UNIVERSITÀ DI PISA
Facoltà di Scienze Matematiche, Fisiche e Naturali.
Corso di Laurea Specialistica in Informatica.

Tesi di Laurea Specialistica

Un'architettura innovativa per la Cooperazione Applicativa nel progetto OpenSPCoop

Candidato: Lorenzo Nardi

Relatori:
Prof. Andrea Corradini
Prof. Tito Flagella

Controrelatore:
Prof. Vincenzo Gervasi

Anno accademico 2006-2007

Ai miei genitori,

Grazie.

Introduzione

Avviato nel 2004 con un'approfondita fase di analisi e progettazione svolta tramite due tesi di Laurea Specialistica dell'Università di Pisa [60,61], il progetto OpenSPCoop, implementazione open source delle specifiche per la Cooperazione Applicativa della Pubblica Amministrazione rilasciate dal *Centro Nazionale per l'Informatica nella Pubblica Amministrazione* (CNIPA), è giunto alla versione 1.0 del software, che rappresenta un'implementazione completa della specifica, sia per quanto attiene ai componenti periferici (Porte di Dominio), che ai componenti centrali (Registro dei Servizi e Gestore Eventi). La soluzione architetturale proposta da OpenSPCoop per la componente di integrazione della Porta di Dominio, come vedremo, è innovativa e risulta qualitativamente migliore di altre architetture proposte in campo Enterprise Service Bus (ESB).

L'utilizzo di OpenSPCoop ha mostrato, in alcune realtà applicative, di non raggiungere un livello di prestazioni soddisfacente. La causa di questo problema è da ricercare in scelte architetturali mirate a soddisfare altri aspetti rilevanti, come la qualità del servizio (QoS), a discapito delle prestazioni complessive del sistema.

Obiettivo di questa tesi è quello di studiare il problema, prendendo in considerazione sia la possibilità di integrare strumenti emergenti in ambito ESB, sia di evolvere l'architettura corrente al fine di gestire tutte le realtà applicative in modo efficiente, rispettando le specifiche SPCoop. Per giungere a questo obiettivo è stata necessaria una lunga e approfondita analisi della letteratura in ambito Web Service, confrontando l'implementazione di SPCoop con le architetture al presente stato dell'arte, analizzando gli strumenti sviluppati dalla comunità open source e verificandone la bontà sul campo, affidandoci ad un approccio pratico oltre che teorico.

Oltre a proporre una modifica all'architettura di OpenSPCoop volta a migliorare le performance, obiettivo della tesi è anche di aggiornare l'implementazione di SPCoop per conformarsi ai nuovi standard in ambito Web Service. Gli standard in questione hanno vissuto negli ultimi anni un'evoluzione tanto radicale quanto rapida ed OpenSPCoop, in previsione di un aggiornamento delle specifiche SPCoop, deve supportare le nuove specifiche. Una spinta decisiva in questo senso è stata l'interruzione

dello sviluppo di Axis, il Web Service Framework scelto per l'implementazione di OpenSPCoop, a favore del suo successore Axis2. La scelta di quale Web Service Framework andrà a sostituire Axis in OpenSPCoop, sarà ponderata in base a come l'architettura di questo componente si integra con quella esistente, quali standard verranno supportati ed il grado di maturità delle loro implementazioni.

Nel **primo capitolo** viene presentato brevemente il paradigma di Cooperazione Applicativa per la pubblica amministrazione, la sua implementazione open source *OpenSPCoop* e come questo di inserisce nel contesto dei Web Service.

Nel **secondo capitolo** viene esposta una panoramica sulla tecnologia Web Service, le architetture proposte e gli standard sui quali si basa, mostrando il campo di applicazione di ognuno. Per quanto riguarda le nuove specifiche, viene mostrata la loro evoluzione, dalla progettazione di OpenSPCoop ad oggi, enfatizzando le differenze con i predecessori e la difficoltà di conformare il progetto ad esse.

Il **terzo capitolo** è un'analisi sulle tecnologie proposte a soluzione delle problematiche in ambito SOA ed ESB. Dopo aver introdotto l'architettura corrente del progetto OpenSPCoop, valuteremo gli ultimi prodotti open source che la comunità sta sviluppando, la loro maturità e la loro aderenza agli standard analizzati nel secondo capitolo.

Nel **quarto capitolo** viene effettuata un'analisi critica degli strumenti introdotti nel terzo capitolo. Una volta approfondita l'architettura corrente di OpenSPCoop ed enfatizzate le scelte architetturali alla base delle inefficienze prestazionali, sono valutati i Web Service Framework candidati a sostituire Axis, ed alcuni prodotti in ambito ESB, valutandone un utilizzo nell'implementazione di SPCoop.

Nel **quinto capitolo** è presentata una nuova architettura per il progetto di OpenSPCoop. Partendo dall'analisi compiuta nel quarto capitolo, viene mostrato come integrare nell'architettura corrente una gestione alternativa del processo di mediazione, con i dettagli dell'architettura che ne permette l'implementazione nel rispetto delle specifiche SPCoop.

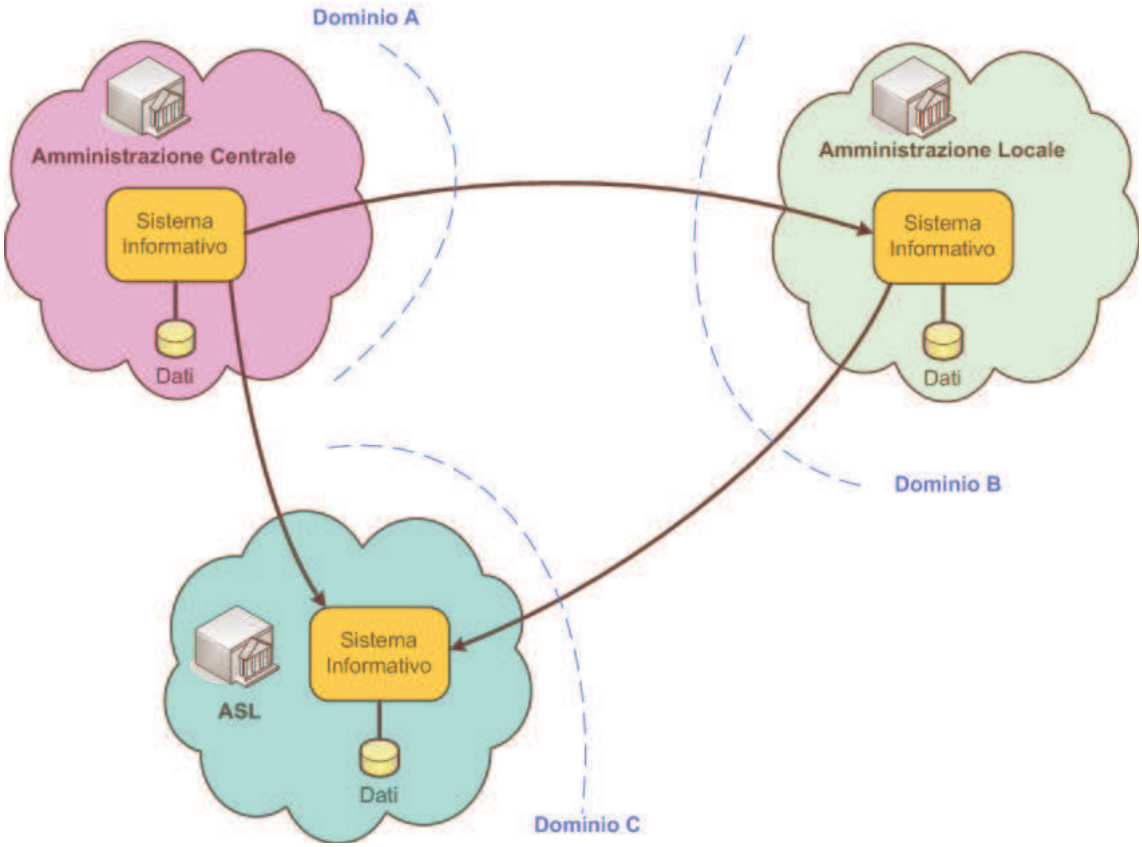
Nel **sesto capitolo** è presentato un prototipo di implementazione dell'architettura proposta, mostrando più in dettaglio le funzionalità dei componenti che lo strutturano. A seguire verranno inserite le parti di codice più rilevanti estratte dai sorgenti..

1. La specifica SPCoop per la Cooperazione Applcativa

Questo capitolo descrive il contesto generale in cui si inserisce il lavoro svolto in questa tesi. Cominceremo con il presentare il paradigma della Cooperazione Applicativa (SPCoop) introdotto dal CNIPA, e le ragioni che hanno motivato la creazione di questa nuova specifica. Proseguiremo introducendo il progetto specifico che questa tesi intende analizzare e migliorare, OpenSPCoop, implementazione open source della specifica SPCoop. Verrà mostrato come questo progetto si inserisce nel panorama dei Web Service, introducendo infine le ultime tecnologie in ambito Web Service, i Web Service Mediator.

1.1 Cooperazione Applicativa

Prima che si cominciasse ad affermare l'idea della standardizzazione di un paradigma di cooperazione applicativa, le comunicazioni nella Pubblica Amministrazione avvenivano tramite collegamenti punto-punto tra i server applicativi interessati. Poiché tipicamente questi server erano dislocati su reti private, raggiungibili quindi esclusivamente da altri server dislocati sulla loro stessa rete privata, era necessario realizzare reti virtuali private (VPN) tra le amministrazioni interessate a cooperare.



Questa soluzione si è rapidamente dimostrata inadatta nella gestione del processo di eGovernment, che prevede che due qualunque enti debbano essere potenzialmente in grado di comunicare tra loro.

SPCoop [01] risolve questo problema imponendo un'infrastruttura standard di comunicazione tra le amministrazioni pubbliche. In tal modo, una volta che l'infrastruttura sia diventata completamente operativa, sarà sufficiente il collegamento di un'amministrazione all'infrastruttura SPCoop per abilitarla alla comunicazione con qualunque altra amministrazione italiana ed europea.

Alla specifica SPCoop si è giunti attraverso un certo numero di passaggi che qui ricordiamo:

2001. Nasce il Piano Nazionale di e-Government, emanato dal Ministero per l'innovazione e le Tecnologie con l'obiettivo di definire una rete di servizi usufruibili dai cittadini e dalle imprese attraverso un mezzo di trasmissione telematico.

2002. Il Centro Nazionale per l'Informatica nella Pubblica Amministrazione (CNIPA) istituisce un gruppo di lavoro, a cui prendono parte circa 120 esperti, in rappresentanza delle Amministrazioni centrali e locali, delle Associazioni dei

fornitori e del CNIPA con lo scopo di definire:

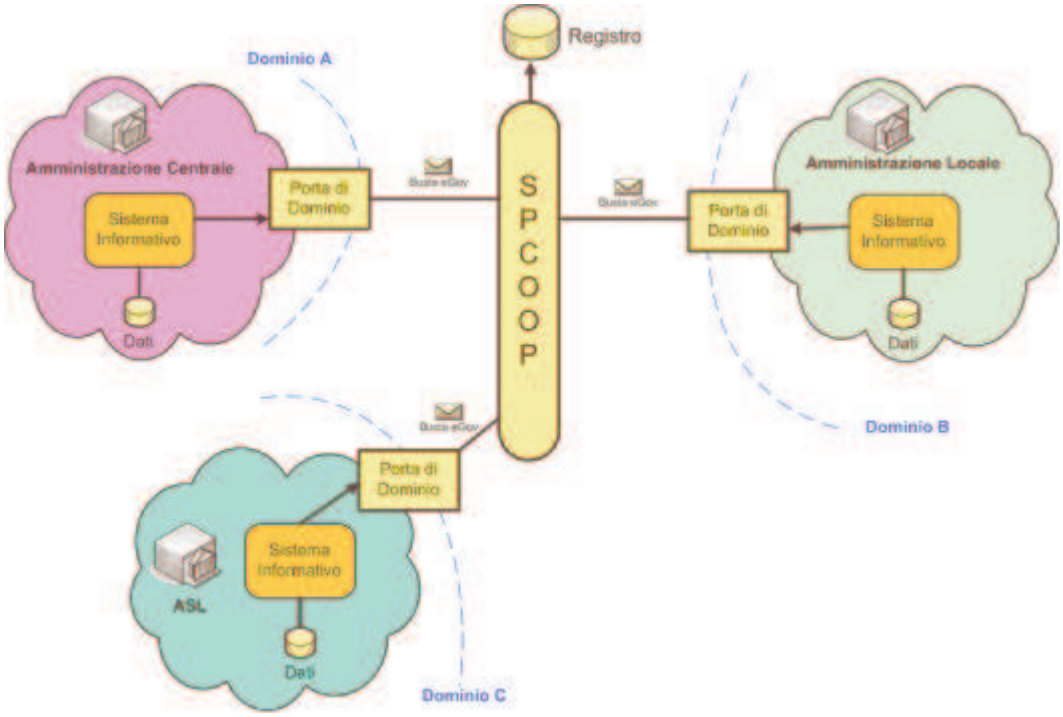
- Un'infrastruttura di comunicazione in grado di collegare tutte le amministrazioni, denominato Sistema Pubblico di Connettività (SPC);
- Un insieme di standard tecnologici e di servizi infrastrutturali che permettano alle amministrazioni di interoperare attraverso interfacce applicative standardizzate, denominato Sistema Pubblico di Cooperazione (SPCoop).

Aprile 2004. Il CNIPA rilascia la versione 1.0 della specifica della busta e-Gov, che definisce il formato standard in cui debbano avvenire le richieste di servizio e lo scambio dei dati tra l'erogatore e il fruitore di un servizio nella Pubblica Amministrazione.

Novembre 2004. Il CNIPA rilascia la versione 1.0 dell'Architettura SPCoop, che descrive i servizi infrastrutturali comuni e le modalità d'interazione tra i vari componenti del Sistema Pubblico di Cooperazione.

Ottobre 2005. Il CNIPA completa la stesura di un insieme di documenti che costituiscono il riferimento tecnico per lo sviluppo dei servizi infrastrutturali volti a delineare il quadro tecnico ed implementativo del Sistema Pubblico di Cooperazione.

La figura seguente mostra i principali componenti dell'infrastruttura SPCoop: la busta eGov [02], la Porta di Dominio [03], le porte delegate e applicative, e il Registro dei Servizi [04].



Uno dei componenti principali della specifica SPCoop è la Porta di Dominio (PdD), che delimita il confine di responsabilità di un ente o soggetto amministrativo e racchiude al suo interno tutte le applicazioni da esso gestite. Le comunicazioni da e verso un dominio devono quindi attraversare la sua Porta di Dominio. Le PdD si parlano tra loro scambiandosi richieste e risposte in un formato standard, denominato busta eGov, che è sostanzialmente una specializzazione di un messaggio SOAP, esteso con un apposito header per definire le caratteristiche del protocollo SPCoop.

La Porta Delegata e la Porta Applicativa costituiscono gli elementi della PdD che mediano gli accessi tra i sistemi interni agli enti e l'infrastruttura SPCoop. In particolare la Porta Delegata è utilizzata come proxy per l'accesso al servizio destinazione, mentre la Porta Applicativa deve essere in grado di gestire la consegna dei contenuti applicativi a un server interno al dominio destinazione.

La specifica SPCoop prevede poi la presenza di un registro dei soggetti e degli accordi di servizio tra essi stipulati. Si prevede l'esistenza di un registro di primo livello gestito dal CNIPA, che includa tutti i servizi ufficiali SPCoop a livello nazionale, e di registri di secondo livello che possano contenere un sottoinsieme di tali servizi. La specifica prevede infine la presenza di un Gestore Eventi per permettere lo scambio di buste eGov secondo l'architettura event-driven (EDA).

1.2 OpenSPCoop

OpenSPCoop [05] è un progetto finalizzato alla realizzazione di un insieme di componenti open source aderenti alle specifiche per la Cooperazione Applicativa nella Pubblica Amministrazione, recentemente rilasciate dal CNIPA e note con il nome di Servizio Pubblico di Cooperazione (SPCoop). Come per ogni nuova specifica non banale, anche per SPCoop è di fondamentale importanza l'esistenza di un'implementazione di riferimento open source, che permetta di sperimentare in maniera condivisa i vari concetti proposti nella specifica, evidenziando e proponendo possibili soluzioni per le potenziali ambiguità o debolezze della specifica stessa. Il progetto OpenSPCoop nasce sostanzialmente con questo obiettivo, enfatizzando i noti vantaggi dell'approccio open source per indirizzare i 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.

Preceduta da un'approfondita fase di analisi e di progettazione svolta presso il Dipartimento di Informatica dell'Università di Pisa, la prima release del software OpenSPCoop è stata rilasciata da Link.it [06] il 27 ottobre 2005. Questa prima release è stata poi seguita da frequenti nuovi rilasci, anche grazie al feedback e al contributo dei primi utenti del software, fino alla versione 1.0b3 del 20 Novembre 2007, terza beta della versione 1.0. Attorno al sito del progetto si è sviluppata sin dal primo momento una comunità di utenti e sviluppatori molto qualificati, provenienti da aziende italiane, pubbliche amministrazioni locali e centrali e centri di ricerca. Già agli albori di questo progetto si è concretizzata una piccola comunità attiva anche nello sviluppo dello stesso,

comunità che nel tempo si è ampliata e resa ancora più partecipe nell'evoluzione del progetto; questo ha dimostrato tra l'altro la bontà della scelta di una soluzione open source in un settore così critico come quello indirizzato dalla specifica SPCoop.

1.3 La Specifica SPCoop nel Contesto dell'Architettura Web Service

Dal punto di vista del contesto tecnologico, la specifica SPCoop è fortemente basata sugli standard dei Web Service. In particolare, essa richiede la compatibilità dei messaggi applicativi con lo standard SOAP 1.1 [13], in accordo alla specifica WS-I Basic Profile 1.1 [14]. La specifica SPCoop comprende due aspetti principali:

1. Il formato di un header di protocollo SOAP, comunemente riferito come busta eGov, che permette la gestione di numerosi aspetti del protocollo di comunicazione tra Enti Pubblici, quali l'indirizzamento, l'affidabilità della consegna e la gestione dei duplicati.
2. Un insieme di norme organizzative, che identificano alcuni componenti (principalmente le Porte di Dominio e i Registri di Servizi) e il loro ruolo di supporto alla cooperazione tra i servizi applicativi dei singoli domini. In particolare la Porta di Dominio ha un ruolo di mediazione tra i servizi applicativi interni al dominio e le Porte di Dominio degli altri Enti.

Poichè non è previsto che il formato della busta eGov sia parlato nativamente dalle applicazioni, la Porta di Dominio deve anche occuparsi di convertire le richieste applicative in formato proprietario nel formato di busta eGov. Facendo riferimento a questa problematica, i compiti della Porta di Dominio sono solitamente divisi in due diverse componenti: il componente di integrazione e quello di cooperazione. Mentre la specifica SPCoop copre completamente gli aspetti relativi al componente di cooperazione, per quanto attiene al componente di integrazione essa si limita a presentare un esempio di massima di una possibile realizzazione. Per questo motivo, i vari prodotti finora realizzati si differenziano significativamente proprio per quanto attiene al componente di integrazione, cioè a come costruire le buste a partire dai dati forniti dall'applicativo.

Il problema rientra nella problematica più generale dell'uso dei Web Service per l'Integrazione Applicativa in ambito Enterprise (EAI), problema di importanza centrale negli ambienti Enterprise, e per il quale si sono andate evolvendo soluzioni sempre più sofisticate.

Approccio "End to End"

Il problema è stato gestito per molto tempo, e in larga parte ancora adesso, direttamente dagli applicativi (mittente e richiedente), che si occupavano in prima persona di gestire sia le modalità di integrazione (imbustamento dei contenuti applicativi in SOAP) che quelle di cooperazione (gestione dei vari servizi a livello di header SOAP).

Approccio "Enterprise Service Bus"

In vista di un significativo incremento dei servizi realizzati tramite tecnologia Web Service all'interno delle organizzazioni, sono stati sviluppati ambienti che forniscono servizi infrastrutturali comuni per la gestione dell'affidabilità, il routing e la trasformazione dei servizi web. Tali soluzioni, note come Enterprise Service Bus (ESB), forniscono quindi servizi comuni sia di Integrazione che di Cooperazione. Tra gli ESB più noti ci sono Mule, ServiceMix, Celtix e Artix.

Gli ESB hanno costituito un significativo passo avanti per la gestione delle problematiche tipiche dell'EAI. Tuttavia, proprio in quanto indirizzati alla gestione di problematiche di integrazione "general purpose", si tratta di ambienti significativamente complessi, tanto da manifestare alcune controindicazioni per l'implementazione dell'ambiente SPCoop:

1. La configurazione di nuovi servizi è in generale piuttosto complessa e richiede solitamente la scrittura di codice ad-hoc per programmare le logiche di trasformazione. Questa complessità aumenta il rischio di errori nella configurazione dei nuovi servizi.
2. Le funzionalità di trasformazione di messaggio dei Service Bus non sono sufficienti a gestire il protocollo SPCoop, che prevede complesse logiche di indirizzamento e affidabilità. Quindi, anche usando un ESB rimane necessario implementare il servizio SPCoop come servizio di base, parte del core del Service Bus.
3. L'uso di una piattaforma "general purpose" di questo tipo rende difficile assicurare performance elevate in termini di numero di messaggi gestibili al

secondo. Questo problema è particolarmente serio per quelle istanze di Porte di Dominio che operano come smistatori e validatori di messaggi in transito, come ad esempio si può verificare per dei gateway interregionali. Queste porte hanno solo funzionalità di cooperazione e non di integrazione, e sono proprio quelle soggette al maggior carico di messaggi da smistare.

4. Gli ESB supportano sempre connettori SOAP su vari trasporti standard (come SMTP, HTTP e HTTPS); tuttavia le funzionalità a maggior valore aggiunto sono specifiche per la piattaforma sulla quale sono implementati (ad esempio disponibilità del trasporto JMS per gli ESB su piattaforma J2EE). Quindi la dipendenza dalle funzionalità a valore aggiunto degli ESB implica spesso anche la dipendenza da una specifica architettura software.

1.4 Web Service Mediation

Un nuovo approccio per la soluzione della stessa problematica affrontata dagli ESB è emerso di recente in alcuni progetti, tra i quali sta raggiungendo una notevole notorietà il progetto Apache Synapse [07]. In questo approccio si parte dalla constatazione che l'ampia diffusione dell'XML e del paradigma dei Web Service permette di considerare i sistemi interni al Dominio di Servizio come già capaci o facilmente adattabili al dialogo tramite Web Service. Se si assume questa premessa, la componente di integrazione del Service Bus non dovrà più supportare tecnologie diverse per ogni possibile sistema legacy da interfacciare (CORBA, RMI, JMS, .NET, etc.), ma potrà essere invece un generico container che funga da mediatore dei messaggi SOAP in arrivo dai clienti interni per i servizi esterni e viceversa.

Il mediatore potrà ancora fornire tutti i servizi a valore aggiunto dei Service Bus, introducendo o interpretando gli headers del messaggio SOAP in transito, ma rispetto agli ESB tradizionali esporrà solo un generico connettore per l'accettazione di buste SOAP, come componente di integrazione.

Dal punto di vista dell'architettura Web Service, il ruolo del Web Service Mediator è quello di un "nodo intermedio". L'architettura dei Web Service prevede infatti che un messaggio SOAP viaggi dal mittente fino al destinatario finale, passando eventualmente attraverso un certo numero di nodi intermedi. Ogni nodo intermedio quindi riceverà il

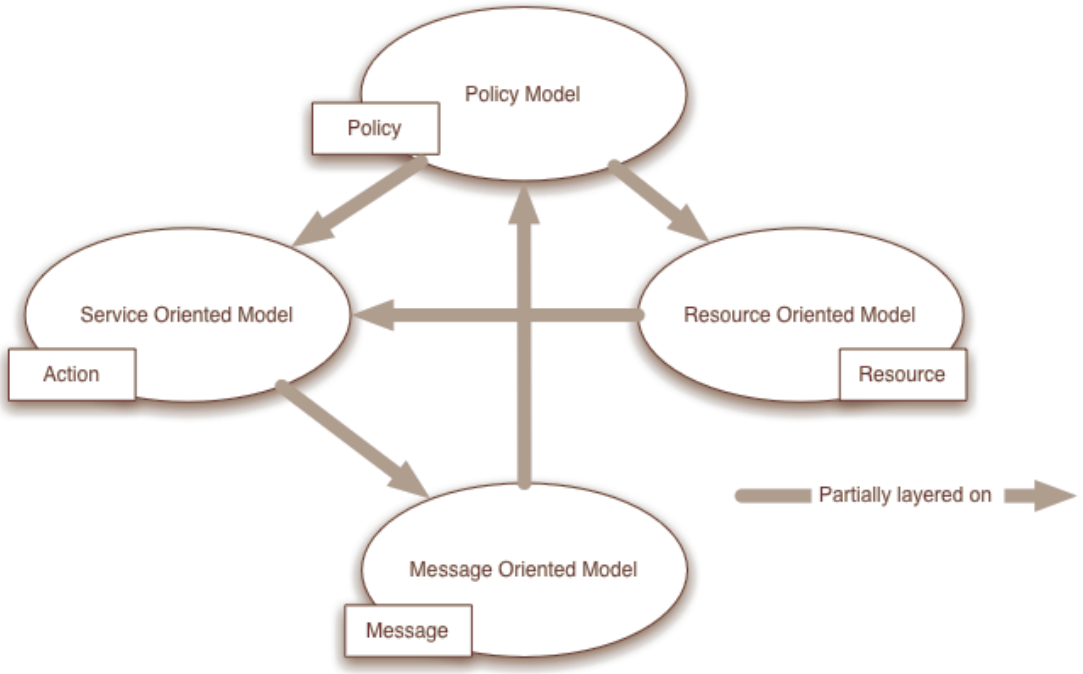
messaggio SOAP e lo ritrasmetterà verso la destinazione finale. Il messaggio passando da un nodo all'altro può essere manipolato secondo le politiche definite per ogni servizio. Di conseguenza quello del Web Service Mediator si pone anche come l'approccio più corretto da un punto di vista architetturale.

2. Tecnologie di riferimento

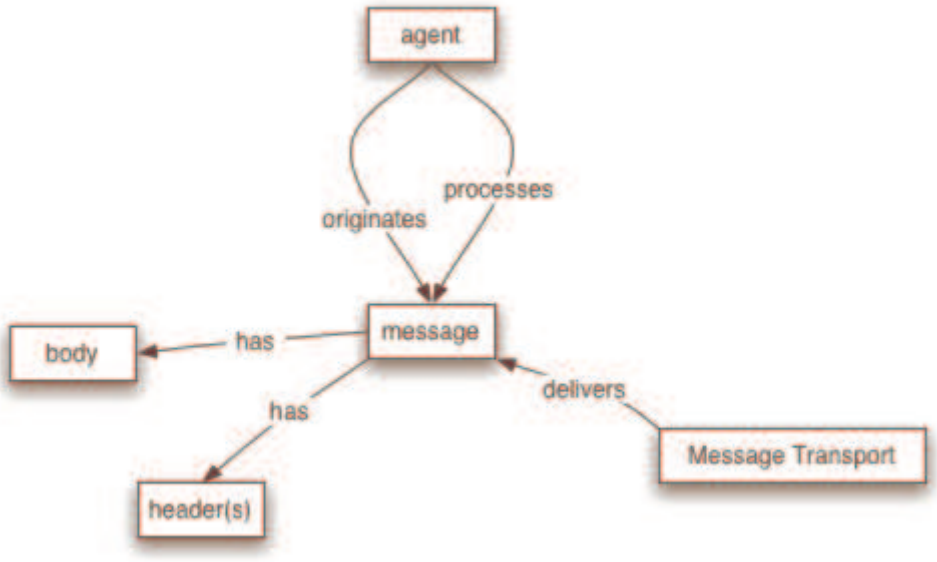
In questo capitolo si presentano le tecnologie utilizzate in ambito Web Service. Il primo paragrafo viene introdotta l'architettura dei Web Service, SOA ed ESB. A seguire vengono introdotte i protocolli e gli standard alla base dei Web Service. Infine una panoramica sulle implementazioni di quegli standard per il linguaggio Java e la loro evoluzione degli ultimi mesi.

2.1 Architettura dei Web Service

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. In alcuni articoli che ho consultato a riguardo ho trovato commenti piuttosto preoccupati su questa mancanza, ma anche commenti di chi preferisce proseguire senza specificare questo aspetto [51,52,53,54,55]. Quella che segue è una descrizione dell'architettura dei Web Service basata su una nota W3C dell'11 Febbraio 2004 chiamata WS-Arch [30]. In questo articolo, l'architettura Web Service è suddivisa in modelli, dove per modello s'intende un sottoinsieme dell'architettura che si occupa di gestire un aspetto specifico trattato dall'architettura generale. I vari modelli, anche se si concentrano su aspetti diversi, possono condividere alcuni concetti, magari visti da punti diversi.

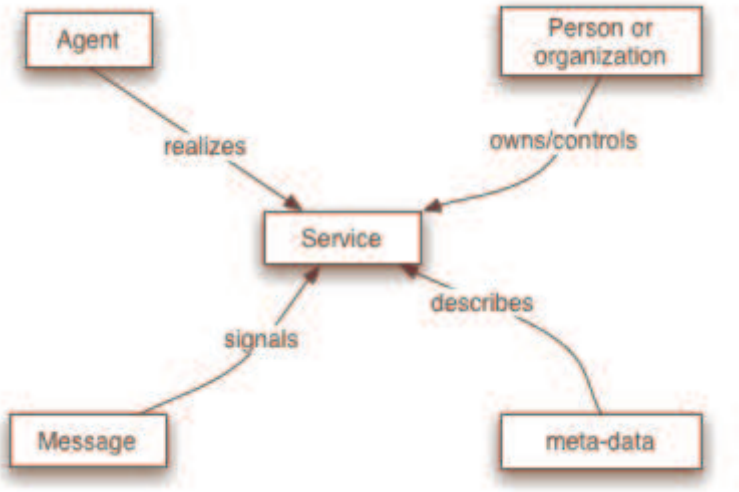


- Il *Message Oriented Model* si concentra sui messaggi, la loro struttura, il loro trasporto etc., senza particolari riferimenti alla ragione per cui i messaggi sono stati creati o al loro significato.



L'essenza del modello orientato ai messaggi ruota attorno ad alcuni concetti chiave illustrati nel grafo precedente: 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.

- Il Service Oriented Model si concentra sulla questione dei 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, il contrario non è vero: un messaggio può non avere nozioni sul servizio a cui è destinato.

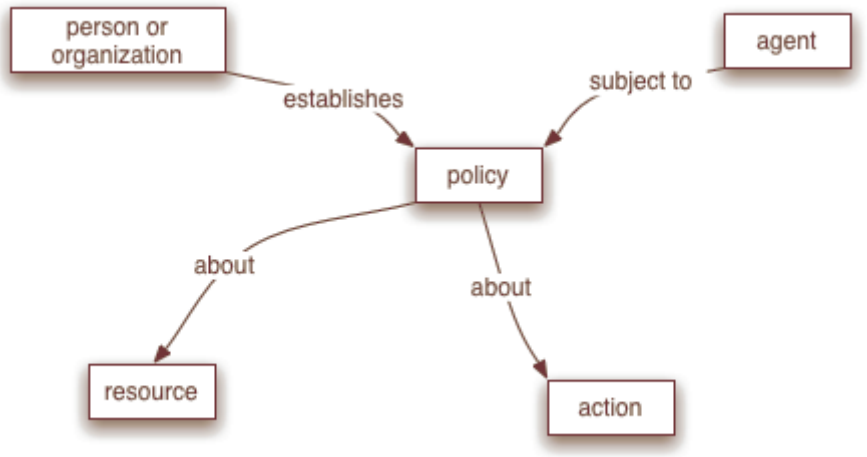


Il modello orientato ai servizi è il più complesso di tutti i modelli dell'architettura. Comunque, anche lui si basa su pochi concetti chiave. Un servizio è realizzato da un agente e utilizzato da un'altro agente. Inoltre il modello orientato ai servizi fa uso di metadati che sono una proprietà chiave delle Service Oriented Architecture (SOA), come vedremo nel prossimo paragrafo. Questi metadati sono usati per rappresentare vari aspetti del servizio: dai dettagli delle interface al binding del trasporto, dalla semantica del servizio a quali politiche restrittive sono applicate al servizio.

- 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 agli agenti che vogliono far uso di determinate risorse dalle persone che le gestiscono. Le Policies possono essere introdotte per gestire aspetti quali la sicurezza, qualità del servizio (QoS), management etc..

2.1.1 Service Oriented Architecture

Un sistema distribuito è formato da diversi agenti software discreti che devono cooperare insieme per svolgere determinate funzioni. Inoltre, gli agenti in un sistema distribuito non operano nello stesso ambiente, quindi devono comunicare affidandosi a protocolli software e hardware attraverso una rete. Questo implica che le comunicazioni in un sistema distribuito sono intrinsecamente meno veloci e disponibili di quelle

tramite invocazione diretta nel codice e usando memoria condivisa. Questo ha delle importanti implicazioni architetturali, poichè i sistemi distribuiti richiedono che gli sviluppatori prendano in considerazione l'imprevedibile latenza di una comunicazione remota e gestire le problematiche derivate dalla concorrenza e dai possibili fallimenti parziali.

I *distributed object systems* sono sistemi distribuiti nei quali la semantica dell'inizializzazione di un oggetto e dei suoi metodi sono esposti a sistemi remoti tramite meccanismi proprietari o standard per inoltrare la richiesta oltre i confini del sistema, effettuare marshall o unmarshall degli argomenti dei metodi, etc.

Una *Service Oriented Architecture* (SOA) [33,35] è una forma d'architettura di sistemi distribuiti tipicamente caratterizzata da queste proprietà:

- *Vista logica*: Il servizio è un'astrazione, vista logica, del programma, database, processo etc., definito in termini di cosa fa, tipicamente astraendo un'operazione.
- *Orientato ai messaggi*: Il servizio è formalmente definito in termini dei messaggi scambiati tra l'agente fornitore e l'agente fruitore e non sulle proprietà degli agenti stessi. La struttura interna degli agenti, compreso ad esempio il linguaggio di programmazione usato per implementarlo, 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. Un beneficio chiave di questa funzionalità interessa i cosiddetti sistemi legacy.
- *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.

2.1.2 Enterprise Service Bus

Un *Enterprise Service Bus* (ESB) è un'infrastruttura software che fornisce servizi di supporto ad architetture SOA complesse. Un ESB si basa su sistemi disparati, interconnessi con tecnologie eterogenee, e fornisce in maniera consistente servizi di orchestration, sicurezza, messaggistica, routing intelligente e trasformazioni, agendo come una dorsale attraverso la quale viaggiano servizi software e componenti applicativi. Un ESB si contraddistingue come soluzione migliorativa, rispetto ad altre più classiche di tipo SOA oriented, in quanto ad esso sono delegati i servizi comuni, denominati core service (Security, tracking, routing etc..), che andrebbero altrimenti realizzati.

