ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

ARCHITETTURE ORIENTATE AI SERVIZI PER LO SVILUPPO DI SISTEMI DISTRIBUITI BASATI SU WEB: ANALISI DEI RESTFUL WEB SERVICE

Elaborata nel corso di: Sistemi Distribuiti LA

Tesi di Laurea di:
ZOFFOLI ELIA

Relatore:
Prof. ALESSANDRO RICCI

Co-relatore:
Dott. Ing. ANDREA SANTI

ANNO ACCADEMICO 2010–2011
SESSIONE III

PAROLE CHIAVE

Servizi web

Service Oriented Architecture

SOAP/WSDL

ReST

Resource Oriented Architecture

Framework

Alla mia famiglia e ai miei amici per avermi supportato
durante tutto il percorso di studio.

Indice

1 Introduzione **1**

2 Servizi Web **5**
2.1 Servizi Web Principali Stili 6
2.2 Comunicazione nei servizi web 7
2.3 Struttura a livelli 8
2.4 Benefici e rischi nell'uso dei Servizi Web 15
2.5 Tecnologie di base 18
2.5.1 HyperText Transfer Protocol 19
2.5.2 eXtensible Markup Language 24

3 Service Oriented Architecture **33**
3.1 Funzionamento ed entità di SOA 34
3.2 Caratteristiche di un'architettura SOA 36
3.3 Struttura a livelli di SOA 41

4 Web Service: architettura service-oriented **43**
4.1 Web Service e SOA 43
4.2 Web Service Definition Lenguage 44
4.2.1 Struttura di un documento WSDL 45
4.2.2 WSDL 2.0 52
4.3 Simple Object Access Protocol 54
4.3.1 SOAP e HTTP 59
4.4 Specifiche WS-* 64
4.5 Universal Description Discovery and Integration 65
4.5.1 API UDDI 67

5 Servizi Web ReSTful 69
5.1 Cos'è ReST? . . . 70
5.2 Servizi Ibridi . . . 73
5.3 Resource Oriented Architecture . . . 75
5.4 Servizi ReSTful: Design . . . 80
5.5 Servizi ReSTful: Best Practices . . . 84
5.5.1 Operazioni asincrone . . . 84
5.5.2 Transazioni . . . 85
5.5.3 Sicurezza e Autenticazione . . . 87
5.6 Service Description: Servizi ReSTful . . . 88
5.6.1 Web Service Description Lenguage 2.0 HTTP binding extension . . . 89
5.6.2 Web Application Description Language . . . 94
5.6.3 WSDL 2.0 vs WADL . . . 103

6 Framework per servizi web ReSTful 105
6.1 Windows Communication Foundation . . . 106
6.1.1 WCF e rappresentazioni . . . 109
6.1.2 WCF lato server . . . 110
6.1.3 WCF lato client . . . 112
6.1.4 Esempio Client/Server WCF . . . 112
6.2 ReSTlet . . . 117
6.2.1 Esempio Client/Server ReSTlet . . . 123
6.2.2 ReSTlet e JavaScript client . . . 127
6.3 Client PHP Servizi web ReSTful . . . 128

7 Conclusioni 131

Capitolo 1

Introduzione

L'importanza della comunicazione e dello scambio informativo, sia in ambito aziendale che tra singoli utenti, ha portato oggigiorno i sistemi informatici ad evolversi da un contesto prettamente concentrato ad un panorama distribuito in cui più applicazioni presenti su diversi dispositivi comunicano fra loro scambiandosi informazioni di ogni genere attraverso la rete, generalmente internet. In uno scenario così eterogeneo vi è un'alta probabilità che queste applicazioni utilizzino differenti tecnologie e piattaforme rendendo in molti casi la comunicazione molto complicata se non impossibile. Nacque perciò la necessita di inserire un componente con lo scopo di garantire l'interoperabilità.

Furono introdotti così i middleware, ovvero strati software con il compito di nascondere le eterogeneità dei livelli sottostanti e permettere la comunicazione. Questi sistemi erano, però, molto costosi e technology dependent perciò a seguito dell'introduzione di una nuova tecnologia necessitavano di aggiornamento o addirittura di sostituzione, con conseguente aumento dei costi.

È in questo ambito che si inserisce il Service Oriented Computing. Esso definisce una "nuova entità", il servizio, con il compito di consentire la cooperazione e comunicazione tra applicazioni diverse. In generale un servizio è un componente computazionale autonomo che incapsula una determinata funzionalità ed è in grado di fornirla attraverso la rete. La comunicazione con questo tipo di entità si basa su scambio di messaggi in un formato indipendente dalla piattaforma su cui si esegue, questo è possibile grazie all'utilizzo di un insieme di standard riconosciuti da tutti i produttori, che

permettono ai servizi di scambiare informazioni con gli utenti e con altri servizi.

In questo elaborato ci si propone di analizzare in particolare due tipologie di architettura: Service Oriented e Resourse Oriented. La differenza tra i due approcci progettuali sta nella visione che si ha durante la costruzione del sistema. Un sistema Service Oriented ha una visione orientata alle attività, ovvero il servizio viene visto come un insieme di operazioni che è possibile richiedere. Mentre un sistema Resource Oriented ha una visione orientata alle risorse, un servizio viene visto come un insieme di risorse su cui è possibile eseguire un insieme di operazioni ben definite e pre–determinate.

La trattazione si divide in sei capitoli.

Innanzitutto vengono presentati i servizi web in generale ponendo le basi per gli argomenti trattati nei successivi capitoli. A tale scopo in questo capitolo vengono introdotti le modalità di comunicazione e l'architettura dei servizi web per meglio chiarire al lettore come operano queste entità. Infine vengono mostrati i possibili benefici e rischi portati dall'utilizzo dei servizi e descritte le tecnologie standard alla base dell'architettura, questa parte è ritenuta fondamentale per riuscire a comprendere le modalità di comunicazione delle diverse tipologie di servizio.

Nel terzo capitolo viene analizzata l'architettura Service Oriented, essa è stata la prima architettura ideata appositamente per la costruzione di servizi. SOA (Service Oriented Architecture) definisce il funzionamento e le caratteristiche che deve possedere un sistema che segue questa architettura. Questo passo risulta fondamentale per definire il primo tipo di servizi web: SOAP services.

Questi tipo di servizi utilizzano una serie di tecnologie basate su XML per permettere la comunicazione tra cliente e fornitore (SOAP), per descrivere come interagire con il servizio (WSDL) e per rendere pubblica questa descrizione e quindi permetterne l'accesso ai clienti (UDDI). Tutte queste verranno approfondite all'interno del quarto capitolo mostrando alcuni esempi che ne chiariranno l'utilizzo.

Il capitolo 5 è incentrato su una nuova ed emergente modalità per realizzare servizi web. Innanzitutto è ritenuto doveroso introdurre i principi ReST che formano le basi per questo tipo di servizi, dopodiché è possibile

analizzare l'architettura Resource-Oriented che descrive i vincoli e i principi da rispettare durante la realizzazione di questi servizi. In seguito vengono definiti i passi da seguire durante la progettazione e un inseme di best practice per gestire i maggiori problemi che si possono incontrare, per infine definire i metodi di descrizione dei servizi ReSTful basati sui linguaggi WSDL e WADL.

Nel capitolo 6 sono studiati i principali framework per la realizzazione di servize e client ReSTful. In particolare vengono analizzate due delle modalità utilizzate da PHP e JavaScript per utilizzare questo tipo di servizi, queste sono le modalità solitamente utilizzate per la creazione di siti web dinamici tipici del cosiddetto WEB 2.0.

In ultima istanza nelle conclusioni vengono tirate le somme circa il lavoro fatto nell'elaborato ed effettuato un confronto tra le due principali modalità di realizzazione dei servizi web: SOAP e ReSTful.

Capitolo 2

Servizi Web

I servizi web sono applicazioni cooperanti, indipendenti dalla piattaforma su cui si trovano e che comunicano con scambio di messaggi attraverso il web.

Il World Wide Web Consortium (W3C) definisce i servizi web come [33]: "un sistema software progettato per favorire l'interoperabilità tra applicazioni attraverso la rete. Essi possiedono un interfaccia autodescrittiva comprensibile da tutte le applicazioni.

Altri sistemi interagiscono con un servizio con le modalità descritte nella sua interfaccia, utilizzando messaggi di solito in formato XML e trasmessi attraverso un protocollo di rete che ne permette la trasmissione in rete (solitamente HTTP)".

L'obbiettivo principale dei servizi web non è solamente quello di fornire servizi attraverso la rete, ma anche quello di fornire meccanismi per condividere le sue funzionalità con altri servizi ed applicazioni. Questo porta ad una nuova filosofia che consiste nello spostare lo sviluppo di software distribuito da programmazione a composizione, dalla codifica a partire da zero all'utilizzo di componenti già esistenti.

Per mantenere la "promessa" di interoperabilità un servizio è costretto ad utilizzare linguaggi e protocolli standard in modo da permetterne l'esecuzione su ogni piattaforma e attraverso qualunque livello di trasporto e da rendere comprensibile la comunicazione ad ogni utente che desidera interagirvi, indipendentemente dal linguaggio scelto per implementarlo. La scelta

di HTTP e di XML come tecnologie sulle quali basare la comunicazione tra cliente e fornitore rispecchia perfettamente questo intento.

Ogni tipologia di servizi web utilizza queste due tecnologie seguendo la propria architettura e suddividendo tra esse le informazioni sullo scopo della richiesta (method information) e sui dati necessari al soddisfacimento della stessa (scooping information).

2.1 Servizi Web Principali Stili

I Servizi Web possono essere costruiti utilizzando diverse tecnologie e stili. In particolare sono tre i principali stili fin'ora realizzati: RPC, SOAP e ReSTful.

- **Remote Procedure Call:** Lo stile RPC per i servizi web fu il primo ad essere adottato, questo approccio riproduce una chiamata a procedura in cui richiedente e fornitore si trovano su due macchine differenti.

  Una chiamata ad un servizio RPC è composta da una serie di messaggi XML (SOAP o XML-RPC) contenenti l'informazione sul metodo che il client vuole utilizzare e i parametri di cui ha bisogno il fornitore per eseguire la procedura, questi ultimi devono essere inseriti all'interno della richiesta nello stesso ordine con cui sono indicati nell'interfaccia della funzione chiamata.

  Questo tipo di servizio crea un accoppiamento più stretto tra i due attori, in quanto il cliente deve conoscere nei particolari la visione che il fornitore ha dell'operazione che si desidera eseguire (deve essere noto il nome della procedura e il nome, il tipo e l'ordine dei parametri così come sono definiti dal fornitore). In caso di modifica da parte del provider anche l'utilizzatore deve aggiornarsi per poter continuare ad operare correttamente.

- **Service Oriented Architecture:** I servizi web si adattano perfettamente per realizzare un sistema che rispetta il modello service oriented.

  I servizi web che adottano questo stile comunicano con il richiedente utilizzando messaggi SOAP nelle modalità descritte nell'interfac-

cia WSDL, quest'ultima viene pubblicata dal provider su un registro UDDI per permetterne la ricerca.

Per utilizzare un servizio SOA un client deve inviare un messaggio SOAP indicando nel body i dati necessari al fornitore per soddisfare la richiesta. Ricevute queste informazioni sarà il fornitore a capire quale operazione il client desidera eseguire è potrà così inviare la risposta adeguata. Il mapping tra operazione e dati richiesti viene effettuato dal documento WSDL che indica al client l'input adeguato per ogni possibile richiesta.

- **ReSTful:** I Servizi web di questo tipo vengono realizzati e progettati seguendo i principi definiti da ReST e cercando di assottigliare la distanza con i comuni siti web. Proprio a conseguenza di questo l'architettura dei servizi ReSTful si basa sull'idea che "qualsiasi concetto rilevante può essere risorsa la quale è individuata da un indirizzo univoco (URI) ed è manipolata attraverso i metodi HTTP".

  In questo tipo di servizi l'azione che l'utente vuole eseguire (ad esempio creazione, modifica o lettura di informazioni) e i dati necessari all'esecuzione della richiesta vengono mappati rispettivamente da i metodi HTTP (ognuno dei quali corrisponde ad un azione possibile dell'utente) e dall'URI della risorsa su cui operare.

2.2 Comunicazione nei servizi web

I servizi web hanno due principali entità che interagiscono fra loro:

- **Service Consumer:** rappresenta il cliente richiedente la funzione offerta dal servizio. Questa entità può essere concretizzata da un applicazione, un altro servizio o da un qualsiasi modulo software che ne necessita.

  È il Service Consumer ad avviare la comunicazione con il provider ed inviare una richiesta al fornitore per l'esecuzione di un operazione.

- **Service Provider:** è un entità indirizzabile in grado di accettare ed eseguire richieste da parte di clients. Esso fornisce una ben definita descrizione del servizio ed implementa le sue funzionalità.

La comunicazione tra questi avviene per mezzo di un messaggio di richiesta, dal consumer al provider, e un messaggio di risposta, dal provider al consumer. Questi devono essere formattati in modo che cliente e fornitore siano in grado di comprenderne il significato. Per garantire questo il service provider può mettere a disposizione un documento chiamato **Service Description**.

Il **Service Description** contiene la descrizione del servizio e le specifiche della formattazione delle richieste e risposte. Oltre a questo può specificare anche un insieme di informazioni utili, ad esempio può indicare un insieme di presupposti e post-condizioni per l'esecuzione della funzionalità, oppure la qualità del servizio (QoS) garantita.

Non per tutti i servizi web è obbligatoria la presenza di quest'ultima entità, in alcune situazioni il service consumer è già a conoscenza dei dettagli del servizio (ad esempio perché vengono utilizzati dei metodi di comunicazione standard).

I due ruoli di consumer e provider non vengono fissati al momento della generazione dei processi. Un processo può essere consumer nel momento in cui necessita di un servizio messo a disposizione da altri e provider quando riceve da un altro processo la richiesta di un certo servizio di cui è fornitore.

2.3 Struttura a livelli

È possibile rappresentare la struttura di un servizio web attraverso un insieme di livelli che aiuta nell'utilizzo e nella comprensione del servizio stesso [32]. Ogni livello è costruito su quello inferiore aumentandone il grado di astrazione.

Questa pila è divisibile in tre parti ognuna delle quali svolge un ruolo fondamentale nell'architettura dei servizi web.

Ogni layer è poi concretizzato da una specifica tecnologia basata sugli standard adottati.

La struttura aiuta a capire il ruolo che ogni standard assume all'interno della costruzione del servizio senza entrare nello specifico.

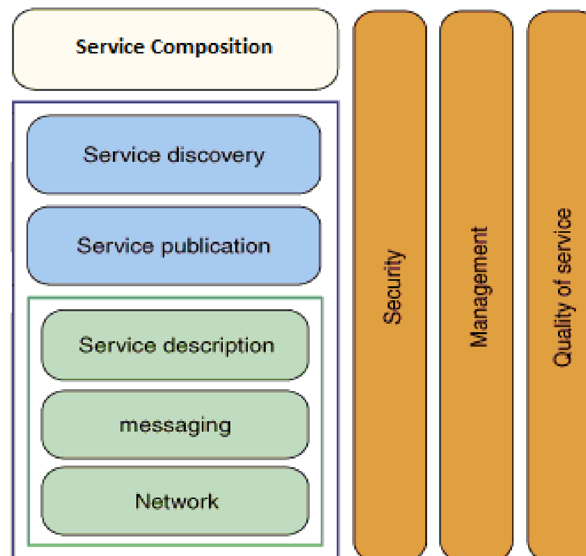

Figura 2.1: Architettura web service

- La prima parte corrisponde alla descrizione del servizio e al trasporto dei messaggi tra fornitore e richiedente. In essa sono presenti 3 livelli: Network, XML Messaging e Service Description, ognuno dei quali rappresenta una funzionalità fondamentale per la comunicazione.
  - **Network Layer:** La rete è la fondamenta dello stack. I servizi devono essere accessibili dalla rete per essere correttamente invocati e utilizzati da chiunque ne abbia necessità. La principale funzione di questo layer è consentire la trasmissione di messaggi da una macchina ad un'altra. Per fare ciò i servizi web possono utilizzare diversi protocolli, anche proprietari, ma solitamente viene scelto HTTP per questo scopo, visto la sua larga diffusione nel mondo di internet. Nei servizi ReSTful HTTP non svolge il ruolo di semplice busta per il trasferimento di messaggi ma fornisce a sua volta informazioni riguardo lo scopo dell'utente.
  - **Messaging:** Il secondo livello descrive la struttura dei messaggi utilizzati durante la comunicazione. La struttura è descritta solitamente da un linguaggio XML e deve essere conosciuta sia dal client sia dal fornitore per far si che questi "parlino la stessa

lingua". Questo livello può assumere un ruolo concettuale differente a seconda dello stile che si intende adottare per i servizi. Ad esempio i servizi SOA utilizzano una busta XML (SOAP) all'interno dei quali inseriscono dati ed altre informazioni importanti per l'esecuzione, mentre i servizi ReSTful, quando ne hanno necessita, utilizzano XML per formattare i dati trasmessi tramite HTTP.

- **Service Description:** È attraverso il Service Description che il fornitore comunica tutte le specifiche per utilizzare il servizio. Questo specifica le operazioni o le risorse che il servizio rende disponibili, i messaggi che il servizio si aspetta di ricevere e il protocollo a cui il client deve legarsi per accedere al servizio. Il Service Description è la chiave per mantenere il servizio loosely coupled e per ridurre la quantità di requisiti condivisi tra richiedente e fornitore.

  Il client può scegliere se utilizzare il Service Description per effettuare un "early binding", generando un stub attraverso il Service Description a cui collegarsi a compile-time, oppure per effettuare un "late binding", il client si lega al fornitore solo a run-time. Quest'ultima logica è quella più utilizzata in quanto più solida rispetto ad aggiornamenti del sistema.

  Possono, inoltre, essere aggiunte altre informazioni al Service Description quali sicurezza, QoS o modalità di comunicazione.

  Non tutti le tipologie di servizio obbligano l'utilizzo del Service Description, in alcune situazioni l'utente potrebbe non avere necessità di questo documento in quanto l'interfaccia del servizio è molto chiara e semplice. La scelta di usare il Service Description è comunque fatta da molti provider per evitare incomprensioni e malfunzionamenti.

- La seconda parte corrisponde alla fase di ricerca e pubblicazione su un registro di un Service Description in modo da permettere a tutti gli utenti di accedere ad un servizio. Questa sezione è divisa in due livelli: Service Publication e Service Discovery rappresentanti le due funzionalità del registro.

- **Service Pubblication:** Il Service Description può essere pubblicato usando diversi meccanismi dipendenti dalla dinamica di utilizzo del servizio. In particolare esistono due tipologie di pubblicazioni principali: diretta e indiretta. La pubblicazione diretta consiste nel trasferimento del Service Description direttamente al client grazie al quale può utilizzare il servizio. Questo meccanismo non è molto resistente agli aggiornamenti, il fornitore non può inviare una descrizione ad ogni cliente ogni volta che viene modificato il servizio, può però essere modificato in modo da aggirare questo problema, il Service Description è pubblicato ad un indirizzo fisso e può essere recuperato dal cliente in qualsiasi momento [33].

  La seconda tipologia di pubblicazione consiste nell'utilizzo di un registro sul quale pubblicare la descrizione e permettere al client la ricerca utilizzando propri parametri. Il cliente in questo modo può effettuare una ricerca e il registro restituisce una lista di servizi che rispettano i vincoli imposti dal richiedente.

- **Service Discovery:** Come per la pubblicazione anche la ricerca del servizio dipende dalla dinamica di utilizzo oltre alla modalità con cui questo è stato precedentemente reso disponibile. Il richiedente può effettuare la ricerca in due tempi differenti del suo ciclo di vita: a design-time o a run-time. A design-time il cliente cerca il Service Description attraverso il tipo di interfaccia che esso supporta. Nel secondo caso il cliente cerca il servizio basandosi sul tipo di operazioni, di comunicazione e sulla qualità del servizio.

  Con l'introduzione dei servizi ReSTful si è passati ad un nuovo concetto di service discovery. Grazie alla proprietà delle risorse di essere accessibili attraverso link è stato possibile pubblicare i servizi su semplici motori di ricerca web i quali ritorneranno un elenco di collegamenti al servizio. Grazie a questo i servizi ReSTful possono anche evitare di utilizzare un documento di Service Description.

• Infine l'ultima parte si occupa delle relazioni tra i servizi e di come questi possono essere composti. Questa sessione è formata da un unico

livello chiamato **Service Composition**, in cui vengono descritte le politiche adottate per la coordinazione di più servizi.

La composizione di servizi e la descrizione del loro flusso di esecuzione non può essere arbitraria, devono essere indicate le modalità con le quali avvengono. A seconda del modello utilizzato per implementare la coordinazione, si parla di orchestrazione o di coreografia [19].

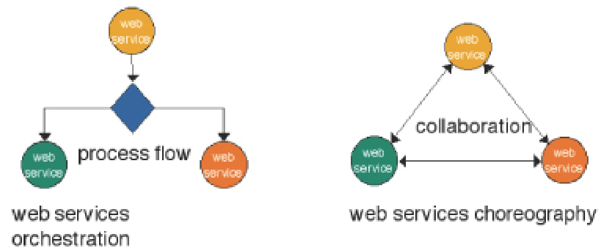

Figura 2.2: Orchestration e Choreography

La differenza sostanziale tra i due modelli consiste nella visione che i servizi hanno l'uno dell'altro.

Nell'orchestrazione l'attenzione è incentrata su ogni singolo partecipante. In pratica vi è un componente detto orchestrator che dirige l'esecuzione di tutti i servizi.

Nella coreografia ad ogni partecipante è offerta una visione parziale o addirittura globale del sistema.

In questa politica ogni partecipante, in funzione delle azioni intraprese dagli altri partecipanti (non necessariamente da tutti) decide quale sia l'operazione da intraprendere. Il maggior vantaggio di questa strategia risiede nella totale distribuzione delle informazioni di workflow ai componenti. Tuttavia, si deve evidenziare il fatto che i servizi coinvolti in una coreografia sono ben consapevoli del contesto in cui si trovano ad agire e, pertanto, risultano meno isolati rispetto a servizi utilizzati nell'orchestrazione. I servizi devono essere progettati preventivando di essere utilizzati all'interno di coreografia per cui devono esportare alcune funzionalità di supporto per poter gestire lo stato del flusso. Solitamente talune strategie che implementano coreografia, impongono il mantenimento di uno stato al fine di mantenere consistenti i dati relativi alla gestione del flusso.

Oltre ai livelli legati alle tecnologie appena descritti lo stack presenta altri concetti generali riguardanti problemi presenti ad ogni livello, illustrati in figura 2.1 come tre strati verticali. Questi rappresentano concetti ad alto livello come sicurezza, quality of service e gestione dei servizi.

- **Quality of Service:** Durante lo sviluppo di un servizio non è sufficiente specificare la sua interfaccia e la sua posizione ma è necessario considerare anche concetti come tempo di risposta e affidabilità che possono portare persino al fallimento nell'esecuzione di una richiesta. Questi concetti, assieme ad altri, fanno parte delle delle specifiche di QoS.

  - *Disponibilità:* Questo aspetto rappresenta la probabilità che il servizio richiesto sia immediatamente disponibile o meno (alti valori di questa probabilità indica il fatto che il servizio è

sempre disponibile, in caso contrario indica l'imprevedibilità di disponibilità del servizio).

Legato a questo vi è anche il tempo di riparazione (TTR) che rappresenta il tempo necessario per riparare un servizio che è fallito.

- *Accessibilità:* L'accessibilità rappresenta il grado di capacità di risposta ad una richiesta di servizio. Come nel caso precedente anche questo può essere rappresentato attraverso una probabilità che denota il tasso di successo. Può esistere la situazione in cui il servizio è disponibile ma non è accessibile.
- *Integrità:* Indica il grado di correttezza della operazioni che il servizio web deve svolgere.
- *Performance:* Questa qualità è misurata in termini di throughput e ritardo.

  Il throughput indica il numero di richieste servite nell'unità di tempo.

  Il ritardo indica il tempo trascorso tra l'invio della richiesta e la ricezione della risposta.
- *Affidabilità:* Rappresenta il grado di capacità da parte di un servizio di mantenere la qualità. Questo può essere misurato attraverso il numero di fallimenti in un dato periodo di tempo.

• **Gestione:** Un organizzazione che implementa servizi web deve interessarsi anche della gestione di questi. Un servizio necessita di cambiamenti di tanto in tanto. I messaggi che un servizio accetta, il routing dei messaggi da un servizio all'altro e l'utilizzo del servizio cambieranno nel tempo. Il fornitore deve essere in grado di gestire questi cambiamenti quando necessari senza che il client se ne accorga.

  Un servizio flessibile consente all'amministratore di aggiornare il proprio comportamento modificando i propri parametri di configurazione senza modificare il codice del programma.

• **Sicurezza:** La sicurezza in ambito aziendale assume notevole importanza durante scambio di informazioni, anche nei servizi web, perciò, è necessario implementare meccanismi che permettano una comunicazione sicura. I servizi web utilizzano firme digitali per identificare gli

utilizzatori del servizio e la crittografia dei messaggi per impedire a terze parti di poter intercettare e comprendere la comunicazione.

2.4 Benefici e rischi nell'uso dei Servizi Web

I servizi web permettono, a chi sceglie di utilizzarli, di ottenere i benefici intrinsechi di questa tecnologia, questi derivano principalmente dalla scelta di utilizzare tecnologie ampiamente riconosciute come standard, è comunque giusto definire questi benefici in modo da chiarire tutti gli aspetti hanno portato all'affermazione di questa tecnologia.

- **Riuso:** uno dei vantaggi più evidenti è la riusabilità. I servizi web possono giocare un ruolo molto importante nel migliorare il fattore di riuso del software all'interno delle aziende. Database, applicazione e componenti possono essere "racchiusi" all'interno di un servizio ed esposti come interfacce riusabili. Il concetto di riusabilità permette agli sviluppatori di comporre i servizi tutte le volte che ne avranno bisogno e utilizzando risorse già a loro disposizione per realizzare nuove funzionalità, questo si traduce in un guadagno di tempo (in quanto non è necessario riprogettare una parte già presente) e di conseguenza anche denaro.

- **Scalabilità:** un sistema si dice scalabile se l'overhead richiesto per migliorarne la potenza computazionale è minore dei benefici aggiunti dalla modifica. Questa proprietà è diretta conseguenza dell'indipendenza dell'utilizzatore dall'implementazione vera e propria del servizio. I client, infatti, conoscono solamente l'interfaccia offerta, una modifica del servizio quindi richiede uno sforzo trascurabile, dovendo preoccuparsi solamente di continuare a rispettare le specifiche precedentemente definite dal Service Description.

- **Load balacing:** potendo l'utente effettuare il collegamento dinamico è possibile da parte del fornitore distribuire le varie richieste su diversi servizi attivi che implementano l'interfaccia desiderata, il tutto in modo trasparente agli utenti, in questo modo il carico di lavoro viene distribuito portando ad un miglioramento delle performance e della Quality of Service.

- **Vendor indipendence:** si riferisce al legame che una aziende deve mantenere con il fornitore delle piattaforme (software o di sviluppo). Nello scenario precedente ai servizi web, chi progettava applicazioni che dovessero interagire con altre già presenti, doveva prendere in considerazione le piattaforme utilizzate per queste ultime e probabilmente riutilizzarle. Con l'inserimento dei servizi come interfaccia di comunicazione questo non è più vero. Questi consentono alle aziende di prendere decisioni sulla piattaforma da usare in base a propri criteri (costi, performance, ecc.) svincolandosi quindi dai marchi precedentemente utilizzati. Questo rappresenta un importante cambiamento "al potere", dal fornitore al consumatore.

