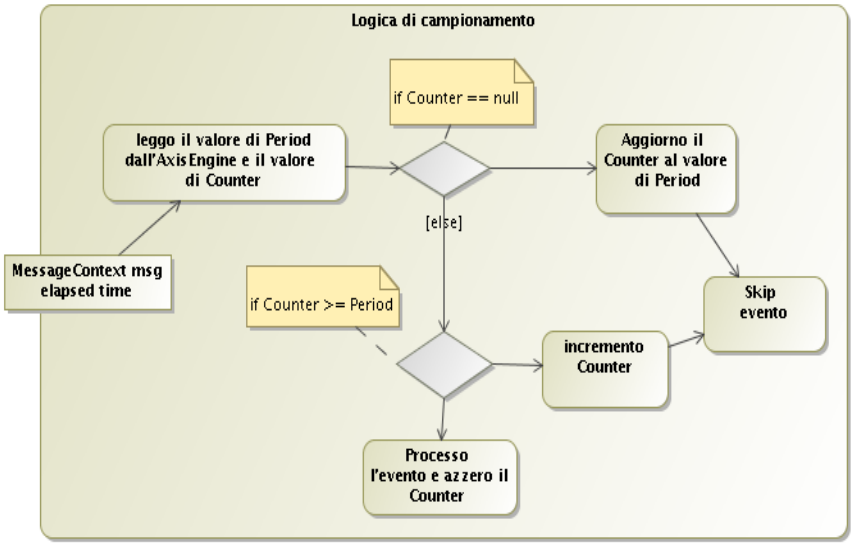

Figura 4.7: Diagramma logica di campionamento dei messaggi

do l'ultimo valore corrente presente nell'AxisEngine e acceduto tramite il MessageContext.

Il *QoSAdaptiveHandler* conserva in una tabella (*hash*), condivisa tra tutte le istanze della classe, le informazioni che gli consentono di implementare la logica di campionamento.
La tabella suddetta conserva una corrispondenza *chiave-valore*, tra il servizio monitorato e il numero di osservazioni (*counter*) che il sistema fino a quell'istante ha analizzato. L'identificativo della classe di QoS congiuntamente al nome del servizio invocato costituisce la chiave per accedere a queste informazioni. La chiave conservata nella variabile key viene costruita durante la fase di request flow (vedi listato 4.1) .

La figura 4.8 analizza una particolare istanza della tabella di corrispondenza, ad un dato istante temporale. La prima tupla indica ad esempio periodo di campionamento (*period*) pari a 5 per le richieste appartenenti alla classe *A* di QoS verso il *Service1*, mentre il contatore (*counter*) degli

Service-QoSClass	Counter	Period
Service1-A	3	5
Service2-C	1	10
Service1-B	9	12
Service3-D	3	3
Service5-E	2	8
Service4-F	5	7

Figura 4.8: Tabella di corrispondenza

eventi non osservati fino a quel momento è pari a 3.

Il listato 4.2 fornisce un possibile esempio di implementazione della logica di campionamento dei messaggi: il metodo *samplingLogic* viene invocato durante la fase di response flow, come mostrato nel listato 4.1.

```
void samplingLogic(long elapsedTime, MessageContext msgContext){
  int period = getLogPeriod(msgContext);
  Integer counter = readHash(key) ;
if(counter == null){
         counter = new Integer(period);
         actionAtTimeToProcess(elapsedTime , msgContext);
  }else{
         if(counter.intValue() >= period){
           actionAtTimeToProcess(elapsedTime,msgContext);
           counter = new Integer(0);
        }else{
            counter = new Integer(counter.intValue()+1);
            actionAtTimeToSkip(elapsedTime,msgContext);
          }
    }
 writeHash(key, counter);
}
```

Listing 4.2: Implementazione della logica di campionamento

I metodi astratti *actionAtTimeToProcess* e *actionAtTimeToSkip* permettono al programmatore che estende la classe *QoSAdaptiveHandler*, di definire il comportamento specifico dell'hanlder.
Il metodo *actionAtTimeToProcess* è invocato ogni volta che la richiesta corrente deve essere processata dal sistema, cioè quando il *counter* raggiunge il valore di *period*. Il metodo *actionAtTimeToSkip* è al contrario invocato in tutte le altre iterazioni in cui la richiesta viene scartata dal sistema.
I parametri di questi due metodi rappresentano rispettivamente il tempo di

risposta per la richiesta corrente (*elapsedTime* ed il *MessageContext*.
La classe *QoSAdaptiveHandler* consente l'osservazione delle attività del servizio ed in particolare offre un meccanismo per consentire l'accesso al flusso di informazioni in tutte le fasi di iterazione del sistema.
Ogni iterazione del sistema, composta dalla fase di request e response, definisce un evento. Il comportamento di un *QoSAdaptiveHandler* identifica esclusivamente la fase di cattura degli eventi, il calcolo della qualità del servizio e l'implementazione della logica di campionamento.