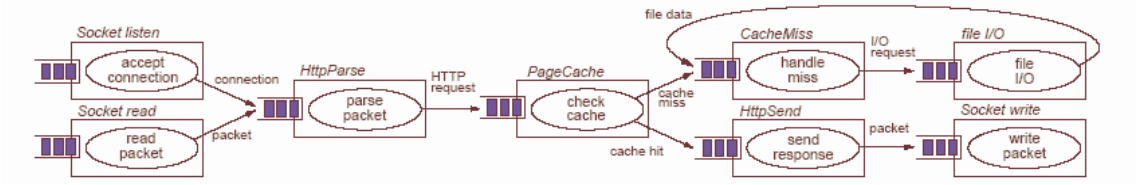
2.1.3 Event-Driven Architecture

Event-driven architecture (EDA) è un paradigma di architettura sofware per la produzione, gestione, utilizzo e reazione agli eventi.
Per evento s'intende "una significativa modifica allo stato". Ad esempio quando un acquirente compra una macchina, il suo stato cambia in "venduta". Il sistema di gestione del venditore può trattare questo cambio di stato come un evento da raccogliere, segnalare e far processare da varie applicazioni a corredo dell'architettura.
Questo paradigma architetturale può essere applicato nella specifica e implementazione di applicazioni e sistemi che trasmettono eventi attraverso componenti software fortemente disaccoppiati e servizi. Un sistema orientato agli eventi tipicamente è costituito da produttori e consumatori di eventi. I consumatori si sottoscrivono a degli intermediari detti event manager, mentre i produttori pubblicano gli eventi a questi managers (*Subscribe and Publish*, S&P). Quando un manager riceve un evento dal produttore, lo notifica al consumatore. Se questo è indisponibile, lo memorizza e tenta un ulteriore invio in seguito. Questo metodo di trasmissione degli eventi è riferito nei sistemi orientati ai messaggi come "*store and foward*"
Costruire applicazioni e sistemi basandosi su questo tipo di architettura permette di

ottenere una maggiore affidabilità, poiché i sistemi basati sugli eventi sono, per costruzione, più tolleranti ad ambienti imprevedibili ed asincroni.

2.1.4 Staged Event-Driven Architecture

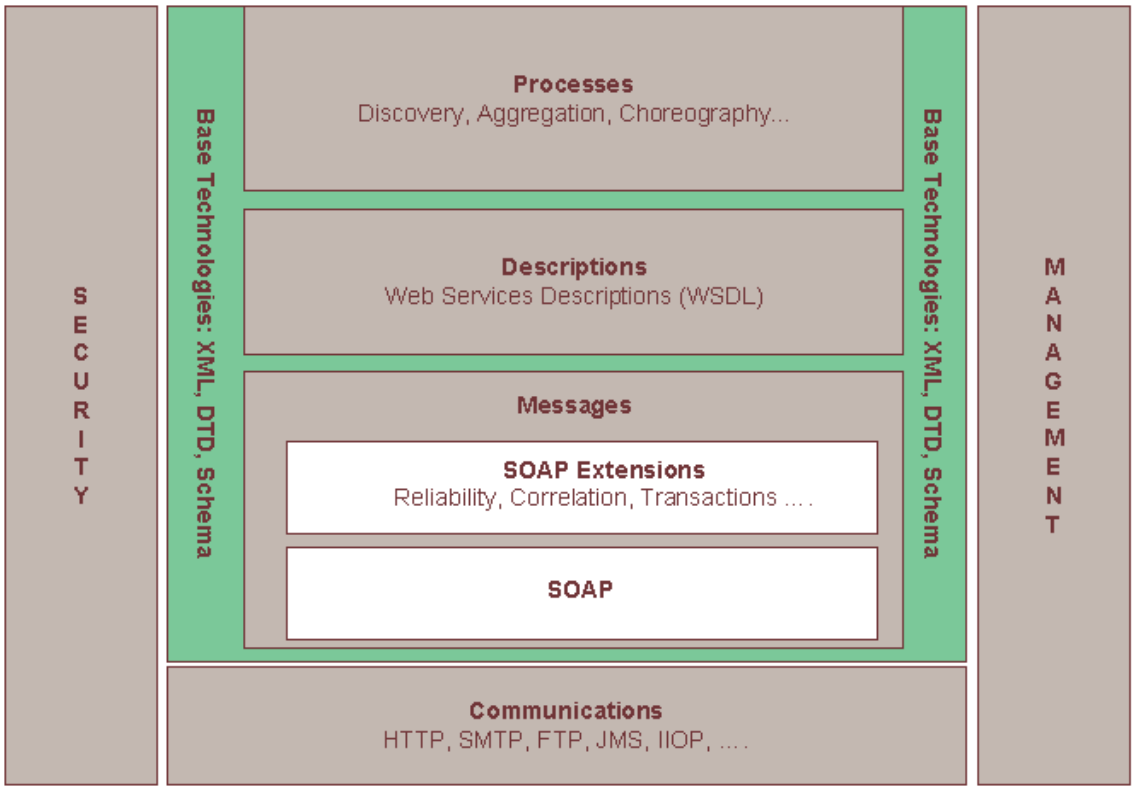
SEDA [31,32] è l'acronimo di *staged event-driven architecture*. Questa architettura è stata ideata da Matt Welsh dell'Università di Harvard e prevede di decomporre un'applicazione complessa, orientata agli eventi, in un set di fasi (*stages*) connessi da code. Questa architettura risolve il problema di overhead presente in modelli basati su thread, e disaccoppia eventi e thread dalla logica applicativa. Questo permette di prevenire che le risorse siano occupate da un carico di lavoro che eccede le loro capacità, evitando un approccio thread-based, dove ogni richiesta è associata ad un thread.



In figura viene mostrato come si può adattare a questa architettura un processo di gestione di una richiesta HTTP ad un Web Server.
Strutturando il processo in un set di fasi (*stages*) si introduce modularità, riuso del codice e permette lo sviluppo di strumenti di debug per applicazioni complesse basate sugli eventi. Ogni fase è separata dalle altre tramite code ed ad ognuna di esse è associata un pool di thread, non esposti all'applicazione. Questa architettura si presta a fornire funzionalità di bilanciamento del carico, controllando il flusso dei dati attraverso le code e agendo sui pool di thread associate ad esse. Abbiamo quindi un buon compromesso tra la programmabilità dei thread ed i benefici dell'orientamento agli eventi.

2.2 Protocolli e tecnologie dei Web Service

L'approccio ai Web Service è basato su un maturo set di standard largamente accettato e usato. Questa sua diffusione permette a client e service di poter comunicare attraverso un'ampia varietà di piattaforme scavalcando i confini dei linguaggi utilizzati.
Una breve panoramica sugli standard che tratteremo e sulla loro struttura gerarchica è mostrata nella seguente figura.



2.2.1 XML

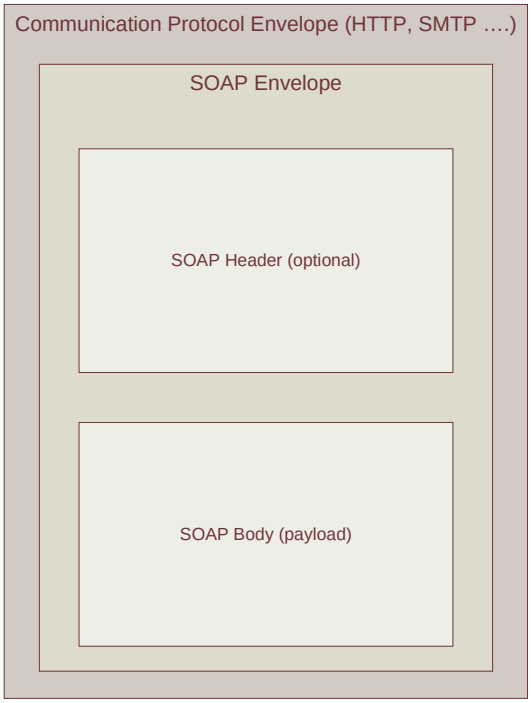
Lo XML [15], acronimo di *eXtensible Markup Language*, ovvero "Linguaggio di marcatura estensibile" è un metalinguaggio creato e gestito dal *World Wide Web Consortium* (W3C), semplificazione e adattamento dello *Standard Generalized Markup Language* (SGML), da cui è nato nel 1998. Esso è un meta-linguaggio di markup, cioè un linguaggio che permette di definire altri linguaggi di markup. A differenza di HTML, XML non ha tag predefiniti e non serve per definire pagine Web né per programmare. Esso serve esclusivamente per definire altri linguaggi. XML è dunque un meta-linguaggio per definire la struttura di documenti e dati. Il termine "documento" ricorre

spesso nella terminologia XML. Anche se esso può far pensare pagine a Web o altri prodotti dell'elaborazione di testo, è utilizzato nella sua accezione più ampia di contenitore di informazioni. L'XML è diventato lo standard de facto per descrivere i dati che devono essere scambiati attraverso la rete.

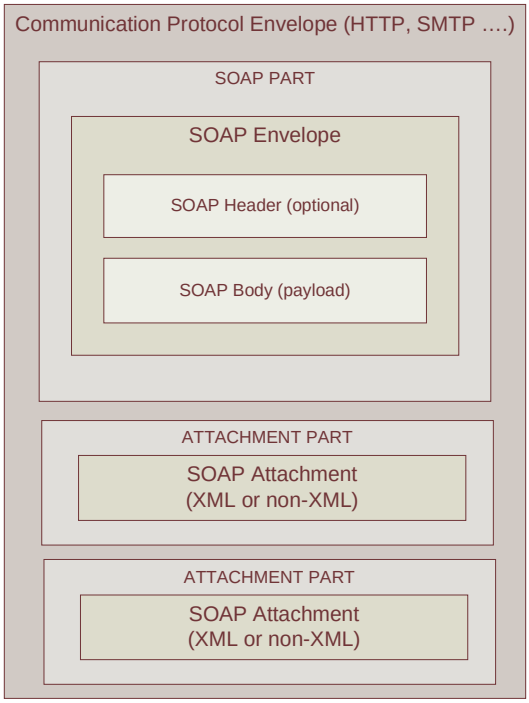
A corredo di XML ci sono una serie di strumenti ad esso correlati:

- *DTD* (acronimo di *Document Type Definition*): è un documento attraverso cui si specificano le caratteristiche strutturali di un documento XML attraverso una serie di "regole grammaticali". In particolare definisce l'insieme degli elementi del documento XML, le relazioni gerarchiche tra gli elementi, l'ordine di apparizione nel documento XML e quali elementi e quali attributi sono opzionali o meno.
- *XML Schema*: come la DTD, serve a definire la struttura di un documento XML. Oggi il W3C consiglia di adottarlo al posto della DTD stessa, essendo una tecnica più nuova ed avanzata. La sua sigla è XSD, acronimo di *XML Schema Definition.*
- *XLink*: serve a collegare in modo completo due documenti XML; al contrario dei classici collegamenti ipertestuali che conosciamo in HTML, XLink permette di creare link multidirezionali e semanticamente avanzati.
- *XSL* (acronimo di *eXtensible Stylesheet Language*): è il linguaggio con cui si descrive il foglio di stile di un documento XML.
- *XSLT* (dove la T sta per *Trasformations*) è una versione estesa dell'XSL. L'obiettivo principale per cui l'XSLT è stato creato è rendere possibile la trasformazione di un documento XML in un altro documento. È possibile anche aggiungere al documento trasformato elementi completamente nuovi o non prendere in considerazione determinati elementi del documento origine, riordinare gli elementi, fare elaborazioni in base al risultato di determinate condizioni, ecc. In termini strutturali, XSLT specifica la trasformazione di un albero di nodi in un altro albero di nodi.
- *XPath*: è un linguaggio con cui è possibile individuare porzioni di un documento XML e sta alla base di altri strumenti per l'XML come *XQuery*.

2.2.2 SOAP



SOAP Message



SOAP Message with Attachment

Il *Simple Object Access Protocol* (SOAP) [13] è un protocollo basato su XML per lo scambio d'informazioni in un ambiente distribuito che descrive un formato comune per trasmettere dati tra clients e services, fornendo le linee guida riguardo la formattazione del documento XML, ad esempio per separare i dati effettivi dai contenuti addizionali. L'elemento base è il messaggio SOAP, che consiste di una SOAP Envelope, un SOAP Header opzionale e un SOAP Body. L'envelope specifica la codifica utilizzata e gli spazi dei nomi (*namespaces*). Questi ultimi sono utilizzati per prevenire eventuali conflitti di nomi, giacché i tags di namespace diversi potrebbero avere lo stesso nome, ma un diverso significato. L'header, se presente, estende il messaggio in maniera modulare.

E' importante capire che un messaggio SOAP può muoversi da un client ad un servizio attraverso più punti intermedi, come avviene nelle architetture ESB. Ognuno di questi intermediari riceve e inoltra il messaggio fornendo servizi aggiuntivi, quali applicare politiche di sicurezza o trasformazione dei dati. I dati contenuti nell'header servono appunto per fornire informazioni utili ad eseguire queste operazioni. Un esempio classico è appunto quello della sicurezza, dove il corpo del messaggio è codificato e

nell'header sono presenti i dati necessari per procedere alla decodifica. Il SOAP Body infine contiene la parte principale del messaggio, chiamata Payload.
Un altro standard, *SOAP Message with Attachment* (SwA) [16], specifica il formato di un messaggio SOAP che comprende degli allegati. Ovviamente il livello di messaggio è indipendente dal livello di trasporto, quindi la busta SOAP può essere impacchettata per essere inviata via HTTP, SMTP, etc. , ma la struttura e il contenuto della busta SOAP rimarrà immutato.

2.2.3 WSDL

SOAP fornisce una struttura standard per organizzare i dati da inviare al servizio, ma come fa un client a conoscere il formato del contenuto di un messaggio SOAP richiesto dal un certo servizio? Per fornire queste informazioni il servizio espone un documento XML, chiamato WSDL, che contiene una descrizione delle interfacce dello stesso. Il documento WSDL è scritto utilizzando il *Web Service Description Language* (WSDL) [17]. Per scoprire la descrizione di un Web Service, un client deve trovare il suo WSDL. Un modo, peraltro il più comune, è quello di trovare il puntatore al documento WSDL nella registrazione del servizio, che può essere in un registro UDDI.

2.2.4 UDDI

Le specifiche *Universal Description, Discovery and Integration* (UDDI) [18] definiscono come pubblicare e reperire informazioni riguardanti un servizio in un registro. Più specificatamente definiscono uno schema UDDI e un'API. Il primo descrive le strutture dati XML da inserire nel registro mentre le API descrivono come i dati devono essere immagazzinati e come possono essere reperiti. Un "registro UDDI" è un registro conforme a questa specifica.

*2.2.5 WS-**

Gli standard definiti finora permettono ad un client di trovare i servizi necessari ed effettuare una richiesta che entrambe le parti possano interpretare correttamente. Ci poi sono una serie di altri standard che vengono definiti di "alto livello" e forniscono servizi

aggiuntivi in modo modulare. I principali sono:

- *WS-Security* [19]: uno standard sviluppato da Oasis-Open [62] per applicare politiche di sicurezza ai messaggi SOAP. Nello specifico indicano come possono essere garantiti integrità e confidenzialità. Inoltre include dettagli sull'uso di SAML e Kerberos, due protocolli di rete per l'autenticazione, e di certificati come X.509, uno standard per certificati a chiave pubblica.
- *WS-Addressing* [20]: uno standard che specifica come le informazioni relative a al trasporto del messaggio devono essere incapsulate nel messaggio stesso. Questo permette ad esempio di specificare a chi inviare la risposta del messaggio rendendo la comunicazione totalmente asincrona e svincolare la durata della comunicazione a quella della connessione.
- *WS-Reliability* [21]: uno standard per implementare strumenti che garantiscono la consegna del messaggio o dei messaggi rispetto alla politica scelta, come InOrder, AtLeastOnce, AtMostOnce etc.

2.3 La tecnologia J2EE per la gestione dei Web Service

Anche se sono progettati per essere indipendenti dal linguaggio e dalla piattaforma utilizzata, un linguaggio preferenziale per sviluppare Web Service e le applicazioni che li utilizzano è Java. La portabilità e l'interoperabilità delle applicazioni scritte in Java si sposano bene con gli obiettivi di interoperabilità dei Web Service. Un set di base di tecnologie Java per lo sviluppo di Web Service è integrato nel *Java 2 Platform Enterprise Edition* (J2EE). Queste tecnologie sono progettate per essere usate con XML e aderenti a standard come SOAP, WSDL e UDDI.
A seguire saranno presentate le principali implementazioni di specifiche in ambito Web Service in Java:

- JAXP per il parsing dei documenti XML.
- JAXB per il data binding.
- JAX-RPC per effettuare chiamate a procedure remote.
- JAX-WS, evoluzione di JAX-RPC.
- SAAJ per creare, modificare e inviare messaggi SOAP.
- JAXR per interfacciarsi ai registri come quelli UDDI.

2.3.1 JAXP

Java API for XML Processing (JAXP) [22,37] è una API utilizzata per processare documenti XML. Usando JAXP, un'applicazione può invocare un parser per eseguire il parsing di un documento XML. Un parser è un programma che effettua la lettura del documento XML e lo divide in blocchi discreti. Inoltre controlla che il documento sia ben formato e può validarlo rispetto ad uno XML Schema (XSD) oppure ad un *XML Document Type Definition* (DTD). E' evidente che la funzione del parser è estremamente importante poiché si occupa di presentare i dati contenuti nel documento XML all'applicazione.

I due classici approcci per processare documenti XML sono:

- *Simple API for XML processing* (SAX)
- *Document Object Model* (DOM)

SAX è un'API di basso livello il cui principale punto di forza è l'efficienza. Quando un documento viene parsato usando SAX, una serie di eventi vengono generati e passati all'applicazione tramite l'utilizzo di *callback handlers* che implementano l'handler delle API SAX. Gli eventi generati sono di livello molto basso e devono essere gestiti dallo sviluppatore che, inoltre, deve mantenere le informazioni necessarie durante il processo di parsing. Oltre ad un utilizzo piuttosto complicato, SAX soffre di due limitazioni di rilievo: non può modificare il documento che sta elaborando e può procedere alla lettura solo "in avanti": non può tornare indietro. Quindi, quello che è stato letto è perso e non è possibile recuperarlo.

DOM invece ha come punto di forza la semplicità d'utilizzo. Una volta ricevuto il documento, il parser si occupa di costruire un albero di oggetti che rappresentano il contenuto e l'organizzazione dei dati contenuti. In questo caso l'albero esiste in memoria e l'applicazione può attraversarlo e modificarlo in ogni suo punto. Ovviamente il prezzo da pagare è il costo di computazione iniziale per la costruzione dell'albero, ma soprattutto il costo di memoria dello stesso, che può diventare un problema cruciale in sistemi che devono processare grandi quantità di dati.

A questi rappresentanti delle due principali tecniche di rappresentazione dei dati XML si aggiungono altri due parser di recente concezione:

- StAX (Streaming API for XML) [45]

- TrAX (Transformation API for XML)

StAX è un *pull parser*. A differenza di SAX, che è un *push parser*, non riceve passivamente i segnali inviati all'handler per elaborarli, ma è l'utente a controllare il flusso degli eventi. Questo significa che il client richiede (*pull*) i dati XML quando ne ha bisogno e nel momento in cui può gestirli, a differenza del modello *push*, dove è il parser a inviare i dati non appena li ha disponibili a prescindere che l'utente ne abbia bisogno o sia in grado di elaborarli. Le librerie *pull parsing* sono molto può semplici delle *push parsing* e questo permette di semplificare il lavoro dei programmatori, anche per documenti molto complessi. Inoltre è bidirezionale, nel senso che oltre a leggere dati XML è anche in grado di produrli. Rimane il limite di poter procedere solo "in avanti" nell'elaborazione del documento XML.

TrAX, con l'utilizzo di un *XSLT stylesheet*, permette di trasformare l'XML. Inoltre poiché i dati vengono solitamente manipolati con SAX o DOM, questo parser può ricevere in ingresso sia eventi SAX che documenti DOM. Può anche essere usato per convertire i dati da un formato all'altro, infatti, è possibile prendere un documento DOM, trasformarlo e scriverlo su file, come prendere l'XML da file, trasformarlo e restituirlo in forma di documento DOM.

La tabella seguente riassume brevemente le caratteristiche principali dei parser presentati:

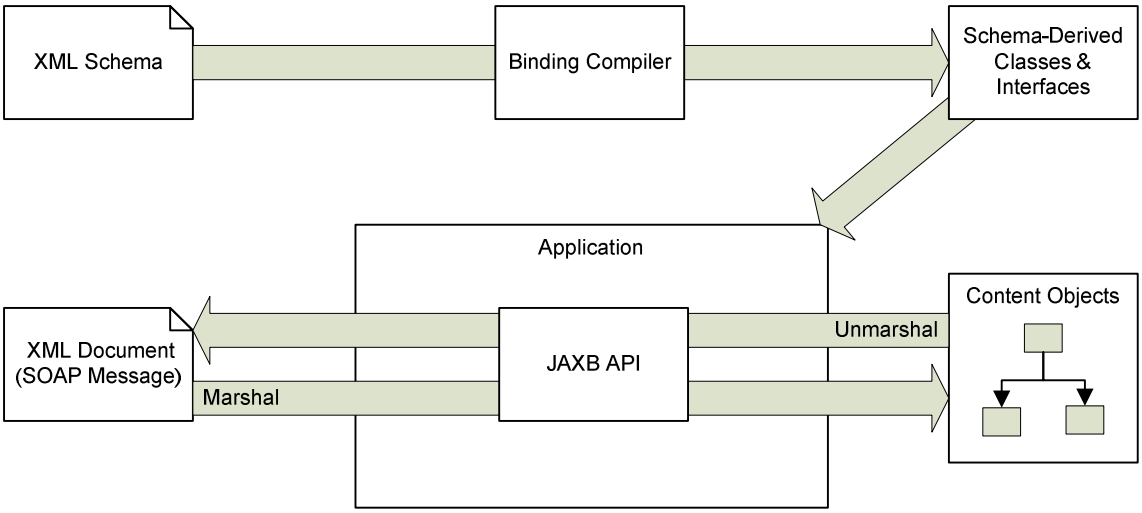
Feature	**StAX**	**SAX**	**DOM**	**TrAX**
Tipo di API	Pull, streaming	Push, streaming	In memory tree	XSLT Rule
Facilità d'uso	Alta	Media	Alta	Media
XPath Capability	No	No	Si	Si
Efficienza CPU Memoria	Buona	Buona	Dipende	Dipende
Solo "in avanti"	Si	Si	No	No
Legge XML	Si	Si	Si	Si
Scrive XML	Si	No	Si	Si
CRUD (Create, Read, Update, Delete)	No	No	Si	No

2.3.2 JAXB

La *Java API for XML Binding* (JAXB) [27,43] fornisce un modo per mappare un documento XML su di una collezione di oggetti Java basandosi sullo schema XML del documento. L'utilità sta nel fatto che l'applicazione così può lavorare direttamente sul contenuto Java invece che sul contenuto XML.

L'utilizzo di JAXB è semplice: si utilizza un binding compiler fornito con l'implementazione di JAXB per trasformare, o legare, lo schema XML del documento in un set di classi e interfacce. La combinazione di queste classi derivate e dell'implementazione del pacchetto javax.xml.bind delle specifiche di JAXB costituiscono il binding framework che permette di:

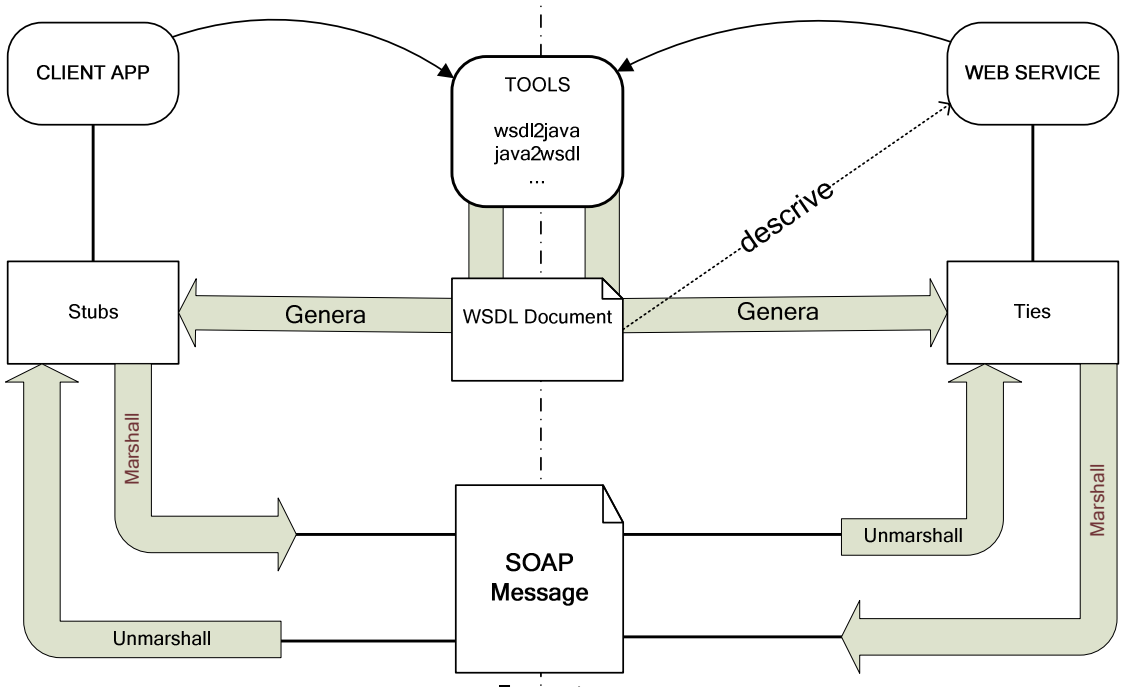
- Convertire il contenuto XML in un albero di oggetti Java (*marshalling*)
- Convertire un albero di oggetti Java in un documento XML (*unmarshalling*)
- Validare un albero di oggetti rispetto ad uno schema.



2.3.3 JAX-RPC

Lo standard *Java API for XML-based Remote Procedure Call* (JAX-RPC) [23,38,39] è una API Java per accedere ai servizi attraverso chiamate remote in XML basate su SOAP. Le API includono le funzionalità per effettuare chiamate RPC basate su XML

rispettando le specifiche SOAP 1.1. JAX-RPC permette ad un client basato su Java di chiamare metodi di Web Service in un ambiente distribuito. Dal punto di vista dell'applicazione, JAX-RPC offre un modo per chiamare un servizio. Dal punto di vista dello sviluppatore, offre un modo per rendere il servizio disponibile ad essere invocato dal client. JAX-RPC è progettato per nascondere la complessità del SOAP e, quando viene utilizzato per effettuare una chiamata RPC, non richiede di esplicitare il codice del messaggio SOAP. JAX-RPC quindi converte una chiamata RPC in un messaggio SOAP e lo invia al server. Dal lato server, JAX-RPC converte il messaggio SOAP e chiama il servizio. L'eventuale risposta del servizio segue il processo inverso, venendo convertita in un messaggio SOAP e inviata al client.



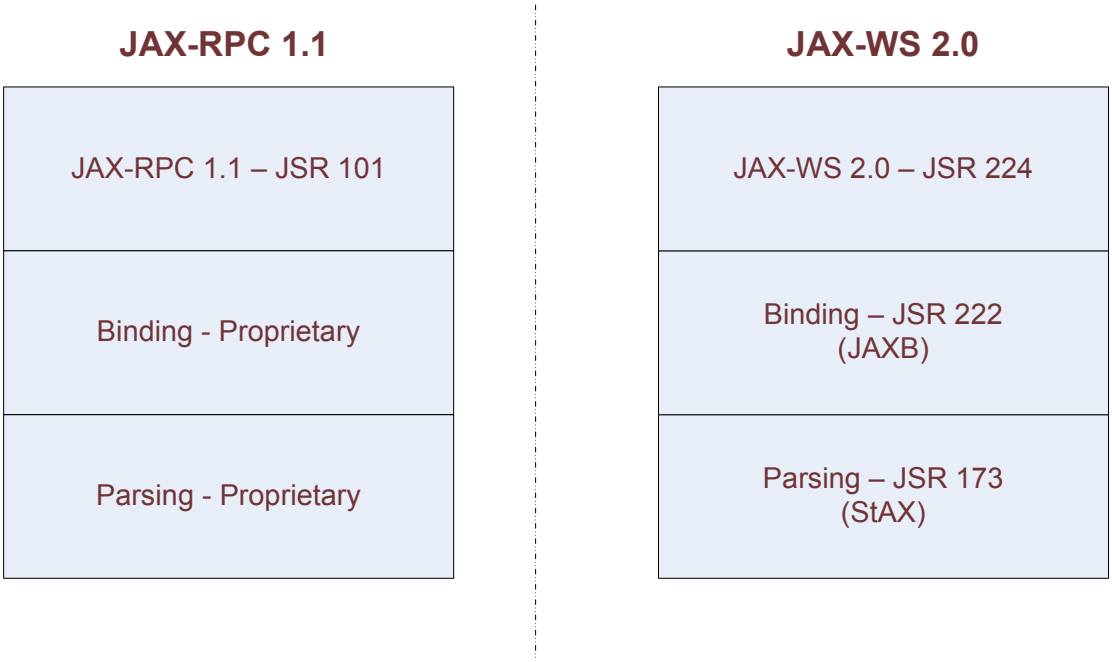
JAX-RPC fornisce numerosi modi per consentire ad un client di invocare un servizio. Il client può invocare il metodo remoto attraverso un oggetto locale chiamato *stub* oppure può usare un proxy dinamico per invocare il metodo o ancora utilizzare il JAX-RPC *Dynamic Invocation Interface* (DII). Il primo, ad ogni modo, rimane lo scenario classico d'utilizzo di JAX-RPC:

- Il client e service usano i tools forniti con JAX-RPC per generare gli *stubs* e *ties* partendo dal WSDL o dalle classi del servizio.
- Il client usa gli *stubs* per effettuare una chiamata di procedura remota come farebbe con un metodo locale.
- Lo *stub* si preoccupa di convertire la chiamata in un messaggio SOAP

(marshalling) e di spedirlo al server

- Dal lato server un oggetto chiamato *tie* o *skeleton* trasforma il messaggio SOAP in una chiamata al servizio (unmarshalling).
- Il processo viene invertito nel caso di una risposta del server per il client.

2.3.4 JAX-WS



Web Service Stack

La versione successiva a JAX-RPC 1.4 è stata chiamata JAX-WS 2.0, acronimo di *Java API for XML-based Web Service* [24,34,36,44].

Prima di discutere cosa è cambiato nel passaggio da JAX-RPC a JAX-WS vediamo cosa è stato mantenuto:

- JAX-WS continua a supportare SOAP 1.1 su HTTP 1.1, quindi l'interoperabilità non viene influenzata. Gli stessi messaggi possono ancora essere scambiati.
- JAX-WS continua a supportare WSDL 1.1, quindi le conoscenze acquisite su quello standard continuano a essere utili. JAX-WS supporta anche WSDL 2.0.

Cos'è cambiato?

- SOAP 1.2
    - JAX-WS, oltre a SOAP 1.1, supporta anche SOAP 1.2.

- XML/HTTP
    - Le specifiche WSDL 1.1 definiscono un binding HTTP, che consiste in un modo per inviare messaggi XML via HTTP senza SOAP. JAX-RPC ignorava l'HTTP binding mentre JAX-WS lo supporta.
- WS-I's Basic Profiles (Web Service interoperability)
    - JAX-RPC supporta WS-I's Basic Profile (BP) 1.0. JAX-WS supporta la versione successiva, BP 1.1.
- Nuove funzionalità Java
    - JAX-RPC si basa su Java 1.4, mentre JAX-WS si basa su Java 5.0. JAX-WS sfrutta le nuove funzionalità offerte dalla nuova versione di Java.
    - Java EE 5, il successore di J2EE 1.4, aggiunge il supporto a JAX-WS, ma mantiene anche quello per JAX-RPC, anche se può creare confusione agli utenti.
- Il modello di data mapping
    - JAX-RPC ha il suo modello di mapping, che copre circa il 90% dei modelli di tipi. Quelli non coperti sono mappati in javax.xml.soap.SOAPElement.
    - Il modello di mapping di JAX-WS è JAXB. JAXB promette di mappare tutti i modelli di tipo.
- Il modello di mapping di interfacce

  Il modello di mapping di interface di JAX-WS non è molto differente da quello di JAX-RPC, comunque:
    - Il modello di JAX-WS fa uso delle funzionalità di Java 5.
    - Il modello di JAX-WS aggiunge funzionalità asincrone.
- Il modello di programmazione dinamica
    - Il modello di client di JAX-WS è abbastanza differente da quello di JAX-RPC.
        - Introduce funzionalità orientate ai messaggi.
        - Introduce funzionalità asincrone dinamiche.
    - JAX-WS aggiunge anche un modello di server dinamico che JAX-RPC non aveva.

- MTOM (Message Transmission Optimization Mechanism) .
    - JAX-WS, tramite JAXB, aggiunge il supporto a MTOM, la nuova specifica per gli allegati. Microsoft non ha mai creduto nella specifica *SOAP with Attachment* (SwA), ma sembra che tutti supportino MTOM, quindi l'interoperabilità per gli allegati sembra raggiunta.
- Il modello degli handlers.
    - Il modello degli handlers è stato modificato da JAX-RPC a JAX-WS.
    - Gli handlers JAX-RPC si basano su SAAJ 1.2. Gli handlers JAX-WS si basano sulla nuova specifica SAAJ 1.3.

Vediamo adesso quali novità sono state introdotte con SOAP 1.2, XML/HTTP, WS-I Basic Profile e Java 5.

SOAP 1.2

Dal punto di vista del modello di programmazione non c'è grande differenza tra SOAP 1.1 e SOAP 1.2. Come programmatore Java, l'unico punto dove si incontrano differenze è nell'uso degli handlers. SAAJ 1.3 è stato aggiornato per supportare SOAP 1.2.

XML/HTTP

Come per i cambiamenti fatti per SOAP, non ci sono molte differenze da un punto di vista di modello di programmazione tra messaggi SOAP/HTTP e XML/HTTP. Anche per questi l'unica modifica rilevante per un programmatore, sta nell'uso degli handlers. Il binding ha il suo handler e il suo set di proprietà nel message context.

WS-I basic profiles

JAX-RPC 1.1 supporta WS-I *Basic Profile* (BP) 1.0 [14]. Il team del WS-I ha continuato a sviluppare BP 1.1 e gli associati AP1.0 [28] e SSBP 1.0 [29]. Questi nuovi profili chiarificano alcuni aspetti secondari e definiscono gli allegati. JAX-WS 2.0 supporta questi nuovi profili. Per la maggior parte, non intaccano il modello di programmazione Java. L'eccezione sono gli allegati. WS-I non solo chiarisce alcune questioni sugli allegati, ma definisce il loro tipo XML: wsi:swaRef.

Vediamo brevemente la storia di questi profili per semplificarne la comprensione. Il primo profilo base WS-I (BP 1.0) fece un ottimo lavoro per

chiarire le varie specifiche. Ma non tutto fu perfetto, soprattutto per quanto riguarda il supporto per i *SOAP with Attachments* (SwA). Nella seconda versione gli allegati furono separati dal profilo base (BP1.1) e alcune aspetti mancanti furono sistemati. A questo punto furono aggiunti due supplementi al profilo base: AP 1.0 e SSBP 1.0. AP 1.0 è l'*Attachment Profile* [28] che descrive come usare SwA. SSBP 1.0 è *Simple SOAP Binding Profile* [29], che descrive un Web Service che non supporta SwA (come il .NET di Microsoft).

