Alma Mater Studiorum
Università degli Studi di Bologna

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

Tesi di Laurea in Reti di Calcolatori

Progetto di Integrazione tra ESB Standard

Candidato:
Matteo Collina

Relatore:
Chiar.mo Prof. Antonio Corradi

Correlatori:
Ing. Samuele Pasini
Ing. Stefano Monti

Anno Accademico 2008/2009 - Sessione I

Indice

Introduzione

Nella società moderna la cultura di massa è entrata sempre più a far parte della vita di ogni individuo, il quale, per propria passione o lavoro, è interessato alle più svariate fonti di informazione, la maggior parte delle quali è reperibile attraverso *Internet*. Negli ultimi anni, inoltre, sono stati resi disponibili una grande varietà di servizi usufruibili attraverso la rete, ed è così nata una nuova generazione di applicativi capace di sfruttarli al meglio per realizzare i loro particolari obiettivi. Questa rivoluzione ha investito anche i sistemi informativi aziendali, le fondamenta su cui ormai poggia l'intero business di molte società; tal sistemi, per essere efficaci, devono automatizzare i processi di business aziendali che rappresentano la principale leva tramite cui le società possono migliorare le proprie prestazioni.

Per supportare l'integrazione dei nuovi servizi e il cambiamento dei processi aziendali è stata sviluppata una nuova metodologia di progettazione per i sistemi informativi: la *Service Oriented Architecture* (SOA). Tale paradigma suddivide il sistema informativo in servizi di base, che vengono poi composti per ottenere i processi di business in maniera semplice e flessibile: i servizi possono essere interni o esterni all'azienda, di nuova o vecchia realizzazione, ma in ogni caso facilmente integrabili fra loro.

La *Service Oriented Architecture*, come potrebbe sembrare in questa breve introduzione, non prescrive il disordine, piuttosto suggerisce l'introduzione di un supporto capace di centralizzare la gestione dei servizi e di favorirne l'integrazione: gli *Enterprise Service Bus* (ESB), che si configurano come l'infrastruttura di elezione per la realizzazione della SOA all'interno di un'azienda. L'obiettivo degli ESB è la centralizzazione

del sistema informativo, cosa che però non è sempre possibile poiché, ad esempio, un'azienda potrebbe avere più succursali distribuite su di un vasto territorio.
Lo scopo di questo lavoro di tesi è perciò studiare le problematiche relative alla integrazione fra *Enterprise Service Bus* al fine di garantire una soluzione flessibile e di facile realizzazione. In particolare il primo capitolo propone una breve disamina dei principi della SOA e delle caratteristiche degli ESB, mentre il secondo analizza i modelli per l'integrazione di più ESB proposti in letteratura per giungere alla possibile architettura di una soluzione. Il capitolo 3 è interamente dedicato all'osservazione di uno standard per gli Enterprise Service Bus, mentre i capitoli 4 e 5 affrontano nel dettaglio il principale problema posto dall'integrazione. Infine nel capitolo 6 viene esposta la progettazione della soluzione e nel capitolo 7 vengono presentati alcuni risultati sperimentali.

Capitolo 1

Evoluzione dei Sistemi Informativi in Ambito Enterprise

Le aziende da sempre si sono poggiate su Sistemi Informativi, siano essi realizzati su carta o su potenti mainframe, per migliorare e portare a compimento il proprio business. Questi sistemi, per essere efficaci, devono riuscire nel difficile intento di automatizzare i processi di business aziendali nel modo più trasparente possibile. Dal momento che la modifica di questi ultimi è una delle principali leve attraverso cui un'azienda può migliorare le proprie performance, quest'ultima sarà portata ad adattare continuamente i processi e, in virtù del legame appena trattato, a continui adattamenti del sistema informativo. In quest'ottica risulta essere un fattore di primaria importanza per il successo di una tecnologia la capacità della stessa di supportare il continuo flusso di modifiche che il business impone, ovviamente mantenendo contenuti i costi [Jur06].

Agli anni '80 risalgono i primi sistemi informativi automatizzati, realizzati in *COBOL* [Oli06], che venivano eseguiti su potenti mainframe, le cui interfacce erano testuali e la manutenzione era difficile e complicata. Questa tipologia di applicativi fu progettata per lavorare in isolamento e perciò non era in grado di soddisfare le pressanti richieste del management: un processo di business spesso deve coinvolgere

più di un applicativo, ma i sistemi realizzati con questa tecnologia difficilmente si adattano a complesse integrazioni. Nonostante il *COBOL* sia un linguaggio di programmazione datato 1959 ancora oggi è alla base della maggior parte delle applicazioni per il business: la motivazione di questa reticenza all'innovazione riguarda sia il costo che il rischio del passaggio ad una tecnologia più moderna; infatti queste vetuste applicazioni sono degli asset vitali senza cui le compagnie non potrebbero operare.
Molte sono state le metodologie proposte al fine di integrare fra loro i vari applicativi che compongono il sistema informativo aziendale, ma solo negli ultimi anni si è giunti ad un approccio condiviso: la *Service Oriented Architecture* (SOA) [Jur06]. Questo paradigma prevede che ogni componente all'interno del sistema informativo esponga le proprie funzionalità, dette servizi, in modo che queste possano essere sfruttate da altri componenti software. La diffusione della SOA è dovuta anche alla facile integrazione, all'interno di un'architettura flessibile, tra sistemi legacy e nuove applicazioni, integrazione adatta ai continui cambiamenti a cui sono soggetti i sistemi software. In particolare, questo approccio consente di sviluppare nuove applicazioni non più secondo un approccio monolitico, ma attraverso il riuso di componenti e servizi già sviluppati e/o forniti da terzi, abbattendo quindi i costi di realizzazione [Bor04].

1.1 Principi per una Service Oriented Architecture

"La *Service Oriented Architecture* è un paradigma che sfrutta servizi debolmente accoppiati (loosely coupled) per garantire flessibilità al business, interoperabilità fra le applicazioni ed indipendenza rispetto alla tecnologia realizzativa degli stessi servizi" [Bor04]. Una SOA consiste in un insieme di servizi a grana grossa (coarse grained) che possono essere sfruttati per realizzare il software di supporto ai processi di business in modo flessibile, dinamico e riconfigurabile, usando

soltanto una particolare descrizione dei servizi, detta interfaccia. I più importanti concetti per una SOA sono: i servizi, le interfacce, i messaggi, la sincronia, l'accoppiamento debole, il registro dei servizi, la qualità di servizio e la composizione dei servizi [Jur06].

Servizi

I servizi devono fornire funzionalità importanti al business, nascondere i dettagli implementativi e, sopratutto, essere autonomi. Un esempio di servizio potrebbe essere un'applicazione per la prenotazione di biglietti aerei: il suo unico scopo è gestire le varie fasi della prenotazione, e perciò deve essere indipendente dai servizi aziendali.

Interfacce

I consumatori dei servizi accedono ad essi attraverso un'interfaccia: questa definisce un insieme di operazioni e stabilisce il contratto fra fornitore (provider) e consumatore (consumer). L'interfaccia è separata dall'implementazione, auto-descrivente e indipendente dalla piattaforma: sono queste semplici proprietà a renderla la base per l'implementazione sia del fornitore che del consumatore. Le operazioni, inoltre, dovrebbero essere idempotenti, criterio per il quale ripetute invocazioni della stessa operazione dovrebbero avere lo stesso effetto di una singola. Considerando sempre l'esempio precedente, si può considerare come l'interfaccia nasconda la realizzazione sottostante: il servizio di prenotazione di biglietti aerei, infatti, potrebbe essere fornito da un partner aziendale, da un tour operator, o addirittura risiedere fisicamente fuori dei confini dell'azienda stessa.

Messaggi

Le varie operazioni effettuabili sui servizi sono definite attraverso l'uso di appositi formalismi che stabiliscono il formato dei messaggi in modo indipendente dalla piattaforma e dal linguaggio usato per l'implementazione; contrariamente a quanto avviene in altri paradigmi, i messaggi contengono soltanto dati, non dati e comportamento. I messaggi

vengono utilizzati dal consumatore per trasmettere le informazioni necessarie ai servizi per operare, sia dai servizi per fornire il risultato: le comunicazioni fra consumatore e servizio avvengono attraverso un semplice scambio di messaggi.

Sincronia ed Asincronia

I consumatori devono poter accedere ai servizi in modo sia sincrono che asincrono. Nel primo caso un'operazione fornisce la risposta quando l'elaborazione è completata: il consumatore deve quindi attendere il completamento della stessa. Questo modo di operare viene utilizzato quando l'elaborazione è semplice e richiede un tempo limitato. Nel caso di operazione asincrona, al contrario, il fornitore non fornisce una risposta al consumatore, che quindi non deve attendere alcun risultato. Spesso però il fornitore, per dare maggiori garanzie sulla corretta ricezione, restituisce al consumatore una conferma. Nel caso invece in cui sia necessaria una risposta, il consumatore deve comunicare al fornitore il riferimento all'operazione che il fornitore dovrà usare per comunicargli il risultato.

Accoppiamento Debole

I servizi debolmente accoppiati sono ottenuti utilizzando tutti i concetti esaminati fino ad ora ed espongono, quindi, solo le dipendenze strettamente necessarie, così da limitare il numero di modifiche da apportare ad un servizio al cambiamento di un altro. Questo approccio da un lato promuove il riuso dei servizi e dall'altro favorisce la costruzione di sistemi robusti anche a fronte di continue modifiche.

Registri di Servizi

Per semplificare la fruizione dei servizi, questi sono iscritti in appositi registri dai fornitori. I consumatori, quindi, possono effettuare ricerche nei registri per nome, per funzionalità oppure per altre proprietà legate ai processi di business.

Qualità del Servizio

In un'architettura enterprise non è importante solo accedere alle operazioni messe a disposizione dai servizi, ma anche il modo in cui questo avviene, detto anche *Quality of Service* (QoS): ne sono un esempio la garanzia di avvenuta ricezione e la sicurezza. All'interno di una SOA deve poter essere garantito che:

- le operazioni vengano effettuate solo dagli aventi diritto (security);
- i messaggi arrivino correttamente (reliable messaging);
- le operazioni siano transazionali (transactions);
- i messaggi siano correttamente correlati fra loro quando si utilizza un meccanismo di comunicazione asincrono (correlation);
- i fornitori possano specificare la QoS dei servizi da loro forniti, mentre i consumatori devono poter specificare la QoS da loro richiesta;
- siano rispettate tutte le altre esigenze dei servizi, troppo tecniche per essere affrontate in questa sede.

Composizione di Servizi

L'ultimo ed il più importante concetto relativo alla SOA è la composizione di servizi in processi di business e ne rappresenta il punto di arrivo, permettendo così di ottenere tutti i benefici di questa architettura. Sarà infatti possibile modificare i processi di business senza intaccare i servizi, in questo modo riducendo sia i rischi che i costi.

1.2 I Web Services

I *Web Services* (WS) sono una moderna tecnologia per la realizzazione di soluzioni di integrazione fra sistemi software e, come si andrà ad analizzare, la più opportuna per la realizzazione della SOA [Jur06]. Infatti,

i WS pongono le fondamenta tecnologiche per ottenere l'interoperabilità fra diverse applicazioni indipendentemente dalla piattaforma realizzativa utilizzata. I WS sono stati realizzati basandosi sull'*eXtensible Markup Language* (XML) [W3C08], il linguaggio standard per l'integrazione a livello di dati, ottenendo così lo standard per l'integrazione fra le applicazioni. XML permette agli sviluppatori di specificare in modo rigoroso la struttura di un file, rendendo così semplice realizzare più applicazioni capaci di interpretare un particolare formato.

Il successo dei Web Services è da ricercasi nel grande consenso ottenuto fra tutti i maggiori produttori di software: è infatti la prima tecnologia che riesce ad adempiere alla promessa di interoperabilità universale, fatta da ogni tecnologia, che consente a due sistemi software di comunicare. Grazie all'utilizzo di XML è possibile specificare (e verificare in qualunque momento) il formato di tutti i messaggi scambiati in modo completamente indipendente dalle tecnologie realizzative; in questo modo si riduce ogni possibile accoppiamento fra produttore e consumatore ai soli dati, così come prescritto dalla SOA. In ultimo è importante rilevare come i WS siano indipendenti dai protocolli usati per accedervi [W3C07a]: di norma sono resi disponibili attraverso i protocolli standard di Internet per le comunicazioni, come, ad esempio, HTTP (web), SMTP (posta elettronica) ed FTP (dati). Da ultimo si nota come i WS risolvano l'annoso problema legato all'implementazione delle tecnologie precedenti, le quali mostravano i loro limiti nella difficoltà ad attraversare firewall e ad essere impiegate, perciò, nella grande rete.

1.2.1 Pubblicazione di un Servizio

Un servizio, così come inteso dai WS, è specificato secondo il *Web Services Description Language* (WSDL) [W3C07a] attraverso l'indicazione di una parte astratta e di una concreta (la versione del WSDL qui trattata è la 2.0, mentre verranno indicati fra parentesi i nomi che gli stessi componenti hanno nella versione 1.1). La prima specifica in maniera

generica come è definito il servizio e come sono formati i dati scambiati, mentre la seconda il protocollo e l'URI a cui è disponibile il servizio.

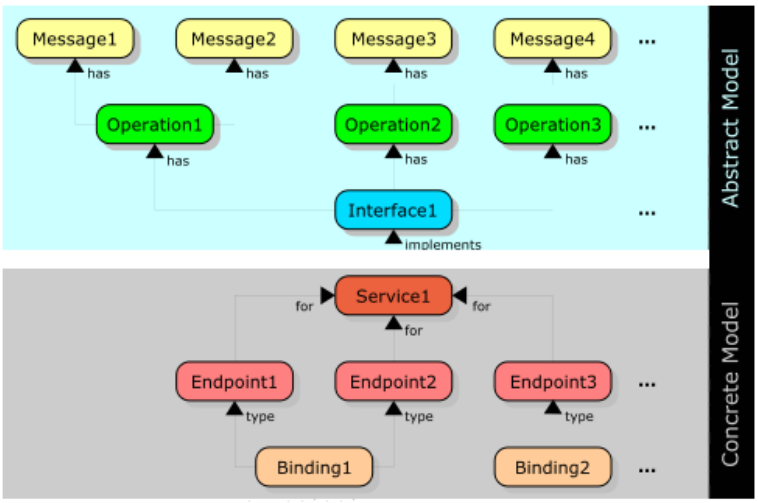


Figura 1.1: Esempio di un file WSDL.

Nella parte astratta, il WSDL (vedasi figura 1.1 [Ten05]) definisce una *operation* come uno scambio di messaggi fra produttore e consumatore, mentre è detta *interface* (*portType* nella versione 1.1) un insieme di operazioni. Nella parte concreta viene invece fornito il *binding*, che rappresenta la realizzazione di un'interfaccia attraverso un determinato protocollo: specifica cioè il modo in cui un messaggio debba essere inviato, ad esempio attraverso lo standard SOAP. Infine un *service* è una collezione di *endpoint* (*port* nella versione 1.1) che specificano l'URI [IET05] a cui sono disponibili i vari *binding*.

Al fine di identificare correttamente i servizi, alla *interface*, al *service* e all'*endpoint* sono assegnati dei nomi univoci all'interno dello stesso WSDL e, per estensione, all'intero sistema SOA in cui andranno ad operare. Ognuno di questi nomi viene fornito come *Qualified Name* (QName), una stringa composta da una prefisso e da una parte locale. Più precisamente il prefisso identifica il *namespace* XML (tipicamente un URL) all'interno del quale sono definite tutte le parti locali che seguiranno [W3C06].

1.2.2 Diffusione dei Web Services e del Paradigma SOA

I *Web Services*, come è già stato accennato sopra, sono la tecnologia di elezione per realizzare una SOA. È importante notare come l'adozione dei WS non sia condizione sufficiente per affermare la completa realizzazione del paradigma architetturale in esame. Infatti, se, riprendendo l'esempio precedentemente esaminato, si utilizzasse un servizio per prenotare un biglietto aereo, avremmo una realizzazione corretta, ma se, al contrario, si realizzasse un servizio per aggiungere, ad esempio, un record ad un database, allora sarebbe sbagliata. È infatti cruciale tenere sempre a mente che la SOA nasce per supportare processi di business a grana grossa.
I *Web Services*, inoltre, sono anche corredati da ulteriori standard per gestire, sempre tramite XML, tutti i principi esposti precedentemente (sez. 1.1). È quindi possibile, attraverso un insieme di estensioni identificabili nel loro insieme come *WS-**, invocare un servizio in modo transazionale, sicuro e specificando altri eventuali parametri per la QoS. Infine è opportuno menzionare un ultimo aspetto importante che riguarda la componibilità dei servizi stessi: il modello espresso nel WSDL è di per sé insufficiente a descrivere i processi di business, si limita solo a modellare un'interazione produttore-consumatore. Al fine di rappresentare correttamente i processi è stato introdotto, come standard, il linguaggio *Business Process Execution Language for Web Services* (BPEL), tramite cui è possibile modellare tutta la complessità dei processi di business in modo dichiarativo, flessibile e facilmente modificabile. Questo linguaggio è basato a sua volta su XML ed è perciò completamente indipendente dall'ambiente su cui andrà ad operare [OAS07].

1.3 Gli Enterprise Service Bus

La realizzazione della SOA basata sui WS porta alla creazione di molte comunicazioni punto-a-punto, rendendo spesso l'intera infrastruttura difficile da manutenere a fronte di cambiamenti nei servizi stessi. Infat-

ti, in questo modello, se cambia anche solo il protocollo per accedere ad un servizio è necessario modificare tutti i componenti che dipendono da quel servizio. Per questo motivo, più un'organizzazione abbraccia il paradigma SOA più sentirà la necessità di un'infrastruttura che, da un lato, renda uniforme l'accesso ai servizi, e, dall'altro, possa essere impiegata per utilizzi più sofisticati dei servizi stessi.

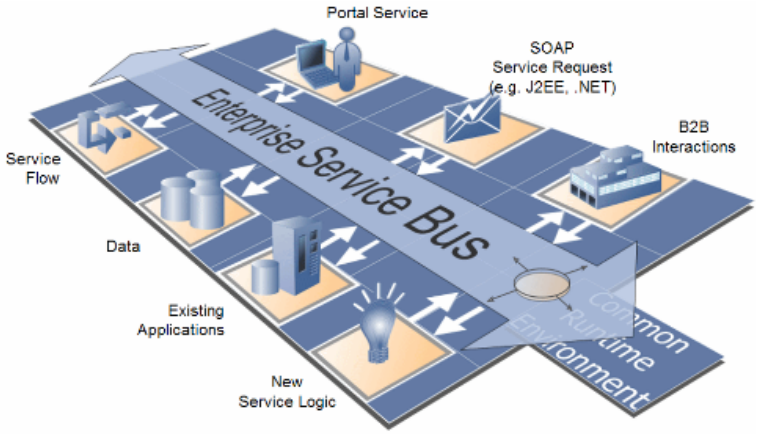


Figura 1.2: Servizi collegati ad un *Enterprise Service Bus*.

