# Service Oriented Architecture and Web Services

Note per il corso di Ingegneria del Software
Università di Camerino – Dipartimento di Matematica ed Informatica

Andrea Polini

11 gennaio 2007

Queste note sono un estratto della tesi di Laurea “Creazione ed Integrazione di Web Service da Applicazione Stand-Alone” discussa presso l’Università di Pisa da Daniele Bonuccelli nel luglio 2005.

# Capitolo 1

# Enterprise Application Integration

L'obiettivo principale di ogni azienda è cercare di rimanere sempre al passo con i tempi, seguendo l'evoluzione tecnologica, in modo da poter offrire ai propri clienti nuovi servizi a prezzi competitivi. Un'attività che si muove in tal senso, cercando di ottimizzare i sistemi informativi di un'azienda è quella che punta all'integrazione delle applicazioni aziendali, definita **EAI (Enterprise Application Integration)**. Scopo di questa integrazione è permettere a due o piú sistemi di business di condividere processi e dati.
Con l'avvento del Web questa necessità di integrazione è divenuta ancora piú pressante. E' necessario evolversi continuamente e per questo motivo ogni azienda deve periodicamente riconsiderare la propria organizzazione in termini sia di tecnologia utilizzata, eventualmente adottandone di nuova, sia della propria struttura interna, fondendola in alcuni casi con quella di altre aziende.
La "rete" mette a disposizione un ottimo mezzo di comunicazione ai fini dell'integrazione dei vari sistemi e le prospettive di crescita ed evoluzione delle aziende sono molto interessanti. Ogni azienda si trova però di fronte ad una scelta; i propri sistemi ed applicazioni sono ormai superati e non adatti ad essere integrati con altri, per cui deve essere presa in considerazione la transizione a sistemi e piattaforme di nuova concezione, il cui impatto non è indolore sia a livello finanziario, per l'acquisto del software, sia in termini di conoscenze sul sistema utilizzato, che dovranno essere riacquisite dai propri tecnici ed operatori.
E' in questo ambito che entra in gioco EAI (Enterprise Application Integration), filosofia di sviluppo di nuovi sistemi, che mira all'integrazione delle strutture esistenti all'interno delle varie aziende, senza che queste debbano essere sostituite o modificate pesantemente.
Prima della nascita di EAI l'integrazione era una scelta rischiosa per un'azienda. Piú aziende che avessero voluto integrare fra loro le prorpie strutture avrebbero dovuto fare uno studio accurato per valutare tempi, costi e rischi:

piú semplicemente "fattibilità". In alcuni casi i risultati di tale studio potevano portare le compagnie a rinunciare all'integrazione; in altri casi invece, dopo aver deciso di intraprendere tale strada, il processo poteva impiegare anni per arrivare a conclusione o addirittura poteva interrompersi prima della completa realizzazione a causa della crescita imprevista dei costi divenuta insostenibile.
L'approccio di EAI permette di superare queste problematiche, definendo una metodologia per far comunicare fra loro applicazioni diverse senza doverle modificare: EAI Middleware. Il processo di integrazione risulta separato dalle specifiche applicazioni, che vengono messe in comunicazione attraverso lo scambio di messaggi. Il middleware EAI non costituisce solamente un mezzo per il trasporto dei messaggi ma si occupa anche di instradarli, filtrarli e processarli e, facendo uso di opportuni adattatori, nasconde l'eterogeneità delle varie applicazioni e sistemi connessi, permettendo così di accedere ad ognuno di essi utilizzando sempre lo stesso modello di programmazione ed il medesimo formato di scambio di dati. EAI è quindi la soluzione ideale per ambienti eterogenei, nei quali le applicazioni si trovano ad esempio su piattaforme diverse.

## 1.1 EAI ed il Web

Il Web, ambiente eterogeneo per eccellenza, è il campo che mette in risalto l'importanza dell'integrazione di applicazioni. Per le aziende si apre la possibilità di fornire ai propri clienti nuovi servizi, in modo semplice e rapido, via Internet. E' quindi di fondamentale importanza, per quelle compagnie che utilizzano il Web per i loro business, riuscire ad integrare, in modo sempre maggiore ed efficiente, le loro applicazioni ed i sistemi informativi interni con quelli di altre società attraverso la Rete, al fine di incrementare il piú possibile quelle che vengono definite interazioni business-to-business (B2B) e business-to-consumer (B2C).
Fino ad oggi le applicazioni potevano essere suddivise in due categorie: *front-office* e *back-office*. La prima categoria identifica quelle applicazioni che si occupano dell'interfacciamento con l'utente, mentre quelle del secondo tipo sono finalizzate alla gestione dell'infrastruttura del sistema informativo aziendale. L'obiettivo di EAI era l'integrazione di questi due tipi di applicazione. Adesso, dopo l'avvento del Web, ciò che viene richiesto ad EAI è la creazione di nuove applicazioni che, oltre a fondere i due livelli front-office e back-office, possiedano metodi per interagire con altri sistemi attraverso la Rete.
L'obiettivo ultimo di EAI è quindi la creazione di sistemi che permettano di "interfacciare" il sistema informativo aziendale (EIS) con il Web ma, come vedremo in 1.3, l'approccio di EAI presenta forti limitazioni in relazione ad un suo utilizzo su Internet.

Il **Sistema Informativo Aziendale (EIS: Enterprise Information**

**System)** consiste di una o piú applicazioni e costituisce l'infrastruttura informativa interna di un'azienda. Gli utenti di tale sistema hanno a disposizione un insieme di servizi di vario tipo, appartenenti anche a diversi livelli di astrazione, e possono accedere alle applicazioni, residenti su un application server, tramite Application Programming Interfaces (APIs).

Esistono molti modi di intraprendere un'integrazione delle applicazioni di un'azienda sia in termini di architettura del sistema che realizza l'integrazione sia in termini del livello a cui essa è stabilita.
Si può ad esempio adottare, per quanto riguarda l'architettura, un'approccio "two-tier", cioè a due strati, oppure possono essere utilizzati adattatori sincroni o asincroni, mentre l'integrazione può avvenire a livello di interfaccia utente, sostituendo quelle da terminale o grafiche con altre tipicamente compatibili con i browser, oppure a livello di dati (ad esempio database), trasferendoli da un tipo di sorgente dati ad un altro.
La scelta della soluzione da adottare dipende da molti fattori, tra i quali vi sono la grandezza della compagnia e dei suoi capitali e gli obiettivi a cui essa punta.
Per la crescente necessità di sistemi che siano abilitati ad accedere al Web ed al tempo stesso accessibili da esso e per gli scopi di questa tesi l'approccio migliore è quello definito "Server-Based", dove l'integrazione avviene tra applicazioni (A2A: application-to-application), residenti anche su piattaforme diverse, su una rete. In un approccio server-based possono essere coinvolte metodologie come Application Programming Interfaces (APIs), Remote Procedure Calls (RPCs), distributed middleware (come CORBA), Remote Method Invocation (RMI), Message Oriented Middleware (MOM) e Web Services.

## 1.2 Integrazione "Server-Based"

L'integrazione *server-based* prevede la presenza di un "application server" sul quale vengono sviluppate e gestite le applicazioni che hanno accesso alla rete. Gli *application server* sono tipicamente utilizzati per implementare applicazioni con un'architettura "multi-tier" (costituita da piú strati), in particolare "three-tier" o "four-tier".
L'architettura *three-tier* (tre strati) è costituita dai tre seguenti livelli:

- Client: può essere composto da un browser web o da un'applicazione.
- Middle (Intermedio): implementa la logica dell'applicazione ed ha accesso ai dati e ai metodi messi a disposizione dalle applicazioni che fanno parte del sistema informativo aziendale (EIS: Enterprise Information System).
- EIS: contiene il sistema interno dell'azienda, sul quale si trovano applicazioni e database.

Facendo un ulteriore distinzione nel dettaglio del livello intermedio si può vedere l'architettura suddivisa in quattro livelli, *four-tier*: il livello intermedio viene spezzato in livello "Web Component", relativo ai componenti che costituiscono l'interfaccia con il Web, e livello "Business Logic", che contiene i componenti che realizzano la logica del sistema.

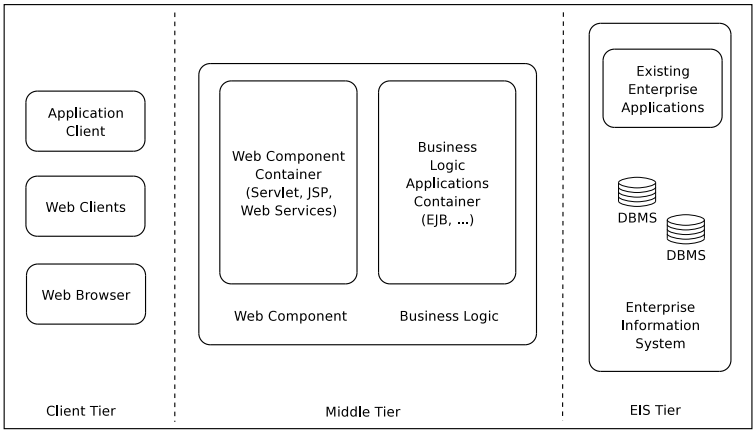


Figura 1.1: Architettura Three-Tier.

Per realizzare il sistema abbiamo bisogno però di una piattaforma software sulla quale costruirlo. La scelta della piattaforma è oggigiorno pressoché limitata alle due principali J2EE e .NET.
La piattaforma J2EE di SUN è basata sulla tecnologia Java; questo la rende indipendente dal sistema operativo e dall'hardware sul quale si trova ma al tempo stesso limita gli sviluppatori al linguaggio di programmazione Java.
La piattaforma .NET di Microsoft è in molti aspetti simile a J2EE: possiede un meccanismo simile alla JVM (Java Virtual Machine) chiamato Common Language Runtime ed un linguaggio concettualmente simile al Java bytecode denominato Intermediate Language (IL). Al contrario di J2EE è però principalmente legata al sistema operativo Windows: esistono alcune iniziative il cui obiettivo è il porting su FreeBSD.

Nel momento in cui viene scelta la piattaforma vi sono dei criteri da tenere in considerazione, come ad esempio:

- la possibilità di integrazione con sistemi ed applicazioni esistenti,
- il supporto a
  - tecnologie B2B,
  - linguaggi di programmazione,
- robustezza, performance e portabilità,

- costi e supporto tecnico.

## 1.3 EAI, SOA e Web Services

Lo scenario nel quale oggi viene perseguita l'integrazione delle applicazioni è, come già detto, il Web.
In passato l'integrazione di applicazioni avveniva tra software che risiedevano sulla stessa piattaforma o middleware (come CORBA[1] o DCOM[2]) e tipicamente all'interno della stessa organizzazione. Volendo invece interconnettere middleware diversi le cose si complicano, dovendo standardizzare le interfacce di applicazioni appartenenti a sistemi differenti per permettere loro di interagire; la difficoltà inoltre aumenta nel caso in cui l'interazione non avvenga fra piattaforme della stessa compagnia ma appartenenti a piú aziende. EAI tentò di risolvere questo problema estendendo il concetto di middleware, il quale punta ad integrare i server che risiedono, nella visione three-tier, nello strato di gestione delle risorse; EAI generalizza quest'idea considerando come blocchi da integrare gli strati relativi alla logica delle applicazioni di middleware differenti.
Con il Web, questi approcci all'integrazione di applicazioni non sono però piú utilizzabili. Questo è dovuto, oltre che all'eterogeneità del campo di sviluppo, anche al fatto che il controllo dell'elaborazione non è piú in mano ad una singola azienda, dato che l'interazione avviene tra applicazioni appartenenti a compagnie differenti. Vi è quindi la necessità di avere un modo "standard" per esporre un software su una rete attraverso un'interfaccia comprensibile da tutte quelle aziende che, volendo far interagire le proprie applicazioni con quelle di altre compagnie, riconoscono tale standard. É così che, per superare le limitazioni di middleware e piattaforme EAI, sono introdotti i **Web Services** e viene adottata l'**Architettura Orientata ai Servizi (SOA)**.

La *Service-Oriented Architecture (SOA)* è un'architettura concettuale che non fa riferimento a nessuna particolare implementazione. Essa pone delle specifiche condizioni che i componenti del sistema devono rispettare e caratteristiche che tale sistema deve necessariamente avere.

I *Web Services* sono invece una nuova tecnologia, che si basa su standard, quali XML, SOAP, WSDL e UDDI (che vedremo in 3.1), la quale fornisce un

[1] **CORBA**, acronimo per Common Object Request Broker Architecture, è una specifica implementazione di un ORB, Object Request Broker, ed è uno standard multipiattaforma e multilinguaggio, progettato da Object Management Group (OMG), per l'interoperabilità dei componenti software. Due programmi basati su CORBA possono interagire anche se appartenenti ad aziende diverse e creati su computer, sistemi operativi, linguaggi di programmazione o reti differenti (in quasi tutti i casi).

[2] **DCOM**, acronimo per Distributed Component Object Model, è un'estensione dell'architettura COM (Component Object Model) di Microsoft per consentire l'elaborazione distribuita; fu realizzato nel 1996 per essere utilizzato nei trasferimenti su reti multiple ed è compatibile sia con applet Java sia con componenti ActiveX.

facile metodo di interfacciamento tra le applicazioni. Questa tecnologia non è legata all'architettura SOA ma, come vedremo nel prossimo capitolo, ha molti punti di contatto con essa e sistemi che utilizzano i Web Services, sfruttando tutte le loro potenzialità, implementano esattamente un'architettura di tale tipo.

# Capitolo 2

# Service-Oriented Architecture e Web Services: concetti base

Scopo di questo capitolo è quello di introdurre i concetti basilari dell'**Architettura Orientata ai Servizi (SOA: Service-Oriented Architecture)**, per poi discutere quali siano i punti di contatto con la tecnologia dei **Web Services** e come queste possano essere utilizzate nella creazione di sistemi residenti su una rete.

## 2.1 SOA: Service-Oriented Architecture

Una Service-Oriented Architecture (SOA, Architettura Orientata ai Servizi) è un modello architetturale per la creazione di sistemi residenti su una rete che focalizza l'attenzione sul concetto di *servizio*. Un sistema costruito seguendo la filosofia SOA è costituito da applicazioni, chiamate *servizi*, ben definite ed indipendenti l'una dall'altra, che risiedono su più computer all'interno di una rete (ad esempio la rete interna di una azienda o una rete di connessione fra più aziende che collaborano: intracompany e intercompany network). Ogni servizio mette a disposizione una certa funzionalità e può utilizzare quelle che gli altri servizi hanno reso disponibili, realizzando, in questo modo, applicazioni di maggiore complessità.

SOA è una forma particolare di *Distributed System*, la cui definizione (da [15]) è la seguente:

> Un *Distributed System (Sistema distribuito)* consiste di vari agenti software distinti che devono lavorare insieme per svolgere alcuni compiti. Inoltre, gli agenti in un sistema distribuito non operano nello stesso ambiente di calcolo, quindi devono comunicare per mezzo di stack di protocolli hardware/software su una

rete. Questo significa che le comunicazioni in un sistema distribuito sono intrinsecamente meno veloci e affidabili rispetto a quelle che utilizzano invocazione diretta del codice e memoria condivisa. Ciò ha importanti implicazioni architetturali perché i sistemi distribuiti richiedono che gli sviluppatori (di infrastruttura e applicazioni) considerino la latenza, fattore imprevedibile dell'accesso remoto, e tengano presente questioni relative alla concorrenza e la possibilità di fallimenti parziali.

### 2.1.1 Caratteristiche di una SOA

L'astrazione delle SOA non è legata ad alcuna specifica tecnologia, ma semplicemente definisce alcune proprietà, orientate al riutilizzo e all'integrazione in un ambiente eterogeneo, che devono essere rispettate dai servizi che compongono il sistema [9, 15]. In particolare un servizio dovrà:

- essere ricercabile e recuperabile dinamicamente.
  Un servizio deve poter essere ricercato in base alla sua interfaccia e richiamato a tempo di esecuzione. La definizione del servizio in base alla sua interfaccia rende quest'ultima (e quindi l'interazione con altri servizi) indipendente dal modo in cui è stato realizzato il componente che lo implementa.

- essere autocontenuto e modulare.
  Ogni servizio deve essere ben definito, completo ed indipendente dal contesto o dallo stato di altri servizi.

- essere definito da un'interfaccia ed indipendente dall'implementazione.
  Deve cioè essere definito in termini di ciò che fa, astraendo dai metodi e dalle tecnologie utilizzate per implementarlo. Questo determina l'indipendenza del servizio non solo dal linguaggio di programmazione utilizzato per realizzare il componente che lo implementa ma anche dalla piattaforma e dal sistema operativo su cui è in esecuzione: non è necessario conoscere come un servizio è realizzato ma solo quali funzionalità rende disponibili.

- essere debolmente accoppiato con altri servizi (loosely coupled).
  Un'architettura è debolmente accoppiata se le dipendenze fra le sue componenti sono in numero limitato. Questo rende il sistema flessibile e facilmente modificabile.

- essere reso disponibile sulla rete attraverso la pubblicazione della sua interfaccia (in un Service Directory o Service Registry) ed accessibile in modo trasparente rispetto alla sua allocazione.
  Essere disponibile sulla rete lo rende accessibile da quei componenti che ne richiedono l'utilizzo e l'accesso deve avvenire in maniera indipendente rispetto all'allocazione del servizio. La pubblicazione dell'interfaccia deve rendere noto anche le modalità di accesso al servizio.

- fornire un'interfaccia possibilmente a "grana grossa" (coarse-grained). Deve mettere a disposizione un basso numero di operazioni, cioè poche funzionalità, in modo tale da non dover avere un programma di controllo complesso. Deve essere invece orientato ad un elevato livello di interazione con gli altri servizi attraverso lo scambio di messaggi. Per questo motivo e per il fatto che i servizi possono trovarsi su sistemi operativi e piattaforme diverse è necessario che i messaggi siano composti utilizzando un formato standard largamente riconosciuto (*Platform Neutral*).
  I dati che vengono trasmessi attraverso i messaggi possono essere costituiti sia dal risultato dell'elaborazione di un certo servizio sia da informazioni che più servizi si scambiano per coordinarsi fra loro.

- essere realizzato in modo tale da permetterne la composizione con altri.
  Nell'architettura SOA le applicazioni sono il risultato della composizione di più servizi. É per questo motivo che ogni servizio deve essere indipendente da qualsiasi altro, in modo tale da ottenere il massimo della riusabilità. La creazione di applicazioni o di servizi più complessi attraverso la composizione dei servizi di base viene definita *Service Orchestration*.

Queste dunque le caratteristiche di un sistema di tipo SOA, di cui adesso passiamo a descrivere il funzionamento.

### 2.1.2 Come funziona una SOA

Gli attori di un sistema SOA sono tre:

- Service Provider
- Service Consumer
- Service Registry.

Il Service Provider è un'entità che mette a disposizione un qualche servizio. Tale servizio, per poter essere trovato da altre entità che vogliono utilizzarlo, deve essere reso visibile sulla rete, in termine tecnico *Pubblicato*. A tal fine il Service Provider comunica al Service Registry le informazioni relative al servizio, perché vengano memorizzate. Il Service Registry possiede quindi le informazioni, come URL e modalità di accesso, di tutti i servizi disponibili. Nel momento in cui un Service Consumer dovrà utilizzare un servizio farà richiesta delle informazioni ad esso relative al Service Registry. Con queste informazioni il Service Consumer potrà comunicare direttamente con il Service Provider ed utilizzare il servizio.
In figura 2.1 sono riportate le interazioni fra le entità appena descritte.

Tutte queste interazioni passano attraverso quella che in figura viene genericamente definita *Rete di Comunicazione*, la quale in un'implementazione

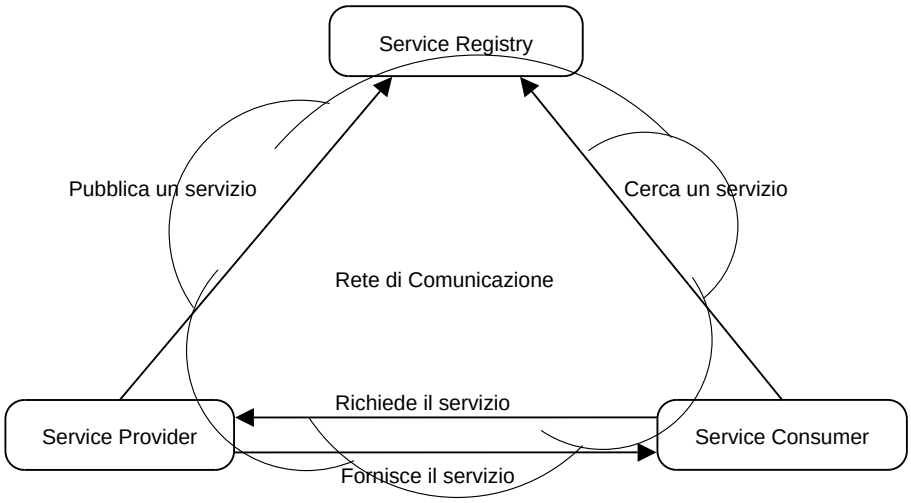


Figura 2.1: Esempio di Architettura SOA.

reale di una SOA può essere costituita sia da Internet sia da una intranet.