Java 5

Ci sono numerosi cambiamenti nel linguaggio Java. JAX-WS ovviamente li sfrutta, come ad esempio l'uso di annotazioni nel codice per definire un servizio già nella classe che lo implementa.

Il cambio di nome da JAX-RPC a JAX-WS ha molte motivazioni, le principali sono:

- Molti sviluppatori sarebbero stati ingannati dal nome e avrebbero pensato che la specifica fosse ristretta all'uso di RPC e non di Web Service.
- JAX-WS usa JAXB per tutti i databinding. Questo ha introdotto molti problemi di compatibilità con la precedente versione visto che utilizza dei binding diversi. Il cambio di nome tende a rimarcare che le due tecnologie sono diverse e la migrazione tutt'altro che banale.

Il porting alle nuove specifiche è quindi un'operazione complessa che può comportare una profonda modifica dei progetti esistenti. Tale operazione deve essere quindi ben ponderata e studiata prima i procedere. Vediamo alcune motivazioni per cui adeguarsi ai nuovi standard.

Ragioni per rimanere a JAX-RPC

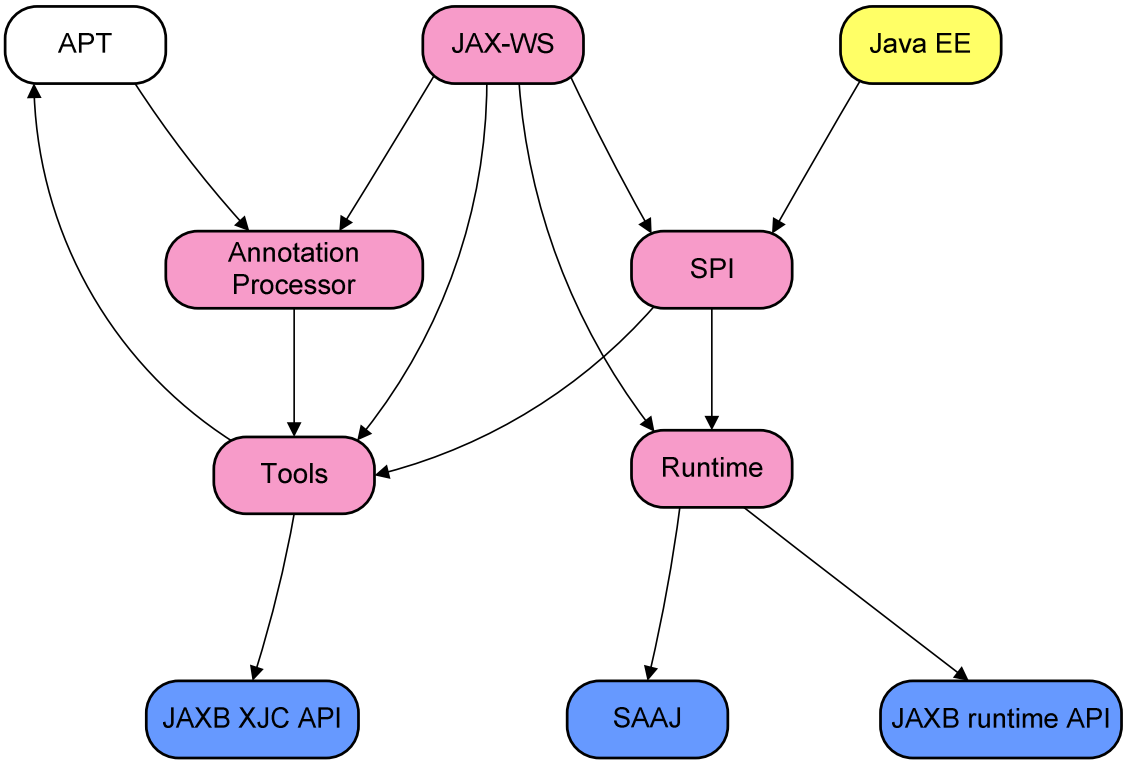
- Si vuole continuare ad utilizzare tecnologie basate su specifiche ben collaudate.
- Non si vuole passare a Java 5.
- Si vuole inviare messaggi SOAP codificati o creare WSDL in stile RPC/encoded.

Ragioni per passare a JAX-WS 2.0:

- Si vuole usare le nuove API orientate ai messaggi.
- Si vuole usare MTOM per inviare dati allegati.
- Si vuole un miglior supporto agli schemi XML attraverso JAXB.
- Supporto al modello di programmazione asincrona.

- Supporto al SOAP 1.2.
- Per avere la possibilità di eliminare il SOAP e utilizzare l'XML/ binding.

Nonostante ci possano essere delle buone motivazioni per ritardare il passaggio alle nuove specifiche, in futuro le applicazioni basate su JAX-RPC smetteranno di essere supportate, come già successo per AXIS, e il passaggio sarà a quel punto inevitabile.
Vediamo l'architettura alla base di JAX-WS:



Runtime:

Il modulo Runtime è disponibile all'applicazione a tempo di esecuzione e rappresenta il cuore del framework dei Web Service.

Tools:

Sono gli strumenti per convertire i WSDL e le classi o sorgenti Java in servizi.

APT.

Uno strumento di Java SE: un framework per elaborare le annotazioni.

Annotation Processor:

Un APT Annotation Processor per elaborare i sorgenti Java con annotazioni javax.jws.* e convertirli in Web Service.

Runtime SPI:

Una parte di JAX-WS che definisce il contract tra il JAX-WS RI (Reference Implementation) runtime e Java EE.

Tools API

Una parte di JAX-WS che definisce il contract tra il JAX-WS RI tools e Java EE.

JAXB XJC-API

Lo schema compiler.

JAXB runtime-API

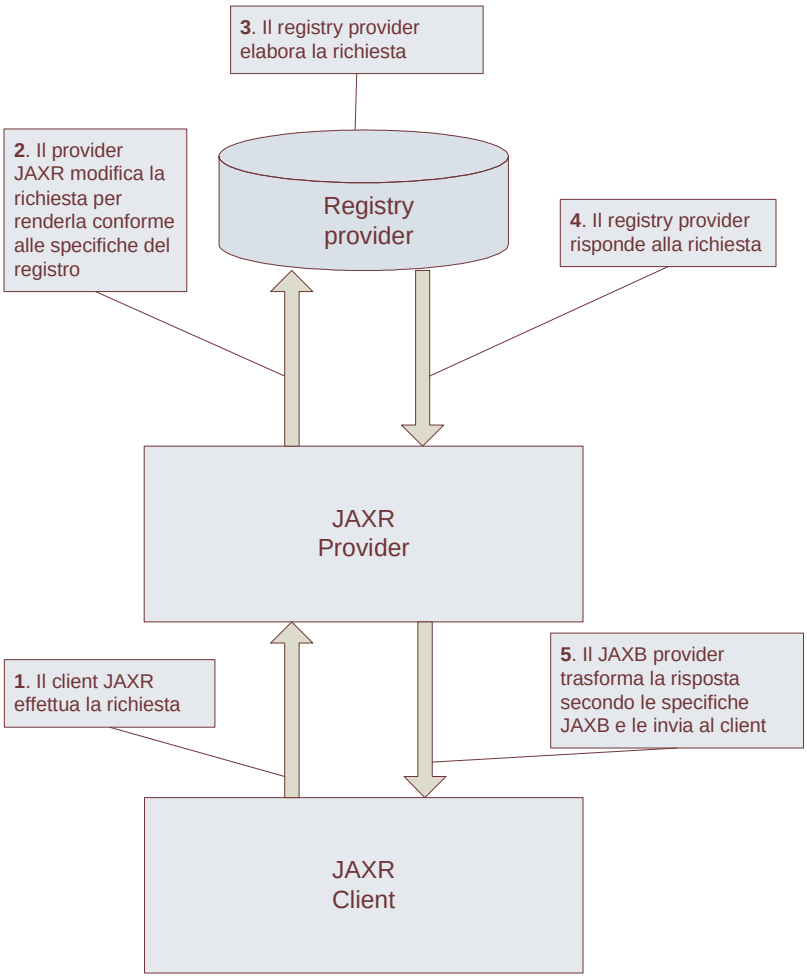
Una parte di JAX-WS che definisce il contract tra il JAX-B RI e JAX-WS RI.

2.3.5 SAAJ

Nella descrizione di JAX-RPC e JAX-WS è stata più volte nominata questa importante API. *SOAP with Attachment API for Java* (SAAJ) [25,40] è una API per la creazione, manipolazione e invio di messaggi SOAP. Altre API XML usano SAAJ per svolgere le loro funzioni, come appunto JAX-RPC e JAX-WS. SAAJ 1.3 è conforme a SOAP 1.2 e SwA, quindi può essere utilizzato per creare e inviare messaggi SOAP con o senza allegati.

2.3.6 JAXR

La *Java API for XML Registry* (JAXR) [26,42] è un'API Java che può essere utilizzata per avere un accesso standard a registri come UDDI. Le specifiche JAXR forniscono dunque un'interfaccia standard per costruire clients indipendenti dal registro utilizzato.

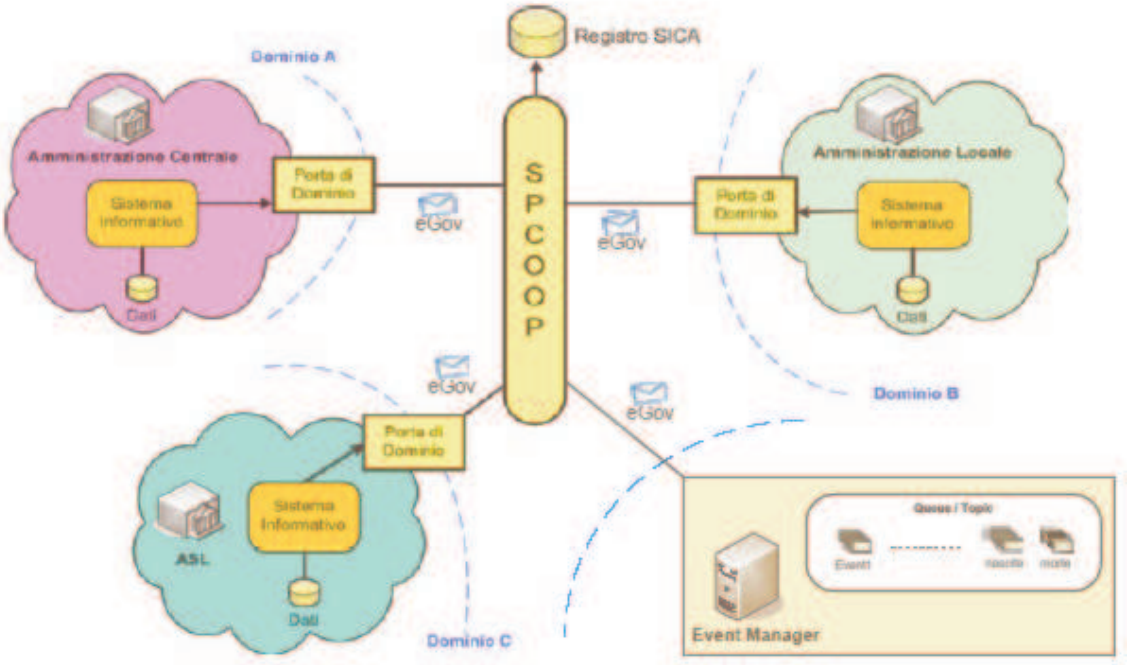

3. Il registry provider elabora la richiesta
2. Il provider JAXR modifica la richiesta per renderla conforme alle specifiche del registro
Registry provider
4. Il registry provider risponde alla richiesta
JAXR Provider
1. Il client JAXR effettua la richiesta
5. Il JAXB provider trasforma la risposta secondo le specifiche JAXB e le invia al client
JAXR Client

3. Architetture per la Cooperazione Applicativa

In questo capitolo valuteremo alcuni strumenti di recente sviluppo o attualmente in fase di completamento che implementano architetture o specifiche presentate nel capitolo precedente.
Nel primo paragrafo introdurremo l'attuale architettura di OpenSPCoop.
Nel secondo paragrafo ci addentreremo in Synapse, promettente progetto di Service Mediation di Apache, sviluppato per implementare un Enterprise Service Bus basato su XML e Web Service in sinergia con Axis2. La valutazione di Synapse è stata fatta prima del rilascio della versione 1.0, quando ancora non era stato promosso a sottoprogetto di Apache. Le funzionalità chiave di nostro interesse comunque non sono cambiate, quindi questa valutazione è da ritenersi valida anche per la versione 1.0 rilasciata a Giugno del 2007.
Nel secondo paragrafo studieremo un'architettura basata su un broker JMS,

3.1 L'architettura del Progetto OpenSPCoop

La progettazione di OpenSPCoop [05] è stata effettuata cercando di fare cooperare tutti i soggetti descritti dalle specifiche CNIPA, quali le porte di dominio, il registro dei servizi ed il gestore degli eventi. La figura seguente illustra una visione d'insieme di tutte le componenti distribuite di OpenSPCoop.



Come si può notare, l'unico mezzo di accesso per un Sistema Informativo Locale, per arrivare a consultare un altro SIL è la busta e-Gov, attraverso la mediazione della propria porta di dominio. Nella figura viene illustrato, tra i vari componenti, anche il gestore degli eventi, che sarà utilizzabile, da un SIL, per pubblicazioni o ricezioni di eventi attraverso la solita mediazione della porta di dominio ed attraverso l'utilizzo della busta e-Gov. Per avere una visione progettuale di ogni singolo componente, viene elencata di seguito una descrizione funzionale generica di ognuno.

La Porta di Dominio è costituita dal componente periferico OpenSPCoop PdD, ospitato da macchine appositamente create ed impostate nel dominio di gestione. La caratteristica principale di questo componente è il disaccoppiamento totale della componente di ricezione e consegna delle buste da quella di controllo e gestione delle buste SOAP/e-Gov in transito, tramite l'uso di code persistenti per il mantenimento dei dati, realizzate tramite JMS. Sarà OpenSPCoop PdD ad interagire direttamente sia con i SIL che con i componenti centrali dell'infrastruttura (Gestore degli eventi e Registro dei servizi). L'adozione di una porta di dominio, strumento di mediazione della comunicazione tra SIL, permette di assicurare un maggior controllo dal punto di vista della sicurezza e del tracciamento dei messaggi. E' stato adottato un componente di

integrazione SIL-to-PdD composto da un proxy dinamico, che richiede la presenza di una tabella per la registrazione di porte delegate e applicative necessarie a OpenSPCoop PdD per sapere in che modo gestire le richieste, sia in uscita (porte delegate), che in ingresso (porte applicative).

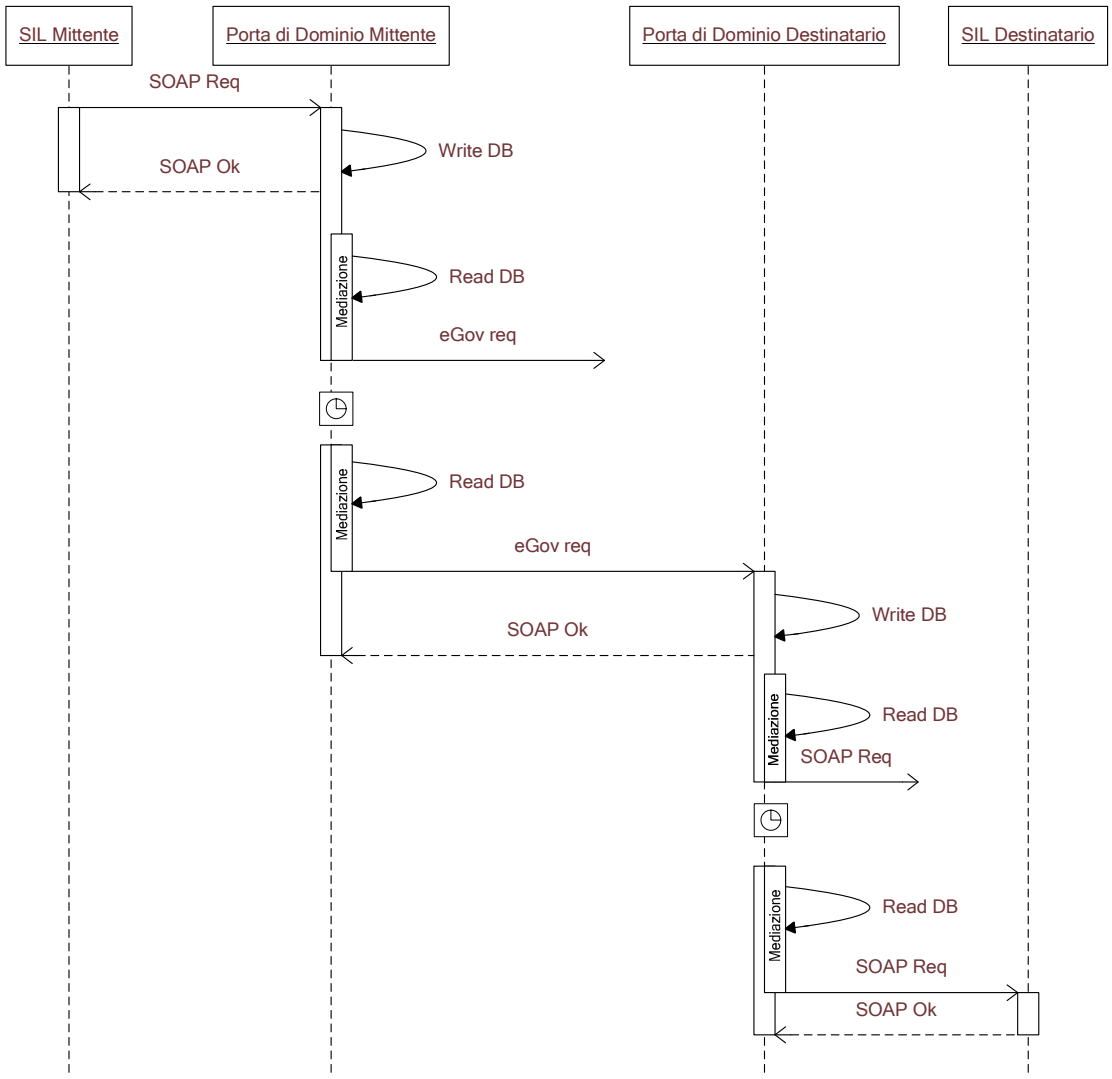
Le porte di dominio, oltre che dialogare con altre porte di dominio, potranno accedere ad un gestore di eventi direttamente tramite listener in ascolto su code (o topic) interne al gestore, ottenendo una maggiore efficienza di prestazioni, o anche interagendoci tramite scambio di buste e-Gov, vedendo il gestore come un servizio che si occupa di trasmettere le comunicazioni di evento tra i soggetti interessati (sistemi informativi locali). Il Gestore degli eventi è strutturato quindi come una porta applicativa, il cui compito è ricevere e consegnare messaggi contenenti eventi, utilizzando la busta SPCoop.

Un componente molto importante della PdD, è quello che si occupa di gestire la logica SPCoop. Il componente in questione, nominato Modulo di Controllo, viene chiamato in causa all'arrivo di una richiesta di servizio da parte di un SIL, prima di redirigere il messaggio verso un'altra porta di dominio, e all'arrivo di una busta SPCoop che includa una richiesta di servizio destinata ad un SIL. Tra i vari compiti del modulo di controllo vi sono tutti gli aspetti di gestione della specifica CNIPA riguardante la busta SPCoop. Viene quindi effettuata una validazione della busta (controllo semantico e sintattico), un controllo di autorizzazione, una eventuale gestione dei duplicati (affidabilità), una gestione dei riscontri (alla ricezione dei messaggi e alla spedizione della conferma ricezione), una gestione del tracciamento del messaggio e una gestione delle eccezioni. Il modulo di controllo si occupa anche di interagire con il registro dei servizi per conoscere le varie informazioni del servizio, sia nel momento dell'imbustamento, per effettuare la creazione della busta, sia al momento dello sbustamento, per effettuare una validazione della busta. Inoltre, durante l'accesso al registro per ottenere informazioni di imbustamento, verrà effettuata una ulteriore ricerca per conoscere l'indirizzo fisico della porta di dominio destinataria a cui spedire la busta SPCoop prodotta.

Il Registro dei Servizi è costituito da un server LDAP, che mantiene le informazioni relative ai servizi e ai diritti di accesso dei Sistemi Informativi Locali ai servizi. E' stato inserito nell'architettura un registro UDDI compatibile con le specifiche SPCoop ed utilizzato da OpenSPCoop PdD per identificare le caratteristiche del Servizio, come il

profilo (sincrono, asincrono, simmetrico, asimmetrico) e gli altri parametri di quality of service.

L'architettura presentata permette di gestire comunicazioni OneWay, sincrone e asincrone con presa in consegna dei messaggi scambiati. Vediamo come questa architettura si comporta negli scenari più rappresentativi.



Quello sopra rappresentato è il diagramma di sequenza di una comunicazione OneWay con possibili errori di comunicazione.

Il diagramma di sequenza è uno schema in *Unified Modeling Language* (UML) utilizzato per descrivere uno scenario. Uno scenario è una determinata sequenza di azioni in cui tutte le scelte sono state già effettuate; in pratica nel diagramma non compaiono scelte, né flussi alternativi. Il diagramma di sequenza descrive le relazioni

che intercorrono, in termini di messaggi, tra Attori, Oggetti di business, Oggetti od Entità del sistema che si sta rappresentando. Le entità che partecipano allo scenario sono rappresentate da linee verticali e parallele mentre, con frecce orizzontali, sono rappresentati i messaggi scambiati tra loro, nell'ordine in cui sono emessi. Le frecce continue sono messaggi di richiesta, con punte piene sono sincroni, con mezza punta sono asincroni. Le frecce tratteggiate sono messaggi di risposta. Le frecce rientranti sono chiamate interne, come ad esempio la *Write DB* del nostro diagramma, che indica l'operazione di salvataggio del messaggio su database. Gli orologi stilizzati rappresentano una sospensione dell'elaborazione. I diagrammi qui rappresentati non intendono essere esaustivi, ma vogliono solo rappresentare la porzione di realtà di nostro interesse, allo scopo di chiarificare i concetti espressi.

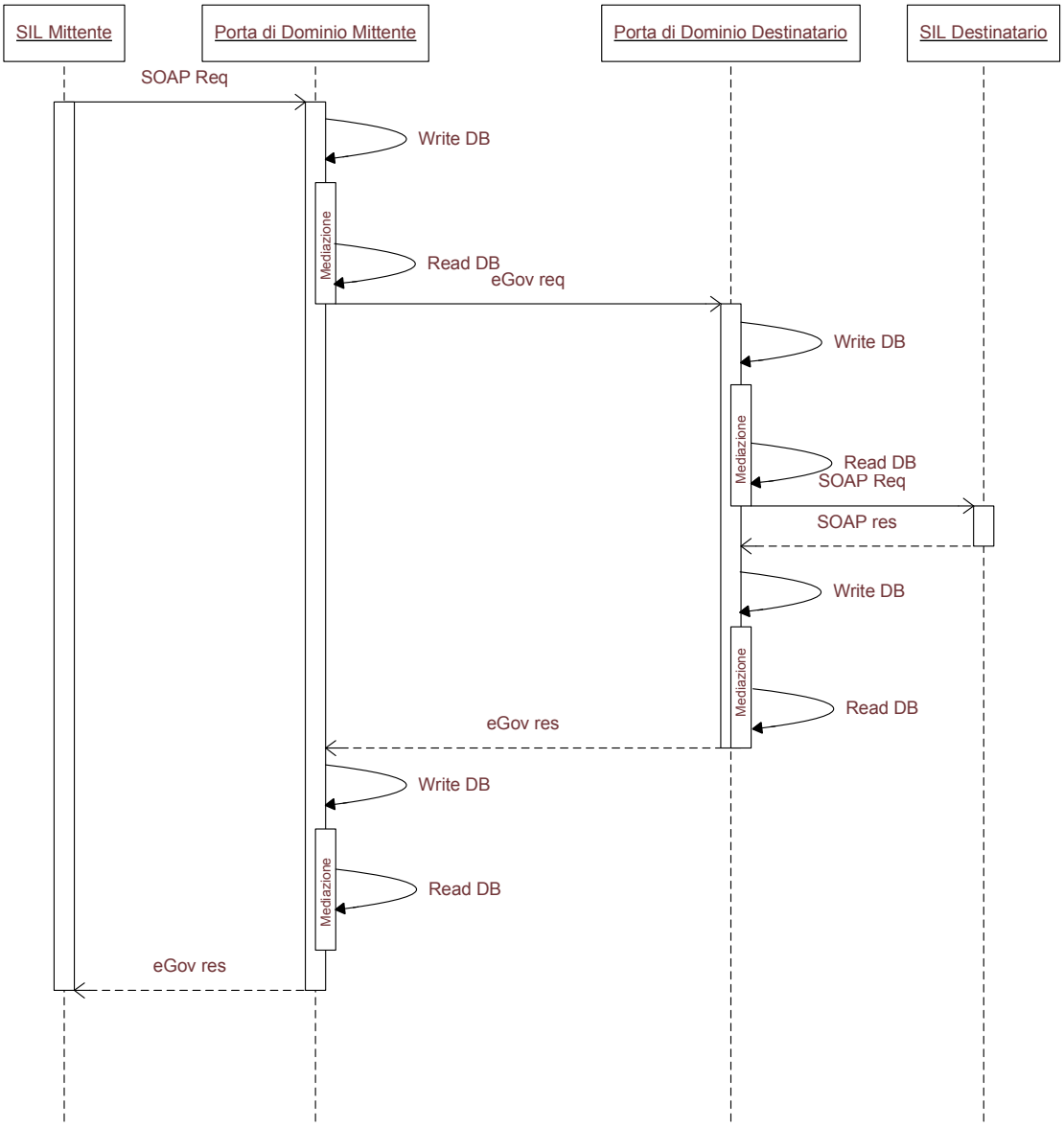
Tornando al diagramma sopra rappresentato, si notano le quattro entità coinvolte nello scenario: i due Sistemi Informativi Locali, mittente e destinatario, e le due Porte di Dominio che fungono da interfaccia al sistema OpenSPCoop e mediano i messaggi. Il SIL Mittente comincia la comunicazione inviando un messaggio SOAP. In comunicazione OneWay, il client invia il messaggio di richiesta aspettando come risposta un messaggio SOAP di ack, se la comunicazione è andata a buon fine, o un messaggio SOAP di fault, altrimenti.

La Porta di Dominio che ha ricevuto la richiesta dal SIL Mittente non conferma subito la ricezione, ma ritarda l'invio dell'ack SOAP finché non ha garantendo così la persistenza, salvando il messaggio su database. Solo dopo aver salvato il messaggio risponderà al mittente, il quale considererà consegnato con successo il messaggio. La Porta di Dominio procede quindi alla mediazione, processo che verrà discusso più approfonditamente nei capitoli successivi, e cercherà di inoltrare la Busta eGov creata alla Porta di Dominio destinataria. Se dovesse fallire per problemi di comunicazione, tenterà l'invio in un momento successivo.

Il sistema quindi è tollerante ad errori di comunicazione o di sistema ed ogni nodo intermedio prende in consegna il messaggio notificandolo al nodo precedente. Il tempo di risposta, in questo tipo di scambio di messaggi, è ottimale per il SIL Mittente. Non appena la Porta di Dominio Mittente può garantire la presa in consegna del messaggio, notifica la ricezione al SIL Mittente, che può considerare conclusa la comunicazione nonostante il messaggio sia ben lontano dall'essere consegnato effettivamente al SIL

Destinatario. Lo stesso vale anche per i nodi intermedi. Il sistema di mediazione, in una comunicazione OneWay, tende quindi ad ottimizzare i tempi di risposta, riducendo al mimino la durata delle connessioni tra i nodi. Questo riduce conseguentemente i casi di errori di comunicazione. Il rovescio della medaglia è rappresentato dal tempo totale della comunicazione. Considerando solo il tempo di presa in consegna, il processo qui presentato aggiunge almeno quattro accessi a database per la scrittura e lettura del messaggio, senza considerare gli accessi per scrivere e leggere le informazioni di corredo per coordinare l'intero processo.

Vediamo le comunicazioni e le operazioni principali in una comnicazione sincrona:



Anche in questo caso vengono fornite funzionalità simili alla comunicazione OneWay. La gestione di fallimenti, non mostrata nel diagramma, è simile a quella del OneWay, avendo memorizzato il messaggio. Dovendo aspettare la risposta, il canale del SIL Mittente ovviamente rimane aperto in attesa della risposta SOAP. In questo scenario risulta ancora più evidente l'overhead introdotto dagli accessi a database, che sostanzialmente raddoppiano dovendo gestire anche la risposta, gravando per il SIL Mittente sul tempo totale che intercorre dall'invio del messaggio alla ricezione della risposta.

3.2 Web Service Framework

I Web Service Framework sono studiati per fungere da endpoint di una comunicazione. Questo significa che il loro compito si limita alla ricezione, elaborazione e replica dei messaggi. Axis [08], attuale Web Service Framework utilizzato da OpenSPCoop, è basato su un modello RPC e non supporta le ultime specifiche SOAP e non ha la capacità nativa di eseguire mediazione tra servizi. Di Axis vengono solo utilizzate le funzionalità di ricezione e manipolazione dei messaggi tramite SAAJ. Tutta la logica di gestione della mediazione è implementata separatamente.

Anche i nuovi Web Service Framework, quali Axis2 [09] e CFX [10], pur supportando nuovi standard non sono in grado di gestire comunicazioni più complesse. Quindi le architetture fornite dai Web Service Framework ci consentono di effettuare comunicazioni punto a punto, e sarà necessaria l'implementazione di una architettura più evoluta per trattare le comunicazioni previste da SPCoop.

Axis è stato uno dei Web Service Framework di maggior rilievo della sua generazione. Quando si voleva valutare un Web Service Framework, solitamente lo si paragonava ad Axis. L'evoluzione delle specifiche esistenti e l'introduzione di nuove, hanno spinto il team di sviluppo a cogliere l'occasione per ripensare l'intera architettura di Axis e risolvere alcune problematiche sorte negli anni di utilizzo e sviluppo. Questo processo di rinnovo ha dato vita ad Axis2, progetto che non mantiene la compatibilità con il predecessore, ed ha decretato l'abbandono dello sviluppo di Axis. Da qui nasce la necessità di migrare il progetto OpenSPCoop verso un nuovo Web Service Framework, sia per il motivo appena descritto, l'interruzione dello sviluppo di Axis, sia per poter

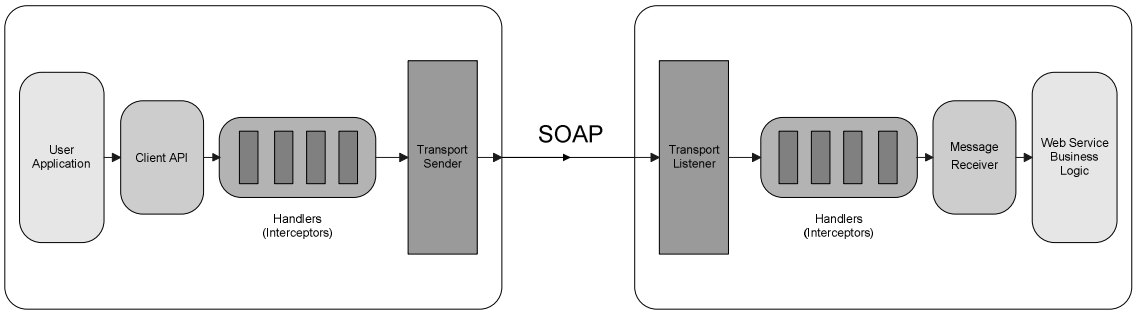
contare sull'appoggio di uno strumento conforme alle specifiche più recenti.
Nei paragrafi successivi saranno valutati i due progetti open source di Web Service Framework di nuova generazione più promettenti, Axis2 e CXF, le specifiche che implementano e le performance ottenute, per poter decidere quale sia il più adatto a sostituire Axis.

3.2.1 AXIS2

Il progetto Apache Axis2 [08] è un'implementazione basata su Java sia della parte client che di quella server del modello di Web Service. Progettato partendo dall'esperienza maturata dal progetto Axis 1.0, 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. Axis2 permette con facilità di:

- Inviare messaggi SOAP
- Ricevere ed elaborare messaggi SOAP
- Creare Web Service partendo da POJO (Plain Old Java Object)
- Creare le classi che implementano sia il lato client sia quello server partendo dal WSDL
- Recuperare facilmente il WSDL di un servizio.
- Inviare e ricevere messaggi SOAP con Attachments.
- Creare e utilizzare servizi basati su REST (REpresentational State Transfer)
- Creare o utilizzare servizi che sfruttano specifiche quali WS-Security, WS-ReliableMessaging, WS-Addressing, WS-Coordination, e WS-Atomic Transaction
- Usare l'architettura modulare di Axis2 per aggiungere il supporto a nuove specifiche quando rilasciate

Dal punto di vista di Axis2, l'architettura atta a gestione lo scambio di messaggi è questa:



Supponendo che su ogni lato (client e server) sia in esecuzione Axis2, il processo per lo scambio di messaggi può essere sintetizzato in questo modo:

- Il mittente crea un messaggio SOAP con Axiom.
- Gli “handlers” Axis2 si occupano di tutte le azioni necessarie al messaggio, come crittografarlo in conformità con WS-Security.
- Il Transport sender invia il messaggio.
- Sul lato ricevente, il Transport listener riceve il messaggio.
- Il Transport listener inoltra il messaggio alla catena di “handlers” definite per il lato server.
- Una volta eseguite le operazioni della fase di “pre-dispatch” il messaggio viene passato ai dispatcher che lo inoltreranno alla giusta applicazione.

In Axis2 tutte queste azioni sono raggruppate in “phases”, molte delle quali pre-definite nell’architettura, come la fase “pre-dispatch”, “dispatch”, “message processing”. Ogni fase è una collezione di “handlers”. Axis2 permette di controllare quali handlers si inseriscono in ogni fase e il loro ordine di esecuzione all’interno di una fase. E’ anche possibile definire delle fasi e degli handlers aggiuntivi. Uno o più handler compongono un “modulo” che può essere inserito in Axis2. Il concetto di modulo rappresenta il cuore dell’espandibilità di Axis2. Ognuno di questi moduli svolge un compito, come ad esempio Sandesha2 che implementa il WS-ReliableMessaging o Rampart2 che implementa il WS-Security, e possono essere inseriti all’interno della catena di handlers in modo da gestire i messaggi e aggiungere le funzionalità implementate in maniera totalmente trasparente all’utente. Inoltre, grazie al modello di deployment di Axis2, è possibile pubblicare un modulo e “agganciarlo” a uno o più servizi senza dover fermare Axis2.

