Università di Pisa

Dipartimento di Informatica
Corso di Laurea in Tecnologie Informatiche

Tesi di Laurea

Un'architettura flessibile per il trattamento dei messaggi XML nella Porta di Dominio OpenSPCoop

Candidato:
Giovanni Bussu

Relatori:
Prof. Andrea Corradini

Controrelatore:
Prof. Vincenzo Gervasi

Prof. Tito Flagella

Anno Accademico 2012-13

Riassunto

Tesi di Laurea Specialistica

Un'architettura flessibile per il trattamento dei messaggi XML nella Porta di Dominio OpenSPCoop

di Giovanni Bussu

Il processamento efficiente dei messaggi XML è un aspetto centrale per un insieme di nuovi prodotti, dai Firewall XML ai Service Bus.

La Tesi si focalizza su questa problematica, proponendo un'architettura flessibile in grado applicare la soluzione più adeguata per ogni diversa tipologia di messaggio.

I risultati ottenuti sono applicati al caso concreto della Porta di Dominio OpenSPCoop, fornendo dei benchmark standard a dimostrazione della bontà della soluzione proposta.

Alla mia famiglia.

Ringraziamenti

Non è facile ringraziare, in poche righe, tutte le persone che hanno contribuito sia allo sviluppo di questo lavoro che al percorso intrapreso in questi anni.

Il primo pensiero va ovviamente a *mamma* e *babbo*, ed a mia sorella *Daniela*. Senza il vostro affetto, i consigli, gli incoraggiamenti e l'esempio che ogni giorno mi date non sarei mai riuscito a raggiungere questo traguardo. Grazie. Le soddisfazioni che vi ho dato, e che spero di darvi in futuro, non potranno mai ripagarvi abbastanza per tutto ciò che da sempre fate per me.

Ringrazio il Prof. *Tito Flagella* ed il Prof. *Andrea Corradini* per avermi seguito in questo lavoro con competenza, passione ed entusiasmo, e per avermi permesso di lavorare ad un progetto così stimolante ed ambizioso.

Ringrazio inoltre i colleghi della *Link.it* sia per l'aiuto che, ciascuno in vari contesti, mi hanno dato durante questo lavoro, sia per aver alleggerito le giornate lavorative con numerosi momenti di svago.

Ringrazio infine i miei amici, in particolare *sos cumpanzos* coi quali condivido gioie e dolori da ventotto anni, e gli amici e colleghi conosciuti qui a Pisa, che hanno arricchito la mia esperienza all'ombra della Torre rendendola indimenticabile non solo dal punto di vista didattico ma anche da quello umano.

Indice

Introduzione

Negli ultimi anni il linguaggio XML si è imposto come un veicolo fondamentale per lo scambio di messaggi tra i diversi nodi di applicazioni distribuite. In seguito all'ampia diffusione di questo linguaggio si è diffusa sul mercato una nuova categoria di prodotti, la cui funzione è quella di intercettare e filtrare i messaggi XML in ingresso ed in uscita da un'organizzazione.

I principali prodotti di questo tipo sono i Firewall XML, la cui funzione è quella di verificare i messaggi in ingresso, prima di inoltrarli verso i servizi applicativi interni al dominio dell'organizzazione. Accanto ai Firewall XML, di cui talvolta sono una semplice estensione, si è diffuso un altro importante filone di prodotti, costituito dagli Enterprise Service Bus. La loro funzione è simile a quella dei Firewall XML, ed inoltre essi hanno lo scopo di integrare i diversi sistemi presenti all'interno della stessa organizzazione, permettendo la comunicazione anche tra sistemi realizzati con sistemi disomogenei.

Obiettivo principale di questa categoria di prodotti è quello di spostare dal livello applicativo al livello infrastrutturale tutta una serie di operazioni di processamento dei messaggi XML, dall'autenticazione, alla firma elettronica, alla validazione dei contenuti, fino alla trasformazione dei contenuti del messaggio, con ovvi benefici dal punto di vista della sicurezza, della facilità di manutenzione e della riduzione dei costi di manutenzione delle applicazioni.

Sulla funzione degli ESB è stato anche formalizzato uno specifico pattern, introdotto nel 2004 con il nome di *VETRO* [1]. Questo pattern prevede che il flusso informativo venga intercettato da un certo numero di componenti prima

di essere effettivamente consegnato al servizio applicativo finale. Questi componenti eseguiranno tipicamente operazioni di *Validazione* del contenuto del flusso ricevuto, *Enrichment*, ovvero arricchimento del flusso con informazioni significative nell'ambito del dominio di appartenenza, *Trasformazione* del flusso in modo da renderlo compatibile con un formato richiesto all'interno del dominio, *Routing* del flusso verso una tra le varie possibili destinazioni, prima dell'effettiva *Operate* che lo inoltra al servizio applicativo finale.

Con la massiccia diffusione delle applicazioni XML, il ruolo degli ESB diventa sempre più importante nei sistemi IT delle organizzazioni pubbliche e private e di pari passo si pone il problema del trattamento efficiente dei messaggi XML, per evitare che firewall XML ed ESB diventino un collo di bottiglia nei flussi informativi.

Centrale al processo di trattamento dei messaggi XML sono le tecniche di parsing e la rappresentazione in formato macchina dei documenti XML da processare. Sono pertanto state sviluppate tutta una serie di diverse tecnologie per il parsing, che spesso si evolvono di pari passo a specifici formati di rappresentazione dei documenti, come ad esempio AxiOM [2], con l'Axis Object Model [3], o VTD [4] con il formato VTD-XML [5].

Trattandosi di tecnologie molto ottimizzate, queste risultano essere spesso più o meno adatte alla gestione di specifici tipi di messaggi XML, per cui potremo avere ad esempio che una specifica tecnologia risulti ottima per messaggi di piccole dimensioni, ma meno adatta a messaggi di grandi dimensioni, oppure mostrare un ottimo risultato se il trattamento richiesto richiede di analizzare il documento XML solo parzialmente, ma peggiore di altre se il documento va trattato completamente.

La Tesi si focalizza su questa problematica, analizzando le principali tecnologie alla stato dell'arte per il trattamento dei messaggi XML, in particolare nel contesto del processamento del messaggio in transito, proponendo una soluzione architetturale in grado di applicare tecnologie diverse a diversi tipi di

documenti XML, o anche diverse tecnologie per la soluzione di problemi diversi sullo stesso documento XML (ad esempio validazione ed estrazione di contenuti).

I risultati ottenuti sono applicati al caso concreto della Porta di Dominio (PdD) OpenSPCoop, un Firewall XML conforme alla specifica SPCoop per la cooperazione applicativa tra le Pubbliche Amministrazioni italiane [6, 7]. I risultati ottenuti vengono poi validati, sia confrontando la versione originale della PdD OpenSPCoop con la nuova versione progettata nella Tesi, sia confrontando la nuova versione della PdD con i più diffusi prodotti ESB, riproducendo il Performance Test Framework di WSO2 [8], considerato lo standard de facto per il confronto di questo tipo di prodotti.

Contenuto della Tesi:

Nel **Capitolo 1** si introduce il contesto tecnologico nell'ambito del quale si pone questo lavoro, ovvero quello delle Service Oriented Architecture. Si presenterà l'architettura Web Service ed in particolare uno dei modelli nei quali quest'architettura è suddivisa, il modello orientato ai messaggi. Contestualmente saranno introdotti il messaggio SOAP, che costituisce il formato di interscambio di dati in questo ambito, e il linguaggio XML, del quale il messaggio SOAP rappresenta una particolare istanza. Infine verrà presentata una descrizione del routing SOAP, e verranno presentati a scopo di esempio tre tipi di prodotti appartenenti a questa categoria di applicazioni, ovvero i Firewall XML, gli ESB, e la Porta di Dominio SPCoop.

Il **Capitolo 2** si occupa di presentare le problematiche relative alla gestione del messaggio SOAP, ad iniziare dal parsing XML, analizzando le varie tecniche di parsing ed evidenziando le differenze tra esse sia in termini funzionali che prestazionali. Oltre al parsing XML saranno descritte nel dettaglio le problematiche che un nodo SOAP sarà tenuto a gestire.

Nel **Capitolo 3** viene descritto il funzionamento di un generico nodo SOAP, a partire dalla descrizione dell'architettura astratta alla quale si riconduce il

funzionamento di un nodo SOAP, seguita da un'analisi sulle differenti problematiche riguardanti i vari casi d'uso che rappresentano le problematiche comuni a questo tipo di architettura. L'analisi metterà in evidenza l'impossibilità di gestire tutti i casi d'uso con la medesima tecnologia, evidenziando al contrario l'esigenza di un'architettura che gestisca lo specifico messaggio con la tecnologia più adatta. Questo risultato verrà applicato nei successivi capitoli al caso concreto della PdD OpenSPCoop.

Il **Capitolo 4** tratta le tecnologie utilizzate per affrontare le problematiche di competenza del nodo SOAP descritte in precedenza, con particolare attenzione al modello di parsing utilizzato da ciascuna tecnologia, mostrando così la possibilità reale di utilizzare diverse tecnologie per affrontare diverse problematiche.

Nel **Capitolo 5** verrà presentata l'attuale architettura della PdD OpenSPCoop, in maniera limitata al punto di vista della gestione del messaggio SOAP, con lo scopo di evidenziare vantaggi e limiti di questa architettura rispetto agli aspetti analizzati nel Capitolo 3. Successivamente verrà introdotta la nuova architettura, che permette una gestione flessibile del messaggio SOAP, ed i plugin realizzati per rendere efficiente tale gestione.

Nel **Capitolo 6** si presenteranno i risultati dei test che, attraverso il benchmark standard WSO2 [8] , sono stati eseguiti per valutare le prestazioni ottenute dalla nuova architettura. Queste sono state confrontate prima con quelle ottenute dalla vecchia architettura, e successivamente con i più diffusi prodotti ESB.

Capitolo 1

Contesto tecnologico

In questo capitolo verrà presentato il contesto tecnologico in cui si pone questo lavoro di Tesi. Dopo una descrizione di Service Oriented Architecture e architettura Web Service, si parlerà del linguaggio XML e dei messaggi SOAP, e verranno introdotte opportunità e problematiche legate al routing SOAP, presentando alcuni esempi di router SOAP.

1.1 Service Oriented Architecture

Una Service Oriented Architecture (SOA) è una forma di architettura di sistemi distribuiti basata sui *servizi*. Un *servizio* è una particolare tipologia di software basata sull'idea di *esporre*, tramite un'interfaccia ben definita, una serie di *operazioni*, ciascuna delle quali può essere invocata da un *agente* fruitore, eventualmente in maniera asincrona.

In ambito SOA, un servizio è tipicamente caratterizzato da queste proprietà:

- Vista logica: Un servizio è un'astrazione del programma, database, processo ecc., definito in termini di cosa fa, tipicamente astraendo un'operazione.

- Orientato ai messaggi: Un servizio è formalmente definito in termini dei messaggi scambiati tra l'agente erogatore e l'agente fruitore e non nei termini delle proprietà degli agenti stessi. La struttura interna degli agenti, compreso ad esempio il linguaggio di programmazione usato per implementarli, la struttura dei processi o la struttura delle basi di dati, sono deliberatamente astratti nella SOA: non deve essere assolutamente necessario sapere qualcosa dell'implementazione di un agente per interagirci.

- Orientato alla descrizione: Un servizio è descritto da metadati interpretabili da una macchina. Questa descrizione deve supportare la natura pubblica della SOA: solo quei dettagli utili all'uso del servizio devono essere resi pubblici. La semantica del servizio deve essere, direttamente o indirettamente, inserita in questa descrizione.

- Granulare: I servizi tendono ad usare poche operazioni con messaggi relativamente grandi e complessi.

- Orientato alla rete: I servizi tendono ad essere utilizzati attraverso una rete, anche se non è un requisito assoluto.

- Indipendente dalla piattaforma: I messaggi sono inviati in un formato standard indipendente dalla piattaforma usata.

1.2 Architettura web service

Un Web Service, così come definito in [9], è un software progettato per fornire un'interazione machine-to-machine attraverso una rete. Ha un'interfaccia definita in un linguaggio interpretabile da una macchina (nello specifico WSDL). Gli altri sistemi interagiranno con il Web Service in modo conforme alla sua descrizione tramite l'invio di messaggi SOAP, tipicamente usando il protocollo HTTP con serializzazione XML, in congiunzione con altri standard web.

Definire con precisione quale sia l'architettura dei Web Service è molto più complesso di quanto si possa pensare. Lo sviluppo di questa tecnologia ha seguito un percorso tale che è venuta a mancare la definizione standard dell'architettura su cui si basa. Nell'articolo [9] la descrizione dell'architettura viene suddivisa in *modelli*, ovvero in sottoinsiemi dell'architettura che si occupano di gestire un aspetto specifico trattato dall'architettura generale.

Il Message Oriented Model si concentra sui messaggi, sulla loro struttura e sul loro trasporto, senza particolari riferimenti alla ragione per cui questi messaggi sono creati o al loro significato.

Il Service Oriented Model si concentra sulla gestione di servizi, action e così via. Per chiarezza, in qualsiasi sistema distribuito i servizi non possono essere adeguatamente realizzati se non si ha qualche nozione sui messaggi scambiati, mentre un messaggio può non avere nozioni sul servizio a cui è destinato.

Il Resource Oriented Model si focalizza sulle risorse esistenti e chi le controlla. Il modello delle risorse è ereditato dal concetto di risorsa dell'Architettura Web e serve ad includere le relazioni tra risorse e i detentori delle stesse.

Il Policy Model si focalizza sulle relazioni tra gli agenti ed i servizi. Le Policies sono applicate dalle persone che le gestiscono agli agenti che vogliono far uso di determinate risorse. Le Policies possono essere introdotte per gestire aspetti quali la sicurezza, la qualità del servizio (QoS), il management, ecc.

Il modello più interessante, ai fini del nostro studio, è il Message Oriented Model, che verrà per questo presentato più nel dettaglio.

1.2.1 Modello orientato ai messaggi

L'essenza del modello orientato ai messaggi ruota attorno ad alcuni concetti chiave illustrati nel grafo in Figura 1.1 [9]: l'agente che invia e riceve il messaggio, la struttura del messaggio in termini di header e body e il meccanismo usato per inviare il messaggio. Nell'ambito dei Web Services il tipo di messaggi scambiati è il SOAP, una particolare istanza di messaggi XML.

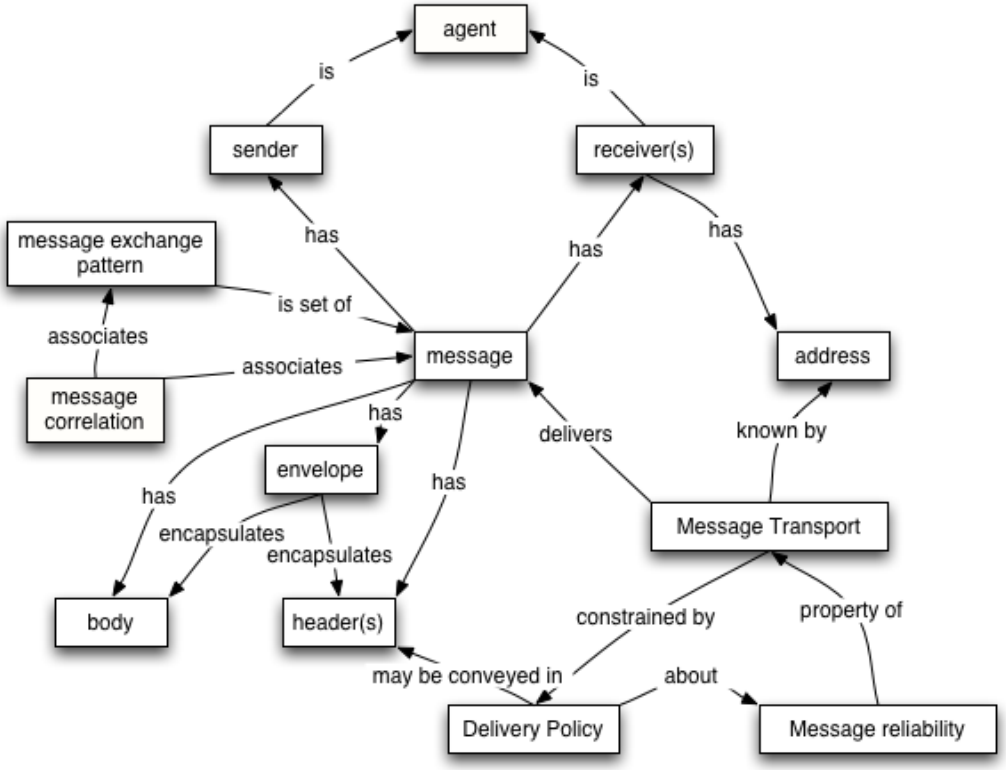


FIGURA 1.1: Modello Orientato ai Messaggi

In questo grafo gioca sicuramente un ruolo centrale il concetto di *messaggio*: esso rappresenta l'entità base che può essere scambiata tra un agente erogatore e uno fruitore (che in questo contesto sono il *mittente* e il *destinatario* del messaggio) nell'ambito di una SOA. La struttura interna del messaggio è costituita da zero o più *header* e un *body*, incapsulati da un *envelope*. Un header è la parte di messaggio atta a contenere informazioni, spesso sotto forma di metadati, riguardo un aspetto specifico del messaggio. Il body rappresenta invece le informazioni applicative destinate al servizio finale. Una serie di messaggi, legati da un legame applicativo, possono essere raggruppati in un *Message Exchange Pattern*, ovvero un pattern che descrive le relazioni tra più messaggi scambiati nell'ambito di una conversazione tra servizi, definendo le regole di questa conversazione e descrivendo la terminazione normale o anormale della stessa. I messaggi appartenenti ad un pattern sono identificati da un identificativo univoco, detto anche identificativo di *correlazione*.

1.2.1.1 XML

XML, acronimo di eXtensible Markup Language, è un linguaggio di markup molto flessibile, creato e gestito dal World Wide Web Consortium (W3C), nato come una semplificazione e adattamento dello Standard Generalized Markup Language (SGML) nel 1998. Pensato inizialmente per scopi di archiviazione di dati, XML si è affermato nell'ambito dello scambio di dati tra le applicazioni in ambito Web Service. Concretamente, un documento XML è un file di testo che contiene una serie di tag, attributi e testo secondo regole sintattiche ben definite.

Un documento XML è intrinsecamente caratterizzato da una struttura gerarchica. Esso è composto da componenti denominati elementi: ciascun elemento rappresenta un componente logico del documento e può contenere altri elementi (sottoelementi) o del testo. Gli elementi possono avere associate altre informazioni che ne descrivono le proprietà. Queste informazioni sono chiamate attributi. L'organizzazione degli elementi segue un ordine gerarchico (struttura ad albero) che prevede un elemento principale, chiamato *root element*, del quale tutti gli altri elementi sono discendenti. Il linguaggio XML non specifica vincoli sul contenuto degli elementi, né elementi predefiniti, e gli elementi in un documento XML non hanno alcun significato semantico intrinseco. In questo senso si può affermare che XML è estensibile.

Per gli scopi di questa tesi l'XML è importante in quanto in ambito Web Services i messaggi scambiati hanno la forma di un'istanza particolare di documento XML, ovvero il messaggio SOAP, descritto più avanti.

1.2.1.2 XML Schema e XML Schema Definition

L'XML Schema o Schema XML è l'unico linguaggio di descrizione del contenuto di un file XML che abbia per ora raggiunto la validazione (la 1.1) ufficiale del W3C. Come tutti i linguaggi di descrizione del contenuto XML, il suo scopo è quello di definire la struttura ed il tipo di dati consentiti in un documento XML

in termini di vincoli: quali elementi ed attributi possono apparire, in quale relazione reciproca, quali tipi di dati sono ad essi associati e quale relazione gerarchica hanno fra loro gli elementi contenuti nel documento XML.

Una XML Schema Definition (XSD) è un'istanza di linguaggio XML Schema, sviluppato con l'intento che la determinazione della validità di un documento possa produrre una collezione di informazioni aderenti a specifici tipi di dati. In altre parole, XSD si ripropone di diventare, per il processamento di documenti XML, quello che è il *type-system* per i linguaggi Object-Oriented. I benefici di un type-system sono molteplici: il vantaggio principale è che esso permette di avere un controllo maggiore sulle operazioni eseguite, dal momento che controlla che l'operazione che si vuole eseguire abbia senso, nel dominio dei tipi a cui si riferisce, ma la sua importanza non si ferma qui. Avere la certezza che un oggetto sia un'istanza di una specifica classe è infatti fondamentale per la scrittura di codice generato, per la serializzazione degli oggetti, oltre che per molti altri servizi che che un linguaggio può fornire a runtime.