SOA definisce, dunque, le caratteristiche che i componenti facenti parte di un sistema devono avere al fine di poter definire quest'ultimo un'architettura orientata ai servizi.

Dopo aver descritto cos'è l'architettura SOA ed il funzionamento di un sistema di questo tipo, vediamo adesso cosa sono i Web Services, quali tecnologie utilizzano ed il loro legame con SOA.

## 2.2 Web Services

I Web Services sono un nuovo tipo di applicazioni web che cooperano fra loro, indipendentemente dalla piattaforma sulla quale si trovano, attraverso lo scambio di messaggi.
Ognuna di queste applicazioni viene chiamato *Web Service (Servizio Web)*, o più semplicemente *servizio*, del quale il **Web Services Architecture Working Group** (del W3C) [15, 16] dà la seguente definizione:

> Un *Web Service* è un'applicazione software identificata da un URI (Uniform Resource Identifier), le cui interfacce pubbliche e collegamenti sono definiti e descritti come documenti XML, in un formato comprensibile alla macchina (specificatamente WSDL, vedi 3.1.2). La sua definizione può essere ricercata da altri agenti software situati su una rete, i quali possono interagire direttamente con il Web Service, con le modalità specificate nella sua definizione, utilizzando messaggi basati su XML (SOAP, ve-

di 3.1.3), scambiati attraverso protocolli Internet (tipicamente HTTP[1]).

Le tecnologie su cui si basano i Web Services, appena citate nella definizione, sono:

- XML, eXtensible Markup Language
- SOAP, Simple Object Access Protocol
- WSDL, Web Services Description Language
- UDDI, Universal Description, Discovery and Integration.

Attraverso l'utilizzo di questi ed altri standard i Web Services rendono possibile la comunicazione e la cooperazione, attraverso il web, di più applicazioni (servizi) che mettono a disposizione alcune funzionalità e, allo stesso tempo, utilizzano quelle rese disponibili da altre. Si può cioè ricercare e invocare servizi che possono essere composti per formare un'applicazione per l'utente finale, per abilitare transazioni di business o per creare nuovi Web Services.[25]

Di queste tecnologie, di cui parleremo nel capitolo 3.1, XML ha dato un contributo molto importante alla nascita dei Web Services. Linguaggio a marcatori (tag) derivato da SGML, Standard Generalization Markup Language, come ad esempio il più conosciuto HTML, HyperText Markup Language, l'XML è utilizzato per la memorizzazione di informazione in maniera strutturata.
XML è un formato indipendente dalle varie piattaforme; ciò è dovuto, oltre che all'essere universalmente riconosciuto come standard, anche al fatto che tale tecnologia si basa sul formato testo e quindi un documento XML può essere letto chiaramente su qualsiasi sistema operativo.
Questa indipendenza lo rende la soluzione ideale per lo scambio di informazioni attraverso il Web.

I vantaggi offerti dai Web Services sono:

- Indipendenza dalla piattaforma: i Web Services possono, infatti, comunicare fra loro anche se si trovano su piattaforme differenti.
- Indipendenza dall'implementazione del servizio: l'interfaccia che un Web Service presenta sulla rete è indipendente dal software che implementa tale servizio. In futuro tale implementazione potrà essere sostituita o migliorata senza che l'interfaccia subisca modifiche e quindi senza che dall'esterno (da parte di altri utenti o servizi sulla rete) si noti il cambiamento.

[1] Hyper Text Transfer Protocol

- Riuso dell'infrastruttura: per lo scambio di messaggi si utilizza SOAP che fa uso di HTTP, grazie al quale si ottiene anche il vantaggio di permettere ai messaggi SOAP di passare attraverso sistemi di filtraggio del traffico sulla rete, quali "Firewall".

- Riuso del software: è possibile riutilizzare software implementato precedentemente e renderlo disponibile attraverso la rete.

Il concetto di Web Services implica quindi un modello di architettura ad oggetti distribuiti (oggetti intesi come applicazioni), che si trovano localizzati in punti diversi della rete e su piattaforme di tipo differente.
Il legame con l'architettura SOA sta nel fatto che, sfruttando al meglio tutte le caratteristiche della tecnologia dei Web Services, il sistema che si ottiene implementa proprio un'architettura orientata ai servizi. Ad oggi i Web Services rappresentano la soluzione migliore per la realizzazione di una SOA su larga scala, ovvero su Internet.

### 2.2.1 Comunicazione fra due Web Service

La figura 2.2 illustra un semplice esempio di applicazione della tecnologia dei Web Services che utilizzeremo per descrivere l'interazione fra due generici servizi, situati in punti qualsiasi del Web.

I due rettangoli in figura 2.2 rappresentano due processi, in ambito di Web Services, definiti *Service Consumer (o Service Client)* e *Service Provider*. Un processo viene definito *Comsumer* o *Provider* a seconda del ruolo che ha nella comunicazione. Il Service Consumer è colui che fa richiesta di un certo servizio di cui ha bisogno, mentre con il termine Service Provider si identifica chi fornisce tale servizio. La comunicazione avviene per mezzo di un messaggio di richiesta (Service Request), dal Consumer al Provider, ed uno di risposta (Service Response), dal Provider al Consumer.
Questi due messaggi (di richiesta e di risposta) devono essere definiti in modo tale da essere comprensibili sia al consumer che al provider (vedi 3.1.3).

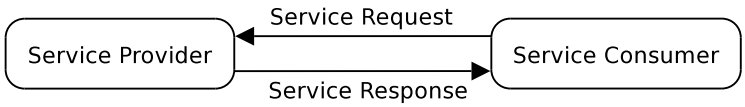


Figura 2.2: Esempio di connessione Consumer-Provider.

I ruoli di Consumer e Provider non sono fissati al momento della generazione dei processi. Un processo può essere sia Consumer, nel momento in cui necessita di un servizio messo a disposizione da altri, sia Provider, quando riceve da un secondo processo la richiesta di un certo servizio di cui è fornitore. Il ruolo (Consumer o Provider) è quindi stabilito dalle connessioni.

### 2.2.2 Web Services e Architettura non SOA

É importante specificare che parlare di Web Services non implica necessariamente parlare di SOA. Si può ad esempio utilizzare i Web Services per aggiungere ad un'architettura esistente una funzionalità basata sui servizi al fine di ottenere un certo requisito richiesto dal progetto che si sta realizzando. Oppure si può realizzare un sistema privato, interno ad una azienda, basato sulle tecnologie dei Web Services (fig. 2.3).

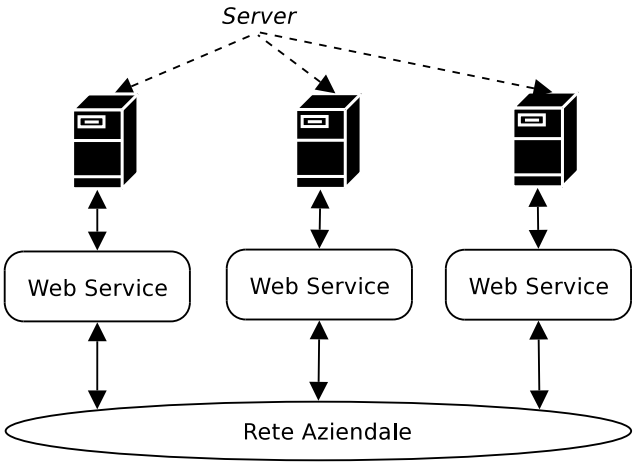


Figura 2.3: Utilizzo di Web Services in un'architettura di tipo non SOA.

In questi casi, come in altri, non è detto che si possa definire il sistema realizzato un'architettura orientata ai servizi; questo perchè, per il particolare utilizzo che si fa dei Web Services all'interno di un certo sistema software, può non essere necessario soddisfare alcuni requisiti richiesti dal modello di architettura SOA.
Poniano, ad esempio, il caso di una rete interna ad una azienda in cui sono state realizzate applicazioni che fanno uso della tecnologia dei Web Services; se si prevede che le modifiche che verranno apportate nel tempo al sistema non siano in misura elevata, si può ad esempio pensare di memorizzare le informazioni riguardanti tutti i servizi nella configurazione di ognuno di essi, escludendo perciò dal progetto la presenza di un Service Registry, entità prevista invece dal modello di architettura SOA.

Un altro possibile utilizzo, molto interessante, dei Web Services è il loro impiego nel processo di integrazione di applicazioni fra più aziende o fra più sedi della stessa azienda, attraverso il Web.

Ad esempio si possono avere due aziende partner interessate a mettere a disposizione l'una dell'altra specifici servizi che si trovano sui propri server, mantenendone però la gestione dell'implementazione. I Web Services rendono questo possibile senza doversi preoccupare della compatibilità fra

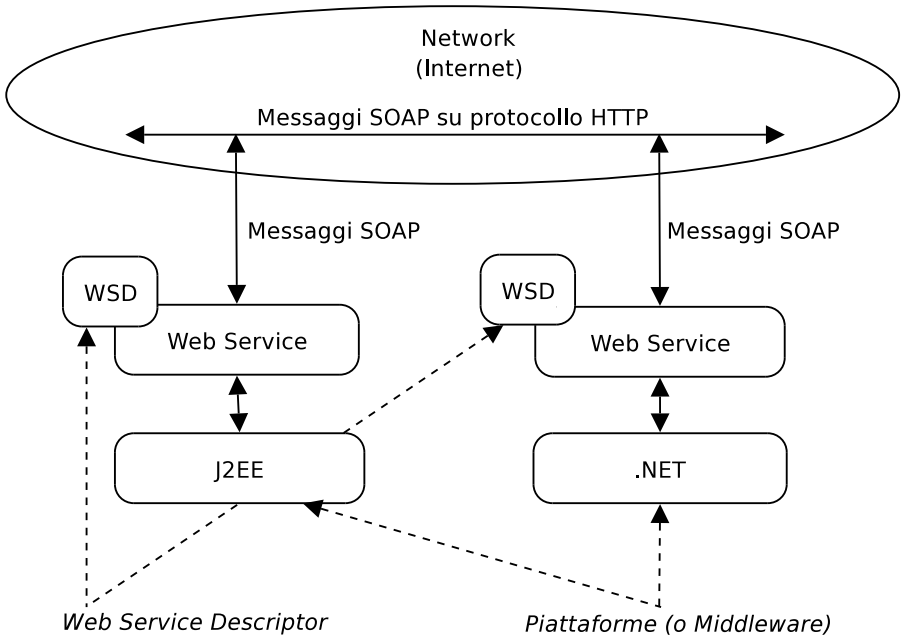


Figura 2.4: Esempio di comunicazione fra Web Service implementati su piattaforme diverse.

piattaforme, sistemi operativi o linguaggi di programmazione, grazie alla totale indipendenza che questa tecnologia ha rispetto all'implementazione del servizio. Un esempio è schematizzato in figura 2.4 dove si ha una cooperazione fra Web Service realizzati su due piattaforme diverse, le due oggi più importanti J2EE (Java 2 Platform, Enterprise Edition) e .NET (Microsoft .NET).

Il vantaggio di una collaborazione di questo tipo fra più aziende non è solo quello di poter avere accesso diretto attraverso la rete a servizi implementati e gestiti da altri ma anche di poterli comporre fra loro, siano essi in locale o in remoto, in modo tale da ottenerne di nuovi che mettano a disposizione nuove funzionalità o che riescano a risolvere problemi di complessità maggiore.

Abbiamo visto come possono essere utilizzati i Web Services per interconnettere servizi situati in punti diversi della rete e con caratteristiche differenti. L'esempio che abbiamo descritto riferendoci alla figura 2.3 fa uso di riferimenti agli altri servizi di tipo **statico**; il Client (o Consumer) conosce già l'indirizzo (URL) al quale poter trovare un certo servizio (Provider). Si può invece desiderare di non dover conoscere, al momento dell'implementazione di un servizio, gli URL di altri servizi di cui farà uso, lasciando ad un'ulteriore agente il compito di fornire, a tempo di esecuzione, questa ed

altre informazioni.
Stiamo parlando del Service Registry ed introduciamo così l'utilizzo dei Web Services nella realizzazione di un'architettura Service-Oriented.

### 2.2.3 Web Services con Architettura SOA

La presenza del Service Registry (o anche Service Directory o Service Broker), di cui abbiamo già parlato nel capitolo 2.1.2, è ciò che rende il sistema, nell'esempio di utilizzo dei Web Services visto precedentemente (in figura 2.3), un'architettura Service-Oriented (SOA).

Per implementare il Service Registry i Web Services fanno uso di UDDI, Universal Description, Discovery and Integration (vedi 3.1.4).
UDDI è un servizio di registro pubblico in cui le aziende possono registrare (pubblicare) e ricercare Web Services. Esso mantiene informazioni relative ai servizi come l'URL e le modalità di accesso.
Anche UDDI è un Web Service, il quale mette a disposizione due operazioni:

- Publish, per la registrazione
- Inquiry, per la ricerca.

Si ottiene così quella che oggigiorno è da molti considerata la migliore soluzione per l'implementazione di un sistema con architettura Service-Oriented.
In figura 2.5 è riportata la schematizzazione del funzionamento di un sistema con architettura SOA, realizzato attraverso l'uso dei Web Services.

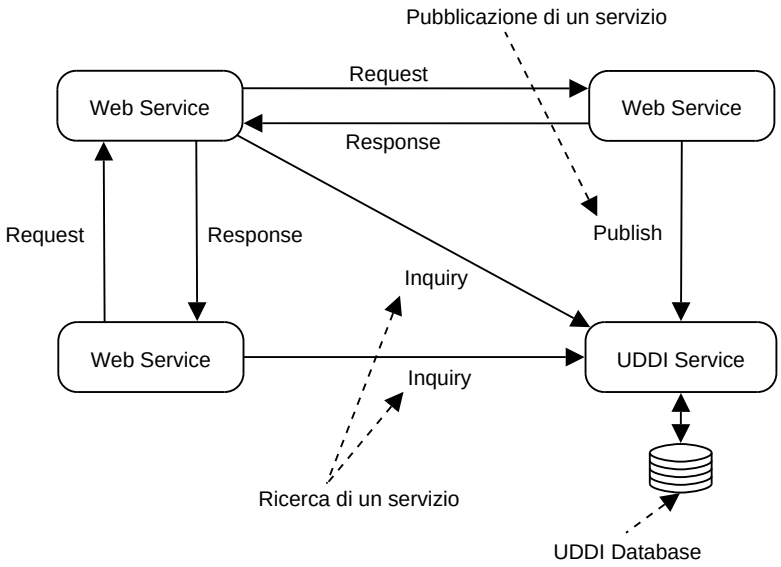


Figura 2.5: Sistema con architettura Service-Oriented realizzato con la tecnologia dei Web Services.

# Capitolo 3

# Web Services: concetti tecnici

Nel capitolo precedente abbiamo parlato in modo generale dei Web Services. Di essi abbiamo dato una definizione, analizzato i vantaggi e descritto i ruoli esistenti in una loro comunicazione. Inoltre abbiamo visto i punti di contatto di questa tecnologia con l'architettura SOA.
Vediamo adesso i Web Services più da vicino, descrivendo in dettaglio le tecnologie di cui sono costituiti.

## 3.1 Standard e Tecnologie alla base dei Web Services

I Web Services si basano su tecnologie oggi riconosciute universalmente come standard: XML, WSDL, SOAP e UDDI. Vediamole quindi una per una partendo da quella che è un po' il comune denominatore di tutte le altre: XML.

### 3.1.1 XML

XML, eXtensible Markup Language, è un *metalinguaggio* nato nel 1998 e derivato da SGML (Standard Generalization Markup Language). Il termine "metalinguaggio" significa che esso è un linguaggio per mezzo del quale se ne possono creare altri.
"Esteticamente" simile ad HTML, poiché anch'esso è basato su marcatori o "tag" (questo è un tag: <tag_name>), ne differisce profondamente. Essi infatti non sono semplicemente due linguaggi diversi ma appartengono a categorie diverse; HTML è un linguaggio, in cui il comportamento di ogni marcatore è già stabilito, mentre XML è, come già anticipato, un metalinguaggio, che permette allo sviluppatore di definire marcatori personalizzati e di specificarne il ruolo.
Il motivo che ha portato alla creazione di XML è stata la necessità di avere

documenti strutturati e flessibili che potessero essere utilizzati sulla rete per lo scambio di dati. I due linguaggi prima citati, HTML e SGML, non erano adatti allo scopo, per due motivi opposti. HTML ha una struttura rigida dove, come già spiegato precedentemente, i tag disponibili sono predefiniti e dal comportamento già stabilito. Al contrario SGML fornisce una struttura personalizzabile ma troppo ampia e complessa per giustificarne un utilizzo via web.
Inoltre XML ha un formato indipendente dalle varie piattaforme; ciò è dovuto, oltre che all'essere universalmente riconosciuto come standard, al fatto che tale tecnologia si basa sul formato testo e quindi un documento XML può essere letto chiaramente su qualsiasi sistema operativo.

Il contenuto di un documento XML è costituito da marcatori e dati strutturati secondo un ordine logico determinato da una struttura ad albero. Il formato del documento è testuale e questo permette all'utente di accedervi direttamente in lettura.

**Esempio di documento XML**

```
<?xml version="1.0"?>
<anagrafica>
  <persone>
    <persona>
      <nome>...</nome>
      <cognome>...</cognome>
      <datadinascita>...</datadinascita>
    </persona>
    <persona>
      <nome>...</nome>
      <cognome>...</cognome>
      <datadinascita/>
    </persona>

    ...

  </persone>
</anagrafica>
```

Il primo *tag* specifica che si tratta di un documento XML che fa riferimento alla versione 1.0. Questo tag, non obbligatorio, è l'unico ad essere "singolo"; tutti gli altri tag devono essere chiusi. Come si può notare nell'esempio, al tag di apertura ne corrisponde un'altro di chiusura costituito dallo stesso nome preceduto dalla barra "/"; in alcuni casi può non essere presente il tag di chiusura ma allora il simbolo "/" è presente in fondo al tag di apertura, prima della parentesi angolata ">", ed esso viene definito un tag "vuoto". Il tag vuoto, come <datadinascita/> nell'esempio, può comunque essere espresso utilizzando i due tag di apertura e di chiusura tra i quali, ovviamente, non è presente alcun dato (<datadinascita></datadinascita>).

Il compito di un documento XML è memorizzare i dati all'interno di una struttura gerarchica che rappresenti le relazioni esistenti fra di essi, senza

curarsi minimamente della loro rappresentazione visuale. Tali dati possono poi essere visualizzati in molti modi differenti, a seconda del caso, come ad esempio una semplice pagina HTML.

Abbiamo detto che ogni sviluppatore può creare i propri marcatori ma, affinché questi siano interpretabili correttamente da chiunque voglia accedere ai dati contenuti nel documento, c'è bisogno di un meccanismo che definisca quali elementi sono presenti, la loro struttura e le relazioni fra di essi. Tecnologie create per assolvere questo compito sono Document Type Definition, XML-Schema e XML Namespace.

### DTD: Document Type Definition

Una Document Type Definition è un file in cui è riportata la definizione di tutti gli elementi, e dei loro attributi, usati nel documento XML, specificando inoltre la correlazione tra di essi. Tale file permette ad un'applicazione di sapere se il documento XML che sta analizzando è corretto o no, dato che gli elementi, essendo stati creati dallo sviluppatore, risulterebbero privi di significato senza una loro definizione.
Una DTD definisce quindi la struttura di un particolare tipo di documento XML, rispetto al quale si può valutare la conformità di una data istanza XML.
Le DTD, primo strumento di questo genere, presentano però delle limitazioni: possiedono una sintassi complessa (non sono scritte in XML), non permettono di specificare tipi di dato e sono difficilmente estendibili.

### XML-Schema

Uno strumento, creato allo stesso scopo delle DTD, che superà le limitazioni di queste ultime è XML-Schema.
Un documento XML-schema definisce[19]:

- gli elementi e gli attributi che possono comparire in un documento,
- quali elementi sono elementi figlio,
- l'ordine ed il numero degli elementi figlio,
- se un elemento è vuoto o può includere testo,
- i tipi di dato per gli elementi e gli attributi,
- i valori di default ed i valori costanti per gli elementi e gli attributi.

