UNIVERSITÀ DI PADOVA

Facoltà di Scienze MM.FF.NN.
Corso di Laurea in Informatica

Tesi di Laurea

Un'architettura orientata ai servizi che offra alta disponibilità

Relatore:
prof. Gilberto Filè
Controrelatore:
prof. Massimo Marchiori

Candidato:
Eric Miotto

Supervisore Aziendale
Davide Bedin

Settembre 2007

Un'architettura orientata ai servizi che offra alta disponibilità

Candidato Eric Miotto
Relatore prof. Gilberto Filè

15 settembre 2007

Sommario

In questa tesi verrà illustrato in che modo è stato realizzato un sistema prototipale di trasmissione log che sia flessibile, scalabile, ad alta disponibilità ed economico da mantenere. In particolare si enfatizzerà:

- la scelta di un'architettura di tipo Service Oriented Architecture (SOA), che permette di realizzare un sistema flessibile;
- la comunicazione disaccoppiata tramite le code di Amazon Simple Queue Service (SQS), che permettono di raggiungere scalabilità ed alta disponibilità ad un costo ragionevole;
- l'implementazione della comunicazione usando Windows Communication Foundation (WCF), tecnologia semplice ed allo stesso tempo aperta a nuovi scenari.

Indice

1 Introduzione **2**
1.1 Convenzioni ed organizzazione della tesi 3

2 Requisiti e scelte progettuali **4**
2.1 Descrizione del problema 4
2.2 Presentazione della soluzione 5
2.3 Architettura SOA 7
2.4 Comunicazione tramite code 9
2.5 Realizzazione tramite WCF 12

3 Realizzazione del prodotto **13**
3.1 Compendio di WCF 13
3.2 Progettazione del contratto del servizio 14
3.3 Funzionamento del trasporto per SQS 17

4 Strumenti **21**
4.1 Service Oriented Architecture 21
4.1.1 Web Service 24
4.2 Amazon Simple Queue Service e Software as a Service 25
4.3 Windows Communication Foundation 27

Glossario **30**

Sigle e acronimi **32**

Bibliografia **33**

Capitolo 1

Introduzione

Durante lo stage l'Autore ha implementato un sistema prototipale di trasmissione dei log con il quale le applicazioni dell'azienda possano segnalare una serie di informazioni sui vari eventi occorsi al loro interno (esecuzione delle normali operazioni, errori, ...). I requisiti principali che il sistema deve soddisfare sono:

- *possedere un'architettura flessibile*. In particolare, deve essere possibile che ogni parte del sistema distribuito possa essere aggiornata indipendentemente dalle altre;
- *garantire alta disponibilità*, in modo che ci sia sempre qualcuno in grado di ricevere i log delle applicazioni;
- *essere scalabile*. Nel nostro caso significa sostenere picchi di traffico dovuti per esempio da segnalazioni contemporanee di un errore che interessa un numero cospicuo di applicazioni;
- *essere economico da mantenere*. Il raggiungimento dei due precedenti requisiti richiede solitamente notevoli risorse. Noi vogliamo invece che il sistema possa essere mantenuto da una piccola/media azienda.

Nei prossimi capitoli vogliamo spiegare nel dettaglio il problema affrontato, come è stato realizzato il sistema e le motivazioni che hanno portato alle scelte fatte. Concentreremo l'attenzione su tre aspetti dell'architettura:

1. l'adozione di un'architettura di tipo Service Oriented Architecture (SOA). SOA è un paradigma per la realizzazione di sistemi distribuiti che promuove:
   - il riuso delle varie parti, permettendo che queste ultime siano impiegate per altre soluzioni;
   - l'interoperabilità tra parti realizzate da diverse entità con tecnologie differenti;
   - la possibilità di cambiare una soluzione in modo più semplice rispetto al passato.

2. l'uso del servizio di code di Amazon Simple Queue Service (SQS). Le applicazioni non inviano i log direttamente ad un server dell'azienda, ma ad SQS, che li memorizza al suo interno. Il server dell'azienda recupererà in un secondo momento i log da SQS.

   In prima approssimazione possiamo dire che i requisiti di alta disponibilità e di scalabilità ricadono su SQS. Non solo SQS soddisfa pienamente i requisiti, ma il suo uso risulta economico. Questo perché SQS aderisce al paradigma Software as a Service (SaaS), che permette alle aziende di investire meglio il budget destinato al software.

3. l'utilizzo di Windows Communication Foundation (WCF) per implementare il prodotto. WCF è una tecnologia per realizzare sistemi orientati ai servizi. Nel progetto sono state apprezzate due caratteristiche:

   - la facilità con cui si modella la comunicazione;
   - la possibilità di estendere i meccanismi di comunicazione in modo da astrarre l'uso di SQS e permettere quindi di cambiare modalità di trasporto in futuro, senza dover buttar via tutto ciò che è stato fatto.

1.1 Convenzioni ed organizzazione della tesi

Nel testo si useranno le seguenti convenzioni:

- il codice verrà riportato con carattere sans serif, per esempio <DataContract>;
- i collegamenti a siti internet verranno riportati in carattere typewriter e in colore magenta, per esempio `http://aws.amazon.com/sqs`;
- i collegamenti a capitoli, sezioni, liste, figure, voci di glossario ed elementi della bibliografia verranno riportati in colore blu, per esempio Attributo.

La trattazione è organizzata come segue. Nel capitolo 2 verranno discussi il problema affrontato nello stage, la soluzione adottata e le motivazioni dei tre aspetti appena introdotti (architettura SOA, uso di SQS, utilizzo di WCF).

Nel capitolo 3 si parlerà in dettaglio della progettazione del contratto esposto dal servizio di trasmissione log e si spiegherà come è stato realizzato il trasporto che permette di astrarre la comunicazione tramite SQS.

Nel capitolo 4 si introdurranno gli strumenti concettuali ed effettivi con i quali è stato realizzato il progetto dello stage: SOA, SQS e SaaS, WCF.

Capitolo 2

Requisiti e scelte progettuali

2.1 Descrizione del problema

Lo stage è consistito nella realizzazione di un sistema prototipale per la trasmissione di log con garanzie di alta disponibilità e scalabilità.

L'azienda presso la quale è stato fatto lo stage realizza diversi prodotti per la gestione dei negozi. In questi programmi si verificano continuamente eventi, come per esempio il login di un utente in un sistema, l'esecuzione di una particolare operazione, il sollevamento di un'eccezione, ... Con il termine log indichiamo una collezione di informazioni relative ad una serie di eventi che sono accaduti nel tempo in una o più applicazioni. E' opportuno che questi log vengano trasmessi all'azienda in modo che le applicazioni possano essere migliorate e corrette.

Lo scenario che il sistema supporta è il seguente:

1. in una applicazione si verificano una serie di eventi che devono essere segnalati;
2. l'applicazione si collega ad Internet e trasmette il log degli eventi che si sono verificati ad un server dell'azienda;
3. il server riceve il log e lo salva su database;
4. all'interno dell'azienda i log vengono consultati periodicamente per capire come le applicazioni vengano usate, per individuare aree di miglioramento e per correggere gli errori.