- **Mantenimento degli investimenti:** si riferisce agli investimenti fatti in precedenza sui sistemi informatici già presenti in azienda (detti legacy), questi sistemi anche se antiquati continuano ad essere usati poiché gli utenti non vogliono o non possono rimpiazzarli. I servizi web possono "avvolgere" queste applicazioni, mantenendole in vita e salvando gli investimenti già fatti, questo è possibile perché i servizi web specificano solo il metodo di interazione e non quello di attuazione. Nel caso in cui l'azienda decida in un tempo successivo di sostituire il sistema il servizio manterrebbe il modo con cui i consumatori vi accedono senza modifiche ed investimenti ulteriori.

I pregi sopra elencati sono i punti di forza dei servizi web, ma essi portano con se anche qualche problema o difetto.

Una delle critiche principali veniva fatta a questa tecnologia era la scarsità di meccanismi di sicurezza [2]. L'utilizzo di HTTP come protocollo di trasporto infatti non permette di criptare la comunicazione e i messaggi scambiati tra fornitore e client vengono trasmessi in chiaro, quindi facilmente intercettabili da terzi. HTTPS (Hypertext Transfer Protocol over Secure Socket Layer) sarebbe in grado di garantire la riservatezza dei dati trasmessi, ma non sempre è utilizzabile e potrebbe causare problemi nell'attraversamento di alcuni meccanismi di sicurezza (come firewall).

Un meccanismo di autenticazione è fornito da HTTP, ma questi metodi sono primitivi e facilmente aggirabili, è necessario perciò utilizzare altri stratagemmi più sofisticati.

Negli ultimi anni si sono formati gruppi specializzati nello studio della sicurezza nei servizi web (un esempio è WS-Security uno standard creato da un gruppo di aziende del panorama internazionale). I meccanismi di sicurezza introdotti da questi studi si basano principalmente sull'uso di chiavi pubbliche per criptare il contenuto del messaggio. Il funzionamento di questi sistemi di sicurezza è molto semplice, ogni utente possiede una coppia di chiavi, una resa pubblica e una privata, matematicamente correlate, ma non deducibili una dall'altra. Chiunque voglia comunicare deve recuperare la chiave pubblica e criptare il messaggio con questa, in questo modo solo il destinatario sarà in grado di capirne il contenuto decriptando con la chiave privata.

Ma cosa accade se un terzo crea una coppia di chiavi a nome di un altro? Sarebbe in grado di ricevere e leggere i messaggi non a lui diretti. Per cui il mittente deve verificare che la chiave che utilizza sia effettivamente quella giusta, questo viene fatto con l'uso di certificati che correlano una chiave pubblica con un'identità con il supporto della Public Key Infrastructure (PKI) [7].

Oltre al problema della sicurezza sopra descritto, l'utilizzo di un servizio web all'interno di un'applicazione può portare ad una riduzione delle performance causata da diversi fattori ([6] e [5]).

Come per tutti i sistemi che utilizzano la rete come mezzo di comunicazione si deve tenere in considerazione il ritardo e i problemi che questa introduce, purtroppo questi non sono controllabili e variano a seconda delle condizioni del traffico.

Vi sono però altre questioni che non possono essere trascurate nel momento in cui si sceglie di utilizzare un web service. L'utilizzo di XML come standard per la costruzione di messaggi è ottimo per quanto riguarda la flessibilità e la manutenibilità (in quanto è facilmente comprensibile dall'uomo e basato su file di testo) ma è uno dei punti deboli, per quanto riguarda le performance, nei servizi. Esso infatti, rispetto alla semplice serializzazione del messaggio in binario, aggiunge anche una serie di informazione (ad esempio la descrizione dei dati presenti nel messaggio) che semplificano di molto la comprensione del contenuto ma occupano una banda molto superiore. Oltre a questo durante la costruzione e la decodifica del messaggio vanno effettuate una serie di operazioni imprescindibili come analisi, con-

valida dello schema, parsing e traduzione. Queste operazioni utilizzano in modo intensivo risorse di sistema (CPU e memoria). Ad esempio l'analisi e la convalida dello schema comporta un sacco di codifica/decodifica dei caratteri e elaborazione di stringhe.

Tutte queste operazioni riducono ulteriormente la velocità comunicazione ed elaborazione dei servizi.

È possibile agire su diverse parti per rendere il sistema più reattivo.

Innanzitutto si può pensare di diminuire la dimensione del messaggio XML ad esempio comprimendolo. L'XML si presta molto bene alla compressione essendo un documento di solo testo. L'utilizzo di compressioni come GZip garantisce un risparmio di banda pari a circa il 60% del traffico altrimenti utilizzato da un comune Web service. È chiaro tuttavia che il Service Provider deve inviare al client risposte compresse solamente se questo è in grado di gestirle. Normalmente un client specifica al server che è in grado di gestire GZip grazie ad una riga nell'header HTTP.

```
Accept-encoding: gzip
```

Il fornitore di contro deve comunicare al richiedente se il messaggio è stato compresso o meno anch'esso con una riga nell'header HTTP. È chiaro che in questo modo il documento XML non è più comprensibile per l'uomo fino ad una sua decompressione.

Il documento XML, inoltre, può essere convalidato nell'instante in cui è stato originato. Dal momento che il documento è stato convalidato al di fuori della richiesta è possibile disattivare il meccanismo di validazione. Purtroppo questa operazione non è possibile per scambi di messaggi tra applicazioni diverse.

Grazie a queste strategie è possibile ottenere buone prestazioni dei servizi, considerando che comunque le condizioni della rete influenzeranno molto sulle performance.

2.5 Tecnologie di base

Come già detto i web service per consentire il loro utilizzo su ogni piattaforma sono "costretti" ad utilizzare tecnologie (di trasporto e costruzione di

messaggi ed entità) ampiamente diffuse. Questo giustifica la scelta di HTTP e XML come basi per questa tecnologia sulle quali poi sono stati definiti dai vari enti (in particolare W3C e OASIS) i protocolli tutt'ora usati per l'iterazione con i servizi (SOAP, WSDL, WADL, UDDI, XML-RPC, ecc.).

È quindi necessario introdurre, innanzitutto, queste due tecnologie per poi poter descrivere i linguaggi di comunicazione dei web service.

2.5.1 HyperText Transfer Protocol

HTTP (Hyper Text Transfer Protocol) è un protocollo applicativo standard per la comunicazione attraverso il web, e definisce come deve avvenire lo scambio di messaggi tra due entità (client e server) ed in particolare quali sono i tipi di messaggi scambiati, la loro sintassi e la semantica dei campi in essi contenuti [8].

Questo protocollo è utilizzato per il trasferimento di diversi tipi di risorse, ad esempio file (in particolare HTML), risultati di query o script, documenti di vario tipo e altro ancora. Per identificare una risorsa vene utilizzato un URI (Uniform Resource Identifier) che ne indica la locazione ed alcune informazioni aggiuntive.

HTTP permette due tipi di connessione permanente o non, la differenza tra i due tipi risiede nel fatto che nella connessione non permanente dopo ogni scambio di richiesta risposta la connessione viene chiusa e per ogni nuova richiesta è necessaria una nuova apertura, mentre in quella permanente viene utilizzata la stessa connessione per diverse richieste (disponibile solo con la versione 1.1). Il primo caso ovviamente creava problemi nel caso si avessero una ingente quantità di risorse che era necessario scambiare, in quanto per ognuna si doveva inserire il preambolo di apertura della connessione.

Il protocollo HTTP viene detto stateless in quanto non conserva memoria sullo stato del client e server (ad esempio un client può eseguire la stessa richiesta al server per due volte a pochi secondi di distanza ricevendo la stessa risposta).

I messaggi sono sempre composti di tre parti:

- Linea di comando (Contiene il tipo di richiesta o lo stato in caso di risposta).

  In caso di *risposta* la riga contiene un numero di tre cifre indicante lo stato della richiesta in cui la prima cifra contiene l'informazione sul tipo di stato (error, success, informational, ecc.).

- Header: contiene informazioni aggiuntive riguardo i dati contenuti nel messaggio o riguardo client o server.

- Body: corpo della richiesta/risposta, contiene i dati utente.

HTTP si basa su un meccanismo di richiesta/risposta, vi sono perciò due tipi di messaggi.

Messaggio di richiesta

Questo tipo di messaggio è sempre mandato dal client al server.

La prima linea del messaggio di richiesta contiene tre campi: Metodo richiesto, URI, versione HTTP. Essa indica il metodo da applicare sulla risorsa identificata dall'URI, oltre alla versione HTTP utilizzata per la comunicazione.

L'*header* è composto da un insieme di linee facoltative che permettono di dare delle informazioni supplementari sulla richiesta e/o il client (Browser, sistema operativo, ecc.). Ognuna di queste linee è composta da un nome che qualifica il tipo di intestazione, seguito da due punti (:) e dal valore dell'intestazione. Gli header di richiesta più comuni sono:

- *Host*: Nome del server a cui si riferisce l'URI (È obbligatorio nelle richieste conformi HTTP/1.1 perché permette l'uso dei virtual host basati sui nomi).

- *User-Agent*: Identificazione del tipo di client (tipo browser, produttore, versione...)

Il *body*, infine, è un'insieme di linee opzionali e che permettono l'invio di dati al server.

Uno dei più importanti attributi di una richiesta HTTP è il *metodo* che il client intende eseguire, in HTTP 1.1 sono 8 i metodi che è possibile richiedere:

- **GET:** Serve ad un client per recuperare una risorsa dal server. Il metodo GET consiste in un semplice scambio di due messaggi. Il client mandando un messaggio con tale metodo al server. Il messaggio identifica l'oggetto che il client sta richiedendo tramite il campo URI. Se il server può inviare l'oggetto richiesto, lo fa nella sua risposta, indicando lo stato.

- **POST:** l'operazione POST fornisce ai client un modo per trasmettere informazioni ai server. Anche questa operazione si compone di due messaggi. Il client manda un messaggio POST e include l'informazione che desidera mandare al server indicando attraverso l'URI la risorsa che è in grado di processarle. Il server manda l'informazione elaborata al client.

- **PUT:** Analogamente a POST, l'operazione PUT fornisce ai client un modo per fornire informazioni ai server. La modalità di interazione è esattamente identica a quella del metodo POST. La differenza tra i due metodi sta nell'interpretazione dell'attributo URI. In questo caso infatti esso indica un oggetto sul quale il server dovrebbe immettere i dati inseriti nel campo body della richiesta (ad esempio un file). Il server risponde con lo stato dell'operazione.

- **DELETE:** L'operazione di DELETE da all'utente la possibilità di rimuovere oggetti dal server. Il client invia una richiesta con l'URI della risorsa che desidera eliminare e il server risponde con lo stato dell'operazione. In caso di risposta positiva non è detto che la risorsa sia stata eliminata, indica soltanto che il proposito del server è quella di eliminarla.

- **HEAD:** Il metodo HEAD ha lo stesso principio di funzionamento del GET ad eccetto del fatto che non viene restituita la risorsa ma

solamente informazioni su di essa. Solitamente questo metodo viene utilizzato per verificare l'esistenza di un oggetto.

- **TRACE:** Questo metodo da la possibilità al client di capire il percorso di rete che seguono i messaggi per arrivare al server. Il server risponde inserendo nel corpo della sua risposta il messaggio "TRACE". Nel caso in cui vi siano più salti intermedi ognuno inserisce il proprio messaggio in modo che il client possa riconoscerli.

- **OPTIONS:** Il metodo OPTION è utilizzato dai client per conoscere le capacità di un server. Se nella richiesta è inserito un URI il server risponde con le opzioni attinenti a quella risorsa.

- **CONNECT:** Il metodo di richiesta CONNECT è riservato esplicitamente ai server intermedi per creare un tunnel verso il server di destinazione. Sia dalla prospettiva di un client che da quella di un server, un tunnel è trasparente. Esso rimane stabilito finchè la connessione TCP rimane aperta.

Messaggio di risposta

Dopo che il server ha ricevuto una richiesta HTTP effettua le operazioni necessarie a soddisfarla ed invia una risposta al client.

La prima linea del messaggio risposta HTTP contiene tre campi: versione del protocollo, ID stato, messaggio di stato. Essa include quindi la più alta versione che il server supporta, un codice di stato che indica il risultato della richiesta e un messaggio di stato corrispondente.

L'ID di stato è un numero di tre cifre che indica il risultato di una richiesta. La prima cifra identifica il tipo di risultato e dà un'indicazione di alto livello, le altre forniscono più dettagli.

I codici di stato sono:

Codice	Descrizione
1xx	Informational (messaggi informativi)
2xx	Success (la richiesta è stata soddisfatta)
3xx	Redirection (non c'è risposta immediata, ma la richiesta è sensata e viene detto come ottenere la risposta)
4xx	Client error (la richiesta non può essere soddisfatta perché sbagliata)
5xx	Server error (la richiesta non può essere soddisfatta per un problema interno del server)

L'*header* è composto da un insieme di linee facoltative che permettono di dare delle informazioni supplementari sulla risposta e/o il server. Come per la richiesta ognuna di queste linee è composta da un nome che qualifica il tipo di intestazione, seguito da due punti (:) e dal valore dell'intestazione.

Gli header della risposta più comuni sono:

- *Server*: Indica il tipo e la versione del server. Può essere visto come l'equivalente dell'header di richiesta User-Agent.

- *Content-Type*: Indica il tipo di contenuto restituito. La codifica di tali tipi (detti Media type) è registrata presso lo IANA (Internet Assigned Number Authority); essi sono detti tipi MIME (Multimedia Internet Message Extensions). Alcuni tipi usuali di tipi MIME incontrati in una risposta HTML sono:

  – text/html Documento HTML

  – text/plain Documento di testo non formattato

  – text/xml Documento XML

  – image/jpeg Immagine di formato JPEG

Infine il *body* della risposta contiene il documento richiesto o informazioni necessarie per considerare soddisfatta la richiesta.

2.5.2 eXtensible Markup Language

XML (eXtensible Markup Language) è un meta-linguaggio a marcatori nato per dare la possibilità ai programmatori web di definire proprie estensioni a questo linguaggio creando tag proprietari. Esso non è altro che un insieme standard di regole sintattiche per stabilire le modalità secondo cui è possibile arrivare a tale scopo.

Anche se gli obiettivi iniziali della nascita di XML erano rivolti alla soluzione di un problema di standard per il Web, ben presto ci si accorse che XML non era limitato al solo contesto Web. Esso risulta essere abbastanza generale per poter essere utilizzato nei più disparati contesti: dalla definizione della struttura di documenti allo scambio di informazioni tra sistemi diversi, alla definizione di formati di dati. [3]

Ciò che rende XML indipendente da varie piattaforme è il fatto che tale tecnologia si basa sul formato testo e quindi un documento XML può essere letto chiaramente su qualsiasi sistema operativo, inoltre utilizzando regole standard questo meta-linguaggio risulta indipendente da specifiche piattaforme hardware, software o da uno specifico produttore, pertanto i marcatori da noi definiti sono comprensibili da tutti.

Il contenuto di un documento XML è costituito da marcatori e dati strutturati secondo un ordine logico determinato da una struttura ad albero. Ogni elemento dell'albero può essere caratterizzato da attributi che lo descrivono, e può contenere a sua volta altri elementi o testo, la struttura logica di un documento XML, comunque, dipende dalle scelte progettuali ed è chi lo costruisce a decidere come organizzare gli elementi.

Un documento XML inizia con una riga che lo identifica come XML e ne indica la versione

```
<?xml version="1.0" ?>
```

il corpo del documento, che segue questa prima riga, è composto da un insieme di elementi che devono essere delimitati da un tag di apertura e uno di chiusura. I tag sono composti da un insieme di caratteri circoscritti dai simboli '<' e '>'. Questi tag possono essere liberamente definiti dal-

l'utente e integrati con attributi inserendoli all'interno del tag di apertura dell'elemento come ad esempio

```
<Tag1 attributo="valore">
.
.
.
</Tag1>
```

XML richiede un certo rigore riguardo la formattazione dei documenti. Tutti i documenti XML devono essere come si suol dire "ben formati".

Perché un documento XML sia ben formato deve rispettare le seguenti regole:

- ogni documento deve avere un unico elemento root che contiene tutti gli altri elementi definiti. Le uniche parti che possono essere esterne al root sono i commenti e le direttive di elaborazione (ad esempio la riga iniziale);
- ogni elemento deve avere un tag di chiusura;
- gli elementi devono essere opportunamente nidificati, cioè i tag di chiusura devono seguire l'ordine inverso dei rispettivi tag di apertura;
- il linguaggio è case-sensitive;
- i valori degli attributi sono sempre racchiusi tra apici singoli o doppi.

Anche la scelta dei nomi dei tag è soggetta a determinate regole, ad esempio può iniziare solamente con una lettera o underscore e non può contenere spazi o caratteri speciali (come ad esempio * o +). Il contenuto del documento invece non segue particolari regole, qui possono essere inseriti tutti i caratteri appartenenti ai primi 128 della tabella ASCII.

All'interno del documento possono essere inserite anche istruzioni dirette al software incaricato di elaborare del documento XML e sono indicate tramite elementi speciali detti direttive di elaborazione, queste sono inserite nella prima riga assieme all'indicazione della versione separati da spazio.

I caratteri speciali che potrebbero provocare problemi durante l'uso nel testo possono essere utilizzati in due modi: o attraverso oggetti speciali, detti entità, che si sostituiscono a questi caratteri (ad esempio ”<” definisce il carattere ”<”), o attraverso la definizione di un blocco, delimitato dalle istruzioni ”<![CDATA[” e ”]]”, all'interno del quale non viene interpretato come XML.

Validità di un documento XML

Affinché questo documento sia interpretabile correttamente da chiunque voglia accedervi c'è bisogno di un meccanismo che definisca quali elementi sono utilizzabili e la loro struttura e relazioni fra essi. Per questo ad ogni documento XML ne viene associato un altro che ne descrive la grammatica. Un documento che rispetta le regole definite da una grammatica viene detto *valido*. La caratteristica di documento valido si affianca a quella di documento ben formato per costruire documenti XML adatti ad essere elaborati automaticamente. Attualmente due sono gli approcci più diffusi alla creazione di grammatiche per documenti XML: Document Type Definition (DTD) e XML Schema.

Tutti gli impieghi dei documenti XML si fondano su due operazioni preliminari: la *verifica* che un documento sia ben formato e la sua *validità* rispetto ad una grammatica. Chiunque utilizzi questo linguaggio perciò deve utilizzare un software che esegua questi due passi.

Nasce ora la necessità di studiare i linguaggi di definizione delle grammatiche.

Document Type Definition Il Document Type Definition (definizione del tipo di documento) è uno strumento il cui scopo è quello di definire le componenti ammesse nella costruzione di un documento XML e la struttura logica che devono avere questi elementi.

Un DTD:

- Definisce gli elementi leciti all'interno del documento.

- Definisce la struttura di ogni elemento. La struttura indica cosa può contenere ciascun elemento, l'ordine, la quantità di elementi che possono comparire e se sono opzionali o obbligatori.

- Dichiara l'insieme di attributi per ogni elemento e che valori possono o devono assumere.

- Fornisce infine alcuni meccanismi per semplificare la gestione del documento (come la possibilità di dichiarare entità).

La sintassi di un Dtd si basa principalmente sulla presenza di due dichiarazioni: <!*ELEMENT* > e <!*ATTLIST* >. La prima definisce gli elementi utilizzabili nel documento e la struttura del documento stesso, la seconda definisce la lista di attributi per ciascun elemento. All'interno della dichiarazione di un elemento sono indicati tra parentesi tutti i sotto-elementi che esso può contenere e il numero di occorrente di questi ultimi (l'indicazione è fornita attraverso un insieme di simboli. Ad esempio il simbolo '+' indica una o più occorrenze, mentre il simbolo '*' indica 0 o più). È necessario dichiarare anche se gli elementi sono vuoti o se contengono testo.

La dichiarazione degli attributi avviene indicando per ogni elemento il nome dei singoli attributi e, se si vuole aggiungere vincoli, l'insieme dei valori da essi assumibili oltre all'indicazione di obbligatorietà o meno dell'attributo.

È inoltre possibile inserire la dichiarazione di elementi supplementari, ad esempio di entità inserendole semplicemente nel documento.

Un esempio di DTD:

```
<!--- Dichiarazione di elementi -->
<!ELEMENT articolo(paragrafo+)>
<!ELEMENT paragrafo (immagine*, testo+, codice*)>
<!ELEMENT immagine EMPTY>
<!ELEMENT testo (#PCDATA)>
<!ELEMENT codice (#PCDATA)>

<!--- Dichiarazione degli attributi -->
<!ATTLIST articolo titolo CDATA #REQUIRED>
```

```
<!ATTLIST paragrafo
       titolo    CDATA #IMPLIED
       tipo      (abstract|bibliografia|note) #IMPLIED
>
<!ATTLIST immagine file CDATA #REQUIRED>

<!--- Dichiarazione di un'entità -->
<!ENTITY html "HyperText Markup Language">
```

Infine per associare ad un documento XML il relativo DTD che ne descrive la grammatica sono possibili due modi:

- inserirlo direttamente all'inizio dell'XML:

  ```
  <!DOCTYPE root[
  <!--- Dichiarazioni DTD -->
  ]>
  ```

- utilizzare un file esterno e riferire quest'ultimo attraverso l'URI, sempre all'interno del tag DOCTYPE e indicando l'elemento root sul quale applicare la grammatica.

Tuttavia l'uso dei Dtd per definire la grammatica di un linguaggio di markup non sempre è del tutto soddisfacente. I Dtd non consentono di specificare un tipo di dato per il valore degli attributi, né di specificare il numero minimo o massimo di occorrenze di un tag in un documento o altre caratteristiche che in determinati contesti consentirebbero di ottenere un controllo ancora più accurato sulla validità di un documento XML. Queste limitazioni hanno spinto alla definizione di approcci alternativi per definire grammatiche per documenti XML, tra questi il più noto è XML Schema.

XML SCHEMA XML Schema è un linguaggio di descrizione per i documenti XML. Questo è l'unico ad aver ricevuto la validazione ufficiale della W3C.

XML Schema, a differenza di DTD, è basato su XML, questo è indice dell'estrema flessibilità dell'eXtensible Markup Language.

In quanto documento XML, uno XML Schema ha un root element che contiene tutte le regole di definizione della grammatica, questo è rappresentato dal tag $< xs : schema >$.

All'interno del tag di root vengono inserite le dichiarazioni degli elementi, questo avviene utilizzando la dichiarazione $< xs : elementname ='' testo''/ >$ in cui l'attributo name indica il nome dell'elemento stesso. Ogni elemento può essere di due tipi:

- semplice: può contenere al suo interno un singolo valore (ad esempio stringa, intero, ecc.), il tipo del contenuto viene definito attraverso l'attributo *type*. Un elemento di tipo semplice non può contenere altri elementi e non prevede attributi;
- complesso: si riferisce ad elementi che possono contenere altri elementi e possono avere attributi. Definire un elemento di tipo complesso corrisponde a definire la relativa struttura nel seguente modo:

  ```
  <xs:element name="NOME_ELEMENTO">
      <xs:complexType>
          ... Definizione dei sotto-elementi ...
          ... Definizione degli attributi ...
      </xs:complexType>
  </xs:element>
  ```

  la definizione dei sotto-elementi avviene esattamente nello stesso modo della definizione dell'elemento padre, mentre gli attributi vengono definiti attraverso il tag $< xs : attribute\ name ='' titolo''\ type ='' xs : string''\ use ='' default''\ value ='' test''/ >$, dove 'use' può assumere il valore "required" se l'attributo appena dichiarato è obbligatorio o "default" se è previsto un valore di default che viene indicato in value.

  È inoltre possibile indicare per ogni sotto-elemento il numero di occorrenze previste utilizzando gli attributi minOccurs e maxOccurs.

  Per rendere il documento XML Schema più leggibile si può spostare la definizione della struttura di un elemento all'esterno del tag stesso, associando poi l'elemento alla struttura utilizzando l'attributo type

nel tag element ed assegnandovi come valore il nome della struttura precedentemente definita.

È infine necessario collegare il documento XML alla definizione della grammatica. XML Schema permette l'utilizzo di più definizioni grammaticali in modo da poter riutilizzare parti già scritte in precedenza. Questo apre a due possibili scenari:

- l'utilizzo di un solo documento che definisce l'intera struttura dell'XML. Questo va indicato all'interno del tag root assegnando ad uno specifico attributo l'URI dello schema. Esempio:

```
<elemento
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

   <!--- Dichiarazione dello schema utilizzato -->
   xsi:noNamespaceSchemaLocation="grammatica.xsd"
   titolo="titolo" >
```

- l'utilizzo di più documenti. Quest'ultimo scenario può portare a diversi problemi. Cosa accade, infatti, se all'interno di due diversi schemi sono presenti elementi o attributi con lo stesso nome? Questo viene risolto in due modi:

  a) viene data la possibilità di inserire il documento XML Schema solo al momento del bisogno dichiarando che un determinato elemento segue uno schema diverso da quello indicato per il root;

```
<!--- Dichiarazione dello schema utilizzato
per tutti i sotto-elementi -->
<articolo xmlns="articolo.xsd" titolo="Titolo">
 <paragrafo titolo="Introduzione">
  <testo>
   ......
  </testo>
 </paragrafo>
 <paragrafo titolo="Bibliografia">
```

```
<bibliografia

 <!--- Dichiarazione dello schema utilizzato
 per il sotto elemento specifico -->
 xmlns="bibliografia.xsd">

 <autore>
  Tizio
 </autore>
 <titolo>
  Opera citata
 </titolo>
 <anno>
  1999
 </anno>
</bibliografia>
</paragrafo>
</articolo>
```

b) le definizioni degli schemi vengono fatte nell'elemento root. Per ogni sotto-elemento viene indicato come prefisso il nome dello schema utilizzato separato dal nome dell'elemento dal simbolo ":".

```
<art:articolo titolo="Titolo"
<!--- Dichiarazione delle grammatiche
utilizzate per il documento-->
 xmlns:art="articolo.xsd"
 xmlns:bibl="bibliografia.xsd" >


  <art:paragrafo titolo="Introduzione">
   <art:testo>
    .......
   </art:testo>
  </art:paragrafo>
  <art:paragrafo titolo="Bibliografia">
   <bibl:bibliografia>
```

```
      <bibl:autore>
       Tizio
      </bibl:autore>
      <bibl:titolo>
       Opera citata
      </bibl:titolo>
      <bibl:anno>
       1999
      </bibl:anno>
     </bibl:bibliografia>
   </art:paragrafo>
  </art:articolo>
```

Capitolo 3

Service Oriented Architecture

SOA è uno stile architetturale ed un insieme di entità che mira a raggiungere l'interoperabilità di applicazioni omogenee o eterogenee situate sulla stessa macchina o attraverso la rete, utilizzando la logica di servizio riusabile [11].

Questi insieme di principi vengono usati durante le fasi di progettazione e integrazione di un sistema informatico distribuito, rappresentando tradizionalmente una base per l'architettura dei sistemi distribuiti.