Il *QoSAdaptiveHandler* delega allo sviluppatore che estende tale classe, l'onere di definire le operazioni e le attività successive alla cattura degli eventi e al calcolo del tempo di risposta: non viene infatti previsto nessun meccanismo per la serializzazione o il salvataggio degli eventi e nessuna assunzione implementativa viene fatta per la definizione del sistema di log.
La figura 4.9 illustra un possibile schema di riferimento del sistema di log in cui ogni evento processato è salvato in un file di log in formato testo.
I file di log che compongono il sistema raccolgono e suddividono, per classe

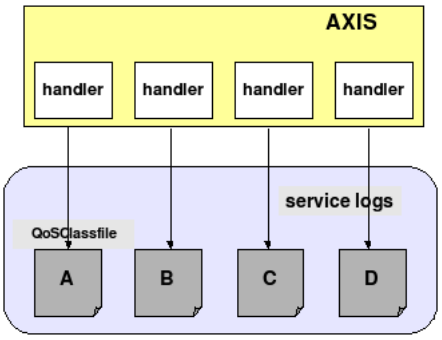

Figura 4.9: Sistema di log

di *QoS*, gli eventi processati. Ad ogni file di log appartengono esclusivamente le richieste provenienti da client con la stessa classe di *QoS*.

Il listato 4.3 è un esempio di salvataggio delle informazioni di una richiesta proveniente da un client di classe *A*, verso il servizio *ServiceUnderTest*, con tempo di risposta di 4509 millisecondi.

```
=======================================================
-> Request TimeStamp: 1179755421326
-> Reply TimeStamp: 1179755425835
-> Response Time: 4509 milliseconds
-> QoSClass: A
-> Service: ServiceUnderTest
=======================================================
```

Listing 4.3: Rappresentazione informazioni su file

4.2.3 QoSAnalyzer

L'*Analyzer* è il componente del framework che svolge l'attività di lettura e d'analisi degli eventi catturati dall'*QoSAdaptiveHandler*.

Il diagramma rappresentato in figura 4.10 mostra le attività che compongono la fase di analisi degli eventi monitorati attraverso le principali componenti del sistema. L'analisi degli eventi, come discusso nella sezione 4.1 riguar-

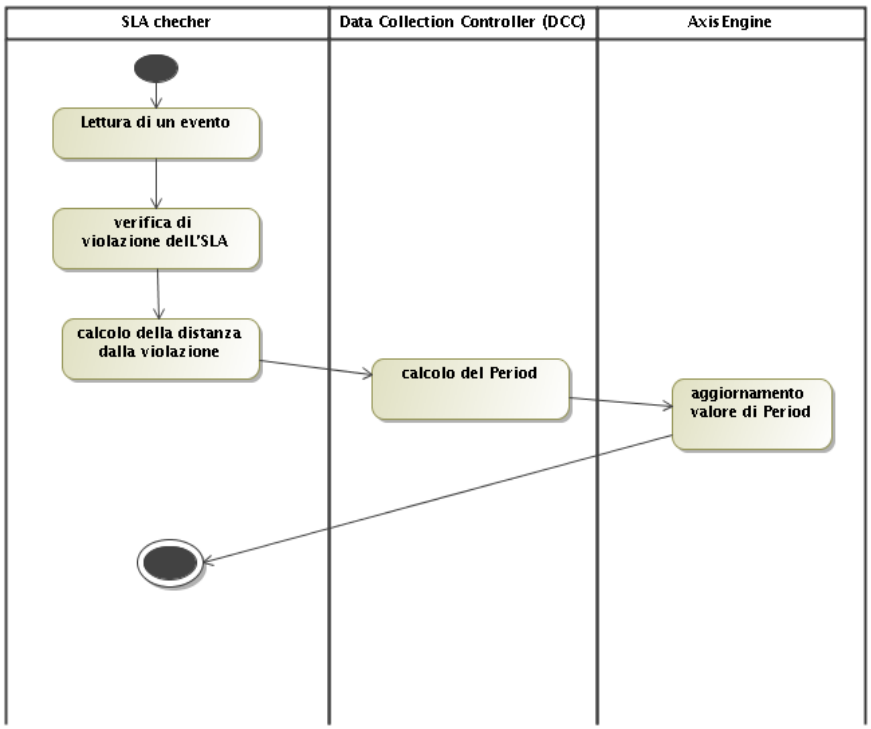

Figura 4.10: Diagramma delle attività dell'Analyzer.

dante l'architettura su Apache Axis, è svolta dal componente *SLAChecker*: dopo aver letto un evento, questo componente verifica se i valori di QoS osservati rispettano i vincoli espressi nell'SLA. Successivamente all'attività di verifica, l'SLAChecker calcola una stima percentuale della distanza dalla violazione osservata. Il DCC calcola, sulla base della stima prodotta dall'SLAChecker, il valore del periodo di campionamento (period) e aggiorna

l'AxisEngine con questa informazione. Dopo l'aggiornamento del periodo di campionamento, l'Analyzer itera lo stesso procedimento.