Indicheremo con *client* un'applicazione che contatta il server per trasmettere dei log.

E' richiesto che vengano soddisfatti i seguenti requisiti:

R1. l'interfaccia tra client e server deve essere progettata ed implementata in modo da essere retrocompatibile. In particolare, se l'interfaccia deve essere aggiornata, devono verificarsi due condizioni:

a) l'aggiornamento dell'interfaccia deve richiedere solo aggiunte e non modifiche, cancellazioni o cambiamenti di semantica di ciò che già esiste. In questo modo l'ultima versione dell'interfaccia conterrà sempre le più vecchie;

b) un client che conosce una versione precedente dell'interfaccia può ancora comunicare con il server. Il client non deve essere costretto a conoscere la nuova interfaccia per fare le cose che già faceva con la vecchia interfaccia.

R2. l'architettura deve essere predisposta per poter utilizzare agevolmente funzionalità esposte da terze parti, indipendentemente dalle tecnologie utilizzate da noi e dalla terza parte;

R3. il sistema deve essere ad alta disponibilità e scalabile. Con questa affermazione chiediamo due cose:

a) il client deve sempre poter inviare log (a meno di problemi di connettività o di congestione). Non è detto infatti che, in caso di impossibilità a mandare i log, il client li conservi e ritenti la spedizione in un momento successivo;

b) il server deve sempre ricevere tutti i log che i client hanno inviato, a prescindere dal traffico che i client possono generare. Non possiamo permetterci di perdere log.

R4. il costo dell'infrastruttura del sistema deve essere sostenibile da una realtà piccola o media;

R5. il sistema deve essere implementato usando .NET. L'azienda infatti lo utilizza per tutte le sue applicazioni;

R6. è desiderabile che le tecnologie usate permettano di modellare la comunicazione tra client e server astraendo dai particolari protocolli usati.

In particolare vogliamo in futuro poter cambiare i protocolli usati senza dover cambiare le implementazioni di client e server, ma semplicemente aggiungendo componenti che gestiscano i nuovi protocolli che ci interessano.

2.2 Presentazione della soluzione

In risposta ai requisiti 1 e 2 abbiamo scelto una architettura di tipo Service Oriented Architecture (SOA). SOA è un paradigma per l'organizzazione e l'utilizzo di capacità distribuite che possono essere sotto il controllo di diverse entità. Lo strumento per organizzare ed esporre le capacità è il servizio.

I punti di forza di SOA sono l'accoppiamento lasco tra le varie componenti, l'indipendenza dalle tecnologie e la facilità con cui è possibile far evolvere la soluzione in risposta al nascere di nuove esigenze.

Avremo quindi un servizio attraverso il quale i client potranno segnalare i loro log. La comunicazione tra client e servizio non è diretta: per poter soddisfare il requisito 3 si è preferito invece disaccoppiare la comunicazione tra le due parti facendo passare tutti i messaggi tramite una coda scalabile e ad alta disponibilità fornita da una terza parte.

Il servizio di code scelto è Amazon Simple Queue Service (SQS), che offre ottime garanzie ed ha costi molto bassi (soddisfando così il requisito 4). Le caratteristiche di SQS derivano dal fatto che aderisce al paradigma Software as a Service (SaaS), che concepisce il software non come prodotto da comprare ed installare ma come un insieme di funzionalità pronte da usufruire.

Tutto il progetto è stato realizzato utilizzando .NET Framework 3.0 e Visual Studio 2008, soddisfacendo banalmente il requisito 5. I linguaggi utilizzati sono Visual Basic .NET e C♯.

Per implementare la comunicazione tra il servizio e il client è stato usato Windows Communication Foundation (WCF), una delle tecnologie che compongono .NET Framework 3.0. WCF permette di modellare in modo molto semplice il servizio e di consumarlo altrettanto semplicemente. L'estensibilità di WCF ha anche permesso un'implementazione un po' particolare: al posto di implementare direttamente la comunicazione con SQS, è stato studiato ed adattato un componente, detto *trasporto*, che nasconde questo comportamento. In questo modo si possono programmare client e servizio come se comunicassero direttamente. In questo modo è soddisfatto anche il requisito 6.

La soluzione presenta l'architettura illustrata in figura 2.1. L'illustrazione evidenzia una parte server, che si occupa di ricevere e memorizzare i log, ed una parte utente, la quale spedisce i log degli eventi che in essa si sono verificati.

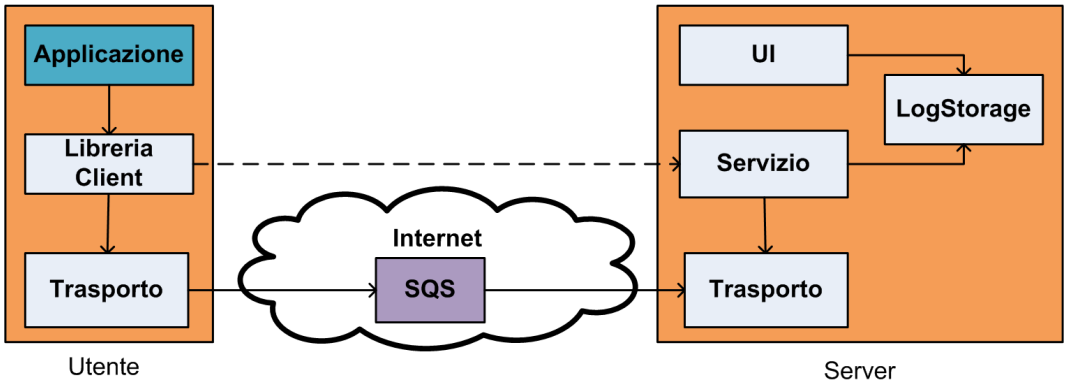

Figura 2.1: Architettura del prodotto. La linea tratteggiata tra libreria client e servizio indica che ad alto livello è come ci fosse comunicazione diretta tra le due componenti: in realtà avviene tramite il trasporto e SQS.

Nella parte utente troviamo la *libreria client*, che fondamentalmente è un facade per usare correttamente il servizio. In linea teorica ogni applicazione che vuole usufruire del servizio potrebbe consumare direttamente il servizio. La libreria client non fa altro che semplificare la comunicazione con il servizio occupandosi di tutti i dettagli riguardo la semantica delle operazioni e richiedendo all'applicazione di fornire le informazioni desiderate. Questo fatto è stato evidenziato nel diagramma in figura 2.1 disegnando una generica applicazione che usa la libreria.

Nella parte server troviamo invece:

- il *servizio* di trasmissione log;

- l'*interfaccia grafica* per consultare i log;
- *LogStorage*, il luogo in cui i log vengono memorizzati, usato dal servizio e dall'interfaccia grafica. Il componente è composto dal database e dalle classi che permettono l'accesso al database;

La parte utente e la parte server condividono l'uso dei componenti più importanti dall'architettura:

- *SQS*, la coda attraverso la quale passano tutti i messaggi:
- il *trasporto* usato dalla libreria client e dal servizio, che fa uso di SQS per la trasmissione dei messaggi.