L'infrastruttura sopra delineata viene chiamata *Enterprise Service Bus* (ESB) ed, infatti, è un vero e proprio bus di comunicazione fra i vari servizi (fig. 1.2 [Kee04]), più correttamente è un *middleware* capace di integrare fra loro i servizi in una SOA. Infatti gli ESB vengono utilizzati per connettere le più disparate applicazioni e risorse, mediando le loro incompatibilità, orchestrando la loro interazione e rendendole utilizzabili per scopi futuri. L'ESB svolge quindi un ruolo di primaria importanza all'interno della SOA, poiché agisce sia come un registro di servizi sia come un unico punto centralizzato di gestione [Kee04]. In particolare questa soluzione consente di gestire in modo unico problematiche quali il clustering e la fault tolerance.
Gli *Enterprise Service Bus* hanno inoltre il grande compito di uniformare l'accesso ai servizi, in particolare soluzioni middleware pre-esistenti e sistemi legacy: gli ESB, infatti, rendono accessibili tutti gli applicativi in modo assolutamente omogeneo e coerente con il modello basato sui WS. Mediante l'introduzione di un ESB, tutte le comunicazioni fra

i servizi vengono effettuate attraverso di esso in modo assolutamente trasparente agli stessi servizi: è addirittura possibile trasformare i messaggi prima che questi vengano effettivamente consegnati, consentendo una normalizzazione utile durante l'intera esecuzione dei processi di business. Questa caratteristica è di primaria importanza, poiché in questo modo è possibile evitare la propagazione dal produttore al consumatore di eventuali modifiche: ad esempio, se cambiasse il formato dei messaggi in ingresso ad un servizio, basterebbe introdurre, all'interno dell'ESB, una trasformazione dal vecchio al nuovo formato.

1.3.1 Caratteristiche di un Enterprise Service Bus

La definizione di ESB non è univoca: al contrario, è possibile osservare la presenza sul mercato di molteplici prodotti che si autodefiniscono ESB, ognuno con caratteristiche differenti. È quindi importante delineare in questa sede l'insieme minimo delle caratteristiche che identificano un ESB in quanto tale, e suddividere ulteriormente tale insieme in quattro gruppi di funzionalità: comunicazione, integrazione, interazione con i servizi e management. Oltre alle funzionalità fondamentali che verranno delineate nel proseguio, ogni realizzazione aggiunge delle proprie peculiarità che la caratterizzano ma che, purtroppo, risultano al tempo stesso molto tecniche e non adatte ad essere escusse in questa sede.

Comunicazione

Il primo insieme di funzionalità mira a garantire la trasparenza alla locazione, garantire cioè che i servizi siano accessibili indipendentemente dalla loro locazione fisica. Dopo aver pubblicato un servizio all'interno di un ESB, questo è reso disponibile a tutti gli altri servizi ivi contenuti senza che sappiano la reale posizione del servizio. Questo requisito è essenziale per garantire la manutenibilità dei servizi: se la locazione di uno di questi cambia è necessario modificare soltanto l'ESB e non tutti i suoi consumatori.
Per garantire una comunicazione corretta fra i servizi l'ESB deve poter

effettuare un corretto instradamento dei messaggi scambiati fra di essi: come si avrà modo di vedere successivamente, il sottosistema che prende in carico questo compito è il *Normalized Message Router* (NMR). È importante notare che i messaggi scambiati sono normalizzati, ovvero che si basano tutti sul medesimo formato. L'instradamento non è però possibile senza un valido schema di indirizzamento che, stando al modello basato sui *Web Services*, non può che essere basato sui nomi di interfaccia, di servizio e di endpoint (vedasi 1.2.1).

Integrazione

Il secondo insieme di funzionalità è necessario per garantire la comunicazione con sistemi legacy: l'ESB deve infatti rendere accessibile l'intera gamma di servizi disponibili attraverso i più svariati protocolli. Normalmente questo requisito viene soddisfatto realizzando molteplici adattatori capaci di tradurre i messaggi scambiati all'interno dell'ESB nei relativi protocolli di accesso e viceversa. È attraverso questa importante funzionalità che viene realizzata la normalizzazione richiesta per instradare correttamente i messaggi all'interno dell'ESB.

Interazione con i Servizi

Un ESB deve necessariamente garantire il rispetto dei principi SOA, in particolare la separazione dell'interfaccia dall'implementazione. L'ESB, per effettuare l'instradamento, deve conoscere l'interfaccia e, dal momento che un ESB adotta il modello dei WS (vedasi 1.2.1), deve consentire ad ogni servizio di specificare la propria interfaccia attraverso un file WSDL.

Management

L'amministrazione è l'ultima e non meno importante funzionalità che ogni ESB deve fornire. Infatti, è necessario avere un punto di accesso che consenta l'analisi dei servizi pubblicati al fine di monitorare sia il loro stato come unità singole sia il flusso di messaggi che vengono fra loro scambiati.

1.3.2 Il Modello a Componenti

A questo punto della trattazione è importante introdurre il concetto di componente perché verrà usato estensivamente nel seguito: per i principi SOA, infatti, i servizi sono normalmente esterni all'ESB e per accedervi è necessario l'utilizzo di un protocollo specifico ma, poiché molti servizi condividono i medesimi protocolli, è possibile ridurre a comun denominatore tutta la logica di accesso ai servizi in ciò che è chiamato componente. Per rendere disponibile un servizio all'interno di un *Enterprise Service Bus* è quindi sufficiente istruire un componente specifico su come rintracciare il servizio stesso: infatti l'unità minima allocabile all'interno di un ESB non è il singolo servizio, ma, piuttosto, il componente che fa da contenitore per tutti i servizi accessibili tramite il medesimo protocollo.

1.4 Integrazione fra Enterprise Service Bus

L'introduzione di un ESB in un'infrastruttura SOA basata sui WS è vincolata al rispetto di un unico requisito: l'unicità. Infatti, per sua stessa natura, l'ESB aziendale si colloca come l'infrastruttura di elezione su cui si andranno ad eseguire tutti i processi di business dell'azienda stessa. Purtroppo questo requisito, dettato da problemi di natura tecnologica, non è sempre facilmente realizzabile: come si vedrà più nel dettaglio nel capitolo 2, sussistono importanti motivazioni che spingono a ritenere che la via dell'unicità non risulti sempre percorribile [Kee05].
Ad esempio, una banca può avere svariate filiali distribuite su un determinato territorio che devono necessariamente comunicare sia fra loro sia con la sede centrale. Una filiale potrebbe anche essere fisicamente distante dalla sede centrale, potenzialmente collocata in un'altra città o addirittura in un altro stato, e tuttavia non si vuole dipendere dalla presenza di connettività per operare sui conti correnti dei propri clienti. Infatti, se si adottasse un unico ESB per tutte le filiali, ogni processo di business dovrebbe raggiungere l'ESB collocato nella sede centrale. Se, per qualunque ragione, la connettività venisse a mancare in una filia-

le, questa si troverebbe isolata e non più in grado di operare. Inoltre, vista la grande distanza, potrebbero esserci dei problemi di congestione della rete tali da rallentare notevolmente il lavoro della filiale. Per risolvere questo problema è possibile adottare, come si può vedere in figura 1.3, una soluzione che preveda molteplici ESB, condivisi anche tra più filiali.

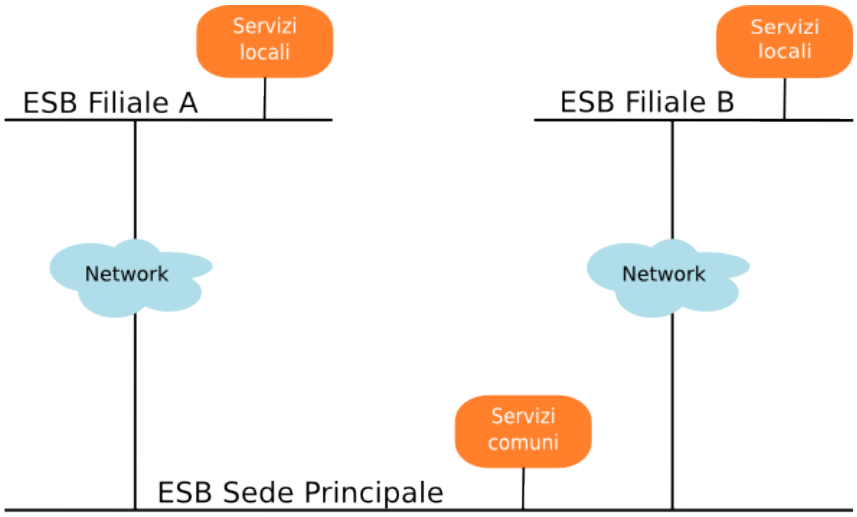


Figura 1.3: Esempio di integrazione di più ESB federati fra loro.

Scopo di questo lavoro di tesi è esplorare come sia possibile superare le limitazioni tecnologiche imposte dal requisito dell'unicità; si vuole cioè indagare le modalità con cui sia possibile far dialogare più ESB fra loro. L'obiettivo è, quindi, analizzare i principali problemi tecnologici che vengono imposti dal rilassamento di tale requisito, per poi evidenziare come risolverli in maniera efficiente, mantenendo inalterata la qualità di servizio richiesta ed erogata dai servizi stessi.
In sostanza si vuole discutere di come sia possibile organizzare più ESB in modo che ne rimangano inalterate tutte le proprietà fondamentali: si vuole cioè che un gruppo di ESB operi all'unisono. Verranno quindi esaminati non soltanto i principali problemi realizzativi, ma anche i vincoli imposti dai principali casi d'uso. Come si vedrà nel dettaglio nel capitolo successivo, esistono delle notevoli implicazioni realizzative dovute alle motivazioni che portano all'integrazione di più ESB fra

loro. Basandosi su questi requisiti verrà poi affrontato nel dettaglio un possibile progetto per l'integrazione che, da un lato, permetta di essere impiegato in tutti gli scenari che verranno delineati e, dall'altro, garantisca robustezza a fronte dei requisiti che verranno posti nel prossimo futuro.

Capitolo 2

Modelli per l'Integrazione fra Enterprise Service Bus

La letteratura consiglia l'adozione di un solo *Enterprise Service Bus* all'interno di una stessa organizzazione ed arriva addirittura a definirla una best practice [Kee05]. Dal momento che lo scopo di un ESB è quello di facilitare l'integrazione tra i servizi, utilizzarne più di uno introduce un ulteriore livello di complessità, consistente proprio nell'integrazione dei cosiddetti "integratori".
Non sempre, però, è possibile seguire quella che è una best practice dal punto di vista tecnico, potendoci essere numerose motivazioni organizzative tali da far preferire la presenza di più ESB che collaborano fra loro:

- All'interno di una stessa organizzazione potrebbero esserci più organismi di controllo indipendenti fra loro e potrebbe perciò essere impraticabile, per motivazioni organizzative, progettare e realizzare una soluzione comune. Ciascun organismo dovrà quindi definire quali sono i servizi accessibili all'esterno in modo programmatico per poter poi integrare le soluzioni autonome.

- La dislocazione geografica dell'azienda può rendere impraticabile l'utilizzo di un singolo ESB attraverso differenti sedi aziendali, specie in presenza di collegamenti non affidabili. Potrebbe risul-

tare più semplice dotare ogni sede di un proprio ESB, integrandoli fra loro in maniera da condividere soltanto quanto necessario.

- Spesso le società di grandi dimensioni sono il frutto di fusioni di più imprese fra loro, ed è quindi possibile che siano già presenti delle soluzioni ESB per ognuna di esse. Analogamente è possibile che organizzazioni multinazionali debbano sottostare a differenti legislazioni per quanto riguarda l'infrastruttura IT, così che la realizzazione di una soluzione univoca potrebbe comportare alcuni problemi. Ad esempio, in Italia il nome dell'amministratore di sistema, colui che è responsabile del trattamento dei dati all'interno del Sistema Informativo, deve essere depositato presso la camera di commercio.

2.1 Direttrici per un Progetto di Integrazione

L'integrazione di più ESB all'interno di una stessa organizzazione è un compito complesso poiché non è soltanto un problema tecnologico ma, come è stato appena esaminato, riflette anche la struttura stessa dell'organizzazione su cui è difficile intervenire. Per questo motivo sono state individuate in letteratura tre principali coordinate di analisi su cui basarsi per strutturare un progetto d'integrazione: la topologia, l'amministrazione e l'adattamento [Kee05].

2.1.1 Modelli Topologici

È possibile avere differenti topologie all'interno di un'organizzazione: collegamento diretto, collegamento indiretto, e esb federati.

Collegamento Diretto

In questo caso i service provider di un ESB sono mappati direttamente su un altro ESB tramite una connessione punto-a-punto (fig. 2.1). È pertanto richiesto che gli ESB siano a conoscenza di alcuni dettagli

implementativi (ad esempio quale ESB offre un servizio, come raggiungerlo, e in che formato inoltrare la richiesta). Questo approccio porta ad avere delle regole di instradamento dei messaggi (routing) frammentate attraverso molteplici ESB; è pertanto difficile amministrare correttamente un sistema di dimensioni considerevoli basato su questa topologia: il numero di collegamenti fra ESB può crescere esponenzialmente col numero di ESB che vengono integrati. Dal momento che non vi è un'infrastruttura comune, ciascun ESB mantiene la propria lista di servizi, la propria amministrazione e i propri servizi per la sicurezza, ed è compito di ogni collegamento coordinare e tradurre queste informazioni al passare delle richieste.

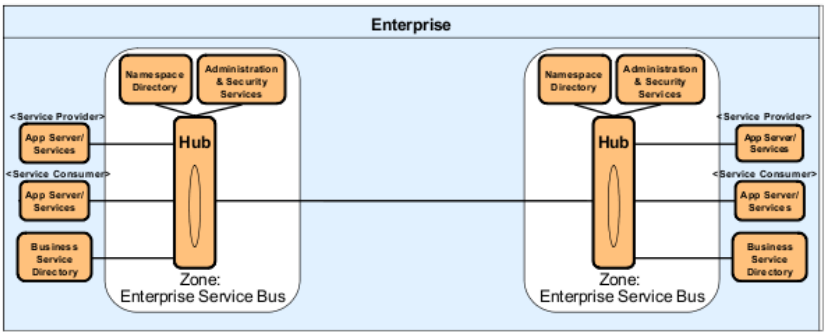


Figura 2.1: Esempio del modello topologico a collegamento diretto.

Collegamento Indiretto

Questa topologia si basa sull'introduzione di un ESB gateway (o hub) che connette ed integra fra loro gli ESB (fig. 2.2 [Kee05]). Questo componente fa in modo che gli ESB non si conoscano fra loro e centralizza il controllo delle connessioni fra gli ESB, riducendo, rispetto al collegamento diretto, il numero di connessioni punto-a-punto. Una nota importante è che, per questo modello, il gateway non ha né *service provider* né *service consumer*, ma contiene soltanto la logica di integrazione. Lo si può definire, in sostanza, un super-ESB dedicato alla sola funzione di instradamento fra gli altri ESB.

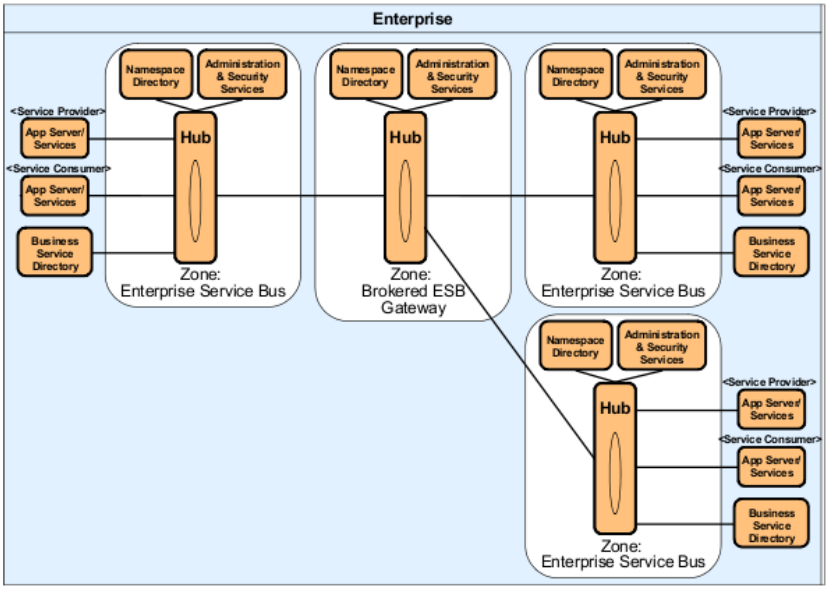


Figura 2.2: Esempio del modello topologico a collegamento indiretto.

ESB Federati

Questo modello rappresenta un'estensione del collegamento indiretto nel quale l'hub diventa un ESB vero e proprio a cui possono essere collegati dei *service consumer* (fig. 2.3 [Kee05]). In particolare, questo approccio consente di gestire al meglio il caso in cui molti service consumer utilizzino service provider che appartengono a ESB diversi, dal momento che è possibile collocare il service consumer direttamente sull'hub, oppure pubblicare su di esso un servizio di orchestrazione. In questo modo l'architettura è notevolmente semplificata e riduce sia il traffico fra gli ESB sia, di conseguenza, il numero di connessioni richieste.

2.1.2 Modelli per l'Amministrazione

Una zona amministrativa contiene tutti i componenti gestiti da un unico organismo di controllo, che però non ha alcun potere al di fuori della zona stessa. Due zone si dicono accoppiate quando una zona è a conoscenza di come un'altra zona realizza un servizio: questo tipo di accoppiamento richiede nuovi livelli di coordinazione che possono

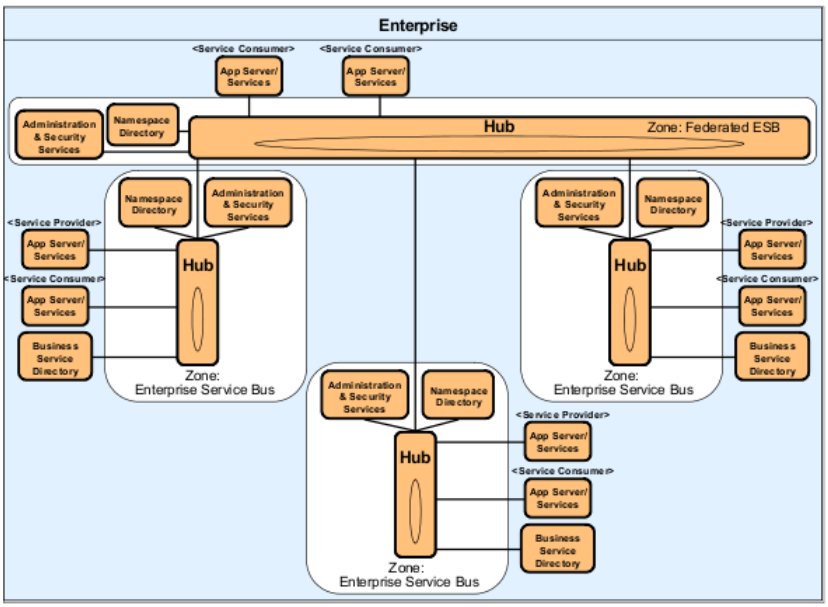


Figura 2.3: Esempio del modello topologico di ESB federati.

potenzialmente inibire o rallentare la possibilità di effettuare cambiamenti. Mediare l'accoppiamento fra le zone amministrative attraverso l'uso degli ESB abbassa notevolmente il grado di accoppiamento fra le stesse, poiché in questo modo si uniforma l'accesso a tutti i servizi condivisi fra le due zone.
Le possibili modalità di interazione fra diverse zone amministrative sono: locale, intermediata, e federata.

Modalità di Interazione Locale

Questa modalità di interazione si sceglie quando due zone necessitano di un servizio in comune ed è al tempo stesso necessario che vi sia conoscenza diretta della prima zona da parte della seconda, ovverosia l'ESB consumatore deve sapere su quale ESB è collocato il servizio fornitore (fig. 2.4 [Kee05]).

Modalità di Interazione Intermediata

In questo caso non si vuole introdurre una dipendenza fra le due zone e, perciò, il servizio necessario viene collocato in un ESB intermediario

Figura 2.4: Esempio del modello amministrativo ad interazione locale.

esterno ad entrambe (fig. 2.5 [Kee05]). Così facendo l'accoppiamento fra le due zone è notevolmente ridotto, e si mantiene tutta la logica di routing nello stesso luogo.