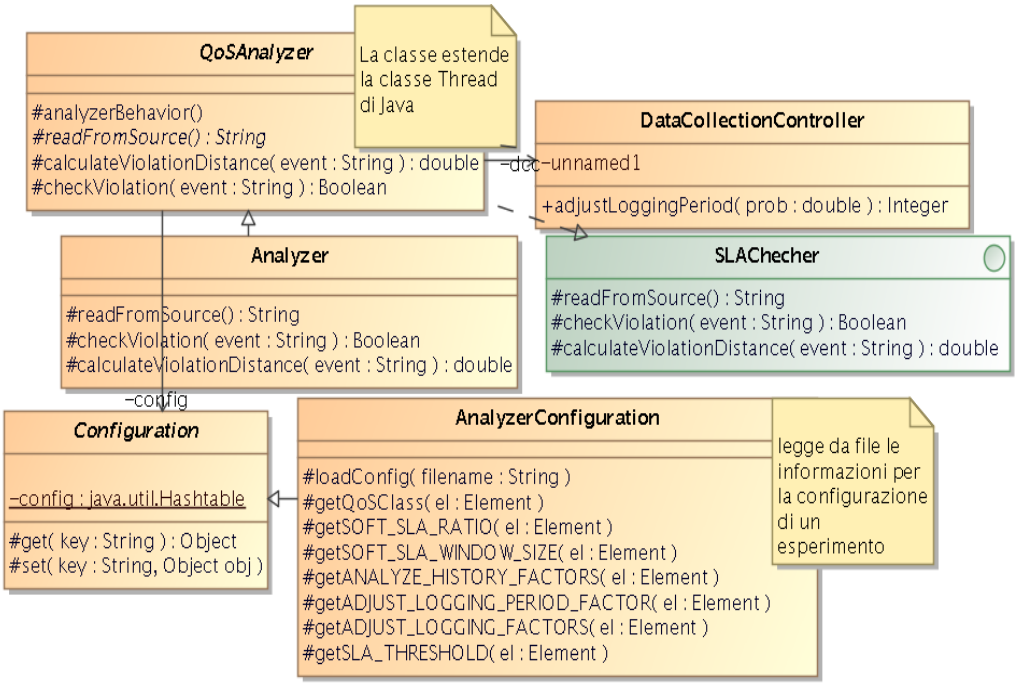

Figura 4.11: Diagramma delle classi Analyzer, QoSAnalyzer.

Il flusso di controllo delle attività sopra descritto è realizzato dal metodo *analyzerBehavior* della classe astratta *QoSAnalyzer* (vedi diagramma delle classi di figura 4.11). Il listato 4.4 mostra una possibile implementazione di tale metodo.

All'interno del ciclo *while* vengono gestite le seguenti operazioni:

- lettura di un evento con il metodo *readFromSource*
- verifica di violazione da parte dell'SLAChecker con il metodo *checkViolation*
- calcolo distanza dalla violazione con il metodo *calculateViolationDistance*

```
public void analyzerBehavior(){
  String event = null;
  double prob;

  while(isRunning()){
       event = SLAChecker.readFromSource();

       if(SLAChecker.checkViolation(event)){
              reactionToViolationChecked(event);
      }

     prob = SLAChecker.calculateViolationDistance(event);
     DCC.adjustLoggingPeriod(prob);
  }
}
```

Listing 4.4: Lettura e analisi degli eventi

- aggiornamento del periodo di campionamento da parte del *Data Collection Controller* con il metodo *adjustLoggingPeriod*

Lettura eventi

Nella classe QoSAnalyzer non viene fatta nessuna assunzione riguardo l'implementazione del sistema di log dal quale vengono estratti gli eventi; una tra le possibili scelte implemetative riguarda un sistema di log composto da file di log organizzati secondo lo schema proposto nella figura 4.9.
Il seguente listato di codice definisce l'operazione di lettura di un evento dal sistema di log.

```
private String readFromSource(){
 String event = null;
  while (event == null){
            event = readEvent();
            Thread.sleep(ANALYZE_SLEEP_PERIOD);
 }
return event;
}
```

Listing 4.5: Lettura di un evento

Il ciclo *while* all'interno del metodo *readFromSource* individua lo schema generale dell'estrazione di un evento dal sistema di log. Esso è composto da

una prima fase di lettura e da una fase di *sleep* dell'esecuzione.
L'operazione di lettura è definita nel metodo astratto *readEvent*. La definizione di tale metodo è compito dello sviluppatore: il quale dovrà tener conto dell'organizzazione e dell'implementazione del sistema di log.

```
protected String readEvent(){
 String qosFileLogName = "AServiceUnderTest.log":
 FileInputStream fis=new FileInputStream(filelog);
 InputStreamReader isr=new InputStreamReader(fis);
 BufferedReader br=new BufferedReader(isr);
 String event = null;
 try{
      event = this.br.readLine();
      if((event != null)&&(!(event.startsWith("->␣ElapsedTime:␣"))))
         event = null;
 }catch(IOException e){
       e.printStackTrace();
  }
 return event;
}
```

Listing 4.6: Lettura di un evento da file

Una possibile implementazione del metodo *readEvent* applicato allo schema di riferimento in figura 4.9 è rappresentata nel listato 4.6. Questa implementazione prevede l'apertura e la lettura sequenziale del file di esempio "AServiceUnderTest.log".