1.2.1.3 Il messaggio SOAP

Un messaggio SOAP [10] è l'entità più piccola che può essere trasmessa da un agente ad un altro nell'ambito Web Services. Il messaggio SOAP rappresenta la struttura dati inviata dall'agente fruitore all'agente erogatore del servizio, ed è formato dall'envelope, a sua volta formato da zero o più headers, e dal body. L'envelope è un contenitore di messaggi, e può contenere informazioni necessarie al livello di trasporto per effettuare la consegna del messaggio. L'header è una parte di messaggio contenente informazioni su un aspetto specifico del messaggio. Queste informazioni possono essere standardizzate diversamente, e possono avere semantiche diverse, da quelle del messaggio. La funzione principale dell'header è quella di facilitare il processing modulare del messaggio, anche se questi possono essere usati per fornire informazioni di routing ed altri aspetti relativi al processing del messaggio. Ogni messaggio può avere più header, ciascuno dei quali si riferisce potenzialmente ad un diverso ambito del

servizio. Gli header dovrebbero essere processati indipendentemente dal resto del messaggio, ed indipendentemente gli uni dagli altri. Il body contiene il messaggio applicativo vero e proprio o le URI alle quali reperirlo.

1.2.1.4 Il messaggio SOAP with Attachments

Un messaggio SOAP potrebbe essere trasmesso assieme ad allegati di vario tipo, e questi dati sono spesso codificati in un formato binario. Il documento [11] descrive una maniera standard di associare un messaggio SOAP ad uno o più *attachments*, spediti nel loro formato nativo ed associati al messaggio tramite una struttura Multipurpose Internet Mail Extensions (MIME) Multipart/Related.

La specifica Soap Messages with Attachments, descritta in [11] definisce un formato di struttura del messaggio SOAP di tipo MIME Multipart/Related, descritto in [12], nel quale alla parte SOAP del messaggio, ovvero al SOAP Envelope, sono associati uno o più allegati. La struttura così definita è chiamata SOAP Message Package, ed è costruita secondo le seguenti regole:

- L'elemento Content-Type relativo al SOAP header deve essere il tipo multipart/related.
- Il messaggio SOAP è trasportato nella parte principale della struttura multipart/related, ed il parametro type relativo ad esso dev'essere text/xml.
- gli attachments MIME sono riferiti all'interno del body del messaggio SOAP tramite l'uso di references.
- gli attachments riferiti devono contenere o un Content-ID o un Content-Location corrispondente a quello indicato nel body del messaggio SOAP.

Inoltre la specifica raccomanda che, per ragioni di robustezza, la parte principale, contenente il messaggio SOAP, contenga un header Content-ID, e che l'header multipart/related contenga un parametro start corrispondente.

1.2.2 Routing SOAP

Il paradigma di sviluppo software basato sui Web Service ha portato diverse innovazioni dal punto di vista architetturale. Queste innovazioni sono diretta conseguenza dell'adozione di un modello orientato ai messaggi, nel quale i dati viaggiano tra le applicazioni in un formato ispezionabile (puro testo) e non più in un formato binario seppur condiviso, come avveniva nelle precedenti architetture distribuite. L'ispezionabilità del messaggio ha portato lo sviluppo di software distribuito da una prospettiva *point-to-point*, in cui il client e il servizio dialogavano esclusivamente tra loro, ad una prospettiva *end-to-end*, dove il client e il servizio sono le estremità di una catena di nodi intermedi che il messaggio dovrà attraversare nel corso dell'invocazione del servizio. Conseguenza di questo passaggio è la nascita di una nuova classe di componenti applicativi, noti come router SOAP, che si occupano di smistare i messaggi tra client e servizio, fornendo al contempo una serie di servizi infrastrutturali tipicamente gestiti in precedenza a livello applicativo.

1.2.2.1 Firewall XML

Dal momento che i Web Services si basano sul trasferimento di dati tra macchine attraverso la rete internet, le applicazioni basate su tale paradigma devono necessariamente porre un'attenzione aggiuntiva ai problemi relativi alla sicurezza, dal momento che ogni messaggio ricevuto può contenere codice maligno. Dalla ricerca di una soluzione infrastrutturale, e non applicativa, a questo problema, nasce l'idea che sta alla base della classe di software noti come Firewall XML.

Un Firewall XML è un firewall applicativo che ha lo lo scopo di proteggere le applicazioni XML-based da una serie di attacchi che cercano di sfruttare vulnerabilità a livello di messaggio XML e di parser. I firewall XML mirano a migliorare la sicurezza delle applicazioni basate su XML, prevenendo una serie di attacchi che causerebbero l'interruzione del servizio, nel caso venissero

portati a termine su un application server. I firewall XML sono progettati per proteggere l'applicazione dai tipici attacchi web-based che possono essere eseguiti attraverso l'invio di un messaggio XML, come ad esempio SQL injection, o cross-site scripting (XSS). Alcuni tipici esempi sono attacchi che mirano a saturare la memoria dell'applicazione con messaggi molto estesi, saturarne il parser a causa della presenza di elementi annidati con un livello di profondità troppo alto, o messaggi che contengano al loro interno codice eseguibile. Il firewall XML è annoverato nella classe dei router SOAP perché il suo compito è quello di ispezionare i messaggi in arrivo e in uscita prima che essi vengano passati all'applicazione o al client, e di conseguenza esso è fornito generalmente come proxy applicativo.

1.2.2.2 Service Bus

Un Enterprise Service Bus (ESB) è un'infrastruttura software che fornisce servizi di supporto ad architetture SOA complesse, fornendo servizi di orchestration, sicurezza, messaggistica, routing intelligente e trasformazioni dei messaggi scambiati. L'idea dell'ESB nasce da quella di Enterprise Integration Broker (EIB). Un EIB è un componente che agisce nell'ambito dell'integrazione tra applicazioni, permettendo loro di interagire in maniera disaccoppiata attraverso il mascheramento delle differenze tecnologiche. In questo caso lo svantaggio principale era quello di avere queste funzionalità di integrazione concentrate in un unico punto all'interno dell'architettura, il che rendeva il sistema poco scalabile ed esponeva l'architettura a rischi di malfunzionamenti e cali di performance. L'ESB nasce come soluzione migliorativa rispetto all'EIB in quanto è pensata in modo da essere presente in maniera decentralizzata e scalabile, agendo come una dorsale attraverso la quale viaggiano servizi software e componenti applicativi. In questo modo le unità elementari interconnesse con l'ESB possono realizzare i propri servizi mantenendone un controllo locale, e nel contempo essere capaci di unire ogni singolo progetto in un progetto globale. Inoltre all'ESB sono delegati i servizi comuni, denominati core service,

che andrebbero altrimenti realizzati da ogni singola unità elementare. Un'altra caratteristica di un ESB è quella di permettere un'integrazione incrementale: è possibile infatti integrare in momenti differenti le varie parti di un sistema per permettere la completa indipendenza delle unità elementari.

1.2.2.3 PdD SPCoop

Il Sistema Pubblico di Cooperazione (SPCoop) è un insieme di standard tecnologici e di servizi infrastrutturali il cui obiettivo è di permettere l'interoperabilità e la cooperazione di sistemi informatici per la realizzazione di adempimenti amministrativi. Tali sistemi sono sotto la responsabilità di soggetti pubblici, appartenenti ad amministrazioni centrali, enti pubblici, regioni, provincie, comuni, comunità di enti locali, e soggetti privati (imprese e associazioni accreditate). L'insieme dei soggetti pubblici e privati operanti sul Servizio Pubblico di Cooperazione costituiscono la comunità dei soggetti del Servizio Pubblico di Cooperazione. Tale cooperazione è motivata da due esigenze fondamentali:

- L'esigenza di coordinamento di processi realizzati con il concorso di trattamenti distribuiti tra sistemi informatici di cui sono responsabili soggetti pubblici e privati, al fine di assecondare l'esecuzione di procedimenti amministrativi e la produzione di atti e provvedimenti amministrativi. Il coordinamento e la collaborazione di detti sistemi devono essere corredati dalla capacità di ispezionare in ogni momento lo stato di avanzamento (gli adempimenti amministrativi effettuati e quelli ancora da effettuare) dei processi applicativi e l'origine di ogni atto amministrativo effettuato nell'ambito del processo applicativo, al fine di realizzare concretamente la trasparenza dell'azione amministrativa nel doveroso rispetto delle norme sulla confidenzialità e riservatezza dei dati.

- Il coordinamento e la collaborazione dei trattamenti distribuiti tra più sistemi informatici appartenenti a più soggetti pubblici e privati, al fine di assicurare il funzionamento interno delle amministrazioni e di fornire

servizi di utilità alle altre amministrazioni, ai cittadini, alle imprese e alle associazioni, in conformità con i compiti istituzionali delle diverse amministrazioni.

Il progetto OpenSPCoop nasce sostanzialmente con l'obiettivo di una implementazione di riferimento Open Source che permetta di sperimentare in maniera condivisa l'implementazione dei concetti proposti nella specifica, evidenziando e proponendo possibili soluzioni per le potenziali ambiguità o debolezze della stessa. I vantaggi dell'approccio Open Source adottati da OpenSPCoop si possono evidenziare nei seguenti aspetti:

- Interoperabilità, OpenSPCoop intende rappresentare un riferimento per disambiguare diverse possibili interpretazioni della specifica SPCoop;
- Sicurezza, l'apertura del codice assicura quelle caratteristiche di trasparenza del codice ormai considerate un atto dovuto in molti settori della sicurezza informatica;
- Comunità d'Utenza, OpenSPCoop tende a fungere da catalizzatore per le esperienze e le competenze degli utenti, permettendo di ricapitalizzarle in risultati concreti e riusabili;
- Innovazione, un'implementazione Open Source è il veicolo ideale per proporre delle implementazioni condivisibili di quanto non ancora trattato nelle specifiche SPCoop.

Dopo un'ampia fase di analisi, svolta attraverso lo studio di vari progetti pilota tra cui il progetto CART di Regione Toscana e il progetto SOLE di Regione Emilia Romagna, è emersa la proposta di un'architettura innovativa per la realizzazione di un'infrastruttura compatibile con le specifiche CNIPA (successivamente DigitPA [13]), che riducesse però significativamente l'impatto sui sistemi preesistenti rispetto alle altre soluzioni disponibili. La prima release del software OpenSPCoop, la 0.1, ha richiesto dapprima una approfondita fase di analisi e di progettazione che ha costituito il lavoro oggetto della tesi di

Ruggero Barsacchi [14]. In una fase successiva, la progettazione è stata estesa ed è stata prodotta la prima versione del codice OpenSPCoop grazie al lavoro di tesi di Andrea Poli [15]. Questa prima release, rilasciata il 27 ottobre 2005, è stata seguita da frequenti nuovi rilasci, anche grazie al feedback e al contributo dei primi utenti del software. Attorno al sito [6] ha cominciato subito a svilupparsi una comunità di utenti e sviluppatori molto qualificati, provenienti da grandi aziende italiane, pubbliche amministrazioni locali e centrali e centri di ricerca, e l'architettura di OpenSPCoop si è ulteriormente evoluta come descritto nei lavori di tesi di Lorenzo Nardi [16] e Aldo Lezza [17]. OpenSPCoop è stato inoltre discusso approfonditamente in alcuni articoli presentati in varie conferenze scientifiche, come ad esempio in [7] e in [18]. L'interesse diffuso per il progetto ha dimostrato tra l'altro la grande disponibilità di tutti i soggetti interessati verso una soluzione Open Source in un settore così critico come quello indirizzato dalla specifica SPCoop.

Capitolo 2

Modalità di gestione del messaggio SOAP

In questo capitolo verranno affrontate le principali tematiche relative alla gestione di un messaggio SOAP. Queste tematiche dovranno essere affrontate e risolte da un Router SOAP. Tra queste tematiche figurano il parsing XML, che vedremo analizzando le differenze tra i vari modelli di parsing, la validazione di messaggi XML tramite XML Schema Definition, la ricerca all'interno di messaggi XML, il parsing SOAP, l'archiviazione del messaggio SOAP e l'applicazione della WS-Security al messaggio SOAP.

2.1 Parsing XML

Il parsing è il processo atto ad analizzare uno stream ricevuto in input in modo da determinare la sua struttura sintattica grazie ad una data grammatica formale. Nello specifico, il parsing XML si occupa di trasformare l'input XML in modo che esso possa essere fruibile a livello applicativo. L'entità che si occupa di effettuare il parsing XML si chiama parser. Esistono vari modelli di parsing XML e, come vedremo, a seconda del modello scelto le prestazioni delle applicazioni possono variare anche sensibilmente.

In generale, esistono tre modelli di parsing per documenti XML. Il primo è il *Document Object Model*, abbreviato in DOM, di tipo *tree-based*. Il secondo è lo *Streaming Model*, di tipo *event-based*. Il modello streaming si divide a sua volta in due categorie, ovvero *push* (ad esempio SAX) e *pull* (ad esempio StAX) parsing. Il terzo modello è il *non-extractive parsing*, abbreviato in NEP.

Di seguito verranno presentati i concetti relativi ai differenti modelli di parsing. Gli esempi forniti a corredo sono estratti di codice Java, ma tali concetti restano validi a prescindere dal linguaggio di programmazione utilizzato.

2.1.1 Document Object Model (DOM)

Il modello DOM si basa sulla costruzione in memoria della rappresentazione ad oggetti dell'intero documento XML. Una volta ricevuto il documento, il parser si occupa di costruire un albero di oggetti che rappresenta il contenuto e l'organizzazione dei dati contenuti. L'albero viene creato in memoria e l'applicazione può attraversarlo e modificarlo liberamente. Inoltre l'albero modificato può essere a sua volta riconvertito in un documento XML. Il funzionamento del modello DOM può essere riassunto nell'immagine 2.1:

L'applicazione inizializza un DocumentBuilder, ad esempio attraverso una DocumentFactory, e gli fornisce in ingresso lo stream di dati rappresentante il documento XML; in output otterrà l'oggetto rappresentante l'albero gerarchico dei nodi di quel documento. Se prendiamo ad esempio il seguente documento XML:

```
<Authors>
    <Author>
        <firstName>Alfred</firstName>
        <lastName>Aho</lastName>
    </Author>
    <Author>
        <firstName>Ravi</firstName>
        <lastName>Sethi</lastName>
    </Author>
```

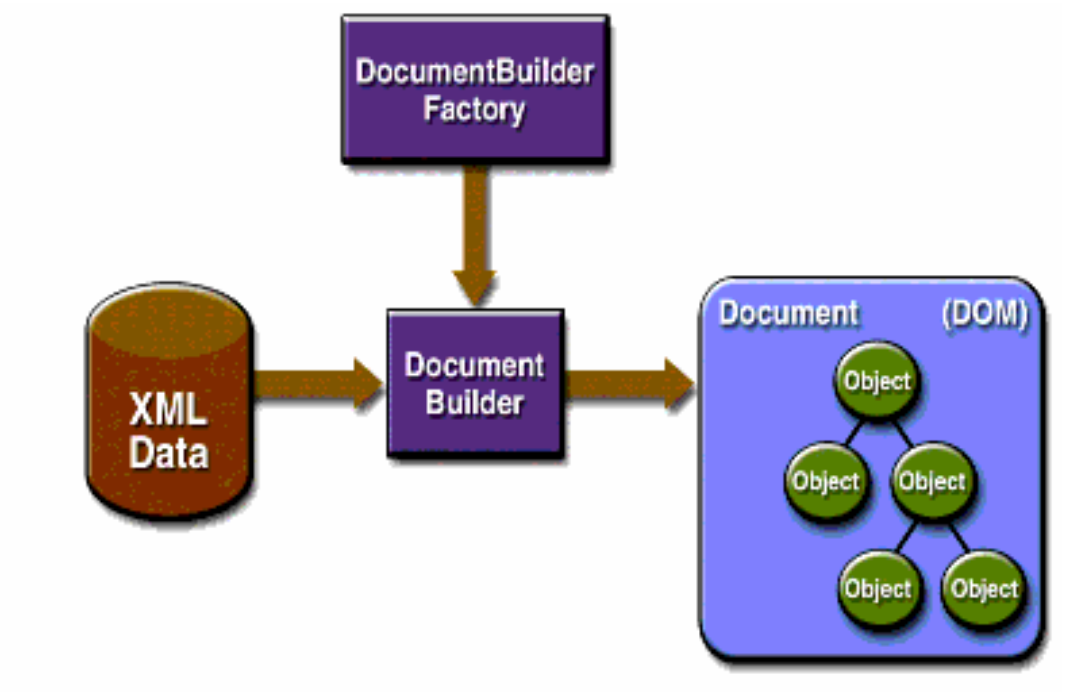


FIGURA 2.1: Parsing DOM

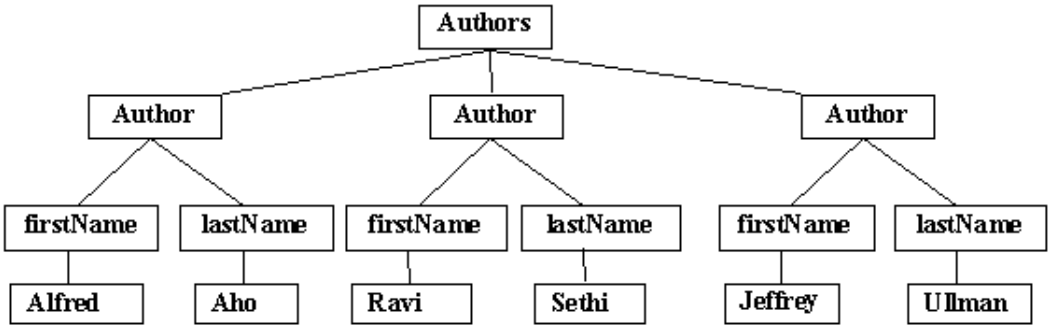


FIGURA 2.2: Esempio di un documento XML dopo il parsing DOM

```
    <Author>
        <firstName>Jeffrey</firstName>
        <lastName>Ullman</lastName>
    </Author>
</Authors>
```

fornendolo in input ad un parser DOM otterremo in output l'albero in Figura 2.2.

Il seguente estratto di codice ci mostra l'utilizzo delle API DOM: le prime tre righe di codice ci mostrano l'inizializzazione del parser e l'operazione di

parsing del documento (memorizzato nel file authors.xml). A questo punto è possibile accedere alla radice del documento (il primo nodo), e da lì effettuare una ricerca basata sul nome dell'elemento. In questo caso vogliamo ottenere tutti i nodi che si chiamano Author. Per ogni elemento Author possiamo accedere al valore del sottoelemento firstName, e stamparlo. Inoltre potremmo avere l'esigenza di modificare uno specifico nodo, come si vede nell'esempio in cui tutti i nodi che hanno Sethi come lastName vengono modificati. Alla fine dell'elaborazione possiamo riversare il contenuto dell'albero in un output stream.

DOMParser.java

```
//Ottiene un DocumentBuilderFactory
DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
//Esegue il parsing con un comando
Document doc = builder.parse(new File("authors.xml"));
Node root = doc.getFirstChild();
NodeList listOfAuthors =
            doc.getElementsByTagName("Author");
//Naviga la lista dei nodi
for(Node author : listOfAuthors) {
    NodeList firstNameNodeList =
        author.getElementsByTagName("firstName");
    String firstName =
            firstNameNodeList.item(0).getNodeValue();
    System.out.println("Author: first name= "
                        +firstName);
    if("Sethi".equals(lastName)) {
        firstNameNodeList.item(0).removeChild();
        firstNameNodeList.item(0).
                appendChild(doc.createTextNode("Ravi"));
    }
}
```

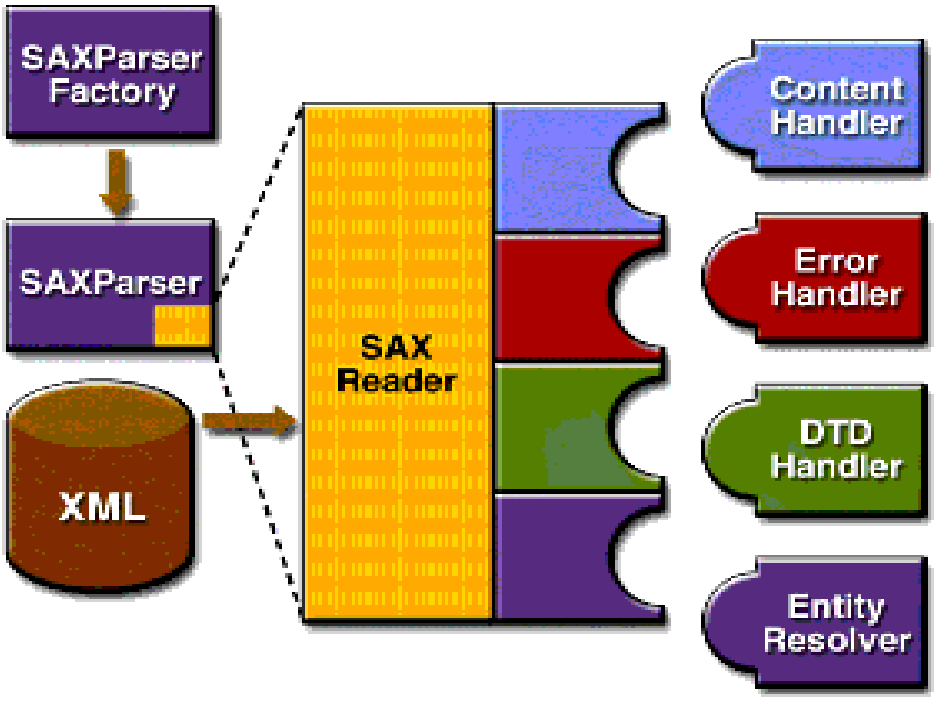


FIGURA 2.3: Parsing SAX

Tra i vantaggi di questo modello abbiamo la semplicità di utilizzo (il paradigma usato è esattamente quello object-oriented), e una grande flessibilità ottenuta grazie alla presenza di API di alto livello che permettono di elaborare il documento con grande facilità. Il prezzo da pagare per tanta potenza e flessibilità è un costo in termini di memoria occupata che, per documenti estesi, arriva ad essere oltre sette volte la dimensione del documento stesso, come descritto nell'articolo [19].