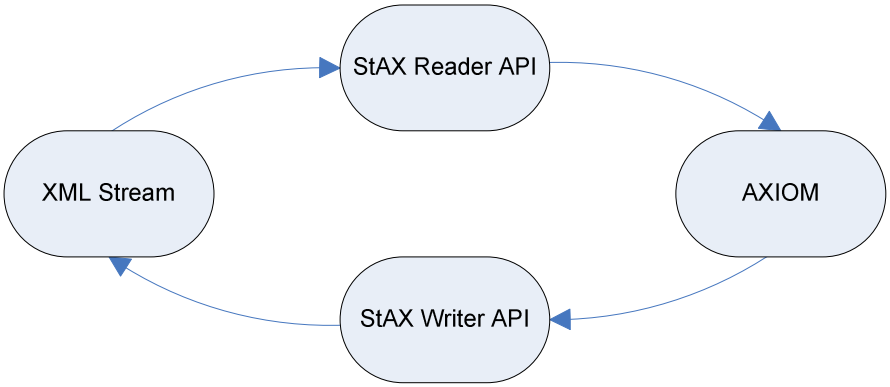
3.2.1.1 AXIOM

Come anticipato nell'introduzione, Axis2 fornisce un Object Model completo. Quando si è parlato di JAXP, sono stati individuati due modelli di processare un documento XML:

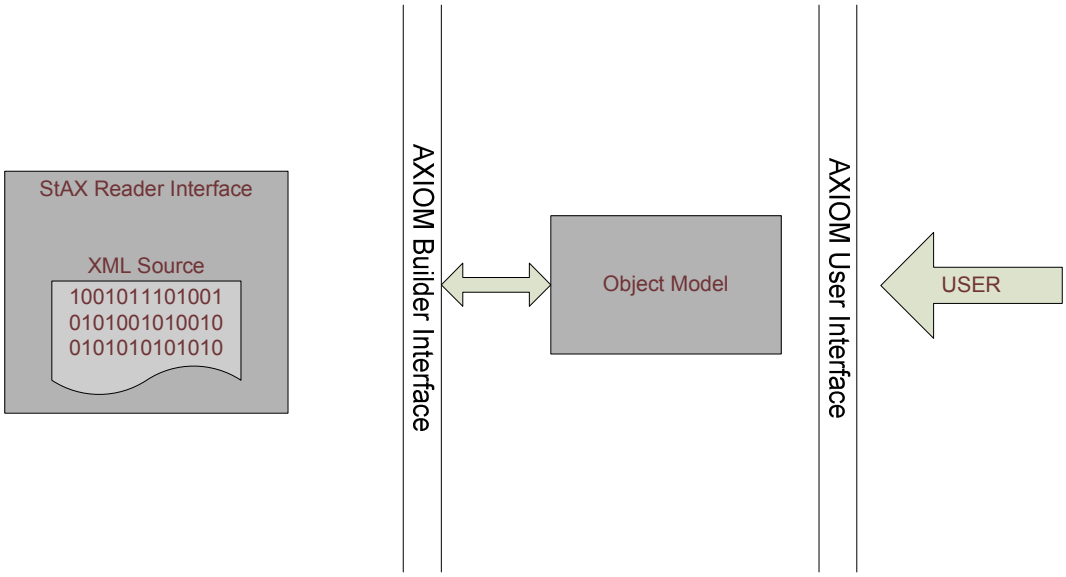
- *Tree-based*: caricato in memoria e facile da gestire per il programmatore. Costo di memoria elevato e costo computazionale per la costruzione degli oggetti.
- *Event-based*: gestisce direttamente lo stream con ottime prestazioni. Difficile gestione per il programmatore, si legge solo "in avanti" e non si può tornare indietro.

Uno degli obiettivi del recente StAX era quello di fornire un modello Event-based, quindi dalle ottime performance, facile da utilizzare come un Tree-based, ma il limite invalicabile di un modello basato sugli eventi è quello che gli eventi passati non sono riproducibili, lo stream può essere processato una sola volta.

AXIS Object Model, a.k.a. OM (AXIOM) [11,46,47,48] è stato introdotto proprio per ovviare a questo limite e fondere finalmente il modello Event-based con quello Tree-based. AXIOM si basa sulle API StAX per l'input e l'output dei dati. Il punto innovativo di questo modello è il supporto alla costruzione del modello differita. Questo modello, infatti, non costruisce l'intero documento una volta ricevuto, come avviene ad esempio per il DOM, ma ritarda la costruzione fino a quando non è necessaria. Il modello ad oggetti contiene solo quello che è stato costruito, il resto è ancora nello stream. Inoltre AXIOM può generare eventi StAX partendo da qualsiasi punto del documento ricevuto, indipendentemente dal fatto che sia stato processato o meno. Ancora, AXIOM può generare questi eventi StAX costruendo o meno il modello corrispondente. Questo in AXIOM è chiamato caching.



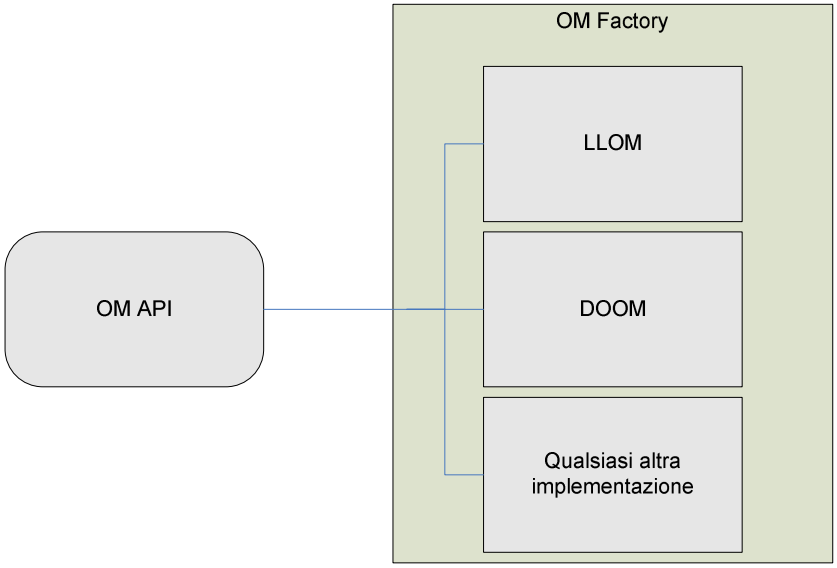
Come mostrato in figura, AXIOM accede allo stream XML attraverso StAX. L'implementazione del parser StAX utilizzata da AXIOM è quella Woodstox, ma ovviante è possibile sostituirla con qualsiasi altra implementazione delle API StAX.



AXIOM "vede" l'input stream XML attraverso lo stream reader di StAX, che viene incapsulato dall'interfaccia del costruttore fornito. Ci sono più implementazioni del costruttore, ad esempio StAXOMBuilder che costruisce completamente l'XML info-set model, oppure lo StAXSOAPModelBuilder che costruisce l'object model specifico per SOAP. Ogni implementazione supporta la costruzione differita ed il caching.

L'API AXIOM lavora sopra l'interfaccia dei costruttori e si interfaccia all'utente attraverso una semplice, ma potente, API. Fornisce la massima flessibilità potendo cambiare l'implementazione del costruttore o dell'object model in maniera indipendente dal resto. AXIOM è fornito con due implementazioni di questa API. La prima è basata su un modello a liste collegate (Linked List Object Model, LLOM) considerata

l'implementazione di default. L'altra, conosciuta come DOOM (DOM over OM) fornisce un livello di interfaccia DOM su AXIOM. DOOM consente così di poter sfruttare le implementazioni di altre API sviluppate per DOM, come ad esempio WSS4J usato per implementare WS-Security.
Per lavorare con più implementazioni AXIOM usa il concetto di Factory. La factory è concepita in modo che se non è specificata un'implementazione da utilizzare, sarà usata quella di default dal class path.



3.2.2 CXF

Divenuto uno dei Web Service Framework per Java, Apache CXF è un framework open source sviluppato dalla Apache Software Foundation. Nato dalla prosecuzione e fusione di due progetti open source, il progetto XFire di Codehaus [56] e il progetto Celtix di IONA [57], CXF consente la creazione di Web Service usando tecnologie Java. Fornisce supporto per una varietà di standard Web Service, inclusi WS-I Basic Profile 1.0, WS-Addressing, WS-Policy e WS-Security.
CXF permette di utilizzare più protocolli di trasporto, come e JMS. Il formato dei messaggi supportati includono SOAP, XML, RESTFul e CORBA. Oltre a supportare lo standard JAXB per il databinding, CXF fornisce anche il databinding Aegis [58].

CXF è JAX-WS compatibile e la versione 2.0 ha passato con successo il Test Compatibily Kit (TCK) per JAX-WS 2.0. Quest'ultimo fa un uso intenso di Java 5 e delle annotazioni, quindi richiede JDK/JRE 5.0 e successivi per funzionare.
L'architettura alla base di CXF può essere suddivisa nel seguenti componenti:

1. Bus: costituisce la spina dorsale dell'intera architettura.
2. Messaging & Interceptors: forniscono il livello di messaggio e di pipeline sopra i quali sono costruite molte funzionalità.
3. Front ends: forniscono un modello di programmazione per creare i servizi (es. JAX-WS).
4. Services: i servizi ospitano un Service model, un modello basato sul WSDL, che descrive il servizio stesso.
5. Transport: *Destinator* e *Conduit* forniscono la necessaria astrazione per garantire la neutralità di CXF rispetto al trasporto utilizzato.

Il bus si occupa di gestire e fornire risorse condivise a CXF. Può essere facilmente esteso per aggiungere risorse e servizi propri. L'implementazione di default del bus è basata su Spring [59], un framework open source per lo sviluppo di applicazioni web su piattaforma Java, che lega i componenti a runtime tra loro. CFX è costruito su un generico livello di messaggio comprendente Interceptors e InterceptorsChains. L'Interceptor costituisce il mattone fondamentale per implementare funzionalità. Può essere configurato in qualsiasi punto del processamento può essere concatenato ad altri Interceptors, formando una InterceptorsChain. Esempi di Interceptors predefiniti sono:

- Un Interceptor che parsa solo l'header di un messaggio SOAP in elementi DOM.
- Un WS-Security Interceptor che decripta o autentica un messaggio in ricezione
- Un Interceptor che esegue il data binding di un messaggio in uscita

Gli Interceptors sono uni-direzionali e non sono informati del fatto che il messaggio che stanno elaborando sia di richiesta, di risposta o di errore. Per comunicare con CXF viene fornito un frontend di programmazione. Il principale frontend è JAX-WS e permette di definire i servizi e configurare gli Interceptors per ognuno di essi. L'implementazione di SAAJ fornita con CXF è quella sviluppata dalla SUN, ma è possibile sostituirla con quella preferita semplicemente sostituendo la libreria corrispondente.

3.3 Synapse

Apache Synapse [49] è un broker di mediazione utilizzabile per costruire ESB (Enterprise Service Bus). Synapse permette di porsi come intermediario tra due o più servizi in maniera semplice ed efficiente secondo un modello basato su ruoli.

Synapse offre una capacità di mediazione ampia, incluso il supporto a vari standard aperti, inclusi XML, XSLT, XPath, SOAP, , JMS, WS-Security, WS-ReliableMessaging, WS-Addressing, SMTP.

Synapse incorpora un buon numero di funzioni predefinite, senza necessità di programmare, ma fornisce la possibilità di ampliare questa gamma usando linguaggi di programmazione diffusi come Java e Javascript.

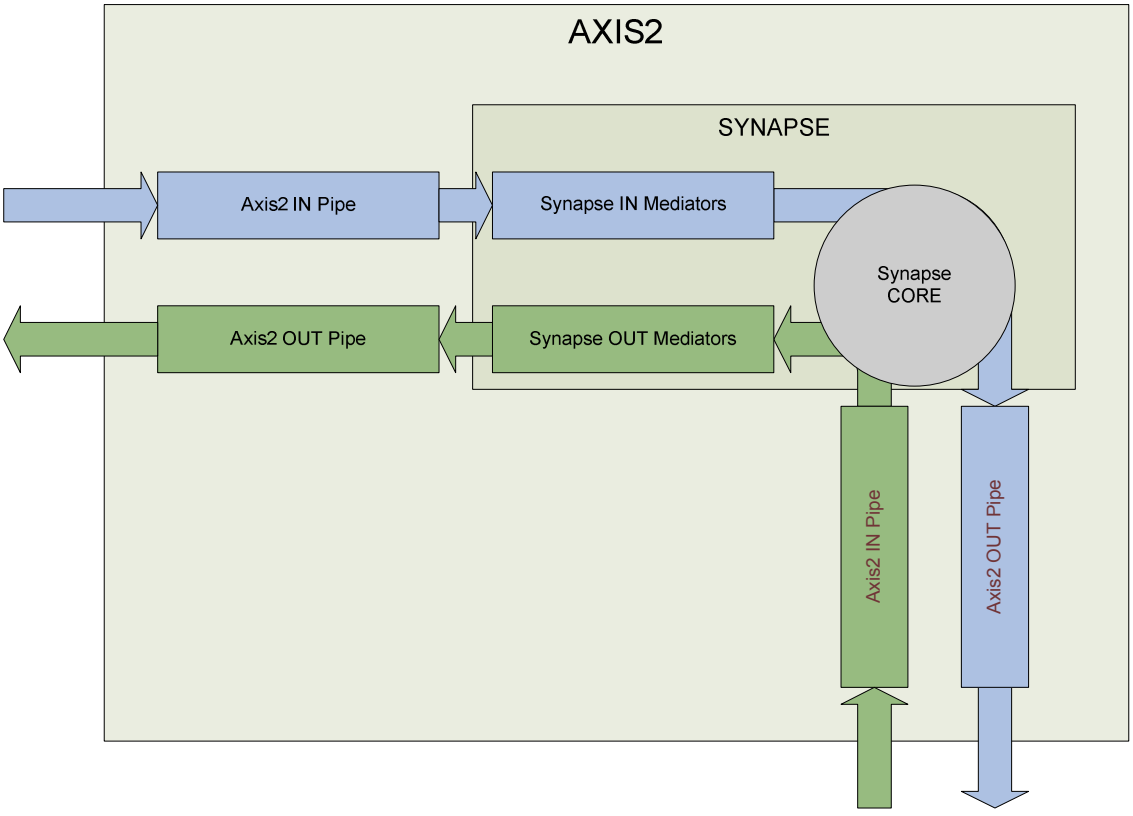
Synapse ha lasciato lo status di Incubator ed è stato promosso come sottoprogetto di Apache Web Service nel gennaio 2007.

La versione 1.0 è stata rilasciata a Giugno 2007 e le sue funzionalità chiave sono:

- Proxy services - facilitating transport, interface (WSDL/Schema/Policy), message format (SOAP/POX), QoS (WS-Security/RM) and optimization switching (MTOM/SwA)
- Trasporto /s non bloccante basato su Apache Core per l'esecuzione e il supporto di migliaia di connessioni ad altissima velocità.
- Registro/repository integrato per facilitare l'aggiornamento della configurazione e delle risorse associate.
- Facile estensione grazie a mediatori Java o linguaggi di scripting (Javascript, Ruby, etc..).
- Supporto a Load-balancing/Fail-over e Throttling.
- WS-Security, WS-Reliable Messaging e Throttling configurabili via WS-Policies.
- Supporto ai messaggi JMS per payload binari, plain text, XML e SOAP.
- Modello incentrato sui messaggi.
- Configurazioni serializzate nel file system per versioning, backup e ripristino.
- Supporto per gestione degli errori, timeouts e ripristino.

Synapse si basa su Axis2 ed a breve dovrebbe diventarne un modulo. Fino ad oggi ne usa le librerie funzionando come server stand alone. Synapse aggiunge il concetto di

mediatore. Un mediatore è una classe Java che esegue delle azioni sui messaggi intercettati. Alcuni mediatori sono già implementati, come il SendMediator, che si occupa di inoltrare i messaggi, o il LogMediator per fare logging dei messaggi, mentre è fornita la possibilità di crearne di propri implementando le appropriate classi. Per l'inoltro dei messaggi, utilizza un oggetto org.apache.axis2.client.OperationClient pensato originalmente per l'invio dei messaggi lato client. Questo ovvia al limite sofferto dai Service Framework che sono pensati per ricevere ed eventualmente rispondere al mittente e non possono ricoprire la funzione di intermediari in una comunicazione. Possiamo allora schematizzare l'architettura di Synapse in questo modo:



3.4 JMS

Un'architettura alternativa già implementata e testata è quella che prevede l'utilizzo di un broker di messaggistica JMS.

Java Message Service (JMS) [50] è l'insieme di API che consentono lo scambio di

messaggi tra applicazioni Java distribuite sulla rete. In ambito JMS un messaggio è un raggruppamento di dati che viene inviato da un sistema all'altro pensato più per la notifica di eventi che dovranno essere interpretati da programmi anziché direttamente da un utente umano. Vediamo gli elementi principali che compongono questo standard:

- *JMS provider*: Un'implementazione dell'interfaccia JMS per Message Oriented Middleware (MOM). Possono essere sia implementazioni Java che MOM non-java.
- *JMS client*: una applicazione o processo che produce o elabora messaggi.
- *JMS producer*: Un client JMS che produce e invia messaggi.
- *JMS consumer*: Un client JMS che riceve messaggi.
- *JMS message*: Un oggetto contenente i dati trasferiti tra un client JMS e l'altro.
- *JMS queue*: Una struttura dati contenente i dati inviati che devono ancora essere letti. Come suggerisce il nome, i messaggi sono rilasciati nell'ordine in cui sono stati inviati. Un messaggio viene rimosso dalla coda una volta letto.
- *JMS topic*: Un meccanismo di distribuzione di messaggi che devono essere recapitati a molteplici iscritti.

Il sistema di messaggistica che si può costruire appoggiandosi al JMS è puramente asincrono:

- Il mittente invia un messaggio ad un ricevente (o ad un gruppo di riceventi).
- Il destinatario, quando disponibile, riceve il messaggio ed esegue le operazioni corrispondenti.

In questo senso, il sistema di messaggistica così ottenuto differisce da un sistema RMI o RPC dove il client che invoca un metodo remoto deve rimanere in attesa della risposta per poter continuare. Il sistema si basa sul paradigma peer-to-peer quindi un client può inviare e ricevere messaggi verso qualsiasi altro client utilizzando un provider. Ciascun client si connette all'agente (gestore) di messaggistica che fornisce gli strumenti per creare, inviare e ricevere i messaggi. Questo tipo di comunicazione si può definire *loosely coupled* (debolmente accoppiata), infatti il mittente e il destinatario non devono necessariamente essere contemporaneamente disponibili per poter comunicare. Inoltre i membri della comunicazione possono non avere alcuna nozione l'uno dell'altro grazie all'intermediazione del messaging agent. L'unica nozione che li accomuna è l'agente stesso e il formato dei messaggi scambiati che tutti devono saper interpretare. Il sistema

JMS fornisce comunicazioni

- Asincrona: il JMS Provider consegna il messaggio al destinatario non appena questo si è reso disponibile, senza che questi ne faccia richiesta specifica
- Affidabile: JMS può assicurare che il messaggio sia consegnato una e una sola volta.

Ulteriori livelli di robustezza sono garantiti in JMS

- Tramite il cosiddetto *Guaranteed Message Deivery* è possibile prevenire la perdita d'informazioni in caso di malfunzionamento o di crash del message server rendendo i messaggi persistenti prima di essere recapitati ai consumatori (per esempio mediante JDBC).

JMS comunque non include:

- Un sistema di Load Balancing e Fault Tolerance: la API non specifica il modo in cui più client debbano cooperare alla fine di implementare un unico servizio critico.
- Notifica degli Errori
- Security: JMS non fornisce garanzie sulla privacy e l'integrità dei messaggi gestiti

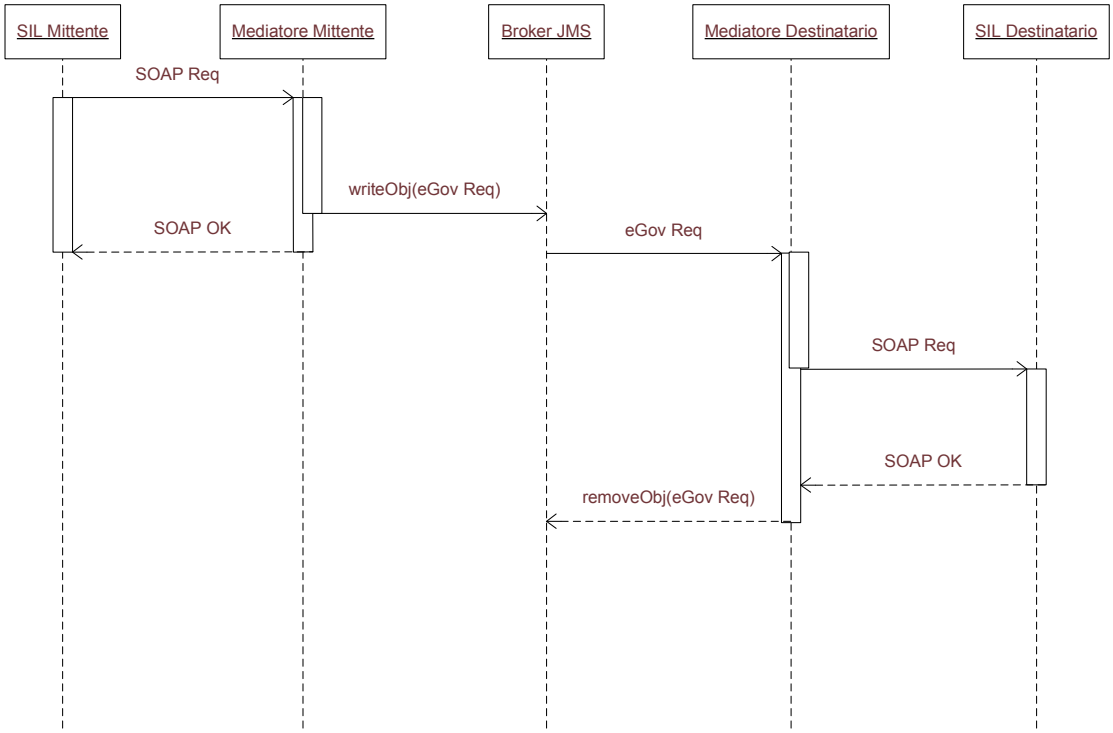
Come detto precedentemente il sistema JMS è intrinsecamente asincrono, ma viene fornita comunque la possibilità di effettuare una comunicazione Sincrona usando un MessageConsumer. Il consumer richiede esplicitamente alla destinazione di prelevare il messaggio (*fetch*) invocando il metodo *receive*(). Questo metodo è sospensivo, vale a dire rimane bloccato fino alla ricezione di un messaggio o di un timeout impostato in precedenza.

Oltre alla modalità sincrona e asincrona, JMS fornisce anche comunicazioni punto a punto (P2P) e punto a molti. La prima modalità prevede che il messaggio sia scritto su una coda di messaggi dalla quale può essere letto una ed una sola volta, quindi possiamo avere più di un lettore associato ad una lista, ma solo uno di questi può accedere al messaggio.

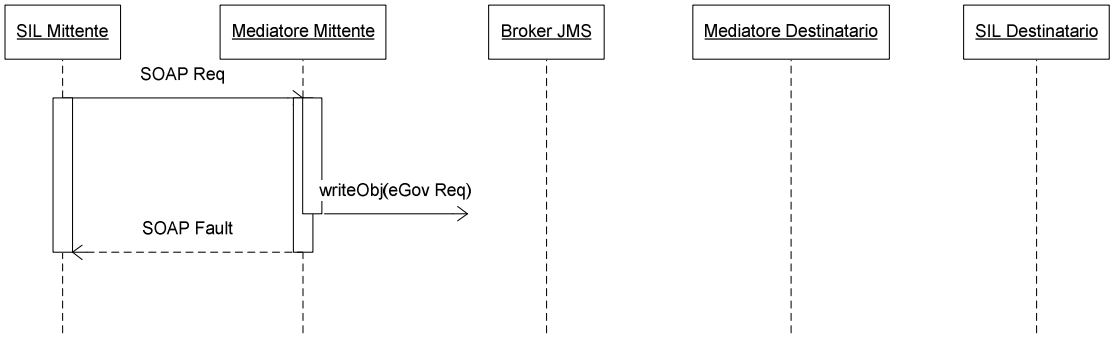
L'altra modalità prevede che più client si possano iscriversi ad un *topic*. Una volta che un messaggio è aggiunto al topic, tutti i client ad esso iscritti lo ricevono (*Publish and Subscribe*).

Per confrontare questa architettura con quella fornita da OpenSPCoop, esaminiamo la

comunicazione OneWay con un broker JMS.

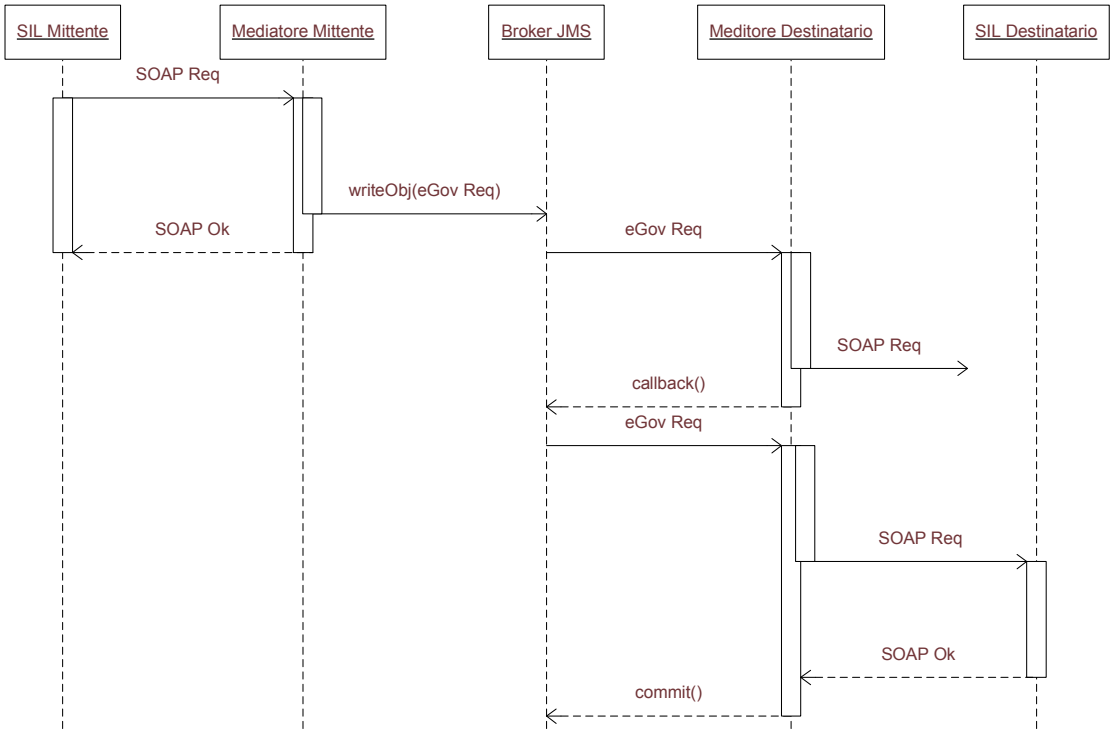


Come si può notare, non appena la Porta di Dominio del mittente è riuscita a scrivere il messaggio sulla coda JMS invia l'ack SOAP al client mittente che può procedere all'elaborazione, con garanzia che il messaggio verrà recapitato a destinatario, anche se ancora non l'ha ricevuto, grazie alla natura asincrona del broker JMS.
Anche in caso di errore viene garantita la consegna del messaggio. Se il messaggio non raggiunge il broker il fallimento viene comunicato al mittente.



Se l'errore si verifica dopo la presa in gestione del broker JMS, il client mittente procede come se la comunicazione si fosse conclusa con successo grazie alla presa in

consegna dell'infrastruttura e visto che il OneWay non prevede un messaggio di risposta. Il Broker e il Mediatore continueranno a tentare di concludere la comunicazione successivamente.



Abbiamo quindi un sistema

- Affidabile: grazie alla transazionalità del JMS possiamo garantire la consegna dei messaggi
- Veloce: grazie alla natura asincrona del Broker possiamo disaccoppiare il mittente dal destinatario, prendere in consegna il messaggio e chiudere la connessione col mittente subito dopo l'invio del messaggio.

In quest'architettura possiamo integrare il broker in due modi:

- Distribuito: Il broker è integrato nella Porta Di Dominio. Questa soluzione aumenta il carico del sistema che ospita la Porta di Dominio che deve gestire anche la coda JMS e lega la disponibilità del servizio ad una sola componente.
- Centralizzato: Il broker è stand alone e può gestire più code per più Porte di Dominio. Questa soluzione vincola la disponibilità del servizio alla disponibilità del broker. Inoltre alleggerisce il carico dei sistemi che ospitano le Porte di Dominio che non devono più preoccuparsi della persistenza dei messaggi. Il

problema di questa soluzione è che se si rende indisponibile il broker saranno non disponibili tutti i servizi che si appoggiano ad esso.

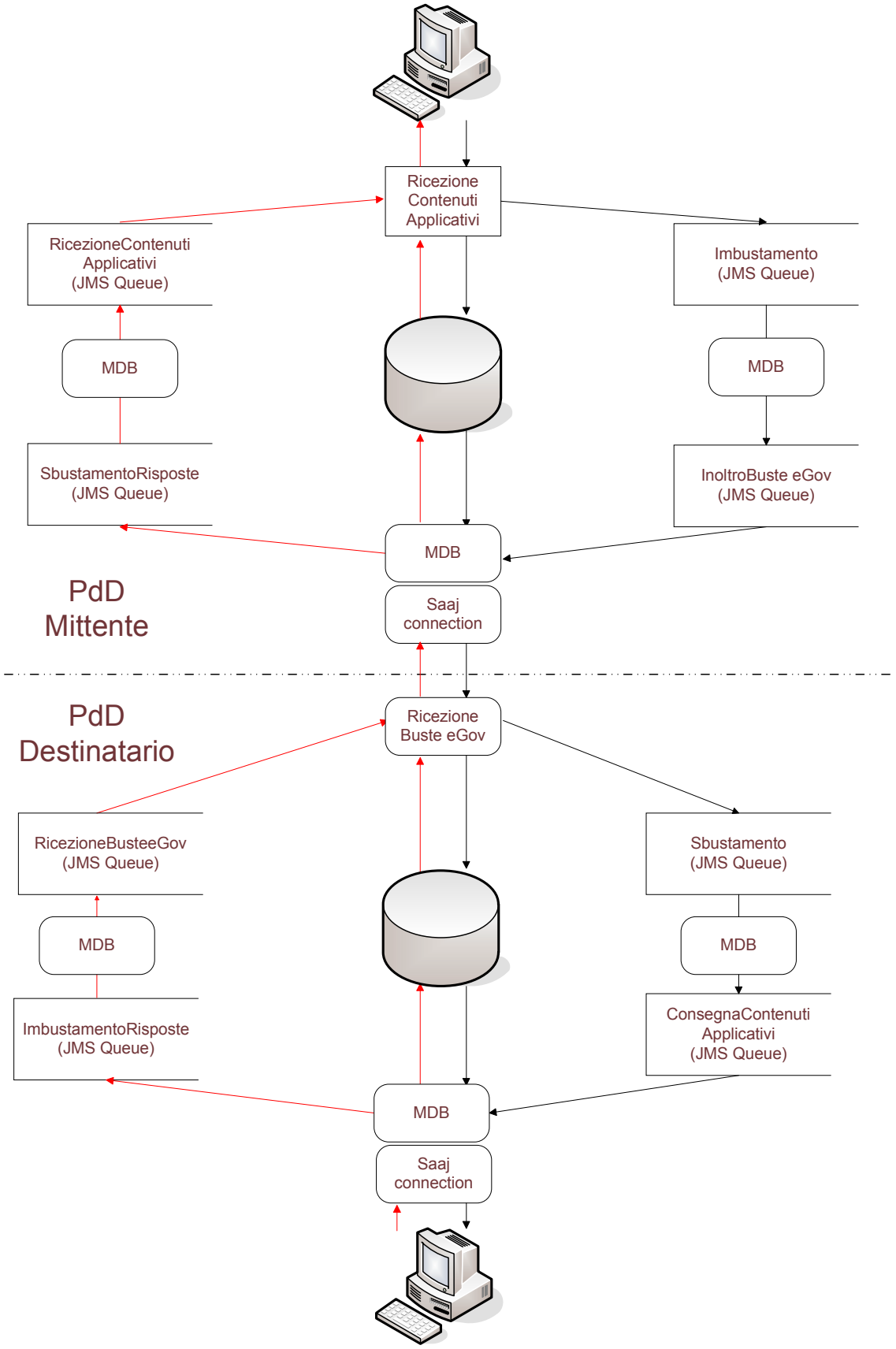
4. Analisi delle tecnologie studiate

In questo capitolo ci occuperemo di valutare come le tecnologie introdotte nel capitolo precedente si adattano a risolvere le problematiche di un'implementazione di SPCoop. Partiremo da un'analisi dell'architettura attualmente in uso in OpenSPCoop confrontandola con le architetture introdotte nel terzo capitolo. Durante questa analisi metteremo alla luce i comportamenti che hanno generato la necessità di una revisione dell'architettura corrente. A seguire sarà valutata l'applicazione delle tecnologie introdotte nei capitoli precedenti al progetto OpenSPCoop, analizzando i benefici che possono apportare e se questi costituiscono una soluzione alle problematiche presentate.

4.1 Architettura e tecnologie utilizzate in OpenSPCoop

L'architettura SOA, introdotta nel primo capitolo, si occupa della gestione dei servizi e di comunicazioni punto a punto. Su quest'architettura si basa quella ESB, che concerta l'uso di più entità (servizi) per il completamento di una comunicazione più complesa, prendendosi carico di fornire alcuni servizi evitando che siano gli utenti ad implementarli. A livello di processo, è stata presentata l'architettura SEDA, nata da quella EDA, che fornisce un modello di sviluppo della logica di un processo e che si propone di risolvere alcune problematiche in ambito Web Service. Quella che viene implementata in OpenSPCoop è una soluzione che sfrutta tutte queste architetture.
Abbiamo quindi a livello di servizio un'architettura ESB, con i servizi forniti dai Sistemi Informativi Locali che per comunicare gli uni con gli altri passano attraverso nodi intermedi, le Porte di Dominio. Inoltre implementa l'architettura EDA attraverso il Gestore degli Eventi, usando il modello *Publish and Subscribe* (P&S) attraverso scambio di Buste eGov con una Porta di Dominio di interfaccia al Gestore Eventi.
Infine il modulo di controllo della Porta di Dominio è implementato in modo da ricordare il modello SEDA, con il processo suddiviso in moduli *Event Driver Bean* (EDB) che implementano porzioni della logica ben definiti separati da code JMS. Questo approccio consente di svincolare il numero di richieste accettate dal numero di thread attivabili. Una gestione così modulare si presta ad operazioni di load balancing,

operando sui pool di thread in ascolto su ogni coda. L'attuale implementazione prevede che le code JMS siano persistenti, ma è in previsione un'implementazione che consenta di renderle volatili, pur mantenendo la robustezza del sistema.
I test effettuati hanno messo in luce un problema legato alle performance del modulo di controllo della Porta di Dominio in OpenSPCoop. Inviare il messaggio tramite l'infrastruttura di OpenSPCoop, rapportato ad un invio diretto, accumula un overhead eccessivo dovuto al processo di mediazione. I test hanno evidenziato che gran parte di questo overhead è imputabile agli accessi in database effettuati per salvare il messaggio e le informazioni sullo stato del processo di mediazione, utili a concertare le operazioni che lo implementano, senza considerare quello introdotto dalle code JMS persistenti che saranno sostituite da code volatili in future release del sistema. La figura seguente schematizza la struttura del modulo di controllo mostrando come la logica sia suddivisa in blocchi applicativi che si scambiano i dati da processare tramite code JMS.