Verifica della violazione

La verifica della violazione avviene tramite il metodo *checkViolation* dichiarata nell'interfaccia SLAChecker. Questo metodo delega allo sviluppatore che lo implementa la gestione delle violazioni. Il metodo astratto *reactionToViolationChecked*, lascia allo sviluppatore il compito di definire il comportamento dell'Analyzer in caso di violazione dell'SLA.
Una tra le possibili implementazioni di questo metodo potrebbe prevedere la memorizzazione delle violazioni riscontrate durante la fase di analisi su

file suddivisi per classe di QoS.

Calcolo della distanza dalla violazione

La figura 3.2 è lo schema di riferimento per il metodo *calculateViolationDistance* dichiarato nell'interfaccia SLAChecker. Questo metodo coincide con l'utilizzo delle funzioni f e h e permette di stimare in termini probabilistici, quanto la *qualità del servizio* dell'evento analizzato sia prossimo al valore del vincolo di QoS espresso nel *Service Level Agreement* (vedi sezione 3.3).
Il metodo *adjustLoggingPeriod* coincide, invece, con la funzione g (vedi sezione 3.3). Ricevuto come parametro di ingresso il valore di *prob*, associa a questo numero un corrispondente periodo di campionamento.
Il valore di period è quindi trasferito dall'oggetto *Data Collection Controller* all'*Axis Engine* che associa questa informazione al servizio e alla classe di QoS dell'evento analizzato. Questa associazione o corrispondenza è implementata attraverso una tabella hash contenuta all'interno dell'AxisEngine. La figura 4.12 illustra una possibile implementazione di questa corrispondenza.

Distanza dalla violazione	periodo
15,00%	10
30,00%	7
50,00%	5
75,00%	2
100,00%	0

Figura 4.12: Tabella di corrispondenza distanza dalla violazione e periodo di campionamento

Capitolo 5

Caso di studio

In questo capitolo verrà presentato un particolare caso di studio condotto sull'architettura presentata nei precedenti capitoli, congiuntamente all'analisi e alla discussione dei risultati ottenuti.

5.1 Descrizione

Il caso di studio presentato fa riferimento allo scenario descritto nella sezione 3.2. Esso considera due differenti tipi di clausole di SLA: entrambe esprimono i vincoli del tempo di risposta del servizio.
La prima tipologia di clausole è costituita da vincoli di tipo "hard constraint": la verifica di queste clausole prevede il confronto tra il valore istantaneo dell'osservazione e il valore del vincolo della relativa metrica di QoS. Un esempio di "hard constraint" potrebbe essere così espresso: "Il tempo di risposta del servizio deve mantenersi al di sotto della soglia di **R** millisecondi".
Al contrario, la seconda tipologia di clausole considera vincoli di tipo "soft constraint": tali clausole consentono di violare, occasionalmente, il vincolo del tempo di risposta.

Un esempio di questo tipo di vincolo potrebbe essere espresso come segue: "Il tempo di risposta del servizio non deve eccedere per più di Y volte, gli X millisecondi nelle ultime Z unità di tempo". L'utilizzo di diverse tipologie di vincoli esprime la complessità e la diversità dei possibili SLA.

Ogni client definisce con il suo SLA la soglia del vincolo del tempo di risposta per entrambe le tipologie di vincoli. Le clausole del contratto includono inoltre, la dimensione della finestra temporale all'interno della quale i vincoli di tipo "soft constraint" devono essere verificati ed infine il numero di possibili violazioni ammesse all'interno di tale intervallo di tempo.

La figura 5.1 mostra il grafico del workload utilizzato per condurre gli esperimenti.
Il workload è caratterizzato: da un tasso di interarrivo costante e da una domanda di servizio che segue un preciso schema periodico. Il carico di lavoro al quale il servizio è sottoposto può essere controllato per mezzo di tre parametri fondamentali: la durata del *busy/quite period*, la fase del *busy period* ed infine il *peak demand*.
Si consideri un periodo di workload singolo. Come mostrato in figura 5.1 tale periodo è composto di due fasi, il *busy period* e il *quite period*.

Durante il proprio *busy period*, il client accresce il proprio carico di richiesta di servizio. La richiesta di servizio è definita in termini di occupazione di CPU e di latenza relativa all'attività di rete. Entrambe i parametri sono regolabili dai client ad ogni loro richiesta. La crescita del workload del client determina un maggiore consumo di risorse server utilizzate dal servizio web per gestire le richieste. La fase di *busy period* è quindi caratterizzata da una probabilità di violazione dei SLA molto elevata.
Il monitoraggio adattivo di SLA fa si che il sistema adatti la propria attività di monitoring al cambiamento del carico di lavoro del servizio. Durante