Rispetto alle DTD, gli XML-Schema sono estensibili e scritti in XML, rendono possibile la definizione di tipi di dato e di vincoli, ammettono l'ereditarietà e supportano i namespace.

### XML Namespace

XML Namespace è utilizzato per risolvere la possibile ambiguità fra elementi di documenti diversi.
Gli elementi di un documento XML sono identificati da un nome che è unico all'interno del documento stesso, ma può accadere che un elemento appartenente ad un altro file abbia lo stesso nome. Questo fatto crea un problema di ambiguità quando ci si riferisce a questi elementi; tale ambiguità viene risolta con l'introduzione dei *namespace*. Un namespace identifica l'insieme di tutti gli elementi di un documento, semplicemente associando un prefisso ai loro nomi. In questo modo due elementi appartenenti a file diversi ed aventi lo stesso nome possono essere identificati univocamente grazie al fatto che essi appartengono a namespace differenti.

### Validità di un documento XML

I documenti XML vengono analizzati da applicazioni chiamate *parser* che ne verificano la correttezza e la conformità a certe specifiche. Un documento XML può essere di due tipi: "Well-Formed" e "Valid".
Un documento è semplicemente **Well-Formed**, cioè ben formato, se rispetta la sintassi XML, definita dalle seguenti regole:

- Il documento deve utilizzare una DTD o iniziare con una dichiarazione XML.
- I valori degli attributi degli elementi devono essere inclusi tra virgolette.
- Tutti gli elementi non vuoti devono avere un tag di apertura ed uno di chiusura.
- Un elemento vuoto può essere rappresentato come tutti gli altri da due tag o più semplicemente da un tag che contiene la barra "/" prima della fine.
- I tag di apertura e chiusura devono essere annidati correttamente.
- I caratteri < e >, utilizzati per la marcatura, ed altri caratteri speciali non possono essere inseriti direttamente nel testo ma, affinché siano considerati come parte del contenuto, è necessario fare ricorso alle entità, cioè codici particolari relativi ad ognuno di tali caratteri (es: &lt per <).

Inoltre un documento "Well-Formed" è anche considerato **Valid**, cioè valido, se è conforme ad una DTD in esso specificata o ad un dato XML-Schema.

### 3.1.2 WSDL

WSDL, ovvero *Web Services Description Language*, è un linguaggio, basato su XML, usato per descrivere, in modo completo, un Web Service. Più precisamente un documento WSDL fornisce informazioni riguardanti l'interfaccia del Web Service in termini di:

- servizi offerti dal Web Service,
- URL ad essi associato,
- modi per l'invocazione,
- argomenti accettati in ingresso e modalità con cui debbono essere passati,
- formato dei risultati restituiti,
- formato dei messaggi.

In altri parole si può dire che un file WSDL fornisce la descrizione relativa ad un Web Service in termini di:

- cosa fa,
- come comunica,
- dove si trova.

Attraverso tale file si può quindi conoscere tutti i dettagli per poter invocare correttamente un servizio.

#### Struttura di un documento WSDL

Un documento WSDL è un file XML costituito da un insieme di definizioni. Il documento inizia sempre con un elemento radice chiamato **definitions** ed al suo interno utilizza i seguenti elementi principali nella definizione dei servizi:

- Types - definizione dei tipi dei dati utilizzati.
- Message - definizione dei messaggi che possono essere inviati e ricevuti.
- Port Type - insieme di servizi, Operation, offerti da un Web Service.
- Binding - informazioni sul protocollo ed il formato dei dati relativo ad un particolare Port Type.
- Service - insieme di endpoint relativi al servizio.

Ciascuno di questi elementi identifica una distinta sezione del documento contenente informazioni relative ad uno specifico aspetto. Vediamo meglio come è strutturato un file WSDL analizzando un esempio e descrivendo i costrutti di cui è costituito.

**Documento WSDL**

```
<wsdl:definitions name="nmtoken" targetNamespace="uri">

    <import namespace="uri" location="uri"/>

    <wsdl:types>
        <xsd:schema .... />
    </wsdl:types>

    <wsdl:message name="nmtoken">
        <part name="nmtoken" element="qname" type="qname"/>
    </wsdl:message>

    <wsdl:portType name="nmtoken">
        <wsdl:operation name="nmtoken">
           <wsdl:input name="nmtoken" message="qname">
           </wsdl:input>
           <wsdl:output name="nmtoken" message="qname">
           </wsdl:output>
           <wsdl:fault name="nmtoken" message="qname">
           </wsdl:fault>
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="nmtoken" type="qname">
        <wsdl:operation name="nmtoken">
           <wsdl:input>
           </wsdl:input>
           <wsdl:output>
           </wsdl:output>
           <wsdl:fault name="nmtoken">
           </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>

    <wsdl:service name="nmtoken">
        <wsdl:port name="nmtoken" binding="qname">
        </wsdl:port>
    </wsdl:service>

</wsdl:definitions>
```

Riprendendo la sintetica schematizzazione di ciò che un documento WSDL descrive, vediamone la corrispondenza con l'esempio sopra riportato.

Cosa fa un Web Service è descritto nelle sezioni delimitate dai tag:

- *types*
- *message*
- *portType*

Le caratteristiche di comunicazione sono invece descritte in

- *binding*

Infine il punto di accesso ad un servizio è definito da

- *service*

L'elemento **types** racchiude le definizioni dei tipi dei dati che sono coinvolti nello scambio dei messaggi. Come sistema standard per la tipizzazione, WSDL si basa su quello definito per gli schemi XML (*XSD, XML Schema Definition*) ed allo stesso tempo è però possibile aggiungere anche altri tipi.

**types**

```
<wsdl:types>
  <xsd:schema targetNamespace="..."
    xmlns="http://www.w3.org/2001/XMLSchema"/>
</wsdl:types>
```

La sezione relativa ai messaggi definisce invece l'input e l'output dei servizi. Ogni elemento **message** racchiude le informazioni, come parametri e loro tipi, relative ad uno specifico messaggio. Si possono avere ad esempio due message: uno relativo al messaggio di input ed uno relativo al messaggio di output. Ogni elemento **part** identifica un parametro.

**message**

```
<wsdl:message name="nomeMessaggio">
  <part name="nomeParametro" element="qname" type="xsd:string"/>
  <part name="nomeParametro" element="qname" type="xsd:int"/>
  ...
</wsdl:message>
```

La sezione **portType** riporta dettagli relativi alle operazioni che un Web Service rende utilizzabili. Ogni operazione viene descritta facendo uso di un'ulteriore elemento chiamato **operation**. Al suo interno vi sono i messaggi di input e/o di output accettati dal servizio e l'ordine che è necessario seguire per il passaggio dei parametri. Messaggi di input ed output vengono identificati rispettivamente dagli elementi **input** e **output** e la presenza di solo uno dei due o di entrambi e, nel secondo caso, l'ordine in cui vengono riportati determinano la tipologia del servizio che, come vedremo fra poco, può avere diversa natura. L'elemento (opzionale) **fault** specifica il formato del messaggio per ogni errore che può essere restituito in output dall'operazione.

**portType**

```
<wsdl:portType name="nomeWebService">
  <wsdl:operation name="nomeOperazione"
      parameterOrder="par1 par2 ...">
    <wsdl:input name="nomeMessaggio" message="qname">
    </wsdl:input>
```

```
    <wsdl:output name="nomeMessaggio" message="qname">
    </wsdl:output>
    <wsdl:fault name="nomeMessaggio" message="qname">
    </wsdl:fault>
  </wsdl:operation>
</wsdl:portType>
```

La sezione **binding** ci porta alla seconda parte di un documento WSDL, passando da cosa fa un Web Service a come comunica. Qui viene stabilito il legame di ogni servizio del Web Service al protocollo per lo scambio di messaggi, il quale può essere HTTP GET/POST, MIME o SOAP; quest'ultimo è il protocollo che viene utilizzato più frequentemente. In particolare si specifica il protocollo per ognuno dei messaggi di un servizio offerto dal Web Service.
Inoltre di solito viene scelto HTTP(S) come protocollo di trasporto per i messaggi SOAP. Questa combinazione viene appunto definita "SOAP over HTTP(S)".
Assumendo "SOAP over HTTP(S)" come protocollo di scambio messaggi e trasmissione, all'interno di binding avremmo prima di tutto la definizione dello stile del collegamento, che può essere *rpc* o *document*, e la specificazione di HTTP come protocollo di trasporto, facendo uso di un elemento relativo a SOAP chiamato **wsdlsoap**.
Per quanto riguarda lo stile del collegamento, la differenza fra "rpc" e "document", che influisce sulla struttura del corpo del messaggio SOAP (elemento *<Body>*), è la seguente:

- Document: il contenuto dell'elemento <Body> del messaggio SOAP è costituito da un documento.
- RPC: l'elemento <Body> contiene l'invocazione ad una procedura remota, una *Remote Procedure Call (RPC)* appunto. La struttura del corpo del messaggio SOAP (elemento <Body>) deve rispettare alcune regole riportate nelle specifiche di SOAP. Secondo queste regole l'elemento <Body> può ad esempio contenere solamente un elemento il cui nome è lo stesso dell'operazione, cioè la procedura remota, che viene invocata e tutti i parametri di tale operazione devono essere riportati come sottoelementi di questo elemento.

Fatte queste assunzioni ed utilizzando *rpc* come stile del collegamento, la parte relativa a binding risulterebbe la seguente:

**binding**

```
<wsdl:binding name="nomeWebServiceSoapBinding" type="qname">
    <wsdlsoap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="nomeOperazione">
       ...
    </wsdl:operation>
    <wsdl:operation name="nomeOperazione">
```

```
      ...
    </wsdl:operation>

    ...

</wsdl:binding>
```

Vi sono poi tanti elementi **operation** quanti sono i servizi (operazioni) messi a disposizione. All'interno di ogni elemento *operation* si possono trovare gli elementi **input**, **output** e **fault**; input ed output riportano, facendo uso dell'elemento **wsdlsoap:body**, il tipo di encoding che può essere *encoded* o *literal*: nel caso encoded deve essere specificato anche l'attributo *encodingStyle*, che nel nostro esempio è SOAP.
Di seguito vediamo un operazione che accetta un messaggio di input di tipo *encoded*, con SOAP come *encodingStyle*, e restituisce un messaggio di output di tipo literal.

**operation**

```
<wsdl:operation name="nomeOperazione">
  <wsdl:input name="nomeMessaggio">
    <wsdlsoap:body use="encoded"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
  </wsdl:input>
  <wsdl:output name="nomeMessaggio">
    <wsdlsoap:body use="literal"/>
  </wsdl:output>
</wsdl:operation>
```

Infine possiamo avere uno o più fault ognuno dei quali è relativo alla notifica di un errore. Nel caso di fault il messaggio è costituito da un'unica parte.

**operation**

```
<wsdl:operation name="nomeOperazione">
  <wsdl:fault name="nomeMessaggio">
  </wsdl:fault>
</wsdl:operation>
```

L'ultima sezione, identificata dall'elemento **service**, è relativa alla localizzazione del Web Service. Al suo interno si trova l'elenco di tutti i servizi messi a disposizione; ognuno di essi è definito da un'elemento **port** che riporta il collegamento utilizzato ed al suo interno ha un'ulteriore elemento **wsdlsoap:address** che indica l'indirizzo URL, chiamato anche *endpoint*, al quale può essere trovato il Web Service.

**service**

```
<wsdl:service name="nomeService">
  <wsdl:port name="nomeWebService" binding="qname">
    <wsdlsoap:address location="http://.../.../"/>
  </wsdl:port>
</wsdl:service>
```

### Tipologia di un servizio

Abbiamo accennato poco sopra, nella sezione relativa a portType ed ai messaggi di ingresso ed uscita delle operazioni, al fatto che un servizio può essere di natura diversa, cioè appartenere ad una tipologia piuttosto che ad un altra, in relazione alla presenza e all'ordine degli elementi *input* ed *output*. Le tipologie a cui un servizio può appartenere sono quattro:

- One-way
- Request-response
- Solicit-response
- Notification

**One-way.** Nel caso One-way è presente il solo elemento *input*. Come intuibile dal nome, la comunicazione avviene in una sola direzione: viene inviato un messaggio da un client al servizio, dopodiché il primo continua la sua computazione senza attendere una risposta dal secondo.

**Request-response.** In questo caso sono presenti entrambi gli elementi *input* ed *output*, in questo ordine. Il servizio riceve il messaggio Request dal client e, dopo aver eseguito l'elaborazione relativa alla richiesta, manda indietro un messaggio Response.

**Solicit-response.** Come nel caso precedente vi sono entrambi gli elementi ma in ordine inverso. E' il servizio ad iniziare la comunicazione inviando un messaggio al client ed attendendo poi una sua risposta.

**Notification.** Questo caso è l'opposto del One-way. Il servizio spedisce un messaggio al client senza attendere una sua risposta. É quindi presente solo l'elemento *output*.

Ognuna di queste tipologie individua la natura del servizio che stiamo descrivendo. Ad esempio la tipologia *Request-response* è quella utilizzata nel modello di comunicazione RPC (Remote Procedure Call), mentre si può avere il caso *One-way* quando sia presente un servizio che memorizza dati ricevuti da più client, senza restituire un messaggio di risposta.

### Utilizzo di un documento WSDL

Esistono alcuni strumenti, come il tool *WSDL2Java* (facente parte del framework **Axis** di cui parleremo più avanti), che prendono in input un documento di questo tipo per creare a runtime l'implementazione del client per accedere al servizio. Ma più semplicemente si può vedere il vantaggio apportato dall'uso dei documenti WSDL nel disaccoppiamento del servizio Web dal protocollo di trasporto e dai percorsi fisici, come gli endpoint. Si ottiene così un livello di indirezione, simile a quello che si ha fra i DNS e gli indirizzi internet, grazie al quale non è necessario configurare direttamente l'URL del servizio. In questo modo se vi saranno molti client che utilizzano

il servizio, nel momento in cui esso cambia indirizzo, non si dovrà riconfigurarli tutti, ma semplicemente aggiornare tale informazione nel documento WSDL, poiché è da questo che i client otterranno l'endpoint.[8]

### 3.1.3 SOAP

SOAP, Simple Object Access Protocol, è un protocollo di trasmissione di messaggi in formato XML. SOAP mette a disposizione un meccanismo semplice, ma allo stesso tempo solido, che permette ad una applicazione di mandare un messaggio XML ad un'altra applicazione. Un messaggio è costituito da una trasmissione in un senso, dal mittente al ricevente, ma, come abbiamo descritto precedentemente, si possono avere uno o più messaggi, che definiscono così il tipo di comunicazione che stiamo stabilendo: One-way, Request-response, Solicit-response, Notification.
SOAP è un protocollo di alto livello ed è completamente indipendente dal protocollo di trasmissione sottostante, che può essere indifferentemente HTTP (Hypertext Transfer Protocol), JMS (Java Message Service), SMTP (Simple Mail Transfer Protocol), MIME (Multipurpose Internet Message Encapsulation) o altri. Tra questi il più usato è HTTP.

#### Struttura di un messaggio SOAP

Un messaggio SOAP, di cui si può vedere una rappresentazione grafica in figura 3.1, è costituito da un contenitore, chiamato **Envelope**, all'interno del quale vengono distinte due sezioni principali denominate **Header** e **Body**, che contengono rispettivamente l'intestazione ed il corpo del messaggio.
La rappresentazione grafica di figura 3.1 aiuta a comprendere meglio la struttura di un messaggio SOAP, di cui riportiamo i costrutti principali:

**Envelope**

```
<env:Envelope
    xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
    env:encodingStyle="http://schemas.xmlsoap.org/schemas/encoding/">
  <env:Header>
    <!-- Intestazione -->
  </env:Header>
  <env:Body>
    <!-- Corpo del messaggio -->
  </env:Body>
</env:Envelope>
```

Notiamo la presenza di due attributi all'interno dell'elemento Envelope. Il primo, identificato da *xmlns:env*, è un attributo obbligatorio che specifica il **namespace** relativo al SOAP Envelope. Tutti i nomi in un documento sono unici in modo da non avere ambiguità. Ma nel momento in cui si fa riferimento a documenti esterni tale problema si ripresenta. Il namespace serve ad identificare tutti gli elementi di un certo documento come appartenenti ad esso; in questo modo, se un certo nome è presente all'interno di

due documenti distinti, il riferimento corretto ad essi è garantito dalla loro appartenenza a namespace differenti. Il secondo attributo, identificato da *env:encodingStyle*, specifica regole per la serializzazione dei dati e può essere applicato a qualsiasi elemento; inoltre tutti i figli dell'elemento che specifica tale attributo lo ereditano.

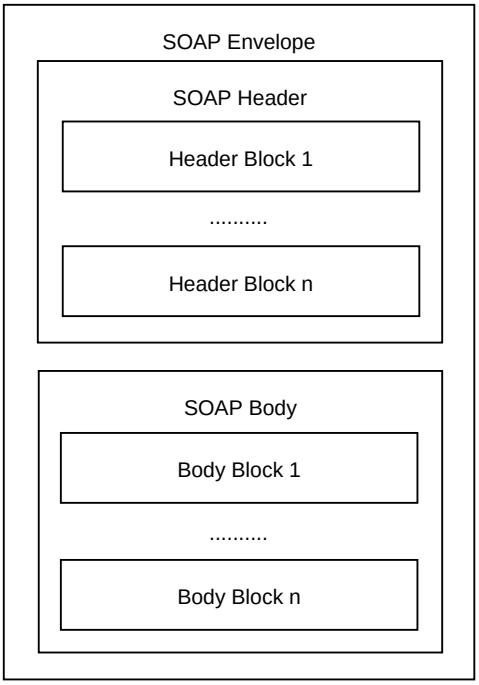


Figura 3.1: Struttura di un messaggio SOAP.

Un messaggio SOAP viene mandato da un mittente ad un destinatario, i quali sono gli attori della comunicazione. Ad essi ci si riferisce facendo uso del termine *Nodo*. Un nodo, oltre che mittente e destinatario, può essere anche intermediario. Un messaggio SOAP può attraversare anche molti intermediari prima di raggiungere il destinatario ed ogni intermediario può elaborare le intestazioni del messaggio. Affinché tali intestazioni vengano elaborate correttamente esistono degli attributi specifici che determinano ruolo e comportamento degli intemediari.

**Header.** La parte Header può non essere presente ma quando ne viene fatto uso in essa troviamo specificate informazioni riguardanti l'instradamento del messaggio sulla rete, che sono dirette ai nodi intermediari. L'informazione, strutturata in blocchi, può essere relativa al percorso che deve seguire il messaggio, attraverso certi nodi piuttosto che altri, oppure ad azioni che il nodo deve compiere.
La sezione Header è suddivisa in blocchi, che possono essere aggiunti e tolti dai nodi intermediari; ogni blocco è indirizzato ad uno o più nodi.

Il comportamento che i nodi devono seguire quando ricevono un messaggio, che discuteremo fra poco facendo uso della tabella 3.1, è stabilito dal valore dei tre attributi seguenti[1], i quali sono specifici per ogni blocco:

- Role
- MustUnderstand
- Relay

L'attributo **Role** determina a quali nodi è indirizzato un certo blocco. I valori che esso può assumere sono riportati di seguito:

- next
  Abbreviazione di "http://www.w3.org/2003/05/soap-envelope/role/next"
  Ogni nodo, intermediario o destinatario, che riceve il messaggio può processare il blocco.
- none
  Abbreviazione di "http://www.w3.org/2003/05/soap-envelope/role/none"
  Nessun nodo può processare il blocco.
- ultimateReceiver
  Abbreviazione di
  "http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver"
  Il nodo destinatario può processare il blocco.

L'attributo **MustUnderstand** può avere valore true o false (nel caso in cui l'attributo non sia presente si assume il valore false). Quando il valore è true il nodo è obbligato ad analizzare il blocco. Nel caso in cui il nodo non riesca a processare correttamente il blocco deve generare un fault.
Infine abbiamo l'attributo **Relay**; anch'esso assume valori booleani. Se ha valore true, il nodo (destinatario di tale blocco) deve ritrasmettere (passare avanti) il blocco nel caso in cui non sia stato processato. L'assenza dell'attributo Relay equivale alla presenza con valore false.