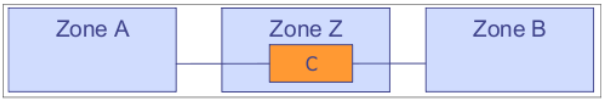


Figura 2.5: Esempio del modello amministrativo ad interazione intermediata.

Modalità di Interazione Federata

In questo modalità, diversamente dalle precedenti, non si ha una dipendenza diretta di un ESB nei confronti di un altro, ma due servizi, uno per ESB, che collaborano tra loro per portare a termine i propri compiti (fig. 2.6 [Kee05]). Questa modalità porta al più stretto accoppiamento fra i due ESB, essendo importante che lo scambio di messaggi fra i pari sia sicuro.

Figura 2.6: Esempio del modello amministrativo ad interazione federata.

2.1.3 Modelli per l'Adattamento del Contesto

Nel momento in cui si integrano due o più ESB non è difficile inoltrare le chiamate ai servizi remoti, essendo basate sugli standard per i WS, mentre può essere più complesso mantenere il contesto (ad esempio:

logging, QoS, sicurezza, ecc.) semanticamente equivalente durante il passaggio. In particolare la traslazione del contesto fra due o più ESB può essere statica o dinamica.

Traslazione Statica

In questo modello la logica di traduzione è inserita all'interno del componente per l'integrazione, il connettore, e perciò determinata in fase di sviluppo (fig. 2.7 [Kee05]). Dal momento che le operazioni effettuabili da questo componente sono fissate a priori, nel momento in cui dovessero sorgere nuove esigenze di integrazione sarà necessario modificare il connettore. Questo tipo di componente può essere sviluppato velocemente, ma, purtroppo, è difficilmente manutenibile poiché, mano a mano che vengono introdotte modifiche per supportare le nuove esigenze, il codice del connettore diviene sempre più complesso e difficile da adattare.

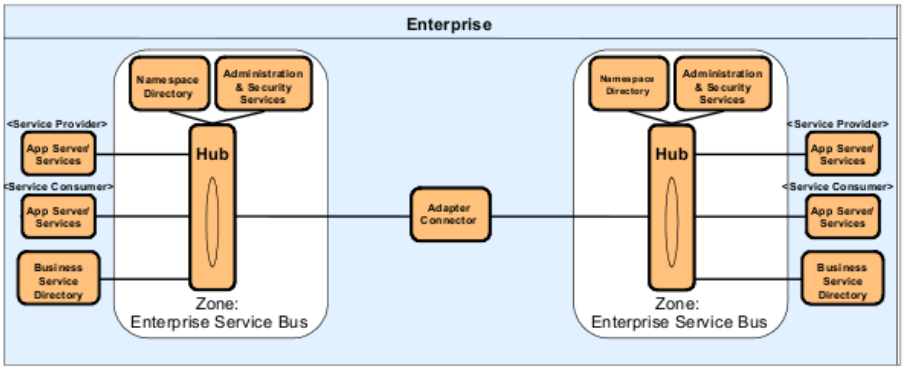


Figura 2.7: Esempio del modello statico per l'adattamento del contesto.

Traslazione Dinamica

In questo caso la logica di traslazione è suddivisa fra molteplici *Boundary Services* (BS), ognuno dei quali si occupa di gestire un particolare aspetto del contesto secondo le best practices della SOA: in particolare devono nascondere i loro dettagli implementativi e non fare assunzioni riguardanti il contesto e lo stato della chiamata.

Il connettore, per determinare quali *Boundary Services* dovrà usare per tradurre dinamicamente il contesto locale nel linguaggio usato per la comunicazione remota (fig. 2.8 [Kee05]), utilizza i metadati contenuti nella chiamata locale. I metadati devono avere una semantica predeterminata e rappresentano il contratto di servizio fra gli ESB; è importante rilevare che possono essere espressi anche in formati differenti. La semantica contenuta nei metadati descrive le responsabilità che competono all'ESB remoto nel momento in cui riceve la chiamata dall'ESB locale.

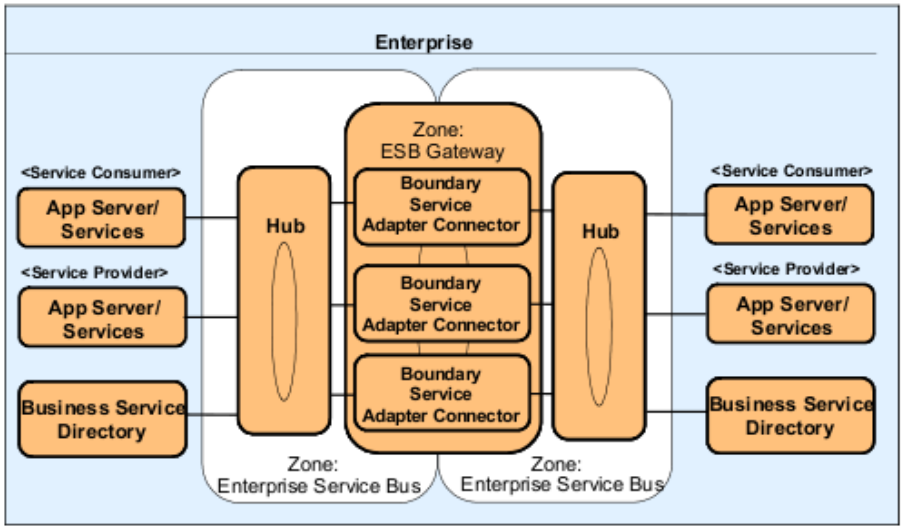


Figura 2.8: Esempio del modello dinamico per l'adattamento del contesto.

I metadati del contesto possono cambiare da una chiamata all'altra anche se consumatore e produttore rimangono immutati, e l'approccio a BS riesce a gestire questi cambiamenti. In particolare ciò è reso possibile grazie ad una selezione dinamica dei BS attivi per una determinata chiamata: questo insieme viene determinato analizzando il contesto, ed è perciò estremamente variabile.
Al fine di garantire il corretto funzionamento del meccanismo di selezione, ciascun gruppo di *Boundary Service* si deve occupare del trasferimento di un determinato insieme di funzionalità. È altresì importante che i BS seguano i principi dettati dalla SOA: in particolare devono, da

un lato, nascondere i dettagli implementativi grazie all'utilizzo di interfacce e, dall'altro, non effettuare assunzioni di alcun tipo sul contenuto del contesto corrente e sullo stato della chiamata.
Questo approccio garantisce un'elevata flessibilità dal momento che i *Boundary Services* all'interno del connettore possono essere sviluppati in un secondo tempo ed essere combinati, perciò, in modi per i quali non erano stati progettati.

2.2 Il Componente Integratore

La principale responsabilità dei componenti, come ampiamente illustrato nella sezione 1.3, è presentare all'ESB un modello univoco di servizi: nel caso si volesse realizzare un componente capace di collegare due o più ESB questo compito risulta ampiamente semplificato. Infatti le interfacce dei servizi presenti sugli ESB devono essere espresse attraverso il WSDL, e perciò le uniche mediazioni richieste al componente consistono in:

- istruire l'ESB della presenza dei servizi "importati" da altri ESB;
- trasmettere le richieste locali ai servizi "importati" verso l'opportuno ESB remoto;
- instradare le richieste provenienti da altri ESB sull'ESB locale.

Purtroppo, però, alla luce di quanto precedentemente esaminato, queste poche responsabilità diventano assai gravose. Infatti, al fine di garantire un'adeguata flessibilità, il componente per l'integrazione deve poter gestire tutti i modelli topologici precedentemente indicati e supportare i modelli amministrativi ad interazione locale e intermediata. Il modello amministrativo ad interazione federata non viene considerato, in quanto porta ad un accoppiamento troppo stretto fra le parti e quindi contrario ai principi SOA. Infine, dal momento che si ricerca una soluzione generale, si ritiene opportuno adottare il modello dinamico per l'adattamento del contesto.

2.2.1 Struttura del Componente

Per realizzare un componente integratore con le volute caratteristiche è opportuno adottare una netta separazione dei problemi: infatti, se da un lato vi sono delle problematiche prettamente tecnologiche, come effettuare la comunicazione fra gli ESB, dall'altro vi sono dei problemi meramente organizzativi, come, ad esempio, decidere quali servizi debbano essere accessibili attraverso un collegamento fra ESB.

Risulta quindi di vitale importanza suddividere il componente in sottosistemi interagenti fra loro, ma non dipendenti gli uni dagli altri. Si ritiene infatti necessario adottare anche in questo caso i principi SOA e nascondere le implementazioni tramite l'uso di interfacce. In figura 2.9 è possibile osservare la divisione proposta in cui si trovano: il *Generic Service Provider* (GSP), il *Generic Service Consumer* (GSC), l'*Imported Services Repository* (ISR), l'*Exported Services Repository* (ESR), il *Context Translation Mechanism* (CTM), il *Boundary Service Repository* (BSR), e infine il *Remote Connector* (RC).

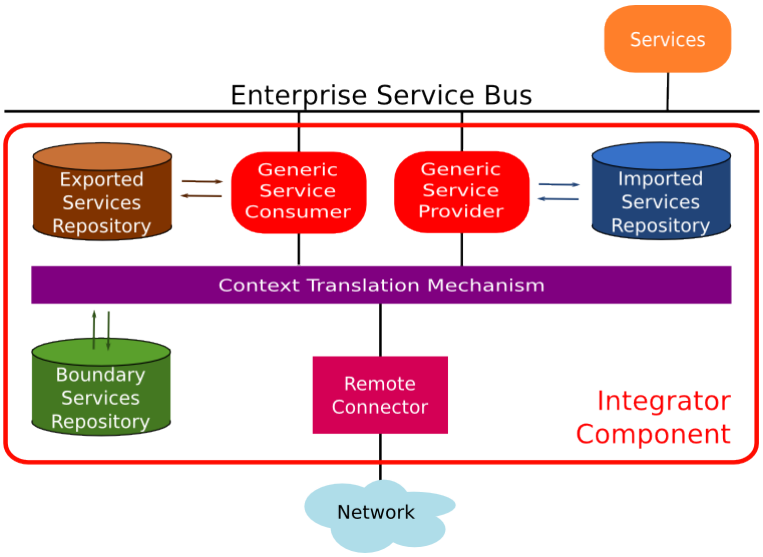


Figura 2.9: Struttura del componente integratore.

Generic Service Provider

Il *Generic Service Provider* ha il compito di realizzare, agli occhi dell'ESB locale, tutti i servizi resi disponibili dagli ESB remoti. Dal momento che l'ESB fornisce trasparenza alla locazione, questo espediente garantisce la possibilità, per gli altri servizi presenti sull'ESB, di utilizzare l'instradamento reso disponibile dall'ESB per rintracciare e consumare i servizi remoti. La responsabilità fondamentale del GSP riguarda l'individuazione dell'ESB sul quale il servizio remoto è collocato.

Generic Service Consumer

Il *Generic Service Consumer* svolge un ruolo duale rispetto a quello del GSP: riceve tutte le richieste di servizio remote e le avvia all'interno dell'ESB locale. Dal momento che l'ESB fornisce trasparenza alla locazione, questo sottosistema consente ai servizi locali di ricevere chiamate da parte dei servizi sugli ESB remoti nello stesso modo in cui ricevono le chiamate locali. La responsabilità fondamentale del GSC consiste nella verifica dell'origine del messaggio, della sua integrità e destinazione: soltanto se proviene da un'origine autorizzata all'accesso del servizio richiesto allora il messaggio verrà instradato sull'ESB locale.

Imported Services Repository

Questo sotto-sistema è necessario per mantenere traccia dei servizi remoti resi disponibili sull'ESB locale e viene utilizzato dal GSP per individuare a quale ESB è necessario indirizzare la chiamata. È attraverso l'ISR che l'amministratore dell'ESB istruisce il componente sui servizi remoti disponibili sull'ESB locale.

Exported Services Repository

L'*Exported Services Repository* è necessario per mantenere traccia dei servizi locali disponibili per invocazioni remote ed è utilizzato dal GSC per verificare se un determinato messaggio ha le giuste autorizzazioni.

È attraverso l'ESR che l'amministratore dell'ESB istruisce il componente su quali servizi locali devono essere resi disponibili per essere invocati remotamente da una lista specifica di ESB.

Context Translation Mechanism

Questo sotto-sistema è il più importante dell'intero componente: è sua responsabilità determinare quali *Boundary Services* utilizzare ed in che ordine farlo. Al fine di svolgere questo compito nel modo più opportuno, è necessario che i BS offrano un'interfaccia di amministrazione ben determinata tramite la quale sia possibile stabilire come comporli fra loro. È un compito molto delicato che richiede uno studio approfondito e un'adeguata formalizzazione, che perciò verrà approfondito nei capitoli successivi.

Boundary Services Repository

Il *Boundary Services Repository* è il contenitore dei BS, ma il suo compito non è solo quello di mantenere una lista aggiornata dei diversi BS disponibili, ma anche di garantire un accesso rapido agli stessi. Dal momento che i BS sono entità attive a cui corrisponde necessariamente del codice eseguibile, il BSR deve occuparsi del caricamento e dello scaricamento degli stessi in modo sicuro ed efficace.

Remote Connector

Il *Remote Connector* è forse la parte più ovvia di questo componente: riceve i dati di indirizzamento dell'ESB remoto dagli strati superiori e invia il messaggio al suo pari che risiede sull'altro ESB. Il protocollo di comunicazione remota è opportuno che sia realizzato sfruttando gli standard per i *Web Services*: solo in questo modo è possibile garantire l'indipendenza dalla singola implementazione del componente integratore.

2.2.2 Comunicazione fra ESB

In figura 2.10 è possibile vedere il generico schema di comunicazione fra due servizi, il *Consumer* ed il *Provider*, collocati su ESB differenti. Il GSP deve quindi "implementare" agli occhi dell'ESB locale l'interfaccia del servizio *Provider*, in modo che quest'ultimo sia disponibile per l'invocazione da parte del *Consumer*. Grazie alla struttura delineata, i servizi remoti sono accessibili alla stregua di quelli locali, e perciò nulla osta a rendere disponibile per l'invocazione remota una qualunque delle topologie precedentemente analizzate: in particolare la chiamata potrebbe attraversare *N* differenti ESB senza risentirne.

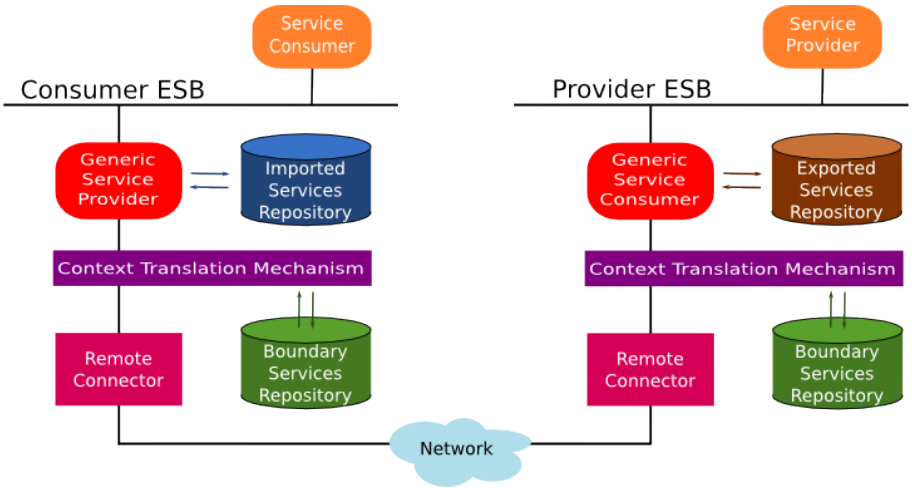


Figura 2.10: Comunicazione fra due ESB

A valle del GSP si trova, come già esaminato, il vero motore per l'integrazione fra gli ESB: il *Context Translation Mechanism*. Tramite l'utilizzo dei BS, il CTM costruisce un messaggio il cui formato è indipendente dall'ESB locale e che viene poi inviato attraverso il *Remote Connector* al pari remoto, che attiverà meccanismi speculari a quelli appena discussi. In particolare, sull'ESB remoto sarà interrogato l'*Exported Services Repository* per verificare che la chiamata che sta per essere effettuata sia legittima o meno.

Per supportare correttamente le modalità di interazione previste è opportuno che la comunicazione fra gli ESB avvenga in modo sicuro:

deve essere instaurato un collegamento (link) tramite l'uso della crittografia, essendo la sicurezza un requisito molto importante che deve essere tenuto in considerazione già a partire dalle prime fasi dello sviluppo. Dal momento che gli algoritmi e le tecnologie per creare un canale sicuro sono in rapida evoluzione, è opportuno che il componente sia completamente indipendente da esse: per raggiungere tale scopo è necessario adottare un meccanismo analogo a quello utilizzato per i *Boundary Service* e che consenta di inserire tali meccanismi a runtime.

2.2.3 La Management Interface

La gestione di una rete di ESB può essere effettuata secondo diverse politiche, dipendenti dalle scelte topologiche ed architetturali; l'approccio più efficace è sicuramente la realizzazione di un'interfaccia di management abbastanza flessibile, così da consentire l'utilizzo del componente in tutti i possibili scenari di impiego. In particolare, si ritiene necessario fornire un'interfaccia semplice, mentre sarà compito dei componenti di livello superiore realizzare le politiche e fornire l'interazione con l'amministratore.

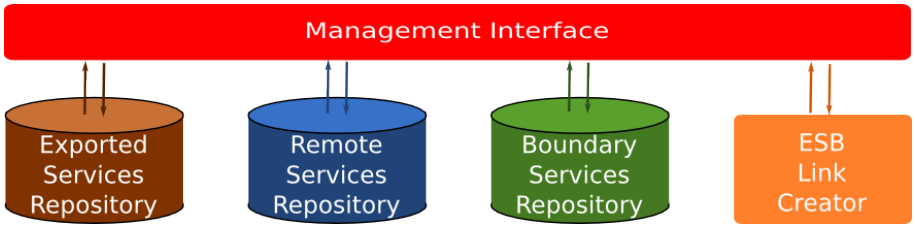


Figura 2.11: Componenti della Management Interface

L'interfaccia di management deve consentire l'accesso in maniera uniforme alle seguenti funzionalità (fig. 2.11):

- la creazione di un collegamento fra due ESB (*Link Creator*);
- l'aggiunta e la rimozione dei *Boundary Services*;
- l'aggiunta o la rimozione di servizi dal *Remote Services Repository*, che si ricorda contiene i servizi "importati" da altri ESB;

- l'aggiunta o la rimozione di servizi dall'*Exported Services Repository*, che si ricorda contiene i servizi "esportati" verso altri ESB.