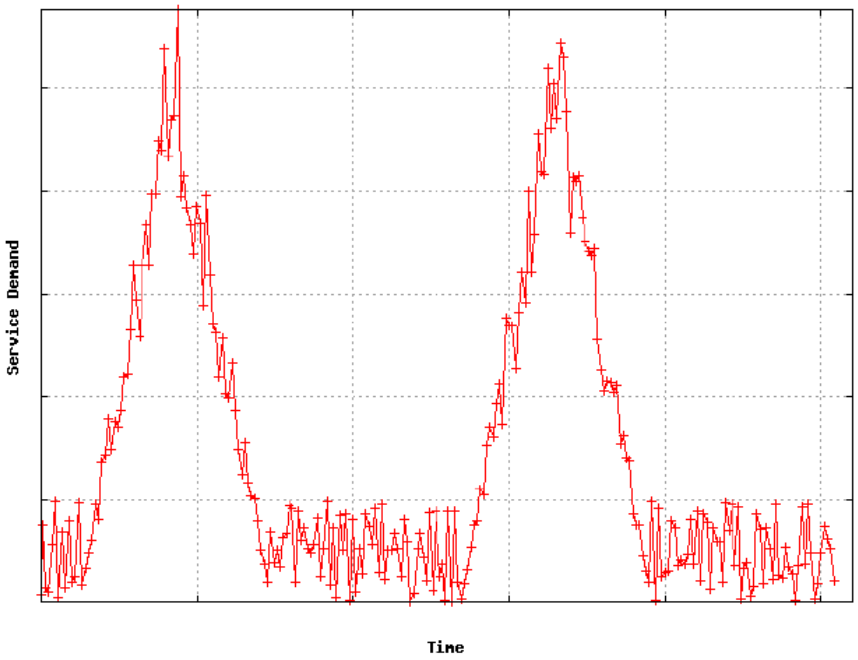

Figura 5.1: Workload d'esempio.

tale fase il sistema passa ad una modalità di tipo FULL-mode che attiva il logging (osservazione) e l'attività dell'Analyzer ad ogni interazioni client-server. Il time-zero del *busy period* è definito come l'istante temporale di partenza di tale periodo. Il *peak value* determina invece, il valore massimo che la richiesta di servizio può raggiungere, per una determinata classe di QoS.

Il *quite period* è definito come la differenza tra il periodo di workload e il *busy period*. Durante questa fase, la richiesta di servizio dei client non è causa diretta di violazione di SLA. In questo caso, il *monitoraggio adattivo* filtra l'input verso il sistema di monitoraggio; le richieste la cui probabilità di violare i vincoli di QoS è ritenuta improbabile, saltano sia la fase di logging e sia la fase di analisi.

Le fasi di *busy period* e di *quite period* caratterizzano la periodicità del

workload e determinano un profilo d'utilizzo del servizio da parte del client prevedibile e facilmente approssimabile. La misurazione della metrica di QoS espressa nell'SLA, infatti, evolve nel tempo con un'andamento regolare e privo di picchi improvvisivi. Questa caratteristica del workload stabilisce una forma di "Località temporale" che permette di predire il comportamento futuro del client.

In ultima analisi, gli esperimenti condotti utilizzano client tra loro concorrenti e caratterizzati da classi di SLA differenti. Inoltre, i workload di client appartenenti a diverse classi di SLA risultano non essere in fase: la figura 3.3 mostra il workload di due client appartenenti a due classi di SLA distinte: la fase di *busy period* del client A coincide con la fase di *quite period* del client B.

Un meccanismo che adatta la *frequenza di checking*, se applicato al workload suddetto, ripartisce le risorse del sistema tra i client che hanno workload distinti e non in fase. Esso riduce, infatti, il numero di richieste analizzate per i client nelle fasi di *quite period*, mentre intensifica i controlli sulle richieste dei client nelle fasi di *busy period*.

5.2 Esperimenti: analisi dei risultati

Le prestazioni che caratterizzano il monitoraggio adattivo sono influenzate dallo scenario applicativo nel quale si utilizza (vedi sezione 3.3). Più precisamente, questo approccio è sicuramente condizionato da numerosi fattori: la diversità e la complessità della verifica degli SLA, il diverso profilo di utilizzo del servizio da parte dei client ed infine la natura stessa del servizio. Esistono tipologie di scenari e condizioni in cui questo approccio non apporti vantaggi, ma sia persino inopportuno: gli esperimenti condotti sono un iniziale lavoro di ricerca per esaminare quale siano le condizioni migliori per un meccanismo di verifica adattivo.

Gli esperimenti condotti avevano l'obiettivo di verificare e quantificare i vantaggi apportati dall'utilizzo di un meccanismo di monitoraggio adattivo rispetto ai meccanismi di monitoraggio non adattivi. La verifica considerava due principali aspetti: il risparmio di risorse disco utilizzate ed il risparmio di occupazione di CPU. Il workload utilizzato per condurre gli esperimenti è quello descritto nella precedente sezione.