**Header**

```
<env:Header xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <b1:Block1
    xmlns:b1="http://example.org/2001/06/tx"
    env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
    env:mustUnderstand="false"
    env:relay="true">
   ...
 </b1:Block1>
 <b2:Block2
    xmlns:b2="http://example.org/2001/06/tx"
    env:role="http://www.w3.org/2003/05/
              soap-envelope/role/ultimateReceiver"
```

[1] La versione di SOAP alla quale ci stiamo riferendo è la 1.2

```
    env:mustUnderstand="true">
   ...
 </b2:Block2>

 ...

</env:Header>
```

Per chiarire i concetti introdotti illustriamo tutti i casi possibili, che sono stati riportati nella tabella 3.1 assieme al comportamento che deve avere il nodo. Per semplicità assumiamo che nel caso in cui MustUnderstand sia true non vi sia possibilità di fault, cioè siamo sicuri che il nodo analizzi e "capisca" il blocco ("Understood & Processed").

| **Attributi del blocco** | | | **Blocco** | Casi |
|---|---|---|---|---|
| **Role** | **MustUnderstand** | **Relay** | **Inoltrato** | |
| next | true | — | no, a meno che il nodo non decida di reinserirlo | 1 |
| | false | true | si | 2 |
| | | false | no | |
| ultimateReceiver | true | — | — | 3 |
| | false | — | — | |
| none | — | — | si | 4 |

Tabella 3.1: Attibuti del blocco e comportamento del nodo

Caso 1 (Role=next, MustUnderstand=true): il valore dell'attributo Relay non è importante poiché il nodo ha processato correttamente il blocco (data l'ipotesi che in caso MustUnderstand=true il nodo riesca sempre a processare il blocco). Ogniqualvolta un nodo processa correttamente un blocco, quest'ultimo deve essere tolto dal messaggio, a meno che il nodo non decida di reinserirlo nel messaggio, inoltrandolo così al nodo successivo.

Caso 2 (Role=next, MustUnderstand=false): il nodo può decidere se processare il blocco oppure no. In tal caso, per sapere se il blocco possa essere tolto dall'header del messaggio (anche se non è stato processato) o se debba essere lasciato (reinserito), entra in gioco l'attributo Relay che determina il comportamento del nodo: se Relay è true il blocco dovrà essere reinserito in modo che tutti i nodi intermediari abbiano la possibilità di esaminarlo.

Caso 3 (Role=ultimateReceiver): l'unico nodo che può processare il blocco è il destinatario (ultimo nodo sul percorso del messaggio) ed in questo caso Relay non è rilevante poiché il messaggio non può più essere inoltrato.

Caso 4 (Role=none): il valore none di Role impone che nessun nodo processi direttamente il blocco, il quale può contenere informazioni supplementari di qualche genere. In questo caso il blocco deve essere inoltrato da

tutti i nodi che si trovano sul percorso del messaggio.

**Body.** La sezione Body è la parte in cui si trova l'informazione principale trasportata dal messaggio SOAP dal mittente al destinatario. E' strutturata secondo il formato XML e può contenere sia semplici dati che chiamate a procedura (RPC).

**Body**

```
<env:Body>
  ... (Dati in formato XML)
</env:Body>
```

Inoltre questa sezione può essere usata per trasportare informazione relativa a situazioni di Fault; in tal caso l'elemento utilizzato, che deve essere immediato figlio dell'elemento Body, è appunto **Fault**.
Fault è costituito dai sottoelementi obbligatori **Code** e **Reason** e dagli opzionali **Detail** e **Node**. Code e Reason specificano quale dato contenuto nel messaggio ha generato l'errore mentre Detail riporta informazioni relative all'applicazione che lo ha rilevato; infine Node identifica il nodo nel quale si è verificato il Fault.
La sezione Fault può essere presente a fianco di altri dati trasportati all'interno di Body, ma nella maggior parte dei casi essa viene usata da sola all'interno di messaggi di risposta che comunicano situazioni d'errore, come nell'esempio seguente.

**Fault**

```
<env:Body>
  <env:Fault>
    <env:Code>
      <env:Value>...</env:Value>
      <env:Subcode>
        <env:Value>...</env:Value>
      </env:Subcode>
    </env:Code>
    <env:Reason>
      <env:Text>...</env:Text>
    </env:Reason>
    <env:Detail>
      ...
    </env:Detail>
    <env:Node>
      ...
    </env:Node>
  </env:Fault>
</env:Body>
```

Riportiamo quindi la struttura completa di un messaggio SOAP.

**Messaggio SOAP**

```
<env:Envelope
    xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header xmlns:env="http://www.w3.org/2003/05/soap-envelope">
   <b1:Block1
      xmlns:b1="http://example.org/2001/06/tx"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
      env:mustUnderstand="false"
      env:relay="true">
     ...
   </b1:Block1>
   <b2:Block2
      xmlns:b2="http://example.org/2001/06/tx"
      env:role="http://www.w3.org/2003/05/
                soap-envelope/role/ultimateReceiver"
      env:mustUnderstand="true">
     ...
   </b2:Block2>

   ...

  </env:Header>
  <env:Body>

    ... (Dati in formato XML)

    <env:Fault>
      <env:Code>
        <env:Value>...</env:Value>
        <env:Subcode>
          <env:Value>...</env:Value>
        </env:Subcode>
      </env:Code>
      <env:Reason>
        <env:Text>...</env:Text>
      </env:Reason>
      <env:Detail>
        ...
      </env:Detail>
      <env:Node>
        ...
      </env:Node>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

### SOAP e Remote Procedure Calls

Uno degli obiettivi di SOAP, oltre al trasferimento di dati, è l'incapsulamento all'interno dei messaggi delle chiamate a procedure remote (Remote Procedure Calls). Le informazioni necessarie ad invocare una SOAP RPC sono le seguenti:

- Indirizzo del nodo SOAP a cui è indirizzata la chiamata.
- Nome della procedura o metodo da invocare.
- Parametri con relativi valori da passare al metodo e parametri di output.

- Trasmissione (opzionale) della signature del metodo.
- Dati (opzionali) che possono essere stati trasportati come parte dei blocchi header.

SOAP RPC stabilisce come inserire e rappresentare queste informazioni all'interno del corpo di un messaggio SOAP. Prima di tutto specifica come viene mappata l'interfaccia del metodo in strutture request/response, le quali vengono poi codificate come XML. La struttura relativa alla richiesta (request) viene chiamata con lo stesso nome del metodo che è invocato. Al suo interno sono contenuti tanti elementi quanti sono i parametri di input e input/output; tali elementi sono denominati con lo stesso nome dei parametri ai quali fanno riferimento ed appaiono nello stesso ordine che questi ultimi hanno nell'interfaccia del metodo. Allo stesso modo viene modellata la risposta del metodo (response) con la differenza che il nome associato alla struttura è per convenzione il nome di tale metodo seguito da "Response". All'interno di tale struttura si trova un elemento contenente il valore di ritorno del metodo e tanti elementi quanti sono i parametri di output e input/output.

Vediamo un esempio. Prendiamo in considerazione la seguente interfaccia di un metodo Java, dove consideriamo x parametro di input ed y parametro di input/output.

```
public static int somma(int x, int y)
```

La struttura request corrispondente è la seguente

```
struct somma {
  int x;
  int y;
}
```

Questa struttura viene poi mappata in XML seguendo le regole definite a questo scopo dalla specifica di SOAP e ciò che otteniamo e che sarà inserito nel corpo del messaggio è il seguente codice (assumendo ad esempio x=1 e y=2):

```
<somma>
  <x>1</x>
  <y>2</y>
</somma>
```

Il messaggio di risposta relativo avrà nel corpo il seguente frammento:

```
<sommaResponse>
  <result>3</result>
  <y>2</y>
</sommaResponse>
```

Abbiamo visto quindi due possibili modi di utilizzare la tecnologia SOAP: *Document* e *RPC*. Nel primo caso il corpo del messaggio contiene un documento XML, mentre nel secondo vi si trova una rappresentazione in XML di una chiamata a procedura.

### SOAP e Allegati

Lavorando con i Web Services si può avere la necessità di inviare assieme al messaggio uno o più file in allegato. Ciò accade ad esempio quando abbiamo la necessità di spedire file di tipo binario, poiché essi non possono essere incapsulati all'interno di un documento XML. Possono essere allegati file di qualsiasi tipo, come ad esempio .mp3, .jpg o .mpg, ma anche documenti XML. Sebbene documenti di tipo XML possano essere inseriti direttamente all'interno del corpo di un messaggio SOAP, si può decidere di inviarli come allegato per una qualche scelta di progetto.

Le tecnologie più utilizzate per l'invio di allegati con SOAP sono MIME (Multipurpose Internet Message Encapsulation) e DIME (Direct Internet Message Encapsulation), di cui, per il nostro approfondimento su "SOAP e Allegati", prendiamo in considerazione il primo.

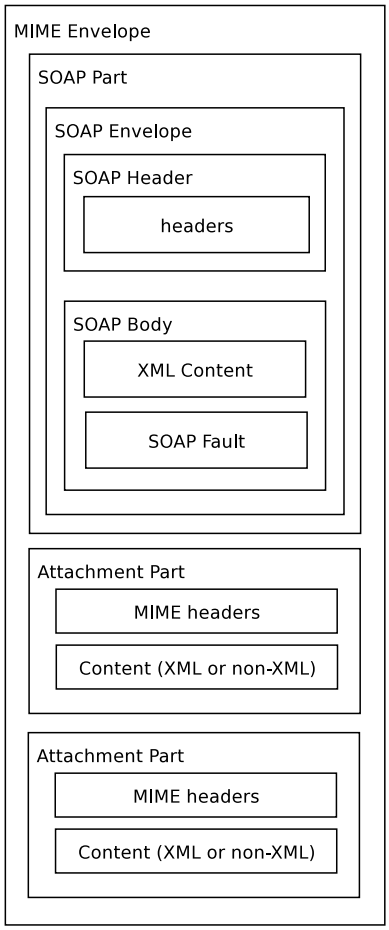


Figura 3.2: MIME-SOAP Message.

Quando si invia un messaggio con allegati, questo viene inserito all'interno di un "MIME Envelope", come si può vedere in figura 3.2. Un MIME Envelope è una sorta di contenitore suddiviso in più parti: la prima parte è quella relativa al messaggio SOAP classico, di cui abbiamo precedentemente visto una rappresentazione della struttura in figura 3.1; oltre al messaggio SOAP principale vi sono altri elementi (Attachment Part) all'interno dei quali si trovano gli allegati.

Una Attachment Part è suddivisa in due sezioni: Header e Content. In **Header** si trovano alcuni campi che riportano informazioni sul contenuto dell'allegato:

- Content-Type (obbligatorio)
- Content-ID (opzionale)
- Content-Transfer-Encoding (opzionale)
- Content-Location (opzionale)

Content-Type, unico ad essere obbligatorio, specifica il tipo dell'allegato: image/tiff, image/gif, text/xml o altri. Gli altri, tutti opzionali ma in certi casi molto utili, definiscono rispettivamente l'identificativo, la codifica (binary, base64 o altre) e la locazione dell'allegato.
Nella sezione **Content** si trova invece il codice dell'allegato.
Possiamo vedere di seguito un'esempio di messaggio SOAP con un allegato.

**Messaggio SOAP con allegato**

```
MIME-Version: 1.0
Content-Type: Multipart/Related;
              boundary="--MIME_boundary--";
              type=text/xml;
              start="<id_starting_point>"

--MIME_boundary--

Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <id_starting_point>

  <?xml version='1.0' ?>
  <env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
    <env:Body>
      ...
      <attach_ref href="cid:attach_id"/>
      ...
    </env:Body>
  </env:Envelope>

--MIME_boundary--

Content-Type: image/tiff
Content-Transfer-Encoding: binary
```

```
Content-ID: <attach_id>

[...binary TIFF image...]

--MIME_boundary--
```

Il messaggio è suddiviso in più parti, separate da stringhe particolari che qui identifichiamo con la stringa "–MIME_boundary–". In testa al messaggio troviamo la versione di MIME e il Content-Type. All'interno di quest'ultimo si specifica che il tipo (Multipart/Related) utilizzato per costruire il SOAP Message Package, e vengono definiti il boundary, cioè il limite che separa due parti (cui abbiamo già accennato), il tipo del messaggio (es.: text/xml) e l'identificativo del punto d'inizio (¡starting_point_id>).
Subito dopo il primo boundary troviamo, preceduta da alcuni campi di intestazione come l'ID, la parte principale del messaggio SOAP, di cui abbiamo già visto la struttura.
La parte relativa all'allegato, separata da quella principale da un secondo boundary, è anch'essa costituita da campi di intestazione seguiti dal codice che costituisce il file allegato. Il campo identificativo Content-ID può essere usato come riferimento dal testo nella sezione Body del messaggio SOAP.

**Differenze tra le specifiche MIME e DIME.** MIME usa, come abbiamo visto, stringhe speciali per separare le parti relative agli allegati; inoltre permette di aggiungere ulteriori metadati nel messaggio creando campi d'intestazione personalizzati. Al contrario DIME non utilizza separatori ma memorizza, in campi appositi nelle intestazioni, la lunghezza di ciascuna parte del messaggio; ha un piccolo e prestabilito insieme di campi di intestazione che assicura le caratteristiche richieste per un messaggio SOAP. Il formato più semplice di DIME lo rende meno flessibile ma più efficiente rispetto a MIME in termini di velocità nel momento in cui il messaggio viene processato. Se volessimo sapere ad esempio il numero degli allegati in un messaggio MIME, dovremmo leggere quest'ultimo completamente; al contrario in un messaggio DIME sarebbe sufficiente contare il numero di record. In figura 3.3 è riportato un esempio di record DIME.

La descrizione dei campi contenuti nel record DIME è riportata in tabella 3.2.

### 3.1.4 UDDI

Abbiamo visto come i Web Services vengono descritti (WSDL, 3.1.2) ed in quale modo essi comunicano (SOAP, 3.1.3). Ma, a questo punto, una cosa che ci appare evidente è il fatto che due Web Service possono comunicare tra loro solo se l'uno conosce locazione e modalità di accesso dell'altro; ciò che manca è un sistema che renda possibile la ricerca di Web Service, secondo certi criteri come ad esempio la tipologia del servizio, sapere cioè quali mettono a disposizione una certa funzionalità, o l'appartenenza ad una data

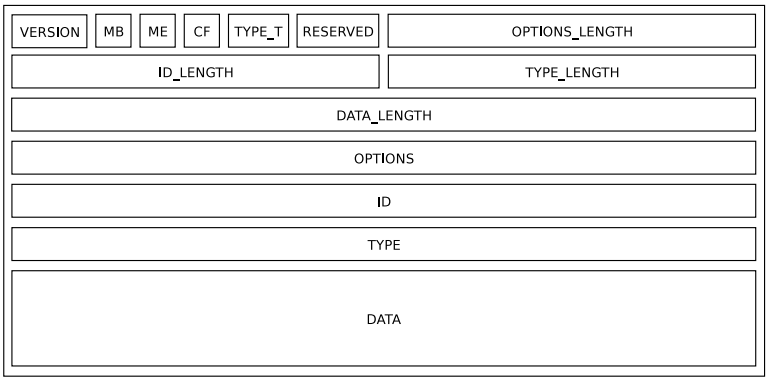


Figura 3.3: Record DIME.

azienda. La tecnologia che si occupa di questo aspetto è UDDI.
**UDDI (Universal Description, Discovery and Integration)**, che è basato su XML ed utilizza SOAP per le comunicazioni da e verso l'esterno, definisce un meccanismo comune per pubblicare e trovare informazioni sui Web Services, in base alle loro descrizioni WSDL. Ciò che UDDI mette a disposizione è un registro nel quale si possa accedere, attraverso specifiche funzioni, per

- pubblicare servizi che un'azienda rende disponibili,
- cercare aziende che mettono a disposizione un certo tipo di servizio,
- avere informazioni "Human Readable", cioè comprensibili all'utente, circa indirizzi, contatti o altro relativi ad una azienda.
- avere informazioni tecniche "Machine Readable", cioè interpretabili ed utilizzabili dalla macchina, relative ad un servizio in modo tale da potervisi connettere.

Un registro UDDI è costituito in realtà da un database distribuito, cioè da molti registri distribuiti sulla rete, ognuno dei quali si trova sul server di una azienda che contribuisce allo sviluppo di questo archivio pubblico, e connessi fra loro. Il sistema mantiene una centralizzazione virtuale, cioè l'informazione che viene registrata su uno dei nodi (registri) del sistema viene propagata e resa disponibile su tutti gli altri tramite una loro sicronizzazione (da [30]). Questo, oltre che ad alleggerire il carico di lavoro che un singolo nodo deve sopportare, contribuisce alla protezione del sistema contro possibili situazioni di failure del database, grazie alla ridondanza dei dati.

Per semplificità consideriamo UDDI come un unico grande registro. Esso può essere visto come le pagine gialle, nelle quali cerchiamo informazioni sulle aziende che attraverso esse ottengono maggiore visibilità e di conseguenza la possibilità di avere un più alto numero di clienti.

| Campo | Descrizione |
|---|---|
| VERSION | Versione del messaggio DIME. |
| MB | Specifica se questo record è il primo record del messaggio. |
| ME | Specifica se questo record è l'ultimo record del messaggio. |
| CF | Specifica che il contenuto del messaggio è stato suddiviso in parti. |
| TYPE_T | Struttura e formato del campo TYPE. |
| RESERVED | Riservato per uso futuro. |
| OPTIONS_LENGTH | Lunghezza in bytes del campo OPTIONS. |
| ID_LENGTH | Lunghezza in bytes del campo ID. |
| TYPE_LENGTH | Lunghezza in bytes del campo TYPE. |
| DATA_LENGTH | Lunghezza in bytes del campo DATA. |
| OPTIONS | Contiene tutte le informazioni opzionali utilizzate da un parser DIME. |
| ID | Contiene un URI (Uniform Resource Identifier) per l'identificazione univoca del record. |
| TYPE | Specifica la codifica dei dati contenuti nel record. |
| DATA | Contiene i dati trasportati dal record. |

Tabella 3.2: Descrizione dei campi del record DIME.

La registrazione di un servizio all'interno di un UDDI Registry è costituita da 3 parti:

- Yellow Pages
- White Pages
- Green Pages

Con le **Yellow Pages** (pagine gialle) le aziende ed i loro servizi vengono catalogati sotto differenti categorie e classificazioni.
Nelle **White Pages** (pagine bianche) possiamo trovare informazioni come gli indirizzi di una azienda o contatti ad essa relativi.
Infine vi sono le **Green Pages** (pagine verdi), dove sono contenute informazioni tecniche relative ai servizi, grazie alle quali questi ultimi possono essere invocati.

Questa è la suddivizione dell'informazione dentro UDDI a livello concettuale ma vediamo come è realmente strutturato un UDDI Registry.

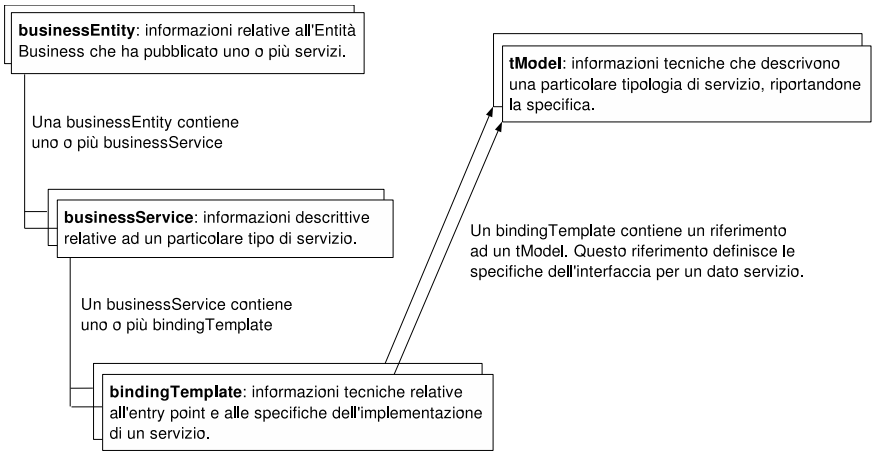


Figura 3.4: Struttura principale di un registro UDDI.

### Struttura di UDDI

Le parti principali di un registro UDDI, di cui la figura 3.4 riporta anche le relazioni esistenti fra esse, sono quattro, in ognuna delle quali è memorizzato un certo tipo di informazione:

- businessEntity: informazioni relative alle compagnie o aziende che hanno pubblicato uno o più Web Service (White Pages).
- businessService: informazioni descrittive relative ad un particolare servizio (Yellow Pages).
- bindingTemplate: informazioni tecniche relative ad un Web Service, come ad esempio l'*entry point* (Green Pages).
- tModel: informazioni tecniche che descrivono una particolare tipologia di servizio, riportandone la specifica dell'interfaccia (Green Pages).

Una *businessEntity* possiede informazioni come nome, indirizzo e contatti di una azienda che ha pubblicato dei Web Service. Ad ogni businessEntity fanno riferimento uno o più *businessService*, ognuno dei quali memorizza dati di tipo descrittivo, come nome e categoria di appartenenza, circa uno dei Web Service di tale azienda. I dettagli tecnici relativi alla locazione del servizio ed alle modalità di accesso si trovano in una o più strutture, a cui questo businessService si riferisce, che si chiamano *bindingTemplate*; un binding (cioè un legame) mette in relazione un businessService, cioè le informazioni generiche descrittive di un servizio, con una sua reale implementazione. Infine completano l'architettura di UDDI i tModels. Un *tModel* definisce la specifica di un certo tipo di servizio, precisando ad esempio l'interfaccia che esso deve o dovrà avere; il termine "dovrà" sta appunto a chiarire il fatto che un servizio, specificato da un tModel, non deve necessariamente essere

stato implementato ma può anche essere solamente stato ideato in termini di specifiche che una sua implementazione è tenuta a rispettare. Per fare ciò un tModel memorizza l'URL di un documento WSDL che riporta appunto le caratteristiche del servizio. Nel momento in cui viene realizzato un servizio, i bindingTemplate (uno o più) ad esso relativi faranno riferimento ad uno o più tModel, a precisare che esso rispetta un certo tipo di specifiche.

### Interazioni con UDDI

Le interazioni con un registro UDDI sono suddivisibili in due categorie: quelle finalizzate alla ricerca di dati e quelle il cui scopo è l'inserimento o la modifica degli stessi. Questa suddivizione è rappresentata da due API (Application Programming Interface) tramite le quali si accede ai contenuti di UDDI: *Inquiry API* e *Publish API*.

**Inquiry API** è utilizzata per la ricerca di informazioni tramite i metodi seguenti, di cui si riportano i dati restituiti (da[14]):

- `find_binding`: bindings relativi a un businessService.
- `find_business`: informazioni su uno o più business.
- `find_relatedBusinesses`: informazioni relative a businessEntity collegate ad una specifica di cui viene fornito l'identificativo.
- `find_service`: trova i servizi di una data businessEntity.
- `find_tModel`: trova uno o più tModel.
- `get_bindingDetail`: informazioni di un bindingTemplate, utilizzabili per effettuare invocazioni ad un servizio.
- `get_businessDetail`: informazioni di un businessEntity.
- `get_businessDetailExt`: informazioni, in forma completa, di un businessEntity.
- `get_serviceDetail`: dettagli di un businessService.
- `get_tModelDetail`: dettagli di un tModel.

**Publish API**, utilizzata invece per l'inserimento, l'aggiornamento e la cancellazione, richede un'autorizzazione per accedere ai dati e mette a disposizione i seguenti metodi, di ognuno dei quali viene specificata la funzionalità realizzata (da [14]); il termine "publisher", di cui si fa uso, identifica un soggetto, i cui dati sono memorizzati nel sistema UDDI, che possiede l'autorizzazione per accedere al registro, tramite la Publish API, e manipolare le informazioni relative ai servizi da lui pubblicati. Per ottenere l'accesso ai

dati vengono utilizzati degli "authToken" (Authorization Token, cioè gettoni di autorizzazione), i quali altro non sono che stringhe di caratteri il cui utilizzo equivale a quello della classica funzione di login.

- `add_publisherAssertions`: Aggiunge una publisherAssertion all'insieme di Assertions esistenti; una "assertion" definisce una relazione tra due registrazioni di servizi, che esprime un certo tipo di legame esistente fra loro: una relazione possibile può essere quella di sottoservizio, cioè un servizio si occupa di implementare una parte di un servizio più generale.

- `delete_binding`: cancella un bindingTemplate dall'insieme dei bindingTemplate che fanno parte della struttura di un dato businessService.

- `delete_business`: cancella le informazioni relative ad un dato businessEntity.

- `delete_publisherAssertions`: cancella specifiche publisherAssertions dall'insieme di publisherAssertions relative ad un publisher.

- `delete_service`: cancella un dato businessService.

- `delete_tModel`: nasconde un tModel; in questo modo, tale tModel non può più essere restituito da una ricerca *find_tModel* ma ad esso è ancora possibile fare riferimento ed accedervi tramite la funzionalità *get_tModelDetail*. Non esiste una funzionalità che cancelli un tModel definitivamente.

- `discard_authToken`: informa l'Operator Node (cioè il sistema che gestisce il nodo UDDI) che un dato token di autenticazione non è più valido ed una richiesta di accesso tramite tale token dovrà essere rifiutata fintantoché esso non sarà riattivato dall'operatore.

- `get_assertionStatusReport`: restituisce un resoconto, status report, relativo alle publisherAssertions già stabilite (visibili) ed allo stato di quelle non ancora corrisposte.

- `get_authToken`: richiede un token di autenticazione ad un Operator Node. I token di autenticazione vengono utilizzati dai metodi di Publish per accedere ai dati e rappresentano il login al registro UDDI.

- `get_publisherAssertions`: restituisce la lista delle publisherAssertion (visibili) di un dato publisher.

- `get_registeredInfo`: restituisce un sommario di tutte le informazioni inserite da un publisher.

- `save_binding`: registra nuovi bindingTemplate o aggiorna i dati di quelli esistenti.

- `save_business`: registra nuovi businessEntity o aggiorna i dati relativi a quelli già presenti.
- `save_service`: registra o aggiorna le informazioni relative ad un businessService.
- `save_tModel`: registra o aggiorna le informazioni relative ad un tModel.
- `set_publisherAssertions`: salva l'insieme completo delle publisherAssertions relative ad un publisher sostituendo quelle esistenti, che vengono così cancellate.

Come abbiamo già accennato, un servizio UDDI è basato su XML e comunica con gli altri Web Service attraverso messaggi SOAP. Per avere un'idea di quali sono e come vengono strutturate le informazioni restituite ad un Web Service da parte di un UDDI Registry, in seguito ad un'operazione di inquiry, vediamo i due seguenti messaggi SOAP, rispettivamente di richiesta e di risposta.

**Request Message**