2.3 Architettura SOA

Esistono vari paradigmi per la realizzazione di sistemi distribuiti. Possiamo individuare due obiettivi che un paradigma deve raggiungere:

- adottare un'astrazione semplice da usare e da capire;
- rendere evidenti i problemi dei sistemi distribuiti, in modo che la progettazione ne tenga conto e li affronti. Secondo [3], i problemi dei sistemi distribuiti sono:
  - la latenza delle operazioni;
  - il fatto che il sistema non è compreso all'interno di un'unica area di memoria;
  - il fatto che il sistema può non funzionare in parte (alcuni nodi attivi ed altri no, comunicazione tra i nodi assente);
  - la concorrenza, la cui presenza è più marcata rispetto all'ambito locale.

Prima dell'avvento di SOA, il paradigma predominante era quello ad oggetti distribuiti. In questo paradigma il sistema viene visto come un insieme di oggetti - tipicamente sia locali che remoti - le cui interazioni realizzano le funzionalità desiderate. Tre sono i vantaggi di questo paradigma:

1. usa un concetto - gli oggetti - già conosciuto e ben sfruttato nei linguaggi di programmazione. Non c'è quindi necessità di imparare nuove cose, ma si può sfruttare quasi subito ciò che si conosce;
2. l'uso degli oggetti porta con se il principio di astrazione, che enfatizza il ruolo dell'interfaccia e nasconde l'implementazione agli occhi dell'utilizzatore;
3. il programmatore tratta gli oggetti remoti come fossero oggetti locali, senza rendersi conto che in realtà sono su un'altra macchina. In questo modo abbiamo un solo modo per trattare tutti gli oggetti del sistema, senza dover distinguere tra oggetti locali e remoti: le differenze tra i due tipi di oggetti vengono gestite in modo trasparente.

Si può quindi progettare un sistema pensandolo come un insieme di oggetti locali, in modo che sia più facile ragionarci sopra e testare la sua correttezza; solo in un secondo momento decideremo quali oggetti devono diventare remoti, senza dover cambiare ciò che è già stato fatto.

Per i requisiti individuati, il paradigma ad oggetti distribuiti non va bene:

- l'idea di trattare in modo uniforme oggetti locali ed oggetti remoti con il tempo non si è rivelata vincente. Il problema fondamentale è che nell'interazione con oggetti remoti subentrano delle problematiche che non ci sono trattando con oggetti locali. Per esempio, l'invocazione di un metodo su un oggetto remoto risente della latenza della rete, cioè del tempo che passa tra l'invocazione del metodo sul client e l'inizio della sua esecuzione da parte del server. Il valore della latenza tende ad essere elevato, nell'ordine dei secondi, e quindi influisce sul tempo di esecuzione del metodo. Il problema si amplifica quando dobbiamo invocare molti metodi su un oggetto remoto.

  Questi problemi non possono essere risolti solo dall'infrastruttura che rende gli oggetti remoti come fossero locali, ma devono essere affrontati con una progettazione adeguata dell'interfaccia degli oggetti: tornando all'esempio della latenza, è opportuno che l'interazione con un oggetto remoto richieda l'invocazione di pochi metodi.

  L'approccio alla progettazione illustrato al punto 3 porta ad avere oggetti remoti con interfacce che non sono adatte a gestire i problemi dei sistemi distribuiti. Gli oggetti remoti devono essere quindi differenti dagli oggetti locali;

- il cambiamento dell'interfaccia di un oggetto remoto richiede di aggiornare e ricompilare tutte le applicazioni che fanno uso dell'oggetto remoto, anche quando la nuova interfaccia contiene esattamente quella vecchia e le applicazioni non fanno uso delle nuove aggiunte. Come espresso nel requisito 1 della sezione 2.1, noi vogliamo invece più flessibilità;

- le tecnologie per implementare sistemi ad oggetti distribuiti, nella maggior parte dei casi, sono scarsamente interoperabili. Il paradigma in se non impedisce l'interoperabilità, ma le sue implementazioni non raggiungono questo obiettivo.

Alla fine degli anni 90 ha cominciato ad affermarsi un nuovo paradigma per i sistemi distribuiti, Service Oriented Architecture (SOA). SOA non è rivoluzionario ma nasce sostanzialmente dalle osservazioni fatte sopra. Infatti:

- lo strumento per modellare la soluzione è il servizio, inteso come generica entità che offre delle capacità a chi ne fa richiesta. L'introduzione di una nuova entità serve a far capire che questa è diversa da un oggetto. In effetti, l'unica caratteristica che il servizio eredita dagli oggetti è la separazione tra interfaccia ed implementazione;

- la comunicazione con un servizio avviene attraverso scambio di messaggi. Così facendo è più chiaro che stiamo comunicando con una risorsa remota: per esempio, è più chiaro che i messaggi impiegano tempo per essere trasmessi dal client e ricevuti dal servizio;

- SOA promuove una maggior flessibilità nella gestione e nell'uso dei servizi, in modo che la soluzione possa evolvere aggiungendo o togliendo servizi. I servizi stessi devono essere progettati per avere un'interfaccia il più stabile possibile o che possa evolvere senza rompere la retrocompatibilità;
- SOA non impone alcuna tecnologia per la sua adozione in un sistema; anzi, ogni servizio deve rendere esplicito il modo in cui bisogna comunicare con lui, in modo da facilitare i suoi potenziali client. Questo punto ha maggior efficacia se i servizi tendono ad usare standard riconosciuti (come succede nei Web Service). L'interoperabilità si può dunque raggiungere in modo efficace.

In sezione 4.1 è possibile trovare una breve introduzione a SOA.

2.4 Comunicazione tramite code

In questa sezione discuteremo più dettagliatamente della scelta di disaccoppiare la comunicazione tra la libreria client e il servizio tramite l'uso di una coda e come questa scelta soddisfi il requisito 3. A tal scopo illustreremo tre proposte, di cui l'ultima è quella adottata:

a) utilizzo di un servizio ospitato dentro l'azienda (figura 2.2a);

b) utilizzo di un servizio ospitato presso una terza parte (figura 2.2b);

c) utilizzo di un servizio di code di una terza parte (figura 2.2c).

Utilizzo di un servizio ospitato dentro l'azienda La prima cosa che viene in mente è ospitare il servizio di trasmissione log nella stessa infrastruttura usata per rendere disponibili gli altri servizi dell'azienda. Questa soluzione non è adatta per due motivi:

- per una realtà di piccole/medie dimensioni è troppo costoso dotarsi di un'infrastruttura che riesca a garantire i requisiti di alta disponibilità e di scalabilità che sono richiesti dal problema;
- il fatto di usare la stessa infrastruttura dei servizi usati dalle applicazioni mina la disponibilità del servizio di trasmissione log.