Un primo esperimento aveva l'obiettivo di confrontare il comportamento del monitoraggio adattivo rispetto al monitoraggio FULL-mode (non adattivo). Il monitoraggio FULL-mode è basato sull'osservazione e sull'analisi di tutte le interazioni del sistema: il sistema non filtra le richieste la cui probabilità di violazione risulta essere molto bassa. La modalità FULL-mode garantisce, quindi, la massima efficacia nella rilevazione delle violazioni. Il listato 5.1 mostra i risultati ottenuti per mezzo di un approccio adattivo. Il numero di eventi generati dall'esperimento (*generated events*) è di 5398, mentre il numero di eventi analizzati dal sistema (*analyzed events*) è di 1935. Il numero di violazioni generate sui vincoli di tipo hard (*generated hard vio-*

lations) e di tipo soft (*generated soft violations*) sono rispettivamente di 701 e 465.

Tra i risultati sono, inoltre, riportati la misura dell'efficacia sui i vincoli di tipo hard (*effectiveness on hard constraints*) e sui vincoli di tipo soft (*effectiveness on soft constraints*). L'efficacia è la misura del rapporto tra il numero di violazioni rilevate e il numero di violazioni totali: $\frac{\#sampledViolation}{\#realviolation}$.

L'analisi dei risultati ha dimostrato come l'*approccio adattivo* segua l'efficacia del monitoraggio FULL-mode, su entrambe le tipologie di vincoli: il valore dell'efficacia ottenuta sui vincoli hard è di 0.87 mentre l'efficacia sui vincoli soft è di 0.93.

```
//*******************************************************************************

FINAL REPORT:

#ANALYZED EVENTS: 1935   // -->EFFECTIVENESS ON HARD CONSTRAINT: 0.87731814
#DETECTED VIOLATIONS:615      //         (# sampled Violations / #realViolations)

#DETECTED SOFT VIOLATIONS: 437 // --> EFFECTIVENESS ON SOFT CONSTRAINT:0.93978494
#GENERATED EVENTS: 5398            // (# sampled Violations / #realViolations)

#GENERATED HARD VIOLATIONS: 701 // -->EFFICIENCY:0.6415339
#GENERATED SOFT VIOLATIONS:465 //       ( 1 - ( #sampledLogEntries / # logEntries) )


//*******************************************************************************
```

Listing 5.1: Monitoraggio adattivo

Il listato 5.1 riporta infine il valore dell'efficenza calcolata come:

$1 - \frac{\#sampledLogEntries}{\#logEntries}$ dove *sampledLogEntries* è il numero di scritture su file di log campionate dal sistema, mentre *logEntries* è il numero totale di scritture su file generate. Il *monitoraggio adattivo* preserva quasi la metà del numero di scritture sui file di log che si avrebbero utilizzando un monitoraggio FULL-mode.

Questo notevole risparmio di risorse disco garantisce un'alleggerimento del

carico di lavoro da parte dell'SLA Analyzer.

Un'altro esperimento ha confrontato il monitoraggio adattivo con un approccio basato su una politica di campionamento di tipo random. L'esperimento è stato condotto sotto le medesime condizioni di carico al fine di rendere gli esperimenti confrontabili: stesso numero di violazioni soft e di violazioni hard generate. Il listato 5.2 mostra i risultati dell'esperimento adottando un meccanismo di campionamento di tipo random. Con lo stesso tasso d'efficenza in termini di numero di eventi analizzati, l'approccio adattivo rileva più del 40% di violazioni rispetto all'approccio random.

```
//*****************************************************************************

FINAL REPORT :

 # ANALYZED EVENTS: 2182        // --> EFFECTIVENESS ON HARD CONSTRAINT : 0.50927246
 # DETECTED VIOLATIONS : 357    //       (# sampledViolation / # realViolation)

 # DETECTED SOFT VIOLATIONS : 248 // --> EFFECTIVENESS ON SOFT CONSTRAINT : 0.53333336
 # GENERATED EVENTS: 5398       //       (# sampledViolation / # realViolation)

 # GENERATED HARD VIOLATIONS : 701 // --> EFFICIENCY : 0.5957762
 # GENERATED SOFT VIOLATIONS : 465 //    (1 - (# sampledLogEntries / # logEntries))


//*****************************************************************************
```

Listing 5.2: Monitoraggio con meccanismo di campionamento random

Se si considera il guadagno in termini di risparmio d'utilizzo di CPU, il monitoraggio adattivo apporta un lieve miglioramento rispetto al monitoraggio FULL-mode.
Intuitivamente, poichè il *monitoraggio adattivo di SLA* è un approccio in grado di limitare il numero totale di controlli effettuati, esso si comporta meglio dell'approccio FULL-mode quando la verifica è fatta su SLA complessi. La tabella 5.1 riassume i risultati ottenuti confrontando l'utilizzo dei due meccanismi: il risparmio di CPU apportato cresce all'aumentare della

complessità del meccanismo di SLA checking. La complessità del meccanismo di SLA checking è espressa nei termini del numero di clausole di SLA verificate.

Number of SLA clauses	1	10	20	50	100
CPU gain	2.66%	3.25%	3.24%	3.22%	4.58%

Tabella 5.1: Risultati esperimenti CPU gain

I risultati ottenuti, possono essere migliorati ulteriormente attraverso una più efficace implementazione dell'approccio, con l'obiettivo di minimizzare l'overhead introdotto dall'infrastruttura. Inoltre, si può agire sul carico computazionale del meccanismo di SLA checking attraverso la costruzione di SLA ancora più complessi.