```
<?xml version="1.0" encoding="UTF-8"?>
  <soapenv:Envelope
     xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
      <get_businessDetailExt generic="2.0"
         xmlns="urn:uddi-org:api_v2">
        <businessKey>
           6738A270-8718-11D9-BE19-E3A8EDD29D26
        </businessKey>
      </get_businessDetailExt>
    </soapenv:Body>
  </soapenv:Envelope>
```

Nel messaggio di richiesta si trova il costrutto "get_businessDetailExt", usato per ottenere tutte le informazioni riguardanti un servizio, ed al suo interno l'elemento "businessKey" che specifica l'identificativo di tale servizio.

**Response Message**

```
<?xml version="1.0" encoding="utf-8"?>
  <soapenv:Envelope
     xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
      <businessDetailExt generic="2.0"
         operator="eurodyn.com"
         xmlns="urn:uddi-org:api_v2">
        <businessEntityExt>
          <businessEntity authorizedName="ROOT"
```

```
        businessKey="6738A270-8718-11D9-BE19-E3A8EDD29D26"
        operator="eurodyn.com">
      <discoveryURLs>
        <discoveryURL useType="businessEntity">
           http://localhost:8080/juddi//uddiget.jsp?businesskey=
             6738A270-8718-11D9-BE19-E3A8EDD29D26
        </discoveryURL>
      </discoveryURLs>
      <name>ISTI Web Services</name>
      <contacts>
        <contact useType="publisher">
          <personName>ROOT</personName>
        </contact>
      </contacts>
      <businessServices>
        <businessService
           businessKey="6738A270-8718-11D9-BE19-E3A8EDD29D26"
           serviceKey="7CD2E0A0-8718-11D9-BE19-CCA88D805CB7">
          <name>MergeService</name>
          <bindingTemplates>
            <bindingTemplate
               bindingKey="D38E48A0-87E8-11D9-B722-A26BE7DE4B86"
               serviceKey="7CD2E0A0-8718-11D9-BE19-CCA88D805CB7">
              <accessPoint URLType="http">
                 http://localhost:8080/axis/services/MergeService
              </accessPoint>
              <tModelInstanceDetails>
                <tModelInstanceInfo
                   tModelKey="uuid:B5CA6C80-8717-11D9-BE19-FAE3FA307183">
                  <instanceDetails>
                    <overviewDoc>
                      <overviewURL>
                         http://localhost:8080/axis/services/MergeService?wsdl
                      </overviewURL>
                    </overviewDoc>
                  </instanceDetails>
                </tModelInstanceInfo>
              </tModelInstanceDetails>
              <categoryBag/>
            </bindingTemplate>
          </bindingTemplates>
          <categoryBag/>
        </businessService>
      </businessServices>
      <identifierBag/>
      <categoryBag/>
    </businessEntity>
  </businessEntityExt>
</businessDetailExt>
</soapenv:Body>
</soapenv:Envelope>
```

Il messaggio di risposta contiene tutte le informazioni di una registrazione UDDI relativa ad un'entità come un'azienda o una compagnia. Si può notare come nella struttura ad albero delle informazioni siano implicitamente riportate le relazioni di inclusione e appartenenza dei quattro elementi principali, *businessEntity*, *businessService*, *bindingTemplate* e *tModel*, rappresentati in figura 3.4.
Entrando nel dettaglio vediamo quali sono i sottoelementi e gli attributi

rilevanti dei vari elementi che compongono il corpo del messaggio:

- **<businessEntity>**
  - businessKey, identificatore dell'entità.
  - operator, nodo operatore di UDDI.
  - <name>, *nome della compagnia*
  - <discoveryURLs>, *URL relativi alla compagnia*
  - <discoveryURL>, *singolo URL*
  - <businessServices>, *elenco dei businessService della compagnia*
- **<businessService>**
  - businessKey, identificatore dell'entità di cui fa parte.
  - serviceKey, identificatore del businessService.
  - <name>, *nome del servizio*
  - <bindingTemplates>, *elenco dei bindingTemplates relativi al servizio*
  - <categoryBag>, *categorizzazione del servizio*
- **<bindingTemplate>**
  - bindingKey, identificatore del bindingTemplate
  - serviceKey, identificatore del servizio a cui si riferisce.
  - <accessPoint>, *URL di una implementazione del servizio*
  - <tModelInstanceDetails>, *elenco dei tModel a cui fa riferimento*
- **<tModelInstanceInfo>**
  - tModelKey, identificatore del tModel.
  - <instanceDetails>, *dettagli del tModel*
  - <overviewDoc>, *documenti relativi al tModel*
  - <overviewURL>, *URL relativo ad un documento che descrive il tModel (ad esempio un file WSDL)*

## 3.2 Tecnologie per Web Services di seconda generazione

Nelle sezioni precedenti abbiamo descritto le tecnologie standard dei Web Services e come, grazie ad esse, sia possibile descrivere, ricercare ed invocare un servizio. Ogni servizio mette a disposizione alcune funzionalità, le quali però sono, in relazione alla complessità del compito che svolgono, di basso livello. Ciò che queste tecnologie non forniscono è un punto di vista più

elevato, al di sopra dei Web Service, che permetta di vedere ogni servizio come parte di un processo più grande e complesso ottenuto dalla collaborazione di molti servizi web. Una tale collaborazione, che può coinvolgere più organizzazioni mosse da un'obiettivo comune, viene definita "business process".

I termini "Orchestration" e "Choreography" identificano due attività che si occupano di descrivere un *business process* da punti di vista differenti.
**Orchestration:** orchestrazione o coordinamento, descrive il processo business ed in particolare la sua logica in termini del flusso di esecuzione controllato da una singola parte. Definisce le interazioni, costituite da scambi di messaggi, che vi possono essere fra i servizi web e stabilisce il loro ordine di esecuzione.
**Choreography:** coreografia, definisce la sequenza dei messaggi che può coinvolgere i vari web service. In tale sequenza viene identificato il ruolo che ogni parte svolge all'interno del processo. Il processo è quindi descritto a livello collaborativo, rappresentando la cooperazione fra i servizi che ne fanno parte. [21, 22]

La differenza importante fra queste due attività risiede, come possiamo già aver intuito, nel punto di vista da cui si osserva il processo. Per "Orchestration" il punto di vista è quello relativo ad una delle parti che costituiscono il processo, in particolare quella che ne detiene il controllo mentre per "Choreography" è quello generale di una visione della cooperazione fra i vari servizi.

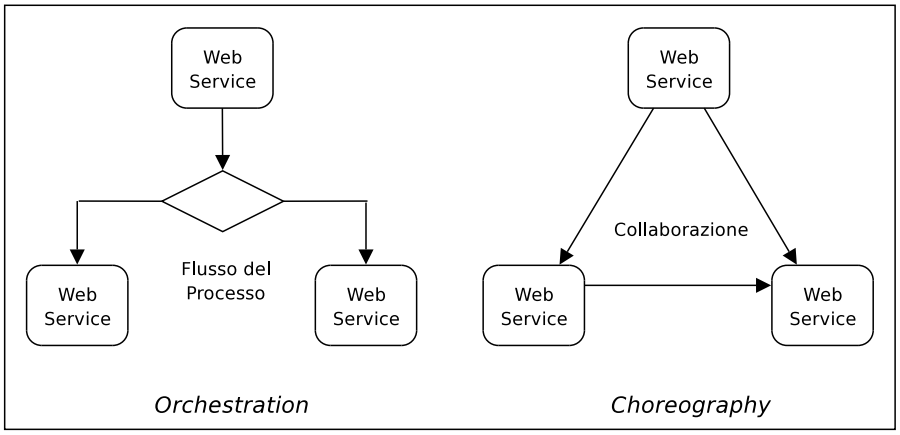


Figura 3.5: Orchestration e Choreography.

Quando si progettano e sviluppano business process che coinvolgono molti web service e la cui esecuzione può avere un tempo di durata elevato bisogna tenere presenti alcuni requisiti tecnici molto importanti. Per prima cosa è necessario che i servizi possano essere invocati in modo asincrono; in

questo modo un business process può invocare concorrentemente più web service per migliorare le prestazioni. Devono poi essere gestite le eccezioni e l'integrità delle transazioni, definendo il comportamento del sistema in caso di errore o di mancata risposta di un servizio invocato entro un certo limite di tempo. Infine caratteristiche importanti nell'attività di coordinamento dei web service sono dinamicità, flessibilità, adattabilità e riutilizzabilità; tenendo separata la logica del processo dai web service che lo realizzano si ottiene un alto livello di flessibilità che permette al business process di essere dinamicamente adattato per vari scopi, semplicemente richiamando i servizi necessari; inoltre un business process può essere visto come un qualsiasi servizio ed essere invocato da altri, riutilizzandolo all'interno di vari processi costituiti così da una composizione ricorsiva di web service. [21, 22]

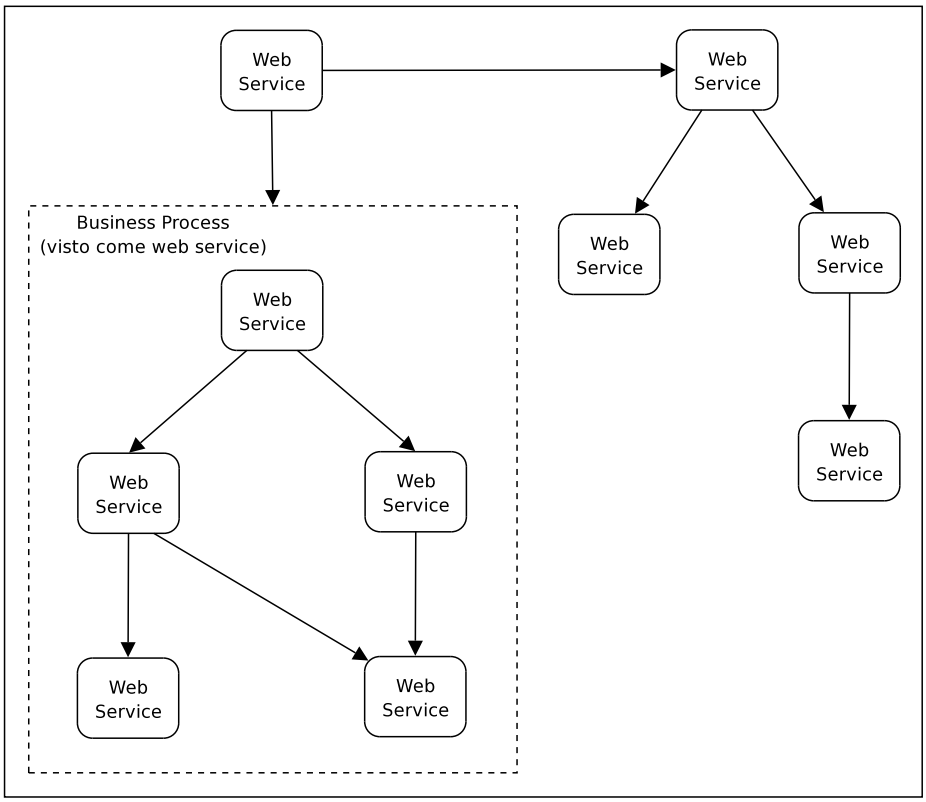


Figura 3.6: Business process e composizione ricorsiva di web service.

Vi sono molte tecnologie, sviluppate autonomamente o in collaborazione da grandi organizzazioni come IBM, Microsoft, BEA, SUN o altre, il cui scopo è quello di gestire le due attività di *Orchestration* e *Choreography*. All'interno di queste tecnologie la distinzione fra Orchestration e Choreography spesso si perde, perciò nel seguito considereremo semplicemente la **Web Services Orchestration** come l'attività che le comprende entrambe.

[21]

Le tecnologie più promettenti in ambito di *Web Services Orchestration* sono **BPEL4WS**, **WSCI** e **BPML**.

# Capitolo 4

# Web Services: tecnologie

L'implementazione e l'esecuzione di web service richiedono l'adozione di alcune tecnologie necessarie al loro funzionamento. Di tali tecnologie fanno parte ad esempio il supporto al linguaggio di programmazione e il web server.
Grande parte del tempo impiegato per la realizzazione di questo progetto è stato dedicato allo studio di queste tecnologie, per comprendere la loro struttura ed il loro funzionamento e per effettuare correttamente la loro installazione e configurazione. Le tecnologie utilizzate sono state scelte in base a semplici criteri di valutazione del prodotto come il fatto di essere software Open Source o meno, la libertà di utilizzo, di modifica e di personalizzazione, chiarezza e semplicità d'uso, costo ridotto o nullo ed infine inesistenza di alternative valide.

Il punto di partenza, nella preparazione della macchina su cui è stato realizzato il software, è stata la scelta del sistema operativo. In base ad alcuni dei criteri appena citati, quali codice Open Source, libertà di utilizzo e gratuità, è stato adottato Linux, in particolare la distribuzione Mandrake 10.1.

Vediamo adesso una per una le tecnologie utilizzate spiegando quale compito assolvono, come funzionano e quali sono i passi da seguire per una corretta installazione, su macchina Linux.

## 4.1 JAVA

Il linguaggio di programmazione scelto per l'implementazione dei web service che compongono il progetto è JAVA. Linguaggio molto diffuso, Java è utilizzabile pressoché su qualsiasi macchina semplicemente installando l'opportuna JVM (Java Virtual Machine), disponibile per i principali sistemi operativi, quali Windows, Linux, Mac o Solaris.
Il supporto Java utilizzato per questo progetto è il J2SDK 1.4.2_05 (Java 2

Starndard Development Kit, versione 1.4.2_05).
E' stato scelto Java come linguaggio di programmazione perché, oltre al fatto che è semplice trovare una grande quantità di informazioni che lo descrivono, è su di esso che sono basati molti dei software che abbiamo adottato e che vedremo più avanti.

### 4.1.1 Installazione di JAVA

Per l'installazione del supporto Java è stato utilizzato il pacchetto rpm j2sdk-1_4_2_05-linux-i586.rpm, scaricato da Internet.
Lanciando da shell il comando

```
rpm -Uvh j2sdk-1\_4\_2\_05-linux-i586.rpm
```

verrà installata la JVM nel sistema; in particolare viene creata in /usr-/java la cartella "j2sdk1.4.2_05".
Viene richiesta la password di *root* poiché si sta installando il software in cartelle con accesso in scrittura non permesso agli utenti standard; tale software sarà reso disponibile a tutti gli utenti.
Sono poi necessarie alcune modifiche al file /etc/profile, dove devono essere aggiunte alcune linee che specificano al sistema, al momento dell'avvio, la cartella HOME di Java ed includono nel PATH i file eseguibili che permettono ad esempio l'esecuzione, $JAVA_HOME/bin/java, o la compilazione, $JAVA_HOME/bin/javac, di codice java.
Le linee da aggiungere sono le seguenti:

**Modifiche al file /etc/profile**