  Supponiamo che un'applicazione abbia dei problemi a comunicare con un servizio, che chiameremo svc1 per chiarezza di esposizione. Supponiamo che questo accada perché svc1 non è attivo. Lo stato di svc1 potrebbe essere tale da inficiare la raggiungibilità degli altri servizi ospitati nell'infrastruttura, compreso quello di trasmissione log. Se quindi l'applicazione volesse trasmettere il log sulla mancata comunicazione con svc1, potrebbe trovarsi impossibilitata a farlo perché non riesce a comunicare nemmeno con il servizio di trasmissione log;
- si mina anche la scalabilità del servizio. Poniamoci in uno scenario in cui abbiamo una serie di client su ognuno dei quali gira una data applicazione. L'applicazione per svolgere il suo lavoro accede ad un

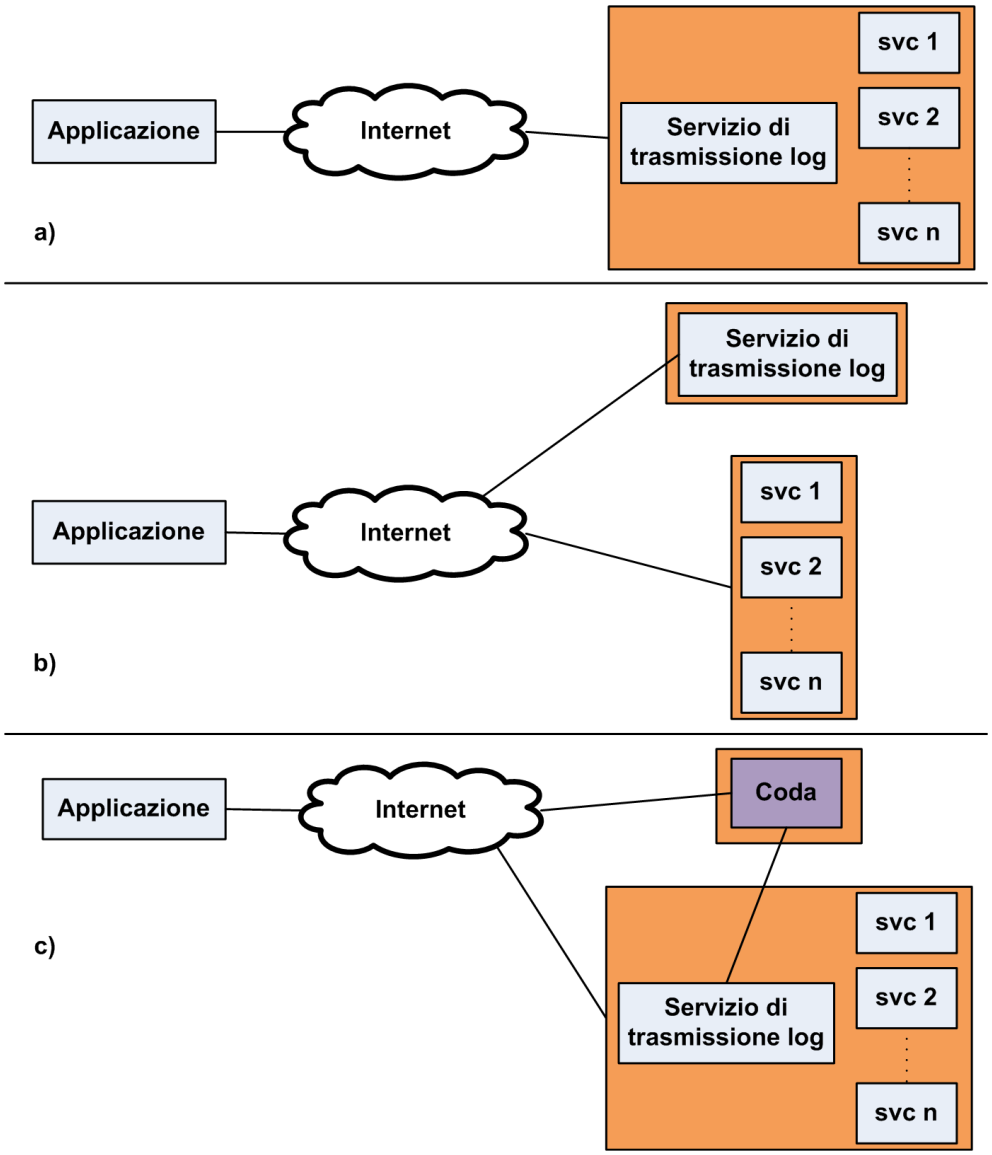

Figura 2.2: Proposte di soluzione per quanto riguarda la comunicazione tra client e servizio. I servizi e le applicazioni sono in azzurro, mentre le infrastrutture sono in arancione. Le connessioni che arrivano alle infrastrutture indicano che vengono usati tutti i servizi contenuti; le connessioni che arrivano direttamente ai servizi segnalano l'uso specifico di un servizio. a) Utilizzo di un servizio ospitato dentro l'azienda. b) Utilizzo di un servizio ospitato presso una terza parte. c) Utilizzo di un servizio di code di una terza parte. La coda è stata evidenziata in viola.

servizio dell'azienda (che chiamiamo sempre svc1). L'applicazione inoltre è progettata per segnalare *appena possibile* problemi ed errori che si sono verificati nell'interazione con svc1.

Possiamo individuare tre tipologie di errori:

1. *errori che interessano un solo client.* Questi non ci creano problemi modulo le considerazioni del punto precedente. Notiamo che questo tipo di errori saranno poco frequenti: sarà invece più probabile che un errore riguardante l'applicazione o il servizio sia segnalato da più client;
2. *errori che riguardano un gruppo di client.* Ci interessano in particolare due problemi:
   - ai client manca la connettività ad Internet (ma non all'azienda);
   - i client hanno un accesso rallentato a svc1 a causa di congestioni, per cui all'atto pratico svc1 risulta inutilizzabile.

   Quando le condizioni tornano normali, tutti i client vorranno trasmettere i log sul problema. Siccome queste trasmissioni saranno effettuate in automatico, tutti i client invieranno messaggi in un lasso di tempo contenuto;
3. *errori che riguardano tutti i client.* Questo succede perché svc1 non funziona o è congestionato. I client vorranno quindi segnalare il problema, e lo faranno in un lasso di tempo contenuto.

Nel caso di comunicazione diretta tra client e servizio di trasmissione log, nel caso 2 avremo un'enorme quantità di traffico che difficilmente riusciremo a sopportare, mettendo a rischio la disponibilità dell'infrastruttura dell'azienda. Nel caso 3 il rischio è quello di aggravare ulteriormente lo stato del servizio e dell'infrastruttura: l'enorme traffico contribuirà a peggiorare la situazione di scarsa disponibilità già presente.