2.3 Integrazione all'Interno di Ambienti Standard

La soluzione sopra delineata, e che verrà approfondita nel seguito, è non schierata né vicina a soluzioni proprietarie per la realizzazione di ESB, ma, al fine di validare l'approccio proposto, è necessario scegliere una o più implementazioni di riferimento da integrare fra loro. Lo sviluppo di un siffatto componente è un compito molto complesso che può essere articolato in modi anche molto distinti fra loro, perciò la scelta della soluzione di riferimento assume un ruolo fondamentale, anche considerando che ogni ESB è corredato da una suite completa di strumenti di sviluppo.
In primo luogo si potrebbe considerare l'integrazione fra ESB identici: infatti questo approccio porta ovvi benefici come, ad esempio, poter realizzare un solo componente per entrambi gli ESB. Tale approccio semplificherebbe notevolmente lo sviluppo, ma porterebbe ad una soluzione specifica della piattaforma utilizzata e perciò poco portabile. Si potrebbe allora pensare, al contrario, che la soluzione stia nell'integrare due ESB diversi tra loro ma, viste le specificità di queste tecnologie, tale approccio richiederebbe un impegno doppio rispetto alla prospettiva appena esaminata.
Dal momento che l'unica via adottabile implica lo sviluppo di un singolo componente da collocare su entrambi gli ESB da integrare, è opportuno che questo venga sviluppato per un ambiente standard. La specifica *Java Business Integration*, come si vedrà nel capitolo 4, rappresenta uno standard adeguato per la realizzazione dei componenti da innestare sull'ESB, garantendo così la realizzazione di un'unica soluzione capace di eseguire su tutti gli ESB che si basano sullo standard. Se sviluppato basandosi sullo standard sarà perciò possibile integrare fra

loro diversi ESB standard in modo assolutamente indipendente dagli stessi.

Capitolo 3

Considerazioni sulla Integrazione tra ESB

Il presente capitolo è dedicato allo studio della maggiore problematica rilevata durante l'analisi dei modelli per l'integrazione tra Enterprise Service Bus (sez. 2.1.3): il trasferimento delle informazioni a contesto della chiamata attraverso gli ESB coinvolti nello scambio di messaggi. Questa problematica è altamente significativa, infatti rappresenta un problema non strettamente legato all'ambito nel quale è stato inizialmente presentato ma, al contrario, può riguardare tutti quei sistemi per il distribuito, i *middleware*, in cui è prevista un'invocazione remota.

L'approccio a *Boundary Services*, come proposto nel capitolo 2, è una metodologia per affrontare il problema in esame coerentemente con i principi della SOA. Alla luce di quanto esaminato si ritiene opportuno dedicare il presente capitolo all'analisi e alla progettazione di un meccanismo per il trasferimento del contesto in modo assolutamente indipendente dall'obiettivo di integrazione fra ESB. L'unico vincolo imposto riguarda l'obbligo di effettuare il trasferimento del contesto basandosi su XML che, da un lato, garantisce la possibilità di realizzare differenti implementazioni, e, dall'altro, consente una facile gestione dei messaggi scambiati fra i servizi, anch'essi codificati in XML.

3.1 Requisiti per un Meccanismo di Traslazione

L'analisi dei requisiti per un generico servizio di traslazione del contesto deve necessariamente partire dalla disamina delle informazioni che compongono il contesto della chiamata. In particolare può essere corredata da informazioni riguardanti, ad esempio, la sicurezza, le transazioni o la qualità di servizio. Inoltre, durante il trasferimento della chiamata, è opportuno verificare che il contenuto del messaggio sia correttamente inviabile attraverso la rete: in particolare potrebbe includere informazioni audio-video che necessitano di ricodifica, si pensi ad esempio ad un file audio che, se codificato in WAV, occuperebbe decine di MBytes, mentre lo stesso file in formato MP3 risulterebbe essere di un ordine di grandezza inferiore. Si potrebbe pensare che quest'ultimo compito non sia imputabile al meccanismo di traslazione, ma tale sistema deve comunque svolgere compiti analoghi: bisogna tenere presente che il sistema potrebbe essere usato per trasferire il contesto di chiamate basate sui *Web Services*. Come è stato evidenziato nella sezione 1.2, sono disponibili, per gestire il trasferimento delle medesime informazioni, gli standard WS-* utilizzabili per incapsulare alcuni aspetti del contesto direttamente all'interno del messaggio vero e proprio. È plausibile che si possa voler trasferire tali aspetti andando a modificare il messaggio direttamente, piuttosto che attraverso informazioni aggiuntive.
Le due esigenze sopra menzionate si traducono formalmente in due requisiti assolutamente indipendenti fra loro:

1. gestire informazioni propriamente di contesto, in modo che sia possibile inviare o ricevere tutto ciò che non è contenuto all'interno del messaggio vero e proprio;

2. trasformare il messaggio, aggiungendo o rimuovendo alcune parti per adattare il contenuto alla trasmissione sulla rete e viceversa.

3.2 I Boundary Services

Nella sezione 2.1.3 sono stati introdotti, per risolvere i problemi esposti nella precedente sezione, i *Boundary Services*. Ognuno di questi deve occuparsi di un particolare aspetto del contesto e limitarsi strettamente a formalizzare la caratteristica di cui è responsabile in un linguaggio, basato su XML, che deve essere comprensibile dal suo pari remoto.
Questi particolari servizi devono rispettare i principi SOA nella loro interezza, perciò ogni BS deve:

- definire l'interfaccia offerta nei termini del formato dei messaggi accettati ed emessi dagli stessi;
- essere senza stato, nel senso che nulla deve rimanere fra un invocazione e l'altra;
- non deve fare alcuna assunzione sullo stato della chiamata corrente.

Dopo questa necessaria formalizzazione del concetto di BS è opportuno, alla luce dei requisiti esaminati nella sezione precedente, suddividere i BS in due insiemi disgiunti a seconda di come sono utilizzati all'interno del meccanismo di traslazione:

1. Si definiscono "BS di aggiunta" i servizi che trasformano un aspetto del contesto vero e proprio in XML, aggiungendo quindi al meccanismo di traslazione la capacità di gestire un nuovo aspetto del contesto.
2. Si definiscono "BS di trasformazione" i servizi che modificano le informazioni già codificate in XML in altro XML, rendendo così possibile, per il meccanismo di traslazione, la trasformazione di alcuni aspetti che potrebbero non essere adatti al trasferimento sulla rete.

In realtà questa suddivisione consente di ottenere un grado di flessibilità maggiore rispetto a quanto richiesto, dal momento che è possibile

combinare fra loro le due tipologie di BS indipendentemente dal fatto che le trasformazioni XML riguardino o meno il messaggio vero e proprio. In sostanza, il meccanismo che si sta delineando non tratta in maniera separata contesto e messaggio, ma li accomuna realizzando un modello unico per la trasformazione e serializzazione della chiamata. L'unificazione riesce a garantire una semplificazione del problema che si riflette sulle capacità del sistema di comporre i Boundary Services in modo da garantire i gradi di manutenibilità e flessibilità richiesti.

3.3 I Service Access Point

Il modello formale per descrivere i *Boundary Services* risale al 1979 e più precisamente corrisponde all'*OSI Reference Model* [Day83]. Infatti è opportuno riportarsi ad un modello teorico al fine di studiare le proprietà dei *Boundary Services* e, di conseguenza, come sia possibile organizzarli nel modo più opportuno.

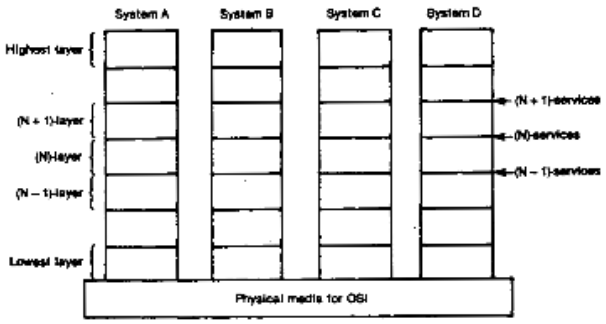


Figura 3.1: Struttura a livelli secondo l'*OSI Reference Model*.

Nell'*OSI Reference Model* (RM) la comunicazione avviene fra processi applicativi che vengono eseguiti su differenti sistemi (nella specifica originale un computer autonomo). La suddivisione in livelli (*layering*) è la tecnica ingegneristica usata per definire correttamente il modello. In particolare, ogni sistema viene visto come logicamente suddiviso in una sequenza di sotto-sistemi, ognuno dei quali ha un grado o posizione. Il RM definisce il concetto di livello come l'insieme dei sottosistemi dello stesso grado su tutti i sistemi della rete (fig. 3.1).

All'interno dei livelli vivono le *entity*, che possono comunicare fra loro sia se sono a livelli diversi sia se appartengono allo stesso livello, ed in quest'ultimo caso si definiscono *peer entity*.
Per semplicità, un generico livello è identificato tramite la dicitura (N)-layer, mentre i livelli sovrastante e sottostante sono indicati rispettivamente come ($N+1$)-layer e ($N-1$)-layer. La stessa notazione è usata per identificare ogni cosa relativa ai livelli, ed in particolare le entità del (N)-layer sono le (N)-entities (fig. 3.2).

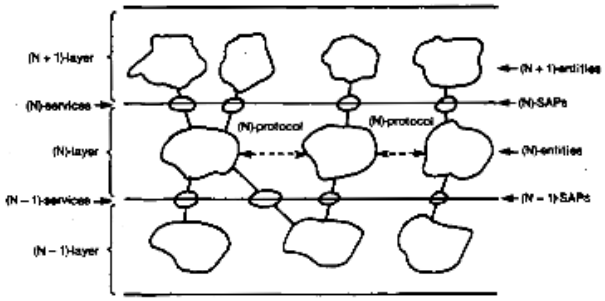


Figura 3.2: Schema di funzionamento dei SAP.

Alla base della suddivisione in livelli per le applicazioni distribuite vi è l'idea che ogni livello aggiunga valore ai servizi offerti dai livelli precedenti: in questo modo al livello più alto vengono offerti tutti i servizi necessari ad eseguire le applicazioni distribuite. Il modello offre una buona divisione dei problemi poiché introduce un concetto analogo all'accoppiamento debole proposto dalla SOA, anche se non si sono ancora posti i problemi che la SOA propone di risolvere. In particolare, ogni livello offre al sovrastante soltanto una definizione dei propri servizi completamente indipendente dalla realizzazione degli stessi.
Dal momento che ciascun livello offre dei servizi al livello superiore, è possibile affermare che un servizio è una funzionalità offerta dal (N)-layer alle ($N+1$)-entities attraverso i (N)-service access point, (N)-SAP per brevità (fig. 3.2). I SAP rappresentano le interfacce logiche fra le (N)-entities e le ($N+1$)-entities, in modo tale per cui una ($N+1$)-entity possa comunicare con una (N)-entity soltanto attraverso un (N)-SAP. Inoltre, un (N)-SAP può essere offerto ed utilizzato soltanto ad una

sola ($N + 1$)-entity, ma una (N)-entity può offrire svariati SAP ed una ($N + 1$)-entity può usarne molteplici.
È importante notare come vi siano due differenti tipologie di comunicazioni in atto all'interno di questo modello concettuale: una verticale ed una orizzontale. La comunicazione verticale si riferisce allo scambio di informazioni fra entità di livelli adiacenti sullo stesso sistema, mentre si ha una comunicazione orizzontale quando comunicano fra loro due entità dello stesso livello ma su sistemi differenti (remoti).
Questo modello, anche se solo accennato in questa sede, offre una solida base per descrivere il processo di trasformazione dell'informazione imposto dal secondo requisito del meccanismo per la traslazione del contesto (sez. 3.1). Purtroppo attraverso l'OSI RM non è possibile modellare altrettanto bene l'indipendenza dei vari aspetti del contesto, come imposto dal primo requisito. Anziché rigettare un modello che si è dimostrato estremamente valido, si ritiene opportuno valutare come estenderlo al fine di gestire, nel modo migliore, l'indipendenza dei vari aspetti del contesto.

3.4 I Piani di Interazione

Al fine di garantire la completa modellazione teorica di entrambi i requisiti esposti nella sezione 3.1, è opportuno riferirsi ad una proposta fatta da alcuni ricercatori tedeschi sull'introduzione dei piani di interazione nel modello basato sui SAP [Her01]. Nell'articolo i ricercatori notano un'importante similitudine sul meccanismo di funzionamento di tre protocolli per le telecomunicazioni: ISDN, GSM e UMTS.
In particolare, in figura 3.3 [Her01] è possibile esaminare come il protocollo GSM (ma anche ISDN e UMTS) sia basato sul concetto di piano (*plane*): ognuno di questi incapsula delle funzionalità di servizio e può essere basato internamente su una infrastruttura a livelli, in modo analogo a quanto proposto da OSI. Nelle telecomunicazioni ogni piano ha le sue specifiche responsabilità:

- l'*user plane*, con la sua struttura a livelli, trasferisce i dati relativi

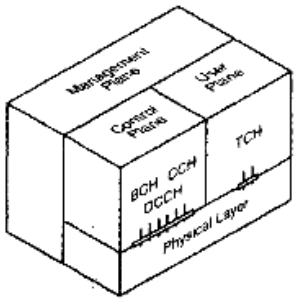


Figura 3.3: Piani di interazione del protocollo GSM.

alle chiamate ed effettua il controllo di flusso delle connessioni;

- il *control plane* si occupa di stabilire e rilasciare le connessioni;
- il *management plane* offre i servizi di coordinamento fra gli altri piani.

L'OSI RM non è adeguato a gestire i piani e perciò le informazioni di controllo vengono mescolate con i dati utente ad ogni passaggio di livello che i dati stessi devono affrontare. L'idea proposta dagli autori è semplice: se è possibile distinguere i SAP gli uni dagli altri è anche possibile separarli in piani. Infatti una qualsiasi classificazione dei SAP, ad esempio associando ad ognuno di essi un attributo, può consentire di identificare chiaramente un SAP come appartenente ad un insieme di SAP, che è, per gli autori, un altro modo di identificare i piani.
Al fine di suddividere i SAP in piani, gli autori si concentrano sulla relazione fra le $(N + 1)$-entities e i (N)-SAP: in particolare affermano che una $(N + 1)$-entity debba offrire un (N)-SAP^{-1} che rappresenta la controparte del (N)-SAP. Ciascun SAP e SAP^{-1} sono identificati non più sulla base di un "numero di livello", come poteva avvenire nell'ambiente OSI, bensì attraverso la definizione della loro interfaccia: P e P^*. Questi definiscono, in modo facilmente intuibile, i messaggi inviabili e ricevibili attraverso un particolare SAP, seguendo le stesse regole che portano alla SOA.
Tornando al problema originario, ovvero la ricerca di un modello teorico capace di supportare entrambi i requisiti proposti nella sezione

3.1, si può facilmente intuire come la suddivisione in piani proposta dagli autori dell'articolo considerato garantisca la flessibilità richiesta. Infatti ogni informazione di contesto deve, al fine di garantire l'indipendenza dei *Boundary Services*, essere realizzata su un diverso piano. Ad esempio, si potrà avere un piano dedicato alla trasformazione del messaggio e svariati piani per gestire aspetti come la qualità di servizio e l'autenticazione. In particolare, considerando il trasferimento delle informazioni relative all'utente che effettua la chiamata, si può vedere come queste siano completamente indipendenti dal trasferimento del messaggio, ma vadano in ogni caso comunicate al pari remoto. Inoltre, realizzando il trasferimento dei dati utente tramite un altro piano, non si introduce dipendenza fra i *Boundary Services*, consentendo una facile estendibilità del sistema

In modo simile è possibile aggiungere le medesime informazioni all'interno del corpo del messaggio vero e proprio se questo lo supporta: infatti se svariate entity offrono un linguaggio P e consumano P^*, inverso di P, è possibile comporle in qualsiasi modo. Ad esempio, riferendosi nel particolare ai *Web Services* (sez. 1.2), è possibile che le informazioni sulla sicurezza non debbano essere codificate in un piano diverso, ma in un'apposita sezione all'interno del messaggio. Il *Boundary Service* che trasforma il messaggio per aggiungere queste informazioni non ne modifica il linguaggio e perciò avrà un linguaggio di uscita P^*. Se, come è possibile, vi sono parecchi BS che effettuano questo particolare tipo di trasformazione, allora è altrettanto vero che non importa l'ordine nel quale questi operano, garantendo così una efficace componibilità.

3.5 Utilizzo dei SAP per la Traslazione del Contesto

Nella sezione precedente si è analizzato come i SAP, nella loro ultima definizione, possano fornire un valido modello teorico per i *Boundary Services*, ma, come tutte le analisi teoriche, è opportuno specificare come applicarli al caso concreto.

Si propone quindi di considerare un *Boundary Service* come una (*N*)-entity che esponga al BS di strato superiore un SAP, come definito nella sezione precedente, la cui interfaccia *P* deve definire il formato del messaggio XML che può essere ricevuto dal BS. Dal momento che il principale linguaggio utilizzato per specificare il formato di un file XML è *XML Schema* [W3C04], risulta logico che *P* sia espresso tramite tale standard e sia identificato da un *namespace*.
Dopo avere analizzato come i BS realizzano i SAP è opportuno considerare i SAP^{-1}, poiché sono possibili due casi a seconda della loro posizione all'interno della pila di trasformazione:

- se il BS non è ultimo, la definizione del suo linguaggio di uscita P^*, anch'esso uno schema, deve corrispondere a quello del BS sottostante;
- se il BS è ultimo, deve in ogni caso definire il suo linguaggio di uscita P^* al fine di garantire l'estendibilità futura della pila.

Ogni *Boundary Service* è perciò definito da una coppia di namespace, *P* e P^*, che ne garantiscono la componibilità. È importante notare come *P* possa assumere valore nullo nel caso in cui il BS sia di "aggiunta", al contrario P^* deve essere sempre specificato. Oltre alla definizione dei linguaggi, ad ogni *Boundary Service* deve, al fine di garantire un corretto indirizzamento interno del meccanismo di traslazione, essere attribuito un nome univoco all'interno del sistema stesso.
Ad esempio, riprendendo quanto affermato nella sezione precedente, è possibile definire un BS che crei un piano relativo all'autenticazione ed abbia *P* nullo, mentre P^* è possibile porlo a "*http://nome.dominio/autenticazione*". In questo modo, se un altro servizio deve elaborare le stesse informazioni, basterà che ponga il suo *namespace* di ingresso uguale a "*http://nome.dominio/autenticazione*". In modo simile, se si vuole aggiungere informazioni all'interno di un messaggio scambiato fra *Web Services*, questo dovrà essere codificato secondo il namespace SOAP "*http://www.w3.org/2001/12/soap-envelope*": riportando tale namespace come linguaggio di ingresso e di uscita si consente la componibilità dei vari *Boundary Services*.

La modellazione proposta, dal momento che è basata su una coppia di schemi XML, potrebbe condizionare in maniera negativa le prestazioni del meccanismo nel momento in cui sono da effettuare molteplici trasformazioni (e validazioni) per ogni chiamata, ed è lecito supporre che il meccanismo di traslazione verrà sfruttato in continuazione. Si ritiene però questa modellazione l'unica possibile al fine di riuscire a realizzare nel modo corretto i requisiti, ma sarà sicuramente necessario studiarne le prestazioni in appositi ambienti di verifica.

Capitolo 4

Java Business Integration

La specifica *Java Business Integration* (JBI) [Ten05], definita dalla Sun Microsystems, già fornitrice del linguaggio Java, pone l'obiettivo di fornire un'architettura standard per i problemi di integrazione in cui il formato dei componenti sia ben definito, così da consentire l'integrazione con soluzioni di parti terze: l'intento è quello di definire un "ecosistema" in cui più soggetti possono fornire i componenti agli utenti finali, che possono quindi scegliere il componente che soddisfa maggiormente le proprie esigenze [Ten05].
JBI specifica un'architettura a componenti *plug-in* interagenti fra loro (figura 4.1 [Ten05]) tramite uno scambio di messaggi mediato dal container stesso. JBI definisce:

- le interfacce che il container ed il *plug-in* devono implementare per realizzare lo scambio di messaggi;
- un modo flessibile per istruire i componenti sui servizi che debbono esporre, detto *deployment*;
- la procedura da utilizzare per inserire un nuovo componente all'interno del container, detta *installation*;
- una interfaccia di management unica per gestire le varie fasi del ciclo di vita dei componenti.