```
JAVA_HOME = ‘‘/usr/java/j2sdk1.4.2_05’’
PATH = $PATH:$JAVA_HOME/bin
export JAVA_HOME PATH
```

Possiamo adesso effettuare il test dell'installazione di java. Per fare questo è necessario prima di tutto chiudere la sessione e rientrare in modo che il sistema possa aggiornare il PATH. Dopodiché lanciamo da shell il comando *java*, o altro comando disponibile dopo l'installazione, quale *javac*, *jdb*, *javap*; se l'installazione di java ha avuto successo, tale comando viene trovato e ciò che viene restituito in output è l'help del comando stesso, altrimenti il sistema risponde con un messaggio del tipo "command not found", cioè comando non trovato.

## 4.2 Tomcat

Tomcat è un application server basato sulla tecnologia java, un web server ed un contenitore di servlet e JSP. Tomcat è necessario per fare del nostro pc o del computer che vogliamo connettere alla rete un server che possa ospitare le nostre applicazioni che desideriamo siano utilizzate da altri utenti via

web. Su Tomcat possono essere caricate semplici pagine HTML come anche servlet e JSP. Relativamente semplice da configurare ed utilizzare, Tomcat è *Open Source*, cioè con codice liberamente visibile e modificabile, al contrario di prodotti analoghi, come ad esempio IIS (Internet Infomation Service) di Microsoft ed inoltre è liberamente scaricabile ed utilizzabile.
La versione di Tomcat scelta è la 5.0.27. Vediamo di seguito i passi da seguire per una corretta installazione.

### 4.2.1 Installazione di Tomcat

Il pacchetto utilizzato per l'installazione di Tomcat è jakarta-tomcat-5.0.27-.tar.gz, scaricabile dal sito di Apache.
Da shell spostiamoci nella directory /usr/java e decomprimiamo il pacchetto con i seguenti comandi (è necessario essere super-user):

```
cd /usr/java
tar -xzf jakarta-tomcat-5.0.27.tar.gz
```

Il comando *tar* estrae tutti i file creando la cartella principale "jakarta-tomcat-5.0.27".
Dobbiamo adesso eseguire alcune modifiche al file /etc/profile in modo da informare il sistema circa la locazione degli eseguibili di tomcat.
Le linee da aggiungere sono le seguenti:

**Modifiche al file /etc/profile**

```
CATALINA_HOME = ‘‘/usr/java/jakarta-tomcat-5.0.27’’
PATH = $PATH:$CATALINA_HOME/bin
export CATALINA_HOME PATH
```

Adesso possiamo riavviare il sistema e testare l'installazione di tomcat. Per avviare e terminare il servizio tomcat dobbiamo utilizzare, da linea di comando, le seguenti istruzioni:

**Avvio ed Arresto di Tomcat**

```
$CATALINA_HOME/bin/startup.sh
$CATALINA_HOME/bin/shutdown.sh

o pi\‘u semplicemente

startup.sh
shutdown.sh
```

*Oss.:* se le directory relative a Tomcat non appartengono all'utente che ne effettua l'avvio, i comandi restituiranno un errore poiché non si possiede l'accesso in scrittura a tali cartelle. Per poter avviare Tomcat si deve quindi eseguire i comandi, sopra riportati, da super-user oppure, per avviarli da utente standard, è necessario modificare (da super-user) proprietà e gruppo di file e cartelle di Tomcat.

Testiamo dunque l'installazione lanciando il comando startup.sh e verificando il funzionamento da browser. Aprendo un qualsiasi browser all'indirizzo http://localhost:8080/ viene presentata la pagina principale di tomcat.

La configurazione di Tomcat può essere personalizzata cambiando le impostazioni dei parametri presenti nel file server.xml in "$CATALINA_HOME-/conf/".
I file di log delle esecuzioni di Tomcat vengono invece salvati nella directory "$CATALINA_HOME/logs/".

## 4.3 Axis

Axis è una piattaforma Java per la creazione e l'utilizzo di applicazioni che si basano sui Web Services, cioè il motore per l'esecuzione di web service.
Axis è una servlet che, per la sua esecuzione, necessita di un contenitore di servlet che nel nostro caso è Tomcat. Axis rappresenta il supporto ai Web Services, cioè la base su cui essi poggiano. Una possibile alternativa ad Axis è costituita da JWSDP, Java Web Service Developer Pack (SUN). Nella fase di studio e confronto di questi due software ho riscontrato una maggiore semplicità di utilizzo ed una maggiore flessibilità di Axis rispetto a JWSDP ed è per questo motivo che la scelta è ricaduta sul primo dei due.
Le caratteristiche più importanti di Axis sono:

- implementazione di SOAP 1.1/1.2,
- implementazione di WSDL,
- supporto JWS (Java Web Services),
- meccanismi per il deployment dei web service,
- supporto a differenti metodi per l'invocazione di web service (che vedremo in 4.7) come
  - Stub creato da WSDL,
  - Dynamic Proxy e
  - Dynamic Invocation Interface (DII),

  ai quali, nel caso in cui si faccia uso di un servizio UDDI (3.1.4) come jUDDI (di cui parleremo in 4.5), si aggiunge anche il metodo
  - Dynamic Discovery and Invocation (DDI),
- tool di monitoraggio delle trasmissioni di messaggi: SOAP Monitor e TCP Monitor,
- utility WSDL2Java e Java2WSDL.

Il supporto JWS costituisce una semplificazione nel deployment dei web service su Axis ma questo strumento non può però essere sempre utilizzato. Parleremo del supporto JWS e del metodo di deployment standard di Axis in 4.3.2.
Vediamo adesso invece la procedura la procedura da seguire per ottenere un'installazione corretta di Axis.

### 4.3.1 Installazione di Axis

Il pacchetto utilizzato per l'installazione di Axis è axis-1_1.tar.gz, scaricabile dal sito di Apache. Da linea di comando decomprimiamo il pacchetto in /usr/java eseguendo da super-user:

```
cd /usr/java
tar -xzf axis-1_1.tar.gz
```

Dobbiamo adesso copiare la cartella "/usr/java/axis-1_1/webapps/axis/" in "/usr/java/jakarta-tomcat-5.0.27/webapps/". Si crea così una sottodirectory "axis/" nella directory webapps di Tomcat. Questo è ciò che costituirà il supporto ai Web Services ed è appunto all'interno di questa directory (in apposite subdirectory) che verranno inseriti i programmi che implementano i nostri Web Service.
Dobbiamo adesso fare alcune modifiche a due file: il già citato /etc/profile ed il file server.xml di tomcat. Per quanto riguarda il file server.xml, situato nella directory "$CATALINA_HOME/conf/", è necessario aggiungere la seguente riga all'interno dell'elemento <HOST>:

**Modifiche al file $CATALINA_HOME/conf/server.xml**

```
<Context path=''/axis'' docBase=''axis'' debug=''0''
reloadable=''true'' />
```

Questo informa Tomcat della presenza di axis e specifica che per accedere ad esso bisogna aggiungere "/axis" all'indirizzo "http://localhost:8080" di tomcat. Inserendo l'indirizzo "http://localhost:8080/axis" nel browser si accede alla prima pagina di Axis.

Le modifiche che riguardano invece il file /etc/profile sono le seguenti:

**Modifiche al file /etc/profile**

```
AXIS_HOME = ‘‘/usr/java/axis-1_1/’’

CLASSPATH = $CLASSPATH:$CATALINA_HOME/common/endorsed/xercesImpl.jar
                      :$CATALINA_HOME/common/endorsed/xml-apis.jar
                      :$AXIS_HOME/lib/axis.jar
                      :$AXIS_HOME/lib/commons-discovery.jar
                      :$AXIS_HOME/lib/commons-logging.jar
                      :$AXIS_HOME/lib/saaj.jar
                      :$AXIS_HOME/lib/log4j-1.2.8.jar
                      :$CATALINA_HOME/common/lib/servlet-api.jar
```

che specificano la directory principale di Axis e modificano il CLASSPATH inserendo i pacchetti necessari al fuzionamento del software stesso.

A questo punto, dopo aver riavviato Tomcat, apriamo il browser ed inseriamo l'indirizzo "http://localhost:8080/axis" al quale si trova la prima pagina di Axis. Se la pagina viene visualizzata significa che Axis è presente ma dobbiamo controllare che siano presenti tutti i pacchetti necessari al suo corretto funzionamento.
Sulla prima pagina di Axis possiamo notare alcuni link il primo dei quali è *Validate*. Facendo click su di esso ci viene restituita una pagina chiamata "Axis Happiness Page" che riporta i risultati dell'analisi della configurazione di Axis. Vengono riportati i componenti necessari ed opzionali al funzionamento di Axis ed il path ai relativi pacchetti nel caso in cui siano stati trovati.

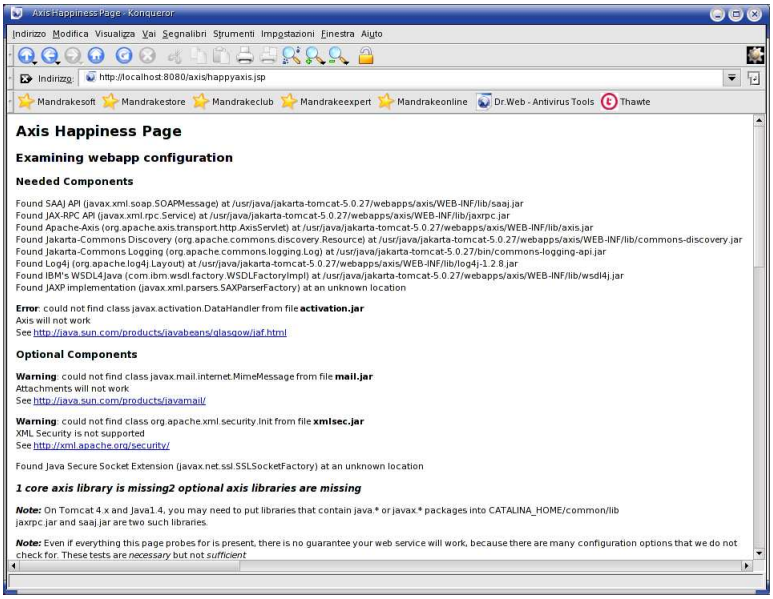


Figura 4.1: Axis Happiness Page - Alcuni componenti mancanti.

I componenti necessari devono ovviamente essere tutti presenti, poiché senza di essi Axis non funzionerà correttamente. I componenti opzionali devono essere presenti solo nei casi in cui le applicazioni da noi sviluppate ne facciano uso.
I pacchetti mancanti, nel caso ve ne siano, devono essere inseriti nella directory "$CATALINA_HOME/common/lib/"; riavviando Tomcat tali pacchetti vengono rilevati automaticamente e ciò può essere verificato semplicemente ricaricando la pagina di validazione di Axis ("Axis Happiness Page", raggiungibile dall'indirizzo http://localhost:8080/axis).

Per quanto riguarda questa specifica versione di AXIS, la 1.1, la pagina di validazione di Axis rileva la mancanza di *Activation API* fra i componenti necessari e di *Mail API* e *XML Security API* fra i componenti opzionali (vedi figura 4.1).

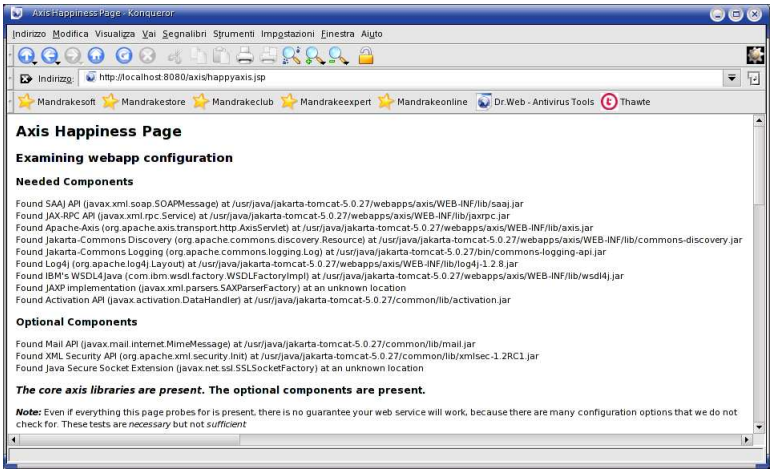


Figura 4.2: Axis Happiness Page.

La notifica della mancanza dei pacchetti è presentata da Axis assieme ai link alle pagine web dove questi possono essere trovati e scaricati. I file da me utilizzati sono jaf-1_0_2-upd.zip, al cui interno si trova il pacchetto activation.jar, javamail-1_3_2.zip, al cui interno si trova mail.jar, ed infine xmlsec-1.2RC1.jar (per xmlsec.jar).
Inserendo tali pacchetti nella directory "$CATALINA_HOME/common/lib/" si ottiene una pagina di validazione di Axis in cui tutti i componenti sono presenti, come si può vedere in figura 4.2.

### 4.3.2 Deployment di web service su Axis

Il *deployment* è un'operazione che deve essere effettuata su un web service affinché questo possa essere utilizzato. Attraverso l'operazione di deployment si notifica ad Axis la presenza di un nuovo servizio specificando il suo nome e la sua locazione.
In Axis, come già accennato in precedenza, vi sono due modi per effettuare il deployment di un servizio: [28, 3]

- deployment dinamico attraverso l'uso di file .jws,
- deployment statico attraverso l'uso di deployment descriptor.

Nel caso di un web service semplice si può utilizzare il primo tipo di deployment. Dopo aver creato il file java che implementa il web service è sufficiente sostituire l'estensione **.java** di tale file con l'estensione **.jws** e

copiarlo nella directory $CATALINA_HOME/webapps/axis/ o in una sua subdirectory. A questo punto il web service è immediatamente disponibile all'indirizzo

```
http://localhost:8080/axis/nomeFile.jws?method=nomeMetodo
```

poiché Axis tratta i file .jws in modo simile a quello in cui Tomcat tratta una JSP (Java Server Page). Ciò che Axis fa è localizzare il file, compilarlo e posizionare attorno ad esso un *wrapper*, cioè una sorta di filtro, che converte le chiamate SOAP dirette verso il servizio in invocazioni di metodi java.
E' anche possibile vedere il WSDL del servizio all'indirizzo

```
http://localhost:8080/axis/nomeFile.jws?wsdl
```

Via browser apparirà una pagina bianca della quale è necessario visualizzare il sorgente per vedere il WSDL.
Questa soluzione rappresenta un meccanismo di deployment facile ed immediato, ma che può essere utilizzata solo nei casi di web service molto semplici. Inotre tale soluzione presenta alcuni inconvenienti come ad esempio il fatto di non poter specificare quali metodi devono essere esposti e quali invece non devono esserlo. Altro svantaggio è quello di non poter definire un mapping dei tipi SOAP/Java personalizzato.
Nel caso invece di web service più complessi o nel caso in cui le caratteristiche della soluzione appena vista non siano adeguate per la nostra applicazione, vi è il metodo di deployment standard di Axis.
Questo metodo utilizza dei particolari file chiamati **Web Services Deployment Descriptor (WSDD)** all'interno dei quali sono inserite le informazioni relative al servizio di cui vogliamo fare il deployment. I file **.wsdd** permettono di specificare molte più cose (vedi [5]) riguardo all'esposizione in rete del servizio. Vediamo nel seguente codice un esempio di file .wsdd.

**Esempio di Web Services Deployment Descriptor (WSDD)**

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

All'interno dell'elemento deployment, che riporta i namespace a cui si fa riferimento, si trova l'elemento service che specifica le informazioni relative al servizio di cui vogliamo fare il deployment. Nel tag service possiamo vedere due attributi con i quali vengono specificati il nome del servizio e lo stile dell'applicazione, in questo esempio RPC. All'interno dell'elemento service vi sono gli elementi parameter: il primo specifica il nome della classe che implementa il servizio ed il package in cui si trova mentre il secondo riporta i metodi che possono essere esposti ("allowedMethods": metodi concessi) ad

un utilizzo diretto da altri servizi (in questo esempio l'asterisco "*" significa che tutti i metodi della classe sono utilizzabili dall'esterno).
Questo è un semplice esempio di file .wsdd al quale possono essere aggiunti molti altri elementi ed informazioni come ad esempio il tag operation per specificare le caratteristiche di un metodo ed i suoi parametri o il tag typeMapping per il mapping dei tipi, come vediamo nel frammento seguente.

**Esempio di Web Services Deployment Descriptor (WSDD)**

```
<deployment ... ... xmlns:ns="nomeNameSpace">
  <service name="NomeServizio" provider="java:RPC">

    ...

    <operation name="nomeMetodo" qname="qName" returnQName="returnQName"
      returnType="ns:DataHandler">
      <parameter name="nomeParametro" type="tipoParametro" mode="IN|OUT" />
      <parameter ... />
      ...
    </operation>
    ...

    <typeMapping
      languageSpecificType="java:javax.activation.DataHandler"
      qname="ns:DataHandler"
      serializer="org.apache.axis.encoding.ser.JAFDataHandlerSerializerFactory"
      deserializer="org.apache.axis.encoding.ser.JAFDataHandlerDeserializerFactory"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      />

    ...

  </service>
</deployment>
```

Una volta creato il file .wsdd, che potremmo ad esempio chiamare deploy.wsdd, dobbiamo per prima cosa copiare il package contenente le classi che realizzano il web service nella directory "$CATALINA_HOME/webapps/axis/WEB-INF/classes" e, successivamente, da tale posizione eseguire l'operazione di deployment utilizzando un apposito tool di Axis da linea di comando:

**Deployment**

```
java org.apache.axis.client.AdminClient ‘‘PackagePath’’/deploy.wsdd
```

A questo punto il servizio è disponibile.
Nel caso in cui si voglia rendere tale servizio non disponibile dobbiamo utilizzare lo stesso tool con un file di undeployment, ad esempio undeploy.wsdd, come quello che segue:

**undeploy.wsdd**

```
<undeployment xmlns="http://xml.apache.org/axis/wsdd/"
        xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="NomeServizio" provider="java:RPC">
    <parameter name="className" value="it.cnr.isti. . .NomeClasse"/>
    <parameter name="allowedMethods" value="*"/>

    <typeMapping
      languageSpecificType="java:javax.activation.DataHandler"
      qname="ns:DataHandler"
      serializer="org.apache.axis.encoding.ser.JAFDataHandlerSerializerFactory"
      deserializer="org.apache.axis.encoding.ser.JAFDataHandlerDeserializerFactory"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      />

  </service>
</undeployment>
```

**Undeployment**

```
java org.apache.axis.client.AdminClient ‘‘PackagePath’’/undeploy.wsdd
```

La lista di tutti i servizi disponibili può essere ottenuta dalla prima pagina di axis “http://localhost:8080/axis/”, cliccando sul link *View*. Un esempio è riportato in figura 4.3.

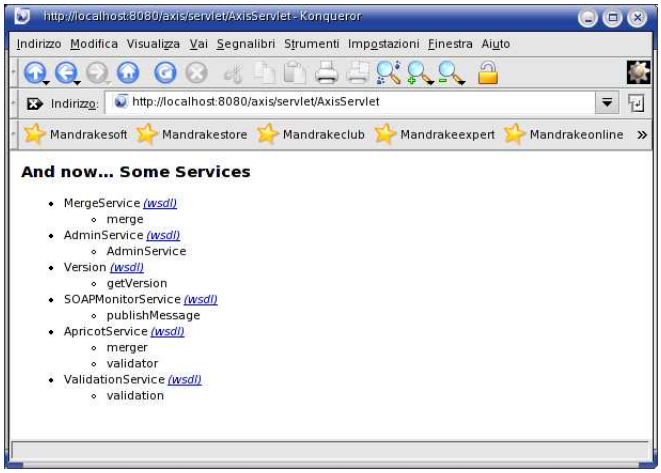


Figura 4.3: Axis - Servizi Disponibili.

### 4.3.3 Strumenti per il monitoraggio delle comunicazioni

Axis mette a disposizione dello sviluppatore due utili strumenti per il monitoraggio di connessioni e comunicazioni, quali **TCPMonitor** e **SOAPMonitor**.

**TCPMonitor** è utilizzato per monitorare il flusso di dati su una connessione TCP. Viene posizionato fra un client ed un server, dove acquisisce i dati, inviati in una connessione stabilita con esso dal client, li visualizza

nella sua interfaccia grafica ed infine li inoltra al server. Allo stesso modo, i dati inviati come risposta dal server verranno visualizzati nell'interfaccia di TCPMonitor prima di essere inoltrati al client.
Per utilizzare TCPMonitor è necessario digitare da linea di comando la seguente istruzione:

**TCPMonitor**