2.1.2 Streaming Push parsing (SAX)

Il modello di parsing di tipo Streaming Push agisce sullo stream di dati in transito, trasformandolo in eventi che dovranno essere gestiti da opportuni handler. Un esempio di API streaming di tipo push è Simple API for XML (SAX). Il suo funzionamento è riassunto in Figura 2.3

In questo modello il parser è costituito da un SAX Reader, che si occupa di

leggere lo stream in input. Al Reader possono venire agganciati quattro handler, ognuno dei quali è incaricato di gestire differenti tipi eventi che verranno via via lanciati dal Reader:

- Content Handler riceve notifica degli eventi che hanno a che fare col contenuto vero e proprio del documento. Gli eventi principali gestiti dal Content Handler sono:
  - *startDocument* evento lanciato all'inizio del documento
  - *endDocument* evento lanciato alla fine del documento
  - *startElement* evento lanciato all'inizio di ogni elemento
  - *endElement* evento lanciato alla fine di ogni elemento
  - *characters* evento lanciato alla lettura di caratteri all'interno del documento
- Error Handler riceve notifica degli eventi che hanno a che fare con eventuali errori sintattici presenti sul documento. Gli eventi gestiti dall'Error Handler sono *warning*, *error* e *fatalError*, a seconda della gravità dell'errore riscontrato.
- DTD Handler riceve notifica degli eventi che hanno a che fare con le entità Document Type Definition (DTD) nel documento. Una DTD in un documento XML fornisce una lista degli elementi, degli attributi, dei commenti, delle note, o delle entità contenute nel documento. Essa indica inoltre le relazioni che intercorrono tra le entità. Le entità DTD sono considerate antenati delle XML Schema Definition. Queste ultime sono ormai molto più comuni delle prime perché prevedono l'uso dei namespace, e quindi sono estensibili, e soprattutto perché consentono la definizione di tipi di dati.
- Entity Resolver riceve notifica degli eventi che hanno a che fare con le entità esterne, ed è usato in particolare per mappare correttamente eventuali URN presenti nel documento nei corrispondenti URL.

Un esempio di uso di un parser di questo tipo è mostrato di seguito.

SAXParser.java

```
SAXParser saxParser = new SAXParser();
MyContentHandler myHandler = new MyContentHandler();
saxParser.setContentHandler(myHandler);
saxParser.parse(new File("authors.xml"));
```

Questo codice permette di inizializzare un SAX parser, agganciarvi un handler di tipo Content (non è necessario agganciare tutti e quattro i tipi di handler), e chiamare il metodo parse definito nel SAXParser. Questa chiamata farà sì che venga eseguito il parsing totale del documento. Il codice relativo alla classe MyContentHandler è riportato di seguito.

MyContentHandler.java

```
public class MyContentHandler extends DefaultHandler {
    // siamo all'interno di un tag chiamato firstName?
    private boolean isFirstName;

    // chiamato all'inizio di un elemento
    public void startElement(String uri, String localName,
                             String qName, Attributes atts) {
        if ("firstName".equals(localName))
            isFirstName = true;
    }

    // chiamato alla fine di un elemento
    public void endElement(String uri, String localName,
                           String qName) {
        if("firstName".equals(localName))
            isFirstName = false;
    }

    // chiamato quando ci sono caratteri
    // all'interno di un elemento
    public void characters(char[] chars, int start,
                           int length) {
```

```
            if(isFirstName)
                System.out.println(
                    new String(chars, start, length));
        }
    }
```

Le classi che, come MyContentHandler, implementano l'interfaccia DefaultHandler devono implementare tre metodi: *startElement*, *endElement*, e *characters*. Il metodo parse scorrerà il documento XML e, al verificarsi di uno specifico *evento*, come l'inizio di un elemento, la fine di un elemento, e la presenza di caratteri all'interno di un elemento, richiamerà il rispettivo metodo della classe configurata come ContentHandler, all'interno del quale deve essere implementata la logica di gestione dell' *evento*.

L'handler mostrato sopra permette di stampare il contenuto di tutti i nodi che si chiamano *firstName*. La differenza rispetto alla stessa operazione effettuata con le API DOM è che in questo caso si deve procedere sempre in avanti, consumando quindi lo stream già letto, e che quindi eventuali informazioni sulla struttura del documento devono essere salvate manualmente al momento del parsing per poter essere utilizzate in seguito.

2.1.3 Streaming Pull parsing (StAX)

Il modello di parsing di tipo Streaming Pull, come quello di tipo Streaming Push, agisce sullo stream di dati in transito, trasformandolo in eventi. Un esempio di API streaming di tipo push è Streaming API for XML (StAX). Il modello di un parser StAX è mostrato in Figura 2.4:

In questo caso, come si può vedere, è l'applicazione che richiede il prossimo evento al parser non appena ne ha bisogno ed è in grado di gestirlo, a differenza del modello push dove era il parser ad avere il controllo del flusso dei dati. Un esempio d'uso di questo tipo di parser è mostrato nel frammento di codice seguente:

StAXParser.java

FIGURA 2.4: Parsing StAX

```
XMLInputFactory fac = XMLInputFactory.newInstance();
XMLEventReader eventReader = fac.createXMLEventReader(
                    new FileInputStream("authors.xml"));
while(eventReader.hasNext()) {
    XMLEvent event = eventReader.next();
        if (event instanceof StartElement){
            String localName =
                ((StartElement)event).getLocalName();
            if("firstName".equals(localName)) {
                Characters nextChars =
                        (Characters) eventReader.next();
                System.out.println(nextChars.getData());
            }
        }
}
```

Questo frammento di codice permette di stampare il contenuto di tutti i nodi che si chiamano *firstName*. Come nel caso del parser SAX, il concetto chiave su cui si basa il parsing è quello di *evento*. In questo caso però non

abbiamo un metodo che gestisce in maniera automatica il parsing, ma la sequenza di eventi deve essere gestita dal programmatore. A questo scopo è stata creata la classe XMLEventReader. Il suo funzionamento è riassunto di seguito: essa viene creata a partire da uno stream rappresentante il documento XML su cui eseguire il parsing ed espone, tra gli altri, due metodi: il primo, *hasNext*, permette di sapere se siamo arrivati alla fine dello stream, e il secondo, *next*, permette di leggere dallo stream il prossimo token, ovvero il prossimo frammento di XML rilevante, come può essere l'inizio o la fine di un elemento, o i caratteri al suo interno. Il metodo *next* restituisce un oggetto di tipo XMLEvent, la cui classe di appartenenza determina il tipo di evento verificatosi. Come si può vedere nell'esempio, il parsing può essere effettuato tramite un loop nel quale vengono recuperati tutti gli eventi e, per ognuno di questi, viene effettuata la specifica gestione a seconda del tipo di evento verificatosi.

Le analogie rispetto al modello push sono evidenti, sia per quanto riguarda la granularità degli eventi da gestire che per quanto riguarda la natura streaming del modello, e di conseguenza il fatto che lo stream letto viene consumato e non può essere recuperato. Rispetto al modello push ci sono alcuni indubbi vantaggi. Oltre a quello già citato dell'inversione del controllo sul flusso dei dati, i vantaggi risiedono nel poter leggere più documenti con un solo thread, e nel fatto che i client di un parser pull sono significativamente più semplici di quelli di un client push, anche per documenti più complessi. Vi sono inoltre vantaggi di altro tipo, come ad esempio il poter produrre XML invece che semplicemente leggerlo, che non hanno però impatto sull'applicazione descritta nella tesi.

2.1.4 Non-extractive Parsing (NEP)

Un'ulteriore alternativa ai modelli di parsing event-based e tree-based è quella del *non-extractive parsing* (NEP). Con questo termine si intende un modello di parsing in cui il documento originale viene memorizzato completamente e

non trasformato in una precisa rappresentazione ad oggetti, come avviene nel DOM, o scartato, come avviene nei modelli Streaming.

Questo modello si basa sull'indicizzazione dei dati, una tecnica che viene di norma usata per salvare dei puntatori a dati che si presume vengano utilizzati spesso, quando si ha a che fare con documenti molto estesi. In particolare nel non-extractive parsing vengono indicizzati i dati relativi ad ogni singolo nodo, in modo da garantire un accesso casuale a tutte le informazioni del documento.

Un esempio di API che usa un modello di non-extractive parsing è VTD-XML [4, 19]. In questa API l'indicizzazione è basata sui *Virtual Token Descriptor* (VTD). Virtual Token Descriptor è una codifica binaria per l'indicizzazione di documenti XML che memorizza tutte le informazioni necessarie in termini di offset e lunghezza di ogni token XML.

Le informazioni salvate in un record VTD sono:

- L'offset (in byte) di inizio del token.
- La lunghezza del token.
- Il livello di profondità del token, ovvero il numero di nodi che si trovano tra la radice del documento e il token.
- Il tipo di token (inizio o fine di un elemento, attributo, testo).

Dato il tipo di memorizzazione delle informazioni le API non sono basate sui nodi, come succede nel DOM, ma sui cursori. Un cursore può essere visto come un puntatore ad un nodo nel documento XML. Può puntare ad un solo nodo alla volta ma, quando richiesto, può essere spostato di nodo in nodo. In modo analogo a quanto accade nelle tecnologie di accesso ai database, un cursore è una struttura di controllo che permette l'attraversamento dei record in un documento, e quindi il processamento sequenziale delle informazioni che altrimenti in un documento memorizzato con il formato VTD sarebbero sparse.

Un insieme di record di VTD, ciascuno dei quali ha una lunghezza fissa di 64 bit, sono memorizzati sotto forma di array e costituiscono l'intera rappresentazione di un documento XML. La combinazione di questi due fattori, ovvero il record di lunghezza fissa e l'insieme di record memorizzati in un array, causa una drastica riduzione dell'overhead dovuto all'allocazione in memoria di oggetti.

Un esempio di uso di questo modello di parsing è mostrato nell'estratto di codice seguente:

VTDParser.java

```
public void parse(byte[] xmlDocument) {

    VTDGen vg = new VTDGen();
    vg.setDoc(xmlDocument);
        vg.parse(true);  // set namespace awareness to true

        VTDNav vn = vg.getNav();
        AutoPilot ap = new AutoPilot(vn);
        ap.selectElement("firstName");

        while(ap.iterate()){
                int indexFirstName = vn.getText();
                if (indexFirstName!=-1) {
            String firstName = vn.toString(t);
                    System.out.println(firstName);
        }

        }

}
```

Questo metodo riceve in input l'array di byte rappresentante il documento XML. Per prima cosa si usa l'oggetto VTDGen per eseguire il parsing del documento, ovvero generare l'insieme dei VTD che lo indicizzano. Da quest'oggetto è possibile ottenere gli oggetti AutoPilot e VTDNav. Il cursore è contenuto

all'interno dell'oggetto VTDNav. L'oggetto AutoPilot pilota il cursore del VTDNav per navigare nel documento, e permette di ottenere la posizione di un certo elemento all'interno dello stesso, e successivamente di ottenere il nodo stesso.

2.1.5 Differenze tra DOM, SAX, StAX e NEP

Una comparazione tra i modelli tree-based, event-based e non-extractive parsing è efficientemente sintetizzata nella tabella seguente, liberamente adattata da [20].

Funzionalità	StAX	SAX	NEP	DOM
Tipo di API	Streaming Pull	Streaming Push	In memoria (array)	In memoria (albero)
Efficienza CPU/Memoria	Buona	Buona	Buona	Variabile
Lettura all'indietro	No	No	Sì	Sì
Scrittura XML	Sì	No	Sì	Sì

Da questa tabella si evince facilmente qual è il principale svantaggio del modello tree-based e del non-extractive parsing rispetto a quello event-based, ovvero l'efficienza in termini di occupazione di memoria e uso della CPU. Come abbiamo detto infatti, nei modelli tree-based viene costruita una rappresentazione ad oggetti completa del documento non appena viene iniziata la lettura dello stesso. Questa rappresentazione può avere dei costi, in termini di memoria occupata, inaccettabili già per messaggi di media dimensione. Anche nel modello non-extractive il documento viene tenuto totalmente in memoria, ma in una forma molto più efficiente di quella usata dal tree-based.

D'altro canto il principale svantaggio del modello event-based è dato dal fatto che l'input viene consumato e può essere letto solo in avanti, e quindi per eseguire la maggior parte delle operazioni è comunque necessario costruire il modello ad oggetti (manualmente) riconducendoci alle problematiche di efficienza del DOM. Infine, tipicamente le API che si basano sul modello SAX

non permettono di produrre e scrivere XML, ma solo di leggerlo, a differenza degli altri modelli in cui questo è possibile.

2.2 Validazione messaggi XML tramite XML Schema Definition (XSD)

La validazione XSD è il processo di controllo di conformità di un documento XML alle regole definite in uno specifico schema XSD. Il primo passo, affinché un documento sia valido rispetto ad uno schema XSD, è che esso sia un documento XML *ben formato*. Questa condizione è verificata dall'operazione di parsing, in analogia a quanto avviene nei linguaggi formali. Rispetto al parsing la validazione XSD rappresenta un passo avanti, in quanto restringe ulteriormente l'insieme di documenti validi a quelli conformi alla specifica di tipi definita nell'XSD. In altre parole se il parsing equivale ad una validazione sintattica del documento rispetto alla grammatica che definisce i documenti XML, uno schema XSD definisce concettualmente un typesystem e la validazione XSD svolge il ruolo di *type-checking*.

L'importanza della validazione XSD è evidente già nelle applicazioni che usano i documenti XML come database, in quanto permette di determinare la validità della stessa. Tuttavia è nelle applicazioni basate su Web Services che la validazione XSD diventa un processo di fondamentale importanza, dal momento che validare il messaggio in ingresso significa assicurarsi che l'oggetto in questione appartenga alla classe desiderata, dato che i messaggi SOAP scambiati tra client e server non sono altro che una rappresentazione degli oggetti usati dall'applicazione.

2.3 Ricerca all'interno di messaggi XML (Query XPath)

Come spiegato in precedenza, un documento XML è assimilabile ad un albero costituito da nodi. Ciascun nodo può avere dei figli, ed esiste un nodo radice del quale, direttamente o indirettamente, tutti gli altri nodi sono figli. XPath è un linguaggio utilizzato per identificare particolari porzioni di documenti XML. Esso può individuare nodi e insiemi di nodi all'interno dell'albero. XPath è in grado di indicare se un nodo *verifica* l'espressione XPath, ovvero soddisfa la condizione data da quella espressione, in base alla sua posizione assoluta o relativa, al suo tipo, al suo contenuto e in base a molti altri criteri.

Di seguito è presentato un insieme produzioni che definiscono una grammatica XPath. Per la lista completa delle produzioni si rimanda alla grammatica presente sul sito del W3C [21].

⟨*PathExpr*⟩ ::= (‘/’ | ‘//’ | ‘/’ | ‘ϵ’) | ⟨*RelativePathExpr*⟩

⟨*RelativePathExpr*⟩ ::= ⟨*StepExpr*⟩ ((‘/’ | ‘//’) ⟨*StepExpr*⟩)*

⟨*StepExpr*⟩ ::= (⟨*ForwardStep*⟩ | ⟨*ReverseStep*⟩ | ⟨*PrimaryExpr*⟩) ⟨*Predicates*⟩

⟨*ForwardStep*⟩ ::= (⟨*ForwardAxis*⟩ ⟨*NodeTest*⟩) | ⟨*AbbreviatedForwardStep*⟩

⟨*ForwardAxis*⟩ ::= child:: | descendant:: | attribute:: | self::
| descendant-or-self:: | following-sibling::
| following:: | namespace::

⟨*AbbreviatedForwardStep*⟩ ::= ‘.’ | (@⟨*NameTest*⟩) | ⟨*NodeTest*⟩

⟨*ReverseStep*⟩ ::= (⟨*ReverseAxis*⟩ ⟨*NodeTest*⟩) | ⟨*AbbreviatedReverseStep*⟩

⟨*ReverseAxis*⟩ ::= parent:: | ancestor:: | preceding-sibling::
| preceding:: | ancestor-or-self::

⟨*AbbreviatedReverseStep*⟩ ::= ‘..’

⟨*NodeTest*⟩ ::= ⟨*QName*⟩ | ⟨*Wildcard*⟩

⟨*Wildcard*⟩ ::= '*' | ⟨*NCName*⟩:* | *:⟨*NCName*⟩

⟨*PrimaryExpr*⟩ ::= ⟨*Literal*⟩ | ('$'⟨*VarName*⟩) | ⟨*ParenthesizedExpr*⟩

⟨*Literal*⟩ ::= ⟨*NumericLiteral*⟩ | ⟨*StringLiteral*⟩

⟨*ParenthesizedExpr*⟩ ::= (⟨*Expr*⟩ ?)

⟨*Predicates*⟩ ::= ([⟨*Expr*⟩]*)

Nella notazione usata in questa grammatica, il simbolo *?* denota un elemento opzionale, mentre il simbolo | denota la possibilità di scegliere, in presenza di più elementi separati da esso, arbitrariamente uno di questi. Il simbolo non terminale iniziale di qeusta grammatica è *PathExpr*, che definisce un'espressione XPath. Questa consiste in un *RelativePathExpr*, preceduto da un simbolo / o // o da nessun simbolo. Nel primo caso, il *RelativePathExpr* dovrà necessariamente essere figlio del nodo radice del documento, nel secondo dovrà esserne semplicemente un discendente. Nel terzo caso il path si riferisce al contesto corrente (sottoalbero), non al documento totale. Un *RelativePathExpr* è una sequenza di *StepExpr*, tra le quali può intercorrere una relazione padre-figlio o antenato-discendente anche qui a seconda della presenza del simbolo / o //.

Uno *StepExpr* può essere un *ForwardStep*, un *ReverseStep*, o una *PrimaryExpr*, seguito da un *Predicate*. Il *ForwardStep* e il *ReverseStep* rappresentano, rispettivamente, un passo in avanti o all'indietro nel documento, e sono formati da un *Axis* e un *NodeTest* o da uno step abbreviato(*AbbreviatedForwardStep* o *AbbreviatedReverseStep*). L'*Axis* rappresenta la relazione che intercorre tra due *Step*, e il *NodeTest* restringe il campo di ricerca con un test sul nodo corrente. Il *Predicate* restringe a sua volta il campo di ricerca con una o più espressioni che devono essere verificate nel nodo finale.

Sono state definite due notazioni, una abbreviata, compatta, che permette la realizzazione di costrutti intuitivi e facilmente realizzabili, e una completa, più complessa, che contiene maggiori opzioni, in grado di specificare con più particolarità gli elementi.

Un esempio di espressione XPath nella sintassi completa è:

- /child::A/child::B/child::C

che seleziona gli elementi C che sono figli degli elementi B che sono figli dell'elemento A che rappresenta la radice del documento XML. Espressioni più complesse possono essere costruite specificando un Predicate, ad esempio:

- child::A/descendant-or-self::B/child::node()[1]

questa espressione seleziona il primo elemento ([1]), figlio di B, indipendentemente dal suo nome, e dal numero di nodi che intercorrono tra A e B (*descendant-or-self*). Da notare che l'espressione non inizia con /, quindi A è un nodo del contesto corrente. Negli esempi sopra, per ogni *StepExpr*, l'*Axis* è specificato esplicitamente, seguito dal test del nodo, come A o node().

Nella sintassi abbreviata, i due esempi qui sopra sarebbero scritti:

- /A/B/C
- A//B/*[1]

Come si può notare dalla grammatica precedentemente descritta, alcune di queste query hanno bisogno di una conoscenza globale del messaggio per essere eseguite. In particolare, le produzioni derivabili da *ReverseStep* hanno bisogno, per essere risolte, di informazioni posizionate in un punto precedente del documento. Lo stesso avviene per la produzione *following-sibling*, che identifica il prossimo nodo, a partire dal nodo attuale, che abbia lo stesso padre. In questo caso anche se dobbiamo identificare un nodo posizionato in un punto successivo del documento, abbiamo bisogno di un'informazione (il padre) che si trova in un punto precedente del messaggio. Questo vincolo fa sì che queste query possano essere risolte solo usando dei parser che permettano di mantenere il documento in memoria e navigarne il contenuto, escludendo automaticamente i parser SAX o StAX che, per la loro natura streaming, possono leggere il documento solo in avanti. Questi potranno essere dunque utilizzati solo nel caso che l'informazione ricercata non comprenda le produzioni elencate.

XPath è alla base di molti altri linguaggi per la manipolazione di dati XML. Ad esempio XSLT, un linguaggio che permette di creare documenti di vario tipo a partire da documenti XML, utilizza espressioni XPath per cercare corrispondenze con particolari elementi del documento di input che debbano essere copiati nel documento di output oppure sottoposti a elaborazioni ulteriori.