I servizi forniti dai componenti sono descritti dal componente stesso al container JBI, utilizzando gli standard WSDL 1.1 o 2.0. Da un lato,

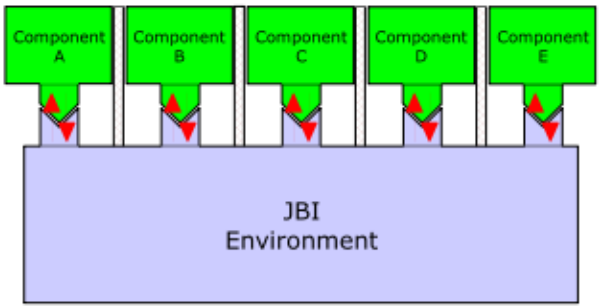


Figura 4.1: L'architettura a componenti di JBI.

questo meccanismo fornisce un modello dei servizi astratto ed indipendente dalla tecnologia, e, dall'altro, permette di dichiarare i metadati di interesse al consumatore e allo stesso container JBI. È possibile interrogare JBI per ottenere i WSDL dei servizi disponibili, ed in questo modo JBI fornisce la base per un rudimentale registro dei servizi.

4.1 Componenti

La specifica inizialmente divide i componenti in due categorie:

- I *Service Engine* (SE) sono utilizzati per contenere tutti i servizi che devono essere eseguiti all'interno del container JBI. In particolare sono pensati per contenere sia servizi che realizzino la logica di business sia servizi di trasformazione (come, ad esempio, modificare la struttura di un documento XML tramite l'utilizzo di XSLT), ma possono anche contenere intere applicazioni Java.

- I *Binding Component* (BC) sono quindi usati per integrare applicazioni che non hanno a disposizione una API Java. In particolare consentono di integrare servizi esterni all'interno di un ambiente JBI, utilizzando protocolli sia standard che proprietari.

Anche se la distinzione fra le tipologie di componenti viene effettuata all'interno della specifica basandosi sopratutto sulla disponibilità di API Java, lo stesso standard puntualizza poi che la distinzione per garantire una corretta realizzazione della SOA è differente. Infatti, dal

momento che i servizi di business sono autonomi (vedasi 1.1), è opportuno che questi siano realizzati esternamente all'ESB e quindi integrati tramite l'uso di BC. Ad esempio, se si vuole esporre un servizio avente un'API Java, è necessario istruire un componente per renderla disponibile, ma questa operazione implica scrivere del codice assolutamente dipendente dal componente di destinazione, quindi contraria ai principi SOA. Al contrario, è opportuno che tutta la logica di integrazione fra i servizi, ad esempio processi BPEL e trasformazioni XSLT, risieda all'interno del container JBI, in quanto soltanto in quella sede sono disponibili i riferimenti di tutti i servizi.
I componenti si interfacciano con il container JBI attraverso due meccanismi: le Service Provider Interface (SPI) e le Application Program Interface (API). Le SPI devono essere implementate dal componente, sia esso un SE o un BC, mentre le API sono esposte ai componenti dal container. La distinzione fra le tipologie di componenti, SE e BC, è quindi soltanto teorica, in quanto la struttura degli stessi è la medesima: questo approccio garantisce un'ottima flessibilità realizzativa.

4.2 Architettura Generale

L'ambiente JBI viene eseguito interamente all'interno di una singola *Java Virtual Machine* (JVM), mentre le entità esterne sono rappresentate dai fornitori e consumatori dei servizi che vengono integrati da JBI.
In figura 4.2 è possibile notare i principali componenti dell'ambiente JBI, di cui il più importante è il *Normalized Message Router* (NMR), che fornisce il meccanismo di comunicazione fra i servizi. Questi ultimi possono essere esposti indipendentemente sia dai SE che dai BC, ed è compito del NMR indirizzare correttamente i vari servizi, realizzando una comunicazione basata sulle interfacce WSDL (vedasi 1.2.1).
Infine JBI definisce anche delle interfacce per la gestione dell'ESB accessibili tramite lo standard *Java Management Extension* (JMX). Questa specifica definisce un'architettura standard per l'amministrazione di applicazioni, sistemi e reti; appoggiandosi ad essa l'applicazione non si deve occupare di realizzare l'interfaccia utente, poiché è possibi-

le utilizzare le molteplici realizzazioni già realizzate per questo standard [Sun06]. L'utilizzo dello standard in questione permette quindi la realizzazione di strumenti di gestione completamente indipendenti dall'ESB e dai componenti di destinazione.

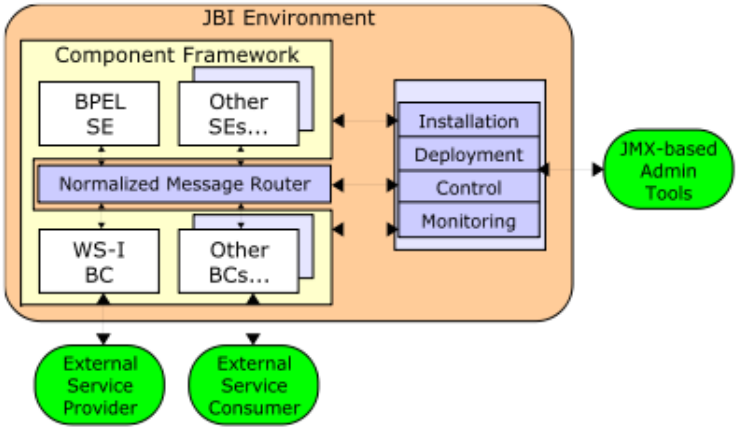


Figura 4.2: L'architettura JBI [Ten05].

4.3 Pattern per lo Scambio di Messaggi

Il container JBI deve poter modellare l'invocazione delle operazioni sui servizi, e per questo definisce dei *Message Exchange Pattern* (MEP), concetto definito dalla *WSDL 2.0 Predifined Extentions* come la sequenza e la cardinalità dei messaggi astratti scambiati durante un'operazione [W3C07b]. In pratica, un MEP definisce delle sequenze predeterminate di messaggi che devono obbligatoriamente avvenire durante l'invocazione di un'operazione fra il produttore ed il consumatore (un MEP rispetta sempre il punto di vista del fornitore del servizio). Anche se ogni implementazione può definire i propri pattern per l'interazione, la seguente lista seguente rappresenta quelli che devono essere obbligatoriamente presenti:

- *In-Only*: il consumatore invia una richiesta al fornitore senza avere alcuna conferma dell'avvenuta ricezione (fig. 4.3 [Ten05]);

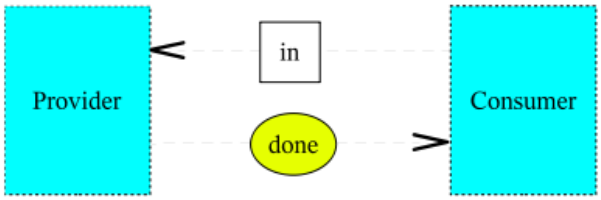


Figura 4.3: Il MEP *In-Only*.

- *Robust In-Only*: il consumatore invia una richiesta al fornitore e questi può comunicare un eventuale fallimento nell'elaborazione della richiesta (fig. 4.4 [Ten05]);

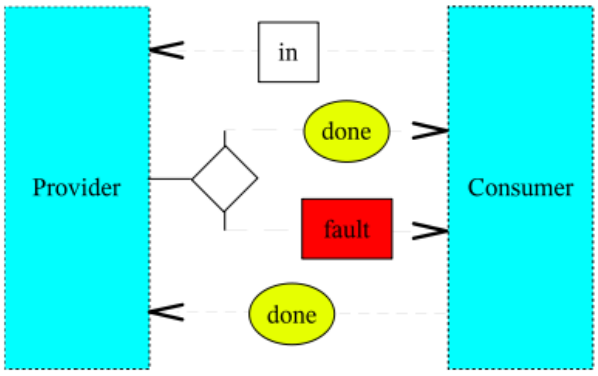


Figura 4.4: Il MEP *Robust In-Only*.

- *In-Out*: il consumatore invia una richiesta al fornitore e questi deve comunicare la risposta o il fallimento(fig. 4.5);

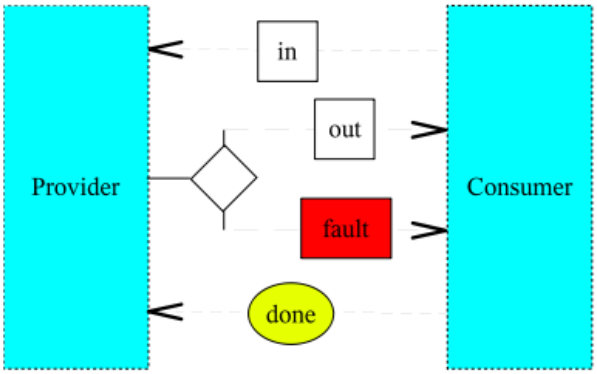


Figura 4.5: Il MEP *In-Out* [Ten05].

- *In Optional-Out*: il consumatore invia una richiesta al fornitore e questi può terminare lo scambio, comunicare una risposta o un fallimento (fig. 4.6 [Ten05]). In questo MEP sia il consumatore che il fornitore hanno la possibilità di inviare notifiche dei rispettivi fallimenti a seguito della ricezione di un messaggio.

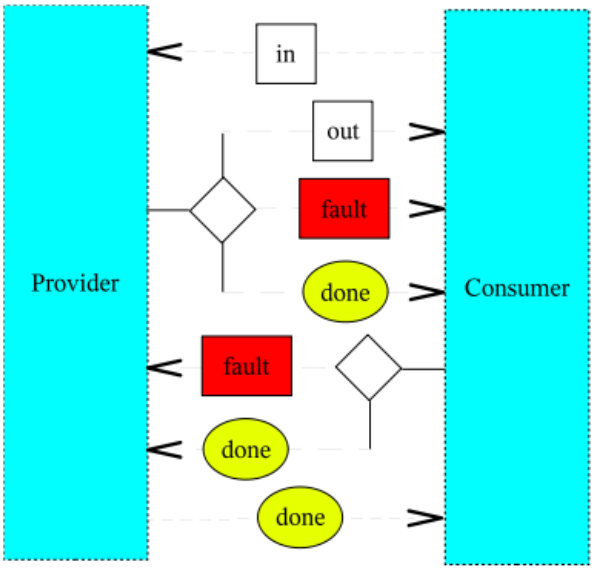


Figura 4.6: Il MEP *In Optional-Out*.

4.4 Instradamento dei Messaggi

Per poter recapitare correttamente i messaggi il consumatore deve specificare il servizio di destinazione secondo le modalità fornite da JBI. Si ricorda (vedasi 1.2.1) che ogni servizio è definito da un *interface name*, un *service name* ed un *endpoint name*, ed è perciò possibile utilizzare questi elementi per poter determinare il servizio di destinazione. I possibili indirizzamenti, in ordine crescente di specificità, sono:

- Per tipo: il consumatore specifica solo l'*interface name* indicando così il tipo di servizio desiderato; è compito del container scegliere un fornitore quando il messaggio viene inviato.

- Per servizio: il consumatore specifica solo il *service name*, indican-

do così il fornitore desiderato, ma non l'endpoint di destinazione; il sistema sceglierà un endpoint quando il messaggio viene inviato.

- Per endpoint: il consumatore specifica l'esatto endpoint che vuole utilizzare. Se si opta per questo tipo di instradamento è importante specificare che l'endpoint può essere determinato sia in maniera statica, fissandolo cioè in sede di progettazione, sia in maniera dinamica, chiedendo durante l'esecuzione tutti gli endpoint disponibili per una determinata interfaccia.

4.5 Processo di Sviluppo di un'Applicazione JBI

La specifica JBI si rivolge ad un pubblico composto principalmente dagli implementatori di *Enterprise Service Bus* e dei componenti: si suppone cioè che lo sviluppatore di una soluzione di integrazione basata su JBI dovrebbe interfacciarsi soltanto con i componenti che andrà ad utilizzare. A tal scopo JBI definisce il modo in cui si effettua il *deployment* di una applicazione, rendendola un'operazione completamente indipendente dai componente che si va ad utilizzare. JBI specifica perciò due tipologie di archivi relativi all'operazione in esame: le *Service Unit* e i *Service Assembly*.

Service Unit

Una *Service Unit* (SU) è l'entità minima di cui è possibile effettuare il *deployment*, ed è perciò destinata ad un singolo componente. Il contenuto di questo archivio è opaco al container ma trasparente al componente di destinazione. L'unico contenuto di interesse per JBI è un file di descrizione per definire i servizi prodotti e consumati dalla SU.

Service Assembly

Un *Service Assembly* (SA) consente di effettuare, all'interno di un ambiente JBI, il *deployment* di molteplici SU contemporaneamente. Questa funzionalità è necessaria poiché spesso sono necessarie molteplici SU, da collocarsi su molteplici componenti, per realizzare una nuova applicazione di integrazione. JBI fornisce quindi la possibilità di gestire tutte le SU che compongono l'applicazione collettivamente, semplificando così la gestione del sistema.

4.6 Limiti dell'Architettura

Il principale limite dell'architettura JBI risiede nell'assunzione iniziale: la specifica prospetta una realtà dove molteplici produttori vendono componenti che verranno poi usati dagli utenti/sviluppatori. Questa realtà, purtroppo, non si è venuta a creare: infatti, anche se vi sono molteplici implementazioni della specifica JBI, ognuna di queste propone i propri componenti per risolvere la stessa gamma di problemi, ad esempio i processi di business, le trasformazioni XSLT, i binding componente per i WS su HTTP, ecc... . Ogni implementazione crea, quindi, un lock-in indipendentemente dall'utilizzo della piattaforma standard. La specifica consente comunque di portare un componente da una piattaforma ad un altra, ma non è un'operazione immediata, poiché i componenti sono pensati per sfruttare il pieno potenziale di una particolare implementazione. Questa problematica fornisce una motivazione tecnica all'integrazione fra più ESB, modalità che risulta essere molto efficace per superare il limite appena esposto.
Un argomento non trattato né all'interno di questo capitolo né nella specifica è la durata degli scambi di messaggi, infatti il modello che viene proposto implica l'utilizzo di un processo leggero, un *Thread*, per ogni scambio. In particolare tale modellazione implica la necessità di tenere allocate delle risorse per ogni scambio, anche se questo ha una lunga durata e questo causa una limitazione delle prestazioni ottenibili.
Vi è un'ulteriore questione non trattata all'interno della specifica: l'in-

vocazione remota all'interno dell'ambiente JBI. La specifica descrive in maniera dettagliata come avvengono le chiamate all'interno del container, ma nulla indica su come debba essere gestita l'invocazione di un servizio attraverso più container JBI. Questa mancanza è sentita soprattutto per quanto riguarda le problematiche relative all'alta affidabilità, infatti è possibile pensare di replicare il container su vari nodi e di gestire le invocazioni in modo trasparente alla locazione dei servizi. Il problema in questione è simile a quanto analizzato nel capitolo 2: vi è, anche in questo caso, la necessità di garantire il corretto trasferimento del contesto fra i vari nodi; il lavoro svolto in questa sede si inserisce perciò in un filone più ampio e potrebbe quindi essere influenzato da eventuali sviluppi in quell'area.

Capitolo 5

Progettazione del Componente Integratore

Nei capitoli precedenti si sono proposte considerazioni teoriche al riguardo dell'integrazione fra *Enterprise Service Bus* ed in particolare il trasferimento del contesto. Avendo posto le basi teoriche per affrontare i principali problemi che possono occorrere durante l'integrazione, si può ora affrontare la progettazione dell'intero componente integratore. Si ricorda infine che si vogliono fare considerazioni questa sede il più generale possibile, sebbene, per motivazioni implementative, si farà riferimento alla piattaforma *Java Business Integration*, così come esposta nel capitolo 4.

Questo capitolo non vuole descrivere nel dettaglio come sia stato progettato il componente, piuttosto vuole affrontare tutti quei problemi di natura progettuale che è necessario chiarire e risolvere prima di procedere alla realizzazione. In particolare nella prima sezione, riprendendo quanto già esposto precedentemente, si affronta l'architettura del componente, esplicitando anche le scelte atte a semplificarla. Nella seconda sezione, invece, si parla dell'interfaccia di gestione del componente stesso poiché rappresenta, come espresso nel capitolo 2, un elemento di primaria importanza. Infine vengono esposte le maggiori problematiche relative ai singoli sotto-sistemi del componente, senza però entrare in eccessivi dettagli realizzativi.

5.1 Architettura del Componente

Per affrontare la progettazione del componente è opportuno riprendere la struttura delineata nel capitolo 2, osservabile in figura 5.1. Nell'architettura proposta in quella sede si trovano i seguenti sotto-sistemi: il *Generic Service Provider* (GSP), il *Generic Service Consumer* (GSC), l'*Imported Services Repository* (ISR), l'*Exported Services Repository* (ESR), il *Context Translation Mechanism* (CTM), il *Boundary Service Repository* (BSR), e infine il *Remote Connector* (RC).

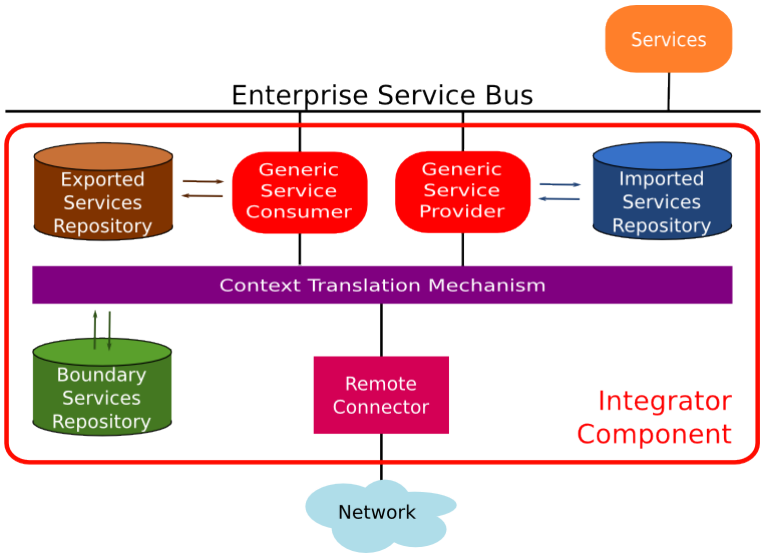


Figura 5.1: Struttura del componente integratore.

Al fine di realizzare un sistema manutenibile e facilmente estendibile si è ritenuto opportuno adottare, nella progettazione dello stesso, tutti i principi riguardanti la SOA, ed in particolare l'accoppiamento debole. Infatti ogni sotto-sistema del componente espone un'interfaccia attraverso cui permette la collaborazione con le altri parti: attraverso questa tecnica è possibile effettuare ampie modifiche (refactoring) senza che le funzionalità del componente vengano intaccate. Questo approccio si scontra con la necessità delle singole parti di essere a conoscenza dell'implementazione delle altre al fine di poterle rintracciare ed invocare i servizi che queste offrono.
Per garantire una corretta integrazione fra i sotto-sistemi è stato adot-

tato il framework *Spring* [Spr09], che ha consentito una notevole semplificazione della logica di avvio del componente stesso. La corretta separazione dei vari moduli del componente e l'interazione attraverso interfacce consente di verificare in maniera automatizzata il funzionamento di ogni sotto-sistema in modo indipendente: intatti, se non si fosse utilizzato *Spring*, molte delle dipendenze fra i moduli avrebbero dovuto essere statiche, rendendone impossibile la verifica.