Capitolo 6

Conclusioni

Questa tesi elabora e propone un approccio adattivo di SLA checking per ambienti pervasivi mobili, denominato *monitoraggio adattivo di SLA*.
Il *monitoraggio adattivo di SLA* ha il duplice obiettivo di rendere scalabile l'SLA checking negli scenari con un numero elevato di client e di SLA differenti, e di mantenere inalterata (o quasi), la capacità di rilevare le violazioni degli SLA. Tale approccio adatta dinamicamente l'attività di SLA checking alle condizioni di funzionamento del servizio, sfruttando a proprio vantaggio la diversità dei comportamenti dei client in scenari generati dalla combinazioni di Web Service e di ambienti pervasivi mobili.

Dopo una breve introduzione sui principi architetturali che caratterizzano le applicazioni orientate ai servizi, il capitolo 2 presenta la tecnologia Web Service ed i relativi standard tecnologici di riferimento: WSDL, SOAP, UDDI.
Inoltre, vengono trattati gli standard tecnologici che caratterizzano le architetture *agreement driven*. Più precisamente, sono presentati i linguaggi WSA e WSLA per la definizione degli aspetti extra funzionali dei servizi web come ad esempio la qualità del servizio.

Il capitolo 3 tratta gli aspetti generali del monitoraggio a run time nelle architetture SOA e motiva, attraverso un esempio di scenario applicativo, l'esigenza di utilizzare meccanismi *dinamici* per l'osservazione e il controllo degli attributi di qualità dei servizi forniti.Nello stesso capitolo sono forniti alcuni esempi di approcci mirati al monitoraggio delle proprietà funzionali di un servizio ed altri esempi di approcci orientati alla misurazione dei più importanti attributi non funzionali della qualità del servizio. Il capitolo 3 individua, successivamente, le motivazioni che rendono non praticabile l'utilizzo di approcci di SLA checking standard negli ambienti pervasivi mobili: l'ampia gamma di esigenze e di aspettative dei client e la diversità degli scenari applicativi tra i client e la rete. Le tecniche e i meccanismi di SLA checking standard non sono infatti scalabili, specialmente in caso di SLA complessi, applicati su diversi e sempre più grandi gruppi di client, e forniti da dispositivi con limitate capacità di banda e memoria. Tali tecniche trascurano l'overhead derivato dalla verifica degli SLA che è causa di degradazione delle prestazioni: i consumi di risorse server riducono, infatti, la capacità del servizio e possono causare, inoltre, nuove potenziali violazioni.

In questo capitolo viene presentata l'intuizione che sta alla base della formulazione del monitoraggio adattivo di SLA e viene fornita una prima istanziazione dell'approccio attraverso la definizione di un meccanismo che adatta la frequenza di checking alla stima della distanza dalla violazione dei client. Infine, vengono illustrati i vantaggi apportati da un meccansimo adattivo di questo tipo. In primo luogo, la riduzione dell'attività di checking nelle interazioni dei client lontani dalla violazione determina il risparmio di numerose risorse del sistema da un inutile utilizzo. In secondo luogo, come conseguenza immediata del vantaggio della ripartizione delle risorse, si ottiene la possibilità che tale meccanismo possa scalare meglio di altri

approcci, negli scenari in cui i client esibiscono un differente profilo di utilizzo del servizio nel tempo.

Il capitolo 4 presenta, invece, gli aspetti tecnologici riguardanti l'implementazione del meccansimo di SLA checking sull'architettura concreta del framework di Apache Axis. Nello stesso capitolo vengono fornite le motivazioni che hanno portato alla modifica del codice sorgente del framework.

Il capitolo 5 presenta, infine, un particolare caso di studio condotto sull'architettura presentata nei precedenti capitoli, congiuntamente all'analisi e alla discussione dei risultati ottenuti. Nello stesso capitolo si motiva la scelta di un workload con determinate caratteristiche: periodicità, andamento regolare e facilmente approssimabile, tasso di interarrivo costante ed infine workload non in fase per i client appartenti a differenti classi di SLA. Infine, vengono mostrati i miglioramenti apportati dall'utilizzo di un meccanismo adattivo rispetto ai meccanismi non adattivi. Tali vantaggi sono valutati in termini di risparmio di risorse disco e di occupazione di CPU.

Concludendo, questo studio ha fornito un approccio alternativo alle tecniche standard di SLA checking, esaminando gli scenari e le condizioni di workload migliori per un meccanismo di monitoraggio adattivo. Il monitorggio adattivo di SLA sfrutta la diversità che caratterizza gli ambienti pervasivi al fine di fornire un meccanismo di verifica con maggiori possibilità di scalare rispetto alle tecniche standard. Questo lavoro fornisce un'infrastruttura software, attraverso l'estensione del framework Apache Axis, che rende istanziabile differenti meccanismi adattivi di verifica: una sua prima istanziazione adatta la frequenza di logging e di checking dell'SLA alla stima della distanza della violazione del client.