2.4 Parsing SOAP

Abbiamo visto in precedenza che il parsing XML si occupa di trasformare l'input XML in modo che esso possa essere fruibile a livello applicativo. Il parsing SOAP estende questo concetto alla struttura del messaggio SOAP.

Le tecnologie che implementano un parser SOAP permettono di navigare la struttura del messaggio tramite una serie di API con cui è possibile creare, modificare o accedere agli elementi presenti nell'Header e nel Body del messaggio SOAP, oltre a gestire anche eventuali attachments da allegare al messaggio SOAP.

Inoltre, dato lo stretto legame che intercorre tra messaggi SOAP e Web Services, le tecnologie che implementano un parser SOAP spesso forniscono anche facility per la gestione delle comunicazioni da e verso un Web Service, gestendo aspetti come la connessione, la serializzazione e la spedizione del messaggio verso un endpoint.

2.5 Archiviazione del messaggio

L'archiviazione del messaggio è una funzionalità che si occupa di salvare un messaggio SOAP processato, e conservarlo ad esempio in un database, come traccia dell'avvenuta ricezione ed elaborazione dello stesso. Non sempre è richiesta l'archiviazione del messaggio totale, ma in alcuni specifici casi può esserne richiesta l'archiviazione selettiva di alcune parti, come ad esempio dei metadati di trasporto, o di informazioni relative all'header. Un altro caso

è quello di un messaggio SOAP with Attachments, nel quale non sempre è necessario archiviare tutti gli attachments, ma potrebbe essere necessario archiviarne solo una parte, oppure l'archiviazione degli stessi potrebbe essere disabilitata.

2.6 WS-Security

Nell'ambito delle tradizionali applicazioni client-server, i dati scambiati tra le applicazioni erano resi sicuri rendendo sicura la comunicazione tra il client e il server a livello di trasporto, tipicamente tramite la tecnologia SSL. Nell'ambito Web Service, le tradizionali tecnologie di sicurezza non sono sufficienti a causa dell'esigenza di rendere sicuri i dati scambiati con una grana molto più fine. Infatti, dal momento che i Web Services usano tecnologie basate sui messaggi per effettuare transazioni complesse ed interdominio, un messaggio Web Service può attraversare diversi intermediari prima di raggiungere la sua destinazione.

Dal momento che alcuni di questi intermediari potrebbero avere l'esigenza di ispezionare le informazioni non sensibili presenti nel messaggio è necessario prevedere dei meccanismi che permettano di cifrare o firmare anche solo le parti sensibili del messaggio, lasciando inalterate e quindi ispezionabili le altre, escludendo la possibilità di usare un approccio che preveda la cifratura del canale di trasporto. Per far fronte a questa esigenza nasce la specifica WS-Security. Questa spcifica, conosciuta anche come Web Services Security e definita dal consorzio Organization for the Advancement of Structured Information Standards (OASIS) [22] in [23] definisce un'estensione di SOAP che implementa autenticazione, integrità e confidenzialità a livello messaggio. L'obiettivo di questa specifica non è introdurre nuove tecniche, ma quello di utilizzare le soluzioni esistenti in materia di sicurezza delle comunicazioni con SOAP ed i Web Service, per garantire una sicurezza *end-to-end*. Il messaggio viene in questo caso arricchito da uno o più header (a seconda dell'aspetto da gestire) e alcuni nodi vengono di conseguenza modificati o sostituiti.

Di seguito sono presentati i requisiti centrali che un'infrastruttura di comunicazione che implementa WS-Security deve fornire.

2.6.1 Verifica dell'identità del mittente

Uno dei requisiti centrali di un'infrastruttura di comunicazione è la possibilità di fornire e ricevere referenze attendibili sull'identità dei soggetti coinvolti nella comunicazione. Queste referenze sono informazioni aggiuntive che trovano naturale collocazione nell'header SOAP del messaggio.

Ci sono molteplici modi per fornire prove della propria identità. Un primo metodo è quello di fornire una coppia di credenziali, Username/Password, da allegare al messaggio, inserendola nell'header SOAP. Essendo il messaggio SOAP ispezionabile, tale metodo non è sufficiente da sé, ma deve essere accompagnato ad una cifratura delle credenziali. Un metodo alternativo si basa sull'uso di certificati e di una Public Key Infrastructure (PKI). PKI è un insieme di tecnologie che consentono a terze parti fidate di farsi garanti dell'identità di un utente, tramite la presenta di una gerarchia di specifiche autorità di certificazione, Certificate Authority (CA), abilitate all'emissione, sospensione e revoca dei certificati verso gli utenti, e che si fa garante della corrispondenza tra il certificato posseduto da un utente e l'utente stesso.

2.6.2 Integrità del messaggio e non ripudio

Con il processo di autenticazione viene verificata l'identità del soggetto con cui vengono scambiate informazioni. Questo però non è sufficiente a garantire che le informazioni inviate dal mittente siano le stesse ricevute dal destinatario. Non viene quindi garantita l'integrità del messaggio. Questa garanzia può essere raggiunta tramite l'applicazione della firma ai dati scambiati.

WS-Security si appoggia alla specifica XML Signature per firmare un messaggio. Un'impronta del messaggio, chiamata *digest*, viene cifrata utilizzando

la chiave privata del mittente ed inserita in un header SOAP, e la verifica consiste nel confronto del digest (precedentemente decifrato utilizzando la chiave pubblica del mittente) con il digest calcolato dal destinatario. Una volta che un messaggio è stato firmato, ogni modifica allo stesso comporta una discrepanza tra i due digest. La firma assicura quindi al ricevente che le parti firmate del messaggio non siano state modificate dopo la firma, e nel contempo che il soggetto che ha apposto la firma sia lo stesso identificato dal certificato.

2.6.3 Confidenzialità

Un ulteriore requisito di sicurezza è quello di assicurare la confidenzialità delle informazioni scambiate, ovvero impedire che dati sensibili possano essere letti da soggetti non autorizzati. Lo scopo di questo requisito è la possibilità di cifrare il contenuto del messaggio, o parti di esso, di modo che solo il destinatario sia in grado di decifrarlo e leggerlo. Anche in questo caso WS-Security si appoggia ad una specifica preesistente, l'XML Encryption.

Per quanto riguarda la codifica dei dati, la scelta è tra la codifica simmetrica e quella asimmetrica. La prima richiede la condivisione di un'informazione segreta tra le parti, dal momento che la chiave usata per cifrare è la stessa usata per decifrare. Questa soluzione è efficace se le parti in causa operano in un contesto di cui si detiene il controllo, ma pone il problema della distribuzione delle chiavi. Qualora questo aspetto causasse dei problemi, si può utilizzare la codifica asimmetrica. Anche la cifratura a chiave asimmetrica si basa sull'uso di certificati e di una PKI, usati in maniera analoga a quanto già visto per il caso dell'integrità del messaggio ma nella cifratura il mittente usa la chiave pubblica del destinatario per cifrare e il destinatario usa la sua chiave privata per decifrare.

Capitolo 3

Funzionamento di un nodo SOAP

In questo capitolo si analizzeranno le problematiche relative al processamento dei messaggi SOAP, descritte nel Capitolo 2, collocandole nel particolare contesto dei componenti applicativi che prendono il nome di nodi SOAP. Verrà inizialmente descritta la tipica architettura di un nodo SOAP, seguita da un'analisi sulle differenti tecniche studiate per affrontare i casi d'uso che rappresentano le problematiche comuni a questo tipo di architettura. Il risultato dell'analisi verrà utilizzato per affrontare, nei successivi capitoli, le criticità relative ad una particolare istanza di nodo SOAP. la Porta di Dominio OpenSPCoop.

3.1 Architettura

Di seguito è descritta l'architettura tipica di un nodo SOAP, concentrandosi su come un'architettura di questo genere affronti le problematiche presentate nel Capitolo 2. Questa descrizione rappresenta un'astrazione di un reale nodo SOAP, funzionale all'analisi sui casi d'uso affrontata di seguito.

Un'architettura di questo tipo sarà tipicamente uno dei nodi intermedi in una comunicazione in ambito Web Service, e quindi riceverà un messaggio

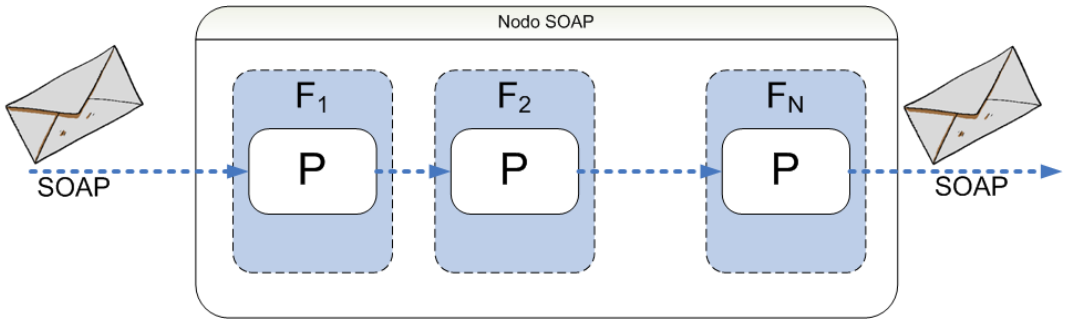


FIGURA 3.1: Architettura astratta di un nodo SOAP

SOAP in ingresso, e dovrà sottoporlo ad una serie di elaborazioni prima di poterlo inoltrare in uscita verso il nodo successivo. Una descrizione di tale architettura è sintetizzata in Figura 3.1.

In tale architettura ciascuna elaborazione viene eseguita fa una unità funzionale F, che per eseguirla incapsulerà al suo interno un parser P. Come vedremo di seguito, la scelta di un tipo di parser rispetto ad un altro determinerà una differenza nelle prestazioni dell'unità funzionale, e quindi del nodo stesso.

3.2 Casi d'uso

Di seguito verrà effettuata un'analisi sulle problematiche esaminate nel Capitolo 2 e su come esse vengano affrontate dai diversi tipi di parser. Per ogni problematica sarà presentata una panoramica di casi d'uso evidenziando per ciascuno quale sia il parser più adatto alla sua gestione. Questa panoramica sarà indipendente dal linguaggio di programmazione usato, e quindi anche dall'implementazione del parser effettivamente utilizzata. In questo senso è utile suddividere i modelli di parsing in due insiemi. Il primo insieme è formato dai parser che costruiscono una struttura in memoria, sia essa un albero di oggetti (come succede nel modello DOM) o una lista di indici (come nel NEP), e permettono quindi successivamente di accedere all'intera struttura e navigarla completamente. Questo tipo di parser sarà riferito successivamente come parser *in memory*. Il secondo insieme è formato dai parser che permettono il processamento del messaggio in streaming, come i parser di tipo SAX e StAX,

e permettono quindi la lettura del messaggio solo in avanti, e causano la perdita di tutte le informazioni relative ad esso una volta consumato lo stream. Questo tipo di parser sarà riferito sucessivamente come parser *streaming*.

3.2.1 Validazione XSD

Come visto nel Capitolo 2, validare un documento XML rispetto al suo Schema XSD significa controllare che la struttura del documento sia valida in accordo a quanto dichiarato nello Schema. Un parser di tipo *in memory* permetterà di effettuare la validazione attraverso l'esame della struttura dalla quale è composto e la verifica della corrispondenza tra questa e lo Schema dato. Al contrario un parser di tipo *streaming* dovrà tenere in memoria informazioni sugli elementi attesi in un certo punto del documento e, via via che questo viene letto, tenere traccia degli elementi che realmente vi appaiono. Se durante la scansione dovesse apparire un elemento non dichiarato oppure un elemento richiesto dovesse mancare, l'elaborazione si bloccherà immediatamente.

Le differenze tra i due tipi di elaborazione porteranno inevitabilmente ad una differenza di performance, che è opportuno analizzare per diversi casi d'uso. Un primo esempio è quello di un documento di 500 byte, valido rispetto allo Schema rispetto al quale viene eseguita la validazione. Date le dimensioni limitate del documento, l'overhead della costruzione in memoria della sua struttura, effettuata dai parser di tipo in memory, è limitato. D'altro canto i parser di tipo streaming pagheranno in questo caso il fatto di dover aggiornare continuamente il set di informazioni aggiuntive durante la lettura del documento. Questo potrebbe portare al fatto che il modello in memoria ha prestazioni migliori di quello streaming per messaggi così piccoli.

Un altro esempio è un messaggio da 10 Gigabyte, anche in questo caso valido rispetto allo Schema con cui viene eseguita la validazione. In questo caso l'overhead di memoria sarà decisamente più alto (come analizzato nel Capitolo 2 la dimensione della struttura crescerà con una velocità più che lineare rispetto alla dimensione del documento stesso), e potrebbe saturare la

memoria riservata all'applicazione. Il modello streaming in questo caso invece tenderà ad avere prestazioni migliori in quanto esso leggerà il documento a gruppi di pochi byte alla volta, e quindi la memoria occupata non dipenderà dalla dimensione del documento processato.

3.2.2 Ricerca XPath

Come visto nel Capitolo 2, la ricerca XPath serve ad effettuare delle ricerche all'interno di un documento XML. Un parser di tipo in memory permetterà di effettuare la ricerca attraverso la navigazione della struttura dalla quale è composto fino al nodo ricercato. Al contrario un parser di tipo streaming dovrà tenere in memoria informazioni sulla posizione in cui si trova e, via via che questo viene letto, verificare se l'espressione XPath è verificata in quel punto. Di norma, nel contesto di un nodo SOAP, non è interessante l'insieme di risultati che verificano la ricerca, ma solo il primo, in quanto questo viene utilizzato per pilotare la scelta del prossimo nodo a cui inoltrare il messaggio. Per questo motivo non appena il primo risultato viene trovato, l'elaborazione si arresta.

Come succedeva per la validazione XSD, anche in questo caso un parser di tipo in memory avrà prestazioni migliori per ricerche all'interno di documenti piccoli, dove sfrutterà la velocità con cui può trovare il risultato una volta costruito il modello, di dimensioni limitate, in memoria. Al contrario i parser di tipo streaming nel caso di messaggi di dimensione limitata pagano la costruzione di un motore che, per permettere la valutazione dell'espressione XPath senza bufferizzare l'intero messaggio, risulta meno efficiente. L'opposto succede per messaggi di dimensione elevata, nei quali i parser in memory pagheranno la crescita lineare di memoria rispetto alla crescita di dimensione del documento, rispetto ai parser streaming che sfrutteranno il vantaggio dell'occupazione di memoria costante.

La dimensione del documento non è però l'unico parametro da considerare, nella scelta del modello di parsing più appropriato da utilizzare nei vari casi

d'uso: un esempio sta nel considerare una serie di documenti di dimensione tale per cui, considerando l'esempio precedente, il modello in memory e il modello streaming abbiano prestazioni simili, e nei quali vari la posizione dell'elemento cercato all'interno del documento.

Prendendo in considerazione vari casi, in cui l'elemento cercato si trova più o meno vicino all'inizio del documento, vedremo che le prestazioni dei modelli in memory saranno costanti, al contrario di quelle dei modelli streaming. Questo è dovuto al fatto che i primi costruiscono in ogni caso l'intera struttura del documento e successivamente eseguono la ricerca. Essendo la costruzione della struttura l'operazione che richiede più risorse, i risultati non varieranno sostanzialmente. Al contrario, un parser che utilizzi il modello streaming processerà ogni porzione di documento appena ne avrà letto i byte, e potrà quindi sapere in qualsiasi punto della computazione se si è trovato l'elemento ricercato o meno. Di conseguenza in questo caso la computazione potrà essere arrestata non appena sarà stato trovato l'elemento, permettendo un miglioramento delle prestazioni nel caso in cui esso sia più vicino all'inizio del documento.

Un ulteriore caso d'uso interessante è quello di ricerche che hanno bisogno di una conoscenza globale del messaggio, delle quali abbiamo parlato nel Capitolo 2.1.5. In questo caso il parser dovrebbe bufferizzare parte del messaggio per poterlo leggere nuovamente in un punto successivo della scansione, e quindi un parser streaming non può essere usato per risolvere la query, restringendo così il campo dei parser disponibili per questo caso d'uso a quelli di tipo in memory.

3.2.3 Archiviazione del messaggio

Come visto nel Capitolo 2, l'archiviazione del messaggio permette il salvataggio in un repository di un messaggio elaborato dal nodo SOAP, come traccia della sua elaborazione o per un successivo riuso. Anche per questa funzionalità è possibile individuare vari casi d'uso, nei quali è più appropriato usare un parser rispetto ad un altro. Il primo caso è quello di un messaggio che deve

essere salvato in maniera totale ed indiscriminata: in questo caso un modello streaming permette di salvare i byte così come vengono letti, senza avere alcuna conoscenza sul loro significato. La stessa cosa può essere fatta mediante un parser in memory semplicemente navigandone la struttura e riversandone il contenuto in un output stream. Come visto per gli altri casi d'uso, le prestazioni nei due casi saranno dettate dal diverso uso della memoria che fanno i diversi parser. In questo caso però, a differenza degli altri, un parser streaming avrà sempre prestazioni migliori di un parser in memory, in quanto non dovrà compiere alcuna operazione aggiuntiva.

Un altro caso d'uso è quello del salvataggio parziale di un messaggio: potrebbe ad esempio essere significativo salvare solo un header, o una specifica parte del body. In questo caso non sarà possibile utilizzare un parser streaming, in quanto esso non ha alcuna conoscenza sulla struttura del messaggio, mentre un parser in memory può sfruttare in questo senso la struttura che ha costruito. Lo stesso discorso può essere fatto per il caso dell'archiviazione parziale di un messaggio contenente attachments: in questo caso si avrà bisogno di un parser SOAP, che a differenza dei parser XML permette di accedere agli attachments, e quindi di archiviarli in maniera selettiva.

3.2.4 WS-Security

Abbiamo visto che la WS-Security si occupa di rendere sicura la comunicazione tra due nodi SOAP, modificando i messaggi scambiati per garantire features come l'autenticazione, la confidenzialità e la riservatezza delle informazioni scambiate. Per fare questo il nodo mittente dovrà aggiungere degli header e sostituire dei nodi al messaggio che riceve prima di inviarlo al nodo destinatario. Dal momento che i nodi da aggiungere sono degli header, e che essi vengono creati usando informazioni che si trovano in un punto arbitrario del messaggio, questo deve essere letto più volte durante la computazione. Per questo motivo il processamento in streaming in questo caso è impossibile da attuare, e di conseguenza si dovrà ricorrere ad un parser XML in memory o un parser SOAP.

I casi d'uso individuati per questa funzionalità riguardano l'applicazione della firma o della cifratura a specifiche parti del messaggio, come l'intero body, o uno specifico campo contenente un dato sensibile, o un attachment. La scelta su quale sia il parser più adatto da utilizzare in questo caso non riguarda strettamente il parsing, dal momento che un parser SOAP includerà al suo interno un parser XML. Essendo l'unica possibilità quella di utilizzare un parser in memory, le performance dipenderanno dalla specifica implementazione del parser utilizzato.

La discriminante tra l'uso di un parser XML o SOAP in questo caso è analoga a quella discussa a proposito dell'archiviazione del messaggio, ed è legata all'esigenza di cifrare o firmare eventuali attachment ricevuti assieme al messaggio SOAP. Questa esigenza restringe la scelta al parser di tipo SOAP, mentre in caso contrario il parser in memory rimane un'alternativa praticabile.

3.3 Criticità nell'implementazione di un nodo SOAP

Come visto nella parte di presentazione dell'architettura astratta di un nodo SOAP, esso deve implementare tutte le funzionalità di cui abbiamo parlato, e di fatto tutti i principali nodi SOAP descritti in precedenza (XML Firewall, PdD OpenSPCoop, Service Bus) incorporano funzionalità di questo tipo. Il limite di un nodo SOAP realizzato in accordo all'architettura astratta presentata in questo capitolo è dato dal fatto che essa non permette l'utilizzo di un diverso parser a seconda della specifica tipologia di messaggio, ma forza le unità funzionali a utilizzare sempre lo stesso parser. Come abbiamo visto nel corso dell'analisi dei casi d'uso, le varie problematiche sono gestite in maniera ottimale da parser diversi, e quindi un'architettura che utilizzi indiscriminatamente lo stesso parser si rivela intrinsecamente inadatta a gestirle in maniera efficiente. La necessità che si viene a creare è quindi quella di un architettura che incorpori al suo interno vari parser per ogni funzionalità e che permetta,

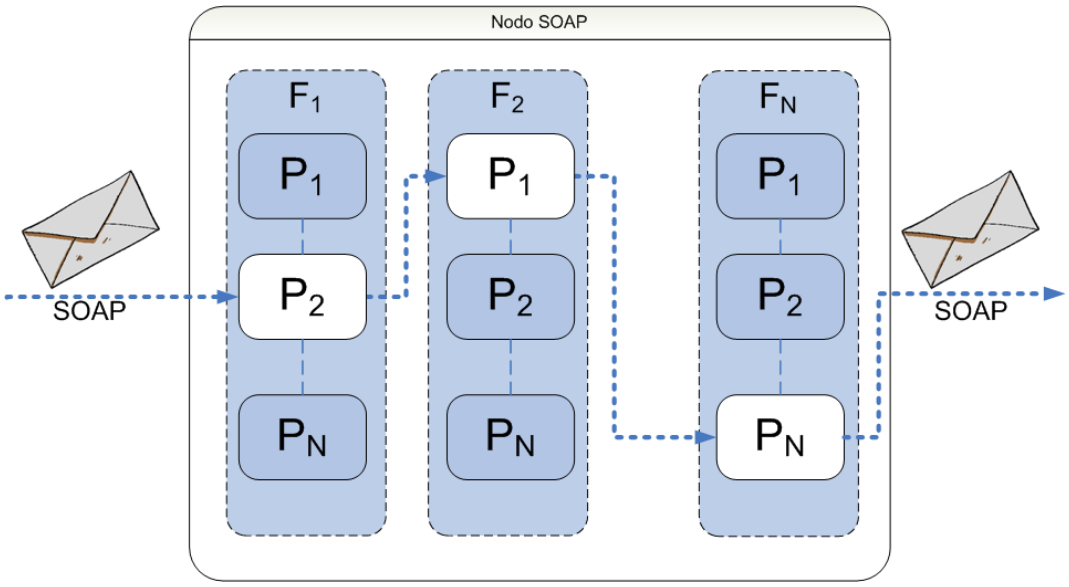


FIGURA 3.2: Architettura astratta di un nodo SOAP, configurabile

in maniera flessibile per ogni caso d'uso, di utilizzare il più adatto tra questi parser. Per ottenere un'architettura di questo tipo è quindi necessario modificare quella presentata nel Capitolo 3.1 in modo che essa si comporti come mostrato in Figura 3.2.