Ricezione
Contenuti
Applicativi
RicezioneContenuti
Applicativi
(JMS Queue)
Imbustamento
(JMS Queue)
MDB
MDB
SbustamentoRisposte
(JMS Queue)
InoltroBuste eGov
(JMS Queue)
MDB
Saaj
connection
PdD
Mittente
PdD
Destinatario
Ricezione
Buste eGov
RicezioneBusteeGov
(JMS Queue)
Sbustamento
(JMS Queue)
MDB
MDB
ImbustamentoRisposte
(JMS Queue)
ConsegnaContenuti
Applicativi
(JMS Queue)
MDB
Saaj
connection

I messaggi, appena arrivati, sono salvati su database, mentre i dati necessari per svolgere le operazioni di contorno del processo di mediazione, attraversano i vari stadi (*stages*) procedendo attraverso le code JMS. Al termine, per essere inviato alla Porta Applicativa destinataria, il messaggio originale viene letto dal database, imbustato in una opportuna Busta eGov e inoltrato al destinatario. Speculare il processo lato destinatario per la richiesta e per l'eventuale risposta. Sono appunto gli accessi a database e, marginalmente, l'overhead introdotto dal partizionamento del processo in *stages* la causa delle prestazioni poco soddisfacenti di OpenSPCoop.
Un altro aspetto da considerare sono le tecnologie utilizzate per l'implementazione di OpenSPCoop 1.0. Il Web Service Framework scelto è Apache Axis. Come mostrato nel terzo capitolo, le specifiche in ambito Web Service si sono evolute, in alcuni casi sono diventate obsolete, in altri ne sono state introdotte di nuove. Il team di sviluppo di Axis ha colto l'occasione per effettuare una profonda modifica all'architettura del loro Web Service Framework, chiamandolo Axis2, cessando contestualmente il supporto del predecessore. Questo significa che alcune delle specifiche più recenti e quelle che verranno non saranno supportate, quindi, prevedendo un adeguamento delle specifiche SPCoop, è necessario sostituire gli strumenti utilizzati con i nuovi proposti dal panorama Web Service.
Procediamo quindi con l'analisi degli strumenti introdotti nel capitolo precedente nel contesto dell'implementazione delle specifiche SPCoop e quali benefici può trarne l'attuale implementazione OpenSPCoop.

4.2 Web Service Framework

Come anticipato, Axis, l'attuale Web Service Framework, non è più mantenuto. Dopo anni di sviluppo, il team che lo curava ha deciso di cogliere il momento di grande cambiamento in ambito Web Service per rivoluzionare l'architettura, applicando migliorie che gli anni di esperienza hanno permesso di ideare, dando vita a Axis2. L'abbandono di Axis ha reso necessaria la sua sostituzione, sia perché si basa su un'architettura datata, quindi potenzialmente meno efficiente di concorrenti più "giovani", sia perché, in previsione di future revisioni delle specifiche SPC, non sarà in grado di fornire supporto a nuove specifiche. Abbiamo introdotto due nuovi Service

Framework nel capitolo precedente, vediamo come si adattano al progetto OpenSPCoop.

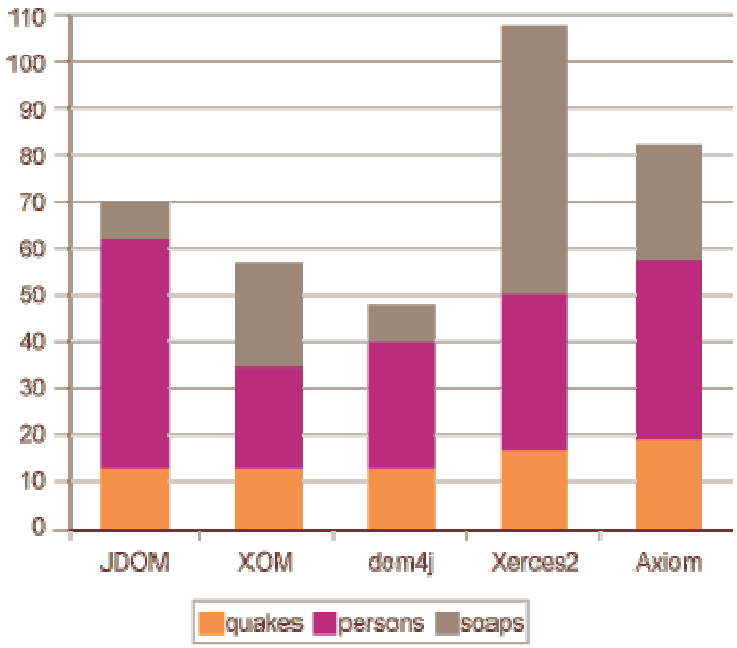
4.2.1 Axis2

Dal momento che la corrente versione di OpenSPCoop utilizza Axis, è parso naturale, visto il suo "pensionamento", orientarsi verso il suo successore Axis2. Vediamo come si adatta ad un'eventuale integrazione in OpenSPCoop.

Partiamo dall'object model AXIOM ed a valutarne la performance rispetto ad altri object models basati su StAX. I test sono stati effettuati utilizzando tre set di documenti XML:

- Quakes: (18k) Elementi ripetuti con numerosi attributi.
- Persons: (202k) Struttura singola ripetuta con poche variazioni nei valori contenuti.
- SOAP: (19k) un set di 30 piccoli messaggi SOAP

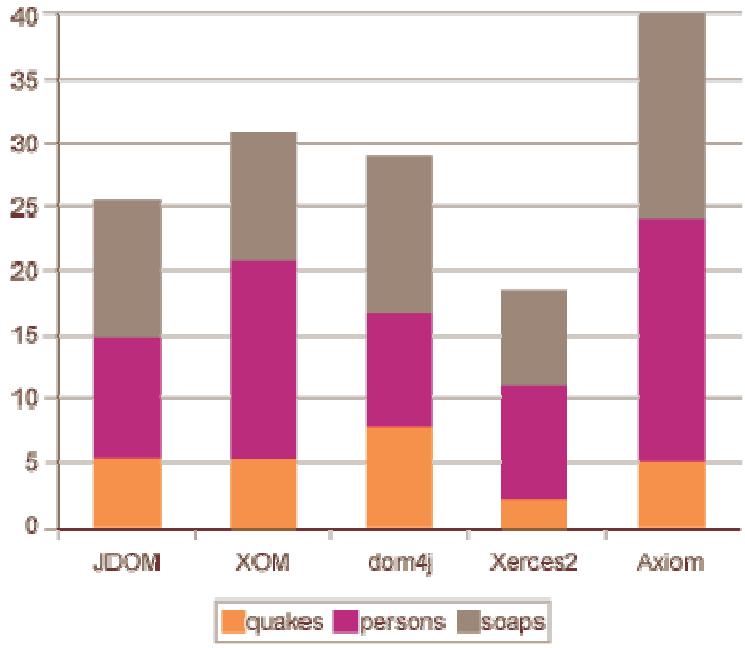
I primo test consiste nella costruzione e nella lettura del contenuto dei documenti presi in esame:



Axiom risulta migliore solo di Xerces2 ed è evidente la difficoltà di alcuni modelli nel gestire i piccoli messaggi SOAP, soprattutto per Xerces2, ma anche Axiom sembra soffrire di questa difficoltà: basta paragonarlo a dom4j che impiega circa un terzo del

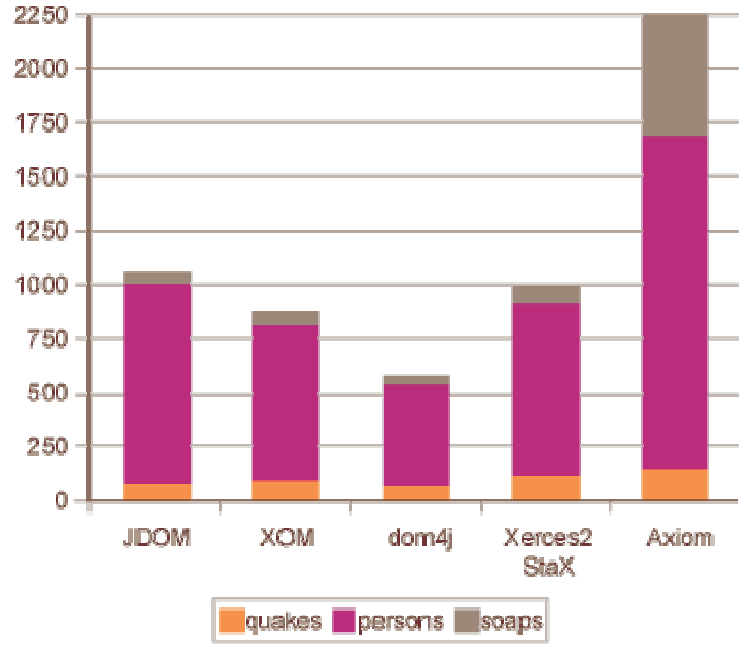
tempo impiegato da Axiom in quella porzione di test.

Nel secondo test sono valutati i tempi di scrittura dei documenti:



In questo caso Xerces2 è il migliore con un buon margine sugli altri, mentre Axiom risulta di gran lunga il peggiore e rimarca la sua difficoltà nel gestire i piccoli documenti.

Nel terzo test è valutata l'occupazione di memoria:

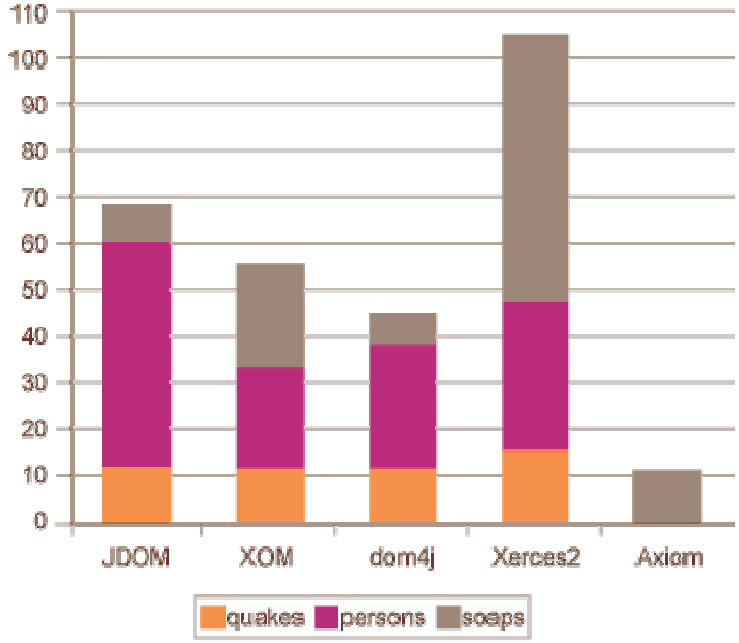


Dom4j è indiscutibilmente il migliore, come Axiom il peggiore. La sua elevata occupazione di memoria è dovuta a due fattori: primo, il parser deve essere referenziato

dal documento costruito, quindi una sua istanza deve essere in memoria finché l'istanza del documento è in uso; secondo, i singoli oggetti del modello sono più "pesanti" in termini di memoria delle corrispondenti controparti negli altri modelli.

Se valutiamo Axiom sulla base di questi tests, risulta il peggiore sotto ogni aspetto, ma dobbiamo ricordare che per l'architettura sottostante, i documenti elaborati costituiscono il caso pessimo, ovvero quando tutto il documento deve essere costruito, mentre i benefici di questo modello vengono alla luce qualora non sia necessaria l'intera costruzione del documento.

Vediamo allora cosa accade se prendiamo in considerazione solo la prima parte dei documenti in esame senza lavorare su tutto il messaggio:



Come da previsioni, in questo scenario Axiom risulta il vincitore indiscusso, soprattutto per quanto riguarda per i documenti di grandi dimensioni, dove è maggiore il lavoro risparmiato da una valutazione parziale del contenuto.

In conclusione questi tests mostrano che Axiom è da prendere in considerazione solo nel caso in cui i messaggi da processare non richiedano la costruzione completa dell'object-tree in memoria, altrimenti sia i tempi di processamento sia l'occupazione di memoria aumentano vertiginosamente rispetto ad altri object model.

Vediamo allora come OpenSPCoop tratta i messaggi che deve mediare e come questo influenza le performances di AXIOM. Logicamente, la parte di mediazione di OpenSPCoop interessante per questa valutazione può essere schematizzata in questo

modo:

- Ricezione del messaggio sulla porta delegata.
- Salvataggio del messaggio.
- Imbustamento.
- Tracciamento.
- Applicazione servizi aggiuntivi.
- Inoltro verso la porta applicativa.

Questo schema considera solo la parte di mediazione lato mittente, quella lato destinatario è sostanzialmente speculare.

La ricezione del messaggio sulla porta delegata non comporta nessun problema. Gli handlers in ingresso lavorano solo sull'header del messaggio e non s'interessano del payload che non deve essere costruito. Quindi siamo in un caso dove Axiom si comporta bene, la costruzione solo dell'header del messaggio.

Per il salvataggio ci possono essere dei problemi, infatti se non viene effettuato il caching dei dati questi non saranno più disponibili per un successivo utilizzo, quindi è importante eseguire il salvataggio sincerandosi di non aver più bisogno del messaggio.

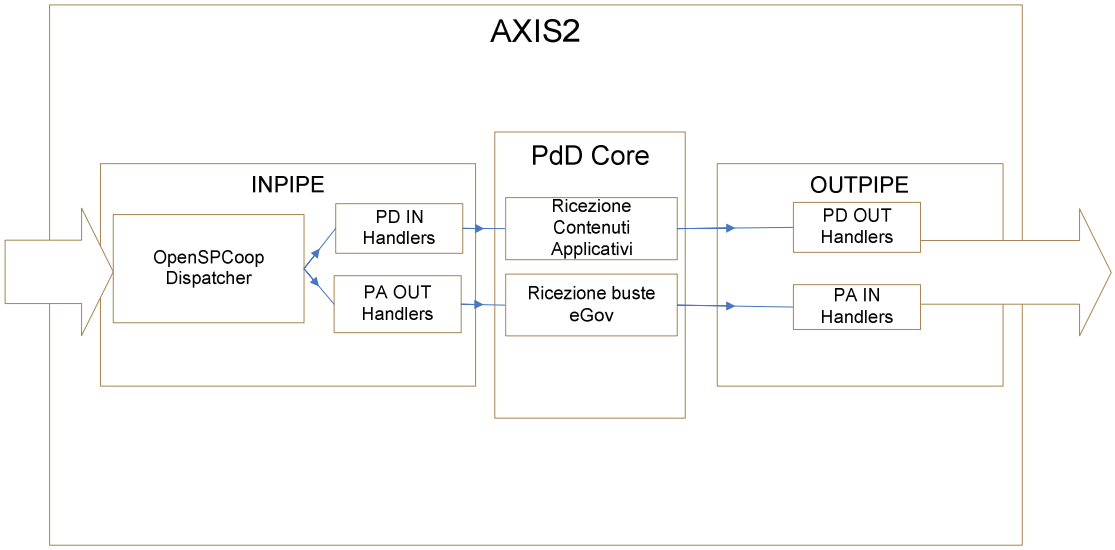
Il messaggio poi sarà riletto e l'header modificato secondo le specifiche della Busta eGov. Qui vanno distinti due casi: il messaggio SOAP contiene o meno allegati. Se non ha allegati le modifiche apportate nel modulo d'imbustamento coinvolgeranno solo l'header, mentre nel caso siano presenti allegati la modifica coinvolgerà anche il body causandone la costruzione da parte dell'Object Model.

Nella fase successiva, il tracciamento, verrà loggato l'header eGov, quindi rimaniamo ancora con una costruzione parziale del modello.

Il passaggio critico è quello dell'applicazione di servizi aggiuntivi, in particolare delle politiche di sicurezza. Se il corpo del messaggio deve essere cifrato, necessariamente sarà effettuata la completa costruzione dell'Object Model del messaggio SOAP. Questo avviene solo se configurato, ma si può assumere che avverrà per la maggior parte delle Porte di Dominio.

Questa situazione costituisce per Axiom un caso pessimo per ogni messaggio. Consideriamo comunque Axis2 nella sua interezza, per valutare se il suo utilizzo in un'implementazione di SPCoop rimane comunque una possibilità da prendere in considerazione.

Inizialmente è stato compiuto un porting di OpenSPCoop usando le soluzioni standard di Axis2, quindi usando l'implementazione LLOM di AXIOM, inserendo gli handlers in un modulo e inserendo un dispatcher personalizzato per indirizzare i messaggi ai giusti servizi. Il grafico seguente mostra solo alcuni componenti, oltre a quelli standard,



Il porting è stato molto difficoltoso, non solo per doversi adattare alla diversa architettura del Web Service framework, ma anche per conformarsi alle nuove specifiche. Utilizzare le API AXIOM anche per la manipolazione del messaggio al posto di SAAJ ha comportato la riscrittura quasi completa del codice, anche se la logica rimaneva la stessa. Non implementando il solito standard, c'era la possibilità di produrre risultati diversi anche utilizzando metodi simili delle due API. Questo imponeva la necessità di eseguire numerosi e approfonditi test al termine di ogni fase. La portabilità di un simile sistema sarebbe compromessa visto che sia le API usate per manipolare i messaggi sia il formato dei dati salvati non si riferiscono a nessuno standard.

Il 13 Agosto 2007 Axis2 è giunto alla versione 1.3 e solo in questa versione è stato aggiunto il supporto a JAX-WS. Si è quindi concretizzata la possibilità di effettuare il porting in standard JAX-WS riutilizzando gran parte del vecchio codice, scritto in standard SAAJ, anche se le versione di SAAJ implementate sono differenti, e contestualmente garantendo portabilità.

Il supporto a JAX-WS si è però dimostrato molto acerbo e afflitto da numerosi bugs. In particolare è stato impossibile definire l'handler chain tramite annotazioni, come prevede lo standard JAX-WS, mentre, cercando di aggirare il problema utilizzando il

metodo standard di Axis2 aggiungendo un componente modulo, si verifica una perdita di informazioni quando il messaggio incapsulato nell'oggetto MessageContext di Axis2 viene convertito in un oggetto SOAPMessage di SAAJ, vanificando il lavoro degli handlers. Inoltre, i moduli di Axis2 che si occupano delle funzionalità aggiuntive, come Rampart2, Sandesha2, Kandula2 lavorano sul messaggio partendo dal MessageContext, quindi occorre passare comunque da uno standard all'altro se non si vogliono sviluppare soluzioni *ad hoc*, sempre pagando poi il costo aggiuntivo per mantenere il codice aggiornato.

E' comunque stato completato un porting funzionante per alcuni degli scenari possibili di utilizzo di OpenSPCoop che utilizza in maniera ibrida le tecniche fin qui descritte. Nel dettaglio usa le API AXIOM ed il formato standard del MessageContext per gli handler e l'inizializzazione, poi passa all'implementazione di SAAJ per la parte di manipolazione del messaggio e il salvataggio ed eventualmente di nuovo al MessageContext per essere passato a Rampart2. Lo sviluppo del prototipo è stato poi sospeso in attesa che le API di Axis2 maturino e permettano una gestione più elegante dei messaggi senza questi "balzelli" che ne compromettono le performance.

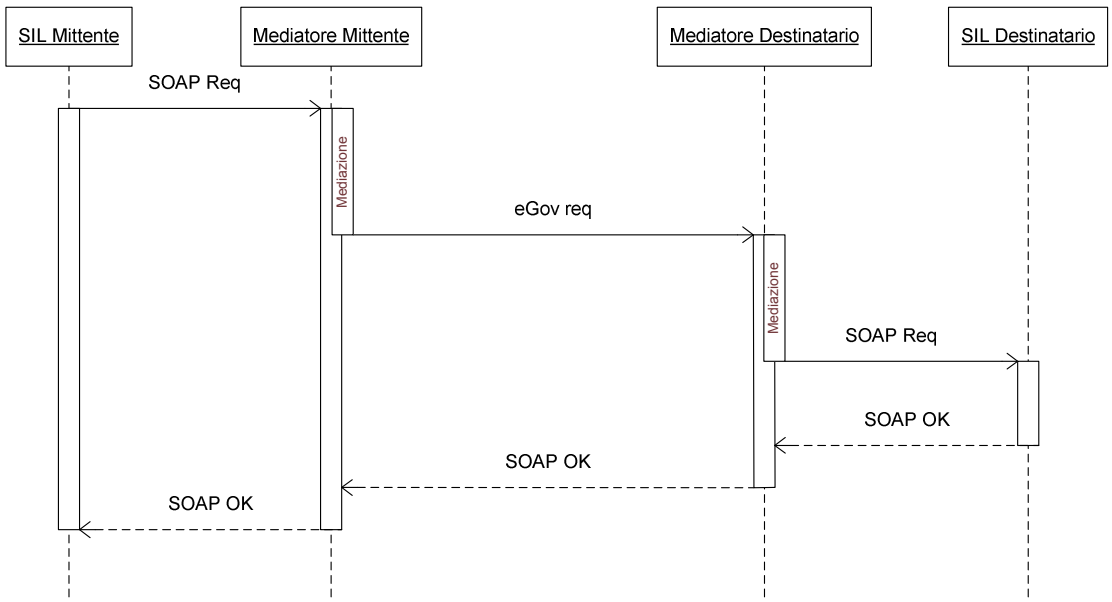
4.2.2 CXF

Abbiamo effettuato alcuni test funzionali con CFX per controllarne le effettive capacità. L'implementazione di JAX-WS è di buon livello e siamo riusciti a configurare i servizi web Porta Delegata e Porta Applicativa con i relativi Interceptors grazie alle annotazioni introdotte con JAX-WS. Sempre grazie a JAX-WS è stato possibile implementare il servizio a livello di messaggio ottenendolo in forma di SOAPMessage di SAAJ, senza dover ricorrere a laboriose conversioni di tipo. Nei test effettuati abbiamo riscontrato dei problemi con l'uso degli attachment, che non venivano correttamente incapsulati dentro l'oggetto SOAPMessage. Collaborando con gli sviluppatori di IONA abbiamo risolto il problema ed i test sono proseguiti con successo. CXF è stato sviluppato seguendo l'approccio "*use standards everywhere when possible*". Al contrario di Axis2 che implementa gli standard con api proprie, CXF usa standard anche per il modello di programmazione, implementando JAX-WS, e come meccanismo di configurazione, scegliendo Spring. Questo non implica certo che l'approccio scelto dagli sviluppatori di

CFX risulti il migliore, ma in un campo in fermento come quello dei Web Service Framework dove non si è ancora delineata una soluzione preferenziale alle altre, l'uso di implementazioni standard garantisce un buon livello di portabilità da uno strumento all'altro.

4.3 Synapse

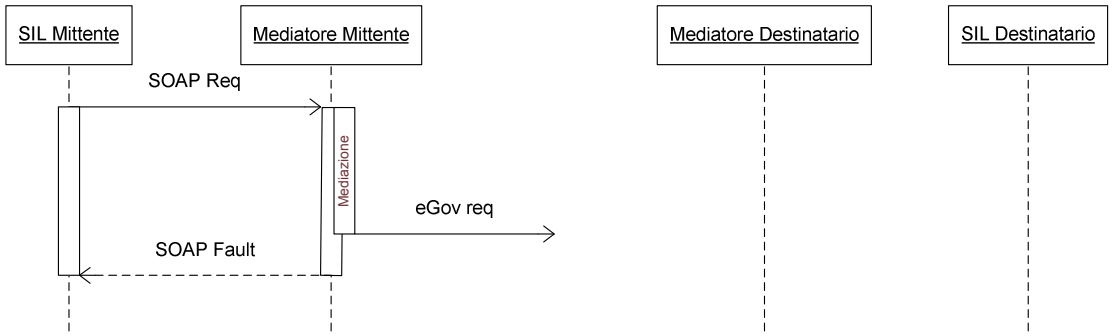
Abbiamo mostrato come Apache Synapse, basato su Axis2, si proponga come valido strumento per l'implementazione di un'architettura ESB. La sua architettura si presenta quindi modulare, ma piuttosto rigida. La principale mancanza è che non può prendersi carico della consegna dei messaggi. Per spiegare questo concetto prendiamo in esame il più semplice dei paradigmi di comunicazione, il OneWay su protocollo . In modalità OneWay il client invia un messaggio al servizio senza aspettarsi una risposta.



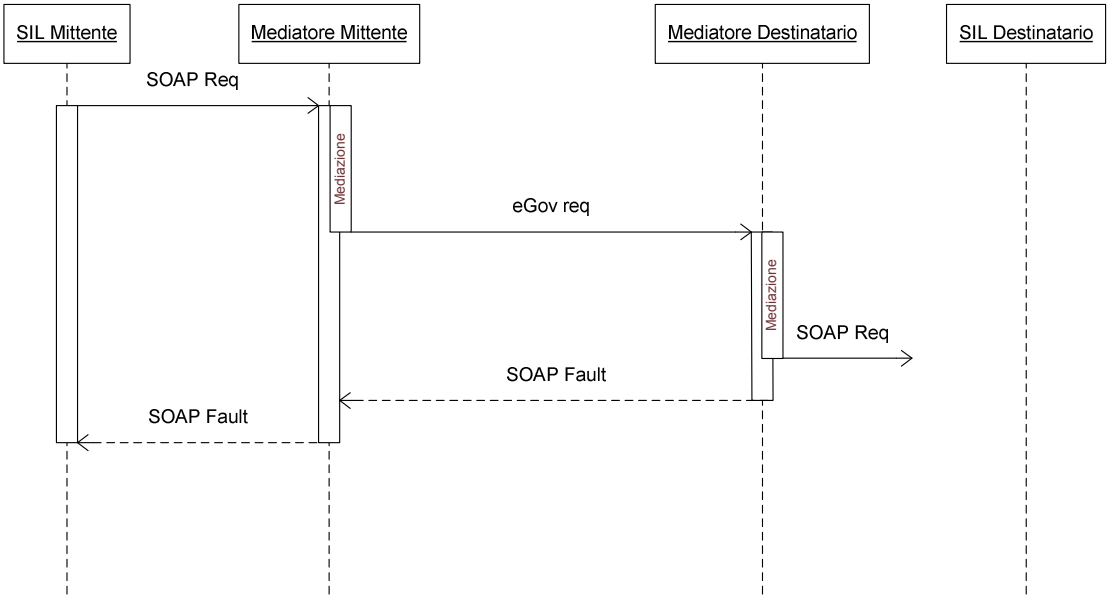
Come mostrato in questo diagramma di sequenza, il mittente invia al mediatore il messaggio e si mette in attesa dell'ack SOAP. Il mediatore applica le proprie politiche di mediazione e costruisce così la busta eGov secondo le specifiche SPCoop e le inoltra al mediatore di destinazione, sempre con una comunicazione OneWay, aspettando un riscontro. Il Mediatore di destinazione procede alla ricostruzione del messaggio orginale ed alle operazioni di corredo. Infine, consegna il messaggio al destinatario che, in caso

di successo, riscontra il messaggio con un ack SOAP che verrà propagato indietro, al mediatore del mittente ed al mittente, chiudendo così le connessioni aperte e rispettando il paradigma OneWay.

Cosa accade se qualcosa dovesse fallire durante la trasmissione del messaggio? Supponiamo ad esempio che uno dei due mediatori coinvolti nella comunicazione non sia raggiungibile:



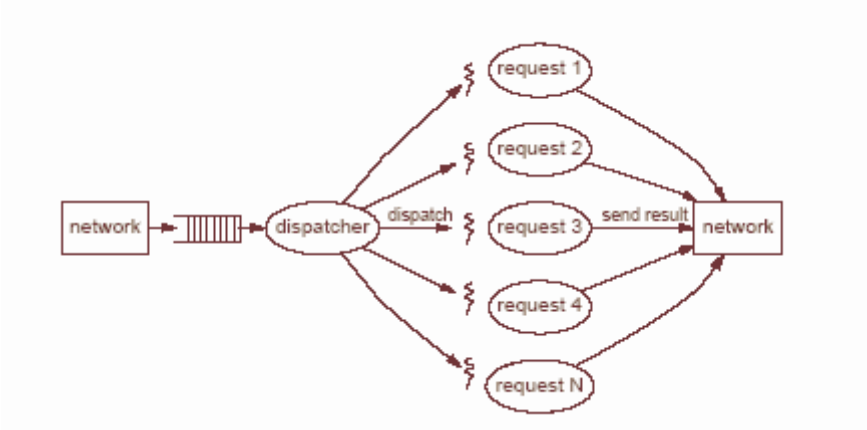
Il mittente riceve dunque l'errore, nonostante parte della comunicazione fosse stata completata con successo. Inoltre sarà compito del mittente la gestione dell'errore, cosa che le specifiche SPCoop non prevede. Ancora, l'elaborazione fin qui completata sarà ripetuta al prossimo tentativo d'invio da parte del mittente, quindi abbiamo un notevole spreco di risorse. Lo stesso scenario si ripropone qualunque sia il punto che genera l'errore.



Immaginiamo quindi uno scenario dove i mediatori coinvolti non siano soltanto due come nel nostro esempio, ma una comunicazione complessa con più punti intermedi. E' facile immaginare quante risorse siano sprecate in una comunicazione che fallisce e deve ripartire da capo, vanificando il lavoro fin li svolto. Uno degli obiettivi dell'infrastruttura SPCoop è appunto evitare questi scenari prendendosi carico della consegna del messaggio. Per far questo sarebbe necessario una profonda modifica ai mediatori forniti a corredo di Synapse per rispettare il nuovo protocollo di comunicazione. Inoltre ci sarebbe bisogno di aggiungere strutture dati e funzionalità per rispettare i requisiti di robustezza che queste nuove funzionalità richiedono.

C'è da notare un altro aspetto dell'architettura, derivato da Axis2, che male si sposa con l'archiettuta SPCoop. Come mostrato precedentemente, una volta mediati i messaggi, questi vengono inoltrati al destinatario attraverso le *pipes* di Axis2 per essere gestiti dai moduli agganciati ad esse. Quello che ci interessa notare è che una volta spediti attraverso le *pipes*, il controllo sul messaggio non è più in mano del programmatore. O tutto va bene o tutto va male, e la gestione dell'errore diventa più complessa. Nell'architettura che cerchiamo di sviluppare si vuole un controllo completo sul messaggio in ogni sua fase, spostando le fasi gestite nelle *pipes* (Security, Addressing etc..) nella parte della logica di mediazione, stravolgendo di nuovo l'architettura di Synapse.

Inoltre questa architettura segue un approccio thread-based, dove ad ogni richiesta viene associato un thread:



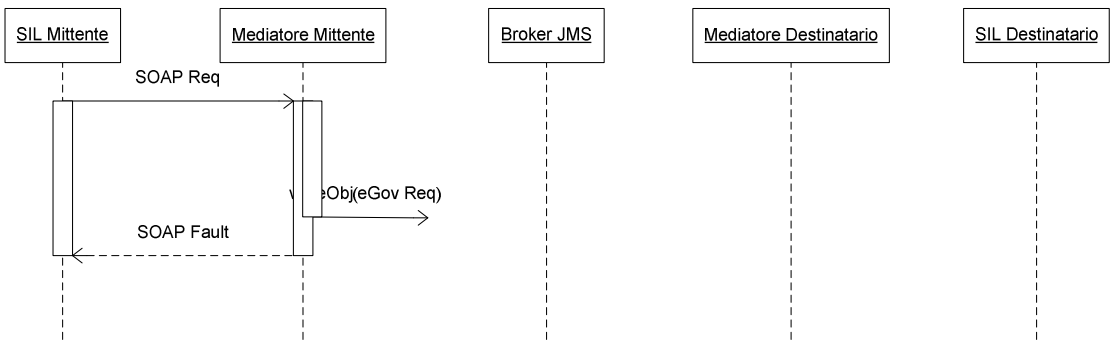
Questo approccio vincola il numero di richieste gestibili al numero di thread che possono essere attivati, anche se Synapse implementa il New I/O API (NIO) potendo quindi gestire un elevato numero di connessioni.

Per ultimo, ma non meno importante, poiché Synapse si basa su Axis2 e ne eredita le scelte implementative, esso ne usa il nuovo Object Model, AXIOM, che, come abbiamo visto, è inefficiente per implementare le specifiche SPCoop.

4.4 JMS

L'analisi di un'architettura basata su un broker JMS ha evidenziato che riesce ad effettuare quella "presa in carico" del messaggio che un sistema come Synapse non è progettato a fare. Inoltre implementa l'architettura EDA in maniera nativa. Ancora, centralizza totalmente lo storing e alleggerisce notevolemente il carico dei mediatori.

Ci sono però scenari di parziale fallimento che non riesce a gestire.



Come evidenziato dal diagramma di sequenza, se il Broker non è disponibile il sistema non effettua la presa in consegna. OpenSPCoop deve prevedere anche queste situazioni e prendersi carico del messaggio.
Il punto cruciale comunque è che viola il requisito delle specifiche SPC che ogni comunicazione deve essere basata su . Il JMS è uno standard JAVA e non garantisce interoperabilità con altri sistemi non Java. Fornire un'interfaccia che aggiri il problema introdurrebbe un'ulteriore overhead. E' comunque una buona fonte di riflessione su come gestire lo scambio di messaggi soprattutto per quel che concerne l'affidabilità e la velocità.

4.5 Conclusioni

Dagli studi fin qui compiuti possiamo arrivare alle seguenti conclusioni:
Synapse è un valido strumento di mediazione, ma la sua architettura non fornisce supporto alla completa gestione dello scambio di messaggi. Basandosi su Axis2 fornisce supporto ai più recenti standard in ambito Web Service, ma ne eredita l'inefficenza di AXIOM nel processare i messaggi nella loro interezza, come avviene in SPCoop. Questo non ne preclude l'utilizzo, ma anche dal punto di vista architetturale si è mostrato non adeguato per affrontare determinati aspetti della Cooperazione Applicativa. L'analisi approfondita del codice necessaria alla sua valutazione ha fornito in ogni caso validi spunti per la soluzione di alcuni punti critici di OpenSPCoop.
L'architettura basata su Broker JMS, come mostrato, non rispetta gli standard SPCoop, quindi non può esser presa in considerazione per essere utilizzata in una sua implementazione. Nonostante questo l'analisi di questo modello fornisce interessanti spunti per la gestione delle comunicazioni tra i partecipanti nello scambio di messaggi e costituisce una base per progettare una nuova architettura votata a migliorare le performances.
Per quanto riguarda la sostituzione di Axis con un Web Service Framework di nuova generazione, abbiamo valutato due candidati, Axis2 e CXF.
Axis2 si è dimostrato un valido strumento supportato da una vasta comunità che partecipa attivamente allo sviluppo. Per gli scenari classici di utilizzo in ambito SOA è certamente una soluzione preferenziale. Dovendo implementare la specifica SPCoop

con un occhio al supporto di altre specifiche abbiamo valutato aspetti dove Axis2 è più carente, ovvero qualità dell'implementazione di JAX-WS e performances nella manipolazione dei messaggi. Come mostrato, nell'implementazione di SPCoop, AXIOM non risulta essere una scelta valida e l'implementazione di JAX-WS è stata introdotta solo da qualche mese, con alcune funzioni ancora da implementare e altre contenenti bugs.

CXF, progetto nato dalla fusione di XFire e Celtic, gode anch'esso di una comunità vasta e attiva. La sua architettura è più semplice e snella di quella Axis2 e, per l'utente, alcune funzionalità sono più complesse da utilizzare rispetto alla sua controparte, ma per quello che concerne il nostro utilizzo si propone come scelta migliore. Si basa principalmente su standard, come richiesto da OpenSPCoop, e la loro implementazione è ad un buon livello di maturità, nonostante ci siano ancora dei problemi da risolvere.