5.1.1 Il Framework Spring

Il framework *Spring* fornisce, in primo luogo, un container capace di gestire il ciclo di vita degli oggetti in esso contenuti in modo trasparente agli stessi. Le funzionalità offerte da questo container sono fondamentalmente due: la *Dependency Injection* (DI) e l'*Aspect Oriented Programming* (AOP) [Wall08].
Martin Fowler, colui che ha coniato il termine *Dependency Injection*, definisce la DI come un modo di disaccoppiare moduli fortemente accoppiati [Fow04]: in particolare è un modo di alterare il funzionamento di un componente (non nel significato JBI del termine), modificando uno dei servizi (anche in questo caso non nel senso proposto dalla SOA) da cui dipende. Secondo questo pattern non è più il componente che recupera o crea i servizi da cui dipende, ma piuttosto questi gli vengono inseriti all'interno in fase di inizializzazione. Dal momento che un componente non dipende più dagli altri per essere sviluppato, è possibile utilizzare dei Mock Objects [Fow07] per simulare il comportamento dei servizi da cui dipende, garantendone così il corretto funzionamento.
La DI consente, più nella pratica, di esplicitare in modo dichiarativo sia gli oggetti contenuti all'interno del container sia le collaborazioni che intercorrono fra di essi. Il framework, infatti, provvede alla gestione del ciclo di vita e al soddisfacimento delle dipendenze: la principale conseguenza di questo meccanismo risiede nel fatto che il progettista è portato a pensare alle entità in termini di semplici *Plain Old Java Object* (POJO), migliorando così la collaudabilità di ogni singolo componente.

La programmazione orientata agli aspetti (AOP) rappresenta un ulteriore modo di ragionare sui componenti, è infatti un paradigma che mira a gestire lo sviluppo degli aspetti trasversali (*crosscutting aspects*) separatamente dal resto del sistema [Kic97]. L'AOP consente al progettista di separare le logiche di business dell'applicazione da problematiche trasversali, come, ad esempio, la gestione delle transazioni, semplificando notevolmente lo sviluppo e la verifica dell'applicazione stessa. Il supporto all'AOP fornito da *Spring* permette di aumentare le funzionalità degli oggetti definiti all'interno del container con i più disparati aspetti: oltre al già citato supporto alle transazioni, *Spring* mette a disposizione, tra i tanti, efficienti meccanismi per l'object pooling, la gestione di oggetti locali ai thread (thread locals), e il logging.
Il framework *Spring* si rivela uno strumento molto flessibile per garantire il rispetto del principio di inversione delle dipendenze; consentendo di esplicitare le dipendenze fra gli oggetti contenuti, risolve il principale problema prodotto dall'accoppiamento debole: rintracciare la realizzazione di una particolare interfaccia.

5.2 Progettazione dell'Interfaccia di Gestione

In primo luogo è opportuno affrontare la progettazione delle informazioni con cui è possibile istruire il componente: infatti le decisioni affrontate in questa sede possono impattare largamente sul proseguio della progettazione. Si ricorda che nella sezione 2.2 era stata evidenziata la necessità, al fine di garantire la corretta operatività del componente, di una interfaccia di gestione attraverso cui fosse possibile configurare il componente in modo da renderlo utilizzabile.
La piattaforma di sviluppo scelta, come riportato nella sezione 2.3, è rappresentata dalla specifica *Java Business Integration*, descritta nel capitolo 4. L'interfaccia di gestione proposta è concettualmente indipendente da tale specifica, dal momento che altre soluzioni offrono meccanismi analoghi per realizzare i medesimi requisiti, ma, al fine di rendere chiara l'esposizione, si è scelto di usare i termini e le definizioni proprie dell'ambiente JBI.

La caratteristica fondamentale che ogni *Enterprise Service Bus* offre è la necessità, al fine di rendere un componente attivo, di effettuare il *deployment* di una *Service Unit*. Tale meccanismo è funzionale alle necessità del componente in esame, ed è perciò possibile "inviare" al componente, tramite un protocollo standard, un archivio contenente la definizione delle informazioni a lui necessarie per operare.

5.2.1 Modello dei Dati

Dopo aver chiarito come viene istruito il componente, è opportuno affrontare la tipologia delle informazioni che può essere fornita. In particolare il componente accetta la definizione di: comunità, collegamenti ad ESB remoti, servizi importati ed esportati e, infine, *Boundary Services*. Più nel dettaglio:

- Una comunità (*community*) rappresenta un insieme di *Enterprise Service Bus* e ne incapsula le informazioni di sicurezza ed eventualmente altri attributi relativi alla qualità di servizio; si noti che non fornisce i dettagli riguardanti gli ESB contenuti al suo interno.

- Un collegamento ad un esb remoto (*esb-link*) definisce una coppia <nome, URL>, che specifica il nome dell'ESB all'interno della comunità e URL a cui è possibile rintracciarlo.

- Un servizio importato è una rappresentazione locale di un servizio disponibile attraverso un collegamento ad un esb remoto; oltre al nome dell'ESB da cui è importato è necessario specificare il nome dell'interfaccia del servizio.

- Un servizio esportato è accessibile da un'intera comunità, rendendo così possibile la realizzazione di tutte le topologie presentate nella sezione. 2.1.1;

- Un *Boundary Service* consente il trasferimento di un aspetto del contesto della chiamata, secondo quanto affermato nella sezione 3.5.

5.2.2 Formato dei Dati

Dal momento che il formato di rappresentazione dei dati all'interno di un ambiente SOA è XML, possiamo formalizzare le informazioni appena presentate tramite *XML Schema*, riportato per completezza nell'appendice B. Lo schema presentato definisce soltanto il formato del descrittore principale e questo deve essere presente, all'interno dell'archivio inviato al componente, con il nome `mesbic.xsd`. Oltre a tale file devono essere ovviamente presenti tutti gli altri file referenziati dal descrittore, tra cui:

- le definizioni degli schemi di ingresso ed uscita dei *Boundary Services*, così come definiti nella sezione 3.5;
- gli archivi contenenti il codice eseguibile dei *Boundary Service*;
- la definizione dell'interfaccia di ogni servizio importato ed esportato.

5.3 Logica di Instradamento

Dopo aver chiarito le tipologie di informazioni che il componente può ricevere, è opportuno affrontare lo schema di indirizzamento che verrà utilizzato durante il trasferimento delle chiamate da un ESB all'altro: infatti deve essere possibile indirizzare correttamente il servizio presente sull'ESB remoto.
Nella sezione 4.4 si è trattato come JBI affronti il problema dell'instradamento e si è visto come sia possibile effettuarlo specificando il nome dell'interfaccia, del servizio, o dell'endpoint. Tale schema offre molta flessibilità, che può diventare eccessiva nel caso dell'integrazione fra *Enterprise Service Bus*.
L'analisi delle dipendenze fra ESB, come presentata nella sezione 2.1.2, affronta le dipendenze che si vengono a creare nel momento in cui un servizio deve essere condiviso fra più zone amministrative. Al fine di ridurre l'accoppiamento si ritiene perciò non opportuno condividere fra gli ESB di una stessa comunità i dettagli implementativi dei servizi,

ed in particolare il nome dei servizi e degli endpoint. In questo modo un servizio può essere facilmente rilocabile da un ESB all'altro in modo assolutamente trasparente agli altri ESB della comunità.

5.4 Progetto dei Repository

Dopo aver chiarito gli aspetti generici della progettazione del componente è opportuno affrontare la prima tipologia di sotto-sistemi: i repository. Questi rappresentano degli archivi che contengono le informazioni che istruiscono il componente sui servizi importati e esportati, nonché sui *Boundary Services* disponibili. Il progetto di tali archivi è affrontabile in maniera unitaria, poiché poche sono le differenze che intercorrono fra di essi.

I requisiti per questa tipologia di sottosistemi sono molto limitati e, in particolare, si riconducono alla persistenza delle informazioni ed alla transazionalità delle modifiche. Le informazioni memorizzate nei repository devono essere immagazzinate in modo che siano recuperabili efficacemente ed allo stesso tempo persistenti. Per quanto riguarda la transazionalità delle modifiche, questa è resa necessaria dal momento che l'operazione di deployment della *Service Unit* potrebbe fallire e, conseguentemente, invalidare tutte le variazioni portate agli archivi. Per motivi analoghi, anche gli accessi a tali repository devono essere transazionali e presentare sempre uno stato coerente.

L'analisi dei requisiti per gli archivi ha evidenziato la necessità di memorizzare tali informazioni in un database che, in fase progettuale, si considera possa essere integrato nel componente stesso. Infatti all'interno del componente è possibile integrare *Apache Derby* [Derby], un database molto flessibile sviluppato in Java. Memorizzando le informazioni all'interno di *Derby*, è possibile garantire la necessaria atomicità delle modifiche.

5.5 Caricamento Dinamico dei Boundary Services

Il componente integratore deve accettare, per i motivi esposti nel capitolo 2, molteplici *Boundary Services* al suo interno: questa tipologia di informazioni si distingue dalle altre accettabili poiché, al contrario delle definizioni dei servizi importabili ed esportabili, implica necessariamente il caricamento e lo scaricamento a tempo di esecuzione di un programma. Per questo motivo all'interno della *Service Unit* è necessario collocare anche gli archivi di codice, i *Jar*, che compongono il BS identificato dal descrittore.
L'ambiente di riferimento, *Java*, consente di effettuare il caricamento dinamico del codice, ma offre delle API molto complesse (ma flessibili) per realizzarlo. Al fine di semplificare questa operazione, si è ritenuto necessario adottare la libreria JCL [Xeus], che offre delle API estremamente semplici per caricare dinamicamente codice eseguibile nell'ambiente *Java*.

5.6 I Generic Service Provider e Consumer

I *Generic Service Provider* e *Consumer* sono i sotto-sistemi tramite cui il componente comunica con l'*Enterprise Service Bus* sul quale è installato. Invece che concentrarsi sul flusso della chiamata, argomento che risulterebbe molto specifico dell'ambiente scelto, si ritiene particolarmente importante esplorare il modello di concorrenza adottato dal componente, ovvero le modalità con cui vengono gestiti più trasferimenti in contemporanea.
L'ambiente *Java*, come tutti i sistemi moderni, adotta un modello a processi leggeri, i *Thread*, che condividono una memoria comune. I *Thread* sono una risorsa limitata all'interno del sistema: se ne possono cioè creare un numero finito, e perciò, al fine di evitare il blocco di un sistema, le applicazioni tendono ad utilizzarne un insieme limitato, chiamato *Pool*, per svolgere i propri compiti. Inoltre tale tecnica au-

menta anche le prestazioni, dal momento che creare un *Thread* ha un suo costo in termini di tempo di esecuzione.
Il modello imposto da JBI si basa ampiamente sulle solide fondamenta fornite da *Java* e offre un valido insieme di API per inviare e ricevere messaggi sull'ESB in modo concorrente. Purtroppo però tali API sono state progettate considerando che uno scambio di messaggi, i MEP di cui si è parlato nella sezione 4.3, sia rapido, ovvero non si invochino in modo sincrono operazioni di lunga durata.
In particolare il modello che viene proposto da JBI è ad attesa passiva: infatti, in un generico scambio di messaggi, normalmente vi è un thread che attende una risposta. Quello che si vorrebbe fosse realizzabile riguarda la necessità di rilasciare le risorse necessarie a mantenere attivo lo scambio di messaggi, eventualmente scrivendo delle informazioni su disco, per poterle poi recuperare nel momento in cui questo si riattiva.
Tale scelta semplifica lo sviluppo dei componenti ma, purtroppo, ha anche un impatto negativo sulle prestazioni, poiché:

1. i *Thread*, seppur leggeri, hanno un costo di creazione e perciò si tende a riusare i *Thread* già creati, inoltre non è possibile crearne un numero illimitato e perciò si tende ad usarne un *Pool*;

2. se l'invio di un messaggio all'interno di uno scambio impiega troppo tempo a completare, il *Thread* viene mantenuto sospeso in attesa che questo completi (altrimenti sarebbe tornato velocemente nel proprio *Pool*);

3. se il messaggio deve essere trasmesso sulla rete, e magari attraversare alcuni ESB, allora accumulerà sicuramente ritardo;

4. se il sistema normalmente ha molti *Thread* sospesi e pochi operativi, allora probabilmente deve avere un *Pool* di *Thread* numeroso.

Il ragionamento appena esposto spiega come il *Pool* debba essere sufficientemente ampio per garantire che vi siano sempre dei *Thread* operativi, ma questa esigenza si scontra con i limiti fisici dell'elaboratore:

ogni *Thread* occupa dello spazio in memoria e sono una risorsa limitata. Stabilire il giusto numero di *Thread* nel *Pool* è un compito che dipende molto dal caso in esame, e può dipendere sia dal carico sia dall'hardware su cui l'*Enterprise Service Bus* andrà ad operare.

Dimensionare tale parametro è sicuramente un problema difficile e JBI non offre alcun suggerimento: in alcuni esempi dove si sono progettati componenti a basso carico viene utilizzato un solo Thread e le richieste vengono serializzate. Al contrario, i *binding component* che consentono la fruizione di servizi remoti offrono la possibilità di configurare a piacimento la dimensione dei *Pool*.

Per garantire delle buone prestazioni ed un consumo di risorse limitato si è proposta una realizzazione abbastanza innovativa, basata sulla disponibilità all'interno dell'ambiente di riferimento di un *Pool* di *Thread* a dimensioni variabili. È possibile imporre a tale *Pool* soltanto un numero massimo di *Thread* ed un timeout: in particolare viene aggiunto un *Thread* se vi sono compiti da eseguire e non ve ne sono di disponibili, mentre viene fermato se trascorre un timeout senza che riceva compiti da eseguire. Si è volutamente tralasciato di specificare ciò che i *Thread* andranno ad eseguire, poiché questo rappresenta il grado di libertà che si sfrutterà per la progettazione del sistema.

Il compito che debbono svolgere il *Generic Service Provider* e il *Generic Service Consumer* riguarda il trasferimento del messaggio dall'ESB locale all'ESB remoto e viceversa; tale compito comincia ovviamente con l'accettazione del messaggio da una delle due possibili sorgenti. L'accettazione del messaggio è bloccante, nel senso che il Thread in esecuzione è fermo finché non è disponibile un messaggio da processare.

Per garantire un buono sfruttamento delle risorse disponibili è opportuno separare le due fasi e gestirle separatamente: in particolare vi saranno due *Thread* che provvedono all'accettazione dei messaggi (uno per sorgente), e un ulteriore *Thread* per ogni scambio di messaggi correntemente in atto. Compatibilmente con le dimensioni massime del *Pool*, infatti, ad ogni scambio di messaggi viene assegnato un *Thread*, e al termine dello scambio il *Thread* ritorna disponibile all'interno del

Pool, minimizzando così le risorse utilizzate.

5.7 Progetto del Remote Connector

Il *Remote Connector* è il componente di più facile realizzazione e, tralasciando la trasmissione sulla rete della chiamata, è opportuno definire in questa sede soltanto le problematiche relative all'interfaccia esposta al meccanismo di traslazione del contesto. Infatti, il *Remote Connector* deve garantire che, in base a quanto esposto nella sezione precedente, sempre lo stesso *Thread* riceva il risultato dello scambio di messaggi, il che può anche essere abbastanza complicato (vedasi sez. 4.3). Più dettagliatamente, per consentire il corretto svolgimento dello scambio di messaggi, è opportuno che il *Remote Connector* attribuisca un identificativo di correlazione locale fra gli invii e le risposte, e trasmetta tale identificativo al pari remoto. Le risposte che questi invierà dovranno contenere tale identificativo che servirà per recuperare il *Thread* locale. Il secondo compito del *Remote Connector* riguarda la gestione delle politiche di sicurezza del collegamento fra gli *Enterprise Service Bus*: infatti, come si è anticipato nella sezione 5.2, ad ogni comunità sono associate le informazioni necessarie alla cifratura del messaggio. In particolare, visto il forte legame che questo componente introduce, si ritiene eccessivamente complessa una soluzione di crittografia asimmetrica e perciò si è optato per una più semplice soluzione basata sulla crittografia simmetrica. Conseguentemente, all'interno delle informazioni relative alla comunità sarà sufficiente introdurre una *passphrase* che verrà poi usata come chiave di cifratura.

5.8 Il Prototipo

In questo capitolo si è discusso come sia stato possibile realizzare in breve tempo un prototipo capace di integrare fra loro ESB tutti conformi al medesimo standard, JBI. Le problematiche poste da tale realizzazione possono essere generiche, come ad esempio il formato dei dati,

oppure derivanti dell'ambiente, come il modello di concorrenza. Da ultimo si vuole puntualizzare come il componente realizzato voglia essere soltanto una semplice implementazione necessaria a mostrare la fattibilità dell'approccio seguito per il trasferimento del contesto, e non aspiri quindi ad avere né la stabilità né le prestazione di una soluzione commerciale.

Capitolo 6

Progettazione del Meccanismo di Traslazione

Nel capitolo 3 è stato proposto un modello innovativo per caratterizzare i *Boundary Services* nel modo più opportuno, e tale modello offre un valido punto di partenza per la realizzazione di un meccanismo per la traslazione del contesto che sia al contempo robusto e flessibile.
La proposta originale effettuata nella sezione 3.5 pone però degli interessanti problemi che è sicuramente opportuno identificare e risolvere per giungere poi alla progettazione del meccanismo di traslazione. In questo capitolo verranno dapprima esposti tali problemi evidenziandone una soluzione e poi, in ultimo, verrà proposto un possibile schema di funzionamento per il meccanismo di traslazione.

6.1 Ragionamento Automatico per la Composizione dei Boundary Services

L'approccio presentato nel capitolo precedente pone un problema notevole per quanto riguarda la composizione dei *Boundary Services* in un sistema funzionante. Infatti, per quanto affermato nella sezione 3.5, questi particolari servizi vengono forniti al meccanismo per la traslazione del contesto attraverso una tripla:

(nome, namespace di ingresso, namespace di uscita)

L'insieme dei BS si trova quindi ad essere in una forma completamente disaggregata, lontana dalla struttura organizzata proposta nella sezione 3.5. Il primo problema che il meccanismo di traslazione è chiamato a risolvere è rappresentato dalla suddivisione di tale amalgama di servizi in un insieme ben strutturato in piani e livelli. L'algoritmo per l'organizzazione dei BS deve perciò:

1. individuare i BS che originano i piani, ovvero quelli con il namespace di ingresso nullo;

2. costruire la pila di trasformazione per ogni piano, tenendo presente che è possibile avere BS con stesso linguaggio di ingresso e di uscita, e in questo caso tali BS hanno la precedenza ad essere inseriti nella pila;

3. identificare i casi particolari in cui sotto un particolare BS possono essere collocati differenti servizi e fornire poi al meccanismo per la traslazione del contesto tale informazione.

6.2 Implementazione in Linguaggio Prolog

L'elaborazione appena esposta richiede un algoritmo sicuramente complesso, perciò si ritiene opportuno descriverlo non attraverso uno pseudo-codice o un diagramma di flusso, ma piuttosto utilizzando un linguaggio proprio dell'intelligenza artificiale: il *Prolog*. Tale linguaggio, progettato negli anni '70 da Philippe Roussel [Col93], è dichiarativo e si poggia sulla logica dei predicati del primo ordine.
La motivazione principale per l'adozione di tale linguaggio in questa sede riguarda la necessità di poter effettuare il cosiddetto *backtracking*: ovvero, se il programma effettua una "scelta sbagliata", l'interprete del linguaggio può ritrattare alcune decisioni e prendere altre vie (che comunque devono essere espresse in fase di programmazione). La suddetta caratteristica consente la prototipazione di complessi algoritmi ricorsivi in modo rapido e compatto, come richiesta dell'algoritmo di cui si esporrano le caratteristiche nel seguito di questa sezione.