Il paradigma SOA suggerisce una metodologia di progettazione in cui applicazioni complesse sono realizzate attraverso l'interazione di entità più semplici chiamate servizi, utilizzabili anche da più sistemi contemporaneamente.

Ogni servizio fornisce agli utilizzatori una specifica funzionalità ed è in grado di utilizzare quelle offerte da altri servizi riuscendo ad eseguire, in questo modo, operazioni anche molto complesse.

Conseguentemente alla semplicità con cui più servizi possono cooperare questa architettura aiuta l'integrazione di applicazioni che non sono state scritte con l'intento di interagire con altre applicazioni distribuite, e definisce tecniche per costruire nuove funzionalità utilizzando quelle già esistenti [17]. Fornendo un uniforme e onnipresente distributore di informazioni per un'ampia gamma di dispositivi informatici e piattaforme software, i servizi sono un'entità estremamente importante per i sistemi distribuiti.

È necessario un nuovo approccio nella costruzione di applicazioni basate sui servizi. SOA ha bisogno di strategie di sviluppo uniche, che sostituiscono i tradizionali approcci di costruzione del software e promettono la concretizzazione di sistemi di applicazioni plug-and-play e di moduli in grado di esprimere definite funzionalità per il dominio analizzato.

3.1 Funzionamento ed entità di SOA

Il metodo di progettazione alla base di SOA permette di fornire servizi sia ad applicazioni finali sia ad altri servizi distribuiti sulla rete utilizzando la pubblicazione e la ricerca di interfacce pubbliche.

Questa architettura definisce l'interazione tra richiedente (client) e fornitore del servizio come uno scambio di messaggi, evitando però di vincolare gli attori ad un preciso ruolo, essi possono assumere contemporaneamente funzione di client e fornitore [15].

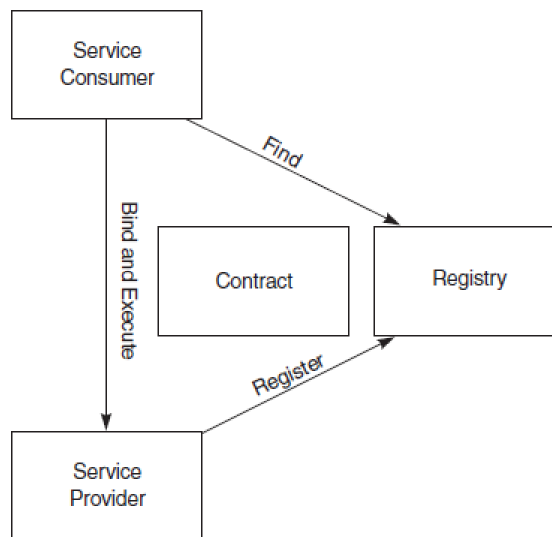

Figura 3.1: Attori di un iterazione service oriented

In un tipico scenario service oriented entrano in gioco diverse entità (come si vede in figura 3.1), molte delle quali sono già state viste nell'interazione tra servizi web nella sezione 2.2. ***Service Consumer***, ***Service Provider*** e ***Service Description*** o ***Contract*** infatti hanno esattamente lo stesso ruolo assunto nel contesto dei servizi web con una differenza: il cliente deve collegarsi in modo dinamico al servizio, il fornitore perciò deve pubblicare la descrizione del servizio su un registro in modo da permettere al cliente la ricerca a runtime. Il registro detto ***Service Discovery Agency*** è un semplice repository accessibile dalla rete contenente tutti i servizi utilizzabili. Il suo compito principale è accettare e memorizzare la descrizione dei servizi (contratto) pubblicate dai Service Provider e consegnare queste ad un Service Consumer interessato.

Oltre alle quattro entità principali ve ne sono altre definite da SOA e che possono essere presenti o meno all'interno di un sistema che segue questa architettura:

- ***Service Proxy:*** Il Service Proxy è un entità fornita dal Service Provider ai clients.

  Il Service Consumer esegue una richiesta al proxy attraverso un'API. Il Service Proxy cerca un contratto ed un riferimento al fornitore e formalizza la richiesta di esecuzione inviando un messaggio per conto del client.

  Il Service Proxy non è obbligatorio nell'architettura SOA, lo sviluppatore richiedente il servizio può scrivere il software necessario per eseguire l'accesso diretto. Questa entità, però, può incrementare le prestazioni del sistema implementando politiche di caching. Ad esempio salvando il riferimento al servizio remoto il Service Proxy riduce le interrogazioni sulla rete per utilizzi successivi dello stesso servizio.

  Questo tipo di iterazione ovviamente porta con se tutti i problemi dovuti al fatto che si mantiene un riferimento ad un oggetto remoto, se l'oggetto remoto cambia il riferimento non è più utilizzabile dal clients.

- ***Service Lease:*** Il Service Lease (contratto di "affitto" del servizio), che la Service Discovery Agency conferisce al richiedente, indica la quantità di tempo in cui il Service Contract resta valido. Quando il tempo è scaduto il client dovrà richiedere un nuovo collegamento alla Service Discovery Agency.

  Il concetto di Service Lease non fondamentale per SOA, ma lo è per il mantenimento delle informazioni di stato nel collegamento tra fornitore e richiedente, esso indica il tempo per il quale esso deve rimanere. Questo diminuisce anche l'accoppiamento tra le due entità interagenti (il client infatti potrebbe mantenere il contratto per lungo tempo senza mai rilasciarlo). Inoltre in caso di bisogno di aggiornamento del servizio il provider può procedere senza preoccuparsi delle richieste attualmente valide, queste riceveranno le modifiche una volta scaduto il contratto.

La SOA definisce, quindi una serie di ruoli ai quali viene assegnato uno specifico compito.

Va notato, infine, che i tre attori principali (Service Consumer, Service Provider e Service Discovery Agency) possono essere distribuiti sul territorio e possono utilizzare piattaforme tecnologiche differenti, con l'unico vincolo di dover utilizzare tutti e tre un canale trasmissivo comune.

Rimanendo sul mezzo trasmissivo, questo risulta essere un parametro dell'architettura, quindi l'approccio adottato dalle SOA ha il vantaggio di potersi integrare con diversi ambienti quali ad esempio la telefonia mobile e il Web, permettendo in tal modo di realizzare applicazioni multi-canale, fruibili cioè attraverso diversi dispositivi.

3.2 Caratteristiche di un'architettura SOA

Il paradigma SOA non è legato a nessuna specifica tecnologia implementativa, ma fornisce un insieme di caratteristiche che i servizi che compongono il sistema devono seguire.

Un sevizio deve possedere le seguenti caratteristiche [15]:

- ***Discoverable and Dynamically Bound:*** Un servizio deve essere poter essere ricercato in base ad un insieme di criteri conosciuti sola a tempo di esecuzione, ed eseguito senza essere a conoscenza di informazioni riguardo la sua implementazione.

  I clients non necessitano di alcuna informazione a compile-time, essi non conoscono nemmeno il formato dei messaggi di richiesta e risposta o la locazione del servizio finché non ne hanno effettivamente bisogno. Tutti questi dati vengono forniti dal contratto di servizio al momento della ricerca sul repository ed attraverso questi il client è in grado di effettuare un collegamento dinamico al servizio.

- ***Self-described:*** Ogni servizio deve, cioè, possedere interfacce ben definite contenenti tutte le informazioni necessarie per essere utilizzato. Il cliente deve poter capire la funzione di un servizio senza essere a conoscenza dei dettagli di progettazione del servizio stesso.

  La comprensibilità è particolarmente importante per i servizi, perché ogni utilizzatore può cercare ed usare un servizio in qualsiasi momento. Se il servizio non è comprensibile dal punto di vista funzionale, il cliente troverà difficoltà nel decidere quale servizio utilizzare.

- ***Interoperability:*** L'interoperabilità è uno dei concetti alla base della Service Oriented Architecture, e consiste nella capacità, da parte delle applicazioni realizzate con SOA, di lavorare insieme indipendentemente dalle piattaforme e dai linguaggi utilizzati per costruirle.

  Ogni servizio fornisce un'interfaccia, che può essere invocata attraverso un connettore, grazie alla quale è possibile definire un protocollo di comunicazione tra le due entità che vogliono interagire. L'interfaccia deve essere fornita in un formato dati che ogni potenziale cliente può comprendere, questo si realizza definendo i linguaggi standard utilizzabili per lo scopo, in modo che essi abbiano un formato decifrabile da ogni piattaforma (ad esempio formato testo) e che siano conosciuti da tutti gli utilizzatori.

- ***Loose Coupling:*** Per accoppiamento si intende il numero di dipendenze tra servizi e con i clienti. L'architettura service oriented promuove un accoppiamento debole tra fornitori e clients e l'idea di poche e ben note dipendenze tra queste figure. Più il service consumer ha bisogno di informazioni per invocare il servizio più aumenta l'accoppiamento.

  Il grado di accoppiamento del sistema influenza direttamente la sua modificabilità. In sistemi strettamente accoppiati la modifica ad un servizio richiederà cambiamenti anche agli utilizzatori del servizio stesso. Le modifiche di un servizio devono avere un basso impatto sugli altri servizi e sulle applicazioni che lo utilizzano. Un'interfaccia che non nasconde sufficientemente bene informazioni implementative crea un effetto domino quando necessita di cambiamenti. Ogni servizio deve nascondere le informazioni riguardo la sua progettazione interna, un servizio che espone queste informazioni limiterà la sua continuità modulare.

  SOA realizza l'accoppiamento debole attraverso il concetto di contratto e collegamento. L'utilizzatore non dipende direttamente dalla realizzazione del servizio ma solamente dal contratto che quest'ultimo supporta.

  L'**implementazione** di più servizi può, comunque, essere strettamente accoppiata, ad esempio se questi condividono un database o possiedono informazioni sulle reciproche implementazioni.

- ***Information hiding***: Ogni servizio deve mostrare attraverso la sua interfaccia solamente le informazioni fondamentali all'esecuzione (precondizioni, post-condizioni, parametri necessari, ecc.) e nascondere tutte il resto in modo da non creare una dipendenza superflua.

- ***Network-Addressable Interface:*** Un servizio deve possedere una interfaccia localizzabile attraverso la rete, utilizzando la quale un client può invocare il servizio. L'abilità di un applicazione di assemblare un insieme di servizi riusabili presenti su macchine diverse è

possibile solo se il servizio supporta un'interfaccia sulla rete, questo gli permette anche di essere indipendente dalla locazione fisica.

- ***Coarse-Grained Interfaces:*** Il concetto di granularità e correlata con il metodo di implementazione di interfacce all'interno di un sistema. I progettisti tendono a realizzare servizi in modo che questi siano il più possibile specializzati. Minore è la logica che il servizio incapsula e maggiore è la possibilità che questo possa essere riutilizzato.

Ogni servizio dovrebbe mappare una funzione distinta all'interno del dominio del problema. Durante l'analisi del problema e la creazione della soluzione il progettista deve creare dei confini ben definiti intorno ai servizi.

Un servizio deve essere in grado di operare in modo indipendente da altri componenti del sistema di cui fa parte.

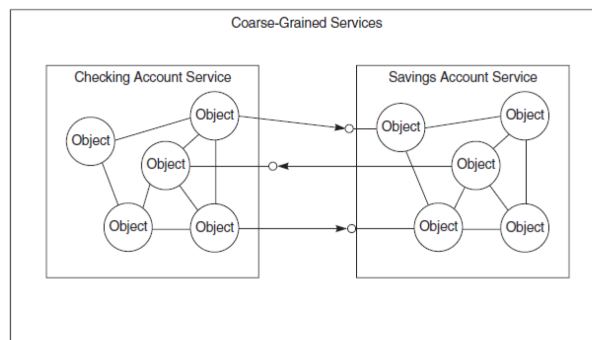

Figura 3.2: Servizi a grana grossa

- ***Location Transparency:*** L'utilizzatore del servizio non conosce l'allocazione del servizio finché non ne ha bisogno ed effettua una ricerca sul repository. Le operazioni di ricerca e collegamento dinamici permettono al servizio di spostarsi senza che i clients se ne accorgano.

  La trasparenza di locazione può incrementare le performance del sistema grazie all'utilizzo di politiche di load balancing. Più richieste di uno stesso servizio possono essere distribuite su più istanze di questo.

- ***Composability:*** La componibilità si riferisce alla produzione di servizi che possono essere liberamente combinati con altri servizi per creare nuovi sistemi. I progettisti dovrebbero creare servizi sufficientemente indipendenti in modo da poterli riutilizzare in applicazioni diverse da quelle per cui erano originariamente destinati.

  La possibilità di composizione di un servizio e legata alla sua struttura modulare. Questa permette permette ai servizi di essere assemblati in un'applicazione senza conoscere nulla sulla progettazione del servizio.

- ***Self-Healing:*** Vista la complessità dei sistemi distribuiti moderni la capacità di un sistema di recuperare da errori senza l'intervento umano è diventato molto importante.

  La capacità di self-healing di un sistema dipende da diversi aspetti.

  L'hardware deve essere in grado di recuperare da errori, la rete deve fornire la possibilità di connessioni dinamiche. Anche l'architettura con cui il sistema è stato progettato influenza la capacità di autoriparazione, un architettura che supporta collegamenti dinamici ed esecuzioni di componenti a runtime sono più preposti al self-healing.

  L'architettura orientata ai servizi possiede tutte le caratteristiche necessarie per essere in grado di auto-ripararsi. Ad esempio a seguito di un guasto se un servizio non è più in grado di eseguire il client può cercare, collegarsi ed eseguire un altro servizio che possiede caratteristiche simili, oppure visto il disaccoppiamento tra interfaccia e implementazione può essere un altro servizio ad eseguire le operazioni

rimanenti senza che il client ne sia a conoscenza.

Un errore durante l'esecuzione di un servizio non deve compromettere il funzionamento dei clienti o di altri servizi o intaccare lo stato dei propri dati interni.

Questi principi devono essere rispettati nella progettazione di applicazioni service oriented, in modo che i servizi possano essere facilmente utilizzati ed aggregati con poche ben note dipendenze.

3.3 Struttura a livelli di SOA

Analizzando a fondo SOA si può capire che un sistema che segue questa architettura è organizzata a livelli come mostrato in figura 3.3 ([1] e [16]).

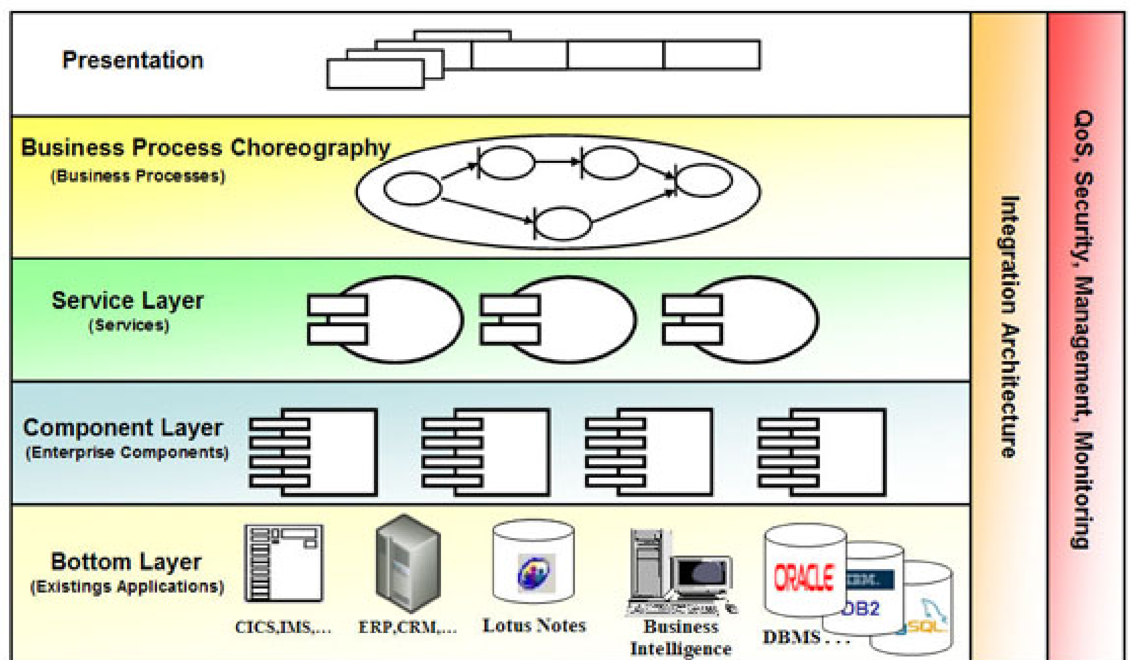

Figura 3.3: Architettura a livelli SOA[16]

Andremo ora ad analizzare ogni livello:

- **Bottom Layer:** contiene applicazioni già esistenti le quali forniscono un appoggio per i servizi. Ognuna di queste conserva la propria struttura e il proprio metodo di accesso alle risorse.

- **Component Layer:** questo livello ha il compito di realizzare le funzionalità e mantenere i servizi esposti. Per far ciò utilizza tecniche come load balacing.

- **Service Layer:** il Service Layer contiene i servizi veri è propri che possono essere cercati ed invocati.

- **Business Process Choreography Layer:** Questo livello si occupa della composizione di più servizi attraverso diverse politiche di coordinamento.

- **Presentation Layer:** Esso fornisce un interfaccia tra utente e servizio.

- **Integration Architecture Layer:** Questo strato permette l'integrazione di servizi attraverso l'introduzione di un insieme di funzionalità affidabili, come il routing intelligente e dei altri meccanismi di trasformazione.

- **QoS, Security, Management, Monitoring:** Questo strato fornisce le capacità necessarie per monitorare, gestire e mantenere QoS, sicurezza e prestazioni.

Capitolo 4

Web Service: architettura service-oriented

Dopo aver discusso, nei precedenti capitoli, tutte le proprietà e la struttura dei servizi web e dell'architettura service-oriented ed aver introdotto le tecnologie generali è ora possibile introdurre i protocolli ed i linguaggi standard per la realizzazione dei costrutti richiesti nell'utilizzo dei servizi SOA, questi sono tutti basati su linguaggi XML.

La W3C definisce due tipi di linguaggi standard per i web services: SOAP e WSDL, questi sono nell'ordine utilizzati per la costruzione dei messaggi da scambiare e per la definizione dei Service Description.

Oltre ai primi due si può introdurre un terzo standard: UDDI attraverso il quale è possibile costruire un registro per pubblicare e cercare i servizi.

4.1 Web Service e SOA

Come già detto SOA non presenta alcun riferimento all'implementazione dei servizi e ai modi con cui essi comunicano, comunque questa architettura presenta alcuni principi e ruoli che devono essere presenti in un sistema per essere definito **service-oriented** (come ad esempio la proprietà di auto-descrizione o di accoppiamento debole, oppure la presenza di una Service Discovery Agency). I sistemi che utilizzano appieno le potenzialità dei Web

Service d'altronde implementano un architettura che rispecchia pienamente SOA, ma non è detto che i servizi non possano essere utilizzati in modo differente.

Ad esempio, poniamo il caso di una rete interna ad un'azienda in cui vi sono applicazioni che fanno uso di web service. Se le modifiche al sistema si prevede non siano elevate, si può pensare di memorizzare tutte le informazioni relative ai servizi all'interno di ogni applicazione, evitando quindi di utilizzare un registry. In questo caso si dice che il riferimento al servizio è di tipo statico, si evita la pubblicazione dell'interfaccia e il client conosce già l'indirizzo e le informazioni necessarie per l'invocazione. Potrebbe, anche, non essere necessario l'utilizzo di un interfaccia per i servizi visto che questi verranno utilizzati solamente da pochi client e che essi possono già conoscere il modo con cui interagirvi.

Con l'introduzione della Service Discovery Agency si è passati ad un approccio dinamico, in cui il provider pubblica su un repository la descrizione del servizio (basato su XML) contenente tutte le informazioni necessarie per un suo utilizzo, il client effettua un interrogazione al repository per sapere quali sono i servizi disponibili effettuando il collegamento con esso solo al momento del bisogno, questo permette ad un componente del sistema di assumere entrambi i ruoli. In questo modo tutti i concetti del SOA vengono rispettati e viene realizzata una vera e propria architettura orientata ai servizi.

4.2 Web Service Definition Lenguage

WSDL è un linguaggio basato su XML utilizzato per descrivere in modo completo e machine-readable un servizio web [31].

Mediante WSDL può essere, infatti, costruita l'interfaccia pubblica del servizio, ovvero la sua Service Description. Un documento WSDL contiene informazioni su:

- le operazioni messe a disposizione dal servizio;

- il protocollo di comunicazione da utilizzare per accedere al servizio, il formato dei messaggi accettati in input e restituiti in output dal servizio ed i dati correlati;

- dove trovare il servizio (ovvero l'URI corrispondente).

Questo linguaggio cerca di separare gli aspetti astratti del servizio da quelli concreti, sono perciò definibili due livelli [22]:

- il livello *astratto* definisce il servizio in modo generico riferendosi alle operazioni offerte e al tipo di messaggi scambiati per ognuno di essi;

- il livello *concreto* in cui le descrizioni astratte vengono istanziate, legandole a una implementazione reale (protocolli, indirizzi di rete, ecc).

I benefici principali di questa distinzione sono portati dalla possibilità per una stessa descrizione astratta di possedere più possibili realizzazioni, potendola così riutilizzare più volte, e dalla capacità da parte di un servizio di mantenere la propria descrizione astratta invariata anche a seguito di una modifica della sua implementazione.

4.2.1 Struttura di un documento WSDL

Un documento WSDL è un file XML costituito da un insieme di elementi ben definiti. La radice del documento è chiamato *definitions*, esso contiene, oltre al riferimento allo specifico namespace, sette principali elementi grazie ai quali viene definito il servizio.

- **types**, fornisce le definizioni di tutti i tipi di dato utilizzati nei messaggi scambiati;

- **message**, fornisce la descrizione astratta dei messaggi che possono essere scambiati;

- **operation**, fornisce la descrizione astratta delle operazioni fornite dal servizio. Questo elemento contiene l'insieme dei messaggi scambiati per la specifica operazione;

- **port type**, rappresenta un insieme di operazioni astratte;

- **binding**, specifica il protocollo concreto e il formato dei dati specifico di un port type. Esso rappresenta l'istanziazione di un port type;
- **port**, associa ad un elemento binding uno specifico indirizzo, definendo così un endpoint di comunicazione;
- **service**, dichiara un servizio web come una raccolta di endpoint.

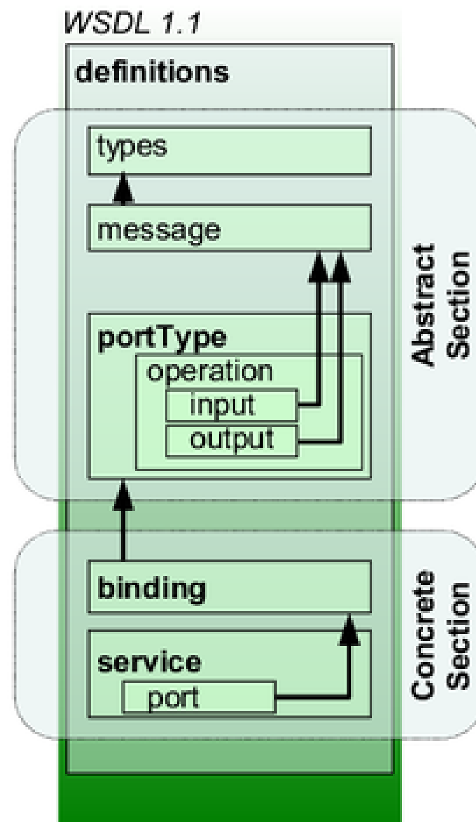

Figura 4.1: Struttura documento WSDL 1.1 [10]

Ogni sezione del documento WSDL descrive una specifica caratteristica del servizio. Come comunicare con il servizio è descritto dagli elementi *types* e *message*. Le operazioni offerte sono delineate da *operation* e *port type*. L'elemento *binding* descrive le caratteristiche di comunicazione. Infine l'indirizzo e i punti di accesso al servizio sono definite dalle sezioni delimitate dai tag *port* e *service*.

L'elemento *type* racchiude la definizione dei tipi rilevanti durante lo scambio di messaggi. Chiunque implementerà o utilizzerà il web service dovrà aspettarsi l'utilizzo di tali tipi all'interno dei messaggi e essere in grado di interpretarli. L'elemento type fornisce la chiave per comprendere i dati di tipo complesso comunicando agli utilizzatori come questi sono costruiti.

Per garantire la massima interoperabilità e neutralità WSDL predilige l'utilizzo di XML Schema come sistema standard di tipazione, è comunque possibile utilizzare altri linguaggi.

All'interno dell'elemento type è possibile fare riferimento a schemi esterni al documento oppure inserire un intero schema XML, o qualsiasi altra definizione di tipo valida e definibile tramite una notazione XML riconosciuta.

```
<definitions .... >
        <types>
                <xsd:schema targetNamespace="..."
                xmlns="..."/>
        </types>
</definitions>
```

In questo caso nella creazione dello schema non interessa definire degli elementi ma solamente dei tipi, perciò lo schema sarà semplicemente una successione di tag di tipo <*complexType*>, contenente l'insieme di tutti gli elementi semplici facente parte del dato.

L'elemento *message* contiene la definizione di tutti i messaggi che possono essere scambiati durante una interazione con il servizio. Ogni sezione di questo tipo assegna un nome al messaggio e definisce i parametri che fanno parte di esso (ovvero il nome e il tipo dei dati interni al messaggio).

Queste informazioni sono di fondamentale importanza all'interno del documento WSDL in quanto permettono la corretta comunicazione.

Il richiedente del servizio dovrà costruire un messaggio con le caratteristiche descritte in una di queste sezioni e riceverà in risposta un altro messaggio con lo stesso criterio.

```
<definitions .... >
        <message name="nomeMessaggio">
                <part name="nomeParametro" element="qname"
                  type="tipo"/>
                <part name="nomeParametro" element="qname"
```

```
                    type="tipo"/>
                 ...
        </message>
</definitions>
```

L'elemento *operation* rappresenta una singola funzionalità che il web service mette a disposizione. Attraverso questo è possibile assegnare un nome all'operazione e definire l'insieme di messaggi che verranno scambiati durante l'utilizzo della funzione stessa (questi devono ovviamente essere preventivamente dichiarati in una sezione message).

All'interno di ogni sezione *operation* sono contenuti una serie di elementi di tipo *input* e *output* che rappresentano rispettivamente i request e i responce message utilizzati per la specifica operazione.

La specifica WSDL definisce, inoltre, una insieme di pattern specifici per la descrizione dello scambio dei messaggi detti MEPs (Message Exchange Patterns), questi descrivono esattamente la presenza di elementi input e output, e la sequenza con cui esso sono riportati.

WSDL 1.1 descrive quattro diversi MEPs [31]:

- One-way: il servizio riceve un messaggio ma non ritorna alcuna risposta;
- Requst-Response: il servizio riceve un messaggio (richiesta) e ritorna una risposta correlata;
- Sollicit-Response: il servizio invia un messaggio e riceve una risposta, ma non rimane in attesa di questa;
- Notification: il servizio invia un messaggio e non riceve alcuna risposta.