```
java org.apache.axis.utils.tcpmon
```

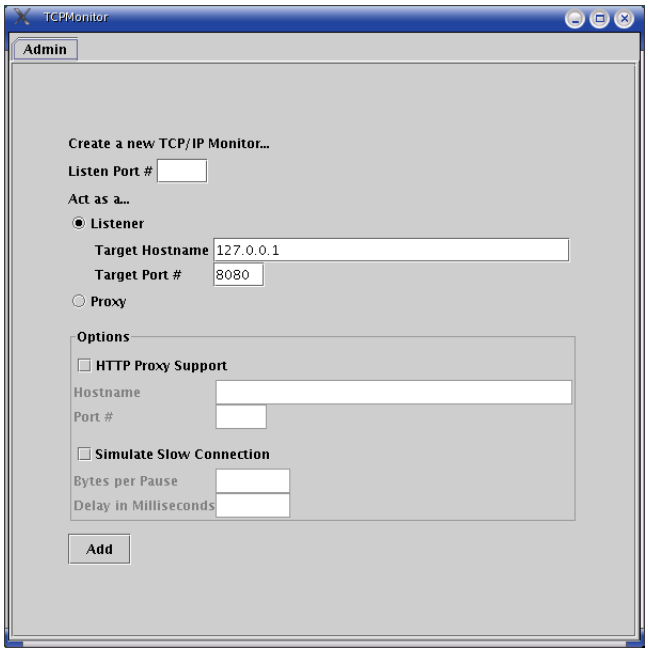


Figura 4.4: TCPMonitor - Admin.

Viene caricata l'interfaccia grafica di TCPMonitor, visualizzando la pagina principale (figura 4.4), dalla quale può essere creato un nuovo TCP/IP Monitor, specificando alcuni dati, come ad esempio la porta sulla quale attendere connessioni in entrata (Listen Port) e l'Hostname (Target Hostname) e la porta (Target Port) sui quali inoltrare tali connessioni.
La crazione di un nuovo TCP/IP Monitor farà apparire una nuova finestra (figura 4.5) nell'interfaccia grafica.

In essa, ogni volta che verrà effettuata una connessione alla "Listen Port", saranno visualizzati i messaggi di richiesta verso il server e quelli di risposta da esso, nei riquadri ad essi relativi (in figura 4.5 sono rispettivamente il box superiore ed il box inferiore).
Nella parte alta del pannello sono riportate tutte le connessioni effettuate, che possono essere selezionate per visualizzare i relativi messaggi di richiesta e risposta.

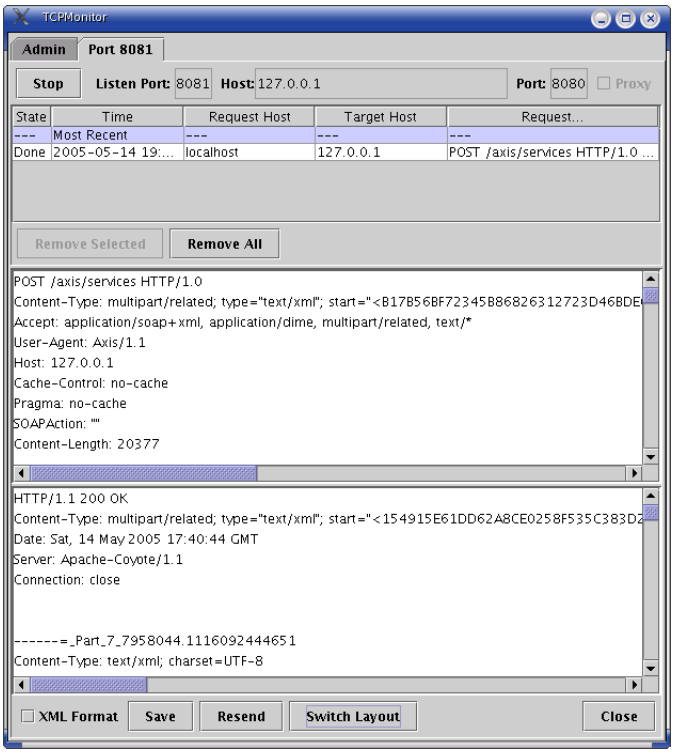


Figura 4.5: TCPMonitor.

**SOAPMonitor** è uno strumento molto utile per gli sviluppatori di Web Service poiché consente di vedere i messaggi SOAP utilizzati per invocare i web service.
Per poter utilizzare questa utility, al fine di monitorare i messaggi SOAP ricevuti e restituiti da un web service, bisogna prima seguire alcuni passi di preparazione.
Per prima cosa è necessario compilare l'applet che implementa SOAP Monitor eseguendo da linea di comando, dalla directory "webapps/axis" di Tomcat, la seguente istruzione:

**SOAPMonitor - Compilazione dell'applet**

```
javac -classpath WEB-INF/lib/axis.jar SOAPMonitorApplet.java
```

*OSS.*: se axis.jar è nel classpath non è necessario specificare l'opzione -classpath ma è sufficiente eseguire "javac SOAPMonitorApplet.java".

Dopo aver compilato l'applet, dobbiamo effettuare il deployment, esattamente come visto in 4.3.2 (II metodo di deployment), creando un file di deployment per SOAPMonitor come il seguente,

**deploy-SOAPMonitor.wsdd**

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

  <handler name="soapmonitor"
      type="java:org.apache.axis.handlers.SOAPMonitorHandler">
    <parameter name="wsdlURL"
      value="/axis/SOAPMonitorService-impl.wsdl"/>
    <parameter name="namespace"
      value="http://tempuri.org/wsdl/2001/12/SOAPMonitorService-impl.wsdl"/>
    <parameter name="serviceName" value="SOAPMonitorService"/>
    <parameter name="portName" value="Demo"/>
  </handler>

  <service name="SOAPMonitorService" provider="java:RPC">
    <parameter name="allowedMethods" value="publishMessage"/>
    <parameter name="className"
      value="org.apache.axis.monitor.SOAPMonitorService"/>
    <parameter name="scope" value="Application"/>
  </service>
</deployment>
```

chiamato ad esempio deploy-SOAPMonitor.wsdd, ed eseguendo il comando

**SOAPMonitor - Deployment**

```
java org.apache.axis.client.AdminClient deploy-monitor.wsdd
```

Dopo aver effettuato tali operazioni, il servizio SOAPMonitor è abilitato ed infatti viene inserito tra i servizi disponibili alla pagina "http://localhost-:8080/axis/servlet/AxisServlet", raggiungibile dalla prima pagina di Axis tramite il link "View".
A questo punto dobbiamo specificare i web service dei quali vogliamo monitorare i messaggi in arrivo ed in partenza. Per fare questo dobbiamo effettuare una modifica al file di deployment di tali web service, inserendo, immediatamente dopo al tag di apertura dell'elemento <service>, le definizioni dei due elementi **requestFlow** e **responseFlow**, come riportato nel seguente codice:

**Modifiche al file di deployment del Web Service da monitorare**

```
...
<service name="MergeService" provider="java:RPC">
  <requestFlow>
    <handler type="soapmonitor"/>
  </requestFlow>
  <responseFlow>
    <handler type="soapmonitor"/>
  </responseFlow>
  ...
```

Con questi due elementi viene specificato soapmonitor come gestore del flusso dei messaggi di richiesta e del flusso dei messaggi di risposta. Affinché queste modifiche abbiano effetto dobbiamo effettuare il deployment del servizio, come visto in 4.3.2 (II metodo), ma non prima di averne eseguito

l'undeployment nel caso in cui tale servizio fosse già attivo su Axis.
A questo punto possiamo lanciare SOAPMonitor, caricando da browser l'URL "http://localhost:8080/axis/SOAPMonitor", ed ogni messaggio ricevuto ed inviato dai web service che vengono monitorati apparirà nei relativi box dell'interfaccia di SOAPMonitor, come si può vedere in figura 4.6.

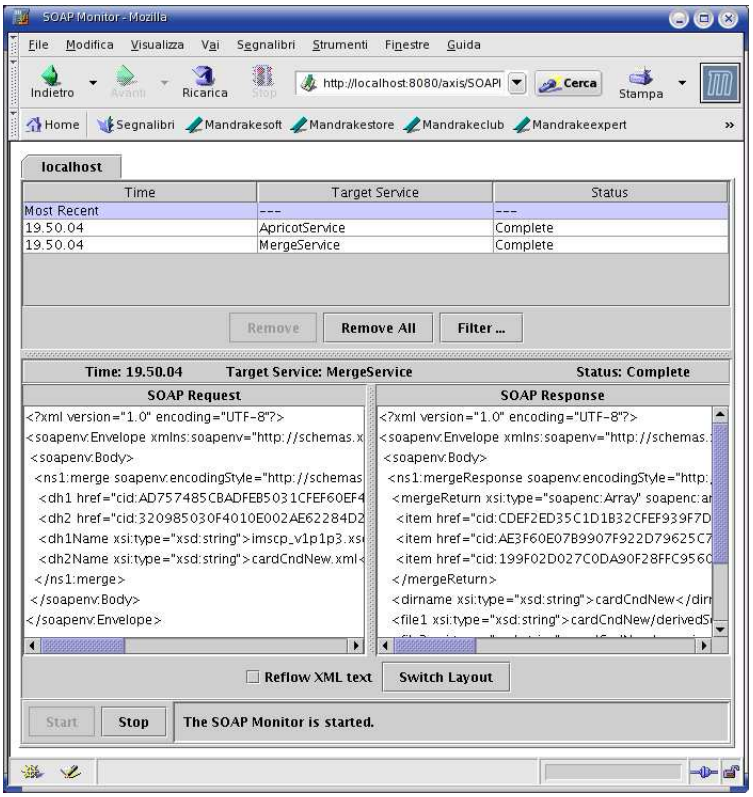


Figura 4.6: SOAPMonitor.

## 4.4 MySQL

Il progetto ha richiesto l'utilizzo di un database che permettesse di immagazzinare informazioni relative al registro UDDI e la scelta è ricaduta su MySQL. MySQL è un **Database Management System (DBMS)** relazionale, cioè un sistema per la gestione di database relazionali, potente e facile da usare. Le alternative a MySQL sono molte; vi sono ad esempio DB2, Oracle, PostgreSql. Fra tutti è stato però scelto MySQL per il fatto di essere *Open Source* e, soprattutto, gratuito.

Per la gestione di MySQL e delle informazioni memorizzate nei database sono stati utilizzati due tool grafici di semplice utilizzo, *MySQL Administator* e *MySQL Query Browser*, dei quali si può vedere l'interfaccia nelle figure

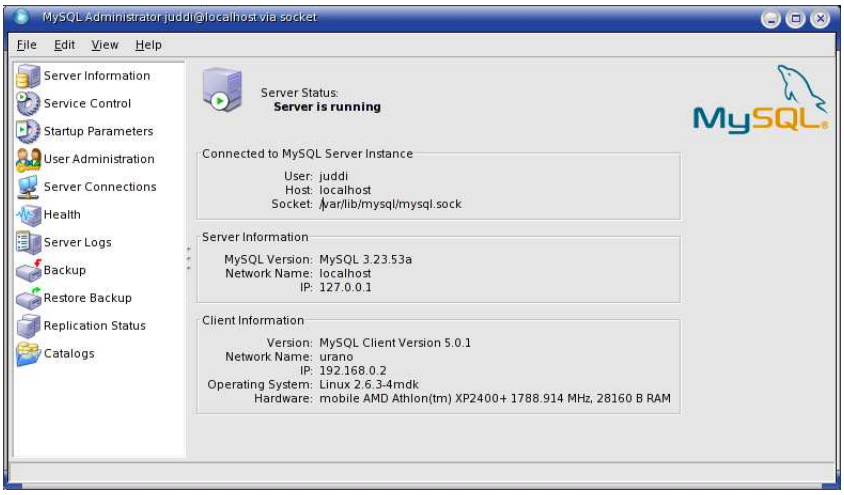


Figura 4.7: MySQL Administrator.

4.7 e 4.8. MySQL Administrator permette di monitorare il funzionamento di MySQL, mostrandone le connessioni ed il traffico da esso sostenuti, e di intervenire sui parametri di configurazione; inoltre consente di specificare alcune proprietà relative ai campi delle varie tabelle contenute nei database. MySQL Query Browser permette invece di eseguire query sui database e sulle loro tabelle; grazie a tali query si può ad esempio inserire, modificare o cancellare dati.

### 4.4.1 Installazione di MySQL

Dopo aver scaricato da internet il pacchetto contenente MySQL, che nel nostro caso si tratta di mysql-3.23.53a-pc-linux-gnu-i686.tar.gz, scompattiamolo con il comando

```
tar -xzf mysql-3.23.53a-pc-linux-gnu-i686.tar.gz
```

Impostiamo i diritti con i seguenti comandi

```
chmod u+x mysql-3.23.53a-pc-linux-gnu-i686/bin/*
chmod u+x mysql-3.23.53a-pc-linux-gnu-i686/support-files/*
chmod u+x mysql-3.23.53a-pc-linux-gnu-i686/tests/*
chmod u+x mysql-3.23.53a-pc-linux-gnu-i686/scripts/mysql_install_db
```

Installiamo adesso il database mysql spostandoci nella directory mysql-3.23.53a-pc-linux-gnu-i686 e lanciando lo script mysql_install_db:

```
cd mysql-3.23.53a-pc-linux-gnu-i686/
./scripts/mysql_install_db
```

Avviamo adesso mysqld

```
./bin/safe_mysqld &
```

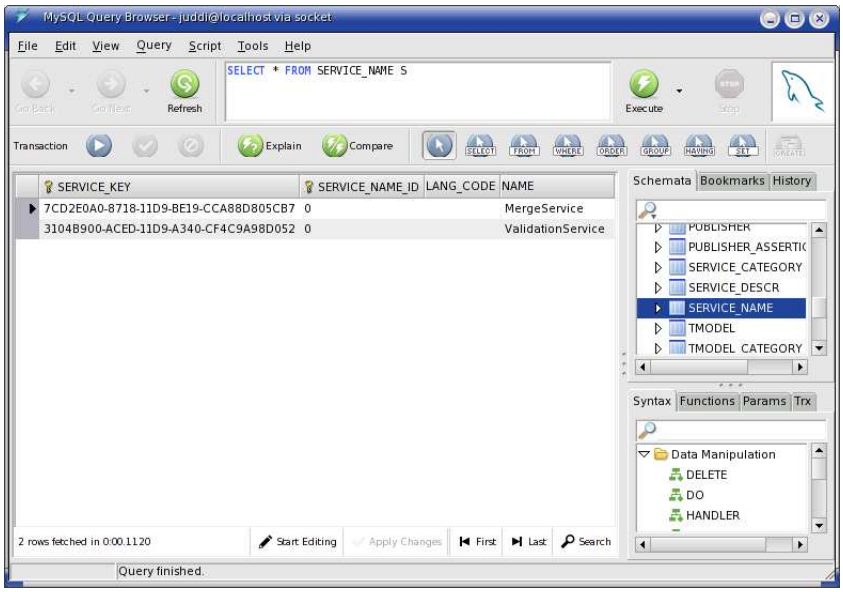


Figura 4.8: MySQL Query Browser.

ed impostiamo le password di accesso

```
./bin/mysqladmin -u root password '123456'
./bin/mysqladmin -u root -h 'host' password '123456'
```

MySQL è a questo punto pronto per essere utilizzato.

## 4.5 jUDDI

jUDDI è l'implementazione Java delle specifiche di UDDI, di cui abbiamo parlato in 3.1.4. Progetto Open Source sviluppato anch'esso da Apache, jUDDI mette a disposizione il servizio UDDI attraverso l'uso di un database, che nel nostro caso è fornito da MySQL, e delle API già descritte in 3.1.4, implementate in Java.
L'utilizzo di MySQL, come già accennato nel paragrafo ad esso relativo (4.4), è stato necessario esclusivamente come supporto per jUDDI.
Vediamo adesso la procedura da seguire per ottenere un servizio UDDI implementato utilizzando jUDDI e MySQL. [6]

### 4.5.1 Installazione di jUDDI

Requisito essenziale per l'installazione di jUDDI è la presenza ed il corretto funzionamento di Tomcat all'interno del sistema. La procedura di installazione di jUDDI prevede come prima cosa la preparazione del database MySQL, specificando una configurazione che ci permetta, attraverso l'uso

di userid e password, di accedere ai dati per leggerli e modificarli.
Connettiamoci quindi a MySQL digitando da shell il seguente comando

```
mysql --host=localhost --port=3306 --user=root --password=123456
```

oppure attraverso un tool grafico, come il già citato MySQL Query Browser, opportunamente configurato. Il comando precedente effettua una connessione con il servizio MySQL, specificando l'host e la porta alle quali deve essere effettuata tale connessione e username e password per ottenere l'accesso; se l'accesso è ottenuto, viene presentato il prompt di MySQL, dal quale possono essere eseguite le query sui database.

```
mysql>
```

Dal prompt di MySQL dobbiamo inserire le seguenti istruzioni, per configurare il database e prepararlo all'utilizzo di jUDDI.

Creazione del database jUDDI.

```
DROP DATABASE IF EXISTS juddi;
CREATE DATABASE juddi;
```

Settaggio dei priviligi generali per l'utente juddi.

```
REPLACE INTO mysql.user SET
  Host = '%', # tutti gli host (incluso localhost)
  User = 'juddi',
  Password = PASSWORD('123456'),
  Select_priv = 'Y',
  Insert_priv = 'Y',
  Update_priv = 'Y',
  Delete_priv = 'Y',
  Create_priv = 'Y',
  Drop_priv = 'Y',
  Reload_priv = 'Y',
  Shutdown_priv = 'Y',
  Process_priv = 'Y',
  File_priv = 'Y',
  Grant_priv = 'Y',
  References_priv = 'Y',
  Index_priv = 'Y',
  Alter_priv = 'Y'
;

FLUSH PRIVILEGES;
```

Cancellazione degli utenti con campo "User" vuoto.

```
DELETE FROM mysql.user WHERE User='';
UPDATE mysql.user SET Password=PASSWORD('123456')
  WHERE user='root';
FLUSH PRIVILEGES; # required
```

Settaggio dei privilegi db e host per l'utente juddi.

```
INSERT INTO mysql.db SET
  Host = '%',
  Db = 'juddi%',
  User = 'juddi',
  Select_priv = 'Y', Insert_priv = 'Y',
  Update_priv = 'Y', Delete_priv = 'Y',
  Create_priv = 'Y', Drop_priv = 'Y',
  Grant_priv = 'N', References_priv = 'Y',
  Index_priv = 'Y', Alter_priv = 'Y',
;

INSERT INTO mysql.host SET
  Host = '%',
  Db = 'juddi%',
  Select_priv = 'Y', Insert_priv = 'Y',
  Update_priv = 'Y', Delete_priv = 'Y',
  Create_priv = 'N', Drop_priv = 'N',
  Grant_priv = 'N', References_priv = 'N',
  Index_priv = 'N', Alter_priv = 'N',
;
```

A questo punto dobbiamo creare le tabelle di jUDDI. Per fare questo dobbiamo prima di tutto scaricare da Internet il pacchetto relativo a jUDDI; per questo progetto è stata utilizzata la versione 0.9rc3, scaricando il pacchetto juddi-0.9rc3.tar.gz. Scompattando questo pacchetto troviamo al suo interno, nella subdirectory "juddi-0.9rc3-src/sql/mysql" il file create_database.sql. Prendiamo il codice che si trova in questo file ed eseguiamolo in MySQL semplicemente digitando al prompt di MySQL la seguente istruzione:

```
SOURCE /.../juddi-0.9rc3/sql/mysql/create_database.sql
```

I puntini /.../ stanno ad indicare che è necessario trovare il path alla posizione in cui abbiamo estratto i file del pacchetto juddi-0.9rc3.tar.gz e con questo sostituirli.

Adesso il database di jUDDI è pronto. Dobbiamo solo aggiungere un jUDDI Publisher, cioè l'entità preposta a pubblicare informazioni sul database jUDDI. L'istruzione da eseguire, che può essere personalizzata cambiando ad esempio identificativo o nome del Publisher, è la seguente. Vengono specificati l'identificativo con cui avviene l'autenticazione del Publisher, il nome del Publisher, l'abilitazione ed il possesso di privilegi amministrativi.

```
INSERT INTO PUBLISHER (PUBLISHER_ID,PUBLISHER_NAME,ENABLED,ADMIN)
VALUES ('juddi','juddi user','true','true');
```

La preparazione di MySQL per jUDDI è terminata. Possiamo quindi uscire dalla sessione di MySQL digitando il comando

```
quit
```

Passiamo adesso alla configurazione della servlet jUDDI.
All'interno di "juddi-0.9rc3-src/webapps/" si trova la directory "juddi"; dobbiamo copiarla nella directory "$CATALINA_HOME/webapps/".