Dal momento che i *Boundary Services* sono identificati con una tripla, è possibile formalizzare tale rappresentazione in *Prolog*:

```
boundary(NAME, LANG_IN, LANG_OUT)
```

Oltre alla modellazione dei BS, è opportuno progettare anche il formato del risultato che deve poter rappresentare:

- un insieme di piani indipendenti fra loro;
- un insieme di BS, identificati per nome, da invocare in sequenza all'interno di ogni piano;
- una realizzazione di un OR esclusivo, in cui un piano potrebbe avere due differenti punti di uscita (nel caso in cui, come accennato sopra, vi sia più di un BS che possa assumere un particolare posto nella pila).

Considerando a titolo esemplificativo i seguenti *Boundary Services* (si ricorda che il linguaggio di ingresso potrebbe essere nullo)

```
boundary(zero, null, a).
boundary(first,a,b).
boundary(second,b,b).
boundary(third,b,c).
boundary(fourth, null, e).
boundary(fifth,b,d).
```

si ha che la notazione scelta per indicare la suddivisione in piani è riportata nel listato seguente (osservabile anche in figura 6.1):

```
[plane([
  service(zero),
  service(first),
  service(second),
  xor([
    plane([service(third)]),
    plane([service(fifth)])
  ])]),
plane([service(fourth)])]
```

Si noti come le biforcazioni all'interno di un piano, rappresentate dall'OR esclusivo di cui si era espressa la necessità, vengano rappresentate come sotto-piani.

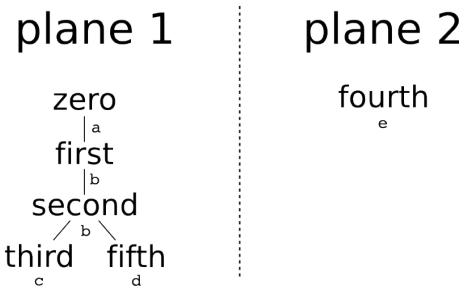


Figura 6.1: Esempio di suddivisione in piani e livelli.

Anche se non necessario al proseguio della trattazione, si ritiene opportuno riportare l'intero algoritmo per la suddivisione in piani e livelli nell'appendice A.

6.3 Struttura dati a supporto della Traslazione

Dopo aver esaminato come determinare la suddivisione in piani e livelli necessaria al meccanismo di traslazione, è opportuno analizzare il secondo problema: la gestione della successione delle chiamate ai Boundary Services in modo da non impattare in maniera negativa sulle prestazioni complessive. Tale problema è riconducibile alla realizzazione di una struttura dati per contenere i piani e livelli che consenta una facile e veloce esplorazione.
La struttura dati dovrà rispondere ai seguenti requisiti:

1. essere indipendente dalla realizzazione dei *Boundary Services*, in particolare l'interfaccia che questi offrono per l'invocazione deve essere completamente nascosta alla struttura dati;

2. essere indipendente dal tipo di dato scambiato fra i BS, che si assume univoco (nei requisiti per il meccanismo di traslazione era un documento XML);

3. consentire l'invocazione sia "dall'alto" che "dal basso", ovvero si deve poter ripristinare correttamente il contesto;

4. garantire la corretta gestione degli OR esclusivi, verificando quale sia il sotto-piano attivo per quel particolare contesto.

Al fine di garantire il requisito 1, è necessario che la gestione dell'invocazione dei BS venga completamente incapsulata da una interfaccia *InvokeService*: tale interfaccia consente di gestire in maniera completamente indipendente dalla struttura dati le problematiche di invocazione quali, ad esempio, la gestione del contesto corrente ed il recupero della classe che realizza il BS. L'unica informazione riguardante i servizi che verrà memorizzata all'interno della struttura dovrà essere il nome univoco dei servizi stessi, necessario a gestire correttamente l'invocazione. Oltre all'invocazione l'interfaccia *InvokeService* dovrà anche consentire la determinazione, dato il particolare contesto a cui si riferisce, dello stato di attivazione di ogni BS.
Per quanto riguarda l'indipendenza dal tipo di dato scambiato fra i *Boundary Services* (requisito 2), si ritiene opportuno l'utilizzo dei generici (o talvolta chiamati *template*) che consentono di scrivere algoritmi basati su tipi specificabili in fase di esecuzione. In questo modo è ovviamente possibile modificare il tipo di dato scambiato senza dover modificare alcunché.
Il requisito 3 impone che sia possibile invocare i BS attraverso la struttura sia in fase di invio dell'invocazione remota che in fase di ricezione della stessa. Per supportare correttamente tale requisito si ritiene necessario adottare il pattern Visitor [GoF95], il quale consente di effettuare molte elaborazioni diverse su una struttura dati complessa, minimizzando la logica di controllo contenuta all'interno della stessa.
Infine il requisito 4 implica che, nel momento in cui durante l'invocazione si incontri un OR esclusivo, sarà opportuno verificare quale dei rami sia attivo e percorrerlo. Tale controllo dovrà essere effettuato tramite l'*InvokeService*.

6.4 Progettazione della Struttura Dati

La struttura dati proposta è una rappresentazione astratta dei piani e non ha nessuna capacità di esecuzione: inoltre la maggior parte delle classi servono soltanto a fornire una rappresentazione del piano, senza contenere alcuna informazione né comportamento. In particolare

ogni oggetto presente nella struttura fornisce un generico metodo di navigazione per rintracciare i "figli": una generica realizzazione della struttura è definita come un grande albero genealogico (al contrario) dove il padre di tutti i servizi è l'oggetto *Planes* che però può avere sotto di lui soltanto dei figli *Plane*. Questi ultimi, a loro volta, possono avere soltanto figli *XorLayer* e *ServiceLayer*, i quali hanno lo stesso significato esposto nella sezione 6.2; i *ServiceLayer* possono avere come figli altri *ServiceLayer* oppure degli *XorLayer*. Come espresso in altra maniera attraverso i termini Prolog, gli *XorLayer* possono avere come figli dei *Plane*, rendendo così possibile la suddivisione in sotto-piani dei piani stessi. Per completezza si riporta il diagramma UML delle classi appena esposte in figura 6.2; si noti l'utilizzo dell'interfaccia *VisitableElement* che definisce un generico elemento della struttura.

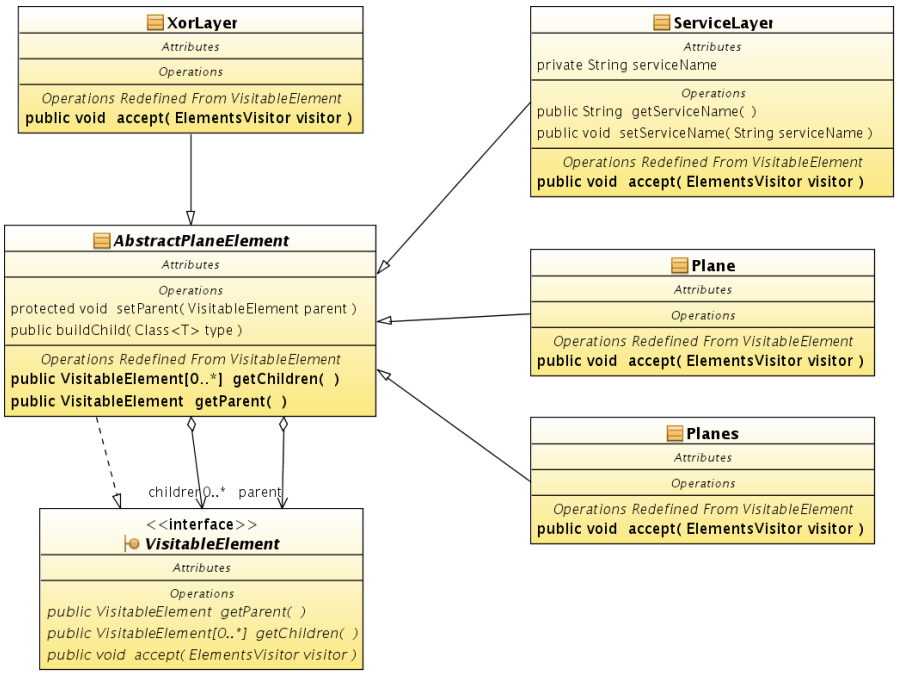


Figura 6.2: Diagramma UML della struttura dati.

Nel paragrafo precedente si è affermato che gli oggetti componenti la struttura non contengono comportamento: in realtà l'unico comportamento specificato in ogni oggetto è contenuto nel metodo *accept())*, il

quale, come specificato nel pattern *Visitor*, andrà a chiamare il metodo opportuno dell'interfaccia *ElementVisitor* (fig. 6.3). Secondo questo pattern, un oggetto *ServiceLayer* che riceve una chiamata a tale metodo deve chiamare a sua volta il metodo *visit(ServiceLayer)* dell'oggetto *Visitor*, secondo una tecnica che prende il nome di *double-dispatch*. Tale tecnica fornisce un ottimo mezzo per accedere alla struttura in modo coerente con i principi SOA: in base a questa, infatti, i dati devono contenere poco comportamento, mentre l'elaborazione deve essere svolta altrove.

I *Visitor* (fig. 6.3), grazie l'utilizzo dei metodi *ElementsVisitor.visit()*, possono essere usati per specificare il comportamento relativo ad ogni entità individuata nel diagramma 6.2. In particolare, come veniva sopra evidenziato, è opportuno che vi siano due modalità di accedere ai dati, che si esplicano in due *Visitor*: il *TopDownInvokingVisitor* e il *BottomUpInvokingVisitor*. Queste due classi si appoggiano ad un'interfaccia *InvokingService*, lasciata volutamente indipendente dal meccanismo di traslazione del contesto.

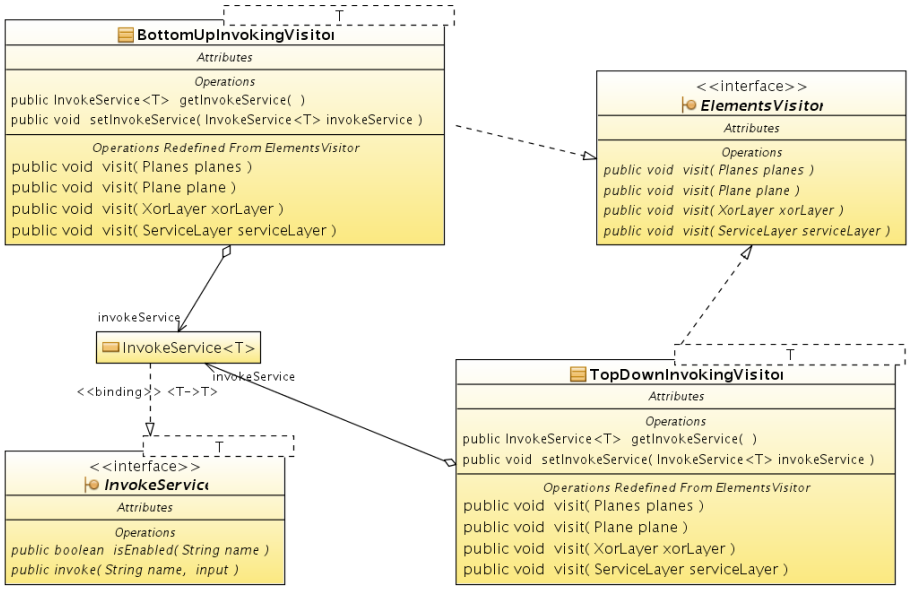


Figura 6.3: Diagramma UML dei *Visitor*.

6.5 Architettura del Meccanismo di Traslazione

Il meccanismo di traslazione deve sfruttare in modo coerente le specifiche sopra definite al fine di realizzare una corretta traslazione del contesto. In particolare, al fine di essere operativo, dovrà costruire la struttura dati ed in questo modo inizializzarsi. Non a caso si è affrontata la presentazione dell'algoritmo per la suddivisione in piani e livelli citando il linguaggio *Prolog*: infatti è disponibile un'implementazione leggera e facilmente portabile, realizzata dal gruppo di ricerca aliCE dell'Università di Bologna: *tuProlog* [Den01].
La disponibilità di un interprete *Prolog* facilmente accessibile da un ambiente *Java* come quello di riferimento semplifica molto la procedura di inizializzazione del meccanismo di traslazione, infatti (fig. 6.4):

1. inizializza l'interprete *tuProlog* con la regole per la suddivisione in piani e livelli, come espresso nell'appendice A,
2. aggiunge all'interno dell'interprete le definizioni dei Boundary Services disponibili,
3. determina la suddivisione in piani e livelli nella forma espressa nella sezione 6.2,
4. trasforma tale rappresentazione nella definitiva struttura dati necessaria al componente per operare.

Durante la normale operatività il meccanismo opera in modo lineare, e, come si può vedere in figura 6.5, è possibile suddividere in fasi il processo tramite cui un messaggio viene trasformato al fine di poterlo inviare sulla rete:

1. il *Context Translation Mechanism* riceve il contesto;
2. il *Context Translation Mechanism*, basandosi sulla struttura dati precedentemente determinata e sull'archivio dei *Boundary Service*, trasforma il contesto in modo che possa essere trasmesso

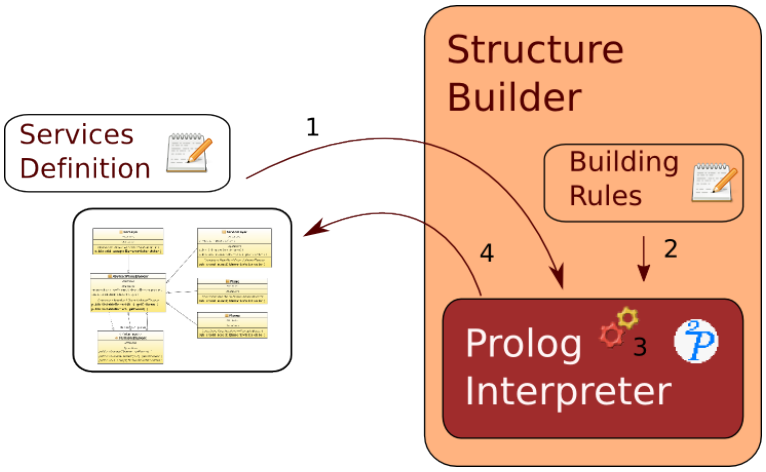


Figura 6.4: Fasi della costruzione della struttura dati.

sulla rete (basandosi sul *TopDownInvokingVisitor*), realizzando un pacchetto che è formato da dai dati di contesto e dalla chiamata;

3. il *Context Translation Mechanism* passa il pacchetto al *RemoteConnector* che si occupa della trasmissione vera e propria;

4. il *RemoteConnector* riceve la risposta del pari remoto e la passa al *Context Translation Mechanism*;

5. il *Context Translation Mechanism* trasforma il pacchetto ricevuto e ricrea il contesto;

6. il *Context Translation Mechanism* fornisce la risposta ed il contesto al mittente originario.

Il *Context Translation Mechanism* è sostanzialmente responsabile di due compiti: il passaggio dal contesto al pacchetto e viceversa. È quindi possibile incapsulare questi comportamenti sotto un'apposita interfaccia in modo tale da rendere il componente in esame completamente indipendente sia dalla realizzazione dell'archivio dei *Boundary Services* sia dal formato dei messaggi scambiati fra questi ultimi. L'unico aspetto rilevante di tale interfaccia riguarda il formato del pacchetto scambiato sulla rete, cioè il risultato della traslazione.

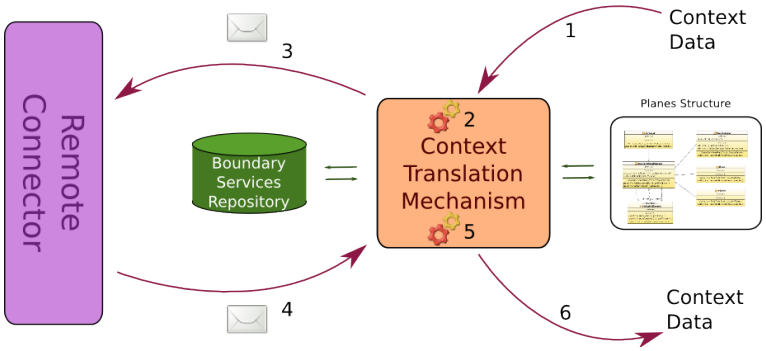


Figura 6.5: Fasi della traslazione del contesto.

6.6 Risultato della Traslazione

Il risultato dell'elaborazione del meccanismo di traslazione del contesto è un pacchetto che deve contenere sia il messaggio vero e proprio che le informazioni di contesto. Dal momento che questi vengono elaborati in modo analogo e producono un risultato in formato XML, sarà secondo tale specifica che tali informazioni dovranno comparire all'interno del pacchetto. Siccome il fondamento di un documento XML sono gli elementi nella forma "<nomeElemento></nomeElemento>", ogni elemento all'interno del pacchetto deve rappresentare una parte specifica prodotta da un piano di *Boundary Services*. È importante però che ogni elemento contenga la definizione del suo namespace di appartenenza [W3C06], poiché è necessario al fine di garantire la corretta invocazione dei piani durante la ricezione del pacchetto.
Il listato che segue presenta il formato del pacchetto definito attraverso *XML Schema*:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <xs:schema version="1.0" targetNamespace="http://mesbic.mc/context/1.0"
3    xmlns:tns="http://mesbic.mc/context/1.0"
4    xmlns:xs="http://www.w3.org/2001/XMLSchema">
5
6    <xs:element name="packet" type="tns:packet"/>
7
8    <xs:complexType name="packet">
9      <xs:sequence>
10       <xs:any processContents="strict" namespace="##other"
11         minOccurs="0" maxOccurs="unbounded"/>
12     </xs:sequence>
```