L'elemento *port-type* è l'ultima sezione della parte astratta di un documento WSDL. Ad ogni port-type è associato un nome ed esso rappresenta un insieme di operazioni legate tra loro.

```
<portType name="Operazioni">
  <operation name="Operazione1">
```

```
    <input message="inputOp1" />
    <output message="outputOp1" />
  </operation>
  <operation name="Operazione2">
    <input message="inputOp2" />
  </operation>
</portType>
```

Con l'elemento *binding* inizia la parte concreta di un documento WSDL. In questa sezione viene descritto come i messaggi di ogni operazione vengono codificati e con quale protocollo vengono trasportati attraverso la rete, in pratica viene descritto come utilizzare concretamente una particolare interfaccia (portType) con uno specifico protocollo.

Ogni elemento *binding* è caratterizzato da un nome univoco e da un tipo che indica il port-type al quale esso fa riferimento.

All'interno di questo vanno poi inseriti degli elementi estensivi contenenti le informazioni specifiche del protocollo scelto.

WSDL definisce tre tipi standard di binding che descrivono come un servizio può essere invocato:

- SOAP binding
- MIME binding
- HTTP binding

Tra questi il primo è più utilizzato ed utilizza SOAP per codificare i messaggi. Per capire meglio come è strutturato un elemento di questo tipo conviene considerare un esempio:

```
http://schemas.xmlsoap.org/wsdl/soap/
```

innanzi tutto è necessario definire il namespace a cui appartengono gli elementi del binding di tipo SOAP.

```
<binding .... >
        <soap:binding transport="uri" style="rpc|document">
```

Dopo aver dato un nome al binding e aver scelto il port-type a cui fare riferimento è necessario inserire alcune informazioni generali.

L'attributo style di soap:binding indica se le operazioni relative allo specifico port-type e successivamente elencate sono di default RPC-oriented (i messaggi contengono parametri e valori di ritorno) o document-oriented (i messaggi contengono documenti). Questa informazione può essere utilizzata per selezionare un appropriato modello di programmazione. Un servizio service oriented utilizza lo stile document-oriented per il passaggio dei dati. L'attributo transport indica il protocollo di trasporto che si intende utilizzare per la trasmissione dei messaggi, ad esempio http://schemas.xmlsoap.org/soap/http indica il protocollo http.

```
<operation .... >
        <soap:operation soapAction="uri"
          style="rpc|document">
```

per ogni operazione che si desidera considerare è necessario ripetere il valore di style se non si desidera utilizzare quello predefinito, ed indicare soapAction, questo attributo è obbligatorio solamente se si utilizza HTTP come protocollo di trasporto, esso indica un elemento da inserire nell'header HTTP per far comprendere al Web service che azione intende intraprendere.

```
<input>
   <soap:body use="literal|encoded">
</input>
<output>
.
.
.
</output>
```

infine si indica come le parti dei messaggi in ingresso e uscita appariranno all'interno di un messaggio SOAP ed imposta la codifica da utilizzare se si è deciso di utilizzarla. Nel caso si utilizzi valore literal non si utilizza codifica e le definizioni astratte divengono definizioni concrete, altrimenti, assegnando un valore encoded vengono utilizzate regole di codifica.

```
    </operation>
</binding>
```

L'elemento *port* specifica un punto di accesso fisico al servizio ovvero indica l'indirizzo con il quale è possibile utilizzare le operazioni indicate in un port-type e descritte in un binding, attraverso uno specifico protocollo. Esso possiede due parametri: name indica il nome della porta appena definita e binding indica l'insieme di operazioni al quale fa riferimento. All'interno dell'elemento deve essere indicato l'indirizzo in cui è possibile invocare le operazioni.

Infine l'elemento *service* definisce il servizio come un insieme di elementi port assegnandogli un nome.

```
<service name="Servizio">
<port name="Port1" binding="Binding1">
<address location="www.localhost:8080"/>
</port>
</service>
```

L'idea di servizio come un insieme di endpoint è alquanto utile, perché lo stesso servizio può essere implementato utilizzando protocolli diversi pur riguardando le stesse funzionalità.

4.2.2 WSDL 2.0

Lo standard studiato fino ad ora riguarda la versione 1.1 delle specifiche WSDL, w3c ha però realizzato una seconda versione di questo standard (definito 2.0), in questa nuova revisione è stata modificata leggermente la struttura del documento eliminando alcuni elementi e modificando altri (figura 4.2) [34].

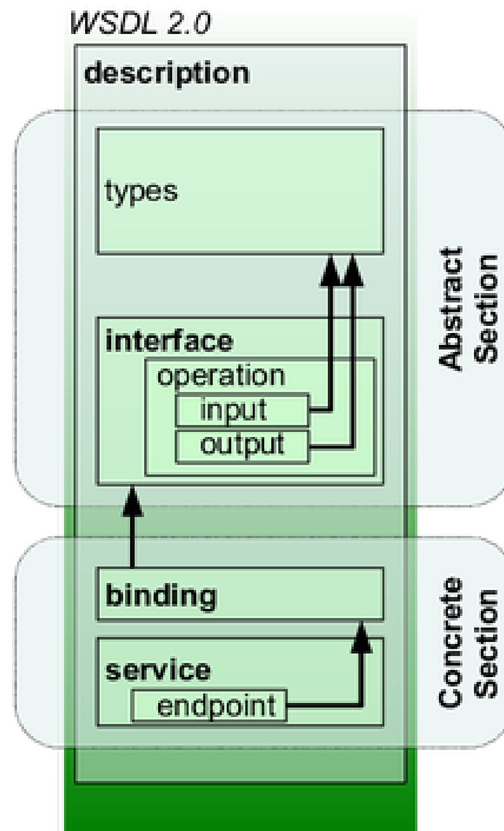

Figura 4.2: Struttura documento WSDL 2.0 [10]

Come si può notare non è più presente message, che è stato incorporato all'interno dell'elemento type, definendo così i messaggi come semplici elementi di un XML-Schema con tipo dipendente dai parametri. Tutti gli altri elementi del documento WSDL sono rimasti praticamente invariati ad eccezione per l'indicazione del MEPs all'interno di ogni operation.

In particolare WSDL 2.0 introduce o affina caratteristiche come l'ereditarietà, l'importazione di funzioni, la descrizione degli errori e il supporto ad HTTP e SOAP (ora definito completo).

Oltre a questo sono, anche, stati introdotti quattro nuovi Message Exchange Patterns e rinominati i precedenti. In particolare:

- In only: Equivalente al MEP one-way della versione precedente;
- In-out: Equivalente al MEP request-responce della versione precedente;
- Out-in: Equivalente al MEP solicit-responce della versione precedente;
- Out only: Equivalente al MEP notification della versione precedente;
- Robust In only: Corrisponde al in only con l'aggiunta della possibilità, da parte del service provider, di inviare un messaggio di fault in caso in cui vi sia un errore nell'esecuzione dell'operazione o nella ricezione della richiesta;
- Robust Out only: Corrisponde al out only con l'aggiunta dell'invio di un messaggio di preambolo e dando la possibilità al richiedente del servizio di inviare un messaggio fault subito dopo il primo messaggio;
- In optional out: Questo pattern permette sia l'invio che la ricezione di messaggi da parte del servizio, il messaggio di risposta però è opzionale e il richiedente potrebbe non ricevere alcun messaggio;
- Out optional in: Corrisponde la pattern in optional out con logica opposta.

Nonostante l'aggiornamento e l'inserimento di nuovi elementi nelle specifiche WSDL la versione più utilizzata per la costruzione di un Service Description di un web service rimane WSDL 1.1

4.3 Simple Object Access Protocol

SOAP è un protocollo leggero utilizzato per lo scambio di messaggi tra componenti software, esso è basato su un linguaggio XML e si appoggia sui principali protocolli a livello applicativo per la trasmissione in rete [30]. Queste due caratteristiche permettono a SOAP di risolvere i maggiori problemi che avevano le precedenti, e ampiamente diffuse, soluzioni proprietarie (ad esempio CORBA o DCOM), infatti queste ultime utilizzavano punti

di accesso alla rete non standard per la comunicazione e si scambiavano messaggi codificati principalmente in binario, cosa che creava problemi nell'attraversamento dei dispositivi di sicurezza utilizzati dalla maggior parte degli utenti. Incapsulando i messaggi SOAP all'interno di HTTP (o comunque altri protocolli standard) e utilizzando una codifica XML si riescono a bypassare queste problematiche permettendo un uso estremamente flessibile.

Nonostante la costruzione e lo scambio di messaggi SOAP sia gestito in modo molto efficace da librerie fornite con tutti i maggiori linguaggi di programmazione è utile approfondirne la struttura e il funzionamento per capire a fondo i web service.

SOAP ha una struttura divisa in tre o quattro parti [20]:

- **Envelope**: identifica il documento come un messaggio SOAP, contiene tutte le parti del messaggio;
- **Header**: opzionale, contiene meta-informazioni (riguardanti routing, sicurezza, transazioni e specifiche WS-*);
- **Body**: contiene le informazioni scambiate dalle richieste/risposte (dati formato XML);
- **Fault**: opzionale, contiene errori e informazioni di stato.

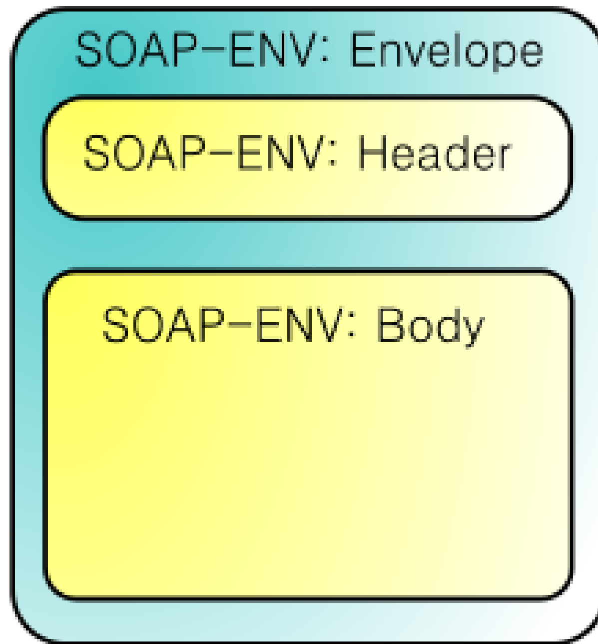

Figura 4.3: Struttura messaggio SOAP [9]

L'elemento *envelope* rappresenta la busta del messaggio, esso, oltre a contenere tutte le parti del messaggio, include due attributi caratteristici del documento. Il primo rappresenta il namespace relativo al messaggio in modo da distinguere gli elementi appartenenti allo standard SOAP da elementi di altri documenti in caso di ambiguità (questo attributo deve assumere il valore http://www.w3.org/2001/12/soap-envelope, altrimenti potrebbero essere generati errori o scartati i messaggi). Il secondo attributo soap:encodingStyle specifica regole per la serializzazione dei dati, può essere applicato a qualsiasi elemento, e tutti i figli dell'elemento che specifica tale attributo lo ereditano.

Nell'esempio seguente sono indicati i valori standard per tali attributi.

```
<?xml version="1.0"?>
<soap:Envelope
        xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
        soap:encodingStyle=literal"
```

```
        http://www.w3.org/2001/12/soap-encoding">
    ...
</soap:Envelope>
```

L'elemento *header* contiene informazioni aggiuntive al messaggio dirette non solo al destinatario finale, ma anche a nodi intermedi che hanno la possibilità di leggerle ed elaborarle per eseguire determinate azioni e/o aggiungere informazioni alla trasmissione.

Ogni elemento facente parte dell'header è chiamato header entries e deve essere associato ad una specifica grammatica definita dall'utente (namespaces esterni). Ad ogni entries possono, poi, essere assegnati ulteriori parametri che caratterizzano il comportamento del messaggio e del destinatario dell'intestazione. Questi sono tre:

- role: indica il destinatario dell'header-entries. Questo attributo può assumere diversi valori significativi. Può indicare direttamente l'indirizzo di un nodo identificando questo come destinatario, può assumere il valore next indicando che sarà il nodo successivo, qualunque esso sia, ad elaborare l'intestazione, infine può essere omesso indicando così il destinatario finale del messaggio.

- mustUnderstand: indica se il destinatario dell'header è obbligato ad elaborarla (se posto a '1') o può ignorarla (se posto a '0'). Nel caso in cui il nodo sia obbligato ma non sia in grado (ad esempio non conosce e non è in grado di recuperare il namespace corrispondente) deve ritornare un messaggio di fault, a meno di diverse indicazioni.

- relay: indica, nel caso in cui l'intestazione non sia stata processata, se il nodo deve ritrasmettere il blocco (valore '1') oppure no (valore '0').

Una volta processata da un nodo una entries deve sempre essere rimossa prima che il messaggio venga inoltrato, possono comunque essere inserite nuove voci nelle intestazioni durante elaborazioni intermedie. L'aspetto rilevante resta comunque l'impossibilità di leggere il contenuto del corpo del messaggio SOAP a meno del destinatario finale.

L'elemento *body* è la sezione principale di un messaggio SOAP, esso contiene l'informazione che il mittente desidera trasmettere al destinatario

finale. I dati al suo interno sono strutturati secondo il formato XML e può contenere qualsiasi tipo di dato. Questo elemento equivale semanticamente ad un header che deve essere elaborato e con destinatario quello finale.

Oltre a messaggi informativi l'elemento body può essere utilizzato per comunicare situazioni di errore, questo compito è svolto dall'unico elemento predefinito che può essere contenuto in questa sezione: l'elemento *Fault*. Esso può comparire una sola volta in un messaggio di risposta se il destinatario o un intermediario non sono stati in grado di elaborare la loro parte del messaggio di richiesta e serve a fornire informazioni su errori derivanti dall'elaborazione del messaggio.

Fault è costituito da quattro sotto-elementi:

- code: obbligatorio, fornisce il codice di identificazione per l'errore. Possono essere utilizzati sia codici standard di SOAP o definire codici personalizzati;
- reason: obbligatorio, fornisce una descrizione dell'errore;
- node: facoltativo, identifica il nodo al quale si è verificato l'errore;
- detail: facoltativo, riporta dettagli riguardo a chi ha generato l'errore.

Si riporta di seguito un esempio di struttura di un messaggio 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>
```

4.3.1 SOAP e HTTP

SOAP è stato pensato per utilizzare HTTP come protocollo di trasporto ed, anche se sono stati realizzati diversi metodi per inserire messaggi SOAP all'interno di altri protocolli internet, questo resta il più utilizzato.

In HTTP esistono due metodi fondamentali per inviare una richiesta ad un server: GET e POST corrispondenti ad altrettanti MEPs per i messaggi SOAP.

- SOAP request-response pattern: Consiste nell'utilizzo del metodo POST a cui viene allegato un messaggio SOAP, la risposta può contenere a sua volta un messaggio SOAP o semplici informazioni riguardo l'esecuzione e la ricezione della richiesta. Un messaggio HTTP post contenente un payload SOAP deve necessariamente inserire nella sua intestazione il campo SOAPAction già definito nella sezione riguardante WSDL.

- SOAP response pattern: Consiste nell'utilizzo del metodo GET (a cui non può essere associato alcun messaggio SOAP), la risposta consiste in un messaggio SOAP o in una segnalazione di errore. Per questo tipo di richiesta deve essere inserito all'interno dell'header HTTP il campo Accept: application/soap+xml.

Esempio servizio web SOA Di seguito sarà esposto un esempio che aiuterà a capire le modalità con cui i web service vengono descritti attraverso WSDL e le operazioni richieste attraverso SOAP.

Consideriamo un servizio costruito per un negozio online che sia in grado di fornire due operazioni: GetClientDetail e GetArticleDetail, utilizzati rispettivamente per ottenere dettagli su un cliente e un articolo.

```
<?xml version="1.0"?>
<definitions name="OnlineShop"
  targetNamespace="http://www.example.com/ShopOnline/Service.wsdl"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
```

```
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.example.com/ShopOnline/Service.wsdl">

<wsdl:types>
  <xsd:schema
    targetNamespace="http://www.example.com/ShopOnline/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <xsd:element name="ClientDetailRequest">
       <xsd:complexType>
         <xsd:all>
           <xsd:element name="IdClient" type="xsd:string"/>
         </xsd:all>
       </xsd:complexType>
     </xsd:element>

     <xsd:element name="ClientDetailResponce">
       <xsd:complexType>
         <xsd:all>
           <xsd:element name="IdClient" type="xsd:string"/>
           <xsd:element name="Nome" type="xsd:string"/>
           <xsd:element name="Cognome" type="xsd:string"/>
           <xsd:element name="NAcquisti" type="xsd:integer"/>
           <xsd:element name="NFeedbackPos" type="xsd:integer"/>
         </xsd:all>
       </xsd:complexType>
     </xsd:element>

     <xsd:element name="ArticleDetailRequest">
       <xsd:complexType>
         <xsd:all>
           <xsd:element name="IdArticle" type="xsd:string"/>
         </xsd:all>
       </xsd:complexType>
     </xsd:element>

     <xsd:element name="ArticleDetailResponce">
       <xsd:complexType>
```

```
          <xsd:all>
            <xsd:element name="NomeArticolo" type="xsd:string"/>
            <xsd:element name="Prezzo" type="xsd:decimal"/>
            <xsd:element name="Disponibilita" type="xsd:string"/>
            <xsd:element name="Descrizione" type="xsd:string"/>
          </xsd:all>
        </xsd:complexType>
      </xsd:element>
  </wsdl:types>

  <wsdl:message name="GetClientDetailInput">
    <wsdl:part name="body" element="tns:ClientDetailRequest"/>
  </wsdl:message>

  <wsdl:message name="GetClientDetailOutput">
    <wsdl:part name="body" element="tns:ClientDetailResponce"/>
  </wsdl:message>

  <wsdl:message name="GetArticleDetailInput">
    <wsdl:part name="body" element="tns:ArticleDetailRequest"/>
  </wsdl:message>

  <wsdl:message name="GetArticleDetailOutput">
    <wsdl:part name="body" element="tns:ArticleDetailResponce"/>
  </wsdl:message>

  <wsdl:portType name="ShopOnlineOperations">
    <wsdl:operation name="GetClientDetail">
      <wsdl:input message="tns:GetClientDetailInput"/>
      <wsdl:output message="tns:GetClientDetailOutput"/>
    </wsdl:operation>

    <wsdl:operation name="GetArticleDetail">
      <wsdl:input message="tns:GetArticleDetailInput"/>
      <wsdl:output message="tns:GetArticleDetailOutput"/>
    </wsdl:operation>
  </wsdl:portType>
```

```
<wsdl:binding name="ShopOnlineBinding"
  type="tns:ShopOnlineOperations">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="GetClientDetail">
    <soap:operation
      soapAction=
        "http://www.example.com/ShopOnline/GetClientDetail"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>

  <wsdl:operation name="GetArticleDetail">
    <soap:operation
      soapAction=
        "http://www.example.com/ShopOnline/GetArticleDetail"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

<wsdl:service name="ShopOnlineService">
  <wsdl:port name="ShopOnlinePort"
    binding="tns:ShopOnlineBinding">
    <soap:address location=
      "http://www.example.com/ShopOnline"/>
  </wsdl:port>
</wsdl:service>
```

```
</wsdl:definitions>
```

Nel momento in cui il client desidera eseguire un operazione dovrà inviare un messaggio SOAP al fornitore tramite HTTP POST indirizzandolo all'endpoit indicato nel documento WSDL. All'interno della richiesta saranno presenti i dati necessari per l'esecuzione, sarà il fornitore ad interpretare questi dati ed eseguire la giusta operazione, nel messaggio SOAP non è presente alcun riferimento diretto all'operazione richiesta dal client.

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ser="http://www.example.com/ShopOnline/Service.wsdl">
  <soapenv:Header/>
  <soapenv:Body>
    <ser:ClientDetailRequest>
      <IdClient>A567</IdClient>
    </ser:ClientDetailRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

Infine il fornitore risponderà al client con i dati elaborati.

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ser="http://www.example.com/ShopOnline/Service.wsdl">
  <soapenv:Header/>
  <soapenv:Body>
    <ser:ClientDetailResponce>
      <IdClient>A567</IdClient>
      <IdClient>Mario</IdClient>
      <IdClient>Rossi</IdClient>
      <IdClient>25</IdClient>
      <IdClient>18</IdClient>
    </ser:ClientDetailResponce>
  </soapenv:Body>
</soapenv:Envelope>
```

4.4 Specifiche WS-*

WS-* è un termine utilizzato per indicare un insieme di estensioni al web service framework, introdotte con la seconda generazione di servizi web.

Questo insieme specifiche è stato creato per gestire e standardizzare una serie di aspetti che precedentemente erano lasciati al fornitore del servizio e per questo erano oggetto di controversie. Gli ambiti inseriti all'interno delle WS-Specs vanno dalla sicurezza all'affidabilità alla politica di coordinamento, coprono la maggior parte delle caratteristiche legate alla QoS.

Queste estensioni sono, comunque, oggetto di continuo studio e miglioramento e molte di esse sono attualmente in fase di sviluppo.

Ogni specifica corrisponde ad una particolare intestazione dei messaggi SOAP. Analizzando queste gli attori partecipanti all'interazione sono in grado di capire come interpretare il messaggio.
Ora andremo a vedere alcune delle specifiche più rilevanti [23]:

- ***WS-Security:*** è un estensione di SOAP che implementa autenticazione e integrità a livello di messaggio. L'obbiettivo del WS-Security non è quello di introdurre nuove tecniche, ma quello di utilizzare le soluzioni già esistenti nell'ambito dei Web Service.

  Questa specifica descrive tre principali meccanismi:

  - come firmare i messaggi SOAP per assicurarne l'integrità;
  - come cifrare i messaggi SOAP per garantirne la riservatezza;
  - come accertare l'identità del mittente.

  Il punto di ingresso di WS-S è un header SOAP, chiamato <*security*> contenente i dati riguardanti la sicurezza e le informazioni necessarie per implementare meccanismi come firma e cifratura.

- ***WS-Policy:*** questa estensione è utilizzata dai Service Provider per definire politiche e condizioni alle quali chiunque utilizzi il servizio deve sottostare. Per esempio, un criterio può indicare che un Web Service accetta solo richieste che contengono una firma valida o la dimensione di un certo messaggio che non deve essere superata.

- ***WS-Adressing:*** questa è una delle specifiche chiave di WS-*. WS-Adressing è una specifica transport-neutral che permette ai web service di comunicare informazioni di indirizzamento. Si compone essenzialmente di due parti: una struttura per comunicare un riferimento a un web service endpoint, e un insieme di message adressing properties che associano una informazione di indirizzamento ad un particolare messaggio.

  WS-Adressing è un modo standardizzato per includere informazioni di routing all'interno di SOAP. Invece di fare affidamento al livello di rete un messaggio SOAP può contenere i propri dati di spedizione in un intestazione standard. Il livello di rete è responsabile solo per la consegna di quel messaggio all'utente finale.

- ***WS-Coordination:*** descrive un'estensione per fornire protocolli che coordinano le azioni di applicazioni distribuite.

  WS-Coordination ha un insieme di estensioni derivate che si basano su essa.

  - ***WS-Transaction:*** derivato da coordination, si occupa della gestione delle transazioni nei web service. Esso definisce due tipi di transazioni: Atomic Transaction per singole operazioni e Business Activity per operazioni composte e più complesse.

4.5 Universal Description Discovery and Integration

Come è stato detto all'inizio del capitolo (web service e SOA 4.1) un servizio web per seguire esattamente un'architettura Service Oriented deve essere pubblicato su un registro e poter essere ricercato da chiunque ne abbia necessità. UDDI fornisce un sistema di registry web e di meccanismi per la pubblicazione e la ricerca di informazioni riguardanti i web service (loacazione, interfaccia, informazioni sul fornitore, ecc.), con lo scopo di fornire una piattaforma pubblica per il service discovery. I registri sono distribuiti su siti di pubblico accesso sincronizzati tra di loro in modo da propagare l'informazione su tutti i nodi.

Come tutti i precedenti standard anche UDDI si basa su linguaggio XML ed utilizza il protocollo SOAP per la comunicazione da e verso l'esterno. In particolare vi possono essere due tipi di iterazione con un registro [21]:

- un fornitore può:
  - inserire le proprie informazioni in un registry;
  - pubblicare e classificare i propri servizi;
  - definire relazioni con altri fornitori;
- un utente può:
  - ricercare un servizio compatibile con le proprie esigenze;
  - ricercare informazioni su un fornitore;
  - avere accesso alle informazioni necessarie ad interfacciarsi con il servizio (WSDL).

Ogni servizio inserito all'interno di un registro deve fornire tutte le informazioni necessarie per garantire ad un utente di poter interagire con lui. Ogni informazione relativa ad un servizio memorizzata all'interno di un registro può essere divisa in tre parti:

- white page: contengono informazioni sul fornitore del servizio. Questa include il nome e una descrizione dell'azienda fornitore del servizio. Utilizzando queste informazioni, è possibile trovare un web service di cui si conoscono già alcuni dettagli;
- yellow page: fornisce una classificazione di un servizio basata su tassonomie standard. Poiché una singola azienda può fornire diversi servizi possono essere presenti diverse pagine gialle associate ad una pagina bianca;
- green page: utilizzate per descrivere come accedere ad un servizio (informazioni relative al binding del servizio). Siccome un servizio può possedere più voci di binding possono essere associate più green page ad uno stesso servizio.

Questa è una divisione concettuale delle informazioni relative ad un servizio, ma ogni registrazione UDDI è divisa in realtà in cinque parti, ognuna appartenente ad una delle tre page precedentemente descritte e definite come strutture XML.

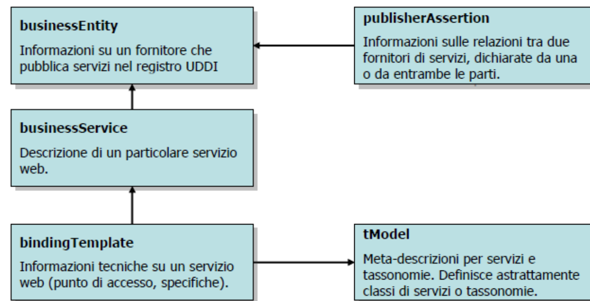

Figura 4.4: Struttura registrazione UDDI

- businessEntity: contiene dati relativi ad un fornitore di servizi. Queste informazioni possono essere ad esempio nome, indirizzo, contatti, ecc. (white page);
- businessService: descrive uno specifico servizio offerto da un fornitore. Contiene dati tipo nome, descrizione, categoria di appartenenza, ecc. (yellow page);
- bindingTemplate: contiene le informazioni utili per effettuare un collegamento al servizio, esso mette in relazione le informazioni generiche descrittive di un servizio con una sua reale implementazione (green page);
- pubblisherAssertion: contiene informazioni relative a relazioni tra i fornitori di servizio (white page);
- tModel: definisce la specifica di un certo tipo di servizio, precisando ad esempio l'interfaccia che esso deve avere. Un tModel in pratica fornisce un collegamento ad un documento WSDL riportante le caratteristiche del servizio. Ad ogni tModel potranno poi fare riferimento uno o più bindingTemplate (green page).

4.5.1 API UDDI

Le API UDDI forniscono ai programmatori un modo semplice per interagire con i dati archiviati all'interno di un registro tramite protocollo SOAP. In particolare queste API si possono dividere in tre categorie:

- inquiry: forniscono funzionalità che permettono agli utenti di effettuare una ricerca all'interno di un registro;
- publishing: forniscono funzionalità che permettono ai fornitori di inserire, modificare e cancellare informazioni su se stessi e su i propri servizi. Solo gli utenti autorizzati possono effettuare modifiche sui dati presenti nel registro, ogni fornitore perciò possiede una coppia username/password che gli permette di accedere al registro e modificare i propri dati;
- replication: forniscono funzionalità di comunicazione tra server per la gestione e sincronizzazione dei registri.