Utilizzo di un servizio ospitato presso una terza parte Il problema della prima proposta di soluzione è che il servizio di trasmissione log condivide la stessa infrastruttura dei servizi usati dalle applicazioni. Possiamo quindi pensare di affidare ad una terza parte la gestione del nostro servizio. Se questa terza parte serve molteplici utenti, avrà un'infrastruttura totalmente indipendente da quella dell'azienda. Da ciò consegue che quando l'infrastruttura dei servizi dell'azienda non funzionerà quella della terza parte sarà con ogni probabilità attiva. In altre parole, quando i nostri servizi non funzioneranno è più probabile che il servizio di trasmissione log sia attivo e pronto a ricevere i log delle applicazioni.

Inoltre questo evita, negli scenari di picco di traffico discussi prima, di introdurre ulteriori problemi oltre a quelli che i client vogliono segnalare, dato che il traffico non va ad interessare l'infrastruttura dell'azienda.

Utilizzo di un servizio di code di una terza parte A questo punto possiamo fare ancora meglio. Non c'è nessuna necessita di dislocare il servizio di trasmissione log. Quello che in realtà ci serve è un servizio di una terza parte che risponda ai seguenti requisiti:

- si comporti come una coda. Una coda è un componente (in questo caso un servizio) il cui scopo è memorizzare una serie di messaggi e renderli disponibili senza perderne neanche uno. Quello che il servizio deve fare è ricevere e memorizzare tutti i log inviati dai client fino a che il servizio di trasmissione log non li recupera;
- assicuri alta disponibilità, sperabilmente la massima ottenibile, in modo che i client possano sempre trasmettere messaggi;
- sia in grado di reggere tutto il traffico generato dai client, specialmente nelle condizioni di picco discusse per la prima proposta;
- abbia un costo sostenibile per una piccola realtà;
- come per la proposta precedente, sia dislocata in una infrastruttura indipendente da quella dell'azienda.

Si può vedere come i requisiti cruciali del progetto siano ora a carico della coda e non più del servizio di trasmissione log.

E' stata quindi implementata la terza proposta di soluzione. Il servizio di code scelto è Amazon Simple Queue Service (SQS), descritto in dettaglio nella sezione 4.2.

2.5 Realizzazione tramite WCF

Il requisito 5 ci impone di usare .NET per implementare il sistema, e quindi restringe l'ambito delle possibili tecnologie per soddisfare il requisito 6 di sezione 2.1.

All'interno del .NET Framework troviamo tre tecnologie per fare applicazioni distribuite orientate ai servizi:

- Web Service di ASP.NET;
- Web Service Enhancements 3.0;
- Windows Communication Foundation.

ASP.NET è una libreria per fare applicazioni Web e permette anche di costruire e consumare Web Service. E' molto semplice da utilizzare, ma non soddisfa il requisito 6: la comunicazione tra client e servizio può avvenire solo tramite HTTP, non c'è modo di introdurre modi diversi di trasmettere messaggi.

Web Service Enhancement 3.0 estende i Web Service di ASP.NET aggiungendo il supporto per diversi protocolli WS-* e introducendo il supporto alla trasmissione di messaggi su TCP/IP. Non fa niente che possa soddisfare il requisito 6.

Windows Communication Foundation (WCF) è una tecnologia introdotta nel 2006. Permette di modellare il servizio in modo più efficace rispetto ad ASP.NET ed è stato progettato affinché i suoi meccanismi possano essere estesi con efficacia, compresa la modalità di comunicazione. Una descrizione più dettagliata di WCF si può trovare in sezione 4.3.

Capitolo 3

Realizzazione del prodotto

3.1 Compendio di WCF

In questo capitolo useremo la terminologia e i concetti della sezione 4.3. In particolare, per chiarezza d'esposizione, denotiamo con *contratto* l'elenco delle funzionalità offerte da un servizio e con *interfaccia* il costrutto usato nei linguaggi orientati agli oggetti per indicare una serie di metodi che una classe deve esporre all'esterno.

Ricordiamo come in WCF il contratto di un servizio si specifica tramite classi ed interfacce opportunamente decorate con attributi. Le classi e le interfacce così decorate vengono manipolate direttamente dal programmatore per implementare la logica del servizio; a runtime gli attributi vengono usati per generare ed interpretare correttamente i messaggi che client e servizio si scambieranno.

Le classi e le interfacce servono solo come modello per stabilire le capacità offerte e i messaggi accettati. Nella progettazione non possiamo quindi sfruttare le caratteristiche object oriented di questi costrutti: solo l'ereditarietà viene tradotta e gestita in WCF e non sempre è il modo migliore per modellare alcuni aspetti. Dobbiamo perciò progettare pensando a servizi e messaggi.

Facciamo un breve riepilogo degli attributi più importanti:

- <ServiceContract> serve per indicare che una determinata interfaccia corrisponde al contratto di un servizio;
- <OperationContract> serve per indicare che un determinato metodo di un'interfaccia <ServiceContract> è una funzionalità esposta dal servizio. In particolare, gli argomenti del metodo determineranno il contenuto del messaggio usato per invocare la funzionalità e il risultato determina il contenuto dell'eventuale messaggio che il servizio restituisce dopo aver svolto la funzionalità;
- WCF sa come serializzare e deserializzare diversi tipi del .NET Framework. Le classi definite dall'utente devono essere invece esplicitamente marcate con l'attributo <DataContract> e al loro interno bisogna marcare con <DataMember> i campi e le proprietà che saranno serializzati e deserializzati in un messaggio.

Una spiegazione più dettagliata ed articolata del ruolo degli attributi di WCF si trova in sezione 4.3.

3.2 Progettazione del contratto del servizio

La definizione del contratto è condizionata dal fatto che usiamo SQS per la comunicazione tra client e servizio:

- la comunicazione è solo dal client verso il servizio e non viceversa (si dice la comunicazione è one way, in un verso solo). Se per esempio all'invio di un log ci aspettassimo che il servizio ci dica l'esito dell'operazione (successo/errore) allora avremmo accoppiato la comunicazione ed avrebbe poco senso usare le code, dato che richiediamo che il servizio di trasmissione log sia sempre attivo;

- SQS prevede una dimensione massima dei messaggi di 256 KB. Questo aspetto è cruciale se pensiamo di trasmettere log verbosi. L'idea è quella di progettare un contratto che porti a trasferire poche informazioni per volta. Per ottenere ciò, è stato introdotto il concetto di entry. Una entry è un'informazione su un singolo evento: un log può essere quindi definito come una collezione di entry. La nostra speranza è che una singola entry sia abbastanza piccola da poter essere contenuta all'interno di un messaggio di SQS.

La progettazione del contratto è guidata inoltre anche dal requisito 1 indicato in sezione 2.1.

Con le considerazioni finora fatte, giungiamo ad una prima versione del contratto visibile in figura 3.1. Il contratto presenta attualmente un solo metodo per spedire una entry generica e che non restituisce niente. Quest'ultimo fatto è ulteriormente enfatizzato dall'aggiunta dell'attributo <OneWay> al metodo, per indicare che non viene restituito nessun messaggio al chiamante (che è diverso da restituire un messaggio che non contiene niente).