La migrazione del framework realizzato verso l'ultima versione di Apache

Axis 2.0 insieme all'integrazione dello stesso all'interno di una infrastruttura di SLA management esistente rappresentano i possibili sviluppi futuri di questo lavoro.

Bibliografia

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services–Concepts, Architectures and Applications*. Springer–Verlag, 2004.

[2] W3C Web Services Architecture. W3c. http://www.w3.org/TR/ws-arch/.

[3] Apache Axis. Axis architecture. http://ws.apache.org/axis/java/architecture/guide.html#ArchitecturalOverview.

[4] L. Baresi, C. Ghezzi, and S. Guinea. Asmart monitors for composed services. In *Int. Conf. on Service-Oriented Computing (ICSOC-04)*, pages 193–202. ACM Press, Nov. 2004.

[5] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web services on demand: Wsla-driven automated management. *IBM System Journal*, 43(1):136–158, 2004.

[6] B.Abrahao V.Almeida J.Almeida A.Zhang D.Beyer and F.Safai. Self-adaptive sla-driven capacity management for internet services. In *In Proc. of Network Operations and Management Symposium (NOMS 2006)*, pages 557–568, 2006.

[7] IEEE. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY, 1990.

[8] C.Dumitrescu I.Foster and I.Raicu. A scalability and performance evaluation of a usage sla.based broker in large grid enviroment. In *Technical report, iVDGL/GriPhyN TechReport*, 2005.

[9] GUNTHER N. J. *The Practical Performance Analyst.* Mcgraw-hill edition, 1998.

[10] A. Lazovik, M. Aiello, and M. Papazoglou. Associating assertions with business processes and monitoring their execution. In *Int. Conf. on Service-Oriented Computing (ICSOC-04)*, pages 94–104. ACM Press, Nov. 2004.

[11] Z. Li, Y. Jin, and J. Han. A runtime monitoring and validation framework for web service interactions. In *Australian Software Engineering Conference (ASWEC'06)*, pages 70–79. IEEE Computer Society, 2006.

[12] Heiko Ludwig, Asit Dan, and Robert Kearney. Cremona: An architecture and library for creation and monitoring of WS-agreents. In *Service-Oriented Computing - ICSOC 2004, Second International Conference, New York, NY, USA, November 15-19, 2004, Proceedings*, pages 65–74. ACM, 2004.

[13] Mark Endrei Jenny Ang Ali Arsanjani Sook Chua Philippe Comte Pal Krogdahl Min Luo Tony Newlin. *Patterns: Service-Oriented Architecture and Web Services.* RedBooks, April 2004.

[14] D.A Menasce. Automatic qos control. *IEEE Internet Computing*, pages 92–95, 2003.

[15] G. Morgan, S. Parkin, C. Molina-Jimenez, and J. Skene. Monitoring middleware for service level agreements in heterogeneous environments. In *Proc. 5th IFIP Conf. on e-Commerce, e-Business, and e-Government (I3E 2005), Poznan, Poland, October 26-28, IFIP Volume 189*, pages 79–93. IFIP, 2005.

[16] A. Dan D. Davis R.Kearney A.Keller R.King D.Kueber H.Ludwig M.Polan M.Spreitzer and A.Youssef. Web service on demand: Wsla-driven automated management. In *IBM System Journal*, pages 136–158, 2004.

[17] OASIS. *Universal Description Discovery & Integration (UDDI) 3.0.2*, October 2004. http://uddi.org/pubs/uddi_v3.htm.

[18] Arulazi D. Rajesh Sumra. Quality of service for web services-demystification, limitations, and best practices. March 2003. (See http://www.developer.com/services/article.php/2027911.).

[19] A. Sabetta A. Bertolino S. Elbaum, G. De Angelis. Scaling up sla monitoring in pervasive enviroments. 2007.

[20] Akhil Sahai, Anna Durante, and Vijay Machiraju. Towards automated SLA management for web services. Technical Report HPL-2001-310R1, Hewlett Packard Laboratories, 2002.

[21] Shuping Ran. A model for web services discovery with qos. March 2003.

[22] G. Spanoudakis and K. Mahbub. Non intrusive monitoring of service based systems. *Int. J. of Cooperative Information Systems*, 15(3):325–358, 2006.

[23] Thomas Erl. *Service-Oriented Architecture and Web Services A Field Guide to Integrating XML and Web Services.* Prentice Hall, 2004.

[24] Bart Verheecke, María Agustina Cibrán, and Viviane Jonckers. Aspect-oriented programming for dynamic web service monitoring and selection. In *Web Services, European Conference, ECOWS 2004, Erfurt, Germany, September 27-30, 2004, Proceedings*, volume 3250 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004.

[25] W3C. *Web Services Description Language (WSDL) 1.1*, March 2001. http://www.w3.org/TR/wsdl.

[26] W3C. *Simple Object Access Protocol (SOAP) 1.2*, June 2003. http://www.w3.org/TR/soap12.