La differenza tra questa architettura e quella precedente sta nel fatto che, mentre nella prima le unità funzionali erano forzate ad utilizzare lo stesso parser, in quest'ultima i parser disponibili per ogni funzionalità sono molteplici. In questo modo è quindi possibile, a seconda dello specifico messaggio ricevuto in ingresso e in funzione delle diverse elaborazioni da eseguire sul messaggio, istanziare il parser più adatto al suo processamento.

3.4 Il caso della Porta di Dominio OpenSPCoop

Nei successivi capitoli l'analisi fin qui eseguita verrà contestualizzata in un caso specifico di nodo SOAP, la Porta di Dominio (PdD) OpenSPCoop. In particolare nel prossimo capitolo verranno analizzate le tecnologie che rappresentano lo stato dell'arte per quanto riguarda la gestione delle problematiche comuni

ad un nodo SOAP, oggetto dell'analisi effettuata in questo capitolo. Essendo la PdD implementata in linguaggio Java, le tecnologie analizzate saranno anch'esse realizzate con questo linguaggio di programmazione. Successivamente, nel Capitolo 5, verrà descritto nel dettaglio come vengono gestite tali funzionalità nell'architettura attuale della PdD, implementata seguendo lo schema descritto in 3.1, dopodiché verrà proposta una nuova architettura che rispecchi invece l'architettura astratta presentata in 3.3, in modo da rendere la gestione del messaggio flessibile e permettere l'utilizzo del parser più adatto a seconda del caso d'uso.

Capitolo 4

Analisi delle tecnologie allo stato dell'arte

In questo capitolo verranno presentate le tecnologie esaminate per affrontare le problematiche emerse nel capitolo precedente. Essendo l'obiettivo quello di ottimizzare le prestazioni della PdD OpenSPCoop, che è scritta in Java, tutte le soluzioni esaminate sono anch'esse scritte con questo linguaggio di programmazione.

4.1 Validazione XSD

Di seguito verranno presentate alcune API che effettuano la validazione XSD. La versione 1.5 del Java Development Kit (JDK) fornisce due modi di effettuare la validazione XSD di un documento XML.

4.1.1 JDK/Xerces (Parser DOM)

Un primo modo di eseguire la validazione XSD consiste nell'uso delle classi del package *javax.xml.validation* fornito da Xerces ed incluso nella JDK 1.5, in particolare la classe *Validator*. Questa classe fornisce le API per la validazione a partire dal DOM di un documento, ovvero espone un metodo che prende in

ingresso il DOM di un documento e ne esegue la validazione rispetto ad uno specifico schema, che deve essere stato specificato in precedenza.

Questa API permette di validare il documento rispetto a vari tipi di schema, ad esempio DTD o XSD. Il comportamento dell'API rispetto agli eventuali errori di validazione rilevati è reso configurabile mediante l'uso di un Error Handler, che si comporta in maniera analoga a quanto visto durante il trattamento del parsing SAX.

4.1.2 JDK/Xerces (Parser SAX)

Un secondo modo per eseguire la validazione XSD è quello di utilizzare la classe SAXParser, fornito da Xerces ed incluso nella JDK 1.5, di cui abbiamo già parlato durante il trattamento del parsing SAX. All'interno di questa classe è integrata una funzionalità di validazione, da abilitare su richiesta. Anche in questo caso è possibile specificare uno schema, ed abilitare un Error Handler per la gestione dei casi di errore.

La validazione viene eseguita in maniera analoga a quanto visto in caso di parsing, con la differenza che, se si deve eseguire solo la validazione, è opportuno disabilitare qualsiasi Content Handler per rendere la computazione più efficiente.

4.2 Ricerca XPath

Di seguito è presentata una panoramica sulle diverse API per eseguire la ricerca XPath.

4.2.1 JDK/Xerces (Parser DOM)

Il primo, e più comune, modo per eseguire la ricerca XPath su un documento è quello di utilizzare le API fornite da Xerces e integrate nella JDK 1.5, precisamente nelle classi del package *javax.xml.xpath*. Questa libreria si aspetta in ingresso il DOM relativo al documento su cui eseguire la ricerca, e restituisce il nodo o l'insieme di nodi per i quali la query è verificata.

Esistono molte tecnologie al di fuori della JDK che, per effettuare la ricerca XPath in maniera più efficiente, basano anch'esse la ricerca sulla costruzione del DOM, ma promettono di migliorare le prestazioni mediante l'uso di un DOM non standard, e più *leggero*. Un esempio di questo tipo di librerie è SAXON [24]. Le tecnologie di questo tipo sono state scartate dopo la fase di studio perché costruiscono il modello ad oggetti, seppur meno invasivo rispetto al DOM standard, e quindi risultano egualmente inefficienti.

4.2.2 VTD (Parser NEP)

Una possibile alternativa alla costruzione del DOM durante la ricerca è quella di utilizzare la funzionalità integrata all'interno della libreria VTD-XML che utilizza un parser di tipo NEP e quindi non costruisce il modello ad oggetti ma agisce su un array di byte che rappresenta la rappresentazione testuale del messaggio.

Tra le caratteristiche di VTD-XML ci sono:

- Il supporto all'accesso casuale a tutte le informazioni del documento.
- Un'implementazione XPath integrata all'interno della libreria.
- Il parsing che, grazie alla codifica VTD, risulta essere 5-10 volte più veloce del DOM e 1.5-2 volte più veloce del più efficiente parser SAX.

- La memoria richiesta, sempre grazie alla codifica VTD, è 1.3-1.5 [19] volte la dimensione dell'XML originale, con una riduzione del 30-45% rispetto al DOM.

4.2.3 SXC (Parser StAX)

Un'ulteriore alternativa è rappresentata dalla libreria SXC [25], che esegue la ricerca XPath agendo sullo stream di dati in transito, senza mantenere il messaggio in memoria né costruire il DOM. SXC non permette solo di eseguire ricerche XPath, ma più in generale di agganciare degli event handler allo stream XML in transito, i quali reagiscono ad uno specifico evento che causa la chiamata di un metodo definito dall'utente. Nel caso dell'XPath l'event handler deve essere agganciato ad una specifica query. Il metodo chiamato avrà a disposizione l'oggetto puntatore posizionato sul punto del messaggio in cui la query è risultata verificata e a questo punto sarà compito del programmatore gestire in maniera appropriata il fetch del risultato, così come l'eventuale interruzione del processamento o la costruzione del nodo corrispondente.

Il fatto che SXC utilizzi al suo interno un parser di tipo StAX ha come diretta conseguenza il fatto che essa non è capace di eseguire tutte le query XPath, ma solo un sottoinsieme (peraltro molto significativo). Come descritto nel Capitolo 2, SXC può eseguire le query che non hanno bisogno di avere una conoscenza globale del messaggio.

4.3 Parsing SOAP

Di seguito verranno presentate alcune API per eseguire il parsing SOAP.

4.3.1 SAAJ (Parser DOM)

Soap with Attachments API for Java (SAAJ) [26] è un insieme di API che permettono di creare, modificare e spedire messaggi SOAP in Java. Il modello

di parsing usato da queste API per elaborare l'XML è di tipo DOM. Le API di SAAJ permettono ad un client di inviare un messaggio direttamente al destinatario usando un oggetto di tipo SOAPConnection, che fornisce una connessione sincrona *point-to-point* con il destinatario. Tali API permettono dunque al programmatore di disinteressarsi degli aspetti di basso livello nella gestione dei messaggi, nonché della comunicazione tra client e server in una comunicazione basata sui messaggi SOAP.

L'attuale versione di SAAJ, 1.3, è conforme a SOAP 1.2 e SwA [11], quindi può essere utilizzato per elaborare messaggi SOAP con o senza allegati. SAAJ è il parser SOAP utilizzato nel progetto Apache Axis [27]. Axis è un Web Service Framework, ovvero una struttura di supporto alla creazione dei software sviluppati in ambito Web Service. Questo tipo di framework si occupa della ricezione, elaborazione e replica di messaggi scambiati tra agenti erogatore e fruitore, nell'ambito di una comunicazione client-service, aspetti in questo caso gestiti tramite le API di SAAJ. Inoltre Axis fornisce varie facilities per la creazione e il deploy di un Web Service a partire dalle interfacce Java che lo definiscono.

4.3.2 AxiOM (Parser DOM/StAX)

L'evoluzione di Axis è il progetto Apache Axis2 [28], un'implementazione basata su Java sia della parte client che di quella server del modello Web Service. Progettato partendo dall'esperienza maturata dal progetto Axis, Apache Axis2 fornisce un Object Model completo e un'architettura modulare che permette di aggiungere facilmente funzionalità e supporto a nuove specifiche collegate ai Web Service.

Il parser SOAP qui utilizzato è l'Axis Object Model, ovvero AxiOM [2, 3, 29, 30]. AxiOM è un parser SOAP che si propone di fondere il modello event-based e quello tree-based. Esso si basa sulle API StAX per l'input e l'output dei dati, ma vi apporta una decisiva innovazione con il supporto alla costruzione del modello differita. AxiOM infatti non costruisce l'intero modello

ad oggetti una volta iniziata la scansione del documento, come avviene ad esempio con SAAJ, ma ritarda la costruzione fino a quando non è necessaria. Ciò significa che, fino a quando un certo nodo non è richiesto, esso rimane nello stream, pronto ad essere letto. Quando il nodo viene richiesto, la parte di DOM relativa viene costruita. Inoltre AxiOM può generare eventi StAX partendo da qualsiasi punto del documento ricevuto, indipendentemente dal fatto che sia stato processato o meno. Ancora, la costruzione del modello in AxiOM può essere disabilitata, permettendo così di usare AxiOM come un parser StAX. La grande flessibilità di AxiOM fa sì che sia possibile configurarne il motore per ottenere una maggiore efficienza a seconda del tipo di operazione da effettuare.

4.4 WS-Security

Di seguito verranno presentate alcune API per applicare la WS-Security ad un messaggio SOAP.

4.4.1 WSS4J (Parser DOM)

WSS4J [31] è un'implementazione Java della specifica OASIS WS-Security fornita da Apache. WSS4J può essere usata per firmare e cifrare messaggi SOAP con informazioni WS-Security, e quindi per mettere in sicurezza un Web service. WSS4J implementa:

- OASIS Web Services Security: SOAP Message Security 1.0 Standard 2004-01, March 2004
- Username Token profile V1.0
- X.509 Token Profile V1.0

Per la natura stessa della specifica WS-Security, il processamento dei diversi aspetti che costituiscono l'applicazione delle specifiche, come ad esempio la firma, la cifratura o l'autenticazione, è modulare e ciascun aspetto deve essere

gestito in maniera autonoma dagli altri. WSS4J implementa questa specifica mediante i *Processor*. Ogni Processor può essere visto come una black box che riceve in input il messaggio SOAP originario e restituisce in output il messaggio risultante dall'applicazione dell'aspetto gestito. Più Processor possono essere concatenati in una pipeline per gestire più aspetti della WS-Security in maniera coordinata. Essendo i Processor delle entità general purpose, e dal momento che la specifica WS-Security permette una grande varietà di scenari (lascia ad esempio libertà sulla scelta dell'algoritmo di cifratura o firma da utilizzare) particolare importanza assume l'aspetto della configurazione di WSS4J. Quali Processor utilizzare, e le opzioni di configurazione degli stessi, sono specificati tramite una serie di proprietà da passare ad un Handler che si occuperà di configurare ogni Processor in maniera corrispondente.

4.4.2 Soapbox (Parser SOAP)

Soapbox è un toolkit che fornisce API per la gestione della WS-Security incluso dalla compagnia AdroitLogic nell'Enterprise Service Bus UltraESB e rilasciato con licenza Open Source. Il toolkit è scarsamente documentato, ed è stato studiato e modificato a partire dal solo codice sorgente. Il funzionamento del toolkit è per molti aspetti riconducibile a quello di WSS4J. Anche qui sono presenti varie entità Processor deputate alla gestione di un aspetto della WS-Security.

Una differenza notabile tra WSS4J e Soapbox è che il primo utilizza un parser DOM, mentre il secondo utilizza un parser SOAP. La differenza tra i due tipi di parser, come visto nel Capitolo 2, è data dal fatto che il secondo permette di accedere anche agli attachment ricevuti assieme al messaggio SOAP. Avere a disposizione anche gli attachment ha dato la posibilità di aggiungere di una serie di funzionalità assenti in WSS4J, ovvero quelle relative alla firma e cifratura degli attachment, implementate secondo le specifiche di [11].

Capitolo 5

La Porta di Dominio OpenSPCoop

In questo capitolo verrà effettuata un'analisi dell'attuale architettura della PdD OpenSPCoop, dal punto di vista del trattamento del messaggio, per evidenziare vantaggi e limiti di tale architettura rispetto a quanto visto nel Capitolo 3. Successivamente verrà presentata una proposta di nuova architettura, che renderà possibile una gestione flessibile del messaggio SOAP, ed i plugin realizzati per rendere efficiente tale gestione.

5.1 Architettura attuale

Di seguito viene presentata un'analisi dell'attuale architettura della PdD OpenSPCoop, analizzandone i principali vantaggi e svantaggi dal punto di vista del trattamento del messaggio SOAP.

5.1.1 Analisi dell'attuale architettura

Essendo la specifica SPCoop piuttosto vasta, e di conseguenza l'implementazione di OpenSPCoop complessa e ricca di sfaccettature, non verrà esposta in questa sede ma si rimanda alla documentazione citata in precedenza. Verrà

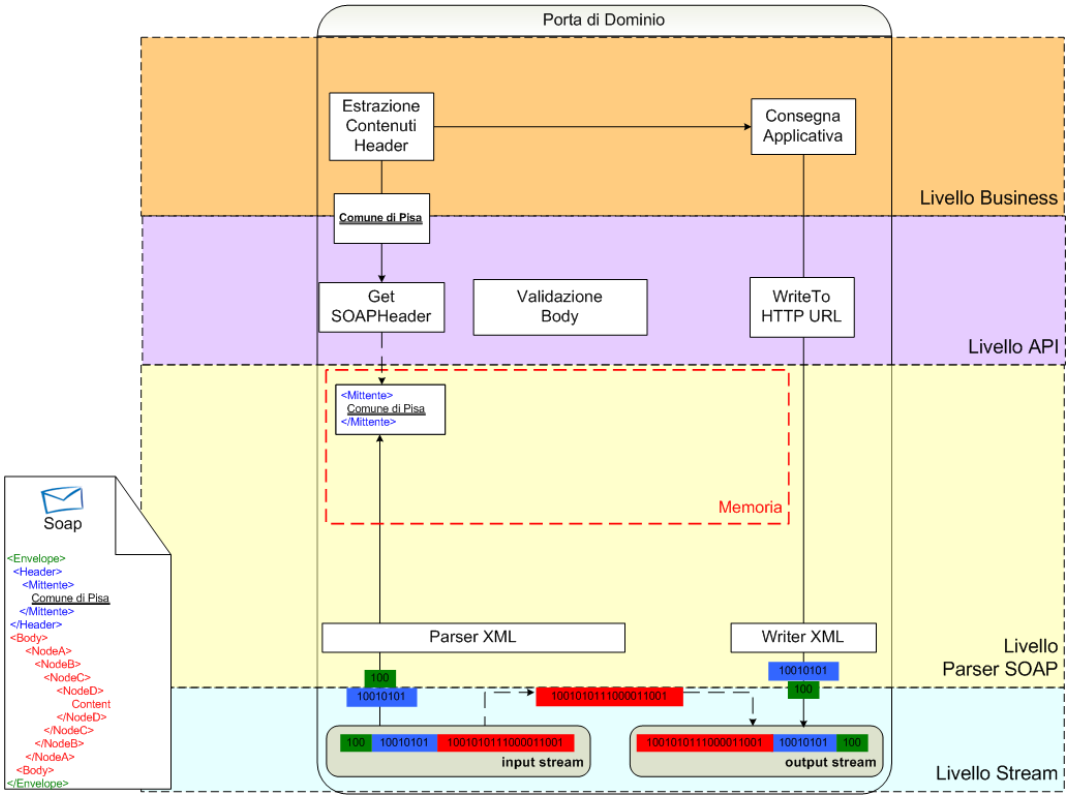


FIGURA 5.1: PdD che esegue solo l'estrazione di contenuti dall'header

qui trattata solo la parte di architettura deputata al trattamento dei messaggi SOAP, contenuta nella parte *core* della Porta di Dominio OpenSPCoop. Di seguito viene presentata l'architettura della PdD dal punto di vista del trattamento del messaggio, presentando i due casi d'uso principali.

Il primo caso d'uso è quello di una PdD configurata per recepire il messaggio, analizzarne informazioni contenute nell'header, e consegnarlo al prossimo nodo applicativo. In questo caso la PdD si comporta come schematizzato in Figura 5.1:

Come si può vedere, la PdD è strutturata in quattro livelli:

- *Stream*: gestisce e inoltra al livello superiore i byte grezzi ricevuti in input, e riceve dallo stesso i byte da riportare in output.
- *SOAP Engine*: serializza e deserializza i byte ricevuti e da inviare al livello Stream, trasformandoli in una rappresentazione SOAP.

- *API*: espone metodi di facility per l'accesso in lettura e scrittura dei messaggi SOAP.

- *Business*: riceve il messaggio, ne esegue la validazione dell'header e lo consegna al successivo nodo applicativo.

L'aspetto su cui è importante focalizzare l'attenzione in questo caso è il livello SOAP Engine: come si può vedere dall'immagine infatti, avendo il livello Business bisogno solo di informazioni contenute nell'header, il DOM del messaggio viene costruito solo parzialmente, grazie all'uso di AxiOM da parte del livello SOAP Engine. AxiOM infatti, come abbiamo visto, ha tra le sue caratteristiche quella di essere un ibrido tra un parser DOM e un parser StAX. Esso infatti, pur offrendo le API dello stesso tipo di quelle offerte dai parser DOM, permette di ottimizzare la gestione della memoria attraverso la costruzione differita del modello. Questo vantaggio viene sfruttato in casi d'uso come quello appena presentato. Come si può vedere dall'immagine, mentre l'header passa dal parser e dal writer XML, il body rimane nello stream in attesa di essere semplicemente rediretto in output. In questo caso il vantaggio dell'utilizzo di AxiOM rispetto ad un parser DOM è evidente: qualsiasi sia la dimensione totale del messaggio, il prezzo pagato in termini di memoria sarà solo quello relativo alla costruzione del modello dell'header, la cui dimensione è tipicamente molto inferiore a quella totale del messaggio.

Il secondo caso d'uso è quello di una PdD configurata per recepire il messaggio, analizzarne informazioni contenute sia nell'header che nel body, e consegnarlo al prossimo nodo applicativo. In questo caso la PdD si comporta come schematizzato in Figura 5.2.