Per le entry è stata prevista una gerarchia, che per adesso contiene il tipo di entry appena citato ed una classe base con le informazioni che ogni entry dovrà fornire. Con il tempo questa gerarchia si arricchirà di nuovi tipi di entry: applicazioni diverse molto probabilmente avranno informazioni specifiche da spedire. Per ogni tipo di entry ci sarà un metodo per la sua spedizione.

Si sarebbe potuto prevedere un unico metodo di spedizione per spedire il tipo base di entry, adatto a spedire tutti i tipi derivati, come si farebbe in ambito object oriented. Questa cosa si può infatti ottenere con WCF senza particolari problemi.

La scelta compiuta permette però di evolvere più agevolmente il contratto senza rompere la compatibilità. Infatti per ogni nuovo tipo di entry che introduciamo aggiungiamo un nuovo metodo: aggiungendo nuove cose non rompiamo la compatibilità. I vecchi client ignorano tranquillamente le nuove entry e i nuovi metodi, mentre abbiamo la possibilità di cambiare i prototipi dei nuovi metodi.

L'entry generica EventLogEntry contiene:

- *un identificativo della entry.* Questo identificativo deve essere generato dalla macchina che invia l'entry, senza nessun controllo centrale. Per assicurare l'univocità di questi identificativi, si è scelto di usare i GUID;

- *un identificativo di correlazione.* Anch'esso un GUID, serve per mettere in relazione entry che provengono dallo stesso log. E' cura del client usare appropriatamente questo campo;

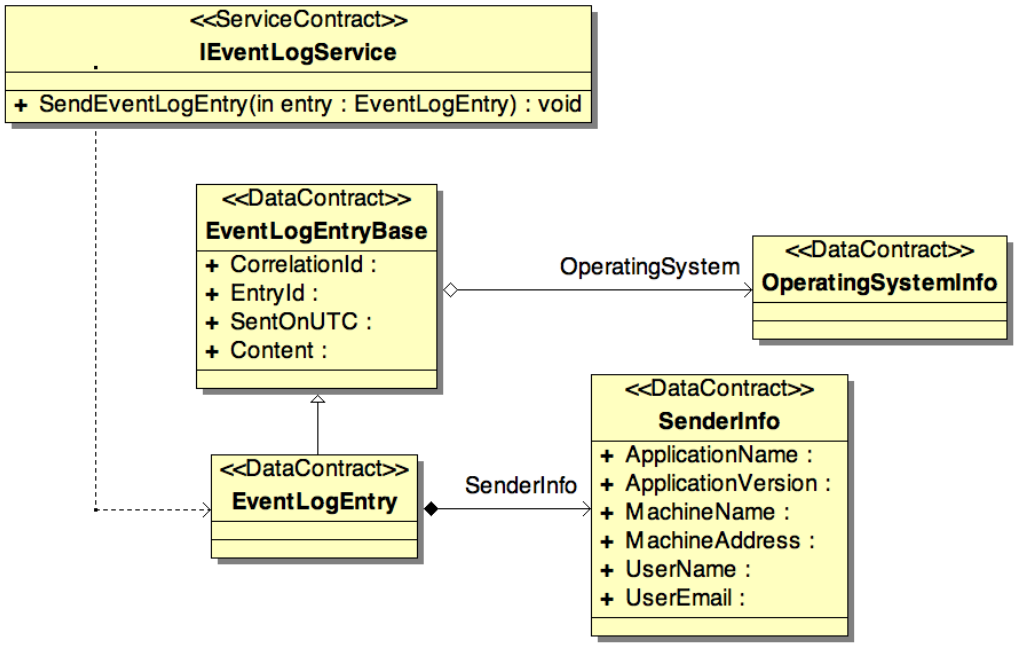

Figura 3.1: Progettazione del contratto del servizio. Si assume che IEventLogService sia un'interfaccia e che il suo unico metodo sia marcato con <OperationContract> e <OneWay>; si assume inoltre che ogni campo delle classi <DataContract> sia marcato con <DataMember>.

- *le informazioni sul sistema operativo*;
- *le informazioni sul mittente*. Si è pensato che informazioni sull'applicazione, sul computer e sull'utente potessero essere utili nell'analisi dei log;
- *la data e l'ora UTC* in cui l'entry è stata spedita;
- *il contenuto dell'entry*. E' semplicemente una stringa: non viene imposto alcun formato sulle informazioni dell'entry.

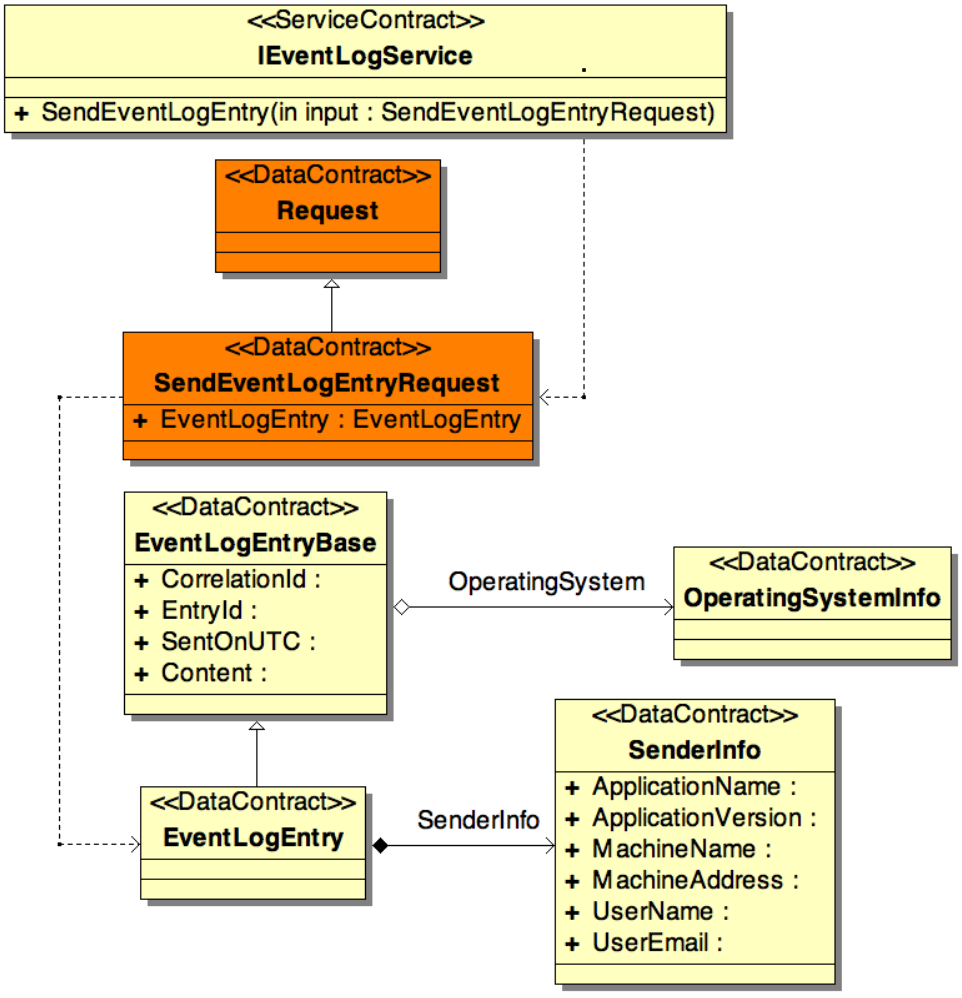

Figura 3.2: Raffinamento della progettazione del contratto del servizio. Request e SendEventLogEntryRequest sono evidenziate in arancione. Valgono le convenzioni di figura 3.1. Questo è il contratto implementato nel prodotto.