Quindi sceglieremo CXF come componente per la gestione dei Web Service, mentre, per l'architettura ESB prevista da SPCoop, valuteremo una modifica a quella implementata in OpenSPCoop, prendendo spunto da quelle analizzate, ma senza possibilità di adottarne completamente alcuna.

5. Una nuova architettura per OpenSPCoop

Nel capitolo precedente abbiamo studiato quello che il panorama SOA ed ESB propone a soluzione dell'implementazione di SPCoop. A conclusione di questo studio abbiamo appurato che le architetture di tali soluzioni non sono ottimali per implementare le specifiche SPCoop. Rimane quindi preferenziale continuare lo sviluppo dell'architettura di OpenSPCoop maturata fino alla versione 1.0. Lo studio condotto ha comunque fornito interessanti punti di riflessione per poter migliorare la gestione dei messaggi modificando l'architettura di OpenSPCoop avvicinandola a sistemi che fanno delle porfomances il loro punto di forza.

In questo capitolo, per primo analizziamo il problema, esplicitando i motivi che ci hanno spinto a proporre una nuova architettura per l'implementazione di SPCoop. Vedremo quali vincoli sono stati imposti per questa proposta di modifica e infine un'accurata descrizione della nuova architettura, dove verranno valutati i vari comportamenti nei diversi scenari di comunicazione, i punti in cui si verificano dei miglioramenti rispetto alla precedente implementazione e dove paghiamo questi miglioramenti.

5.1 Analisi del problema

Come abbiamo spiegato nel capitolo precedente, durante lo sviluppo di OpenSPCoop sono emerse problematiche relative alla performance del sistema. Il tempo di gestione dei messaggi passando per il sistema OpenSPCoop è troppo più elevato rispetto ad una comunicazione diretta. Pur ottimizzando al meglio il codice, il divario è tale che il problema va ricercato altrove. La ricerca dei punti in cui la gestione dei messaggi di OpenSPCoop rallenta, ha mostrato che un notevole collo di bottiglia è costituito dall'accesso in database dovuto alla scrittura e lettura dei messaggi e, in maniera più marginale, all'uso delle code JMS nel modulo di controllo della Porta di Dominio OpenSPCoop. I diagrammi di sequenza e lo schema architetturale del modulo di controllo mostrano che un messaggio, di richiesta o risposta, viene immediatamente salvato su DB, poi i dati rilevanti per il processo di mediazione passano attraverso gli

stadi di processamento tramite code JMS ed infine il messaggio viene riletto da database, modificato e inoltrato al destinatario. Questo garantisce una presa in carico del messaggio immediata da parte di chi lo riceve e, da parte del mittente, conferme di ricezione in tempi ridotti. Nel complesso, però, l'intero processo di mediazione risulta pesante ed il servizio di presa in consegna fornito grava pesantemente in questo senso. Per snellire il processo di mediazione dobbiamo agire appunto su questi due aspetti di OpenSPCoop: accesso a DB e overhead introdotto dal passaggio di processo in processo. Questo può essere effettuato solo modificando l'architettura. Vediamo come.

5.2 Requisiti della nuova architettura da realizzare

Prima di procedere con la presentazione della modifica proposta all'architettura di OpenSPCoop, elenco brevemente i vincoli imposti che la nuova architettura deve rispettare:

- Rispettare le specifiche SPCoop.
- Mantenere i servizi di presa in carico garantiti dall'architettura originale.
- Supportare i nuovi standard.
- Migliorare le performances delle operazioni di mediazione.
- Riuso del codice esistente per ridurre al minimo il processo di implementazione.

5.3 Descrizione dell'architettura proposta

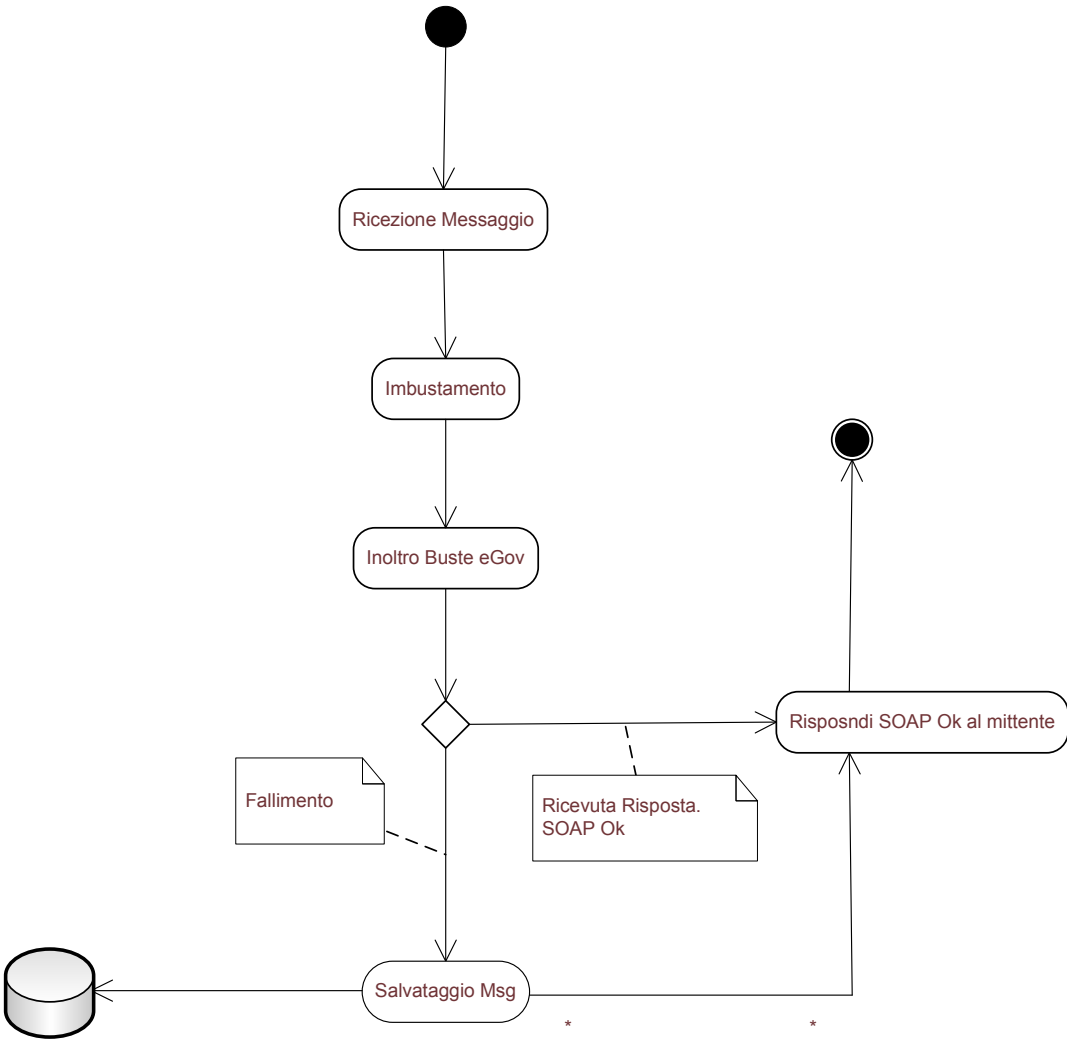
L'idea alla base dell'architettura proposta è quella di limitare il salvataggio dei messaggi solo se si rivela necessario, cercando di effettuare una mediazione "al volo", senza ricorrere all'architettura SEDA. Affiancare quindi alla tradizionale architettura di mediazione di OpenSPCoop un canale più performante e preferenziale.
Ricordiamo come il componente di controllo della Porta di Dominio OpenSPCoop effettua la mediazione dei messaggi:

- Il messaggio viene ricevuto e salvato su DataBase.
- I dati rilevanti passano attraverso code JMS nei vari stadi di processamento.
- Il messaggio da inviare viene costruito partendo dall'originale nel Database e i dati risultanti dalle operazioni di modulo di controllo e inviato al destinatario.

In pratica viene fatto uno "*Store and Try*" dei messaggi. Quello che si vuole implementare è sostanzialmente un *"Try and, maybe, Store"* continuando a fornire gli stessi servizi qualitativi dell'architettura originale.
Dal punto di vista architetturale abbiamo quindi due modelli di gestione affiancati che costituiscono due canali di mediazione, il *tradizionale* ed il *preferenziale*.
Vediamo come questa architettura gestisce i vari paradigmi di comunicazione.

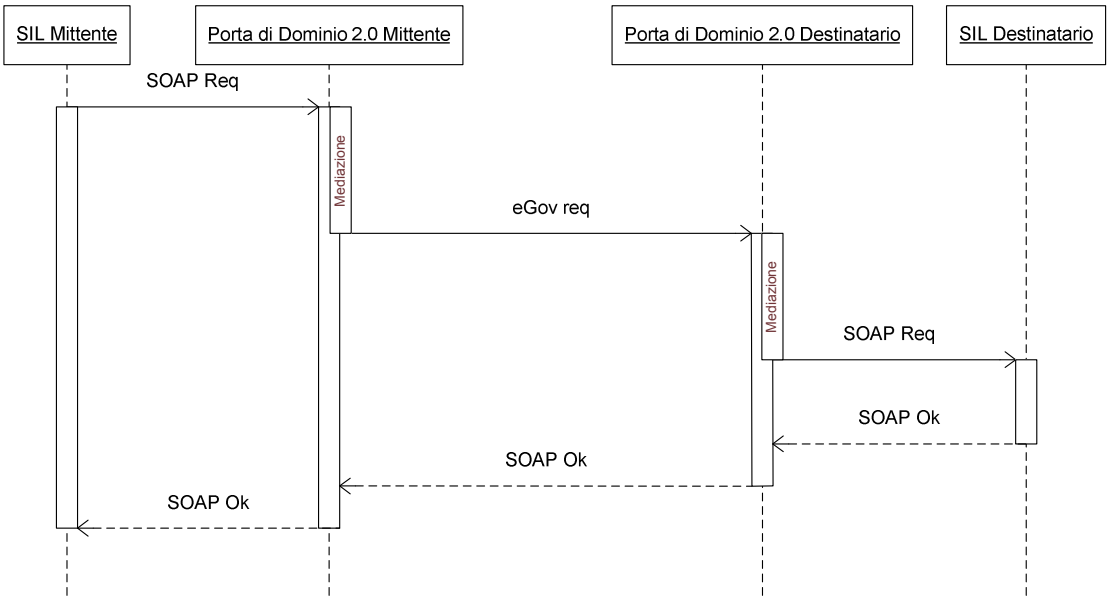
5.3.1 OneWay

Cominciamo dal paradigma OneWay. Vediamo tramite un diagramma di attività come la nuova architettura effettua la mediazione dei messaggi. Il diagramma di attività è un diagramma definito all'interno dello UML e definisce le attività da svolgere per realizzare una data funzionalità. Può essere utilizzato anche in fase di design per dettagliare un determinato algoritmo. Più in dettaglio un Diagramma di Attività definisce una serie di attività o flusso, anche in termini di relazioni tra di esse, chi è responsabile per la singola attività ed i punti di decisione. Il flusso è rappresentato tramite delle frecce orientate, che indicano la sequenza temporale con cui devono essere effettuate le diverse attività. È previsto un simbolo per indicare l'inizio del flusso ed un altro per indicarne il termine. Le attività possono essere anche rese in parallelo, in questo caso il punto di divisione (*fork*) è rappresentato da frecce divergenti rispetto al segmento. Nel caso le attività siano alternative, cioè svolte o meno rispetto ad una scelta, il punto di decisione è rappresentato da dei rombi da cui partono i flussi alternativi. Il punto di ricongiungimento (*join*) è reso tramite un segmento su cui le frecce si ricongiungono.

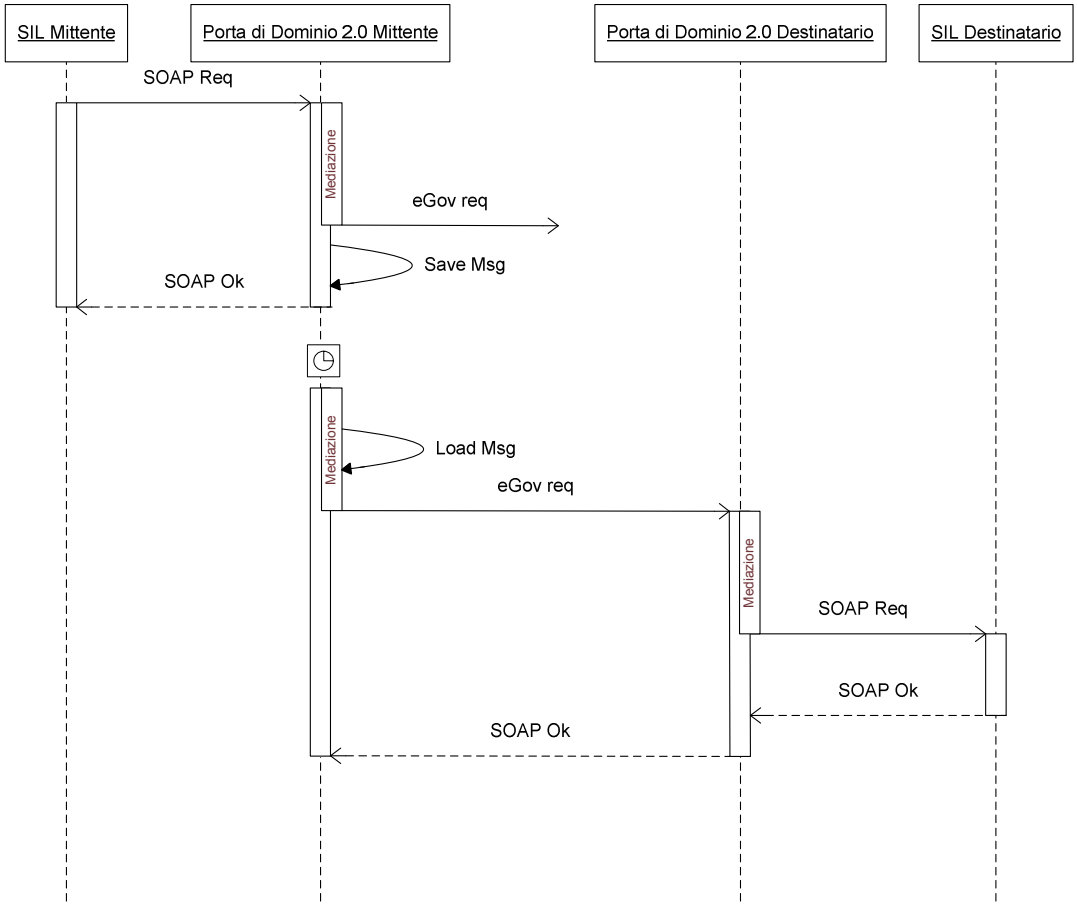


E’ importante notare che il processo è thread-based, in contrapposizione con il processo tradizionale basato sui principi dell’architettura SEDA. Quindi tutta la logica viene svolta da un unico processo nella sua interezza. Ricevuto un messaggio SOAP questo viene imbustato trasformandolo in una Busta eGov e inoltrata alla Porta di Dominio Destinataria. Qualora occorra un problema di comunicazione, il sistema procede con la presa in consegna del messaggio, salvandolo su database e costruendo le informazioni necessarie per l’inoltro da inserire nella coda JMS InoltroBusteEGov. Logicamente, quindi, il messaggio passa dal canale preferenziale al canale tradizionale, sfruttandolo per garantire la persistenza dei messaggi. Il lavoro svolto negli stadi fin qui attraversati viene conservato nel processo di transizione da un canale all’altro. I messaggi gestiti dal canale preferenziale non attraversano code JMS, quindi non si accodano ai messaggi in

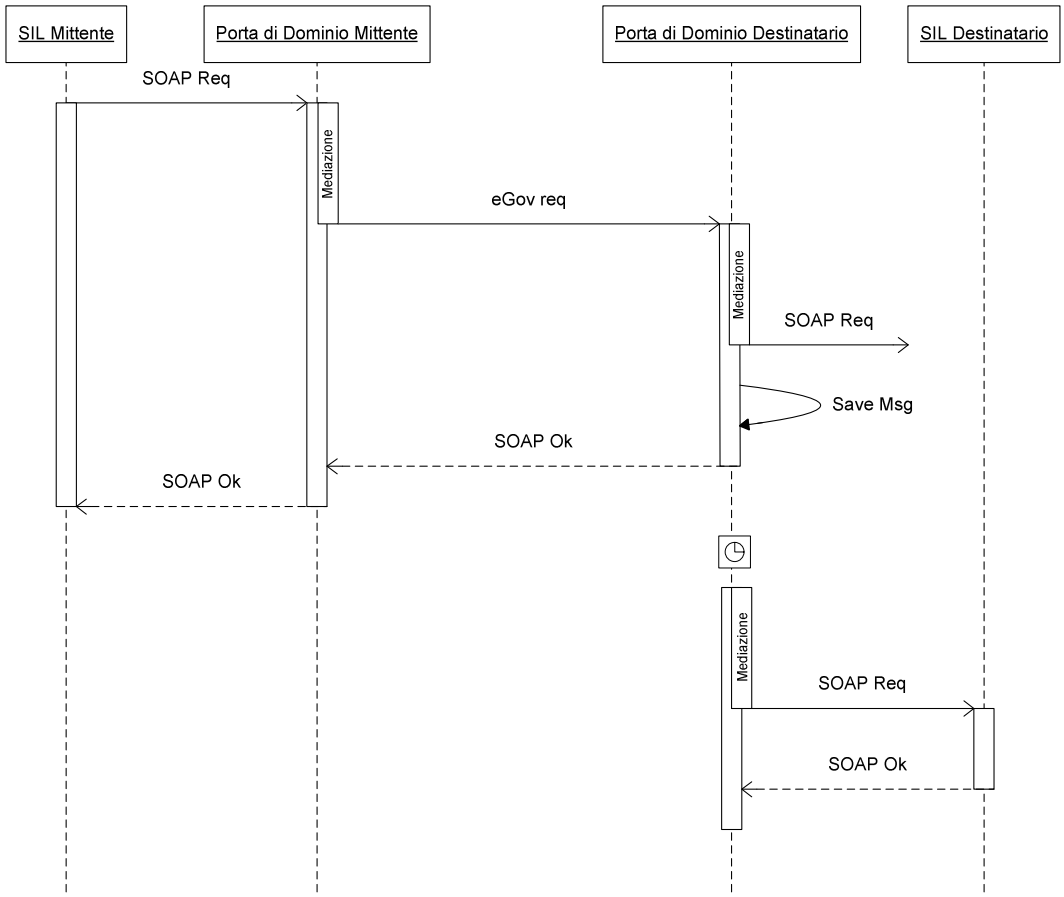
transito. Questo ha due implicazioni: non introduciamo l'overhead dovuto al frazionamento del processo di mediazione e riduciamo drasticamente il numero di accessi a database per salvare le informazioni sullo stato del messaggio, ma rinunciamo a eventuali politiche di load balancing o di priorità.
La nuova gestione a livello di processo modifica conseguentemente anche il livello di scambio di messaggi.



Come si nota dal diagramma di sequenza, in caso di successo il SIL Mittente non riceve subito l'ack SOAP dalla Porta di Dominio. La comunicazione quindi viene sospesa in attesa che il processo di mediazione sia portato a termine, al contrario del canale tradizionale che, a seguito della ricezione del messaggio, si occupava subito della presa in consegna e comunicava al mittente l'ack SOAP di conferma. In questo scenario non viene fatto nessun accesso a database, quindi, nel complesso, il processo di mediazione risulta più snello e rapido. Di contro, sarà chi invia il messaggio a dover attendere un tempo maggiore prima di ricevere la conferma SOAP, quindi, localmente, il processo risulta più lento.
Analizziamo adesso i casi di insuccesso parziale:



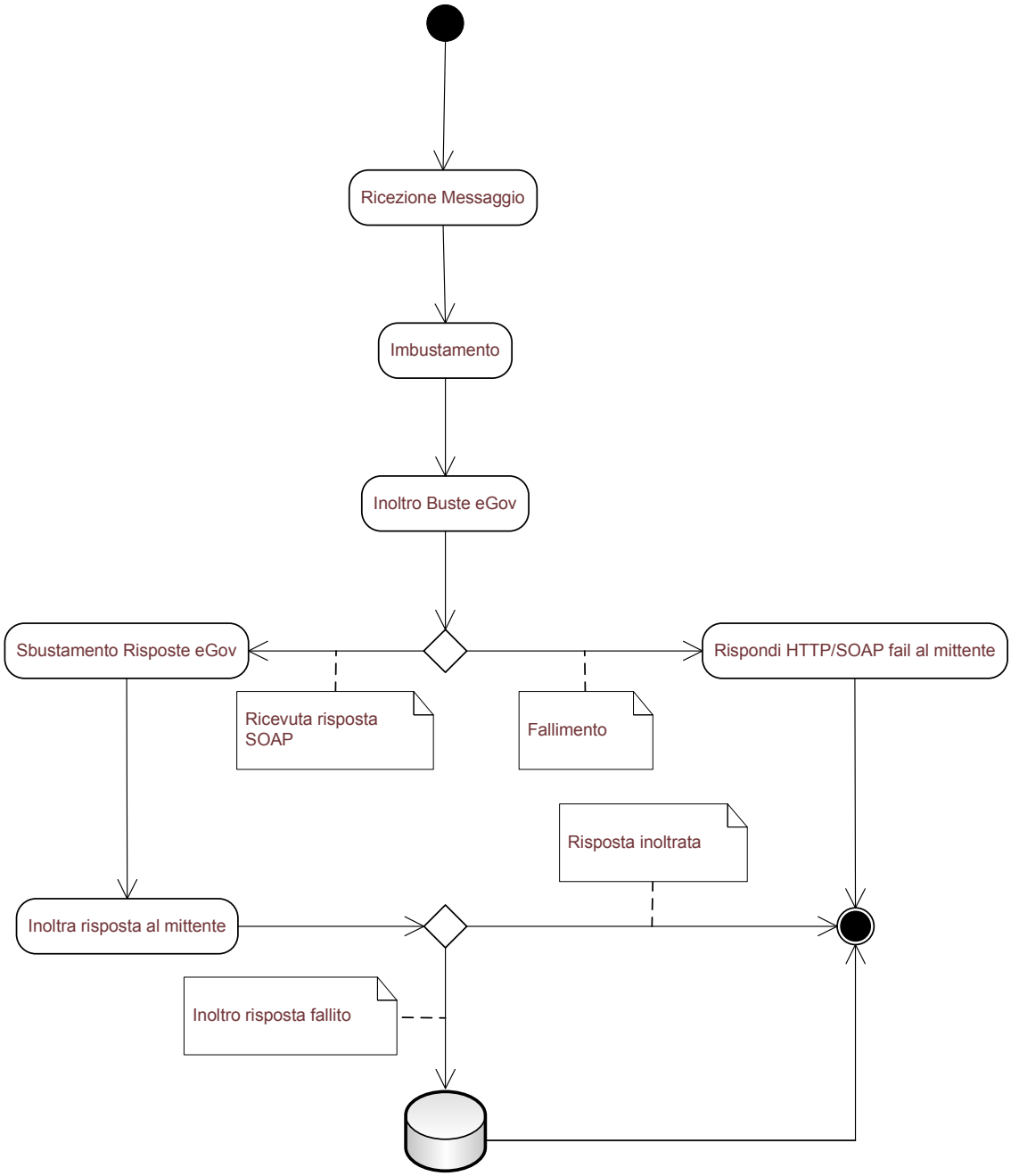
Qualora si verificassero degli errori di comunicazione, il sistema provvede alla presa in consegna del messaggio. Il messaggio viene quindi salvato su Database effettuandone la presa in consegna. Infine viene inviato l'ack SOAP al SIL Mittente. Quando la comunicazione riprenderà, i nodi successivi potranno ancora sfruttare il canale preferenziale, confermando la totale trasparenza del servizio agli altri nodi.



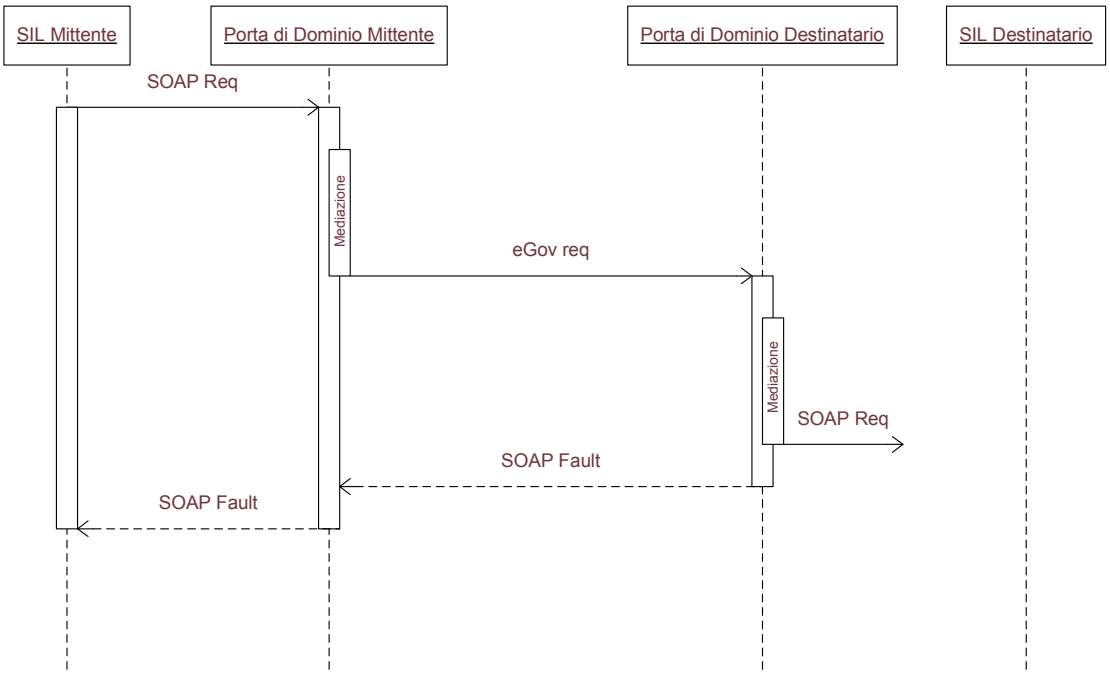
Anche in questo scenario il sistema garantisce la presa in carico del messaggio in caso di errore di comunicazione e provvedere all'ack SOAP. E' interessante notare come l'insorgere di errori di comunicazione influenzano la percezione della qualità del servizio per chi invia un messaggio secondo il paradigma OneWay. Se la comunicazione non ha errori, la ricezione del messaggio verrà confermata più tardi. Questo può sembrare controproducente, ma, visto nell'insieme, il sistema ne trae beneficio. Inoltre, come vedremo, l'uso dei due canali è configurabile, adattandosi alle esigenze funzionali dei SIL che ne fanno uso.

5.3.2 Request/Response Sincrono

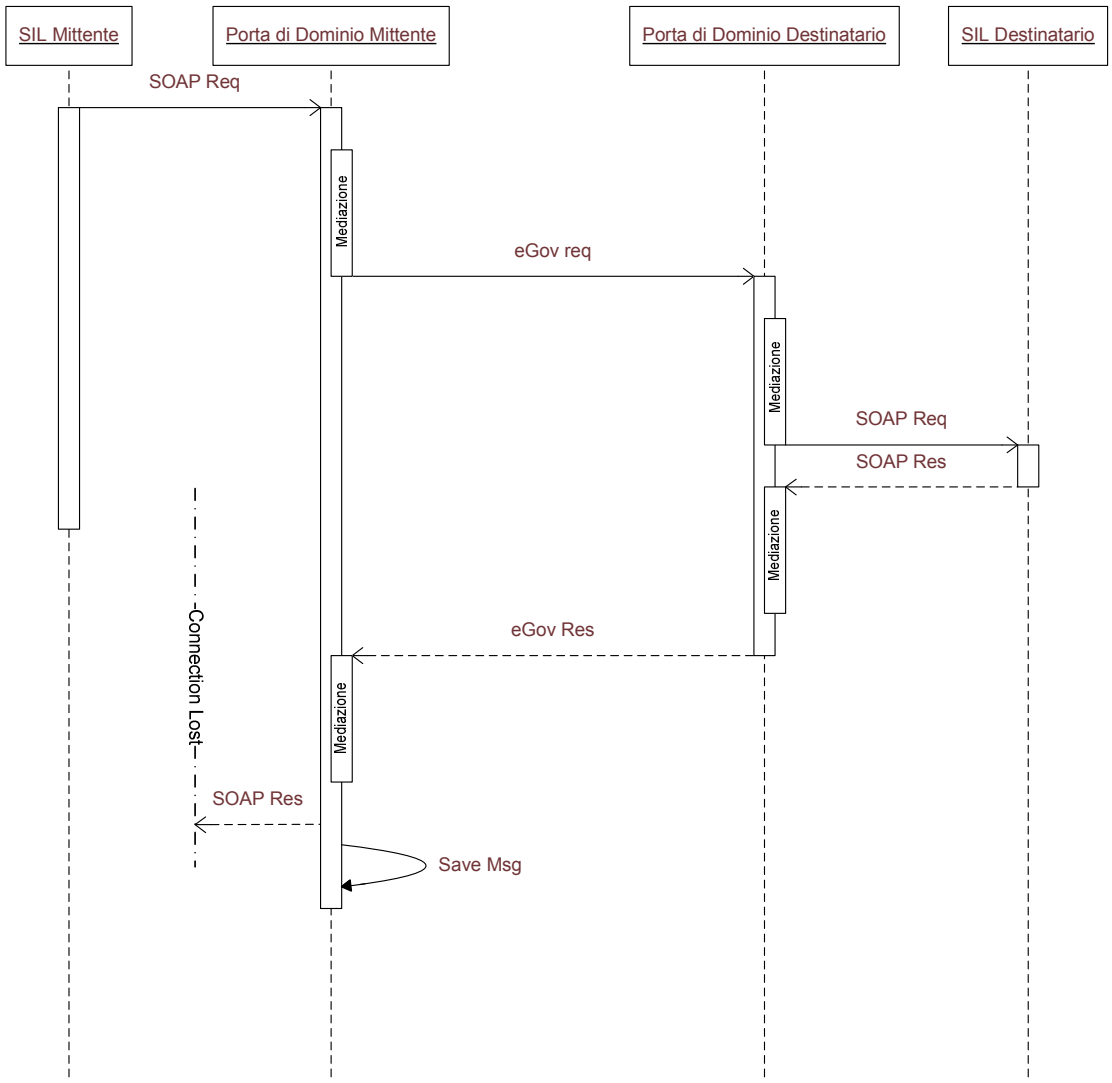
Analizziamo il caso in cui alla richiesta inviata dal SIL Mittente segua una risposta del SIL Destinatario. Andiamo ad analizzare il diagramma di attività dello scenario in questione.



Anche in questo caso abbiamo un'architettura thread-based come per il caso del OneWay, ma la gestione dei messaggi è naturalmente diversa. Nel caso del sincrono, se la comunicazione ha dei problemi di connessione nel tratto della richiesta, il processo complessivo termina totalmente, comunicando al SIL Mittente l'errore.



Se il problema si presenta nella trasmissione della risposta, allora devono essere eseguite delle procedure per utilizzare la risposta già elaborata qualora il mittente eseguisse la stessa richiesta, evitando che il SIL destinatario la processi due volte qualora fosse configurata una consegna “ALPIUUNAVOLTA”.



La gestione di questo tipo di errori è già implementata nell'architettura tradizionale di OpenSPCoop, quindi l'implementazione del canale preferenziale dovrà solamente replicare lo stato del messaggio del canale tradizionale al momento dell'errore per assicurarsi una corretta gestione.

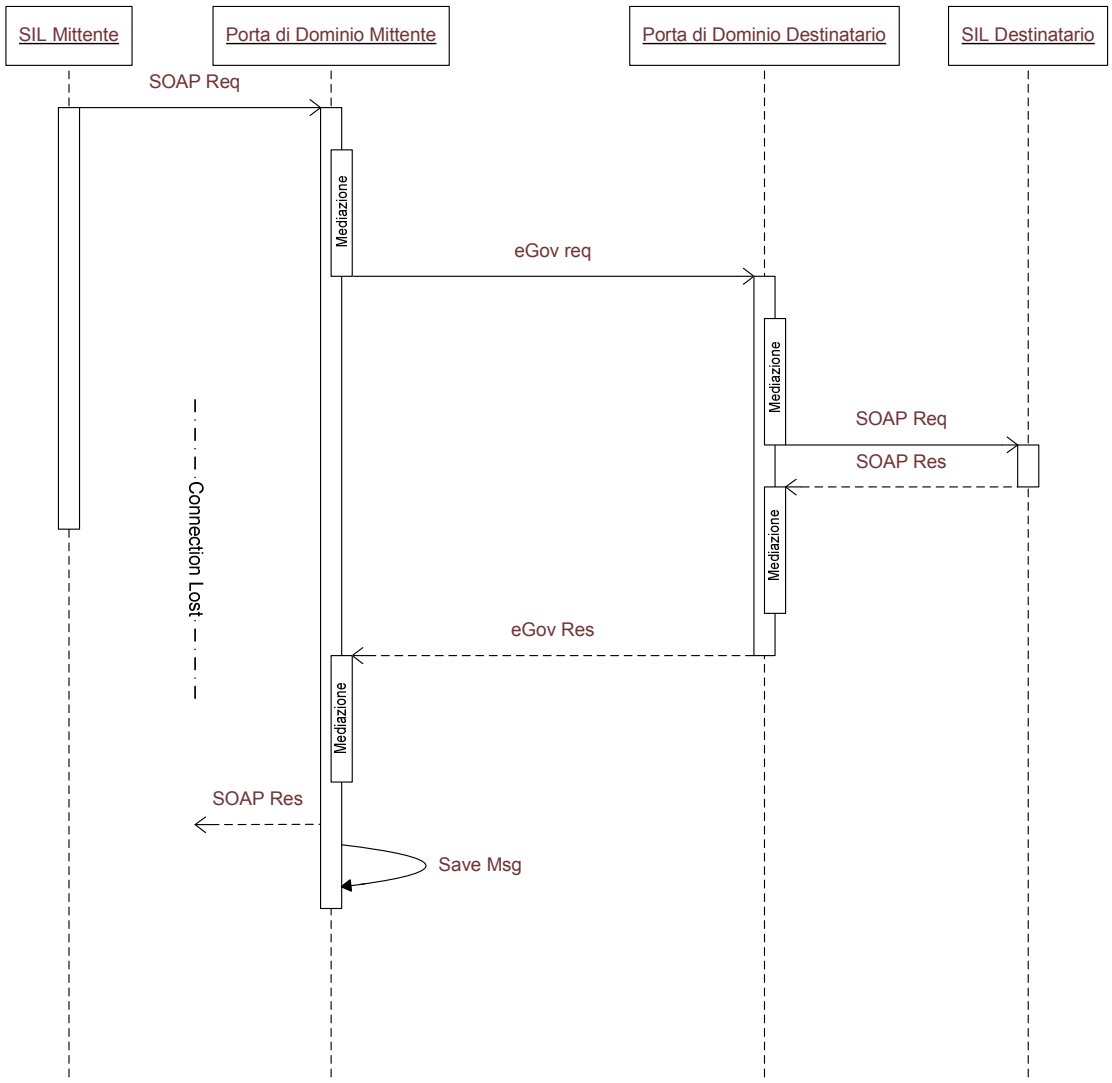
La componente di controllo della Porta di Dominio OpenSPCoop originale invece gestisce i messaggi tutti allo stesso modo, indipendentemente dal tipo di comunicazione, quindi salva sia i messaggi di richiesta che di risposta. In questa situazione quindi i benefici sono fruibili sia localmente sia globalmente. La gestione più snella della comunicazione e un tempo di risposta globalmente più rapido è godibile anche dal mittente, dovendo in ogni caso restare in attesa della risposta.