Stavolta il livello Business ha bisogno di accedere ad informazioni localizzate sia nell'header che nel body, e quindi le ottimizzazioni relative alla costruzione del DOM parziale nella SOAP Engine non hanno più effetto. Si può facilmente notare la differenza di memoria occupata, dovuta al fatto che anche il DOM del body viene costruito per intero. Questa situazione si verifica

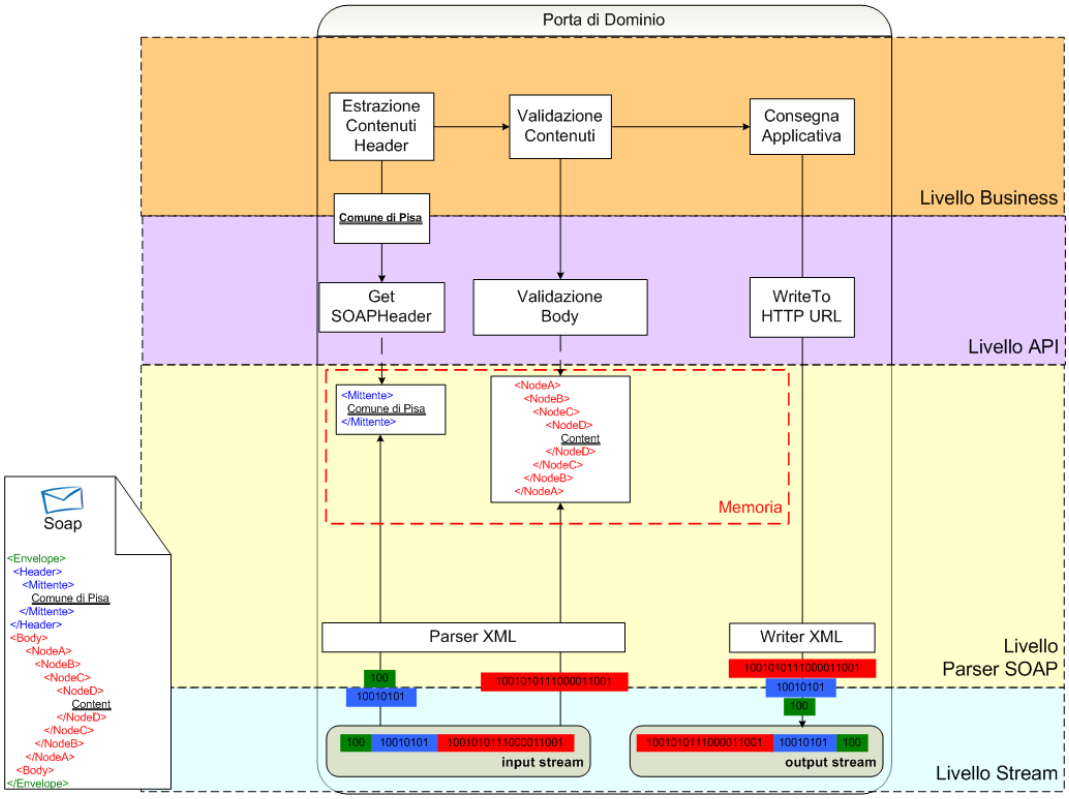


FIGURA 5.2: PdD che esegue validazione contenuti

in tutti i casi in cui sia necessario analizzare l'intero messaggio, come succede in tutti i casi d'uso analizzati nel Capitolo 3.

5.2 Nuova architettura proposta

Il livello su cui è necessario focalizzare l'attenzione per poter eseguire un'analisi delle performance dell'architettura attuale nella gestione dei messaggi SOAP è certamente il livello SOAP Engine. Come analizzato nel Capitolo 3, non esiste un unico parser che ottimizzi le performance in tutti i casi d'uso che interessano un nodo SOAP, ed è quindi necessario utilizzare, a seconda del messaggio, uno specifico parser per la gestione di ogni funzionalità. Basandosi su queste considerazioni si capisce che l'utilizzo indiscriminato di AxiOM come parser non è possibile, e si rende quindi necessaria la ristrutturazione dell'architettura della PdD.

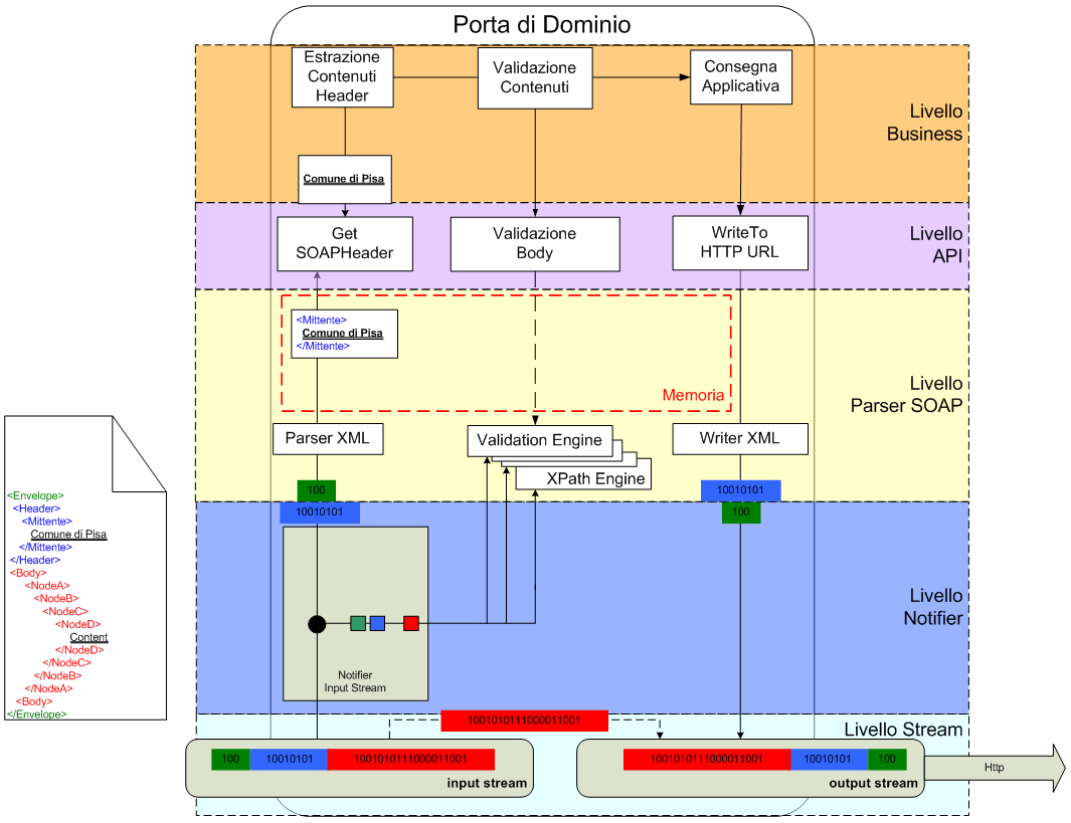


FIGURA 5.3: PdD con nuova architettura che esegue validazione contenuti in modalità streaming

La nuova architettura è stata realizzata in conformità al modello descritto in Figura 3.2, dove più parser coesistono e possono essere utilizzati alternativamente, a seconda delle esigenze legate allo specifico messaggio ricevuto. L'obiettivo era quello di realizzare la nuova architettura con un impatto minimo rispetto a quella attuale, ed in particolare di mantenere la compatibilità con il livello SOAPEngine attuale, il cui funzionamento dovrà essere quindi garantito anche con la nuova architettura. Questo obiettivo è stato dettato dal fatto che il livello SOAPEngine attuale utilizza AxiOM, che è un prodotto standard, ampiamente supportato e che offre buone prestazioni su un gran numero di casi d'uso. La scelta è stata quella di introdurre un livello intermedio, chiamato livello Notifier, tra i livelli Stream e SOAPEngine, come mostrato in Figura 5.3:

Tale livello si occuperà di leggere i byte e passarli al SOAP Engine, utilizzando uno speciale InputStream, il NotifierInputStream, che arricchisce l'InputStream standard con alcune funzionalità. Innanzitutto è dotato di una funzionalità di *notifica*: ad ogni chiamata dell'operazione di read, da parte del livello SOAP Engine, il NotifierInputStream notificherà i byte letti a degli *Engine Streaming* aggiuntivi, che dovranno eseguire una specifica operazione in modalità streaming.

La modalità streaming permette di eseguire le operazioni passando il messaggio un byte alla volta al motore sottostante. Tale modalità si basa su un motore dotato di una logica interna capace di eseguire l'operazione in maniera incrementale, ovvero senza conoscere a priori tutto il contenuto del messaggio, ed ha il vantaggio di essere completamente indipendente dalla dimensione del messaggio processato. Infatti il messaggio sarà passato al motore a gruppi di pochi byte alla volta, permettendo alla JVM di avere poca memoria occupata durante l'operazione. Lo svantaggio è che per alcune operazioni (in particolare per un particolare insieme di ricerche XPath viste nel Capitolo 2) non è possibile utilizzare questa modalità, in quanto è richiesta una conoscenza globale del messaggio.

Per rendere possibile che ciascun Engine abbia a disposizione tutti i byte via via letti dallo stream, è stato necessario ridefinire il metodo *read* in modo che, tra la lettura effettiva del byte (chiamata al metodo read della superclasse), e restituzione del byte letto al chiamante, il byte sia passato a ciascun Engine Streaming configurato. In questo modo si potrà decidere quali operazioni abilitare semplicemente aggiungendo gli Engine Streaming desiderati, rendendo così la PdD altamente configurabile in base alle esigenze. L'InputStream così personalizzato potrà essere infatti usato dal livello SOAP Engine che scorrerà lo stream in maniera indipendente. Questo aspetto è mostrato graficamente dai byte che vengono utilizzati per alimentare gli Engine Streaming che risultano essere gli stessi che vengono passati al livello SOAP Engine. Come si può vedere in questo caso l'operazione API di Validazione Body andrà a leggere il

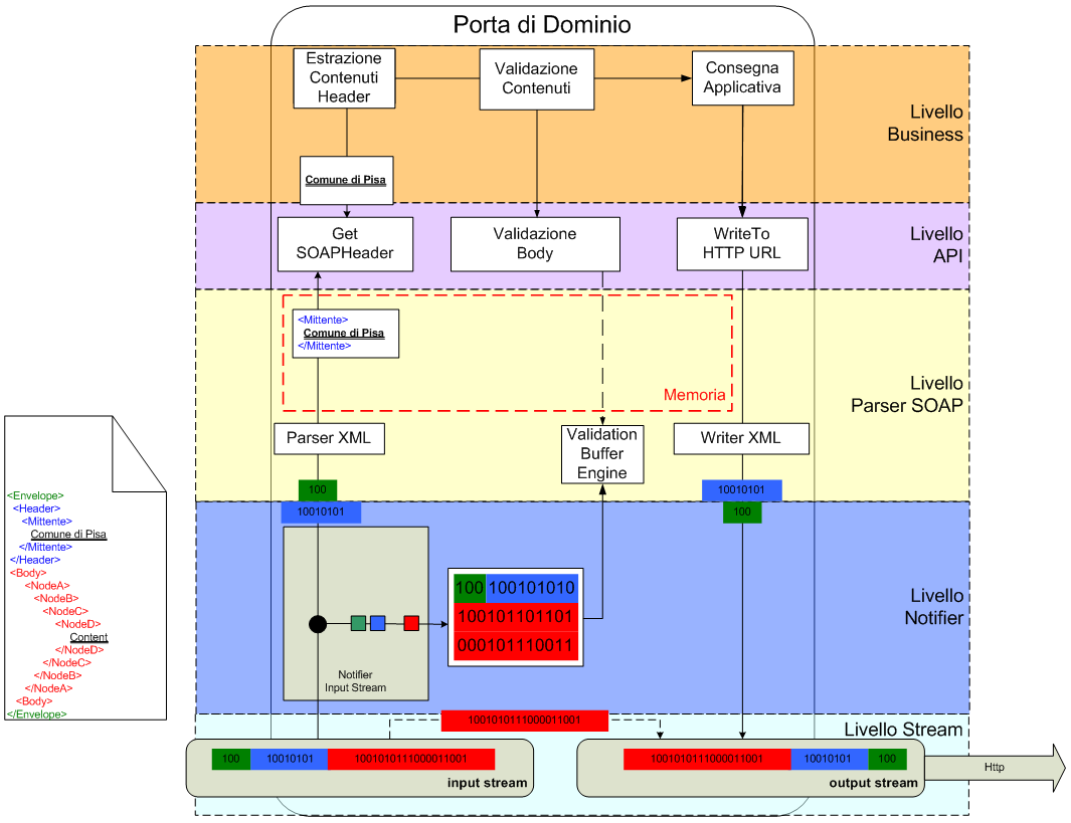


FIGURA 5.4: PdD con nuova architettura che esegue validazione contenuti in modalità buffer

risultato dall'Engine Streaming appropriato, invece che utilizzare il DOM del Body come succedeva con l'uso di AxiOM.

Un'ulteriore evoluzione del NotifierInputStream è che esso può essere abilitato per il salvataggio del messaggio. Questo significa che sarà possibile impostare uno speciale Engine Streaming con l'unica funzionalità di memorizzare ogni byte letto. Al termine della lettura l'array di byte rappresentante il messaggio sarà disponibile e potrà essere utilizzato, oltre che per un'eventuale archiviazione, come input per gli *Engine Buffer*, che utilizzeranno la modalità buffer. La modalità buffer permette di eseguire le operazioni passando al motore l'intero messaggio. Un esempio d'uso di questa modalità è mostrato in Figura 5.4:

Come si può vedere, in maniera analoga alla versione streaming, l'operazione API di Validazione Body legge il risultato dall'Engine Buffer, che riceve dal livello Notifier l'intero buffer del messaggio, così come viene ricevuto in

ingresso.

La modalità buffer è meno vantaggiosa di quella streaming dal punto di vista delle prestazioni, perché l'occupazione di memoria cresce linearmente al crescere della dimensione del messaggio processato. Tuttavia l'occupazione di memoria risulta essere migliore di quella del DOM, e ci sono dei vantaggi anche rispetto alla modalità streaming: ad esempio un motore di questo tipo è spesso più snello del suo omologo streaming, e ha prestazioni migliori per messaggi di piccole dimensioni (come vedremo in fase di benchmark). Inoltre in questo modo si possono eseguire anche le operazioni che richiedono una conoscenza globale del messaggio, come alcune tra le ricerche XPath citate nel Capitolo 2.

5.2.1 Modalita di identificazione del messaggio

Come visto in precedenza, la principale innovazione apportata dalla nuova architettura proposta rispetto a quella attuale riguarda la possibilità di poter scegliere in maniera flessibile il parser a seconda dello specifico messaggio. Per ottenere questo comportamento è necessario quindi introdurre nell'architettura un componente che si occupi di identificare la tipologia di messaggio ricevuto, per indirizzarlo verso la modalità più indicata per la sua gestione. L'implementazione di questo componente non è stata affrontata durante il lavoro di Tesi, ma ne costituisce la naturale continuazione. Di seguito sono presentate alcune ipotesi riguardanti la logica secondo cui questo componente può essere realizzato.

Il lavoro che questo componente deve compiere può essere svolto secondo diverse procedure: il caso ideale sarebbe rappresentato da un componente che scelga il parser migliore in base ad una serie di caratteristiche del messaggio in ingresso, come ad esempio la dimensione del messaggio. Questo però presuppone che il componente esegua un'analisi del messaggio ricevuto prima di iniziarne il trattamento. Essendo questa un'operazione che introduce ulteriore overhead, si può pensare ad un'alternativa che scelga il parser rispetto al servizio invocato. Questa scelta rappresenta una buona approssimazione della

prima, in quanto i messaggi diretti ad uno stesso servizio tipicamente avranno le stesse caratteristiche. A differenza della prima soluzione però si evita l'overhead dell'analisi del messaggio.

Ogni servizio è composto da vari *endpoint*, che rappresentano i punti di accesso alla PdD, e corrispondono alle diverse URL di invocazione della stessa. Ogni endpoint è fornito di una configurazione, che contiene le informazioni necessarie ad identificare il servizio a cui l'endpoint è abbinato, ed altre informazioni come ad esempio il tipo di autenticazione/autorizzazione richiesto. Tale configurazione può essere estesa allo scopo di includere le opzioni per la scelta del parser adeguato per ogni funzionalità, eventualmente disabilitando alcune funzionalità per specifici endpoint. Quando la PdD (e con essa i suoi *endpoint*) viene avviata, tale configurazione verrà letta e di conseguenza potranno essere inizializzati i parser necessari alla corretta gestione del messaggio.

5.2.2 Plugin realizzati

5.2.2.1 Interfacce Streaming e Buffer

Per rendere possibile l'integrazione tra gli Engine, che processano il messaggio in modalità *streaming* e *buffer*, e la PdD sono state create due interfacce, una per il trattamento in streaming dei messaggi, e una per il trattamento dell'intero messaggio una volta bufferizzato. L'interfaccia IStreaming, che dovrà essere implementata da tutti gli Engine che lavorano in modalità streaming, è mostrata di seguito:

IStreaming.java

```
public interface IStreaming {

    /**
     * feeds each engine, one byte a time
     * @param b byte to feed the engine with
     * @throws IOException
```

```
     */
    public void feed(int b) throws IOException;

}
```

L'interfaccia IBuffer, implementata da tutti gli Engine che lavorano in modalità buffer, è mostrata di seguito:

IBuffer.java

```
public interface IBuffer {

    /**
     * feeds engine with the whole message, encoded in a byte
    array
     * @param data the whole message encoded in a byte array
     * @throws IOException
     */
    public void evaluate(byte[] data) throws IOException;

}
```

Gli Engine streaming, rispetto a quelli buffer, si trovano a dover risolvere l'ulteriore problema della ricostruzione dello stream. Nel sistema precedentemente descritto infatti i dati vengono passati a gruppi di byte, non essendo possibile passare l'intero Input Stream originario, che altrimenti non sarebbe utilizzabile dal SOAP Engine. Si è reso necessario quindi implementare una pipe di stream, ovvero una coppia di stream (uno di input e uno di output) connessi tra loro. I due stream sono eseguiti su due thread diversi, e sono configurati in modo che ogni byte scritto su quello di output venga poi reso disponibile su quello di input. In questo modo il thread principale dell'Engine scrive i byte ricevuti dalla *read* su un OutputStream, e questi verranno resi disponibili come InputStream nell'altro thread, in modo da essere utilizzabili dal motore sottostante. L'implementazione di questo meccanismo è stata affidata

alle classi del package *java.util.concurrent*, in particolare all'interfaccia Callable e alle classi ad essa correlate, che rispondono all'esigenza di eseguire in un thread separato delle operazioni che restituiscano un valore o lancino un'eccezione, operazione impossibile nativamente con la classe Thread. Le classi usate sono presenti nella JDK dalla versione 1.5, e si rimanda alla documentazione della stessa per approfondimenti.

5.2.2.2 Engine di Validazione XSD

Di seguito viene presentata l'interfaccia di validazione XSD, che deve essere implementata dagli Engine di validazione.

IXSDEngine.java

```
public interface IXSDEngine {

    /**
     * gets result of validation
     * @return true if message is valid against specified
    schema
     */
    public boolean isValid();

    /**
     * gets details about error in validation, if error occurs
     * @return details about errors in validation, or null if
    no error occurs
     */
    public String getErrors();

}
```

Sono state fornite due implementazioni di questa interfaccia, una delle quali lavora in modalità streaming e l'altra in modalità buffer. Entrambe eseguono la validazione utilizzando lo stesso sistema, ovvero utilizzando le facility già presenti nella JDK 1.5 grazie alle quali, come descritto nel capitolo precedente,

si può eseguire la validazione di un messaggio a partire dallo stream passato in ingresso. La differenza nelle due implementazioni è data dal fatto che, mentre nel primo caso i byte vengono passati uno per uno al motore, mediante il sistema di pipe di stream descritto sopra, nel secondo caso l'array di byte ricevuto in ingresso verrà usato per inizializzare un Input Stream, che verrà poi passato in un'unica soluzione al motore. Questa sottile differenza causerà dei differenti comportamenti e influenzerà le performance delle due implementazioni, come vedremo in maniera più approfondita nel capitolo riguardante il benchmark.

L'inizializzazione delle classi che eseguiranno la validazione è affidata ad una factory esterna, sia per rendere il funzionamento dell'Engine indipendente dall'implementazione usata che per rendere possibile un meccanismo di caching. Questa scelta è stata dettata dal fatto che effettuare l'inizializzazione degli oggetti solo una volta nell'intero ciclo di vita dell'applicazione invece che per ogni messaggio trattato, permette di risparmiare risorse e si traduce in un abbassamento del tempo necessario al processamento del messaggio. Questo è di particolare importanza nell'ottica di un'applicazione pensata per gestire un gran numero di messaggi, eventualmente anche in multithreading. Il meccanismo di caching implementato è basilare, ma permette di migliorare sensibilmente le prestazioni dell'applicazione. Esso prevede semplicemente che la factory sia arricchita da una mappa avente come chiave una stringa rappresentante il nome del file XSD da usare come Schema, e come valore l'oggetto di Evaluator XSD, che incapsula la funzionalità di validazione. La prima volta che l'Engine richiede alla factory un nuovo oggetto Evaluator XSD passando in input il nome del file XSD da usare come Schema, questa andrà a cercarlo nella cache. Essendo la prima volta che alla factory viene richiesto l'oggetto, questo non verrà trovato e sarà quindi necessario crearlo ed inserirlo in cache. Nelle successive chiamate la ricerca dell'oggetto in cache andrà a buon fine, evitando il costo di inizializzazione dell'oggetto.

Il motore di validazione è di tipo SAX, e lancerà quindi un'eccezione al primo errore riscontrato. Sarà l'Engine a gestire quest'eccezione ed a valorizzare

correttamente i campi di validità ed errore. Lo scopo dell'Engine infatti non è quello di bloccare l'esecuzione e restituire un errore alla PdD in risposta ad un'errore di validazione, in quanto la PdD può essere configurata con diverse politiche, tra le quali ci può essere quella di non segnalare gli errori di validazione o di segnalarli con un warning ma procedere comunque con l'inoltro al nodo successivo.