Per raggiungere meglio il requisito di retrocompatibilità, il contratto è stato ulteriormente raffinato. Come si vede in figura 3.2, l'unico argomento non è fornito direttamente, ma è contenuto all'interno di un oggetto. Questo accorgimento è stato adottato per facilitare l'evoluzione del contratto. Vediamo i motivi della scelta.

Il modo in cui WCF stabilisce il formato del messaggio d'invocazione per un metodo <OperationContract> permette la retrocompatibilità: tutti gli argomenti del metodo vengono infatti tradotti come elementi che possono essere presenti zero o una volta, per cui l'aggiunta di un nuovo argomento non inficia la comunicazione con client che si aspettano il vecchio prototipo.

Il problema è che in WCF non c'è modo di controllare in modo più fine come devono essere serializzati i parametri, per esempio stabilendo che devono esserci dei parametri obbligatori o imponendo un ordine di serializzazione diverso da quello che si può desumere dall'ordine degli argomenti. All'interno delle classi marcate come <DataContract> queste cose sono invece possibili.

Nel metodo SendEventLogEntry usiamo quindi come argomento un oggetto ad hoc per contenere i veri parametri, in modo da poter controllare la loro serializzazione in modo più preciso e poter quindi avere più strumenti per gestire la retrocompatibilità.

3.3 Funzionamento del trasporto per SQS

Vediamo ora un po' nel dettaglio come funziona il trasporto che permette la comunicazione tra client e servizio tramite SQS. Premettiamo che non siamo partiti da zero ma da un trasporto disponibile all'indirizzo http://tinyurl.com/2f7xo6, realizzato in C♯. Lo abbiamo studiato, corretto, aggiornato ed espanso appropriatamente per poterlo usare durante lo stage. In figura 3.3 anticipiamo il diagramma delle classi, che verrà spesso riferito durante la spiegazione.

Cerchiamo di capire più precisamente cos'è un trasporto all'interno di WCF. In sezione 4.3 si accenna al binding, che specifica il modo in cui un servizio comunica con i suoi client. Un binding è implementato concretamente da una classe e vive all'interno del channel layer. WCF possiede una serie di binding predefiniti.

Un binding non è monolitico: contiene invece uno stack di binding element. Un binding element si occupa di implementare un particolare aspetto della comunicazione. Possiamo individuare due tipi di binding element:

- binding element che implementano un protocollo (per esempio WS-Security);
- binding element che trasmettono effettivamente i messaggi (per esempio tramite HTTP).

La struttura di un binding è simile a quella di uno stack di rete: il primo binding element dello stack riceve il messaggio da inviare ed applica il protocollo di sua competenza, poi passa il messaggio al binding element successivo. Questo ciclo si ripete finché non si arriverà al binding element che si occupa della trasmissione dei messaggi (che ragionevolmente sarà l'ultimo dello stack), che deve svolgere direttamente il lavoro. Il discorso è duale all'atto della ricezione di un messaggio. Ogni binding element fornisce un servizio al binding element che è sopra di lui e usufruisce del servizio del binding element sotto di lui.

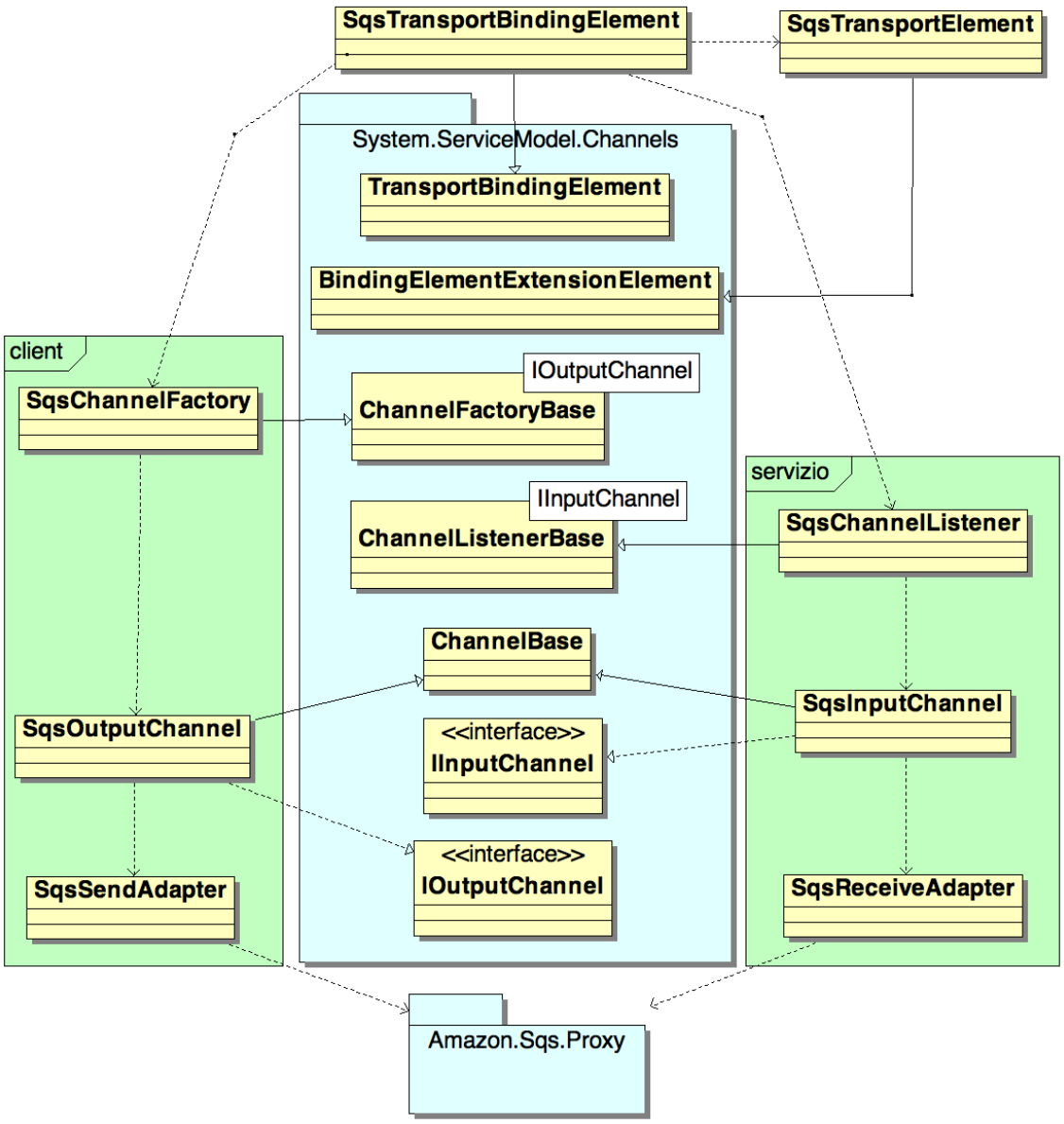

Figura 3.3: Insieme delle classi che implementano il trasporto. Il namespace System.ServiceModel.Channels contiene tutte le classi che compongono il channel layer di WCF, mentre il namespace Amazon.Sqs.Proxy contiene le classi che formano il proxy per comunicare con SQS.