Capitolo 5

Servizi Web ReSTful

Fino ad ora sono stati analizzati solo servizi web orientati a SOA che utilizzano standard definiti dal W3C per comunicare e interagire (WSDL, SOAP, UDDI). Questi non coprono tutte le possibili metodiche di progettazione per i servizi web. Esiste, infatti, un altro tipo di realizzazione per servizi basato su un architettura ReST, in modo da conformarsi il più possibile con il resto del web, l'unica differenza che vi è tra un servizio ReSTful e un normale sito internet è il destinatario dei messaggi di risposta, un servizio è costruito per essere utilizzato direttamente da altre applicazioni alle quali sono indirizzate le informazioni contenute in questi messaggi che saranno poi elaborate (programmable web), di contro un sito web è costruito per essere visualizzato da un utente e le informazioni per essere lette da persone (human web). I motivi principali che, però, hanno spinto gli sviluppatori a questo cambiamento sono stati l'estrema semplicità e chiarezza che i servizi ReSTful hanno introdotto nella gestione delle funzionalità distribuite rispetto a quelli SOA.

Un servizio web ReSTful, infatti, è completamente comprensibile semplicemente guardando il tipo di richiesta HTTP e a chi è indirizzata. In questo modo il fornitore è in grado di capire immediatamente come soddisfare la richiesta senza dover analizzare ulteriormente il messaggio ricevuto, non vi è un ulteriore carico nella trasmissione che permette anche un risparmio di banda e non è necessario utilizzare librerie specifiche per la comunicazione tra server e client che utilizzano standard già ampiamente diffusi per i sistemi distribuiti.

Attualmente nel panorama dei servizi ReSTful si possono individua-

re due possibilità, caratterizzate dalla modalità con cui cliente e fornitore comunicano:

- servizi ibridi: rispetta tutti i vincoli definiti da ReST. Vengono visti come una via di mezzo tra servizi ROA e RPC.
- servizi Resource-Oriented: pienamente aderenti ai principi definiti dall'architettura ROA. Ogni funzionalità e informazione offerta dal servizio viene esposta come risorsa.

In realtà, però, quando si parla di servizi ReSTful viene sempre presa in considerazione solamente la seconda soluzione (definiti anche servizi totaly ReST) nonostante vi siano alcuni sistemi che attualmente utilizzano servizi ibridi. Questo accade perché, pur rispettando i principi ReST, i servizi ibridi non presentano un aspetto simile al resto del web e non sfruttano appieno la potenzialità di HTTP. Caratteristiche che, invece, rappresentano i punti di forza dei servizi ROA. É per questo che da ora in poi parlando di servizi ReSTful si farà implicito riferimento alla tipologia Resource-Oriented.

In questo capitolo della tesi, per completezza, verranno descritte entrambe le possibilità anche se la realizzazione di servizi Resource-Oriented sarà maggiormente approfondita.

5.1 Cos'è ReST?

Prima di poter studiare i servizi ReSTful è obbligatorio capire quali novità ReST ha introdotto nel mondo del web.

ReST (acronimo di Representational State Transfer) è un insieme di principi che dovrebbero essere seguiti per la realizzazione di applicazioni web. ReST ignora i dettagli implementativi dei componenti e i protocolli specifici che essi utilizzano per comunicare, ma si concentra sul ruolo dei singoli e sui vincoli di interazione. L'intero paradigma ruota attorno al concetto di risorsa. Ogni informazione tanto importante da aver assegnato un nome ed essere referenziata può essere considerata risorsa, e deve possedere almeno un identificatore univoco. La risorsa solitamente rappresenta un concetto che non varia nel tempo (come ad esempio l'ultima versione di un documento).

Per utilizzare le risorse le componenti di una rete comunicano attraverso un protocollo standard (ad esempio HTTP, ma non è obbligatorio l'utilizzo di questo) scambiandosi rappresentazioni. Queste ultime sono la concretizzazione dell'informazione referenziata dalla risorsa in un particolare formato e in uno specifico momento. Il contenuto, infatti, può cambiare nel tempo e essere rappresentato in diversi formati. Una rappresentazione per essere ben interpretata da chiunque la richieda necessita di un insieme di informazioni aggiuntive (meta-dati), queste indicano il formato con cui deve essere decifrata. I meta-dati sono nella forma nome-valore dove il nome definisce la struttura e la semantica del valore.

Nonostante la generalità, ReST è costantemente associato al protocollo HTTP, questo accade perché essendo ReST un architettura ideata per applicazioni web deve necessariamente essere legata a standard effettivamente utilizzati sul World Wide Web, inoltre il protocollo HTTP è particolarmente adatto per la richiesta e manipolazione di risorse grazie ai suoi vari metodi (l'ideatore dell'architettura ReST ha, infatti, partecipato alla stesura delle specifiche di HTTP).

L'architettura ReST descrive una serie di principi che un applicazione deve rispettare per essere definita ReSTful. Una buona definizione viene data da Fielding all'interno della sua tesi di dottorato:

> *ReST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.* [12]

Questo insieme di vincoli viene applicato in modo incrementale agli elementi di un sistema partendo dal null style (assenza di vincoli):

1. **client-server:** fornisce il principio di separazione degli interessi. Il server offre uno o più risorse e rimane in attesa di richieste. Il client, che desidera accedere alla detta risorsa, invia una richiesta che il server può scegliere di accettare o meno;

2. **stateless:** ogni richiesta isolata deve contenere tutte le informazioni necessarie per essere comprensibile, senza aver bisogno di riferimenti a precedenti richieste. Questo principio permette una implementazione

più semplice del server e una sua migliore scalabilità in quanto ogni risorsa può essere concretizzata dopo una richiesta.

Si possono distinguere due tipi di stato:

- application state: rappresenta lo stato dell'applicazione client, il quale è differente per ogni possibile utente. L'application state rappresenta il percorso che il client ha seguito attraverso le risorse.
- resource state: rappresenta lo stato della risorsa, è uguale per tutti i client ed è mantenuto sul server finché la risorsa non è eliminata. Questo stato è restituito al client sotto forma di rappresentazione.

La cosa importante è che il server non mantenga l'application state, se questo fosse necessario per la buona esecuzione della richiesta il cliente deve includerla all'interno della stessa.

3. **caching:** il risultato di una richiesta può essere memorizzato da un client o da un intermediario (vicino al client) in modo da diminuire la latenza. Per poter essere memorizzata in cache una risposta deve comunque essere etichettata come cacheable;
4. **layered:** è possibile inserire dei livelli tra client e server sotto forma di componenti intermediari, i quali trasmettono i messaggi e possono offrire ulteriori servizi. Ogni livello è visibile solo dal suo immediato vicino in modo da essere disaccoppiati tra loro e migliorare la flessibilità in caso di aggiornamenti;
5. **Uniform interface:** questo è il principio chiave dell'architettura ReST è ciò che la differenzia dalle altre architetture di rete. Ogni risorsa deve essere accessibile attraverso un insieme standard di operazioni che permetta al client di identificarla e interagirvi. *Il concetto base di questo principio è l'uniformità*: non è rilevante quali standard siano stati scelti per accedere alle risorse ma è importante che chiunque voglia accedervi lo faccia nel medesimo modo. Questo principio è descritto da quattro vincoli:
   - ogni risorsa deve essere identificata in modo univoco (URI - Uniform Resurce Identifier);

- le risorse possono essere manipolate solo attraverso una loro rappresentazione e utilizzando metodi standard comuni a tutti i client;
- ogni messaggio contiene tutte le informazioni necessarie per essere processato. Questo è diverso dal principio stateless sopra citato, in esso infatti si fa riferimento all'indipendenza dai soli stati precedenti;
- il client è responsabile del mantenimento del proprio stato e può effettuare una transizione solo attraverso hyperlinks. La rappresentazione di una risorsa dovrebbe contenere collegamenti agli stati vicini a quello attuale in modo da aiutare le applicazioni ad eseguire le transizioni (hypermedia as engine of application state).

6. **Code on demand:** è un vincolo opzionale che permette di aggiungere funzionalità al client quando ne necessita, in questo modo le funzionalità che ognuno di questi deve possedere possono essere ridotte.

Un ottimo esempio di ReST è costituito da un semplice sito web. Il client effettua una richiesta HTTP GET del documento (considerato risorsa) al server, la richiesta contiene l'indirizzo del sito, il quale rappresenta l'URI, e tutte le informazioni di cui il server ha bisogno per soddisfarla (contenute nell'URI stesso o nel header del messaggio HTTP).

Il server restituisce una rappresentazione in un formato specifico, che il client riesce ad interpretare grazie ai meta-dati allegati alla risposta (ad esempio HTML). Il documento può contenere collegamenti ad altre rappresentazioni in modo che il client possa spostarsi con molta facilità tra di esse. Il client può effettuare qualsiasi cosa voglia con questa rappresentazione senza alcun effetto sulla risorsa vera e propria.

5.2 Servizi Ibridi

La categoria dei servizi ibridi comprende tutta quella fascia di servizi che segue i principi definiti da ReST, ma non è classificabile come Resource-Oriented [24]. Questa tipologia di servizio cerca di inserire all'interno dell'URI tutte le informazioni utili all'esecuzione della richiesta.

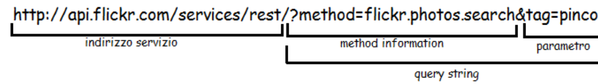

Figura 5.1: Esempio URI

Come si può vedere dall'esempio in figura 5.1 la query string permette al fornitore di conoscere sia l'operazione che il client intende eseguire (method information) sia i parametri richiesti nell'esecuzione (scoping information).

La differenza tra le due tipologie dei servizi ReSTful sta proprio in questa caratteristica. I servizi ibridi infatti utilizzano l'URI per il trasferimento dei parametri, esattamente come ROA, ma i metodi HTTP non mappano le effettive operazioni che poi saranno eseguite le quali, infatti, sono indicate nella query string: l'esempio di figura 5.1 mostra la richiesta di eliminazione di una risorsa attraverso il metodo HTTP GET.

Vi sono dei casi, comunque, in cui diventa complicato distinguere servizi ibridi da quelli Resource-Oriented. Consideriamo ad esempio il caso in cui venga utilizzato un servizio di tipo ibrido per ottenere i risultati di una ricerca. Una richiesta di questo tipo può creare confusione. In questo caso, infatti, viene utilizzato il metodo HTTP GET per ottenere un'insieme di dati, esattamente come è definito nella proprietà di interfaccia uniforme di ROA, l'indicazione dell'operazione inserita nell'URI può essere confusa con l'informazione di scoping.

La differenza si nota quando, però, si vanno a considerare operazioni di modifica, come l'esempio indicato in figura 5.1. In questo caso, infatti, si nota immediatamente l'impossibilita per questo servizio di seguire i principi Resource-Oriented.

Altra peculiarità dei servizi ibridi è rappresentata della definizione della loro *interfaccia uniforme*. Questa è costruita sopra HTTP, ma non utilizza interamente l'insieme dei metodi fornito da questo protocollo. Essendo, infatti, tutte le informazioni necessarie all'esecuzione della richiesta già presenti all'interno dell'URI, è sufficiente l'utilizzo di metodi HTTP che permettano la trasmissione di queste informazioni.

Particolarmente adatti a questo scopo sono i metodi GET e POST, il primo utilizzato nel caso in cui non vi sia necessità di allegare nessun altro tipo di dato (ad esempio richieste di informazioni, eliminazione, modifiche

semplici, ecc.), il secondo utilizzato nel caso in cui si debba inserire un carico nella richiesta (ad esempio modifiche più complesse, creazione di nuove risorse, ecc.).

Un esempio di servizi ibridi nel mondo reale: Flickr.com Flickr è un noto sito che permette agli utenti registrati di condividere foto con altri utenti nel web. Oltre alla normale interfaccia web Flickr offre un insieme di servizi sfruttabili da altre applicazioni. Questi servizi sono perfettamente collocabili all'interno della categoria dei servizi ibridi.

Infatti se un'applicazione intende utilizzare i servizi messi a disposizione di Flickr è sufficiente che esegua una chiamata GET o POST inserendo il metodo che desidera all'interno della query string, potendo cosi eseguire operazioni di eliminazione, creazione, modifica e lettura di risorse utilizzando solamente un ristretto insieme di metodi HTTP. Ad esempio una richiesta del tipo

```
GET http://api.flickr.com/services/rest/?method=flickr
.photos.addTags&photo_id="..."&tags="..."
```

utilizza il metodo GET per eseguire un aggiornamento dello stato della risorsa. Questo non è accettabile per i servizi ROA che obbligano ad inserire l'informazione sullo scopo della richiesta all'interno dei metodi HTTP.

5.3 Resource Oriented Architecture

Resource Oriented Architecture, altrimenti detta ROA, è un architettura software specifica per i servizi ReSTful con lo scopo di rendere questi ultimi più simili possibile al resto del web, questo implica la radicata presenza dei principi ReST e l'inserimento al suo interno di regole non scritte ma considerate da tempo standard de facto per la progettazione web (in modo da chiarire anche quei punti lasciati alla libera interpretazione dei progettisti) [24]. Il legame con HTTP e la ragionata struttura e logica dell'URI (che verrà descritta in seguito) sono la diretta conseguenza di questo tentativo.

Come per ReST, la risorsa è il fondamento della Resource Oriented Architecture. Essa identifica tutti i concetti che vengono esposti come servizi e fornisce un punto di accesso per il loro utilizzo (URI). In ROA qualsiasi funzionalità offerta da un servizio deve essere esposta come risorsa.

Si ha perciò un numero elevato di risorse ognuna corrispondente ad una operazione utilizzabile dal client.

Leonard Richardson e Sam Ruby hanno definito un insieme proprietà che concretizzano i principi discussi all'interno di ReST indirizzandoli verso l'idea di servizio e uniformandoli con gli standard effettivamente utilizzati nella progettazione di applicazioni web:

- La caratteristica principale che accomuna tutti i servizi ROA è la modalità con cui il client comunica al server le sue intenzioni. Ogni richiesta deve rispettare una certa struttura e deve suddividere le informazioni necessarie al server per soddisfare le richieste nel seguente modo:

  - HTTP fornisce *method information* ovvero definisce lo scopo della richiesta. Il metodo HTTP in questo caso indica quale sarà l'azione eseguita a seguito della soddisfazione della richiesta.

    Ad esempio il client potrebbe volere informazioni (ottenute ad esempio a seguito di una qualsiasi elaborazione), la modifica di un certo insieme di dati (ad esempio la modifica di una riga di un database), la creazione di un nuovo dato (ad esempio l'inserimento di una riga in un database), ecc.

  - L'URI deve fornire *scoping information* ovvero indicare su quale risorsa il client intende operare e gli eventuali parametri necessari.

- Come per ReST, la risorsa è il fondamento della Resource Oriented Architecture. Essa identifica tutti gli aspetti che un servizio ReSTful intende mettere a disposizione dell'utente. Siccome una risorsa è esposta tramite un URI, un servizio di questo tipo fornisce un URI per ogni aspetto utile, dando al client un punto di accesso diretto.

  Questa caratteristica è definita *adressability* e rappresenta una delle proprietà più importanti per l'utente finale. In assenza di questa il client dovrebbe trovare un modo indiretto per accedere a ciò che desidera, oppure rinunciarvi.

  Ad esempio consideriamo un'applicazione che ha necessità di effettuare una ricerca, se questo servizio non fosse indirizzabile questa non avrebbe la possibilità di accedere direttamente ai risultati inviando

una singola richiesta (ad esempio una ricerca su google risulta indirizzabile `http://www.google.com/search?q=webservice`, è possibile utilizzare questo URI ed accedere direttamente ai risultati voluti).

Il concetto adressability permette anche un risparmio di banda attraverso l'utilizzo della cache. Se una risorsa non possedesse un indirizzo questa non potrebbe essere recuperata a seguito di una successiva richiesta, in quanto non si avrebbero associazioni tra richiesta e risorsa memorizzata.

- ogni URI dovrebbe essere descrittivo della risorsa a cui è associato e deve avere una struttura che varia in modo predicibile. Vi deve cioè essere una corrispondenza intuitiva e logica fra risorsa e identificatore ed in particolare quest'ultimo deve essere rappresentante del percorso che l'utente ha seguito per raggiungere la risorsa desiderata.

  Nel paradigma ReST l'URI non deve seguire alcuna struttura o logica, ma è un principio ampiamente utilizzato sul web e facilita l'utilizzo del servizio da parte dell'utente ed è, quindi, stato inserito all'interno dell'architettura ROA.

- ogni risorsa può avere più rappresentazioni (ad esempio in diversi formati), un client che effettua una richiesta deve essere in grado di comunicare al server quale rappresentazione desidera tra quelle messe a disposizione. Vi sono due possibilità che il client può utilizzare per scegliere la rappresentazione più adatte alle sue esigenze:

  – inserire il formato all'interno del campo accept dell'header HTTP.

  – includere le informazioni sul formato all'interno dell'URI, ciò significa che quest'ultimo contiene tutto quello che è necessario al server per soddisfare la richiesta.

    L'architettura Resource Oriented predilige questo secondo metodo, ma il rischio è la possibilità di confondere la logica dietro gli URI associati alla risorsa: due rappresentazioni della stessa risorsa con due nomi diversi potrebbero indurre a pensare che riferiscano a due concetti diversi. È possibile ridurre questo problema esponendo un URI che riferisca ad una rappresentazione

standard detta ”forma platonica” a cui si fa riferimento nel caso in cui si voglia riferire alla risorsa di per sé.

- ogni richiesta deve essere completamente isolata, ovvero deve contenere tutte le informazioni necessarie per permettere al server di soddisfarla. Non deve dipendere da richieste precedenti, o riferimenti esterni. Se questi dati fossero importanti per il server, il client dovrebbe includerli all'interno della richiesta stessa.

  L'assenza di stato è uno dei motivi per cui si associa HTTP a questa architettura. Esso, infatti, fornisce la proprietà di stateless di default evitando quindi di dover impiegare ulteriore lavoro per rendere i servizi stateless.

  La proprietà di stateless permette anche l'utilizzo tecniche per migliorare le prestazioni. Grazie a questo è possibile distribuire le varie richieste a diversi server senza che questi abbiano metodi di coordinazione in quanto ogni richiesta è auto-contenuta. Il risultato di una richiesta stateless è anche più facile da memorizzare in cache, infatti un applicazione non deve preoccuparsi della dipendenza da altre richieste, può scegliere di memorizzarla guardando il singolo risultato.

- il risultato della richiesta ad un servizio restful solitamente non contiene solamente dati ma anche collegamenti ad altre risorse. Il server deve fornire al client la possibilità di passare ad uno degli stati successivi con semplicità. In pratica vengono allegati alla risposta gli URI che puntano ad un altro possibile stato dell'applicazione.

Un servizio totalmente ReSTful fornisce un collegamento logico tra tutte le risorse esposte, questo permette di aumentare la scalabilità del sistema. Nel caso in cui mancasse questa proprietà, infatti, il server dovrebbe fornire istruzioni a parole su come costruire l'URI di una risorsa collegata ed i client dovrebbero essere programmati ad-hoc per seguire queste istruzioni. In caso di aggiornamento del servizio, però, si potrebbero avere malfunzionamenti per questi che utilizzerebbero ancora i precedenti pattern per costruzione degli URI e avrebbero bisogno di modifiche per continuare l'esecuzione.

Lo scopo di questa proprietà è proprio quello di evitare questa situazione fornendo dei link diretti che sono immediatamente utilizzabili dai client.

- i servizi resource oriented, come già detto, utilizzano HTTP per definire lo scopo di una richiesta. Un utente deve perciò utilizzare HTTP per comunicare le sue intenzioni e deve essere in grado di prevedere gli effetti di ogni metodo a sua disposizione. Per questo ogni metodo HTTP deve essere ben definito e standardizzato in modo che chiunque voglia utilizzarlo per accedere ad una risorsa lo possa fare nel medesimo modo e non si presentino incomprensioni tra gli utilizzatori. ROA definisce una specifica *interfaccia uniforme* che fornisce un significato ad ogni metodo:
  - GET: richiesta dello stato della risorsa.
  - PUT: richiesta di creazione o modifica di una risorsa. Il client ha la possibilità di associare un payload a questo metodo.
  - DELETE: richiesta di eliminazione una risorsa.
  - POST: utilizzata per creare una nuova risorsa da una già esistente, quest'ultima assume il ruolo di genitore di quella nuova. Da qui nasce il concetto di *"resource factory"* ovvero risorse utilizzate solamente per la creazione di altre risorse sue subordinate (Un esempio concreto può essere ottenuto considerando una tabella di un database, utilizzando POST su una risorsa di tale tipo si otterrebbe ad esempio la creazione di una nuova risorsa rappresentante una riga di tale tabella).

L'interfaccia uniforme impone alcuni vincoli alle operazioni. GET deve essere **sicuro**, a seguito di questa operazione non vi devono essere cambiamenti rilevanti nello stato del server. PUT e DELETE devono essere **idempotenti**, effettuare più di una volta una di queste due operazioni ad uno stesso URI deve avere gli stessi effetti dell'esecuzione di una richiesta.

Overloaded POST: Vi sono casi in cui i metodi HTTP descritti dall'interfaccia uniforme potrebbero risultare inadatti. Pensiamo ad

esempio alla necessità di dover comunicare al client un elevato numero di parametri, inserirli nell'URI potrebbe essere impossibile (ad esempio se il server ha dato un limite alla lunghezza dell'URI) e comunque risulterebbe poco chiaro. In questi casi è possibile utilizzare POST, anche non rispettando l'interfaccia uniforme descritta da ROA.

Il corpo della richiesta formalizzata con POST overloaded deve contenere i parametri e specificare lo scopo effettivo della richiesta (ottenere informazioni, modificare la risorsa, ecc.). Ad esempio per ottenere la rappresentazione di una risorsa si potrebbe utilizzare il metodo POST indicando come destinatario la risorsa desiderata ed indicare all'interno del body method=GET¶m=1.

5.4 Servizi ReSTful: Design

Dopo aver visto i principi che caratterizzano i servizi Resource-Oriented e che permettono, quindi, di capire se un servizio web appartiene o meno a questa classe, è ora possibile indicare l'insieme dei principali passi che devono essere seguiti da chiunque intenda realizzare servizi ReSTful. Le linee guida descritte in seguito aiutano i progettisti a ragionare su quello che si vuole ottenere e a non perdere di vista i principi descritti da ReST e ROA.

Se il progetto viene ben strutturato si possono individuare i seguenti step [24]:

1. Analizzare i dati del problema ed individuare le risorse.

   Per ogni risorsa:

   2. Individuare il tipo.
   3. Assegnare un nome univoco (URI).
   4. Individuare l'insieme delle operazioni permesse (sottoinsieme del set indicato nell'interfaccia uniforme).
   5. Progettare le rappresentazioni rese disponibili.
   6. Individuare i collegamenti con altre risorse.

7. Progettare le risposte del servizio.

Analizzare i dati del problema ed individuare le risorse: Come ogni percorso di progettazione anche questo inizia con l'analisi del problema. In questa fase si dovranno estrarre dai documenti i dati rilevanti e il loro significato nel contesto, per poi poter decidere se è necessario esporli come risorse.

Solitamente durante questa fase di analisi si divide il sistema in entità individuate attraverso la presenza di un sostantivo e si determinano le azioni che esse devono eseguire (ad esempio un cliente di uno shop online deve poter acquistare un prodotto). Per ognuna di queste entità e azioni verranno create delle risorse che ne permettono l'esposizione all'utente.

La creazione di nuove risorse anche per ogni azione è necessaria visto l'obbligo che ha ogni risorsa di rispettare l'interfaccia uniforme. Su queste, infatti, è possibile effettuare solo operazioni di creazione, lettura, modifica ed eliminazione e non è possibile attraverso queste eseguire altre azioni.

Individuare il tipo della risorsa: In generale esistono tre tipi di risorsa in cui è possibile suddividere i concetti individuati al punto precedente:

- *Risorsa radice:* Una risorsa di questo tipo è vista come contenitore di altre risorse. Essa deve fornire, attraverso il proprio URI, un qualche tipo di informazione che accomuna tutte le risorse figlie.

  Su essa non ha nessun senso eseguire operazioni di modifica o cancellazione. È possibile, però, eseguire una lettura (ottenendo l'elenco di tutte le risorse contenute) e una creazione (che si traduce in un inserimento di una nuove risorsa figlia. Questo tipo di risorsa è un tipico esempio di "resource factory").

- *Risorsa oggetto:* Una risorsa di questo tipo corrisponde ad una singola entità individuata durante l'analisi del problema (ad esempio un cliente specifico di uno shop online).

- *Risorsa operazione:* Una risorsa di questo tipo corrisponde ad un'azione individuata durante l'analisi del problema. Un utente può passare alcuni parametri per l'esecuzione dell'operazione attraverso l'URI all'interno della query string.

  Su questo tipo di risorsa può avere significato eseguire soltanto operazioni di lettura in quanto si può voler solo ottenere informazioni.

Assegnare un URI alla risorsa: Dopo aver individuato le risorse ed aver determinato il loro tipo è necessario assegnare un nome, sotto forma di URI, per permettere l'accesso al client. Ogni identificatore deve essere descrittivo della risorsa, ogni client deve, cioè, essere in grado di capire come utilizzare la risorsa solamente vedendo l'URI. Il fornitore deve mettersi nei panni dell'utente del servizio per capire se l'URI sia veramente esplicativo della risorsa ad essa associato.

L'URI deve rispettare un'insieme di vincoli che permette al progettista di costruirlo nel modo più adatto:

- avere una struttura gerarchica in modo che i dati siano organizzati al meglio. Ogni elemento della gerarchia è divisa da '/'.
- utilizzare forme di punteggiatura per evitare di utilizzare gerarchie quando non esistono ad esempio se si voglio referenziare due risorse all'interno dello stesso URI queste sono allo stesso livello. Per separare le variabili si possono utilizzare caratteri come la virgola o il punto e virgola.
- utilizzare la query string nel caso in cui si debbano passare dei parametri alla risorsa.
- preferire l'utilizzo di nomi a quello di verbi. L'utilizzo di verbi, infatti, risulta un po' fuorviante perché induce a pensare che ad esso è associata un'operazione invece che una risorsa.
- contenere la lunghezza degli URI. Si tratta di una indicazione pratica che favorisce la leggibilità, ma anche di un limite fisico imposto dalla maggior parte dei server.

Individuare l'insieme delle operazioni permesse sulla risorsa: Arrivati a questo punto è possibile definire quali metodi facente parte dell'interfaccia uniforme sono applicabili alla risorsa. Ovviamente è da considerare il tipo di risorsa, non tutti i metodi, infatti, sono applicabili a tutte le risorse.

Arrivati a questo punto ci si potrebbe accorgere che non sono sufficienti i metodi definiti dall'interfaccia uniforme. Per questo problema vi sono due possibili soluzioni: si può creare una nuova risorsa che ci permetta di eseguire le nuove operazioni trovate, oppure si può utilizzare il metodo POST "sovraccaricandolo".

Per questo passo della progettazione potrebbe essere utile costruire una tabella in cui vi è una riga per ogni risorsa ed una colonna per ogni metodo (4 colonne: lettura, modifica, eliminazione e creazione). A questo punto si indica all'interno degli incroci se un dato metodo è applicabile alla risorsa.