5.2.2.3 Engine di Ricerca XPath

Di seguito viene presentata l'interfaccia di Ricerca XPath, che deve essere implementata dagli Engine di ricerca:

IXPathEngine.java

```
public interface IXPathEngine {

    /**
     * gets result of XPath query
     * @return a String representation of XPath query result
     * @throws IOException
     */
    String getResult() throws IOException;

    /**
     * sets the query string for XPath search
     * @param query object for XPath search
     * @throws IOException
     */
    void setQuery(XPathQuery query) throws IOException;

}
```

Anche in questo caso sono state fornite due implementazioni dell'interfaccia, una delle quali lavora in modalità streaming e l'altra in modalità buffer. L'implementazione streaming utilizza il toolkit SXC, mentre l'implementazione buffer utilizza il toolkit VTD-XML, entrambi descritti nel capitolo precedente. In

questo caso le implementazioni usate sono naturalmente aderenti alle interfacce descritte: SXC si aspetta un Input Stream in ingresso, che verrà costruito col sistema di pipe di stream, mentre VTD si aspetta un array di byte. In questo caso il problema è che le API di entrambi i toolkit sono molto di basso livello, ed è stato necessario quindi del lavoro aggiuntivo per estrarre effettivamente le informazioni richieste dai dati forniti dai toolkit. L'obiettivo era infatti quello di poter configurare l'Engine per far rendere disponibili i risultati sotto forma di stringa o di nodo, mentre quello che fanno nativamente i due toolkit è localizzare il token desiderato all'interno del messaggio.

SXC infatti effettua la scansione del messaggio tramite un parser StAX, e di conseguenza agirà sull'oggetto XMLEventReader, come abbiamo visto in precedenza. Al verificarsi della query XPath, l'oggetto XMLEventReader verrà passato all'EventHandler deputato alla gestione dell'evento di riconoscimento della query. In quel momento l'XMLEventReader ha localizzato il token desiderato, e sarà quindi possibile tramite i suoi metodi recuperare le informazioni desiderate per costruire il nodo da restituire. Con VTD-XML invece è possibile, dopo aver passato al motore l'array di byte rappresentante il messaggio, richiamare un metodo che restituirà un cursore che punta alla posizione dove si trova il token desiderato, dopodiché sarà necessario ricostruire il nodo corrispondente. Ciò è possibile in quanto abbiamo a disposizione il buffer e le informazioni aggiuntive date dai Virtual Token Descriptor.

Anche in questo caso l'inizializzazione delle classi che eseguiranno la ricerca è affidata ad una factory esterna: i motivi sono analoghi a quelli che hanno portato alla scelta di una factory per gli Evaluator XSD, ma in questo caso c'è un ulteriore motivo: abbiamo detto infatti che non tutte le query XPath sono disponibili in modalità streaming, dal momento che alcune sono intrinsecamente dipendenti dal contesto. Se per la ricerca XPath è stato configurato un Engine streaming, ma la query che si desidera risolvere non è disponibile in questa modalità, il costruttore di SXC lancerà un'eccezione, che verrà gestita a livello di cache. La politica adottata in questo caso è tale che il sistema di caching

provvederà ad inizializzare un Engine buffer, inserirlo nella cache e restituirlo al chiamante, rendendo l'operazione trasparente al resto dell'applicazione.

5.2.2.4 Engine di WS-Security

L'interfaccia per la gestione degli aspetti di WS-Security è divisa in due parti: la prima è l'interfaccia IWSSSender, presentata di seguito:

IWSSSender.java

```
public interface IWSSSender {

    public void process(WSSContext wssContext,
    OpenSPCoop2Message message) throws WSSException;

}
```

e l'interfaccia IWSSReceiver, presentata di seguito:

IWSSReceiver.java

```
public interface IWSSReceiver {

    public void process(WSSContext wssContext,
    OpenSPCoop2Message message,Busta busta) throws
    WSSException;

    public String getCertificate() throws WSSException;

}
```

L'implementazione dell'interfaccia fornita fa uso del toolkit Soapbox, incluso nell'ESB UltraESB, e ne estende le funzionalità. L'implementazione originale prevedeva delle funzionalità molto limitate di cifratura e firma del SOAPBody, ma le prestazioni su queste funzionalità erano molto incoraggianti. Dal momento che, per quanto riguarda la cifratura e la firma, il toolkit prevedeva solo l'applicazione delle funzionalità al messaggio totale, mentre l'obiettivo era di poterle applicare anche solo a specifiche parti del messaggio, il

lavoro svolto è stato un lavoro di estensione delle API. Sono state create due nuove classi, una per la gestione della cifratura, e una per la gestione della firma, configurabili in modo tale da permetterne l'applicazione ad uno o più elementi del messaggio. Inoltre, come già discusso in precedenza, Soapbox utilizza un parser SOAP e quindi lavora sulla SOAPPart, a differenza di WSS4J che, utilizzando un parser DOM, lavora sul SOAPEnvelope. La differenza tra i due approcci come abbiamo già visto è che il parser SOAP permette di applicare le funzionalità citate non solo agli elementi del SOAPEnvelope, ma anche agli attachments che eventualmente potranno essere inviati assieme ad esso. Pur utilizzando un parser di questo tipo, Soapbox non includeva alcuna funzionalità di cifratura o firma degli attachments. Anche queste funzionalità sono state quindi implementate ed integrate nelle classi che gestiscono la firma e cifratura degli elementi.

Soapbox è basato sull'interfaccia Processor, presentata di seguito:

Processor.java

```
public interface Processor {
    public void process(SecurityConfig secConfig,
   MessageSecurityContext msgSecCtx);
}
```

L'unico metodo esposto è in questo caso il metodo process, che prende in ingresso un SecurityConfig, contenente le informazioni di configurazione di WS-Security, e un MessageSecurityContext, che contiene il messaggio su cui applicare la WS-Security. Un processor servirà ad applicare un aspetto della WS-Security, ad esempio firma, cifratura o timestamp. Più Processor verranno concatenati tra loro a seconda delle opzioni di configurazione specificate per uno specifico servizio. Sostanzialmente la porta leggerà la configurazione per un servizio e la passerà alla classe istanziata come IWSSSender (o IWSSReceiver, in caso di ricezione di un messaggio). A questa classe è demandata l'interpretazione dei parametri di configurazione, l'istanziazione dei Processor a seconda dei parametri e la chiamata del metodo process su ognuno di essi,

che determina la conseguente trasformazione del messaggio.

Lavorando sulla SOAPPart, Soapbox agisce necessariamente sul DOM del messaggio, come già avveniva nella precedente versione che utilizzava WSS4J come toolkit. Le prestazioni di Soapbox, come vedremo, sono mediamente superiori a quelle di WSS4J, ed inoltre ora viene supportata anche la gestione degli attachments. Al momento dello sviluppo della tesi non esistevano tecnologie in grado di gestire la WS-Security senza costruire il DOM, ma al momento della stesura sono stati resi disponibili dei toolkit che implementano le API di XML Encryption e XML Signature, sulla base delle quali sono costruite tutte le API di WS-Security, basate su VTD. A partire da queste nuove API è possibile implementare le API di WS-Security in modalità buffer, e questo rappresenta uno dei principali sviluppi futuri individuati da questo lavoro di Tesi.

Capitolo 6

Benchmark

In questo capitolo saranno descritti i test prestazionali della nuova Porta di Dominio. Dopo una panoramica riguardante il benchmark standard Performance Test Framework di WSO2, utilizzato per ottenere le varie misurazioni, sono state confrontate le prestazioni ottenute dall'attuale architettura con quelle ottenute da quella nuova. Successivamente le prestazioni della nuova architettura sono state confrontate con quelle ottenute dalle altre soluzioni che hanno utilizzato questo benchmark.

6.1 Testsuite standard WSO2

Per eseguire tutti i test di performance è stato utilizzato un benchmark standard, ovvero il Performance Test Framework di WSO2[8]. Il Performance Test Framework è un framework per il test delle prestazioni degli ESB. Concepito nel 2005 con l'intenzione di sviluppare una modalità standard per la valutazione delle prestazioni dei prodotti di tipo ESB, è diventato rapidamente lo standard de facto in questo campo, ed è stato utilizzato per confrontare alcuni tra i principali prodotti in questo ambito. E' possibile testare diversi scenari che coprono un vasto insieme di casi d'uso indirizzati da architetture di questo tipo.

I componenti coinvolti nel benchmark sono l'ESB, un client che invierà dei messaggi all'ESB stesso, e un servizio di backend al quale l'ESB dovrà inoltrare il messaggio ricevuto (eventualmente dopo averci eseguito una qualche computazione). Il servizio semplicemente restituirà in risposta all'ESB un messaggio identico, che verrà inoltrato poi dall'ESB al client nella risposta HTTP. Il numero di messaggi inviati dal client, la dimensione di tali messaggi, e il livello di parallelismo con il quale il client li invia, sono configurabili in modo da valutare gli ESB con il massimo numero di casi d'uso possibile. Gli scenari per i quali vengono eseguiti i test sono i seguenti:

- *Direct Proxy*: misura le prestazioni di un ESB che, ricevuto un messaggio dal client, lo inoltra semplicemente al servizio senza eseguire alcuna computazione aggiuntiva.

- *Content Based Routing Proxy basato su header di trasporto (CBRTransportHeaderProxy)*: misura le prestazioni di un ESB che esegue Content Based Routing, ovvero si basa su informazioni ricevute dal mittente per capire qual è il nodo al quale inoltrare il messaggio. In questo caso le informazioni sono da ricercare tra degli header di trasporto HTTP.

- *Content Based Routing Proxy basato su header SOAP (CBRSOAPHeaderProxy)*: anche questo caso misura le prestazioni di un ESB che esegue Content Based Routing, ma in questo caso la ricerca viene eseguita all'interno dell'header del messaggio SOAP ricevuto.

- *Content Based Routing Proxy (CBRProxy)*: terzo caso di misura delle prestazioni di un ESB che esegue Content Based Routing, con ricerca eseguita in questo caso all'interno del body del messaggio SOAP ricevuto.

- *XSL Transformation Proxy*: misura le prestazioni di un ESB che trasforma il messaggio ricevuto prima di inoltrarlo al nodo successivo.

- *Secure Proxy*: misura le prestazioni di un ESB che mette in sicurezza (tramite WS-Security) il messaggio prima di inoltrarlo al nodo successivo.

Periodicamente viene eseguita una sessione ufficiale di test, alla quale qualsiasi ESB può essere iscritto, e i risultati vengono resi pubblici su internet. I risultati dell'ultima sessione (round 6), eseguita il 3 agosto 2012, possono essere consultati all'indirizzo [32], ed è con questi risultati che verranno confrontati i risultati ottenuti dalla nuova Porta di Dominio.

Non tutti i test eseguiti nel round 6 sono stati eseguiti per la Porta di Dominio. In particolare, sono stati esclusi i test riguardanti *XSL Transformation Proxy*, in quanto tale funzionalità non è supportata dalla Porta di Dominio, e *Secure Proxy*, in quanto per questa funzionalità l'attività della Tesi non è stata prevista alcuna attività di miglioramento prestazionale, ma soltanto le estensioni funzionali descritte nel Capitolo 5.2.2.4, riguardanti la possibilità di mettere in sicurezza gli attachments.

Un'altra differenza rispetto ai test del round 6 riguarda le dimensioni dei messaggi usate. Come si può notare, i test riportati all'indirizzo [32] riguardano messaggi con dimensione che varia tra i 500 Byte e i 100 Kilobyte. Questo intervallo di messaggi non evidenzia le criticità nella gestione dei messaggi di grande dimensione, sia per quanto attiene all'occupazione di memoria, sia per quanto attiene alle performance. Per questo si è deciso di estendere questi test con messaggi di dimensione fino a 100 Megabyte.

6.2 Modalità di esecuzione test

L'ambiente su cui sono stati eseguiti i test prestazionali della nuova Porta di Dominio è lo stesso usato nel round 6, ed è descritto in [32]. Usare esattamente lo stesso ambiente di quello usato nel round 6 è stato possibile grazie alla disponibilità di una macchina virtuale già a disposizione in ambiente Amazon Elastic Compute Cloud [33] tramite l'identificativo unico "ami-09ef4560", preconfigurata con tutti gli ESB utilizzati nei test, il client per l'invio dei messaggi e il servizio di backend. Su questa stessa macchina è stata installata la Porta di Dominio.

Per permettere al client di invocare l'ESB per i vari scenari di test, questa deve esporre un endpoint SOAP diverso per ogni caso testato. In questo modo è possibile configurare il client per invocare l'endpoint (definito da un URL) relativo allo specifico scenario di test. Dopo aver configurato l'ESB in modo che essa goda dello stesso regime utilizzato nel round 6 (come descritto in [32] la memoria allocata agli ESB è pari a 2 Gigabyte, e il numero di thread è approssimativamente pari a 300), ed aver eseguito lo startup dell'ESB, viene eseguita una fase di *warm up* in cui l'ESB viene invocato dal client con un numero limitato di messaggi. Questa fase è utile in quanto tipicamente le prime chiamate ad un ESB risulteranno meno efficienti, e per questo i risultati ottenuti non saranno considerati ai fini del benchmark, in modo che esso mostri le prestazioni che l'ESB otterrebbe a regime. In seguito viene eseguito il test per ogni scenario, con diverse combinazioni di numero di messaggi inviati, dimensioni del messaggio e livello di parallelismo del client. Ogni singolo test viene eseguito tre volte, per ottenere una maggiore robustezza nei risultati. L'output del test sarà costituito dai valori elencati di seguito:

- Dimensione dei messaggi inviati dal client
- Livello di parallelismo del client
- Numero totale di messaggi inviati dal client
- Tempo totale di esecuzione del test
- Numero di richieste gestite con successo
- Numero di richieste gestite in maniera errata
- Numero di richieste che non hanno ottenuto una risposta
- Numero di richieste al secondo gestite con successo

Il valore usato in questo test per confrontare i diversi ESB è quello relativo al numero di richieste al secondo gestite con successo dall'ESB, mediato sulle

tre esecuzioni del test. Inoltre verranno evidenziati, ove significativi, i valori relativi al numero di richieste gestite in maniera errata, e al numero di richieste che non hanno ottenuto risposta, anch'essi mediati sulle tre esecuzioni del test.

6.3 Confronto tra vecchia e nuova architettura

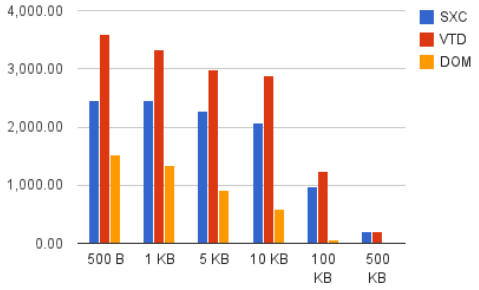
Di seguito verranno presentati i risultati dei benchmark comparativi tra la vecchia e la nuova architettura della Porta di Dominio, mediante i quali è possibile sia valutare i miglioramenti ottenuti con l'inserimento della gestione flessibile del messaggio, in maniera trasparente agli altri aspetti che non sono oggetto di questa Tesi, sia valutare alcuni casi d'uso non inclusi nel Performance Test Framework, ma inclusi nella Porta di Dominio e ottimizzati nell'ambito del lavoro di Tesi.

6.3.1 Ricerca XPath

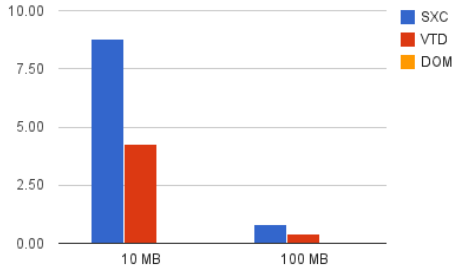
Il primo scenario presentato prevede l'esecuzione di una ricerca XPath sul messaggio prima dell'inoltro dello stesso al nodo successivo. Lo scenario è previsto dal WSO2 Performance Test, ma in questo caso è riportato solo il caso in cui è prevista la ricerca di informazioni contenute nel body del messaggio SOAP. Per questo scenario la nuova architettura è stata testata con due versioni: la prima che lavora in modalità streaming e utilizza un motore di tipo SXC, e la seconda che lavora in modalità buffer e utilizza un motore di tipo VTD. Queste versioni sono state paragonate alla versione presente nell'attuale architettura, che utilizzava un modello DOM. I risultati mostrano il numero di messaggi al secondo che la Porta di Dominio ha gestito in maniera corretta nelle tre versioni, e sono aggregati per dimensione del messaggio e riassunti in Figura 6.1.

Come si può vedere entrambe le versioni realizzate con la nuova architettura si comportano meglio della versione attuale, su tutte le dimensioni dei messaggi. In particolare si può vedere che la versione VTD (che usa la modalità

CBR	SXC	VTD	DOM
500 B	2,457.92	3,601.32	1,524.09
1 KB	2,458.88	3,320.08	1,330.81
5 KB	2,267.88	2,989.84	907.62
10 KB	2,076.59	2,878.76	585.12
100 KB	972.34	1,235.34	56.77
500 KB	203.54	197.01	0.00
10 MB	8.76	4.29	0.00
100 MB	0.80	0.39	0.00



(A) Messaggi piccoli



(B) Messaggi grandi

FIGURA 6.1: CBR nei risultati comparativi tra vecchia e nuova architettura

buffer) ha dei risultati molto migliori per messaggi piccoli, mentre la versione SXC (che usa la modalità streaming) ha prestazioni migliori con messaggi più grandi. Questo rispecchia quanto discusso nel Capitolo 3.2.2, ovvero il fatto che un parser di tipo in memory (come nel caso di VTD) sfrutterà la velocità con cui può trovare il risultato una volta costruito il modello, di dimensioni limitate, in memoria, e quindi avrà prestazioni migliori per ricerche all'interno di documenti piccoli. Viceversa un parser di tipo streaming (come nel caso di SXC) sfrutterà il vantaggio dell'occupazione di memoria costante al crescere

della dimensione del messaggio, per avere prestazioni migliori in questo caso.

6.3.2 Validazione XSD

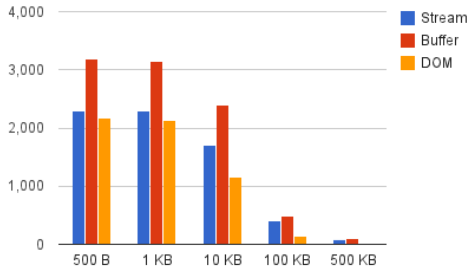
Il successivo scenario presentato è quello di una Porta di Dominio che, ricevuto un messaggio, ne esegua la validazione XSD prima di inoltrarlo al nodo successivo. Nella versione attuale per effettuare questa operazione la Porta di Dominio utilizza il modello DOM, mentre le due versioni proposte per essere utilizzate con la nuova architettura usano entrambe lo stesso modello streaming, ma una di queste prevede la bufferizzazione del messaggio, riconducendosi dunque al caso di un parser in memory, mentre l'altra lo processa in maniera streaming standard. Come già visto per la ricerca XPath, i risultati mostrano il numero di messaggi al secondo che la Porta di Dominio ha gestito in maniera corretta nelle tre versioni, e sono aggregati per dimensione del messaggio e riassunti nella Figura 6.2.

Come si può vedere, anche in questo caso valgono le considerazioni discusse per la ricerca XPath: entrambe le versioni hanno prestazioni migliori della versione attuale in tutte le dimensioni dei messaggi. In particolare si può notare che la versione buffer ha risultati migliori di quella streaming per tutti i messaggi fino a 500 Kilobyte, mentre per messaggi più grandi la versione streaming sfrutterà il fatto di essere indipendente dalla dimensione del messaggio processato, come discusso nel Capitolo 3.2.1, ed otterrà per questo motivo prestazioni migliori.

6.4 Analisi comparativa con prodotti terzi

Di seguito verranno confrontati i risultati dei benchmark descritti nel paragrafo precedente, confrontati con i risultati ottenuti da alcune altre architetture che hanno partecipato all'ultima edizione dell'ESB Performance Test (round 6). I risultati del benchmark sono disponibili pubblicamente, all'indirizzo [32]. Dal

Validazione XSD	Stream	Buffer	DOM
500 B	2,302	3,193	2,181
1 KB	2,287	3,152	2,136
10 KB	1,708	2,400	1,153
100 KB	408	496	141
500 KB	90	96	0
10 MB	3,5	3,3	0
100 MB	0,3	0,2	0



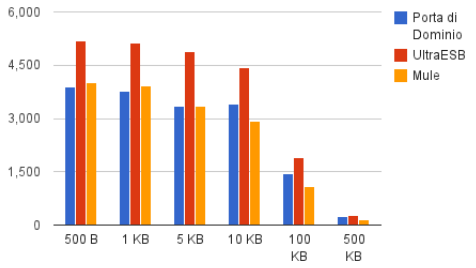
(A) Messaggi piccoli

(B) Messaggi grandi

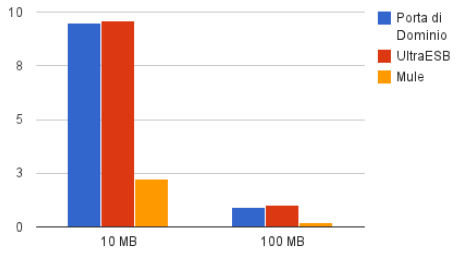
FIGURA 6.2: Validazione XSD nei risultati comparativi tra vecchia e nuova architettura

momento che i test sugli altri ESB sono stati nuovamente eseguiti, per conferma, e sono stati eseguiti inoltre i test sui messaggi più grandi di quelli usati per il round 6, si è deciso di confrontarsi solo con due ESB tra i partecipanti al round 6. Gli ESB scelti per il confronto sono UltraESB-Enhanced e Mule, che avevano riportato i risultati più alti nel round 6.