5.3.3 Gestione della memoria

L'architettura proposta, come mostrato, snellisce la gestione della mediazione dei messaggi da parte dei componenti di controllo nelle Porte di Dominio OpenSPCoop, introducendo al più un ritardo nei feedback verso il mittente nel caso di un messaggio nel profilo OneWay. C'è però un problema nuovo che l'architettura proposta introduce e che va adeguatamente gestito. L'architettura originale salvava tutti i messaggi ricevuti su database, e in memoria si limitava a tenere solo le informazioni necessarie alla mediazione. Nel canale preferenziale, introdotto dalla nuova architettura, il messaggio, per evitare gli accessi a DB, viene tenuto in memoria con tutte le complicazioni che ne derivano. E' quindi necessario introdurre un meccanismo che gestisca l'utilizzo del canale preferenziale filtrando i messaggi in ingresso smistandoli tra i due canali.
A tal proposito verrà effettuato un controllo a runtime che filtri i messaggi indirizzati al canale preferenziale in base alla dimensione del messaggio in transito, per evitare che messaggi troppo grandi, ad esempio con attachments di grandi dimensioni, saturino la memoria a disposizione. Inoltre un'altra selezione viene fatta in base al servizio richiesto a livello di configurazione della Porta di Dominio. Come anticipato, localmente il canale preferenziale può introdurre un ritardo nella comunicazione della ricezione del messaggio, quindi spetta a chi gestisce la Porta di Dominio determinare quale soluzione sia più appropriata per il servizio richiesto.

5.3.4 Errori di comunicazione

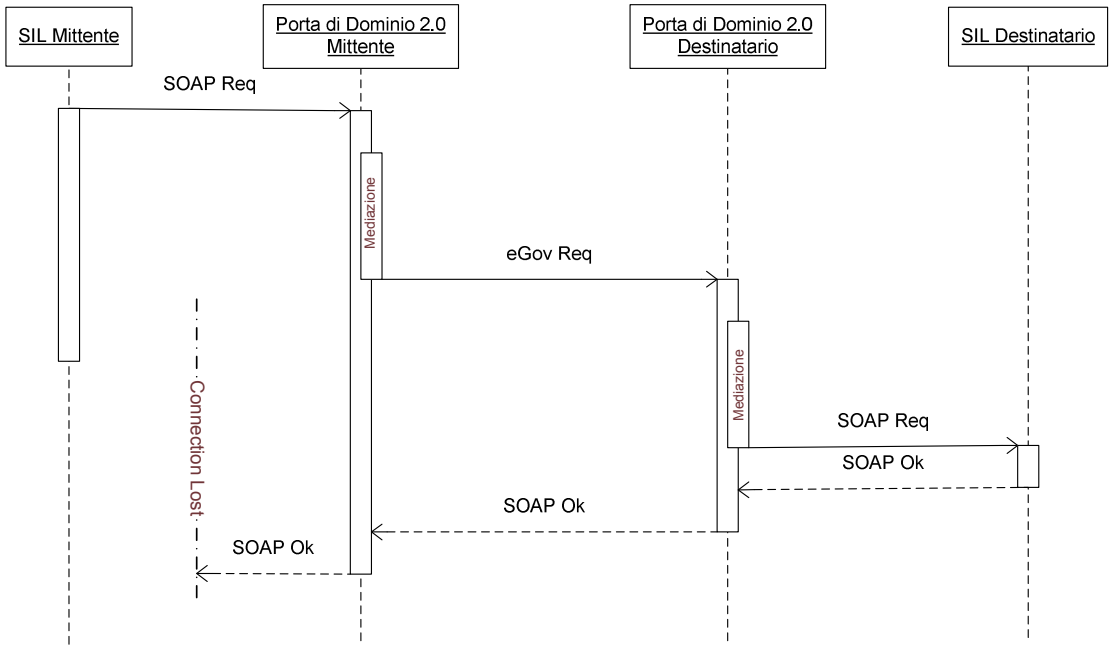
Quando abbiamo introdotto la nuova gestione dei messaggi e le modifiche alla gestione dei protocolli di comunicazione, sono emersi scenari parzialmente fallimentari dovuti ad errori di comunicazione. Come mostrato nel caso della una comunicazione sincrona, può capitare che mentre il messaggio viene mediato, una delle connessioni pendenti già attraversate cada e la risposta non possa più essere consegnata.



Questi parziali fallimenti possono far nascere una serie di questioni sul comportamento di OpenSPCoop per tentare di portare a termine comunque la comunicazione in momenti successivi. Abbiamo, infatti, una situazione in cui il messaggio è stato consegnato, ma il SIL Mittente crede che l'invio sia fallito. Se il SIL invia nuovamente la richiesta il messaggio può essere consegnato due volte.

Ovviamente la specifica SPCoop tratta questi problemi e la soluzione consiste in una serie di servizi addizionali forniti dall'infrastruttura, configurabili secondo le esigenze degli utenti, quali la Gestione dei Duplicati e la Correlazione Applicativa, che permettono, ad esempio, di garantire che una richiesta non venga consegnata due volte, come potrebbe accadere nel caso fallimentare precedentemente illustrato. La Porta di Dominio, per far questo, riconosce il secondo messaggio come duplicato e recupera la

risposta precedentemente ricevuta da consegnare al mittente. L'importante è sottolineare che queste funzionalità aggiuntive e sistemi di gestione degli errori sono già presenti nell'implementazione della Porta di Dominio OpenSPCoop e che la nuova architettura non introduce nuovi scenari fallimentari da dover gestire. Grazie alla possibilità di spostare logicamente il messaggio dal canale preferenziale a quello tradizionale, la nuova architettura eredita completamente le funzionalità implementate nella vecchia architettura. E' comunque da evidenziare che nel caso di comunicazioni con profilo OneWay, situazioni di connessioni pendenti che si interrompono, avranno un'incidenza maggiore, causata dalle connessioni tenute aperte più a lungo:



6. Prototipo di OpenSPCoop 2.0, basato sull'architettura progettata

A conclusione della progettazione della nuova architettura è seguita l'implementazione di un prototipo. La produzione del prototipo è stata condotta in due fasi:

- Riduzione delle funzionalità di OpenSPCoop 1.0 e introduzione di CXF
- Aggiunta del "canale preferenziale"

La prima fase consiste nel ridurre la complessità del modulo di controllo di OpenSPCoop1.0. Per far questo sono state eliminate tutte le operazioni non necessarie alla gestione del messaggio in termini di presa in carico e inoltro al nodo successivo. Per i nostri scopi ci siamo limitati all'implementazione della Porta Delegata del modulo di controllo, tralasciando l'implementazione della Porta Applicativa, sostanzialmente speculare, riducendola ad un semplice servizio di echo del messaggio in arrivo.

Prima di mostrare il comportamento del modulo di controllo, forniamo una breve descrizione delle operazioni effettuate da ogni fase (*stage*) del canale tradizionale nel prototipo:

RicezioneContenutiApplicativi. L'interfaccia con l'esterno è costituita dal Web Service implementato in *RicezioneContenutiApplicativiWS* in standard JAX-WS, gestito dal Web Service Framework CXF.

Questa fase si occupa di:

- Ricevere il messaggio SOAP.
- Identificare la Porta Delegata dall'URL di destinazione del messaggio.
- Salvare il messaggio SOAP su FileSystem.
- Aggiungere su database informazioni sullo stato del messaggio e sulla Porta Delegata identificata precedentemente, utili alle fasi successive.
- Creare il messaggio *ImbustamentoMessage* e inserirlo nella coda JMS Imbustamento
- Aspettare e prelevare il messaggio *RicezioneContenutiApplicativiMessage* dalla JMS RicezioneContenutiApplicativi correlato al messaggio di richiesta.
- Ricostruire il messaggio di risposta recuperato da FileSystem, sbustandolo se necessario.

- Inviare la risposta al SIL Mittente.

Imbustamento. Questa fase si occupa di:

- Individuare il tipo di scambio di messaggi incorso tra i SIL (OneWay o Sincrono).
- Se OneWay:
    - Creare una risposta di OK.
    - Salvare la risposta su FileSystem
    - Inserire un opportuno messaggio *RicezioneContenutiApplicativiMessage* nella coda JMS RicezioneContenutiApplicativi correlato alla richiesta.
- Costruire la Busta eGov da inoltrare alla fase successiva.
- Aggiornare su Database le informazioni sullo stato del messaggio SOAP.
- Inserire un messaggio InoltroBusteEGovMessage contenente la busta creata nella coda JMS InoltroBusteEGov.

InoltroBusteEGov. Questa fase si occupa di:

- Prelevare da FileSystem il messaggio SOAP
- Costruire, partendo dall'header eGov e dal messaggio SOAP, la Busta EGov.
- Inviare il messaggio alla Porta Applicativa opportuna.
- Ricevere la risposta
- Salvare la risposta su FileSystem
- Estrarre la Busta eGov dalla risposta
- Inserire la Busta eGov in uno SbustamentoRisposteEGovMessage e inoltrarlo nella coda SbustamentoRisposteEGov.

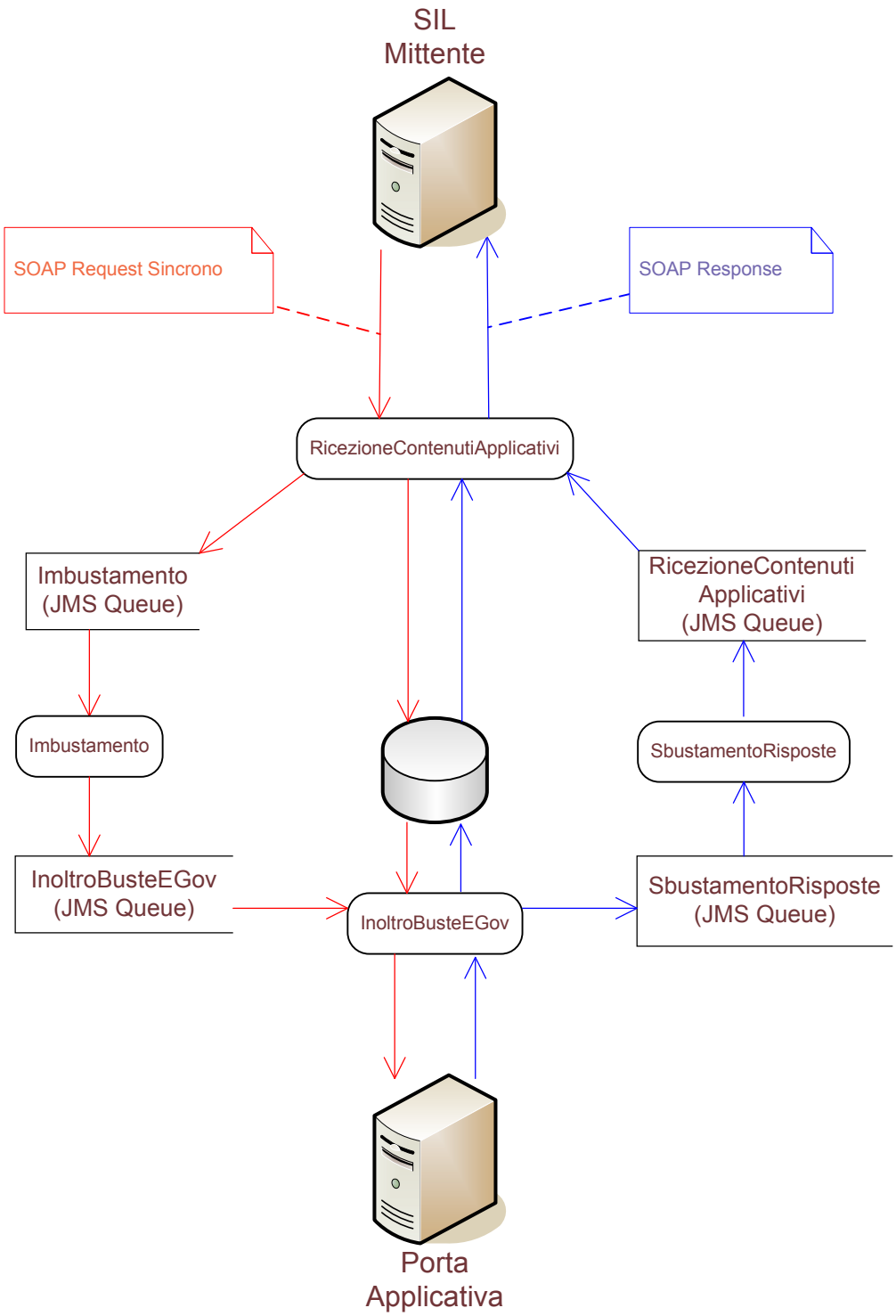
SbustamentoRisposteEGov. Questa fase si occupa di:

- Effettuare parte dei controlli sulla Busta EGov
    - Se il tipo di comunicazione è Sincrono
    - Inoltrare un messaggio *RicezioneContenutiApplicativiMessage* alla coda RicezioneContenutiApplicativi correlato alla richiesta.

L'elenco delle operazioni di ciascuna fase non è esaustivo, ma in linea di massima descrive la suddivisione dei compiti del processo di mediazione e permette di capire meglio i grafici successivi. Si nota comunque la mancanza di molti servizi di base comuni a molti servizi web e specifici di SPCoop che sono stati eliminati nel prototipo per ridurne la complessità e isolare il processo di presa in consegna e inoltro. Tra questi

cito a titolo di esempio: la verifica delle autorizzazioni necessarie all'uso del servizio, il tracciamento, la validazione sintattica e semantica dei messaggi, l'applicazione di politiche di sicurezza, la gestione dei duplicati, la consegna in ordine di messaggi multipli etc. I due grafi seguenti mostrano la sequenza con cui il controllo sul processo di mediazione passa di fase in fase nello scenario di comunicazione OneWay e Request Response Sincrono.

Erreur ! Des objets ne peuvent pas être créés à partir des codes de champs de mise en forme.



Completata la prima fase di produzione del prototipo, possiamo procedere con l'introduzione del nuovo "canale preferenziale", alla base della nuova architettura.

Raccogliendo le operazioni distribuite tra le quattro fasi del canale originale, sono state implementate le funzionalità nel nuovo processo di mediazione, epurandolo degli accessi a database, per lo stato del messaggio, ed a filesystem, per il salvataggio dello

stesso. Per favorire l'interpretazione del codice a chi conosce l'architettura tradizionale, il processo è sempre suddiviso in fasi, anche se non ci sono code JMS a dividerle. Tutte le informazioni sul processo di mediazione del messaggio sono contenute in un oggetto *MediationContext* passato di modulo in modulo.

RicezioneContenutiApplicativi. Utilizza come interfaccia per l'esterno il solito Web Service del canale tradizionale, essendo una funzionalità trasparente all'utente.

Questa fase si occupa di:

- Ricevere il messaggio SOAP.
- Creare il messaggio *ImbustamentoMessage* e inserirlo contesto di mediazione.
- Passare il controllo al modulo successivo e attendere che il controllo torni.
- Inviare la risposta contenuta nel contesto di mediazione.

Imbustamento. Questa fase si occupa di:

- Individuare il tipo di scambio di messaggi in corso tra i SIL (OneWay o Sincrono).
- Costruire la Busta eGov da inoltrare alla fase successiva.
- Inserire un messaggio InoltroBusteEGovMessage contenente la busta creata nel contesto di mediazione.
- Passare il controllo al modulo successivo

InoltroBusteEGov. Questa fase si occupa di:

- Costruire, partendo dall'header eGov e dal messaggio SOAP, la Busta EGov.
- Inviare il messaggio alla Porta Applicativa opportuna.
- Se fallisce
    - One Way
        - Inserisce una risposta OK nel contesto di mediazione
        - Inserisce il messaggio di richiesta nel canale tradizionale
            - Salva il messaggio
            - Aggiunge un messaggio *InoltroBusteEGovMessage* nella coda JMS *InoltroBusteEGov*
            - Inserisce opportuni record sullo stato del messaggio in database.
    - Sincrono
        - Inserisce un messaggio di risposta di errore nel contesto di

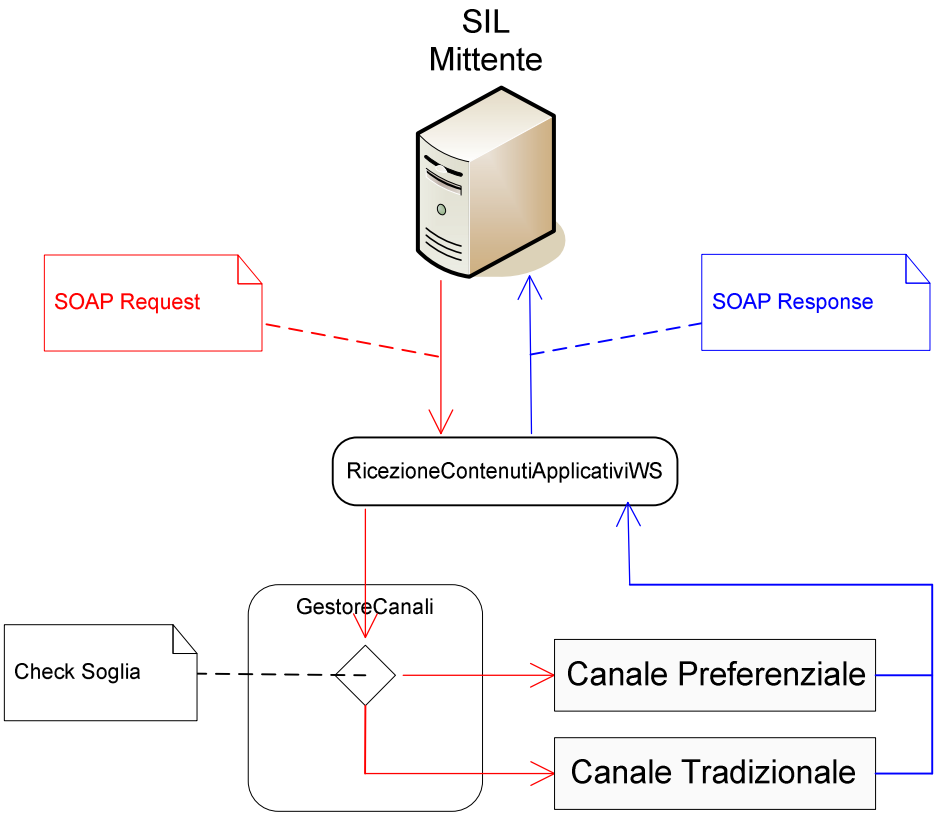
mediazione

- Se, invece, riceve la risposta la inserisce nel contesto di mediazione
- Passa il controllo al modulo successivo

SbustamentoRisposteEGov. Questa fase non compie nessuna operazione nel prototipo, non dovendo sbustare i messaggi e fare tutte le verifiche. Passa direttamente il controllo a *RicezioneContenutiApplicativi*.

Nel Web Service di interfaccia alla porta delegata è inserito il controllo sulla dimensione del messaggio. Questo viene effettuato prelevando dagli headers HTTP l'header Content-Length e confrontandolo con il valore di soglia impostato.

In futuro il canale da dover utilizzare sarà impostato come parametro di configurazione della Porta di Dominio. Se il canale configurato è quello preferenziale, a runtime interviene comunque il controllo sulla dimensione dei messaggi che possono fruire del canale e, in caso il controllo fallisca, ignora la configurazione e instrada il messaggio nel canale tradizionale. Molto importante sarà decidere il giusto valore di soglia da impostare per decidere se una richiesta può o meno essere gestita dal nuovo canale. Un valore troppo basso rischia di inibirne l'utilizzo anche in situazioni favorevoli, mentre un valore troppo alto riduce il numero di messaggi gestibili contemporaneamente.



Il punto cruciale e costituito dalle operazioni che gestiscono la transizione del messaggio dal canale preferenziale a quello tradizionale.

Vengono eseguite solitamente quando, durante l'utilizzo del canale preferenziale, si verifica un errore di comunicazione con il nodo successivo, nel nostro caso con la Porta Applicativa per la richiesta o con il SIL Mittente per la risposta.

L'operazione di "transitamento" implica il dover costruire esattamente lo stato che il messaggio avrebbe nel canale tradizionale.

In caso di un errore di comunicazione per il messaggio di risposta non viene eseguita nessuna operazione, dal momento che, non avendo implementato funzionalità come la Gestione Duplicati, non è di nessuna utilità salvare le risposte non consegnate. Questo ovviamente cambierà quando saranno implementate le funzionalità più evolute.

Il prototipo sviluppato fornisce la conferma sulla correttezza dell'architettura proposta riuscendo a replicare la funzionalità di presa in consegna dell'architettura precedente. Testare la performance del prototipo comparando i due canali, tradizionale e preferenziale, non fornirebbe dati utili. Sarebbe una vittoria schiacciante e scontata del

canale preferenziale. Solo un’analisi su un prodotto più avanzato, valutato in un ambiente che simula la realtà, con percentuali di fallimento delle comunicazioni attendibili e con un processo di mediazione completo fornirebbe una valutazione attendibile sul miglioramento della performance ottenuto.

7. Conclusioni

In questa tesi è stato presentato il processo tramite il quale è stato perseguito l'obiettivo di aggiornare il progetto OpenSPCoop, sia da un punto di vista architetturale e funzionale, sia per quanto concerne le specifiche supportate.

Risultati Ottenuti

Sviluppare una nuova architettura per il progetto OpenSPCoop ha richiesto una lunga ed approfondita analisi delle tecnologie e degli strumenti allo stato dell'arte utili per implementare le specifiche SPCoop. Questa analisi ci ha portato a valutare l'ambito Web Service in ogni suo aspetto: le architetture ideate, le implementazioni delle specifiche, i Web Service Framework ed i Web Service Mediator sviluppati dalla comunità open source. Grazie alle conoscenze acquisite in questa ricerca, è stato possibile ideare e sviluppare una nuova architettura per la mediazione di messaggi in grado di fornire la stessa qualità di servizio della precedente, ma che sopperisce alle sue carenze prestazionali in determinati scenari applicativi.

Come mostrato nel quarto e quinto capitolo, quest'architettura introduce una gestione dei messaggio in transito non riscontrata in altri prodotti concorrenti, evolvendo la precedente ed eliminando alcune inefficienze. La doppia gestione fornita, permette all'architettura di adattarsi alle necessità degli utenti nei diversi scenari di utilizzo.

Per valutare meglio l'architettura proposta è stato creato un prototipo di OpenSPCoop che ne implementa alcune funzionalità cruciali. Oltre questo, il prototipo utilizza un nuovo Web Service Framework, CXF, che sostituisce il vecchio, Axis. Il codice di OpenSPCoop è stato quindi revisionato completamente per utilizzare le API più recenti messe a disposizione da questo strumento, raggiungendo così l'altro obiettivo della tesi, quello di conformare OpenSPCoop alle nuove specifiche in ambito Web Service.

Il prototipo costruito si presenta quindi come un ottimo banco di prova per gli sviluppatori, sia per testare la nuova gestione dei messaggi che l'architettura introduce, sia per valutare le funzionalità aggiunte dalle API.

Sviluppi Futuri

Il prototipo sviluppato non implementa tutta la fase di gestione dei messaggi, ma solo quello che concerne la presa in consegna e l'inoltro in comunicazioni OneWay e

Sincrone, pur mantenendo, per quello definito canale tradizionale, la stessa architettura di OpenSPCoop1.0. Inoltre non sono gestiti tutti i possibili errori che possono presentarsi nel processo di mediazione, ma si limita a gestire parte degli errori di comunicazione con la Porta Applicativa.

In future versioni saranno gestiti tutti gli scenari di comunicazione ed aggiunte le funzionalità che implementano le specifiche SPCoop. Questo inciderà sulle operazioni che dovrà eseguire il modulo preposto alle operazioni di transito dal canale tradizionale al canale preferenziale.

Per poter valutare in maniera esaustiva la nuova architettura occorre implementare tutte le funzionalità di OpenSPCoop e testarlo in scenari reali di utilizzo.

Gestore della Memoria

Il componente che gestisce l'utilizzo del canale rapido sarà cruciale per non sovraccaricare il sistema e contemporaneamente gestire il maggior numero di messaggi con il canale preferenziale.

Componente di Transizione

Le operazioni di accesso a database che permettono di transitare da un canale all'altro nella nuova architettura sono, come era per l'architettura precedente, il maggiore responsabile dell'overhead introdotto dalle operazioni di presa in carico. Sarà necessario quindi un lungo lavoro di ottimizzazione per ridurle al minimo.

Applicazioni

L'introduzione della nuova architettura nel progetto OpenSPCoop è stata programmata per la versione 2.0, non appena il prototipo sarà evoluto in un prodotto completo e testato adeguatamente. L'architettura proposta è comunque applicabile non solo al progetto OpenSPCoop, ma anche ad altri sistemi di Web Service Mediation e in generale ad applicazioni in ambito ESB.

Appendice: codice allegato

Nelle pagine seguenti viene riportata una porzione del codice del prototipo sviluppato.

Il prototipo implementa le sole funzioni di presa in carico e consegna per alcuni scenari di utilizzo. La parte di codice si riferisce all'implementazione del canale preferenziale, mentre è stata omessa quella del canale tradizionale.

Le librerie OpenSPCoop sulle quali si appoggia sono quelle della versione 1.0b1 reperibili sul sito di riferimento [05].

Il Web Service Framework utilizzato è CXF versione 2.0.3-incubator-20071016.175632-4. L'Application Server J2EE usato nello sviluppo è JBOSS, versione 4.0.5GA.

Sono presentati i seguenti file:

- *web.xml*: E' il *Deployment Descriptor*, un componente delle applicazioni J2EE che fornisce informazioni utili all'application server, JBOSS, per eseguire il depoy dell'applicazione.
- *openspcoop_cxf.xml*: Un file di configurazione di Spring, un framework open source utilizzato per lo sviluppo di applicazioni web su piattaforma Java, utilizzabile, con qualche modifica, nei Web Service Framework che lo supportano. Nel nostro caso è configurato per CXF.
- *MediationContext.java*: La classe che definisce il contenitore degli oggetti creati durante il processo di mediazione. Viene solo mostrata la parte di definizione degli oggetti contenuti, omettendo i metodi che ne consentono l'accesso (*setter* e *getter*).
- *RicezioneContenutiApplicativiWS.java*: Implementazione del Web Service di Porta Delegata in standard JAX-WS. Include il modulo che gestisce l'inoltro sul canale tradizionale o su quello preferenziale in base alla dimensione del messaggio di richiesta.
- *RicezioneContenutiApplicativi.java*: Prima fase di mediazione. Si occupa di integrare informazioni sulla porta delegata richiamata e di inoltrare la richiesta al modulo successivo.
- *Imbustamento.java*: Seconda fase di mediazione. Si occupa di creare la BustaEGov.
- *InoltroBusteEGov*: Terza fase di mediazione. Imbusta la richiesta nella

BustaEGov e la inoltra al destinatario, la Porta Applicativa. Se non riesce l'inoltro, effettua la presa in consegna, passando il controllo del processo di mediazione al canale tradizionale. Se la consegna procede correttamente passa la risposta alla fase successiva.

- *SbustamentoRisposte.java*: Quarta e ultima fase di mediazione. Nel prototipo non esegue nessuna operazione. In future versioni eseguirà le operazioni di analisi sulla BustaEGov della risposta. Si limita a consegnare la risposta al Web Service per inviarla al SIL Mittente.
- *SoapUtils.java*: Classe di utility per la creazione di messaggi SOAP di Ok o di errore.

Configurazione deployment descriptor: *web.xml*

```
<web-app>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>WEB-INF/openspcoop_cxf.xml</param-value>
    </context-param>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <servlet>
        <servlet-name>openspcoop</servlet-name>
        <display-name>openspcoop</display-name>
        <servlet-class>
            org.apache.cxf.transport.servlet.CXFServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>openspcoop</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Configurazione Spring di OpenSPCoop: *openspcoop_cxf.xml*

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation=
    "http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">
    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension--binding.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
    <jaxws:endpoint id="PD"
        implementor="org.openspcoop.pdd.services.RicezioneContenutiApplicativiWS"
        address="/PD" />
    <jaxws:endpoint id="PA"
        implementor="org.openspcoop.pdd.services.RicezioneBusteEGovWS"
        address="/PA" />
</beans>
```

Contesto di Mediazione: *MediationContext.java*

```
package org.openspcoop.pdd.fastchannel;

import org.openspcoop.pdd.mdb.ImbustamentoMessage;
import org.openspcoop.pdd.mdb.InoltroBusteEGovMessage;
import org.openspcoop.pdd.mdb.SbustamentoRisposteMessage;
import
org.openspcoop.pdd.services.RicezioneContenutiApplicativiContext;
import
org.openspcoop.pdd.services.RicezioneContenutiApplicativiMessage;

public class MediationContext {

/**
 * Classe contenente tutte le informazioni relative
 * al processo di mediazione di un messaggio.
 * Portarsi dietro anche i dati relativi a fasi già passate per
 * effettuare la presa in consegna in qualsiasi punto della mediazione
 *
 * In future versioni si può notevolmente ottimizzare "pulendolo"
 * di oggetti che non serve più ricordare una volta conclusa una fase.
 */

      private RicezioneContenutiApplicativiContext ricezioneCtx;
      private ImbustamentoMessage imbustamentoMsg;
      private RicezioneContenutiApplicativiMessage ricezioneMsg;
      private String idEGovRichiesta;
      private String idEGovRisposta;
      private InoltroBusteEGovMessage inoltroMsg;
      private SbustamentoRisposteMessage sbustamentoRisposteMSG;
      private String profiloDiCollaborazione;


      /**
       *
       * Setter e Getter
       *
       */

      // Metodi per settare e prelevare gli oggetti contenuti
      // nel contesto di mediazione.

      .....
      .....
}
```

Web Service Porta Delegata: *RicezioneContenutiApplicativiWS.java*

```
package org.openspcoop.pdd.services;

import javax.xml.soap.SOAPMessage;
import javax.xml.ws.Provider;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;
import javax.xml.ws.Service.Mode;

import org.openspcoop.pdd.core.ParametriPortaDelegata;
import org.openspcoop.pdd.fastchannel.MediationContext;

//Modalità di ricezione della richiesta
@ServiceMode(value=Mode.MESSAGE)

//Implementiamo Provider<SOAPMessage> per avere un messaggio SAAJ
@WebServiceProvider

/**
 * Definizione di un Web Service in standard JAX-WS
 *
 * Il servizio è la Porta Delegata di una PdD
 * CXF si preoccupa di fornirci il messaggio in
 * ingresso in forma di SOAPMessage di SAAJ
 *
 * Il servizio è fruibile da
 * http://localhost:8080/openspcoop/PD/
 *
 * Finchè non ritorna il controllo dalle fasi di
 * mediazione il mittente non riceve alcuna risposta
 *
 */

public class RicezioneContenutiApplicativiWS implements
Provider<SOAPMessage>{
      public final static String ID_MODULO =
"RicezioneContenutiApplicativiWS";

      public SOAPMessage invoke(SOAPMessage request){



      /**
       * Parametri forzati per Prototipo.
       */

      RicezioneContenutiApplicativiContext context =
            new RicezioneContenutiApplicativiContext();
      ParametriPortaDelegata parPD = new ParametriPortaDelegata();
      parPD.setLocation("HelloWorld");
      parPD.setUrlInvocazionePD("HelloWorld");

      context.setIdModulo(RicezioneContenutiApplicativiWS.ID_MODULO);
      context.setGestioneRisposta(true);
      context.setInvocazionePDPerRiferimento(false);
      context.setParPD(parPD);
      context.setMessageRequest(request);
```

```
//Cerco l'header ContentType

      int size = 1001;
      MimeHeaders mhs = request.getMimeHeaders();
      Iterator it = request.getMimeHeaders().getAllHeaders();
      while(it.hasNext()){
            MimeHeader mh = (MimeHeader) it.next();
            System.out.println(mh.getName() + " : " + mh.getValue());
            If(mh.getName().equalsTo("Content-Type"))
                  size = Integer.parseInt(mh.getValue());
      }

      /**
       * Chiamo il canale giusto
       */
      if(size<1000){
            //Messaggio sotto la soglia. Invio a Canale Preferenziale

            RicezioneContenutiApplicativi gestoreRichiestaPref =
                  new RicezioneContenutiApplicativi(context);

            MediationContext mc = new MediationContext();
            mc = gestoreRichiestaPref.process(mc);
            context = mc.getRicezioneContenutiApplicativiContext();
      }
      else{
      //Messaggio troppo grande, mando al canale tradizionale
      org.openspcoop.pdd.service.RicezioneContenutiApplicativi
            gestoreRichiestaTrad = new
    org.openspcoop.pdd.service.RicezioneContenutiApplicativi(context);
      gestoreRichiestaTrad.process();

      }
      return context.getMessageResponse();
      }

}
```