Progettare le rappresentazioni: La rappresentazione delle risorse è il mezzo che fornisce al cliente la possibilità ottenere e modificare lo stato del servizio. Questa è una parte fondamentale nella comunicazione tra cliente e server e deve quindi essere progettata con attenzione.

Bisogna considerare che un servizio ReSTful è pensato per essere utilizzato da un applicazione e possibilmente dal maggior numero di client possibile. Questo porta a considerare formati che siano il più possibile condivisi e standardizzati, in modo che sia facilitata l'interpretazione e gestione da parte dei vari possibili client.

Quindi, in linea di massima, se esiste un formato di rappresentazione di risorse universalmente accettato e che può essere utilizzato per i nostri scopi, allora è bene adottarlo. Solitamente si utilizzano formati del tipo XML o JSON in quanto questi sono ampiamente supportati dalla maggior parte dei linguaggi di programmazione senza la necessita di creare parser ad-hoc.

Ovviamente, quando si parla di rappresentazioni, si intendono sia quelle in uscita, dal fornitore verso il cliente, sia quelle in ingresso, dal cliente verso il fornitore. Talvolta la progettazione di queste ultime può essere evitato inserendo le informazioni direttamente all'interno dell'URI, questo è possibile solamente se i dati sono di tipo semplice ed in un numero contenuto, in caso contrario è necessario costruire una rappresentazione strutturata. Solitamente per questo caso si utilizza la stessa rappresentazione utilizzata dal server per inviare dati al client in modo da rendere più lineare ed uniforme la comunicazione delle informazioni.

Individuare i collegamenti con altre risorse: Il progettista deve inserire all'interno delle rappresentazioni i link alle risorse correlate in modo da permettere all'utente di aggiornare anche il proprio stato.

I collegamenti presenti nelle rappresentazioni devono essere ragionati. Non ha senso fornire al client i link a tutte le risorse disponibili, questo provocherebbe molta confusione per il client che non verrebbe guidato nella transizione di stato.

Riprendendo l'esempio di uno shop online, la rappresentazione di un ordine dovrebbe contenere il collegamento al cliente e a tutti gli articoli che esso ha acquistato.

Progettare le risposte del servizio: Infine dopo aver realizzato e studiato tutte le risorse è possibile studiare tutte le possibili risposte che il server deve restituire per ogni situazione. Visto l'imposizione di utilizzo di HTTP il progettista ha a disposizione un insieme di codici che è possibile inserire nell'header del messaggio per comunicare la buona riuscita o meno della richiesta. Questi codici sono standardizzati e quindi è molto semplice per il client capire lo stato dell'esecuzione.

5.5 Servizi ReSTful: Best Practices

Durante la progettazione e l'utilizzo dei servizi ci si può imbattere in alcuni aspetti che non sono gestiti in modo diretto dall'architettura ROA, è necessario perciò ingegnarsi per risolvere questi problemi.

Nei servizi di questo tipo molte soluzioni si basano sull'aggiunta di nuove risorse (come ad esempio transazioni o operazioni asincrone), mentre altre si appoggiano a caratteristiche di HTTP (come ad esempio sicurezza ed autenticazione).

In questo paragrafo saranno analizzate alcune di queste problematiche e i metodi utilizzati comunemente per risolverle.

5.5.1 Operazioni asincrone

HTTP è intrinsecamente un protocollo sincrono, il client apre la connessione, effettua una richiesta ed attende che il server sia pronto ed invii una risposta. Non tutte le operazioni, però, possono essere eseguite con questo processo, alcune ad esempio potrebbero richiedere molto tempo da parte del server e il client potrebbe non essere disposto ad aspettare. In questi casi è necessario utilizzare operazioni asincrone. HTTP non gestisce questo tipo di iterazione è necessario perciò utilizzare uno stratagemma che ci permetta di utilizzarlo.

È necessario utilizzare due o più richieste sincrone. Come prima cosa deve essere richiesto al server l'esecuzione dell'operazione. Quest'ultimo risponderà indicando se la richiesta è stata accettata (stato 202 accepted) oppure respinta ed inserendo nel corpo del messaggio l'URI che rappresenta lo stato dell'operazione. In pratica il server crea una risorsa per ogni richiesta di esecuzione accettata dove inserirà il risultato. Il client in questo modo può manipolare la risorsa richiedendo in ogni istante se il risultato è pronto o è necessario altro tempo per infine eliminarla quando questa non sarà più utile.

L'unico metodo HTTP che rispetti l'interfaccia uniforme ed utilizzabile per richiedere l'esecuzione di un operazione asincrona è POST. Una richiesta di operazione asincrona infatti crea una nuova risorsa e non è né idempotente né sicura, perciò non è utilizzabile nessun altro metodo.

5.5.2 Transazioni

In alcuni contesti è necessario considerare più operazioni eseguite in sequenza come un unica azione evitando cosi perdite di dati o inconsistenze. In scenari come questi è necessario utilizzare il concetto di transazione (come già avviene nei database). I servzi ReSTful non possiedono intrinsecamente la capacità di gestire queste situazioni è perciò necessario gestirlo in qualche altra maniera. Come accade in altri casi simili si utilizza una risorsa per risolvere questo problema.

La nuova risorsa, che rappresenta la transazione, conterrà i dati facente parte dell'operazione e dovrà permettere all'utente di eseguire il comando di commit, sarà poi il server a gestire l'operazione. La risposta a questo tipo di richiesta dipenderà dall'esito della transazione e dalla rappresentazione che il server vuole fornire. A questo punto la risorsa collegata alla transazione può essere eliminata dal client o direttamente dal server una volta inviata la risposta.

È possibile chiarire i concetti sopra esposti attraverso un esempio [24]:

Immaginiamo di dover trasferire una somma di denaro, ad esempio 50€, da due conti. Il primo conto, da cui la somma deve essere prelevata, rappresentato dalla risorsa *http://ws.esempio.it/conto/1a* e contenente 200€. Il secondo conto, in cui la somma deve essere depositata, rappresentato dalla risorsa *http://ws.esempio.it/conto/1b* e contenente a sua volta 200€.

A questo punto è necessario creare la risorsa per la transazione utilizzando una richiesta POST

```
POST /transazione/trasferimentoMonetario HTTP/1.1
Host: ws.esempio.it
```

Il server risponderà comunicando che la risorsa è stata creata (stato 201 created) e indicando l'URI con la quale è possibile interagirvi.

Una volta creata la risorsa di transazione è necessario inserire i dati riguardanti i due conti all'interno di questa.

```
Inserimento conto beneficiario:
PUT /transactions/moneyTransfer/11a5
/ContiCorrenti/Beneficiario/1b HTTP/1.1

Host: example.com
balance=250

Inserimento conto donatore:
PUT /transactions/moneyTransfer/11a5
/ContiCorrenti/Donatore/1a HTTP/1.1

Host: example.com
balance=150
```

Fino a questo punto l'utente può eseguire il roll-back dell'operazione semplicemente cancellando la risorsa rappresentante la transazione, non modificando lo stato dei due conti.

Una volta deciso di confermare l'operazione si invia il comando di commit rendendo permanenti le modifiche.

```
PUT /transactions/moneyTransfer/11a5 HTTP/1.1
Host: example.com
committed=true
```

A questo punto il server si occupa di rendere sicura l'operazione senza avere controindicazioni.

5.5.3 Sicurezza e Autenticazione

I servizi web ReSTful hanno bisogno di appoggiarsi ad HTTP per amministrare l'autenticazione degli utenti e la crittografia dei messaggi [28]. HTTP gestisce l'autenticazione degli utenti attraverso l'header nel quale vengono inseriti diversi parametri a seconda del tipo di autenticazione utilizzato. Il client per capire come accedere alla risorsa deve inviare una prima richiesta a seguito della quale il server restituisce il tipo di scema utilizzato. HTTP fornisce due schemi standard per gestire l'autenticazione: Basic e Digest.

Il primo è più semplice e si basa sulla cifratura di username e password. Il client codifica queste due informazioni in un unica stringa con un codice a base 64 ed inserisce il risultato nell'header della richiesta all'interno dell'attributo *Authorization*. Il server decodifica la stringa con lo stesso metodo e controlla se corrisponde ad un utente autorizzato.

Digest fornisce un metodo di protezione più sofisticato. A seguito della prima richiesta il server fornisce tre ulteriori parametri in cui vi sono due codici generati in modo casuale. Il client genera a sua volta un codice casuale e decifra username e password utilizzando questo codice e quelli forniti dalla prima richiesta. Il server ricevute tutte queste informazioni riesce a decifrare con lo stesso metodo le credenziali e identifica l'utente.

Questi metodi impediscono agli utenti non autorizzati di accedere ad informazioni riservate, ma non proteggono queste informazioni nel caso in cui vi sia un intruso che intercetti la comunicazione client-server. Per prevenire questo tipo di attacco è necessario oscurare le informazioni interne ad HTTP. In questi casi solitamente si utilizza una cifratura SSL a chiave pubblica, ovvero il server mette a disposizione una chiave pubblica con il quale il client può cifrare la comunicazione in questo modo solo il server può comprendere il contenuto della richiesta.

Questi metodi si basano su una variazione di HTTP chiamato Hypertext Transfer Protocol Secure (HTTPS), esso è del tutto simile al suo genitore eccetto per la cifratura delle informazioni e per l'utilizzo della porta di comunicazione (utilizza la porta 443 invece di 80). HTTPS è utilizzabile nella maggior parte dei casi, ma può essere bloccato da alcuni firewall in quanto non utilizza la porta standard per la comunicazione internet.

5.6 Service Description: Servizi ReSTful

Un Service Description mette a disposizione del cliente tutte le informazioni necessarie all'utilizzo delle funzionalità fornite dal servizio. I servizi Resource-Oriented possiedono già una sorta di Service Description intrinseco: l'*interfaccia uniforme* essa descrive ogni operazione eseguibile sulle risorse appartenenti al servizio indicandone lo scopo. Ogni client inoltre può, attraverso l'operazione HTTP OPTION su una risorsa, sapere quali dei metodi inseriti nell'interfaccia uniforme di ROA sono permessi dal fornitore del servizio.

Per rendere questa operazione possibile, però, il cliente deve già conoscere l'URI della risorsa e non può ottenere informazioni aggiuntive (come le rappresentazioni che il server utilizza per le risorse o il numero e il tipo dei parametri necessari per ottenere certe rappresentazioni) che possono essere fondamentali per la corretta esecuzione di certe richieste.

Questo tipo di problemi sono facilmente risolvibili con l'introduzione di documenti Service Description (come avviene per i sistemi Service-Oriented), nati proprio con lo scopo di permettere il collegamento dinamico al servizio e fornire tutte le informazioni necessarie al client per costruire le proprie richieste.

L'utilizzo di questi documenti aiuta anche nella realizzazione delle applicazioni client. Esistono, infatti, degli strumenti in grado di generare applicazioni in modo automatico chiedendo come input un Service Description, questo facilita molto gli utenti del servizio che, possono creare un applicazione client del servizio in poco tempo e senza difficoltà.

Inizialmente i servizi ReSTful utilizzavano documenti human-readable per la definizione delle operazioni consentite. Questo precludeva la possibilità da parte dei tools di automatizzare la costruzione dei client direttamente dalla Service Description (come avviene per i servizi SOAP). Ultimamente, però, sono state sviluppate due tecnologie che permettono la costruzione di documenti più machine-readable: *WSDL 2.0 HTTP binding extension* e *WADL*.

5.6.1 Web Service Description Lenguage 2.0 HTTP binding extension

Vista la sempre più larga diffusione dei servizi web ReSTful nella nuova stesura delle specifiche WSDL si è deciso di inserire un estensione (HTTP binding extension) che permettesse ai fornitori di indicare quali dei metodi HTTP fossero applicabili su ognuna delle risorse messe a disposizione e quali parametri sia possibile inserire (facoltativamente o obbligatoriamente) all'interno della richiesta e in che modo questi ultimi debbano essere strutturati (nome, tipo, ecc.) [34].

Per poter utilizzare HTTP binding extension è necessario indicare che il documento intende utilizzare questa estensione, per fare ciò è sufficiente inserire la dichiarazione di namespace " xmlns:whttp="`http://www.w3.org/ns/wsdl/http`" ", questo permette al client di capire che tipo di documento deve aspettarsi e al fornitore di utilizzare un insieme di costrutti appositi per le operazioni HTTP.

La struttura di un file WSDL di questo tipo tuttavia non differisce molto da uno costruito per SOA, vi sono comunque alcuni concetti che sono stati inseriti con l'estensione e che è bene spiegare.

Come per i servizi SOA anche in questo caso il documento parte con il tag *types*, contenente la dichiarazione dei messaggi di input e output che verranno utilizzati durante la comunicazione tra client e server e dei tipi di dato utilizzati in essi.

I messaggi di input sono includono i parametri che possono essere inviati tramite query string e i payload utilizzabili nei metodi HTTP che lo permettono (POST e PUT).

I messaggi di output indicano la struttura delle rappresentazioni restituite dal server a seguito delle richieste.

La descrizione prosegue con la dichiarazione delle operazioni messe a disposizione dal servizio. Le operazioni vengono costruite attraverso un tag *operation* e raggruppate all'interno di una o più sezioni *interface*. Ognuna di queste sezioni verrà concretizzata con una risorsa ed ogni operazione, perciò, mappa un metodo HTTP. All'interno del tag *operation* vengono indicate: informazioni sul MEP utilizzato per la comunicazione, le proprietà dell'operazione (ovvero si indica se essa è safe ed idempotente) ed i messaggi

di input e output associati.

Infine vi è la parte di binding e dichiarazione del servizio.

Il binding delle operazioni è il cuore di questo documento esso collega le operazioni ai metodi HTTP. Ogni sezione di *binding* è associata ad un tag *interface* e contiene ognuna delle operazioni in esso contenute indicando quale metodo HTTP utilizza per richiederne l'esecuzione.

L'elemento finale è *service* esso contiene tanti tag *endpoint* quante sono le risorse rilevanti esposte dal servizio (ovviamente non è possibile indicare tutte le risorse. Si pensi all'esempio di uno shop online, ogni cliente costituisce una risorsa del servizio e possono essere in un numero molto elevato. Non ha senso indicare ognuno di essi nel documento di service description). Ogni *endpoint* collega una sezione *binding* con una risorsa specifica indicandone L'URI.

Esempio WSDL 2.0 HTTP binding extension È utile a questo punto mostrare un esempio che aiuterà a chiarire i concetti sopra esposti.

```
<?xml version="1.0" encoding="UTF-8"?>
<description xmlns="http://www.w3.org/ns/wsdl"
             xmlns:tns="http://www.tmsws.com/wsdl20sample"
             xmlns:whttp="http://schemas.xmlsoap.org/wsdl/http/"
             xmlns:wsdlx="http://www.w3.org/ns/wsdl-extensions"
             targetNamespace="http://www.tmsws.com/wsdl20sample">


<!-- Dichiarazione dei parametri -->

   <types>
      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                 xmlns="http://www.tmsws.com/wsdl20sample"
                 targetNamespace=
                   "http://www.example.com/wsdl20sample">

         <xs:element name="request">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="header"
```

```
                    maxOccurs="unbounded">
                    <xs:complexType>
                       <xs:simpleContent>
                          <xs:extension base="xs:string">
                             <xs:attribute name="name"
                               type="xs:string"
                               use="required"/>

                          </xs:extension>
                       </xs:simpleContent>
                    </xs:complexType>
                 </xs:element>
                 <xs:element name="body" type="xs:anyType"
                    minOccurs="0"/>

              </xs:sequence>
              <xs:attribute name="method" type="xs:string"
                 use="required"/>

              <xs:attribute name="uri" type="xs:anyURI"
                 use="required"/>

           </xs:complexType>
        </xs:element>

        <xs:element name="response">
           <xs:complexType>
              <xs:sequence>
                 <xs:element name="header" maxOccurs="unbounded">
                    <xs:complexType>
                       <xs:simpleContent>
                          <xs:extension base="xs:string">
                             <xs:attribute name="name"
                               use="required"/>

                          </xs:extension>
                       </xs:simpleContent>
```

```
                    </xs:complexType>
                </xs:element>
                <xs:element name="body" type="xs:anyType"
                   minOccurs="0"/>

            </xs:sequence>

            <xs:attribute name="status-code"
               type="xs:anySimpleType"
               use="required"/>

            <xs:attribute name="response-phrase"
               use="required"/>

         </xs:complexType>
      </xs:element>
   </xs:schema>
</types>


<interface name="RESTfulInterface">

<!-- Dichiarazione delle operazioni per la risorsa -->

   <operation name="Get"
   pattern="http://www.w3.org/ns/wsdl/in-out"
   wsdlx:safe="true">
      <input messageLabel="In" element="tns:request"/>
      <output messageLabel="Out" element="tns:response"/>
   </operation>
   <operation name="Post"
   pattern="http://www.w3.org/ns/wsdl/in-out">
      <input messageLabel="In" element="tns:request"/>
      <output messageLabel="Out" element="tns:response"/>
   </operation>
   <operation name="Put"
   pattern="http://www.w3.org/ns/wsdl/in-out"
```

```
      wsdlx:safe="true">
        <input messageLabel="In" element="tns:request"/>
        <output messageLabel="Out" element="tns:response"/>
      </operation>
      <operation name="Delete"
      pattern="http://www.w3.org/ns/wsdl/in-out"
      wsdlx:safe="true">
        <input messageLabel="In" element="tns:request"/>
        <output messageLabel="Out" element="tns:response"/>
      </operation>
   </interface>



<!-- Binding delle operazioni -->

   <binding name="RESTfulInterfaceHttpBinding"
     interface="tns:RESTfulInterface"
     type="http://www.w3.org/ns/wsdl/http">

      <operation ref="tns:Get" whttp:method="GET"/>
      <operation ref="tns:Post" whttp:method="POST"
        whttp:inputSerialization=
        "application/x-www-form-urlencoded"/>

      <operation ref="tns:Put" whttp:method="PUT"
        whttp:inputSerialization=
        "application/x-www-form-urlencoded"/>

      <operation ref="tns:Delete" whttp:method="DELETE"/>
   </binding>



   <service name="RESTfulService"
     interface="tns:RESTfulInterface">
```

```
<!-- Collegamento delle operazioni alla risorsa -->

      <endpoint name="RESTfulServiceHttpEndpoint"
                binding="tns:RESTfulInterfaceHttpBinding"
                address="http://www.example.com/rest/"/>
   </service>
</description>
```

5.6.2 Web Application Description Language

WADL è un formato XML-based che fornisce un metodo per descrivere, in modo machine-readable, applicazioni basate su HTTP, generalmente servizi ReSTful.

Esso modella le risorse messe a disposizione da un servizio e le relazioni esistenti tra esse, indicando per ognuna di queste risorse i metodi HTTP applicabili [35]. Come WSDL anche WADL è indipendente da linguaggio e piattaforma ed ha lo scopo di permettere il collegamento dinamico ai servizi e di promuoverne il riuso.

Un documento WADL ha una struttura gerarchica ben definita la cui radice è sempre *application*. Essa è utilizzata per la dichiarazione dei namespace, in particolare deve essere presente il riferimento a ***http://wadl.dev.java.net/wadl20061109.xsd*** che descrive tutti i componenti del documento.

All'interno della sezione *application* sono contenuti tutti gli elementi utilizzati dal documento:

- grammars;
- resource_type;
- resources;
- resource
- method;
- param;

- representation;

Grammars è utilizzato come contenitore delle definizioni dei dati scambiati durante la comunicazione client-server, queste sono descritte utilizzando linguaggi standard in modo da conservare la generalità (principalmente XML-Schema). Le definizioni possono essere inserite direttamente all'interno della sezione o importate da file esterni utilizzando l'elemento *include*, il quale, attraverso l'attributo href, indica il riferimento assoluto (URI) al documento desiderato.

Grammars assume esattamente lo stesso ruolo di types in WSDL, permette perciò agli utenti del servizio di capire il significato dei dati e come utilizzarli.

L'elemento *resources* divide le risorse, messe a disposizione dal servizio, in gruppi significativi per il client (ad esempio risorse che operano sullo stesso concetto). Questo elemento può essere presente più volte all'interno del documento, ma ognuno deve contenere risorse diverse, ovvero ogni risorsa può appartenere ad uno ed un solo gruppo. Esso possiede un solo attributo *base* che fornisce l'URI attraverso il quale possono essere costruiti tutti i percorsi assoluti alle risorse in esso contenute.

Ogni risorsa è individuata da un elemento *resource*, esso può contenere definizioni di parametri e metodi HTTP, oltre ad altre risorse che rappresentano le subordinate di quella considerata.

```
<resource id="..." path="..." type="...">
          <param.../>
          <method.../>
          <resource.../>
</resource>
```

L'elemento *param* inserito in *resource* è utilizzato per due motivi: fornire informazioni riguardo alla porzione di URI che verrà compilata a runtime (ad esempio considerando la risorsa cliente di uno shop online il codice cliente presente nell'URI sarà costruito a runtime) e specificare i parametri e le intestazioni che devono essere ereditati da ogni metodo HTTP utilizzato sulla risorsa.

Gli attributi *id*, *path* e *type* di *resource* assegnano rispettivamente un id univoco, un percorso relativo e un tipo alla specifica risorsa.

Il percorso, decritto da *path*, può essere di tipo statico o dinamico. Nel caso di percorso statico esso viene ricostruito unendo tutti i valori degli attributi *path* risalendo nella gerarchia delle risorse fino ad arrivare al percorso base indicato nell'elemento *resources*. Nel caso di percorso dinamico si segue esattamente lo stesso principio con la differenza che alcune parti di URI devono essere sostituite con valori conosciuti solo a runtime, queste porzioni vengono racchiuse tra parentesi graffe.

```
<resource path="cliente/{Id}">
        <param name="Id" style="template" type="xsd:int"/>
                ...
</resource>
```

L'attributo *type* contiene un riferimento ad un elemento *resource_type*.

Resource_type rappresenta l'interfaccia di una risorsa. Esso contiene un insieme di metodi che definiscono il comportamento di un certo tipo di risorse. Un determinato *resource_type* è applicabile a più risorse che da esso ereditano il comportamento.

```
<resource_type id="...">
        <method ... />
</resource>
```

Il cuore della descrizione di una risorsa è individuata dalle sezioni *method*, ognuna di esse descrive un metodo HTTP applicabile sulla risorsa stessa indicando quali sono e come sono strutturati i messaggi di richiesta e risposta.

```
<method name="..." id="...">
        <request.../>
        <response .../>
</method>
```

Ogni definizione di un metodo possiede due attributi: *name* e *id*. Il primo indica il metodo HTTP interessato, mentre il secondo assegna un nome univoco alla sezione. All'interno di una stessa risorsa è permesso avere più elementi *method* con lo stesso valore per l'attributo *name*, questo rappresen-

ta una variazione dello stesso metodo HTTP, che tipicamente ha un diverso tipo di parametri di ingresso.

Come si può vedere dall'esempio sopra citato, all'interno di un tag method sono presenti altri due elementi: *request* e *response*. Questi rappresentano rispettivamente i messaggi di ingresso e risposta per una richiesta di questo tipo.

L'elemento *request* è strutturato come segue:

```
<request>
        <representation ... />
        <param ... />
</request>
```

Come si può notare non possiede attributi ed è al suo interno vengono specificati parametri e rappresentazione da utilizzare durante la costruzione della richiesta. Ovviamente il tag *rappresentation* ha un significato solamente se inserito in metodi in cui è possibile inserire un payload (PUT e POST).

L'elemento *response* è strutturato come segue:

```
<rsponse status="...">
        <representation ... />
        <param ... />
</response>
```

Esso possiede un parametro: *status* il quale fornisce lo stato della particolare risposta HTTP. Gli elementi all'interno della sezione rappresentano rispettivamente lo stato attuale della risorsa (restituito sempre tramite rappresentazione) e l'header corrispondente alla risposta.

L'elemento *response* può apparire più volte ognuno rappresentante uno stato HTTP differente (successo, errore, ecc.).

La rappresentazione da inviare a o ricevere da una risorsa è individuata dal tag *rappresentation*.

```
<representation id="..." mediaType="..."
  element="..." profile"...">
        <param ... />
<representation>
```

Esso ha quattro attributi:

- *id*: assegna un nome univoco alla rappresentazione. Questo attributo è permesso solamente per le dichiarazioni a livello globale (ovvero inserite direttamente nella sezione radice).

- *mediaType:* identifica il tipo della rappresentazione. È nella forma tipo/valore dove "tipo" definisce la struttura e la semantica del "valore". È anche possibile indicare un valore generico per un determinato tipo utilizzando "*" (ad esempio text/* identifica una rappresentazione di tipo testo in nessuno specifico formato). Solitamente *mediaType* assume il valore "text/xml".

- *element:* utilizzato per rappresentazioni in formato XML. Specifica il nome completo dell'elemento root di XML-Schema nel quale sono descritti i dati utilizzati dalla rappresentazione.

- *profile:* fornisce informazioni riguardo ai link presenti nella rappresentazione.

È possibile parametrizzare la rappresentazione utilizzando l'elemento *param*. Esso può assumere due significati a seconda del valore dell'attributo *mediaType*. Nel caso in cui la rappresentazione sia di tipo XML questo elemento fornisce un suggerimento riguardo le voci interne alla rappresentazione stessa. In caso contrario definisce i dati utilizzati nella rappresentazione che saranno poi formattati a seconda del valore di *mediaType*.

Param, in generale, è utilizzato per descrivere i componenti parametrizzati appartenenti all'elemento che lo contiene.

```
<param id="..." name="..." style="..." type="..."
  default="..." path="..." required=true
  repeating=true >
        <option .../>
        <link .../>
</param>
```

Gli attributi di *param* sono:

- *id:* identifica univocamente un parametro all'interno del documento. È utilizzato per i riferimenti attraverso URI;

- *name:* nome del parametro (attributo obbligatorio);
- *type:* indica il tipo del parametro (intero, stringa, ecc.). Di default assume il valore "string";
- *default:* fornisce il valore assunto di default dal parametro;
- *path:* quando il parametro si trova all'interno di una rappresentazione fornisce il percorso per ottenerlo;
- *required:* indica se il parametro deve essere obbligatoriamente presente o meno;
- *repeating:* indica se il parametro ha valore singolo oppure può avere valori multipli;
- *style:* fornisce lo stile del parametro. Questo attributo può solamente valori appartenenti ad un insieme ristretto, riportato nella tabella sottostante

Valore	Elementi padre	Descrizione
header	resource, resource type, request, response	descrive la strutture dell'header HTTP
query	resource, resource type, request	specifica un parametro passato tramite query string
query	representation	descrive un parametro della rappresentazione
template	resource	utilizzato per la creazione di URI dinamici

Come si vede nell'esempio *param* può avere due tipologie di elementi figli: *option* e *link*.

L'elemento *option* può comparire zero o più volte all'interno di un parametro. Esso specifica uno dei possibili valori che il genitore può assumere.

```
<option value="..." mediaType="..."/>
```

I suoi attributi, *value* e *mediaType*, indicano rispettivamente il valore che il parametro può assumere e il valore di meta-tipo che il parametro assume quando quest'ultimo assume il ruolo di selettore di tipo di dato. Il secondo attributo è opzionale, ma quando è presente il parametro ha un ruolo ben

preciso (ad esempio può essere usato per selezionare una rappresentazione XML invece che in un altro formato).