Accediamo adesso al file juddi.properties contenuto nella directory "$CATALINA_HOME/webapps/juddi/WEB-INF/" e modifichiamo come segue.
Alla riga

```
# jUDDI DataSource to use
juddi.dataSource=java:comp/env/jdbc/juddiDB
```

dobbiamo sostituire "juddiDB" con "juddi"; questo specifica il nome della sorgente dei dati, cioè il database.

Dobbiamo adesso, come abbiamo fatto anche nel caso di Axis, informare Tomcat della presenza di jUDDI, specificando alcuni parametri.
Accediamo quindi al file server.xml contenuto in "$CATALINA_HOME-/conf/" ed aggiungiamo le linee riportate di seguito all'interno dell'elemento <HOST>.

```
<Context path="/juddi" docBase="juddi" debug="5" reloadable="true"
  crossContext="true">
  <Logger className="org.apache.catalina.logger.FileLogger"
              prefix="localhost_juddiDB_log" suffix=".txt"
              timestamp="true"/>
  <Resource name="jdbc/juddi"
                auth="Container"
                type="javax.sql.DataSource"/>
  <ResourceParams name="jdbc/juddi">
    <parameter>
      <name>factory</name>
      <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
    </parameter>
    <!-- Massimo numero di connessioni al dB. Assicurarsi di aver
         configurato max_connections in mysqld in modo sufficiente a
         gestire tutte le donnessioni al dB.
         Settare a 0 per nessun limite. -->
    <parameter><name>maxActive</name><value>100</value></parameter>
    <!-- Massimo numero di connessioni al dB inattive da conservare.
         Settare a 0 per nessun limite. -->
    <parameter><name>maxIdle</name><value>30</value></parameter>
    <parameter><name>maxWait</name><value>10000</value></parameter>
    <!-- MySQL dB username e password per la connessione al dB  -->
    <parameter><name>username</name><value>juddi</value></parameter>
    <parameter><name>password</name><value>123456</value></parameter>
    <!-- ClassName per mm.mysql JDBC driver -->
    <parameter>
      <name>driverClassName</name>
      <value>org.gjt.mm.mysql.Driver</value>
    </parameter>
    <!-- JDBC connection url per la connessione al dB MySQL.
         L'argomento autoReconnect=true in fondo all'URL garantisce
         che il mm.mysql JDBC Driver si riconnetter\'a automaticamente
         nel caso in cui mysqld chiuda la connessione.  mysqld per
         default chiude le connessioni inattive dopo 8 ore. -->
    <parameter>
      <name>url</name>
      <value>jdbc:mysql://host.domain.com:3306/juddi?autoReconnect=true</value>
    </parameter>
    <parameter>
          <name>validationQuery</name>
```

```
        <value>select count(*) from PUBLISHER</value>
    </parameter>
  </ResourceParams>
</Context>
```

Effettuata questa aggiunta al file server.xml, dobbiamo eseguire una modifica in due file: "happyjuddi.jsp", contenuto in "$CATALINA_HOME-/webapps/juddi/", e "web.xml", contenuto in "$CATALINA_HOME/webapps-/juddi/WEB-INF/".

E' necessario sostituire "jdbc/juddiDB" con "jdbc/juddi" in happyjuddi.jsp alla seguente linea

```
dsname = ‘‘java:comp/env/jdbc/juddi’’;
```

ed in web.xml alle successive

```
<resource-ref>
  <description>jUDDI DataSource</description>
  <res-ref-name>jdbc/juddiDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Infine abbiamo bisogno del pacchetto contenente la classe del Driver per la connessione al database, che può essere scaricato da internet. Il pacchetto utilizzato è mysql-connector-java-3.0.16-ga-bin.jar contenente la classe org.gjt.mm.mysql.Driver. La locazione in cui va inserito è "$CATALINA_HOME/common/lib/".

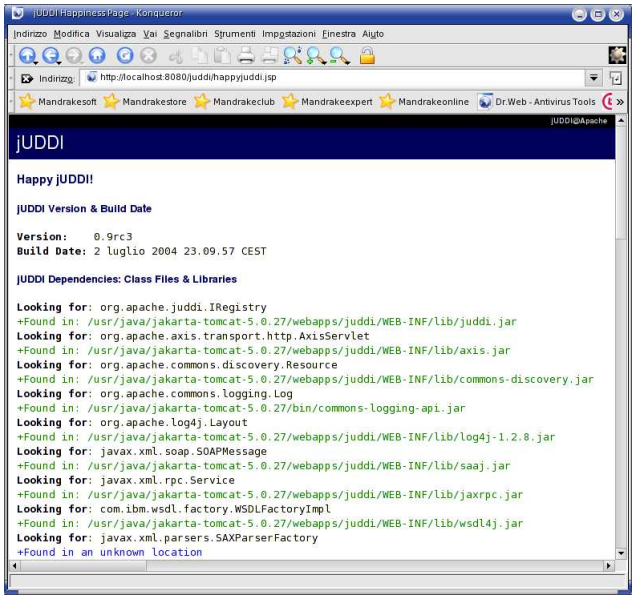


Figura 4.9: Happy jUDDI Page.

A questo punto tutto è pronto e possiamo quindi testare l'installazione. Dopo aver riavviato Tomcat inseriamo nel browser l'indirizzo "http://-localhost:8080/juddi" ed otteniamo la prima pagina di jUDDI. Per il test dell'installazione facciamo click sul link "Validate". Questo chiama la pagina happyjuddi.jsp che effettua la verifica della configurazione e presenta una pagina di report, riportata in figura 4.9.

Se la nostra installazione è corretta vengono specificati tutti i path alle locazioni in cui sono state trovate le classi e le librerie necessarie al funzionamento di jUDDI ed inoltre, più in basso nella pagina, sono riportate le informazioni relative al DataSource, cioè al database jUDDI, ed alla connessione ad esso, come si può vedere in figura 4.10.

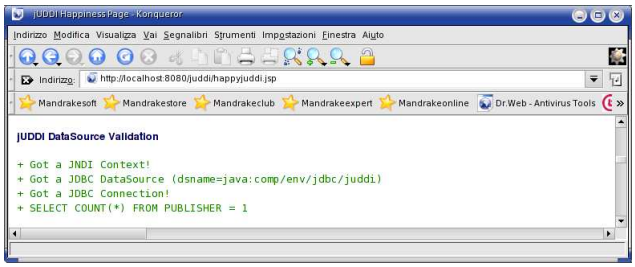


Figura 4.10: Happy jUDDI Page - DataSource.

## 4.6 UDDI4J

UDDI4J è una libreria di classi Java che fornisce un'API per interagire con un registro UDDI. Questa libreria genera i messaggi da spedire ad un server UDDI ed effettua il parsing di quelli ricevuti da un server UDDI.
La classe principale in quest'insieme di API è **org.uddi4j.client.UDDIProxy**. Attraverso di essa viene creato un proxy per l'accesso al server UDDI, mettendo a disposizione dei metodi che rispettano le specifiche delle API di UDDI (UDDI Programmer's API Specification).
Le classi all'interno di **org.uddi4j.datatype** rappresentano gli oggetti-dato utilizzati per spedire o ricevere informazioni da UDDI; datatype è suddiviso in sottopacchetti relativi ai vari argomenti trattati da UDDI, come business, service, tmodel, binding e assertion.
Vi sono poi i due sottopacchetti **org.uddi4j.request** e **org.uddi4j.response** che contengono rispettivamente i possibili messaggi di richiesta di informazioni al server UDDI e quelli che quest'ultimo può inviare come risposta.
Infine in **org.uddi4j.transport** si trovano i supporti ai protocolli di trasmissione che possono essere utilizzati per l'invio dei messaggi.

### 4.6.1 Installazione di UDDI4J

Per utilizzare UDDI4J è necessario il pacchetto uddi4j.jar, scaricabile da internet. In questo progetto è stato utilizzata la versione 2.0.2, ottenuta scaricando il file uddi4j-bin-2_0_2.zip.
Dopo aver acquisito il pacchetto, esso è stato copiato nella directory $CATALINA_HOME/webapps/axis/WEB-INF/lib/ e tale percorso a uddi4j-.jar è stato inserito al CLASSPATH nel file /etc/profile.

```
CLASSPATH=$CLASSPATH: ...
                  :$CATALINA_HOME/webapps/axis/WEB-INF/lib/uddi4j.jar
                  : ...
```

UDDI4J non necessita di alcuna installazione e, dopo aver riavviato il sistema in modo che venga aggiornato il CLASSPATH, UDDI4J può essere utilizzato.

## 4.7 Tecniche di invocazione di web service

La tecnologia utilizzata per l'invocazione dei web service realizzati in questo progetto è **JAX-RPC (Java API for XML-Based Remote Procedure Call)**. JAX-RPC permette di invocare web service, impostando in modo semplice e veloce i parametri di tale invocazione [7]. Questa tecnologia fornisce inoltre il supporto per il mapping dei tipi da XML a Java e viceversa. JAX-RPC, che necessita di SOAP (sopra HTTP), mette a disposizione il supporto per il modello di scambio di messaggi SOAP. Inoltre SAAJ (SOAP Attachment API for Java), utilizzato da JAX-RPC, fornisce una API Java per la costruzione e la manipolazione di messaggi SOAP con allegati. E' possibile trasferire, all'interno di allegati al messaggio SOAP, vari tipi di documento, come ad esempio documenti XML o immagini.

JAX-RPC permette l'utilizzo di varie tecniche per l'invocazione dei web service, che può essere effettuata con metodi differenti, a seconda delle esigenze e del livello di dinamicità che vogliamo ottenere.
Come già accennato in 4.3, le possibili metodologie di invocazione di un web service sono:

- Stub creato da WSDL,
- Dynamic Proxy,
- Dynamic Invocation Interface (DII) e
- Dynamic Discovery and Invocation (DDI).

### 4.7.1 Stub creato da WSDL

Il metodo **Stub**, dove lo stub rappresenta il lato client di un servizio, utilizza il file WSDL per ottenere informazioni riguardanti quel servizio. Prima della fase di esecuzione viene creato uno stub, specifico per la piattaforma che stiamo utilizzando, durante la fase di mapping delle informazioni da WSDL a Java. Dato che tale stub è creato prima della fase di esecuzione esso viene a volte vhiamato *static stub*.
Lo stub è costituito da una classe Java che implementa una SEI (Service Endpoint Interface). La Service Endpoint Interface è la rappresentazione Java delle operazioni del web service che sono descritte dall'elemento PortType all'interno del documento WSDL, cioè un'interfaccia Java che definisce i metodi utilizzati dal client per interagire con il web service. Come abbiamo già detto la SEI è generata prima della fase di esecuzione; la sua creazione avviene attraverso l'uso di appositi tool per il mapping da WSDL a Java, come ad esempio WSDL2Java di Apache Axis.
Lo stub è quindi una classe che agisce da proxy per un servizio remoto.

Il vantaggio del metodo stub è la semplicità; servono infatti poche linee di codice, come si può vedere dal seguente frammento, per invocare un web service.

**Stub**

```
// StubName: nome di esempio della classe che costituisce lo stub.
StubName stub = (StubName) service.getStubName();

// Invocazione del metodo di esempio StubMethod del web service
stub.getStubMethod("...");
```

Necessaria è la conoscenza in fase di sviluppo dell'URL del file WSDL, che deve essere passato al tool di mapping WSDL-to-Java per la generazione dello stub ed inoltre lo stub non è portabile perché dipendente, abbiamo già detto, dalla specifica piattaforma su cui è stato creato.

### 4.7.2 Dynamic Proxy

Il metodo Dynamic Proxy, a differenza del metodo Stub, non crea una classe specifica per un dato web service prima della fase di esecuzione. Questo metodo utilizza un proxy per invocare la specifica operazione del web service. Il proxy è una classe Java che, come nel caso dello Stub, implementa la SEI; il proxy è però creato a tempo di esecuzione ed ottenuto dal metodo getPort(), di JAX-RPC Service, il quale prende il nome della porta relativa al servizio che vogliamo invocare ed il nome della classe che rappresenta la SEI che viene implementata dal proxy. Possiamo vedere nel seguente frammento il codice necessario ad effettuare l'invocazione di un'operazione di un web service facendo uso del metodo Dynamic Proxy.

**Dynamic Proxy**

```
// WSURL.com: indirizzo d'esempio al quale si trova il web service
// che vogliamo invocare.
String namespace = "http://WSURL.com";

// WSName: nome d'esempio del web service che vogliamo invocare.
String portName = "WSName";

// portQN: porta alla quale effettuare l'invocazione.
QName portQN = new QName(namespace, portName);

// WSName: nome della classe generata, che rappresenta il proxy.
WSName proxy = service.getPort(portQN, ProxyName.class);

// Invocazione del metodo di esempio ProxyMethod del web service
proxy.getProxyMethod("...");
```

Questo metodo è definito *dynamic*, cioè dinamico, perché il proxy è creato a tempo di esecuzione. E' comunque necessario, come nel caso di Static Stub, conoscere a development-time[1] l'interfaccia dell'operazione che viene invocata.
Il vantaggio che dà l'utilizzo di questo metodo è la creazione di codice portabile ed indipendente dalla piattaforma.

### 4.7.3 Dynamic Invocation Interface (DII)

Il metodo Dynamic Invocation Interface complica un po' il codice ma fornisce un livello di dinamicità più elevato. Non è più necessario conoscere l'URL del file WSDL a development-time. Inoltre JAX-RPC Service non viene più utilizzato per ottenere un proxy, bensì per l'istanziazione di JAX-RPC Calls, utilizzate per l'invocazione del web service.
Vediamo nel seguente frammento il codice che realizza la Dynamic Invocation Interface.

**Dynamic Invocation Interface**

```
// WSURL.com: indirizzo d'esempio al quale si trova il web service
// che vogliamo invocare.
String namespace = "http://WSURL.com";

// WSName: nome d'esempio del web service che vogliamo invocare.
String portName = "WSName";

// portQN: porta alla quale effettuare l'invocazione.
QName portQN = new QName(namespace, portName);

// getMethod: nome dell'operazione da invocare.
String operationName = "getMethod";

// Creo la chiamata call.
Call call = service.createCall();

// Imposto le caratteristiche della chiamata.

// Setto la porta alla quale viene fatta l'invocazione.
```

[1] Development-time: fase di sviluppo dell'applicazione.

```
call.setPortTypeName(portQN);

// Setto il nome dell'operazione da invocare.
call.setOperationName(new QName(namespace, operationName));

// Setto la codifica.
call.setProperty(Call.ENCODINGSTYLE_URI_PROPERTY, "");

// Setto lo stile dell'operazione.
call.setProperty(Call.OPERATION_STYLE_PROPERTY, "rpc");

// Aggiungo il parametro "param1", di cui specifico il tipo ed il
// fatto che \'e un parametro d'ingresso.
call.addParameter("param1", <xsd:string>, ParameterMode.IN);

// Setto il tipo di ritorno.
call.setReturnType(<xsd:string>);

Creo l'array dei parametri da passare nell'invocazione.
Object[] inParams = new Object[] {"..."};

// Effettuo l'invocazione passando l'array dei parametri inParams
// ed acquisisco il risultato.
String ret = (String) call.invoke(inParams);
```

Il vantaggio di utilizzare DII sta nel fatto che per invocare una procedura remota non è necessario conoscere il file WSDL e non è richiesta la generazione di classi, come avviene nei casi di Dynamic Proxy e Static Stub. Devono però essere conosciuti l'indirizzo del servizio, le operazioni che mette a disposizione ed i parametri da queste accettati. Queste informazioni vengono utilizzate dal client DII dinamicamente a runtime.

### 4.7.4 Dynamic Discovery and Invocation (DDI)

Dynamic Discovery and Invocation rappresenta la soluzione più dinamica nell'invocazione di web service. Questo metodo è un'evoluzione del DII appena descritto, con la differenza per DDI non è necessario conoscere in anticipo l'indirizzo del servizio, le sue operazioni ed i parametri d'ingresso di queste ultime. Tutte queste informazioni verranno reperite a tempo di esecuzione grazie al servizio UDDI, di cui abbiamo parlato in 3.1.4, che permette di cercare web service ed ottenere così tutto ciò che è necessario sapere per invocarli.
Senza riportare il codice che realizza la Dynamic Discovery and Invocation, poiché i modi possibili per farlo sono numerosi, riportiamo i passi che questo metodo segue:

1. Richiedere al servizio UDDI informazioni relative al web service che vogliamo invocare, fra le quali otteniamo l'URL del file WSDL che lo descrive. Questo può essere fatto utilizzando le librerie UDDI4J.

2. Eseguire il parsing del documento WSDL, ad esempio attraverso l'utility WSDL4J di Axis, ed ottenere le informazioni necessarie all'in-

vocazione del web service come namespace, nome dell'operazione e parametri.

3. Invocare il servizio utilizzando il metodo DII con i valori ottenuti al passo precedente.

# Bibliografia


[1] Apache Axis: Sito ufficiale. On line at: http://ws.apache.org/axis/.

[2] Apache jUDDI: Sito ufficiale. On line at: http://ws.apache.org/juddi.

[3] Apache Axis - Web Services Details. On line at: http://www.dsg.cs.tcd.ie/ dowlingj/teaching/ds/lectures/-ws_details.pdf.

[4] Apache Tomcat: Sito ufficiale. On line at: http://jakarta.apache.org-/tomcat/.

[5] Axis WSDD Reference. On line at: http://www.osmoticweb.com/axis-wsdd/.

[6] Deploy jUDDI on Tomcat and MySQL. On line at: http://wiki-.apache.org/ws/Deploy_jUDDI_Tomcat_and_MySQL.

[7] Java API for XML-Based RPC (JAX-RPC) Overview. On line at: http://java.sun.com/xml/jaxrpc/overview.html.

[8] MokaByte : Articles. On line at: http://www.mokabyte.it/.

[9] MokaByte : Service-Oriented Architecture. On line at: http://www-.mokabyte.it/2004/09/soa.htm.

[10] MySQL: Sito ufficiale. On line at: http://www.mysql.com.

[11] Service-Oriented Architecture: Documentazione. On line at: http://-webservices.xml.com/pub/a/ws/2003/09/30/soa.html.

[12] SOAP: Documentazione. On line at: http://www.w3.org/TR/soap.

[13] SUN: Java. On line at: http://java.sun.com.

[14] UDDI Version 2.04 API Specification. On line at: http://uddi.org-/pubs/ProgrammersAPI-V2.04-Published-20020719.htm.

[15] W3C : Web Services Architecture. On line at: http://www.w3.org-/TR/ws-arch/.

[16] W3C : Web Services Architecture Requirements. On line at: http://-www.w3.org/TR/wsa-reqs/.

[17] UDDI: Documentazione. On line at: http://www.uddi.org.

[18] UDDI4J: Sito ufficiale. On line at: http://uddi4j.sourceforge.net/.

[19] W3Schools : XML Schema Tutorial. On line at: http://www-.w3schools.com/schema/.

[20] Web Services e Service-Oriented Architecture: Documentazione. On line at: http://www.service-architecture.com.

[21] Web Services Orchestration: a review of emerging technologies, tools and standards. On line at: http://devresource.hp.com/drc/technical-_white_papers/WSOrch/WSOrchestration.pdf.

[22] Web Services Orchestration and Choreography: a look at WSCI and BPEL4WS. On line at: http://wldj.sys-con.com/read/39800.htm.

[23] WSDL: Documentazione. On line at: http://www.w3.org/TR/wsdl.

[24] XML: Documentazione. On line at: http://www.xml.com.

[25] *The Java Web Services Tutorial*. Sun Microsystems, 2003.

[26] *Practical Considerations for Implementing Web Services*. AmberPoint, 2004.

[27] *Understanding Service-Oriented Architecture*. Versata, 2004.

[28] Greg Barish. Getting started with Web Services Using Apache Axis. On line at: http://www.javaranch.com/newsletter/May2002-/newslettermay2002.jsp#axis, Maggio 2002.

[29] Harumi Kuno Gustavo Alonso, Fabio Casati and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications*. Hewlett-Packard, 2004.

[30] David S. Linthicum. *Next Generation Application Integration*. Addison-Wesley, 2003.

[31] Beth Stearns Rahul Sharma and Tony Ng. *J2EE Connector Architecture and Enterprise Application Integration*. Addison-Wesley, 2001.