```
13      </xs:complexType>
14    </xs:schema>
```

Nel listato XML appena mostrato è possibile vedere l'utilizzo di "<xs:any/>", elemento che garantisce la possibilità di inserire all'interno del pacchetto elementi non specificati nello schema sopra esposto. L'attributo "processContents=strict" specifica invece che gli elementi contenuti devono essere definiti all'interno di un namespace [W3C04], rispettando quindi il requisito sopra esposto.

6.7 Considerazioni sulla Scalabilità

Il meccanismo di traslazione proposto deve necessariamente essere efficace e scalabile: infatti, come è stato già precedentemente anticipato, tale meccanismo dovrà affrontare molteplici trasferimenti contemporaneamente. Inoltre potrebbero essere coinvolti un numero elevato di *Boundary Services*, rendendo quindi il piano estremamente complesso. L'algoritmo per la suddivisione, non dovendo essere eseguito ad ogni trasferimento, ma una volta soltanto, non impatta, se non marginalmente, le prestazioni di trasferimento. Impatta marginalmente poiché tale suddivisione deve essere effettuata all'interno dell'ESB vero e proprio ed in tal caso vengono sottratte risorse al trasferimento delle chiamate. In ogni caso è comunque necessario valutare le prestazioni di tale algoritmo al fine di consumare il minor numero di risorse possibile.
È invece importante esaminare se il meccanismo di invocazione dei *Boundary Services* possa garantire una realizzazione efficiente. La variabilità della struttura ha infatti imposto, per garantire una realizzazione semplice, l'utilizzo del pattern *Visitor*, nonostante questo sia basato sulla ricorsione: vi saranno quindi un numero di chiamate annidate pari al numero di servizi disponibili. La macchina virtuale su cui si basa *Java* alloca uno spazio di memoria prefissato per memorizzare l'elenco delle chiamate in esecuzione, e perciò è possibile saturarlo. D'altro canto la stessa macchina virtuale offre la possibilità di dimensionare tale parametro secondo le esigenze dell'applicazione che andrà ad eseguire, così che questo problema non limiti la scalabilità della soluzione,

ma obblighi ad un opportuno dimensionamento durante la messa in opera. Anche se il principale problema introdotto dalla tecnica utilizzata non comporta una difficoltà insuperabile, si ritiene comunque opportuno valutare sperimentalmente i risultati ottenibili da un siffatto meccanismo di traslazione.

6.8 Un Componente Riusabile

Il meccanismo di traslazione del contesto introdotto nel capitolo precedente ed approfondito in questo rappresenta un valido esempio di come i principi introdotti dalla SOA possano essere estesi anche alle applicazioni più infrastrutturali. La soluzione proposta riesce a separare in maniera netta la gestione dei principali problemi relativi alla traslazione del contesto, ovvero la suddivisione in piani e livelli e l'invocazione nell'ordine opportuno dei *Boundary Services*, da aspetti assolutamente dipendenti dall'applicazione, come ad esempio il formato dei *Boundary Services* stessi e la struttura del contesto. Il sistema sviluppato si pone perciò come una soluzione coerente con la SOA pronta a risolvere non soltanto il problema dell'integrazione fra *Enterprise Service Bus*, ma anche tutti quei problemi in cui vi è un contesto da trasferire.

Capitolo 7

Risultati Sperimentali

Il vero contributo originale di questo lavoro di tesi consiste nell'analisi e nella progettazione del meccanismo di traslazione del contesto e, come anticipato in quella sede, si ritiene necessario effettuare una serie di verifiche sull'effettiva efficienza dello stesso.
Al fine di valutare correttamente le prestazioni ottenute del meccanismo di traslazione del contesto, così come proposto in questo capitolo, si ritiene opportuno considerare due differenti scenari. Nel primo si analizzerà le tempistiche ottenibili dall'algoritmo di suddivisione in piani e livelli realizzato in prolog, mentre il secondo mira a verificare che il meccanismo di accesso ai piani sia efficiente e scalabile.
I due scenari rappresentano due benchmark veri e propri e, vista l'estrema variabilità di questi ultimi, si ritiene necessario puntualizzare come il dato numerico proposto in questa sede possa cambiare significativamente al variare delle condizioni di carico e del sistema su cui viene prodotto. Anche se il numero in sè non è significativo, la comparazione fra i risultati ottenuti nei varie prove può rappresentare un valido punto di partenza per valutare la fattibilità dell'approccio presentato.
Per completezza si riporta che tutti i dati che verranno presentati in questo capitolo sono stati ottenuti su un portatile *DELL Latitude E6400*, equipaggiato con un *Intel Core 2 Duo P9500* (velocità del clock 2.53GHz, velocità del bus 1066MHz, cache 6MB) e 4GB di RAM. Inoltre le varie prove verranno effettuate basandosi sul framework Japex [Sun08], un ambiente per verificare le prestazioni di componenti Java.

7.1 Benchmark dell'Algoritmo di Suddivisione

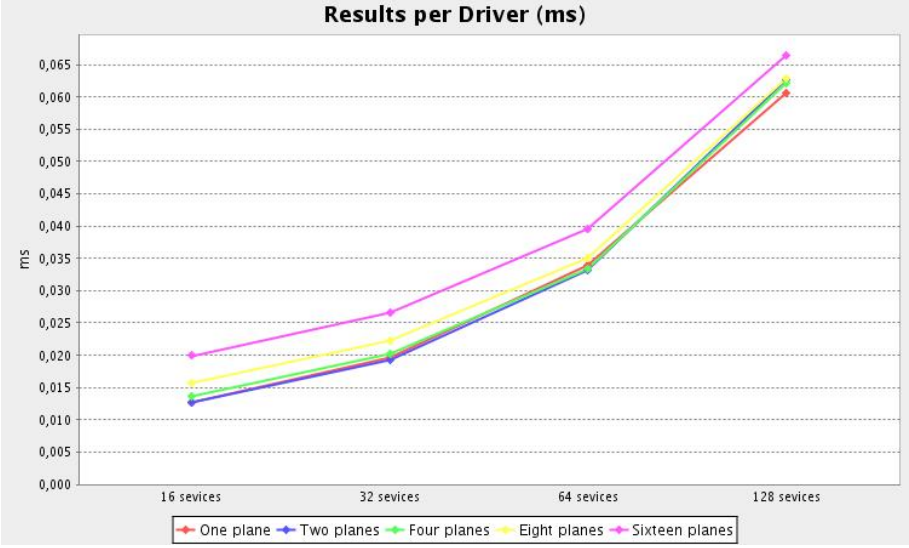
Questo benchmark è stato ottenuto valutando la media di 100 suddivisioni in piani dopo aver fissato il numero di piani voluti e il numero di servizi complessivo. In particolare i numeri di piani che sono stati presi in considerazione sono 1, 2, 4, 8, e 16; in modo analogo i servizi esaminati sono 16, 32, 64, 128, 256, 512, e 1024.
Il grafico (fig. 7.1) mostra perciò sia l'aumentare del tempo necessario per ottenere un risultato sia in base al numero complessivo dei BS sia alla suddivisione degli stessi in piani: è subito evidente come l'aumentare del numero di piani non peggiori significativamente le singole tempistiche su un basso numero di BS, ma piuttosto renda la crescita del tempo all'aumentare del numero dei servizi più esponenziale.
In figura è inoltre possibile vedere come l'algoritmo di suddivisione in piani e livelli non scali linearmente all'aumentare dei servizi, in particolare si può vedere come, nel passaggio da 64 a 128 *Boundary Services*, il tempo necessario per giungere ad un risultato quadruplichi. In ogni caso non si suppone che possano esservi più di qualche centinaio di BS e perciò, visto anche che le tempistiche esaminate sono ben al di sotto del minuto fino a 512 servizi, si ritiene che i risultati ottenuti siano soddisfacenti.

7.2 Benchmark dell'Accesso alla Struttura Dati

Il secondo scenario esaminato vuole valutare l'impatto che le procedure di trasformazione del contesto abbiano sulle performance: in particolare si è voluto analizzare quanto l'accesso ai piani attraverso *Visitor* costi sulle performance totali. A tale scopo i servizi ovviamente non vengono realmente invocati, ma piuttosto si simula un accesso completo alla struttura dati.
Questo benchmark, in modo simile a quello precedente, valuta la media

Figura 7.1: Benchmark dell'algoritmo di suddivisione in piani e livelli

di invocazioni dell'operazione di trasformazione fissando il numero di piani e livelli contenuti nella struttura dati: anche in questo caso sono stati presi in considerazione i numeri di piani 1, 2, 4, 8 e 16, assieme ad insiemi composti da 16, 32, 64, 128, 256, 512 e 1024 *Boundary Services*. Dal momento che le tempistiche esaminate sono estremamente basse la media è stata ottenuta su 10000 invocazioni, è però opportuno notare come le grandezze misurate siano molto variabili dal momento che è difficile misurare con esattezza tempi inferiori al millisecondo.
In figura 7.2 è possibile vedere come le tempistiche siano tutte inferiori al millisecondo, e perciò il ritardo introdotto da tale meccanismo è difficile da notare in una comunicazione distribuita. Inoltre le tempistiche rilevate nell'intervallo esaminato crescono linearmente con l'aumentare dei servizi e ciò fornisce una buona indicazione sulla validità dell'approccio proposto in questa sede.



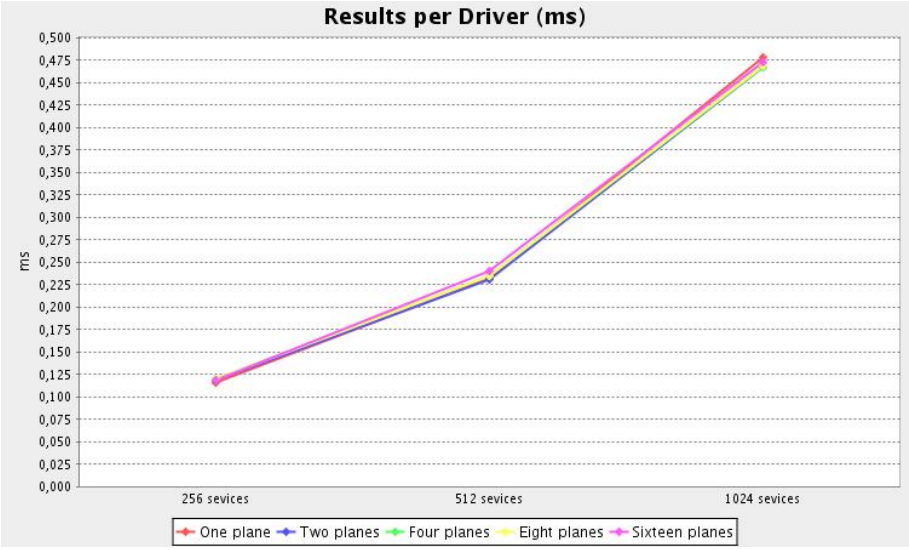


Figura 7.2: Benchmark dell'operazione di marshalling/restoring del contesto.

Conclusione

Nella letteratura disponibile sull'integrazione fra *Enterprise Service Bus* si riteneva che, per realizzare una siffatta soluzione, fosse necessario un grosso investimento, dato il livello di difficoltà posto dal principale problema di progettazione. In questo lavoro di tesi si è in parte smentito questo preconcetto, fornendo una soluzione semplice e, al contempo, teoricamente solida.

Dopo aver attentamente analizzato i requisiti posti dal problema, si è discussa un'estensione ad un modello formale degli anni '70, l'*OSI Reference Model*, capace di garantire una soluzione estremamente flessibile e atta a soddisfarli. Applicando lo strumento formale esaminato, è stato possibile giungere ad una realizzazione che consentisse di garantire sia buone prestazioni che facile estendibilità. Al fine di realizzarla è stato necessario adottare delle tecniche di ragionamento automatico capaci di semplificarne notevolmente la realizzazione. Il prototipo realizzato mostra come la modellazione teorica effettuata sia effettivamente valida e porti a prestazioni interessanti, nonostante quanto sviluppato rimanga pur sempre un prototipo realizzato ad-hoc, che andrebbe necessariamente esteso e collaudato prima di essere messo in produzione.

Volgendo uno sguardo al futuro sono auspicabili ulteriori applicazioni del modello presentato: il problema che si vuole risolvere è infatti comune a molte altre infrastrutture software che ora adottano soluzioni di gran lunga più complesse. Infine è opportuno notare come questo lavoro, seppur non volendo affrontare direttamente l'argomento, abbia mostrato come l'adozione dei principi proposti dalla *Service Oriented Architecture* possa essere estesa anche alle stesse infrastrutture, e già

da ora è possibile intuire come l'adozione di tali principi si estenderà ancora.

Appendice A

Algoritmo per la Suddivisione in Piani e Livelli

Nel listato seguente è possibile osservare il programma *Prolog* che suddivide i *Boundary Service* in livelli e piani:

```
1  build(R) :- build(R, []).
2
3  build([ T | L ], USED_IN) :-
4          build_plane(T, null, USED_IN, USED_OUT),
5          build(L, USED_OUT).
6
7  build([], USED_IN) :- not build_plane(_, null, USED_IN, _).
8
9  build_plane(plane(L), LANG_IN, USED_IN, USED_OUT) :-
10         build_service(L, LANG_IN, USED_IN, USED_OUT),
11         not L = [].
12
13 build_service([ service(NAME) | L ], LANG, USED_IN, USED_OUT) :-
14         not LANG = null,
15         find_layer(USED_IN, NAME, LANG, LANG), !,
16         build_service(L, LANG, [ NAME | USED_IN ], USED_OUT).
17
18 build_service([ xor(L) ], LANG, USED_IN, USED_OUT) :-
19         not LANG = null,
20         find_layers(USED_IN, [ NT | NL ], LANG),
21         not NL = [], !, % check if this list contains at least 2 elements
22         build_xor(L, LANG, USED_IN, USED_OUT, [ NT | NL ]).
23
24 build_service([ service(NAME) | L ], LANG_IN, USED_IN, USED_OUT) :-
25         find_layer(USED_IN, NAME, LANG_IN, LANG_OUT),
26         build_service(L, LANG_OUT, [ NAME | USED_IN ], USED_OUT).
27
28 build_service([], LANG, USED_IN, USED_IN) :-
```

```
29          not find_layer(USED_IN, _, LANG, _).
30
31  build_xor([ ST | SL ], LANG, USED_IN, USED_OUT, [ NT | NL ]) :-
32          append(NL, USED_IN, C_USED_IN),
33          build_plane(ST, LANG, C_USED_IN, C_USED_OUT),
34          !, build_xor(SL, LANG, [ NT | USED_IN ], X_OUT, NL),
35          append(C_USED_OUT, X_OUT, USED_OUT).
36
37  build_xor([], LANG, USED_IN, USED_IN, []).
38
39  find_layer(USED_IN, NAME, LANG_IN, LANG_OUT) :-
40          boundary(NAME, LANG_IN, LANG_OUT),
41          not member(NAME, USED_IN).
42
43  find_layers(USED_IN, [T | L], LANG_IN) :-
44          find_layer(USED_IN, T, LANG_IN, _),
45          find_layers([ T | USED_IN ], L, LANG_IN).
46
47  find_layers(_,[],_).
```

Appendice B

Formato del Descrittore del Componente

Per completezza si riporta in questa appendice la definizione in *XML Schema* del formato del descrittore accettato dal componente integratore (sez. 5.2.2):

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <xs:schema version="1.0" targetNamespace="http://mesbic.mc/descriptors/1.0"
3    xmlns:tns="http://mesbic.mc/descriptors/1.0"
4    xmlns:xs="http://www.w3.org/2001/XMLSchema">
5
6    <xs:element name="mesbic">
7      <xs:complexType>
8        <xs:sequence>
9          <xs:element name="boundary-service" type="tns:boundary-service"
10           minOccurs="0" maxOccurs="unbounded"/>
11         <xs:element name="community" type="tns:community"
12           minOccurs="0" maxOccurs="unbounded"/>
13         <xs:element name="esb-link" type="tns:esb-link"
14           minOccurs="0" maxOccurs="unbounded"/>
15         <xs:choice minOccurs="0" maxOccurs="unbounded">
16           <xs:element name="exported-service" type="tns:exported-service"/>
17           <xs:element name="imported-service" type="tns:imported-service"/>
18         </xs:choice>
19       </xs:sequence>
20     </xs:complexType>
21   </xs:element>
22
23   <xs:complexType name="boundary-service">
24     <xs:all>
25       <xs:element name="class" type="xs:string"/>
26       <xs:element name="class-path">
```

```
27          <xs:complexType>
28            <xs:sequence>
29              <xs:element name="path-element" type="xs:string"
30              maxOccurs="unbounded"/>
31            </xs:sequence>
32          </xs:complexType>
33        </xs:element>
34        <xs:element name="input-xsd-path" type="xs:string"/>
35        <xs:element name="name" type="xs:string"/>
36        <xs:element name="output-xsd-path" type="xs:string"/>
37      </xs:all>
38    </xs:complexType>
39
40    <xs:complexType name="community">
41      <xs:sequence>
42        <xs:element name="name" type="xs:string"/>
43        <xs:element name="passphrase" type="xs:string"/>
44      </xs:sequence>
45    </xs:complexType>
46
47    <xs:complexType name="esb-link">
48      <xs:all>
49        <xs:element name="name" type="xs:string"/>
50        <xs:element name="uri" type="xs:string"/>
51      </xs:all>
52      <xs:attribute name="community" type="xs:string" use="required"/>
53    </xs:complexType>
54
55    <xs:complexType name="exported-service">
56      <xs:all>
57        <xs:element name="endpoint-name" type="xs:string" minOccurs="0"/>
58        <xs:element name="interface-name" type="xs:QName"/>
59        <xs:element name="service-name" type="xs:QName" minOccurs="0"/>
60        <xs:element name="wsdl-path" type="xs:string" minOccurs="0"/>
61      </xs:all>
62      <xs:attribute name="community" type="xs:string" use="required"/>
63    </xs:complexType>
64
65    <xs:complexType name="imported-service">
66      <xs:all>
67        <xs:element name="endpoint-name" type="xs:string"/>
68        <xs:element name="interface-name" type="xs:QName"/>
69        <xs:element name="service-name" type="xs:QName"/>
70        <xs:element name="wsdl-path" type="xs:string"/>
71      </xs:all>
72      <xs:attribute name="esb-link" type="xs:string" use="required"/>
73    </xs:complexType>
74  </xs:schema>
```

Bibliografia

[Bor04] Bernhard Borges, Ali Arsanjani, Kerrie Holley, "*Service-Oriented Architecture*", InfoManagement Direct, Novembre 2004

[Col93] Alain Colmerauer, Philippe Roussel, "*History of Programming Languages*", 1993.

[Day83] John D. Day, Hubert Zimmermann, "*The OSI Reference Model*", Proceedings of the IEEE, vol. 71, no. 12, dicembre 1983

[Den01] Enrico Denti, Andrea Omicini, Alessandro Ricci, "*tuProlog: A Light-weight Prolog for Internet Applications and Infrastructures*", Springer, 2001.

[Derby] Apache Foundation, "*Apache Derby*", http://db.apache.org/derby.

[Fow04] Martin Fowler, "*Inversion of Control Containers and the Dependency Injection pattern*", http://martinfowler.com/articles/injection.html, Gennaio 2004.

[Fow07] Martin Fowler, "*Mocks Aren't Stubs*", http://martinfowler.com/articles/mocksArentStubs.html, Gennaio 20047.

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "*Design Patterns: Elements of Reusable Object-Oriented Software*", Addison Wesley, 1995.

[Her01] D. Herzberg, A. Marburger, "*The use of layers and planes for architectural design ofcommunication systems*", Proceedings of the

Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2001.

[IET05] Internet Engineering Task Force,"*Uniform Resource Identifier (URI): Generic Syntax*", Gennaio 2005.

[Jur06] Matjaz B. Juric, "*Business Process Execution Language for Web Services*", Packt Publishing, 2006.

[Kic97] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar, "*Aspect-Oriented Programming*", 1997.

[Kee04] Martin Keen, Susan Bishop, Alan Hopkins, Sven Milinski, Chris Nott, Rick Robinson, Jonathan Adams e Paul Verschueren, "*Patterns: Implementing an SOA with the Enterprise Service Bus*", IBM Redbooks, 2004.

[Kee05] Martin Keen, Jonathan Bond, Jerry Denman, Stuart Foster, Stepan Husek, Ben Thompso e Helen Wylie, "*Patterns: Integrating Enterprise Service Buses in a Service-Oriented Architecture*", http://www.redbooks.ibm.com, 2005.

[OAS07] OASIS, "*Web Services Business Process Execution Language Version 2.0*", Aprile 2007.

[Oli06] Rui Oliveira, *The Power of Cobol*, BookSurge Publishing, 2006.

[Spr09] Spring Source, "Spring Framework", *http://www.springframework.org*, 2009.

[Sun06] Sun Microsystems, "*Java Management Extensions (JMX) Specification, version 1.4*", Novembre 2006.

[Sun08] Sun Microsystems, "*Japex*", https://japex.dev.java.net/, 2008.

[Ten05] Ron Ten-Hove, Peter Walker, "*Java Business Integration 1.0*", Sun Microsystems, 2005.

[Wall08] Craig Walls, Ryan Breidenbach, *Spring in Action*, Manning Pubblications, 2008.

[W3C04] World Wide Web Consortium, "XML Schema Part 0: Primer Second Edition", Ottobre 2004.

[W3C06] World Wide Web Consortium, "*Namespaces in XML 1.0 (Second Edition)*", Agosto 2006.

[W3C07a] World Wide Web Consortium, "*Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*", Giugno 2007.

[W3C07b] World Wide Web Consortium, "*Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts*", Giugno 2007.

[W3C08] World Wide Web Consortium, *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, Novembre 2008.

[Xeus] Xeus Technologies, "*Jar Class Loader*", http://jcloader.sourceforge.net.