MediazioneFase 1: *RicezioneContenutiApplicativi.java*

```
package org.openspcoop.pdd.fastchannel;

import org.apache.log4j.Logger;
import org.openspcoop.dao.config.driver.IDPortaDelegata;
import org.openspcoop.dao.registry.driver.IDSoggetto;
import org.openspcoop.egov.Imbustamento;
import org.openspcoop.egov.ProprietaErroreApplicativo;
import org.openspcoop.pdd.config.OpenSPCoopProperties;
import org.openspcoop.pdd.config.RichiestaDelegata;
import org.openspcoop.pdd.core.CostantiPdD;
import org.openspcoop.pdd.core.IdentificazionePortaDelegata;
import org.openspcoop.pdd.core.ParametriPortaDelegata;
import org.openspcoop.pdd.core.autorizzazione.IAutorizzazione;
import org.openspcoop.pdd.core.integrazione.IGestoreIntegrazionePD;
import org.openspcoop.pdd.mdb.ImbustamentoMessage;
import
org.openspcoop.pdd.services.RicezioneContenutiApplicativiContext;

/**
 * Implementazione del servizio RicezioneContenutiApplicativi
 * del canale Preferenziale
 *
 * Questa fase si occupa principalmente di raccogliere
 * le informazioni sulla porta delegata chiamata
 * e passarle al contesto di mediazione.
 *
 * Fatto questo passa il controllo alla fase di imbustamento.
 *
 * Quando torna il controllo si occupa di inoltrare il
 * messaggio di risposta al mittente.
 * Se non riesce a inoltrarlo, nel caso Sincrono, dovrebbe salvarlo,
 * ma poichè nel prototipo
 * non gestiamo configurazioni quali la consegna
 * ALPIUUNAVOLTA, se fallisce non facciamo niente.
 *
 * Importante: Non sono gestiti gli errori di processamento.
 * Ne qui, ne in molte altre parti del prototipo.
 * Nel prototipo e in un ambiente di test non si verificano
 * mai e vogliamo tenerlo il più semplice possibile
 *
 */

public class RicezioneContenutiApplicativi {

      /** Indicazione se sono state inizializzate le variabili del
servizio */
      public static boolean initializeService = false;


      /** IGestoreIntegrazionePD: lista di gestori, ordinati per
priorita' minore */
      public static java.util.Hashtable<String,IGestoreIntegrazionePD>
            gestoriIntegrazionePD =
            new java.util.Hashtable<String,IGestoreIntegrazionePD>();
      public static String[] defaultGestoriIntegrazionePD = null;
```

```
/** IAutorizzazione: lista di gestori per l'autorizzazione */
public static java.util.Hashtable<String,IAutorizzazione>
      gestoriAutorizzazione = new
      java.util.Hashtable<String,IAutorizzazione>();


Logger logCore = null;

/**
 * Inizializzatore del servizio RicezioneContenutiApplicativi
 */

/** Contesto della richiesta */
private RicezioneContenutiApplicativiContext msgContext;

/** Modalita' di gestione del Messaggio */
boolean responseAsByte = false;


/** Costruttore **/
public RicezioneContenutiApplicativi
      (RicezioneContenutiApplicativiContext context){
      this.msgContext = context;
}

/** Metodo di mediazione **/
public MediationContext process(MediationContext mc){



/**
 *
 * ------------  Lettura parametri della richiesta -------------
 *
 **/

// Parametri della porta delegata invocata
ParametriPortaDelegata parPD = this.msgContext.getParPD();



// PropertiesReader per la configurazione
OpenSPCoopProperties propertiesReader =
            OpenSPCoopProperties.getInstance();

//    IdentificativoPdD
IDSoggetto identitaPdD =
      propertiesReader.getIdentitaPortaDefault();


//    ProprietaErroreApplicativo
ProprietaErroreApplicativo proprietaErroreAppl =
      propertiesReader.getProprietaGestioneErrorePD();
proprietaErroreAppl.setDominio(identitaPdD.getCodicePorta());
proprietaErroreAppl.setIdModulo(this.msgContext.getIdModulo());
this.msgContext.setProprietaErroreAppl(proprietaErroreAppl);

// Identita' errore
```

```
      String identitaRichiedente = "PD:"+parPD.getLocation()+" Url
invocazione:"+parPD.getUrlInvocazionePD()+" \n";


/**
 *
 * ---------------- Inizio Mediazione --------------
 *
 */


System.out.println("[Canale Preferenziale] Mediazione Richiesta in
corso...");

/**
 *
 *  Identificazione Porta Delegata e SoggettoFruitore
 *          *** Forzati in prototipo ***
 *
 **/



parPD.setLocation("HelloWorld");
parPD.setUrlInvocazionePD("://localhost:8080/openspcoop/PD/HelloWorld"
);
IdentificazionePortaDelegata identificazione =
      new IdentificazionePortaDelegata(parPD);

// aggiorno nome PortaDelegata
parPD.setLocation(identificazione.getNomePDIndivituata());

// Raccolgo dati
IDSoggetto soggettoFruitore = identificazione.getSoggettoSPCoop();
identitaPdD = soggettoFruitore;
IDPortaDelegata idPD = new IDPortaDelegata();
idPD.setLocationPD(parPD.getLocation());
idPD.setSoggettoFruitore(soggettoFruitore);

// altri contesti
proprietaErroreAppl.setDominio(identitaPdD.getCodicePorta());

//Setto i dati raccolti nel contesto di RicezioneContenutiApp.
this.msgContext.setProprietaErroreAppl(proprietaErroreAppl);

String servizioApplicativo = CostantiPdD.SERVIZIO_APPLICATIVO_ANONIMO;

identitaRichiedente = "PD:"+parPD.getLocation()+"
      servizioApplicativo:"+servizioApplicativo+"\n";


/**
 *  Inizializzazione Contesto di gestione della Richiesta
 **/


// Costruzione ID eGov della busta che sarà creata.
String idEGovRequest = null;
```

```
try{
      Imbustamento imbustatore = new Imbustamento();
      idEGovRequest =
            imbustatore.buildID_eGov(
                  null,
                  identitaPdD.getCodicePorta(),
                  identitaPdD.getNome());
      //Inserisco l'idEgov nel contesto
      mc.setIdEGovRichiesta(idEGovRequest);
}
catch(Exception e){
      System.out.println("Errore costruzione eGov");
      return mc;
}

/**
 *  Spedizione a modulo Imbustamento dell canale Preferenziale
 **/

System.out.println("[Canale Preferenziale]
[RicezioneContenutiApplicativi] Invio messaggio al modulo di
Imbustamento...");

String idModuloInAttesa = null;
if(this.msgContext.isGestioneRisposta())
idModuloInAttesa = this.msgContext.getIdModulo();

RichiestaDelegata richiestaDelegata =
      new RichiestaDelegata(
            soggettoFruitore,
            parPD.getLocation(),
            servizioApplicativo,
            idModuloInAttesa,
            proprietaErroreAppl,
            identitaPdD);

// Inserisco il contesto di RicezioneContenutiApplicativi
// nel Contesto di Mediazione

mc.setRicezioneContenutiApplicativiContext(msgContext);

//Creo oggetto ImbustamentoMessage con i dati raccolti
ImbustamentoMessage imbustamentoMSG = new ImbustamentoMessage();
imbustamentoMSG.setRichiestaDelegata(richiestaDelegata);

//Inserisco l'oggetto ImbustamentoMessage nel contesto di mediazione
mc.setImbustamentoMessage(imbustamentoMSG);
org.openspcoop.pdd.fastchannel.Imbustamento imbustamento =
                  new org.openspcoop.pdd.fastchannel.Imbustamento();

//Procedo alla fase successiva
return imbustamento.mediate(mc);
}
}
```

MediazioneFase 2: *Imbustamento.java*

```
package org.openspcoop.pdd.fastchannel;

import org.openspcoop.egov.Busta;
import org.openspcoop.egov.Costanti;
import org.openspcoop.pdd.config.RichiestaDelegata;
import org.openspcoop.pdd.mdb.ImbustamentoMessage;
import org.openspcoop.pdd.mdb.InoltroBusteEGovMessage;

/**
 * Fase imbustamento del canale preferenziale
 *
 * Si occupa di costruire la busta eGov partendo dalle informazioni
 * contenute in ImbustamentoMessage racchiuso nelMediationContext
 *
 * Nel prototipo la costruzione della busta adeguat
 * al messaggio di richiesta
 * è sostituita da una busta di comodo.
 *
 */

public class Imbustamento {

public MediationContext mediate(MediationContext mc) {

System.out.println("[Canale Preferenziale][Imbustamento] Ricezione
richiesta ...");

/*  Lettura parametri da ImbustamentoMessage nel Mediation Context */

ImbustamentoMessage imbustamentoMsg = null;
imbustamentoMsg = (ImbustamentoMessage) mc.getImbustamentoMessage();

RichiestaDelegata richiestaDelegata =
      imbustamentoMsg.getRichiestaDelegata();

/**
 *
 *  Costruisco una busta di prova per i test
 *
 */

Busta eGov_Test = new Busta();
eGov_Test.setDestinatario("MinisteroErogatore");
eGov_Test.setTipoMittente("SPC");
eGov_Test.setTipoDestinatario("SPC");
eGov_Test.setMittente("MinisteroFruitore");
eGov_Test.setServizio("ServizioOneWay");
eGov_Test.setTipoServizio("SPC");
eGov_Test.setAzione("Ping");

if(mc.getProfiloDiCollaborazione().equals(
      Costanti.PROFILO_COLLABORAZIONE_ONEWAY))
      eGov_Test.setProfiloDiCollaborazione(
            Costanti.PROFILO_COLLABORAZIONE_ONEWAY);
else
      eGov_Test.setProfiloDiCollaborazione(
```

```
            Costanti.PROFILO_COLLABORAZIONE_SINCRONO);


      // Per vedere se devo fare un OK al SIL Mittente

if(Costanti.PROFILO_COLLABORAZIONE_ONEWAY.equals(
      eGov_Test.getProfiloDiCollaborazione())){

      richiestaDelegata.setScenario(
            Costanti.SCENARIO_ONEWAY_INVOCAZIONE_SERVIZIO);
            }
else
      richiestaDelegata.setScenario(
            Costanti.SCENARIO_SINCRONO_INVOCAZIONE_SERVIZIO);


/** ------------
 *
 * Spedizione InoltroBusteEGovMessage
 * al modulo InoltroBusteEGov del canale Fast
 *
 * ------------- **/


System.out.println("[Canale Preferenziale][Imbustamento]Invio
messaggio al modulo di InoltroBusteEGov...");

//Creo oggetto InoltroBusteEGovMessage
//con le info raccolte e la busta eGov creata

//Inserisco tutto nel Contesto di Mediazione

InoltroBusteEGovMessage inoltroMSG = new InoltroBusteEGovMessage();
inoltroMSG.setRichiestaDelegata(richiestaDelegata);
inoltroMSG.setBusta(eGov_Test);
mc.setInoltroBusteEGovMessage(inoltroMSG);

//Passo alla fase successiva, l'inoltroBusteEgov

InoltroBusteEGov inoltroBusteEGov = new InoltroBusteEGov();
return inoltroBusteEGov.mediate(mc);
}
}
```

MediazioneFase 3: *InoltroBusteEGov.java*

```
package org.openspcoop.pdd.fastchannel;

import java.sql.Connection;
import javax.xml.soap.SOAPFault;
import javax.xml.soap.SOAPMessage;
import org.openspcoop.dao.config.Connettore;
import org.openspcoop.dao.config.ConnettoreProperty;
import org.openspcoop.dao.registry.driver.IDSoggetto;
import org.openspcoop.egov.Busta;
import org.openspcoop.egov.Costanti;
import org.openspcoop.egov.Eccezione;
import org.openspcoop.egov.ProprietaErroreApplicativo;
import org.openspcoop.egov.XMLBuilder;
import org.openspcoop.pdd.config.ClassNameProperties;
import org.openspcoop.pdd.config.DBManager;
import org.openspcoop.pdd.config.RichiestaDelegata;
import org.openspcoop.pdd.core.CostantiPdD;
import org.openspcoop.pdd.core.GestoreMessaggi;
import org.openspcoop.pdd.core.GestoreMessaggiException;
import org.openspcoop.pdd.core.JMSSender;
import org.openspcoop.pdd.core.connettori.ConnettoreMsg;
import org.openspcoop.pdd.core.connettori.IConnettore;
import org.openspcoop.pdd.logger.MsgDiagnostico;
import org.openspcoop.pdd.mdb.Imbustamento;
import org.openspcoop.pdd.mdb.InoltroBusteEGovMessage;
import org.openspcoop.pdd.mdb.SbustamentoRisposteMessage;

/**
 * Fase di Inoltro del canale preferenziale
 * Oltre ad inviare il messaggio alla Porta Applicativa e
 * ricevere la risposta, si preoccupa di gestire la
 * transizione al canale tradizionale in caso di errori di
 * comunicazione.
 *
 * Le operazioni di transizione al canale
 * tradizionale per messaggi OneWay sono
 * - Salvataggio del messaggio nel repository
 * - Registrazione dello stato del messaggio su DB
 * - Aggiunta di InoltroBusteEGovMessage
 *   nella coda JMS InoltroBusteEGov
 * - Impostazione messaggio OK da inviare
 *   al mittente nel MediationContext
 * - Passare il controllo a RicezioneContenutiApplicativi
 *   (per l'inviio della risposta)
 *
 * Se qualcosa va male invia al mittente un
 * messaggio di errore e fa il rollback di tutto
 *
 * Se l'errore è nella conegna di un Sincrono mando indietro subito
 * il messaggio di errore senza transire nel canale tradizionale.
 *
 * In caso di errore il controllo ritorna a
 * RicezioneContenutiApplicativi.
 *
 * Se la comunicazione va a buon fine trasmette il
 * controllo al modulo successivo
 * per lo sbustamento, SbustamentoRisposte.
```

```
 *
 */

public class InoltroBusteEGov  {

private XMLBuilder xmlBuilder;
private MediationContext mc;

public MediationContext mediate(MediationContext _mc) {


this.xmlBuilder = new XMLBuilder();
this.mc = _mc;

/* ------------  Lettura parametri della richiesta  ------------- */

System.out.println("[Canale Preferenziale] [InoltroBusteEGov]
Ricezione richiesta...");

//          Recupero informazioni dal contesto di mediazione
InoltroBusteEGovMessage inoltroBusteEGovMsg =
mc.getInoltroBusteEGovMessage();
RichiestaDelegata richiestaDelegata =
      inoltroBusteEGovMsg.getRichiestaDelegata();
Busta bustaRichiesta = inoltroBusteEGovMsg.getBusta();
IDSoggetto identitaPdD =
      inoltroBusteEGovMsg.getRichiestaDelegata().getDominio();

// IDEGov associato alla richiesta
String idEGovRequest = mc.getIdEGovRichiesta();


/* Inizializzazione Contesto di gestione */

 ...

 ...

/* Mediazione  */

try{
      SOAPMessage requestSOAPMessage =
mc.getRicezioneContenutiApplicativiContext().getMessageRequest();

/* ------- Spedizione Messaggio Soap a PortaApplicativa -----------*/

/**
 * Connettore usa un Connection di SAAJ per l'invio del messaggio
 * Le api openspcoop forniscono più connettori.
 * Per prototipo usiamo solo .
 * Creo a mano il connettore  con destinazione fissa per test.
 */

String tipoConnector = "";
ConnettoreProperty locationInd = new ConnettoreProperty();
locationInd.setNome("location");
locationInd.setValore("://localhost:9000/openspcoop/PA");

Connettore connettore = new Connettore();
```

```
connettore.setTipo("");
connettore.addProperty(locationInd);
ConnettoreMsg connettoreMsg =
      new ConnettoreMsg(
            tipoConnector,
            requestSOAPMessage,
            connettore.getPropertyList());
connettoreMsg.setBusta(bustaRichiesta);
IConnettore connectorSender = null;

// Risposte del connettore
SOAPMessage responseSOAPMessage = null;
int codiceRitornato = -1;

// Stato consegna tramite connettore
boolean errorConsegna = false;
String motivoErroreConsegna = null;
boolean invokerNonSupportato = false;

// Ricerco connettore nel className.properties.
ClassNameProperties prop = ClassNameProperties.getInstance();
String connectorClass = prop.getConnettore(tipoConnector);

// Carico connettore richiesto (sempre  nel prototipo)
// Lascio questa modalità di
// istanziazione di connettore anche se è fissato .

if(invokerNonSupportato==false){
      try{
            Class<?> c = Class.forName(connectorClass);
            connectorSender = (IConnettore) c.newInstance();
      }catch(Exception e){
            presaInConsegnaFallita("Problemi con Connettore");
            return mc;
      }
}

// Utilizzo il connettore ed
// inoltro la richiesta alla porta applicativa

System.out.println("[Canale Preferenziale] [InoltroBusteEGov] Invio
Messaggio SPCoop a Porta Applicativa...");

errorConsegna = !connectorSender.send(connettoreMsg);
motivoErroreConsegna = connectorSender.getErrore();

//    interpretazione esito consegna
codiceRitornato = connectorSender.getCodiceTrasporto();
responseSOAPMessage = connectorSender.getResponse();

//Controllo la risposta
//Vediamo se c'è stato un errore

if(errorConsegna){
System.out.println("[Canale Preferenziale] [InoltroBusteEGov] Errore
in invio richiesta...["+codiceRitornato+"]");

//Consegna busta di PA fallita.
//Se è un ONEWay faccio presa in consegna.
```

```
if(bustaRichiesta.getProfiloDiCollaborazione().equals(Costanti.PROFILO
_COLLABORAZIONE_ONEWAY)){

      System.out.println("[Canale Preferenziale] [InoltroBusteEGov]
                        Profilo OneWay. Eseguo Presa in consegna");

      //Costruisco un messaggio di OK da mandare al SilMittente

      SOAPMessage okSoap =
            SoapUtils.buildOpenSPCoopOK_soapMsg(idEGovRequest);


      //Instanzio una connessione a Database

      DBManager dbManager = DBManager.getInstance();
      Connection connectionDB = null;

      try{
            // Connessione al DB
            connectionDB = dbManager.getConnection(
                  identitaPdD.getCodicePorta(),
                  org.openspcoop.pdd.mdb.Imbustamento.ID_MODULO);
      }catch(Exception e){
            if(connectionDB != null){
                  dbManager.releaseConnection(
                        identitaPdD.getCodicePorta(),
                        org.openspcoop.pdd.mdb.Imbustamento.ID_MODULO,
                        connectionDB);
            }
            presaInConsegnaFallita("Connessione a DB fallita");
            return mc;
      }

      //Salvo il messaggio di richiesta
      //su database per successivi tentativi di inoltro

      System.out.println("[Canale Preferenziale] [InoltroBusteEGov]
Registrazione messaggio di richiesta nel RepositoryMessaggi...");

      GestoreMessaggi msgRequest =
            new GestoreMessaggi(
                  connectionDB,
                  idEGovRequest,
                  Costanti.OUTBOX);
      try{
            msgRequest.registraMessaggio(okSoap);
            msgRequest.aggiornaProprietarioMessaggio(
                  org.openspcoop.pdd.mdb.Imbustamento.ID_MODULO);
      }
      catch(GestoreMessaggiException e){
      // Chiude le PreparedStatement aperte(e non eseguite) per il
      // salvataggio del Msg
      msgRequest.closePreparedStatement();

      // elimino richiesta salvata su fileSystem
      msgRequest.deleteMessageFromFileSystem();

      // Rilascio Connessione DB
```

```
	dbManager.releaseConnection(
		identitaPdD.getCodicePorta(),
		org.openspcoop.pdd.mdb.Imbustamento.ID_MODULO,
		connectionDB);
	//Mando messaggio di fault al mittente
	presaInConsegnaFallita("Impossibile salvare il messaggio");
	//Torno il controllo a RicezioneContenutiApplicativi
	return mc;
}

/*  Spedizione InoltroBusteEGovMessage alla  JMS InoltroBusteEGov del
canale tradizionale   */

// Creazione InoltroBusteEGovMessage

System.out.println("[Canale Preferenziale] [InoltroBusteEGov] Invio
messaggio al modulo di InoltroBusteEGov del Canale Tradizionale...");

String errorSenderJMS = null;
try{
	JMSSender senderJMS = new JMSSender(
		identitaPdD.getCodicePorta(),
		Imbustamento.ID_MODULO);

	if(senderJMS.send("InoltroBusteEGov",
			mc.getInoltroBusteEGovMessage(),
			idEGovRequest) == false){

	presaInConsegnaFallita("Errore scrittura coda JMS");
	return mc;
	}
} catch (Exception e) {
	presaInConsegnaFallita("Spedizione jms con errore");
	return mc;
}
if(errorSenderJMS!=null){
	dbManager.releaseConnection(identitaPdD.getCodicePorta(),
		Imbustamento.ID_MODULO,connectionDB);
	presaInConsegnaFallita("Errore in scrittura coda JMS");
	return mc;
}

System.out.println("[Canale Preferenziale] [InoltroBusteEGov] Profilo
OneWay. Invio OK al SIL Mittente");

mc.getRicezioneContenutiApplicativiContext().setMessageResponse(okSoap
);
}

else{
	//Errore di consegna con Profilo Sincrono
	//Costruisco il messaggio di errore
	//applicativo da inviare al SIL Mittente

System.out.println("[Canale Preferenziale] [InoltroBusteEGov] Profilo
Sincrono. Inoltro l'errore");
ProprietaErroreApplicativo erroreInvio =
	new ProprietaErroreApplicativo();
erroreInvio.setFaultActor("OpenSPCoop");
```

```
erroreInvio.setIdModulo("InoltroBusteEGov");
erroreInvio.setFaultAsGenericCode(true);
erroreInvio.setFaultAsXML(false);

SOAPMessage erroreSOAP =
this.xmlBuilder.msgErroreApplicativo_Processamento(
      erroreInvio,
      CostantiPdD.CODICE_500_ERRORE_INTERNO,
      CostantiPdD.MSG_5XX_SISTEMA_NON_DISPONIBILE);

mc.getRicezioneContenutiApplicativiContext().setMessageResponse(errore
SOAP);
return mc;
}
}

// Se arrivo qui significa che ho una risposta
// dalla Porta Applicativa. La mando allo Sbustamento

System.out.println("[Canale Preferenziale] [InoltroBusteEGov] Ricevuta
risposta...["+codiceRitornato+"]");

mc.getRicezioneContenutiApplicativiContext().setMessageResponse(respon
seSOAPMessage);

/*** -------------
 * Costruisco un'opportuna busta eGov di risposta
 * Nella realtà sarebbe contenuta dalla
 * risposta della Porta Applicativa
 *** ------------*/

boolean presenzaRispostaSPCoop = true;
Busta bustaRisposta = bustaRichiesta.invertiBusta();

bustaRisposta.setProfiloDiCollaborazione(bustaRichiesta.getProfiloDiCo
llaborazione());
bustaRisposta.setTipoServizio(bustaRichiesta.getTipoServizio());
bustaRisposta.setServizio(bustaRichiesta.getServizio());
bustaRisposta.setAzione(bustaRichiesta.getAzione());
bustaRisposta.setRiferimentoMessaggio(bustaRichiesta.getID());
mc.setIdEGovRisposta(
      org.openspcoop.egov.Imbustamento.buildID_eGov(
            null,
            bustaRisposta.getMittente()+"SPCoopIT",
            bustaRisposta.getMittente()));
bustaRisposta.setID(mc.getIdEGovRisposta());



/*  Gestione Risposta  (operazioni comuni per tutti i profili) */

SOAPFault fault = null;
boolean isMessaggioSPCoopErrore = false;
boolean bustaDiServizio = false;
String idEGovResponse = null;
java.util.Vector<Eccezione> errors = null;
SbustamentoRisposteMessage sbustamentoRisposteMSG = null;

// Gestione Specifica per Buste SPCoop
```

```
idEGovResponse = bustaRisposta.getID();
isMessaggioSPCoopErrore = false;
bustaDiServizio = false;

// Costruzione oggetto da spedire al modulo
// 'SbustamentoRisposte' anche se non fa nulla...
// servirà per applicazioni successive

sbustamentoRisposteMSG = new SbustamentoRisposteMessage();
sbustamentoRisposteMSG.setRichiestaDelegata(richiestaDelegata);
sbustamentoRisposteMSG.setBusta(bustaRisposta);
sbustamentoRisposteMSG.setErrors(errors);
sbustamentoRisposteMSG.setIsSPCoopErrore(isMessaggioSPCoopErrore);
sbustamentoRisposteMSG.setIsBustaDiServizio(bustaDiServizio);

/**
 * Passo alla fase successiva.. lo sbustamentoRisposte
 */

System.out.println("[Canale Preferenziale][InoltroBusteEGov]
Spedizione messaggio verso SbustamentoRisposte...");

mc.setSbustamentoRisposteMessage(sbustamentoRisposteMSG);

}
catch(Exception e){
      presaInConsegnaFallita(e.toString());
      return mc;
}
SbustamentoRisposte sbustamento = new SbustamentoRisposte();
return sbustamento.mediate(mc);
}

/* Funzione che setta il messaggio di errore di risposta */

public void presaInConsegnaFallita(String err){
      //Inotro al mittente un opportuno messaggio di fallimento.
      SOAPMessage soapErr = null;
      System.out.println(err);
      try{
            soapErr = SoapUtils.build_Soap_Fault(
                  err,
                  "OpenSPCoopPD"
                  "000");
      }
      catch(Exception e){}
      mc.getRicezioneContenutiApplicativiContext().setMessageResponse(
soapErr);

}
}
```

MediazioneFase 4: *SbustamentoRisposte.java*

```
package org.openspcoop.pdd.fastchannel;



/**
 * Il modulo sbustamentoRisposte deve
 * elaborare le informazioni dell'header eGov della risposta.
 * Ai fini del prototipo non non è interessante
 * eseguire questi controlli.
 *
 * Il prototipo in questa fase non fa niente e ritorna il
 * controllo a RicezioneContenutiApplicativi
 */

public class SbustamentoRisposte {

	public MediationContext mediate(MediationContext mc) {

	//    Ricezione Messaggio

	System.out.println("[Canale Preferenziale][SbustamentoRis]
Ricezione richiesta (SbustamentoRisposteMessage)...");

	return mc;


	}


}
```

Utility per cla costruzione di messaggi: *SoapUtils.java*

```
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPFactory;
import javax.xml.soap.SOAPFault;
import javax.xml.soap.SOAPMessage;
import org.openspcoop.utils.UtilsException;

/**
 * Classe di utilità per la costruzione di messaggi SOAP
 * OK o FAULT da inviare al mittente
 */


public class SoapUtils {

/**
 * Metodo per la costruzione di una busta di
 * risposta OK per gli ack OneWay
 * @param idEGov della richiesta
 * @return Messaggio SOAP OK
 * @throws UtilsException
 */

public static SOAPMessage buildOpenSPCoopOK_soapMsg(String idEGov)
throws UtilsException{
      try{
      MessageFactory mf = MessageFactory.newInstance();
      SOAPMessage responseSOAPMessage =
            (SOAPMessage) mf.createMessage();
      SOAPBody soapBody = responseSOAPMessage.getSOAPBody();
      soapBody.addChildElement(SoapUtils.buildOpenSPCoopOK(idEGov));
      return responseSOAPMessage;
      }
      catch(Exception e) {
            throw new UtilsException("Creazione MsgOpenSPCoopOK non
riuscito: "+e.getMessage(),e);
      }
}
/**
 * Metodo che si occupa di costruire un
 *messaggio SOAPElement 'openspcoop OK'.
 */

public static SOAPElement buildOpenSPCoopOK(String idEGov) throws
UtilsException{
      try{
      SOAPFactory sf = SOAPFactory.newInstance();
      SOAPElement ok = (SOAPElement)
      sf.createElement("PresaInConsegna");
      ok.setValue("ok");
      return ok;
} catch(Exception e) {
      throw new UtilsException("Creazione MsgOpenSPCoopOK non
      riuscito: "+e.getMessage(),e);
```

```
}
}

/**
 * Costruisce il messaggio di errore SOAP
 * da inviare al mittente se qualcosa fallisce.
 *
 * @param aFault breve descrizione
 * @param aActor chi ha fatto l'errore
 * @param aCode codice errore
 * @return Messaggio di fault da mandare al mittente
 * @throws UtilsException
 */
public static SOAPMessage build_Soap_Fault(String aFault, String
aActor, String aCode) throws UtilsException  {
      try{
            MessageFactory mf = MessageFactory.newInstance();
            SOAPMessage msg = (SOAPMessage) mf.createMessage();
            SOAPEnvelope env =
            (SOAPEnvelope) ((msg.getSOAPPart()).getEnvelope());

            SOAPBody bdy = (SOAPBody) env.getBody();
            bdy.addFault();
            SOAPFault fault = bdy.getFault();

            if(aFault != null)
                  fault.setFaultString(aFault);
            else
                  fault.setFaultString("");
                  if(aCode != null)
                  fault.setFaultCode(aCode);
                  if(aActor != null)
                  fault.setFaultActor(aActor);
                  return msg;
      } catch(Exception e) {
      throw new UtilsException("Creazione MsgSOAPFault non riuscito:
            "+e.getMessage(),e);
      }
}
}
```

Ringraziamenti

Sono molte le persone che devo ringraziare sia per questo lavoro, sia per questi anni universitari. È stata anche la loro presenza ed il loro sostegno che mi hanno permesso di raggiungere questo importante traguardo.

Ringrazio innanzi tutto i miei genitori, a cui dedico la tesi, grazie ai quali ho avuto la possibilità di coronare questo sogno.

Ringrazio i miei colleghi ed amici universitari, con i quali ho condiviso il lungo percorso che mi ha portato sin qui, aiutandomi ad affrontare i momenti difficili.

Ringrazio i ragazzi della Link.it, il Prof. Tito Flagella ed il Prof. Andrea Corradini, che mi hanno aiutato in questo lavoro con professionalità e competenza, riuscendo nel contempo a creare un clima sereno e amichevole.

Grazie a tutti di cuore,

Lorenzo

Bibliografia

[01] “Sistema Pubblico di Cooperazione: Architettura”,
Versione 1.0, 25 Novembre 2004,
http://www.cnipa.gov.it/site/_files/SPCoop-Architettura_v1.0_20041125_.pdf

[02] “Sistema pubblico di cooperazione: Busta di e-Gov”,
Versione 1.1, 14 Ottobre 2005,
http://www.cnipa.gov.it/site/_files/SPCoop-Busta%20e-Gov_v1.1_20051014.pdf

[03] “Sistema pubblico di cooperazione: Porta di Dominio”,
Versione 1.0, 14 Ottobre 2005,
http://www.cnipa.gov.it/site/_files/SPCoop-PortaDominio_v1.0_20051014.pdf

[04] “Sistema pubblico di cooperazione: Servizi di Registro”,
Versione 1.0, 14 Ottobre 2005,
http://www.cnipa.gov.it/site/_files/SPCoop-ServiziRegistro_v1.0_20051014.pdf

[05] “OpenSPCoop”
http://openspcoop.org

[06] “Link.it”
http://www.link.it

[07] “Apache Synapse”
http://ws.apache.org/synapse

[08] “Apache Axis”
http://ws.apache.org/axis

[09] “Apache Axis2”

http://ws.apache.org/axis2

[10] “Apache CXF”
http://incubator.apache.org/cxf/

[11] “Apache AXIOM”
http://ws.apache.org/commons/axiom

[12] “JBoss”
http://www.jboss.org

[13] “Simple Object Access Protocol”,
http://www.w3.org/TR/soap/

[14] “Web Service Interoperability”,
http://www.ws-i.org/

[15] “eXtensible Markup Language”,
http://www.w3.org/xml/

[16] “SOAP with Attachment”,
http://www.w3.org/TR/soap/

[17] “Web Service Description Language”,
http://www.w3.org/TR/wsdl

[18] “Universal Description, Discovery and Integration”
http://www.oasis-open.org/committees/uddi-spec/

[19] “Web Service Security”
http://www.oasis-open.org/committees/wss/

[20] “Web Service Adressing”
http://www.w3.org/Submission/ws-addressing

[21] “Web Service Reliability”
http://www.oasis-open.org/committees/wsrm/

[22] “Java API for XML Processing”
https://jaxp.dev.java.net/

[23] “Java API for XML-based Remote Procedure Call”
https://jax-rpc.dev.java.net/

[24] “Java API for XML-based Web Service”
https://jax-ws.dev.java.net/

[25] “SOAP with Attachment API for Java”
https://saaj.dev.java.net/

[26] “Java API for XML Registry”
http://java.sun.com/webservices/jaxr/index.jsp

[27] “Java API for XML Binding”
https://jaxb.dev.java.net/

[28] “Attachment Profile v1.0”
http://www.ws-i.org/Profiles/AttachmentsProfile-1.0.html

[29] “Simple Soap Binding Profile v1.0”
http://www.ws-i.org/Profiles/SimpleSoapBindingProfile-1.0.html

[30] “Web Service Architecture”
http://www.w3.org/TR/ws-arch/

[31] “SEDA: An Architecture for Scalable, Well Conditioned Internet Services”
Matt Welsh, David Culler, and Eric Brewer
http://www.eecs.harvard.edu/~mdw/talks/seda-sosp01-talk.pdf

[32] “SEDA: An Architecture for Scalable, Well Conditioned Internet Services”
Matt Welsh, David Culler, and Eric Brewer
http://www.eecs.harvard.edu/~mdw/papers/seda-sosp01.pdf

[33] “Service-Oriented Architecture and Web Service: Concepts, Technologies, and tools.”
Ed Ort
http://java.sun.com/developer/technicalArticles/WebServices/soa2/

[34] “Rethinking the Java SOAP Stack”
Steve Loughran, Edmund Smith
http://www.hpl.hp.com/techreports/2005/HPL-2005-83.pdf

[35] “What’s new in SOA and Web Service”
Ed Ort
http://java.sun.com/developer/technicalArticles/WebServices/soa2/WhatsNewArticle.html

[36] “Implementing High Performance Web Service Using JAX-WS 2.0”
Bharath Mundlapudi
http://java.sun.com/developer/technicalArticles/WebServices/high_performance/

[37] “Easy and Efficient XML Processing: Upgrade to JAXP 1.3”
Neeraj Bajaj
http://java.sun.com/developer/technicalArticles/xml/jaxp1-3/

[38] “Getting started with JAX-RPC”

Arun Gupta, Beth Stearns
http://java.sun.com/developer/technicalArticles/WebServices/getstartjaxrpc/

[39] “Java API for XML-based RPC (JAX-RPC)”
Rahul Sharma
http://java.sun.com/developer/technicalArticles/xml/jaxrpc/

[40] “Intro to the JAXM Client Programming Model”
Jennifer Rodoni
http://java.sun.com/developer/technicalArticles/xml/introJAXMclient

[41] “Java Tecnology and XML – Part 1 & 2”
Thierry Violleau
http://java.sun.com/developer/technicalArticles/xml/JavaTechandXML_part1
http://java.sun.com/developer/technicalArticles/xml/JavaTechandXML_part2

[42] “Registration and the JAXR API”
Ed Ort, Ramesh Mandava
http://java.sun.com/developer/technicalArticles/WebServices/WSPack/

[43] “Java Architecture for XML Binding (JAXB).”
Ed Ort, Bhakti Mehta
http://java.sun.com/developer/technicalArticles/WebServices/jaxb/

[44] “JAX-RPC versus JAX-WS”
Russel Butek, Nicholas Gallardo
http://www.ibm.com/developerworks/webservices/library/ws-tip-jaxwsrpc.html
http://www.ibm.com/developerworks/webservices/library/ws-tip-jaxwsrpc2.html

[45] “Introduction to StAX”
S. W. Eran Chinthaka
http://wso2.org/library/1844

[46] “Introducing AXIOM: The Axis Object Model”
S. W. Eran Chinthaka
http://wso2.org/library/26

[47] “Fast and Lighweight Object Model for XML”
S. W. Eran Chinthaka
http://wso2.org/library/291
http://wso2.org/library/351

[48] “Digging into Axis2: AXIOM”.
Dennis Sosnosky
http://www-128.ibm.com/developerworks/java/library/ws-java2/index.html

[49] “Apache Synapse ESB”
Ashankha Perera
wso2.org/library/244

[50] “Java Message Service”
Ruggero Russo
http://www.slideshare.net/federico.paparoni/tutorial-su-jms-java-message-service

[51] “The Architecture of Service”
Kendall Grant Clark
http://www.xml.com/pub/a/2003/05/28/deviant.html

[52] “A Tour of the Web Service Architecture”
Kendall Grant Clark
http ://www.xml.com/pub/a/2003/06/18/ws-arch.html

[53] “More WS-Arch discussion”

Steve Vinoski
http://blogs.iona.com/vinoski/archives/000107.html

[54] “WS-* and the Big Picture”
Steve Vinoski
http://blogs.iona.com/vinoski/archives/000104.html

[55] “WS-Architecture? Uhm…”
Steve Maine
http://hyperthink.net/blog/2004/10/15/WSArchitecture+Umm.aspx

[56] “Codehaus XFire”
http://xfire.codehaus.org/

[57] “IONA Celtix”
http://open.iona.com/

[58] “Aegis Binding”
http://xfire.codehaus.org/Aegis+Binding

[59] “Spring Framework”
http://www.springframework.org/

[60] “Progettazione di un framework open souce per la cooperazione applicative nella pubblica amministrazione”
Ruggero Barsacchi

[61] “OpenSPCoop: un’implementazione della Spwcifica di Cooperazione Applicativa per la Pubblica Amministrazione Italiana”
Andrea Poli

[62] “Oasis-Open”

http://www.oasis-open.org