L'ultimo elemento facente parte di un documento WADL e *link*. Esso è opzionale e può comparire solamente all'interno di un parametro di tipo URI in cui l'attributo *path* identifica la porzione della rappresentazione che contiene un link.

Link viene utilizzato per realizzare collegamenti a risorse nelle rappresentazioni.

```
<link resource\_type="..." rel="..." rev="..."/>
```

Questo elemento ha tre attributi opzionali:

- *resource_type:* contiene un riferimento ad un istanza dell'omonimo elemento. Definisce il tipo di risorsa referenziata dal link;
- *rel:* identifica la relazione che ha la risorsa referenziata dal link con la risorsa la cui rappresentazione contiene il link;
- *rev:* identifica la relazione che ha la risorsa la cui rappresentazione contiene il link con la risorsa referenziata dal link. Questo attributo rappresenta il duale di *rel*.

Nei documenti WADL è possibile utilizzare il riferimento ad una istanza di un elemento in modo da poter riutilizzare un'istanza più volte o

Esempio documento WADL A questo punto è possibile rendere un po più chiari i concetti definiti sopra utilizzando un esempio che li concretizzi.

```
<?xml version="1.0"?>

<---Dichiarazione namespace--->
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://wadl.dev.java.net/2009/02 wadl.xsd"
xmlns:tns="urn:yahoo:yn"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:yn="urn:yahoo:yn"
xmlns:ya="urn:yahoo:api"
xmlns="http://wadl.dev.java.net/2009/02">
```

```
<——Definizione dei dati utilizzati per riferimento——>
  <grammars>
    <include href="NewsSearchResponse.xsd"/>
      <include href="Error.xsd"/>
  </grammars>
  <resources
    base="http://api.search.yahoo.com/NewsSearchService/V1/">

<——Definizione di risorse e metodi——>

    <resource path="newsSearch">
      <method name="GET" id="search">
        <request>
          <param name="appid"
            type="xsd:string"
            style="query"
            required="true"/>

                  <param name="query"
                    type="xsd:string"
                    style="query"
                    required="true"/>

                  <param name="type"
                    style="query"
                        default="all">

                    <option
                          value="all"/>

                        <option
                          value="any"/>

                        <option
                          value="phrase"/>
```

```
  </param>
  <param name="results"
    style="query"
        type="xsd:int"
        default="10"/>

  <param name="start"
        style="query"
        type="xsd:int"
        default="1"/>

  <param name="sort"
        style="query"
        default="rank">

        <option
          value="rank"/>

        <option
          value="date"/>

  </param>

  <param name="language"
        style="query"
        type="xsd:string"/>

</request>

<response status="200">
  <representation
    mediaType="application/xml"
        element="yn:ResultSet"/>

  </response>
  <response status="400">
        <representation
```

```
                        mediaType="application/xml"
                        element="ya:Error"/>

                </response>

            </method>
        </resource>
    </resources>
</application>
```

5.6.3 WSDL 2.0 vs WADL

In conclusione entrambi i metodi sopra descritti costituiscono un ottima soluzione per costruire una descrizione machine-processable per i servizi web ReSTful. Esistono, comunque, alcune differenze che possono portare alla scelta di un tipo rispetto all'altro [29].

WADL è stato pensato per l'utilizzo esclusivo con applicazioni ReST HTTP-based. La sua struttura rispecchia questa volontà descrivendo ogni applicazione come un insieme di risorse alle quali viene assegnato un URI e sulle quali possono essere applicati metodi HTTP. È per questo che viene sempre associato ai servizi ReSTful, il suo orientamento verso HTTP lo rende estremamente adatto per l'utilizzo nella descrizione di questo tipo di servizi, semplificando, così, la costruzione di una Service Description.

WSDL 2.0 è costruito per fornire un'interfaccia a più tipologie di applicazioni che comunicano attraverso il web, queste non è detto che utilizzino HTTP come protocollo di trasporto e occorre, quindi, definire un metodo il più generale possibile. Questo complica la costruzione e la comprensione di una Service Description per i servizi ReSTful, la quale è costruita come un insieme di interfacce e operazioni che vengono poi collegate al metodo HTTP corrispondente.

Una seconda differenza tra i due, oltre all'idea di base, riguarda la gestione di alcuni aspetti rilevanti della comunicazione (come ad esempio sicurezza e QoS). WSDL fornisce un insieme di meccanismi per descrivere come risolvere queste problematiche in modo che l'utente possa immediatamente

capire come sono gestite dal fornitore e comportarsi di conseguenza. WADL non si preoccupa di questi dettagli lasciandone la gestione direttamente al fornitore che si appoggerà sugli standard di HTTP.

La scelta tra i due si deve, quindi, basare su più considerazioni: WADL è più orientato verso applicazioni che seguono i principi ReST ma lascia all'utente la gestione di alcuni aspetti che potrebbero risultare importanti, al contrario di WSDL che risulta più complicato ma fornisce una descrizione più completa.

Capitolo 6

Framework per servizi web ReSTful

Con il passare del tempo sempre l'interesse verso i servizi ReSTful è cresciuto così come il numero di organizzazioni che ne sfruttano le potenzialità. In questo scenario molte piattaforme hanno cercato di rendere il più semplice possibile lo sviluppo di questi servizi. I framework, infatti, impongono di per sé alcuni vincoli automatizzando il più possibile questa fase del progetto e permettendo al progettista di concentrarsi sulla logica applicativa del servizio senza pensare ai dettagli relativi alla comunicazione con il client.

Generalmente tutti i framework per la creazione di servizi web ReSTful mettono a disposizione alcune caratteristiche a supporto degli utenti [4]:

- **Associazione di URI alle risorse o URI routing:** il framework deve consentire di associare uno o più URI ad una risorsa.
- **Gestione della comunicazione HTTP:** è il framework ad occuparsi della comunicazione tra client e server, in questo modo non sarà necessario analizzare ogni richiesta HTTP, mappare la richiesta su un'operazione e creare una risposta manualmente, sarà sufficiente specificare il codice da eseguire in corrispondenza a ciascun metodo HTTP per ogni risorsa.
- **Semplificazione della rappresentazione delle risorse:** la maggior parte dei framework aiuta nella creazione della rappresentazione delle risorse, fornendo supporto per i principali formati, evitando così di dover effettuare questa operazione manualmente.

- **Funzionalità di supporto:** alcuni framework forniscono supporto per funzionalità aggiuntive come autenticazione, gestione degli errori, gestione della cache.

Esistono framework per diverse piattaforme di sviluppo ognuna con le proprie caratteristiche, in questo capitolo verranno analizzate alcuni dei più rilevanti e diffusi nel mondo dei servizi web ReSTful.

6.1 Windows Communication Foundation

Windows Communication Foundation (WCF) è un sottosistema di .NET Framework che offre un insieme di API per la gestione di sistemi distribuiti [14]. Esso permette di creare applicazioni sia client che server per qualsiasi sistema network-based, in questa sezione, comunque, ci si concentrerà sulla parte relativa ai servizi web ReSTful.

WCF ha la sua base sul concetto di *endpoint*. Un endpoint gestisce tutti gli aspetti relativi alla comunicazione e alla gestione di richieste e risposte.

Ogni endpoint è composto da tre parti: un indirizzo, un oggetto binding ed un contratto.

L'indirizzo rappresenta l'URI sul quale l'endpoint opera, ovvero l'indirizzo corrispondente a client o server (a seconda del ruolo che assumerà l'endpoint).

L'oggetto binding concretizza lo stack di comunicazione, cioè l'insieme degli oggetti che si occuperà della gestione completa dei messaggi in entrambe le direzioni (ricezione/invio, decodifica/codifica, instradamento verso la routine che li utilizzerà per i propri scopi).

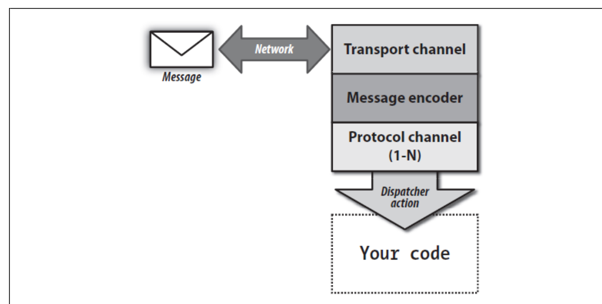

Figura 6.1: WCF comunication stack [13]

Come si può notare dall'immagine 6.1 un messaggio deve passare attraverso quattro step ognuno dei quali fondamentali per l'elaborazione dello stesso:

- *transport channel:* è l'elemento che rimane in ascolto sul canale di comunicazione. Esso può svolgere un ruolo passivo, rimanendo semplicemente in attesa, o un ruolo attivo, andando alla ricerca dei messaggi;
- *message encoder:* ottiene il messaggio di rete dal livello precedente e traduce questo in un oggetto che il resto del sistema riuscirà a comprendere.
- *protocol channels:* sono elementi opzionali che implementano alcuni protocolli che possono essere utili al servizio (come ad esempio riguardanti sicurezza o affidabilità della comunicazione);
- *dispatcher:* è il responsabile dell'instradamento del messaggio all'interno dell'applicazione (ad esempio un servizio avrà più metodi che gestiranno diverse richieste, il dispatcher si occupa di associare il giusto metodo alle richieste eseguendo il compito desiderato dal client).

Insieme questi elementi vengono chiamati *channel stack*. Questa struttura non è legata a nessuna particolare tecnologia ed è per questo che WCF supporta una vasta gamma di meccanismi di comunicazione.

Infine il contratto rappresenta l'insieme delle richieste che il server è in grado di soddisfare, e conseguentemente che il client può effettuare. É

considerato contratto una qualunque classe o interfaccia .NET a cui viene associato l'attributo *ServiceContract*. In pratica il contratto è un insieme di metodi ognuno corrispondente ad una possibile richiesta.

Ora la domanda sorge spontanea, come può il dispatcher ad instradare i messaggi alla giusta voce nel ServiceContract? Questa operazione è possibile grazie al meccanismo degli attributi, ad ogni metodo presente nel contratto vengono associati uno o più attributi che indicano al dispatcher a quale tipo di richiesta esso corrisponde (ad esempio per una richiesta HTTP viene associato ad ogni metodi l'URI relativo).

Con l'uscita della versione 3.5 del .NET Framework è stata introdotta una nuova libreria che semplifica molto la costruzione di servizi ReSTful, permettendo al progettista di non dover scrivere una significativa quantità di codice.

Innanzitutto questa libreria modifica il dispatcher in modo che quest'ultimo indirizzi le richieste a seconda dell'URI (relativo alle risorse) in esse contenute. Questo nuovo comportamento è possibile grazie all'aggiunta di due nuovi attributi: *WebGetAttribute* utilizzato per richieste HTTP GET e *WebInvokeAttribute* utilizzato in tutti gli altri casi. Essi vengono applicati a ciascun metodo appartenente al contratto e possiedono un insieme di proprietà che è possibile impostare per indicare al dispatcher come instradare i messaggi. Tra le proprietà le più rilevanti sono:

- UriTemplate: indica l'URI a cui viene associato il metodo. Esso corrisponde all'indirizzo di una risorsa resa disponibile dal sistema. All'interno della proprietà possono essere presenti porzioni di URI a valore fisso e porzioni a valore variabile, queste ultime sono racchiuse tra parentesi graffe e assumono un valore solamente a runtime nel momento in cui viene effettuata la richiesta.

- Method: indica il metodo HTTP che viene eseguito. Esso è associato solamente all'attributo *WebInvoke*, in quanto *WebGet* è utilizzato solamente per richieste di tipo HTTP GET.

- RequestFormat e ResponseFormat: indicano rispettivamente il formato accettato in ingresso e restituito in uscita.

Oltre all'introduzione di un nuovo metodo di dispatching e di costruzione dei ServiceContract, basato esclusivamente su ReST, la nuova versione

di WCF semplifica, anche, la costruzione dell'oggetto binding grazie all'inserimento della nuova classe *WebHttpBinding*, essa sgrava l'utente da dover costruire manualmente il *channel stack* per HTTP automatizzando questa operazione.

Infine viene introdotta la classe *WebOperationContext*. Un oggetto di questo tipo può essere utilizzato nei metodi del contratto per ottenere informazioni riguardo la richiesta in entrata (tra cui il metodo HTTP utilizzato, l'URI a cui è diretta, ecc) ed agganciare informazioni alla risposta, oltre ad inserire le intestazioni necessarie per gestire la sicurezza tramite HTTP.

```
//Ottengo il contesto corrente
WebOperationContext webCtx;
webCtx = WebOperationContext.Current;
```

6.1.1 WCF e rappresentazioni

WCF fornisce un insieme di classi che permettono di mappare ogni elemento di una rappresentazione in un determinato formato in variabili .NET e viceversa, in modo da permettere alle applicazioni di leggere e creare a loro volta una propria rappresentazione.

Solitamente durante l'interazione tra cliente e servizio vengo scambiate rappresentazioni di risorse in formato XML. .NET fornisce diverse classi per la gestione di questo formato, ma solo *XmlSerializer* permette un controllo completo su tutti gli aspetti di un documento XML. Un oggetto *XmlSerializer* effettua le operazioni di codifica e decodifica in modo automatico, l'unica cosa di cui si deve occupare un utente è costruire le strutture dati in cui verranno inseriti i dati decodificati da un documento XML e da cui verranno prelevati i dati da codificare in XML. Le strutture utilizzate da *XmlSerializer* non sono altro che semplici classi contenente dichiarazioni di variabili, ciò che le differenziano dalle comuni classi è l'utilizzo di attributi che indicano il ruolo che ciascun elemento ha all'interno della rappresentazione XML. E' possibile, inoltre, inserire la dichiarazione di un namespace in modo da far comprendere a chiunque il significato degli elementi presenti nel documento.

```
//dichiarazione elemento radice
//viene indicato il nome dell'elemento
```

```
//e il namespace a cui appartengono i sotto-elementi.
[XmlRoot(Namespace="",ElementName="Esempio")]
public class Esempio
{
//dichiarazione attributo XML
[XmlAttribute(AttributeName="name")]
public string Name;
//dichiarazione di un elemento semplice XML
[XmlElement(ElementName = "tipo")]
public string tipo;
}
```

Per poter utilizzare *XmlSerializer* all'interno di un servizio ReSTful è necessario comunicare le proprie intenzioni, per far ciò si inserisce l'attributo *[XmlSerializerFormat()]* nel ServiceContract.

6.1.2 WCF lato server

La progettazione di un servizio web ReSTful in WCF ha inizio con l'inizializzazione degli elementi che rimarranno in attesa delle richieste da parte dei client. Innanzitutto è necessario dichiarare un endpoint, questo passaggio è automatizzato dall'utilizzo di un oggetto di tipo *WebServiceHost*. Un host rappresenta un insieme di uno o più endpoint ed ha il compito di inizializzare e rendere disponibile sulla rete un servizio. Per svolgere questo compito l'host ha bisogno di un oggetto binding, di un contratto e dell'indirizzo su cui verrà esposto il servizio.

```
//creazione dell'oggetto binding
WebHttpBinding binding = new WebHttpBinding();
//creazione service host e aggiunta endpoint
WebServiceHost sh = new WebServiceHost(typeof(Hello_Service));
sh.AddServiceEndpoint(typeof(Hello_Service), binding,
 "http://localhost:3000/prova");

//inizio ascolto del server
sh.Open();
```

Una volta preparato il server alla ricezione delle richieste è necessario costruire i ServiceContract.

Come spiegato precedentemente il contratto può essere una qualsiasi classe o interfaccia in cui è indicato l'attributo *[ServiceContract]*. I metodi dichiarati in un contratto, comunque, non possono avere un intestazione qualunque, essi devono seguire alcune regole per permettere al dispatcher il giusto utilizzo. I parametri d'ingresso devono essere strutturati nel seguente modo: i primi sono di tipo string e conterranno ordinatamente le parte variabili dell'URI e i valori passati attraverso queryString, l'ultimo parametro è opzionale e conterrà l'eventuale rappresentazione inserita nel body della richiesta. Il valore di ritorno corrisponde alla rappresentazione che il server deve trasferire al client (se il server non ha necessità di comunicare alcuna informazione aggiuntiva al client il ritorno sarà di tipo void).

Ogni metodo inserito nel contratto poi dovrà riportare gli attributi necessari (*[WebGet]* o *[WebInvoke]* ed eventualmente *[XmlSerializerFormat()]*) indicando l'URI della risorsa su cui opera e l'eventuale verbo HTTP.

Una importante proprietà utilizzabile da server all'interno dei metodi del contratto è portata dalla classe *WebOperationContext*. Essa da la possibilità al progettista di ottenere un oggetto di tipo *UriTemplate*. Questo oggetto fornisce tutte le informazioni riguardante l'URI contenuto nella richiesta (come ad esempio un'eventuale query string o l'URI della richiesta appena ricevuta).

```
WebOperationContext ctx ;
ctx=WebOperationContext.Current ;
IncomingWebRequestContext Ictx ;
Ictx=ctx.IncomingRequest ;
//recupero dell'URI associato alla richiesta
string uri;
uri = webCtx.IncomingRequest
.UriTemplateMatch.RequestUri.ToString();

//recupero della lista dei valori della query string
NameValueCollection query;
query = webCtx.IncomingRequest
.UriTemplateMatch.QueryParameters;
```

UriTemplate è utilizzato anche dal dispatcher per effettuare il matching tra l'URI a cui è indirizzata la richiesta ed gli attributi assegnati a ciascun metodo del contratto.

Infine è necessario costruire le strutture atte a contenere gli elementi delle rappresentazioni, queste dovranno essere uguali sia lato client che lato server e dovranno descrivere appunto le rappresentazioni in ingresso ed in uscita.

6.1.3 WCF lato client

E' possibile utilizzare windows comunication foundation anche per progettare client per servizi web ReSTful. Una delle più importanti proprietà di WCF è che il modello di programmazione lato client è esattamente simmetrico rispetto al lato server[13]. Il progetto di un client ReSTful WCF inizia dalla costruzione dei ServiceContract e delle strutture corrispondenti alle rappresentazioni. Questi due costrutti sono praticamente identici a quelli già visti per la progettazione del servizio con l'unica differenza che i contratti sono obbligatoriamente costituiti da interfacce software, questo perché i metodi presenti nel contratto non devono avere alcun comportamento particolare, devono solo dare un indicazione sul tipo di richiesta che il client intende eseguire.

Successivamente si può procedere alla costruzione della parte che si occuperà dell'invio delle richieste e della ricezione delle risposte. Come lato server vi era WebServiceHost per la creazione di endpoint che rimanevano in attesa di richieste, lato client vi è *WebChannelFactory<TChannel>* (dove *TChannel* è il tipo di canale da creare, in questo caso sarà sostituito dal nome di un contratto), questo costrutto vuole in ingresso un oggetto binding (in questo caso WebHttpBinding) e l'URI del servizio con cui il client intende comunicare, ed è utilizzato per costruire un canale tra client e server. Questo canale eredita tutti i metodi dal contratto a cui e associato l'oggetto WebChannelFactory e rende possibile la comunicazione attraverso una semplice chiamata al metodo del ServiceContract relativo alla richiesta desiderata.

6.1.4 Esempio Client/Server WCF

L'esempio considerato è costituito da un semplice sistema client/server. L'utente invia una richiesta HTTP GET ad una risorsa "hello" indicando nella query string il proprio nome.

Il server risponde con una rappresentazione testuale contenente la frase "Ciao #nome".

Server:

Inizializzazione dei componenti server:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.ServiceModel.Web;
using System.Text;
using Prova_Dati;

namespace Prova_Server
{
    public class prova
    {
        static void Main(string[] args)
        {
            //creazione dell'oggetto binding
            WebHttpBinding binding = new WebHttpBinding();
            //creazione service host e aggiunta endpoint
            WebServiceHost sh;
            sh=new WebServiceHost(typeof(Hello_Service));
            sh.AddServiceEndpoint(typeof(Hello_Service), binding,
            "http://localhost:3000/prova");

            //inizio ascolto del server
            sh.Open();
            Console.WriteLine("Servizio attivo");
            Console.ReadLine();
        }
    }
}
```

creazione del ServiceContract:

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ServiceModel;
using System.ServiceModel.Web;
using System.Collections.Specialized;
using Prova_Dati;

namespace Prova_Server
{
    [ServiceContract]
    public class Hello_Service
    {
        static string nome;
        //Operazione GET: restituisce il messaggio 'Hello #nome'
        [OperationContract()]
        [WebGet(UriTemplate="/hello?name={name}")]
        [XmlSerializerFormat()]
        public message hello_user(string name){
            message ret=new message();
            ret.Hello_message="Hello "+name;
            return ret;
        }
        //Operazione PUT: salva il nome dell'utente
        [OperationContract()]
        [WebInvoke(UriTemplate = "/hello", Method = "PUT")]
        [XmlSerializerFormat()]
        public string setName(name n)
        {
            nome = n.nome;
            return nome;
        }
    }
}
```

Client:

inizializzazione client:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ServiceModel.Web;
using System.ServiceModel;
using Prova_Dati;

namespace Prova_Client
{
    public class prova
    {
        static void Main(string[] args)
        {
                //creazione oggetto binding
            WebHttpBinding binding = new WebHttpBinding();
            //creazione ed inizializzazione ascoltatori
            WebChannelFactory<Icontract> cf;

            cf = new WebChannelFactory<Icontract>(binding,
            new Uri("http://localhost:3000/prova/"));

            message mess = new message();
            //creazione canale di comunicazione
            Icontract channel = cf.CreateChannel();
            //invio richiesta e ricezione risposta
            mess=channel.hello_user("Elia");
            IncomingWebResponseContext incResp;
            incResp = WebOperationContext
            .Current.IncomingResponse;

            Console.WriteLine("Stato: {0} {1}",
            incResp.StatusCode, incResp.StatusDescription);

        }
    }
}
```

Definizione ServiceContract lato client:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ServiceModel;
using System.ServiceModel.Web;
using System.Collections.Specialized;
using Prova_Dati;

namespace Prova_Client
{
    [ServiceContract]
    public interface Icontract
    {
        //metodo GET
        [WebGet(UriTemplate = "/hello?name={name}")]
        [OperationContract]
        [XmlSerializerFormat()]
        message hello_user(string name);

        //metodo PUT
        [WebInvoke(UriTemplate = "/hello", Method = "PUT")]
        [OperationContract]
        [XmlSerializerFormat()]
        string setName(name n);
    }

}
```

creazione delle strutture dati:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ServiceModel;
using System.Xml.Serialization;
```

```
namespace Prova_Dati
{
    [XmlRoot(Namespace = "", ElementName = "message")]
    public class message
    {
        [XmlElement(ElementName="Hello_message")]
        public string Hello_message;
    }



    [XmlRoot(Namespace = "", ElementName = "name")]
     public class name
     {
        [XmlElement(ElementName = "nome")]
        public string nome;
     }
}
```

6.2 ReSTlet

ReSTlet è un framework leggero è completo per la costruzione di sistemi ReSTful in Java (non è stato progettato solamente per servizi web, ma per qualsiasi sistema che segua i principi ReST) [24].

La filosofia di ReSTlet non da importanza alla distinzione tra client e server. Il framework è identico sia dal lato server che dal lato client e prevede un insieme di strumenti per la gestione di sicurezza, rappresentazioni e per l'esecuzione in diversi ambienti.

ReSTlet mappa tutti i concetti espressi nei principi ReST all'interno di classi java in modo da rendere il più semplice possibile la costruzione di questi sistemi. Il centro del framework è costituito dall'interfaccia *Uniform* e dalla classe *Restlet* che implementa questa interfaccia. *Uniform*, come si evince dal nome, rappresenta il concetto di interfaccia uniforme così come descritto da ReST, essa fornisce un metodo handle il quale accetta due argomenti: Request e Response. Ogni protocollo che ReSTlet supporta viene

esposto tramite il metodo handle. La classe *Restlet*, esplicita tutte le caratteristiche che devono possedere i componenti di un sistema costruito con questo framework [25].

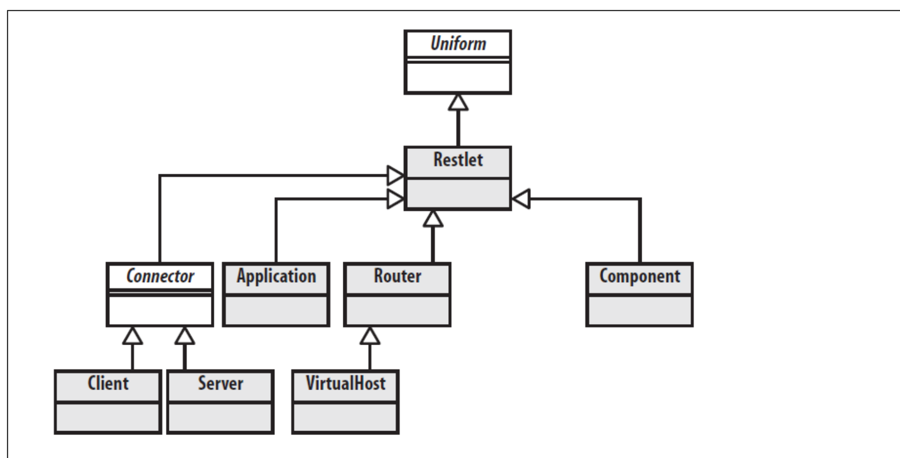

Figura 6.2: Struttura sistema ReSTlet [25]

A questo punto è possibile analizzare i veri e propri componenti che svolgono un ruolo attivo durante l'utilizzo di un servizio.

Così come descritto da Fielding all'interno della sua tesi [12] la comunicazione viene gestita da connectors concretizzati da ReSTlet con un'omonima classe, la quale implementa i vari protocolli di rete [25]. La classe *Connector* è estesa da due sottoclassi: *Client* e *Server*, esse svolgono, appunto, il ruolo di cliente e fornitore del servizio inizializzando la comunicazione e scambiandosi messaggi di richiesta e risposta.

Lato client l'invio delle richieste e la ricezione delle risposte per un dato protocollo di rete viene gestita interamente dal connector, per il quale il programmatore deve costruire un'instanza della classe *Request* in cui viene indicato l'URI della risorsa "target", il metodo che si desidera applicare e gli eventuali parametri di autenticazione. Attraverso l'oggetto di tipo *Request* è possibile indicare al server quale formato di rappresentazione si vorrebbe ricevere, per far ciò si deve costruire un istanza della classe *ClientInfo* in cui va inserito nel costruttore il MediaType desiderato. Infine attraverso il metodo *handle* viene inviata la richiesta al server.

```
//costruzione richiesta
Request request=new Request(Method.GET,
"http://localhost:3000/prova/hello?nome=Elia");
//impostazione MediaType
```

```
request.setClientInfo(new ClientInfo(MediaType.TEXT_XML));
//invio richiesta e ricezione risposta.
Response response=new Client(Protocol.HTTP).handle(request);
```

Lato server la situazione è un po più complicata. Innanzi tutto è necessario creare un *Connector* server in attesa all'indirizzo e sulla porta desiderata, questo riceverà tutte le chiamate per un dato protocollo di rete e lo inoltrerà al giusto *VirtualHost*.

```
Server server=new Server(Protocol.HTTP,"localhost", 3000);
```

Ogni istanza della classe *VirtualHost* rappresenta un ascoltatore per un determinato pattern URI e trasferisce la richiesta all'oggetto che gestisce il routing verso le risorse vere e proprie. Le informazioni necessarie a *VirtualHost* per l'esecuzione del suo compito vengono fornite attraverso il metodo *attach(String URI, Restlet target)*, dove 'URI' indica il pattern URI e 'target' indica l'istanza della classe che gestirà l'indirizzamento, quest'ultima solitamente è di tipo *Application*.