Direct Proxy	PdD	UltraESB	Mule
500 B	3,888	5,192	3,997
1 KB	3,770	5,135	3,926
5 KB	3,352	4,886	3,336
10 KB	3,409	4,442	2,928
100 KB	1,438	1,894	1,075
500 KB	236	257	161
10 MB	10	10	2
100 MB	1	1	0



(A) Messaggi piccoli



(B) Messaggi grandi

FIGURA 6.3: Risultati comparativi scenario Direct Proxy

6.4.1 Risultati Direct Proxy

Di seguito verranno presentati i risultati ottenuti nello scenario Direct Proxy dai tre ESB paragonati. Questi, oltre che nella gestione del messaggio SOAP utilizzata, differiscono in vari altri aspetti che esulano dagli scopi di questa Tesi, e queste differenze avranno un impatto anche sui benchmark relativi. Ciò risulta evidente dall'analisi dei risultati ottenuti dagli ESB sullo scenario Direct Proxy, sintetizzati in Figura 6.3. Questo test, come visto in precedenza,

misura le prestazioni di un'architettura nel caso in cui essa debba semplicemente inoltrare un messaggio ricevuto senza eseguirvi alcuna elaborazione, e quindi in maniera totalmente indipendente dalla gestione del messaggio SOAP.

Dal momento che tutti gli altri test si basano sull'eseguire operazioni aggiuntive a quella eseguita nel test del Direct Proxy, si è deciso di usare il valore ottenuto in questo test per normalizzare i valori ottenuti negli altri test dai vari ESB, in modo da evidenziare la differenza di prestazioni solo sulla gestione del messaggio SOAP. I risultati degli altri test saranno mostrati per questo in percentuale rispetto al valore di questo test.

6.4.2 Risultati CBR Proxy

Di seguito verranno presentati i risultati ottenuti dai tre ESB negli scenari di tipo CBR PRoxy. Uno di questi scenari, il CBR basato su header di trasporto, non è stato mostrato in quanto per la Porta di Dominio questo scenario equivale alla versione Direct Proxy. Più precisamente, non esiste per la Porta di Dominio una versione Direct Proxy in quanto essa decide il valore del nodo successivo da invocare a seconda dell'URL di invocazione. I due test eseguiti in questo ambito sono quindi quelli che misurano le prestazioni di un ESB che esegua CBR basato su informazioni contenute in un caso nell'header e nell'altro caso nel body del messaggio SOAP.

Come già visto nei test di confronto con la vecchia architettura, considereremo in questo scenario due versioni della Porta di Dominio. La prima lavora in modalità streaming, e utilizza SXC come motore interno, e la seconda lavora in modalità buffer, e utilizza VTD come motore interno.

6.4.2.1 Risultati CBR basato su informazioni contenute nel SOAP Header

Di seguito sono presentati i risultati ottenuti dai quattro ESB nello scenario *CBRSOAPHeaderProxy*, mostrati in percentuale rispetto al risultato ottenuto

CBR SOAP Header	Streaming PdD	Buffer PdD	UltraESB	Mule
500 B	73.02%	89.23%	99.34%	40.10%
1 KB	72.77%	93.27%	99.62%	39.40%
5 KB	74.07%	92.25%	92.86%	41.50%
10 KB	66.10%	85.11%	87.77%	42.41%
100 KB	69.05%	86.78%	81.00%	43.85%
500 KB	87.62%	80.52%	61.07%	26.29%
10 MB	96.53%	46.94%	67.09%	62.77%
100 MB	88.97%	43.89%	55.09%	0.00%

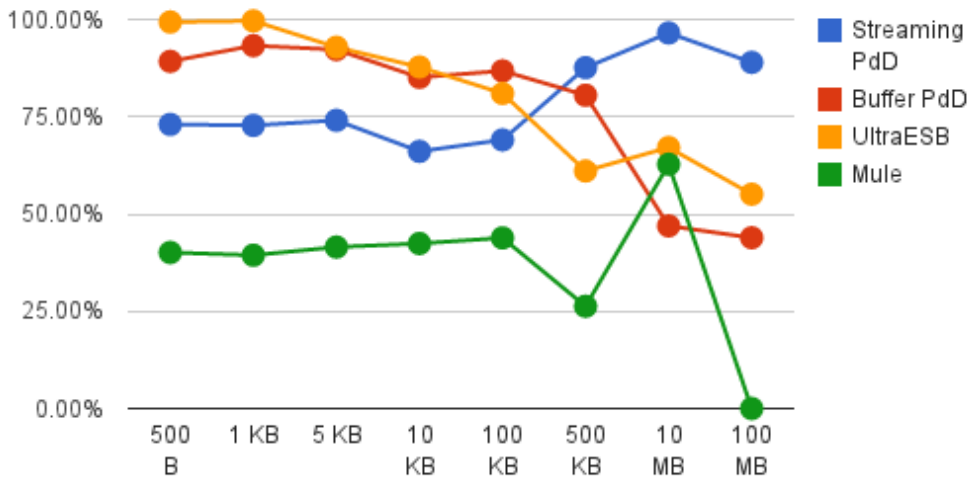


FIGURA 6.4: Risultati comparativi scenario CBRSOAPHeaderProxy

dallo stesso ESB nel test del Direct Proxy e aggregati per dimensione del messaggio.

Come si può vedere dalla Figura 6.4, esiste una versione della Porta di Dominio con prestazioni migliori di tutte le altre architetture per tutte le dimensioni del messaggio, ad eccezione dei messaggi da 500 Byte e da un Kilobyte. Per queste dimensioni il degrado delle prestazioni è per la versione streaming quasi del 30% e per quella buffer del 10-12%. Anche Mule ha prestazioni molto basse (40% rispetto alla versione Direct Proxy), e manterrà questo trend per tutti i messaggi considerati. UltraESB invece gestisce messaggi di questa dimensione con tempi sostanzialmente identici a quelli con cui gestisce lo scenario Direct Proxy. Già con messaggi da 5 Kilobyte però si può notare che la

versione buffer della PdD ha prestazioni paragonabili a quelle di UltraESB, così come succede nei messaggi da 10 Kilobyte. Come già visto nel confronto tra l'attuale architettura della PdD e la nuova, per messaggi di dimensione ridotta la versione buffer è la soluzione che ottiene le prestazioni migliori per la Porta di Dominio, con la versione streaming distanziata di venti punti percentuali.

Analizzando il risultato ottenuto nei test con messaggi da 100 Kilobyte si vede che per la prima volta le due versioni hanno prestazioni paragonabili, anche se la versione buffer ha risultati leggermente migliori, ma soprattutto si può notare che per la prima volta entrambe hanno prestazioni migliori di quelle di UltraESB. Per tutti i messaggi di dimensioni maggiori di 100 Kilobyte vediamo che la versione buffer vedrà le sue prestazioni degradarsi progressivamente, mentre la versione streaming avrà le prestazioni migliori in assoluto, avendo degradi minimi rispetto alla versione Direct Proxy.

6.4.2.2 Risultati CBR basato su informazioni contenute nel SOAP Body

Di seguito sono presentati i risultati ottenuti dai quattro ESB nello scenario *CBRProxy*, mostrati in percentuale rispetto al risultato ottenuto dallo stesso ESB nel test del Direct Proxy e aggregati per dimensione del messaggio.

Come si può vedere dalla Figura 6.5, i risultati dei test di questo scenario rispecchiano sostanzialmente quelli dello scenario precedente, con alcune piccole differenze che vale la pena evidenziare. Innanzitutto in questo scenario la versione buffer della Porta di Dominio è competitiva con UltraESB già per messaggi di 500 Byte, e risulta avere il parser più competitivo fino ai messaggi da 100 Kilobyte. Da questo caso in poi, come nello scenario precedente, la versione streaming ottiene le prestazioni più alte. Inoltre in questo scenario l'architettura Mule risulta essere molto più competitiva rispetto allo scenario precedente, soprattutto nei messaggi di dimensione limitata.

CBR	Streaming PdD	Buffer PdD	UltraESB	Mule
500 B	71.71%	94.68%	93.16%	85.91%
1 KB	74.49%	90.69%	92.94%	86.11%
5 KB	74.57%	90.66%	87.80%	80.25%
10 KB	64.71%	86.10%	83.09%	76.65%
100 KB	67.94%	86.16%	82.05%	70.19%
500 KB	88.29%	85.73%	78.22%	66.90%
10 MB	92.32%	45.24%	67.09%	13.02%
100 MB	86.34%	42.48%	55.09%	0.00%

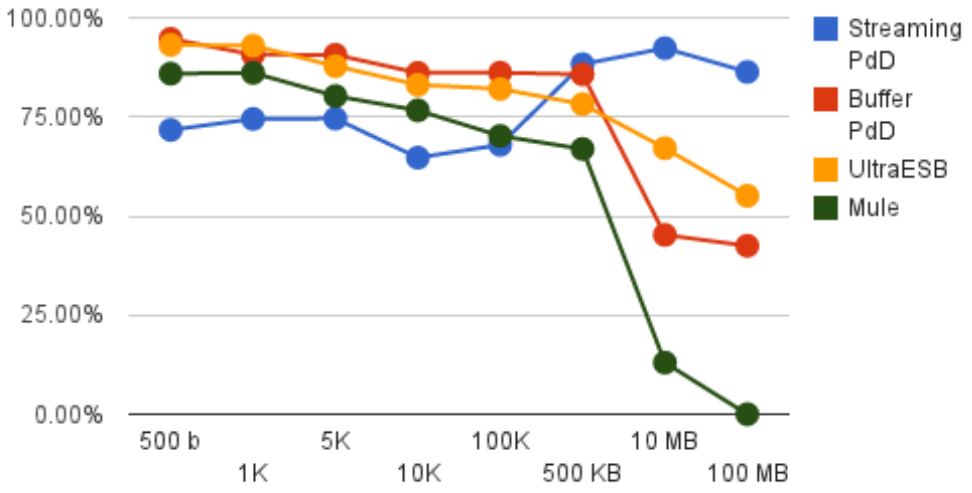


FIGURA 6.5: Risultati comparativi scenario CBRProxy

6.4.3 Risultati CBR con architettura flessibile

Nei precedenti paragrafi sono stati mostrati i risultati ottenuti dalle due versioni della Porta di Dominio comparati con le altre architetture nei vari scenari di Content-Based Routing. Dal momento che l'architettura proposta si basa sulla gestione flessibile del messaggio SOAP, e come discusso in 5.2.1, si può avvalere di un componente che di volta in volta sceglie di utilizzare la tecnologia più adatta alla gestione dello specifico messaggio trattato, possiamo considerare, nel caso ottimo, come risultato della PdD il risultato più alto tra le due versioni. Il grafico corrispondente, per il CBRProxy, è mostrato in Figura 6.6.

CBR	PdD Flessibile	UltraESB	Mule
500 B	94.68%	93.16%	85.91%
1 KB	90.69%	92.94%	86.11%
5 KB	90.66%	87.80%	80.25%
10 KB	86.10%	83.09%	76.65%
100 KB	86.16%	82.05%	70.19%
500 KB	88.29%	78.22%	66.90%
10 MB	92.32%	67.09%	13.02%
100 MB	86.34%	55.09%	0.00%

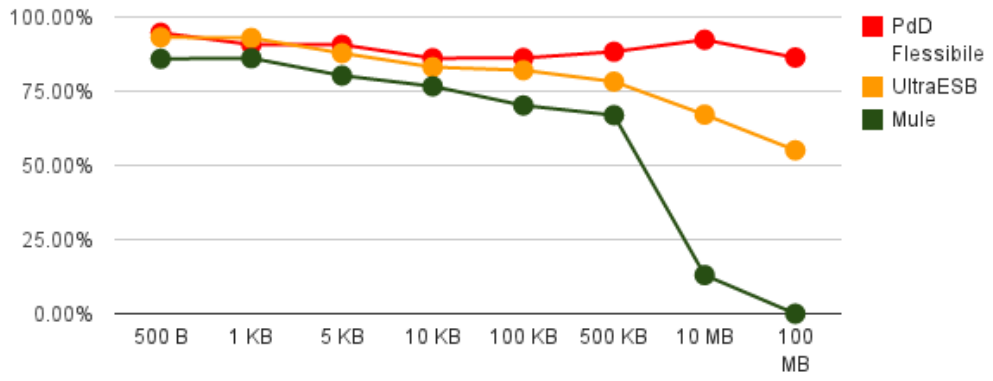


FIGURA 6.6: Risultati comparativi scenario CBRProxy con la gestione flessibile del messaggio SOAP

Conclusioni

L'attività svolta in questo lavoro di Tesi ha riguardato l'analisi di tecniche efficienti per la gestione dei messaggi XML, in particolare nell'ambito dei protocolli SOAP per la gestione di Web Services, con l'esplicita finalità di ottimizzare il comportamento della Porta di Dominio (PdD) OpenSPCoop.

Data la centralità dell'operazione di parsing nel trattamento dei messaggi XML, il lavoro si è quindi inizialmente concentrato sullo studio delle tecnologie di parsing allo stato dell'arte. Dalla successiva analisi dei casi d'uso più comuni nell'ambito della PdD, è emerso che la scelta di una specifica tecnologia per la gestione del messaggio può risultare appropriata in alcuni casi d'uso ma del tutto inefficace per altri.

Siamo giunti, in sostanza, alla conclusione che fosse necessario intervenire sull'architettura della PdD per modificare la gestione del messaggio dall'attuale modello che utilizza staticamente un unico motore di parsing, ad un modello flessibile nel quale ciascun messaggio potesse essere gestito con la strategia di parsing più adatta.

È stata quindi progettata una nuova architettura per la PdD, in grado di adottare diverse stategie di parsing in funzione sia della tipologia di messaggio da gestire, sia delle funzionalità da implementare sul messaggio (estrazione, validazione, ecc.). Tali stategie sono state concretamente implementate in un insieme di *plugin* esterni alla PdD e che possono evolvere nel tempo, a mano a mano che si rendono disponibili nuove soluzioni di parsing che ottimizzano specifici casi d'uso.

Infine la nuova versione della PdD è stata validata utilizzando dei benchmark standard, prima confrontandola con l'architettura attuale, e successivamente con i più diffusi analoghi prodotti di mercato. I risultati ottenuti hanno evidenziato i benefici dell'uso di un'architettura flessibile:

- il confronto con la vecchia architettura ha mostrato un netto miglioramento delle prestazioni su tutti i casi d'uso analizzati;
- il confronto con i principali prodotti di mercato ha mostrato risultati paragonabili con i prodotti più efficienti nella maggior parte dei casi, e risultati migliori in alcuni casi di messaggi di grandi dimensioni, in cui per il trattamento sulla PdD sono state adottate tecniche di parsing in streaming.

Bisogna infine sottolineare come, nella versione di PdD realizzata in questa tesi, la scelta dei plugin da utilizzare per il parsing del messaggio viene definita in fase di configurazione di un nuovo servizio. Questa soluzione, per quanto molto pratica in ambienti reali in cui tutti i messaggi XML indirizzati ad uno stesso servizio hanno caratteristiche simili, potrebbe essere ulteriormente migliorata introducendo nella PdD la capacità di individuare autonomamente le modalità di parsing da adottare. Tale attività, fuori dagli obiettivi iniziali della Tesi, resta quindi come possibile lavoro futuro.

Bibliografia

[1] David A. Chappell: *Enterprise Service Bus*, chapter 11.1. Giugno 2004

[2] Apache AxiOM. `http://ws.apache.org/commons/axiom`

[3] Eran Chinthaka: *Introducing AxiOM: The Axis Object Model* `https://today.java.net/pub/a/today/2005/05/10/axiom.html`, Ottobre 2005

[4] Virtual Token Descriptor. `http://vtd-xml.sourceforge.net`, 2013

[5] Formato VTD+XML. `http://vtd-xml.sourceforge.net/persistence.html`, 2013

[6] Progetto OpenSPCoop. `http://www.openspcoop.org/`

[7] Andrea Corradini e Tito Flagella: *OpenSPCoop: un Progetto Open Source per la Cooperazione Applicativa nella Pubblica Amministrazione* In *Congresso Nazionale AICA*, `http://www.openspcoop.org/openspcoop_v3/doc/papers/OpenSPCoopAICA2007.pdf`, 2007

[8] Asankha C. Perera e Ruwan Linton: *Framework: Objectives and History.* `http://esbperformance.org/display/comparison/Framework+-+Objectives+and+History`, 2012

[9] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, e David Orchard: *Web Services Architecture.* `http://www.w3.org/TR/ws-arch`, Febbraio 2004

[10] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, e Yves Lafon: *SOAP Version*

1.2 Part 1: Messaging Framework (Second Edition). http://www.w3.org/TR/soap12-part1, Aprile 2007

[11] SOAP Messages with Attachments. http://www.w3.org/TR/SOAP-attachments

[12] MIME Multipart/Related Content-Type. http://www.ietf.org/rfc/rfc2387.txt

[13] Agenzia per l'Italia Digitale. http://www.digitpa.gov.it/

[14] Ruggero Barsacchi: *Progettazione di un Framework Open Source per la Cooperazione Applicativa nella Pubblica Amministrazione.* Tesi di Laurea, Università degli studi di Pisa, http://www.openspcoop.org/openspcoop_v3/doc/tesi/TesiRuggeroBarsacchi.pdf, 2005

[15] Andrea Poli *OpenSPCoop: un'implementazione della Specifica di Cooperazione Applicativa per la Pubblica Amministrazione Italiana.* Tesi di Laurea, Università degli studi di Pisa, http://www.openspcoop.org/openspcoop_v3/doc/tesi/TesiAndreaPoli.pdf, 2006

[16] Lorenzo Nardi *Un'architettura innovativa per la Cooperazione Applicativa nel progetto OpenSPCoop.* Tesi di Laurea, Università degli studi di Pisa, http://www.openspcoop.org/openspcoop_v3/doc/tesi/TesiLorenzoNardi.pdf, 2007

[17] Aldo Lezza *La Gestione degli Accordi di Cooperazione nel progetto OpenSPCoop.* Tesi di Laurea, Università degli studi di Pisa, http://www.openspcoop.org/openspcoop_v3//doc/tesi/TesiAldoLezza.pdf, 2008

[18] Andrea Corradini, Tito Flagella, e Andrea Poli: *Aspetti di Interoperabilità della Specifica SPCoop nell'implementazione OpenSPCoop 1.0.* In *PAAL: Pubblica Amministrazione Aperta e Libera*, http://www.openspcoop.org/openspcoop_v3/doc/papers/openspcoopPAAL2007.pdf, 2007

[19] Jimmy Zhang: *A Step in the Right Direction: VTD-XML Improves XML Processing*. http://www.devx.com/xml/Article/30484, Febbraio 2006

[20] Sun Microsystems Inc.: *Why StAX?*. http://docs.oracle.com/cd/E17802_01/webservices/webservices/docs/1.6/tutorial/doc/SJSXP2.html, 2005

[21] XPath grammar. http://www.w3.org/2002/11/xquery-xpath-applets/xpath-bnf.html

[22] Organization for the Advancement of Structured Information Standards (OASIS). https://www.oasis-open.org/

[23] Anthony Nadalin, Chris Kaler, Phillip Hallam-Baker, e Ronald Monzillo: *Web Services Security: SOAP Message Security 1.1*. http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf, Febbraio 2006

[24] Michael H. Kay: *Saxon: the XPath API*. http://saxon.sourceforge.net/saxon7.5/api-guide.html, Aprile 2003

[25] Simple XML Compiler. http://sxc.codehaus.org/Home

[26] SOAP with Attachments API for Java. http://docs.oracle.com/cd/E17802_01/webservices/webservices/docs/2.0/saaj/index.html

[27] Apache Axis. http://ws.apache.org/axis, 2006

[28] Apache Axis2. http://ws.apache.org/axis2, 2012

[29] Eran Chinthaka: *Fast and Lighweight Object Model for XML, Part 1 and 2*. http://wso2.org/library/291,http://wso2.org/library/351, 2006

[30] Dennis Sosnosky: *Digging into Axis2: AxiOM*. http://www-128.ibm.com/developerworks/java/library/ws-java2/index.html, Novembre 2006

[31] Jeff Hanson: *Implementing WS-Security with Java and WSS4J.* `http://www.devx.com/Java/Article/28816`, Agosto 2005

[32] Asankha C. Perera e Ruwan Linton: *ESB Performance Testing: Round 6.* `http://esbperformance.org/display/comparison/ESB+Performance+Testing+-+Round+6`, 2012

[33] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`