```
VirtualHost vh=new VirtualHost();
vh.attach("\esempio",new Application());
```

Ogni istanza della classe di tipo *Application* può svolgere un doppio compito: gestire una serie di servizi utili durante l'esecuzione e associare le richieste alle risorse corrispondenti. Per il primo compito vengono forniti un insieme di metodi attraverso i quali è possibile indicare i servizi che si vogliono utilizzare ed impostarli nel modo più adeguato, questi servizi possono variare da una compressione/decompressione dei messaggi ad una conversione tra due tipi di rappresentazione. Il secondo viene svolto dal metodo *createInboundRoot()* il quale attraverso la classe *Router* indica per ogni tipo di risorsa il percorso relativo.

Router ha il compito di indirizzare le singole richieste alla risorsa destinataria, creando un istanza di questa risorsa per ogni richiesta ricevuta.

```
public class ProvaServerRestlet extends Application {
        .
        .
        .
        @Override
        public Restlet createInboundRoot(){
```

```
                Router router=new Router(getContext());
            router.attach("/resource", UserResource.class);
            return router;
        }
}
```

Tutti questi componenti vengono "impacchettati" all'interno di un oggetto *Component*. Questo, oltre a contenere tutte le informazioni del sistema, effettua l'avvio dei server connector e dei virtual host permettendo così al sistema di ricevere le richieste.

```
Component component=new Component();
component.getServers().add(server);
component.getHosts().add(vh);
comp.start();
```

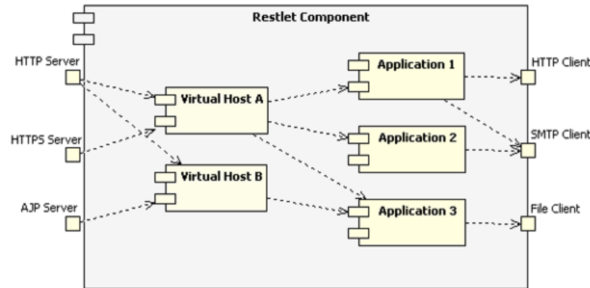

Figura 6.3: Struttura sistema ReSTlet

L'applicazione server, oltre a rimanere in ascolto per le richieste, deve, ovviamente, costruire le singole risorse. Esse vengono concretizzate da classi appositamente definite dall'utente le quali devono estendere la classe *serverResource*, quest'ultima rappresenta il prototipo della risorsa e fornisce un insieme di metodi per aiutare l'utente a gestire le richieste.

In particolare all'interno della risorsa sarà necessario recuperare i parametri passati dal client attraverso URI (questi sono conservati all'interno della richiesta e sono ottenibili con diverse procedure a seconda del tipo di parametri) e smistare le diverse tipologie di richiesta. Vi sono tre modalità con cui è possibile svolgere quest'ultimo compito:

- utilizzando i metodi predefiniti (get(), put(), post(), delete()), essi si comportano esattamente come i metodi HTTP;

- ridefinendo i metodi già presenti ed indicando il comportamento desiderato;

- definendo un nuovo insieme di metodi. Per quest'ultima è necessario utilizzare il meccanismo delle annotation descritto nelle specifiche JSR-311 ed inserito nelle API ReSTlet con le ultime versioni (ad esempio indicando la dicitura get prima della definizione di un metodo che gestische chiamate HTTP GET).

Infine si dovrà costruire la rappresentazione nel caso la richiesta implichi il ritorno della stessa. Una rappresentazione può essere restituita in diversi formati, il server deve gestire ognuno di questi formati e costruire la giusta

rappresentazione. Lato server un progettista è costretto a gestire manualmente la scelta tra vari formati e può recuperare il formato richiesto dal client attraverso un oggetto di classe *ClientInfo*, questo contiene diverse informazioni riguardo il client tra cui il formato desiderato.

```
MediaType tipo;
List lista=new ArrayList();
//lista dei MediaType tra cui
//deve essere presente quello
//richiesto dal client.
//Questa lista può essere utilizzata
//per filtrare i MadiaType supportati
//dal server.
lista.add(MediaType.TEXT_XML);
//Recupero MediaType dal client
tipo=this.getRequest().getClientInfo()
.getPreferredMediaType(lista);
```

Oltre a dare la possibilità di costruire applicazioni ReSTful client/server *ReSTlet è uno dei pochi framework che fornisce il supporto per i documenti WADL.* Esso utilizza un estensione grazie alla quale è possibile configurare i componenti di un applicazione e creare un documento WADL da un applicazione esistente [27].

6.2.1 Esempio Client/Server ReSTlet

L'esempio considerato è costituito da un semplice sistema client/server. L'utente invia una richiesta HTTP GET ad una risorsa "hello" indicando nella query string il proprio nome.

Il server risponde con una rappresentazione testuale contenente la frase "Ciao #nome".

Inizializzazione ascoltatori server:

```
package prova.restlet;

import org.restlet.Application;
import org.restlet.Client;
import org.restlet.Component;
```

```
import org.restlet.Restlet;
import org.restlet.routing.Router;
import org.restlet.routing.Route;
import org.restlet.data.Protocol;
import org.restlet.resource.ServerResource;

//Classe Application nella quale
//vengono gestite le varie richieste
public class ProvaServerRestlet extends Application {

    //Inizializzazione componenti sistema
    public static void main(String[] args) throws Exception {
        //Creazione Component
        Component comp=new Component();
        //Dichiarazione Server per protocollo
        //HTTP all'indirizzo locale sulla porta 3000
        comp.getServers().add(Protocol.HTTP, 3000);
        //inizializzazione del VirtualHost di default, questo
        //viene associato alla classe
        //Application ProvaServerRestlet
        comp.getDefaultHost().attach("/prova",
        new ProvaServerRestlet());

        //Con l'esecuzione della successiva
        //istruzione il sistema è pronto
        //a ricevere richieste dal client.
        comp.start();
    }
    @Override
    public Restlet createInboundRoot(){
        //creazione di un nuovo oggetto
        //Router esso indirizzerà
        //la richiesta alla risorsa destinatario.
        Router router=new Router(getContext());
        //associazione URI relativo-tipo risorsa
        router.attach("/hello", UserResource.class);
        return router;
```

```
    }
}
```

Risorsa "hello":

```
package prova.restlet;

import org.restlet.data.Form;
import org.restlet.data.Parameter;
import org.restlet.representation.Representation;
import org.restlet.representation.StringRepresentation;
import org.restlet.resource.ServerResource;

public class UserResource extends ServerResource{
    //Oggetto che conterrà l'insieme dei
    //parametri passati dall'utente attraverso l'URI
    Form f;
    String Rep;
    //Costruttore
    public UserResource(){
        super();
    }
    @Override
    //Comportamento a seguito di una richiesta HTTP GET
    public Representation get(){
        String nome = null;

        MediaType tipo;
        //Lista dei formati forniti dal server
        List lista=new ArrayList();
        lista.add(MediaType.TEXT_ALL);
        //ottenimento formato richiesto
        tipo=this.getRequest().getClientInfo()
        .getPreferredMediaType(lista);

        //Se il formato è supportato ritorno la rappresentazioe
        if(tipo==MediaType.TEXT_ALL){
            f=this.getRequest().getResourceRef().getQueryAsForm();
```

```
            for (Parameter parameter : f) {
                //recupero dei parametri
                nome=parameter.getValue();
            }
            //Costruzione della rappresentazione di risposta
            Representation rep;
            rep=new StringRepresentation("Ciao "+nome);
            return rep;
        }
        //Altrimenti ritorno un messsaggio di errore
        Representation rep;
        rep=new StringRepresentation("Formato non supportato");
        this.setStatus(Status.CLIENT_ERROR_METHOD_NOT_ALLOWED);
        return rep;
    }
    \\comportamento a seguito di una richiesta HTTP PUT
    \\utilizzando una annotation
    @Put
    public void PutString(){
        Rep=this.getRequest().getEntityAsText();
    }
}
```

Client:

```
package prova.utente.restlet;

import java.io.IOException;
import org.restlet.Client;
import org.restlet.Request;
import org.restlet.Response;
import org.restlet.data.Method;
import org.restlet.data.Protocol;
import org.restlet.representation.Representation;
import org.restlet.resource.ClientResource;

public class ProvaUtenteReSTlet {
    public static void main(String[] args) throws IOException {
```

```
        //Costruzione di una richiesta GET
        Request request=new Request(Method.GET,
        "http://localhost:3000/prova/hello?nome=Elia");

        Representation rep;
        //Invio della richiesta e ricezione della risposta
        Response resp=new Client(Protocol.HTTP).handle(request);
        //Recupero rappresentazione dalla risposta
        rep=resp.getEntity();
        //Visualizzazione risultati;
        System.out.println(resp.getStatus()+"\n"+rep.getText());
        }
}
```

6.2.2 ReSTlet e JavaScript client

JavaScript è un linguaggio di scripting ad oggetti utilizzato per arricchire il comportamento di siti web. Caratteristica principale di questo linguaggio è quella di essere un linguaggio interpretato, ovvero uno script scritto in JavaScript non viene compilato ed eseguito ma interpretato dall'utilizzatore (solitamente un browser web). Altra caratteristica importante riguarda il fatto che il codice inserito in un sito web verrà eseguito lato client in modo da non dover sovraccaricare il server.

Nonostante JavaScript abbia poco a che fare con il linguaggio java (essi hanno solo una somiglianza nella sintassi) i progettisti di ReSTlet hanno inserito un estensione per permettere di ospitare un client ReSTful all'interno di uno script, chiamata JavaScript edition. Questa estensione fornisce i costrutti solamente per l'implementazione lato client di ReSTlet (visto che deve essere eseguito da un browser) adattandone le caratteristiche principali per l'esecuzione in ambiente JavaScript. In particolare vengono utilizzate quattro classi per svolgere questo compito [26]:

- *Request:* rappresenta la richiesta corrente e permette di specificare l'indirizzo del servizio, il metodo HTTP e l'eventuale rappresentazione da includere nel body.

- *Client:* rappresenta il cliente del servizio, è l'elemento che effettua l'invio vero e proprio delle richieste.
- *Response:* rappresenta la risposta ricevuta a seguito di una richiesta, permette di verificare lo stato dell'interazione e di accedere alle eventuali rappresentazioni.
- *Representation:* permette di gestire i contenuti scambiati.

```
var request = new Request(Method.GET, "../resource/contact/1");
var client = new Client(new Context(), [Protocol.HTTP]);
client.handle(request, function(response) {
    var representation = response.getEntity();
    var content = representation.getText();
});
```

Come si può notare dall'esempio sopra la costruzione del client è molto simile a quanto fatto per Java, l'unica differenza consiste nella ricezione delle risposte. Questo è necessario perché l'esecuzione di script è asincrona perciò non aspetterebbe la ricezione della risposta.

6.3 Client PHP Servizi web ReSTful

PHP è un linguaggio di scripting interpretato con licenza open source, esso può essere usato come JavaScript per assegnare un comportamento dinamico ad una pagina web. Se si desidera utilizzare PHP per questo scopo può aver senso solamente creare client per servizi web ReSTful utilizzando questo linguaggio. Per questo ci viene in aiuto una classe pre–costruita chiamata *Request*[18] essa fornisce tutti i metodi per inviare richieste HTTP e gestire le risposte. Attraverso questa classe è possibile:

- creare una nuova richiesta

  ```
  $req =& new HTTP_Request("http://www.exemple.it/prova");
  ```

- impostare il metodo HTTP desiderato

  ```
  $req->setMethod(HTTP_REQUEST_METHOD_POST);
  ```

- aggiungere/rimuovere intestazioni alle richieste

  ```
  $req->addHeader($name, $value);
  $req->removeHeader($name);
  ```

- inserire query string

  ```
  $req->addQueryString($name, $value, false);
  ```

- inserire una rappresentazione all'interno della richiesta

  ```
  $req->setBody($body);
  ```

- inviare una richiesta

  ```
  $req->sendRequest();
  ```

- ottenere lo stato dell'interazione

  ```
  $req->getResponseCode();
  ```

- ottenere il corpo della risposta

  ```
  $req->getResponseBody();
  ```

Tutte queste operazioni fanno si che un qualsiasi sito web integrato con uno script PHP possa facilmente utilizzare un servizio web ReSTful restituendo all'utente una rappresentazione comprensibile dei risultati.

Capitolo 7

Conclusioni

Come è intuibile fin dall'introduzione l'elaborato di tesi si sviluppa in tre macro-sezioni, attraverso le quali vengono approfondite le principali caratteristiche dei servizi web e analizzate le due più importanti architetture che delineano i diversi stili di sviluppo.

La prima parte approfondisce gli aspetti che accomunano tutti (o quasi) i servizi web. In questa sezione emergono le principali caratteristiche (quali riuso, indipendenza dalla piattaforma, scalabilità) che hanno portato i servizi ad una così elevata diffusione nel mondo dei sistemi distribuiti. Fin dalla prima definizione data dall'organizzazione W3C [33] si comprende come questa nuova concezione possa modificare l'approccio che si ha verso la progettazione di nuovi sistemi, passando da uno stile in cui si costruiva un applicazione a partire da zero ad uno stile in cui si utilizza la composizione di più servizi per lo svolgimento dei compiti desiderati.

In questa prima parte, inoltre, viene presentato lo stack su cui i servizi poggiano le loro basi. Lo studio di questa struttura a livelli è fondamentale per capire come è strutturato un sistema orientato ai servizi e come i diversi elementi facenti parte del sistema interagiscano tra loro.

Proseguendo con la tesi si è passati ad un analisi dei due principali stili che attualmente popolano il mondo dei servizi web, concentrandosi in modo particolare sulle modalità e sugli strumenti di sviluppo per i servizi ReSTful.

Nella seconda parte vengono studiati quelli che sono considerati web ser-

vice per eccellenza. Essi sviluppano le caratteristiche descritte dall'architettura Service Oriented la quale si integra perfettamente in questo panorama. Per capire a fondo questo tipo di servizi perciò è necessario definire i principi e le entità introdotti da SOA, i quali vengono direttamente concretizzati attraverso l'insieme di tecnologie utilizzate.

L'introduzione di linguaggi come WSDL, per la costruzione dei Service Description, e SOAP, per la definizione della struttura dei messaggi, hanno infatti dato la possibilità di utilizzare i servizi in ogni contesto ed attraverso qualsiasi protocollo di comunicazione consacrando così questa tecnologia come universale. In questo insieme di tecnologie spiccano ciò che vengono chiamate WS-Specifications. Esse descrivono una serie di estensioni utilizzate per gestire la maggior parte degli aspetti legati alla QoS ed alla gestione di aspetti legati all'esecuzione (transazioni, operazioni di batch, ecc.).

L'estrema flessibilità di utilizzo e la possibilità di gestire diversi aspetti ad "alto livello" hanno reso i servizi SOA particolarmente adatti come soluzione di integrazione in ambito aziendale. Molto spesso vengono utilizzati web service per "avvolgere" sistemi già esistenti e permettere, in questo modo, la comunicazione attraverso la rete, purtroppo non sempre questi sistemi utilizzano protocolli web-friendly, perciò è necessario utilizzare componenti che possano operare in modi differenti permettendo, addirittura, la modifica del protocollo di trasporto durante il percorso. É il caso dei servizi SOA.

Infine nella terza ed ultima parte si analizzano le proprietà dei servizi ReST approfondendo in particolare gli aspetti di sviluppo. In questo contesto si può vedere la ricerca della massima semplicità che, infatti, è caratteristica do questo tipo di servizi, il tentativo di renderli il più simile possibile al resto del web è conseguenza diretta di questa proprietà, e permette agli utenti un approccio identico a quello assunto nei confronti di un qualunque sito web.

Vengono definite le modalità di descrizione del servizio, WADL e WSDL 2.0, i quali però anche attualmente non sono molto utilizzate. Essendo, infatti, definita una precisa interfaccia uniforme ed essendo possibile accedere alle risorse attraverso semplici link, non viene ritenuto utile inserire un nuovo costrutto di descrizione. Questo può essere vero fino ad un certo punto. Innanzitutto una descrizione può essere utilizzata appositamente per automatizzare la creazione di applicazioni client per questo tipo di servizi, aiutando gli sviluppatori a non commettere errori che possono essere evitati.

In secondo luogo la descrizione è necessaria per comunicare agli utenti come strutturare i messaggi che spesso richiedono l'inserimento di parametri per la corretta esecuzione dell'operazione.

Per completare lo studio dei servizi ReSTful, infine, si analizzano i due principali framework per Java e .NET, prestando particolare attenzione alle modalità con cui vengono costruiti i servizi. All'interno di questa sezione vengono mostrati, inoltre, due implementazioni di client JavaScript e PHP. Si è deciso di mostrare queste possibilità in quanto, con l'avvento del WEB 2.0, molti siti web utilizzano i servizi ReSTful per inserire un comportamento dinamico all'interno di pagine HTML altrimenti statiche, portando la diffusione dei servizi ReSTful ad alti livelli.

Come conclusione di questo elaborato è parso, comunque, doveroso effettuare un confronto tra le due tipologie di servizi descritte nei precedenti capitoli (SOA e ReST) indicando quelli che sono sembrati le forze e le debolezze delle due.

É innanzitutto necessario capire che servizi SOA e ReST hanno due obbiettivi principalmente differenti. I primi cercano di favorire l'integrazione di qualunque sistema attraverso ogni piattaforma e tecnologia. I secondi hanno lo scopo di fornire funzionalità attraverso il web sfruttando appieno le tecniche e i protocolli già ampiamente diffusi in questo mondo.

Innanzitutto le differenze derivano dal tipo di visione che le due tipologie hanno del protocollo di trasporto utilizzato.

Un servizio SOA vede il protocollo come semplice busta per i messaggi SOAP che contengono le informazioni vere e proprie. Questo uso spesso si traduce in uno spreco di potenziale, infatti il protocollo potrebbe fornire funzioni per gestire in automatico aspetti che invece necessitano l'utilizzo di apposite estensioni (basta pensare a WS-Adressing su HTTP). Inoltre l'utilizzo di messaggi SOAP, a volte si traduce in uno spreco di banda e di risorse, infatti dovendo sempre impacchettare i dati utili all'interno di SOAP cliente e fornitore sono costretti ad inserire una quantità non indifferente di informazioni all'interno del payload e a dover estrarre i dati dal pacchetto SOAP, anche quando questo si sarebbe potuto evitare. É però da notare che visto lo scopo di integrazione e di generalità proposto dai servizi SOA non vi può essere altro metodo l'invio di messaggi.

Di contro un servizio ReSTful ha una visione più universale del protocollo di trasporto e lo utilizza per gestire tutti gli aspetti relativi alla comunica-

zione, tuttavia, essendo strettamente legato al mondo del web, non vi sono alternative sulla scelta del protocollo di trasporto, HTTP(S) è l'unica soluzione. Questo può essere considerato come un vantaggio in quanto questo protocollo è supportato da qualunque piattaforma ed è ampiamente conosciuto perciò il suo utilizzo risulta molto semplice, *ma vincola molto l'uso dei servizi che non potrebbero funzionare con altri standard.* HTTP forma comunque la base di questa architettura fornendo un interfaccia uniforme che lo sviluppatore deve seguire. Tuttavia non tutti gli sviluppatori possono o vogliono seguire questo principio, vi sono diversi esempi (Flickr.com su tutti) in cui i metodi HTTP vengono utilizzati in modalità non conformi a questo vincolo. Questo causa confusione nell'uso di questi servizi in quanto non è sempre garantita l'uniformità di concezione dei metodi.

Infine un'altra importante differenza che caratterizza le due architettura è rappresentata dall'ambiente in cui lavorano.

SOA è utilizzato in un ambito aziendale in cui diversi sistemi cercano di comunicare tra loro utilizzando i propri standard. Questo ambiente è perciò molto eterogeneo e non è facile trovare un formato di comunicazione comune a tutti.

ReST lavora in un ambiente anch'esso eterogeneo ma in cui tutti utilizzano lo stesso protocollo per comunicare ed un insieme limitato di formati standard. Grazie alla grande flessibilità di HTTP nel trasporto di diversi formati un servizio ReSTful può operare senza problemi in questo panorama.

Come è immaginabile il mondo dei servizi web è in continua evoluzione, lo spostamento continuo verso nuove tecnologie e piattaforme, come ad esempio piattaforme cloud, possono far passare questo paradigma di programmazione da un semplice uso per l'integrazione e la comunicazione di più applicazioni ad uno stile in cui un applicazione è costituita interamente da più servizi cooperanti. Ad esempio si può pensare alla situazione in cui un applicazione risieda interamente su un server ed esponga alcuni suoi aspetti come servizi attraverso i quali un client può svolgere il proprio lavoro. Un esempio pratico di questo è la piattaforma google Document essa fornisce la possibilità di realizzare e modificare documenti mantenuti direttamente on-line. Guardando al futuro questo panorama si potrebbe estendere a tutte le applicazioni di uso comune portando l'idea di servizio web a livelli al momento solo immaginabili.

Bibliografia

[1] A. Arsanjani. How to identify, specify and realize services fo your soa (part ii), 2005. http://www.ebizq.net/topics/soa_security/features/5632.html.

[2] M. Bartoletti, V. Ciancia, G. L. Ferrari, R. Guanciale, D. Strollo, and R. Zunino. Architetture informatiche: l'orientamento ai servizi. *Mondo Digitale*, 2008.

[3] E. F.-M. Carlos Gutierrez and M. Piattini. Web services security: Is the problem solved? *INFORMATION SYSTEMS SECURITY*, 2004.

[4] A. Chiarelli. Guida xml di base, 2011. http://xml.html.it/guide/leggi/58/guida-xml-di-base/.

[5] A. Chiarelli. Restful web service - la guida, 2011. http://programmazione.html.it/guide/leggi/218/restful-web-services-la-guida/.

[6] L. W. David Sprott. Understanding service-oriented architecture, 2004. http://msdn.microsoft.com/en-us/library/aa480021.aspx.

[7] C. division Wily Technology. Soa e i servizi web: il paradosso delle prestazioni. *White Paper*, 2006.

[8] B. Domanski. An introduction to web services and performance issues. *IT management*, 2004.

[9] T. F. e Lorenzo Nardi. Slide corso di laboratorio applicazioni internet. 2011.

[10] W. EN. Hypertext transfer protocol. http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol.

[11] W. EN. Service oriented architecture, 2011. http://en.wikipedia.org/wiki/Service-oriented_architecture.

[12] W. EN. Soap, 2011. http://en.wikipedia.org/wiki/Simple_Object_Access_Protocol.

[13] W. EN. Web application description language, 2011. http://en.wikipedia.org/wiki/Web_Application_Description_Language.

[14] W. EN. Web service, 2011. http://en.wikipedia.org/wiki/Web_service.

[15] W. EN. Web services description language, 2011. http://en.wikipedia.org/wiki/Web_Services_Description_Language.

[16] T. Erl. *Service Oriented Architecture: concepts, technology and design.* Prentice Hall PTR, 2005.

[17] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.

[18] J. Flanders. *RESTful .NET - Build and Consume RESTful Web Services with .NET 3.5.* O'Reilly Media, 2008.

[19] K. S. Gottschalk, Graham. Introducing to web service architecture. *IBM System journal*, 2002.

[20] W. IT. Javascript. http://it.wikipedia.org/wiki/JavaScript.

[21] W. IT. Windows communication foundation. http://it.wikipedia.org/wiki/Windows_Communication_Foundation.

[22] M. S. S. M. James McGovern, Sameer Tyagi. *Java Web Services Architecture.* Morgan Kaufmann Publishers, 2003.

[23] S. R. M. Piraccini. Service oriented architecture: dalla teoria alla pratica, 2005. http://www.mokabyte.it/2005/10/soa-1.html.

[24] S. MaibÃ¼cher. Rest web services, 2008. `http://www.predic8.com/rest-webservices.htm`.

[25] MSDN. Architettura di windows communication foundation. `http://msdn.microsoft.com/it-it/library/ms733128(v=vs.90).aspx`.

[26] MSDN. Concetti fondamentali di windows communication foundation. `http://msdn.microsoft.com/it-it/library/ms731079(v=vs.90).aspx`.

[27] MSDN. A guide to designing and building restful web services with wcf 3.5. `http://msdn.microsoft.com/en-us/library/dd203052.aspx`.

[28] MSDN. How to: Create a basic wcf web http service. `http://msdn.microsoft.com/en-us/library/bb412178.aspx`.

[29] M. Papazoglou. Service oriented computing: concepts, charateristics and directions. *JAm Stat Soc*, 2001.

[30] Pear. request php class. `http://pear.php.net/package/HTTP_Request/docs`.

[31] C. Peltz. Web services orchestration and choreography. *IEEE Computer Society*, 2003.

[32] G. D. Penna. Simple object access protocol, 2007. `http://www.di.univaq.it/gdellape/download.php?fn=SOAPpdf`.

[33] G. D. Penna. Universal description, discovery and integration, 2007. `http://www.di.univaq.it/gdellape/download.php?fn=UDDIpdf`.

[34] G. D. Penna. Web service definition lenguage, 2007. `http://www.di.univaq.it/gdellape/download.php?fn=WSDLpdf`.

[35] B. Pernici and P. Plebani. Un'introduzione ragionata al mondo dei web service. *Mondo Digitale*, 2004.

[36] predic8. Ws-specifications. `http://www.predic8.com/ws-specifications.htm`.

[37] L. Richardson and S. Ruby. *ReSTful Web Services*. O'Reilly Media, 2007.

[38] A. Santi. Web service, 2011. Slide del corso di Programmazione Concorrente e Distribuita LM.

[39] R. S.A.S. Restlet. `http://wiki.restlet.org/docs_2.1/13-restlet.html`.

[40] R. S.A.S. Restlet javadoc. `http://www.restlet.org/documentation/2.1/jse/api/`.

[41] R. S.A.S. Restlet javascript edition. `http://wiki.restlet.org/developers/172-restlet/267-restlet/368-restlet.html`.

[42] R. S.A.S. Wadl extension. `http://wiki.restlet.org/docs_2.0/13-restlet/28-restlet/72-restlet.html`.

[43] V. Smothers. Rest web services design guidelines. *MedBiquitous*, 2009.

[44] S. K. K. U. C. F. Toshiro Takase, Satoshi Makino and A. Ryman. Definition languages for restful web services: Wadl vs. wsdl 2.0. *Tokyo Research Laboratory, IBM Research*, 2008.

[45] G. Turetta. Guida ai web service. `http://xml.html.it/guide/leggi/100/guida-web-service/`.

[46] W3C. Soap specification. `http://www.w3.org/TR/soap/`.

[47] W3C. Web services description language (wsdl) 1.1, 2001. `http://www.w3.org/TR/wsdl`.

[48] W3C. Web service architecture, 2002. `http://www.w3.org/TR/2002/WD-ws-arch-20021114/`.

[49] W3C. Web service architecture, 2004. `http://www.w3.org/TR/ws-arch/`.

[50] W3C. Web services description language (wsdl) version 2.0 part 0: Primer, 2007. `http://www.w3.org/TR/wsdl20-primer/`.

[51] W3C. Web application description language, 2009. `http://www.w3.org/Submission/wadl/`.

[52] W3School. Soap tutorial. http://www.w3schools.com/soap/default.asp.

[53] W3School. Wsdl tutorial. http://www.w3schools.com/wsdl/default.asp.