In questo modo è possibile riutilizzare i binding element per costruire agevolmente nuovi binding. Per facilitare ulteriormente la costruzione di nuovi binding, esiste il custom binding, che è un binding che permette di impostare esplicitamente i binding element di cui è composto (anche da configurazione).

Il trasporto in esame è dunque un binding element che si occupa di inviare messaggi ad una coda di SQS e di recuperarli. Nel diagramma di figura 3.3 troviamo infatti una classe SqsTransportBindingElement che deriva da TransportBindingElement, una classe di WCF da cui derivano tutti i trasporti.

Il binding element non svolge direttamente i compiti che abbiamo spiegato. Compito del binding element è infatti istanziare due factory per la creazione dei canali, gli oggetti che implementano effettivamente la comunicazione. Le due factory sono:

- il channel factory. Viene usato per creare i canali con i quali i client possono comunicare con il servizio. Quando il channel factory termina la sua vita, anche i canali da lui creati sono distrutti: questo per evitare spreco di risorse;
- il channel listener. Viene usato per creare i canali usati dai servizi. Quando un channel listener termina la sua vita, i canali ancora attivi non sono distrutti ma possono continuare il lavoro che stavano facendo.

Anche nel trasporto in esame sono stati implementati il channel factory e il channel listener, rispettivamente SqsChannelFactory e SqsChannelListener. Queste due classi derivano da due classi generiche, rispettivamente ChannelFactoryBase<IOutputChannel> e ChannelListenerBase<IInputChannel>, che contengono già buona parte della logica necessaria e sono parametrizzate sul tipo di canale. (Notiamo che la sintassi C♯ per i tipi generici usa le parentesi angolari, mentre in Visual Basic .NET le parentesi angolari vengono usate per gli attributi)

Arriviamo infine ai canali. I canali sono gli oggetti che si occupano concretamente di lavorare sui messaggi per trasmetterli o per applicare loro un protocollo. I canali in WCF supportano vari pattern di scambio messaggi, che si riflette sull'interfaccia che implementano. Inoltre possono supportare o meno le sessioni, un meccanismo per correlare tra loro messaggi che condividono lo stesso contesto.

Nel trasporto il channel factory crea canali di tipo SqsOutputChannel, che implementano l'interfaccia IOutputChannel, usati nel client per inviare messaggi al servizio. Dualmente, il channel listener crea canali di tipo SqsInputChannel, che implementano l'interfaccia IInputChannel: questi canali si occupano di ricevere i messaggi che arrivano dai client. In questo caso il pattern di scambio messaggi è molto banale e senza uso di sessioni, rispecchiando il fatto che la comunicazione tramite SQS è one way.

Entriamo nei dettagli dell'implementazione dei due canali. SqsOutputChannel deve depositare i messaggi su una coda di SQS, mentre SqsInputChannel li deve recuperare dalla coda. Per comunicare con SQS, nel namespace Amazon.Sqs.Proxy è presente un proxy generato con i tool di WCF a partire dalla descrizione del servizio. La logica per comunicare con SQS non si trova nei canali ma è stata inserita in due classi apposite:

- SqsReceiveAdapter si occupa di recuperare i messaggi presenti su una coda facendo polling;

- SqsSendAdapter si occupa di inviare messaggi ad una coda.

Questo perché l'interazione con SQS è un po' articolata e i canali svolgono anche altre funzioni come la codifica dei messaggi. In questo modo i canali sono più facili da mantenere.

Le impostazioni sulla coda da utilizzare, sulle credenziali per accedere a SQS e sulla dimensione massima dei messaggi vengono impostate a livello di binding element e vengono propagate alle factory e ai canali. Queste impostazioni possono essere fornite da codice o tramite file di configurazione.

Per supportare l'ultimo scenario, è presente la classe SqsTransportElement, derivata da BindingElementExtensioneElement, che modella una sezione del file di configurazione per configurare SqsTransportBindingElement. La classe contiene una serie di proprietà usate per configurare il binding element e dei metodi per applicare la configurazione specificata ad un nuovo binding element o ad uno già esistente. Le proprietà sono decorate con l'attributo [ConfigurationProperty] (in C♯ gli attributi si indicano tra parentesi quadre), per indicare che corrispondono a proprietà di configurazione che possono essere specificate tramite file di configurazione. La presenza di questi attributi fa si che le impostazioni specificate dall'utente nel file di configurazione siano ricopiate automaticamente nelle proprietà di un'istanza di SqsTransportElement e succesivamente usate per configurare un'istanza di SqsTransportBindingElement.

Capitolo 4

Strumenti

4.1 Service Oriented Architecture

Service Oriented Architecture (SOA) è un paradigma per l'organizzazione e l'utilizzo di capacità distribuite che possono essere sotto il controllo di diverse entità. SOA promuove il riuso, l'interoperabilità e la facile evoluzione dei sistemi.

Come lascia trasparire l'acronimo, SOA individua nella soluzione una serie di servizi. Un servizio è un meccanismo per fornire capacità a chiunque ne faccia richiesta. Un servizio espone all'esterno una descrizione che indica le capacità fornite e il modo per utilizzarle. La comunicazione tra un client ed un servizio avviene tramite scambio di messaggi.

Per approfondire cosa sia SOA e quali siano le sue peculiarità, ci serviamo dei *four tenets of service orientation* (i quattro principi dell'orientamento ai servizi) che sono stato enunciati da Don Box (vedi [2] e [4]):

1. *boundaries are explicit*;
2. *services are autonomous*;
3. *services share schema and contract, not class*;
4. *service compatibility is based upon policy.*

Boundaries are explicit (i confini sono espliciti) Questo principio copre vari tipi di confini:

- *il confine tra l'implementazione del servizio e la sua descrizione* (il modo in cui questo appare a chi lo vuole usare). La descrizione è l'unica parte visibile del servizio, mentre l'implementazione è nascosta ed indipendente. In nessun modo la descrizione deve dipendere da dettagli di implementazione;
- *il confine tra il client ed il servizio*, che nella maggior parte dei casi comunicano tra loro in rete (con tutte le problematiche annesse). In alcuni paradigmi si è cercato di trattare risorse remote come fossero risorse locali (si pensi per esempio ai sistemi ad oggetti distribuiti). Questo approccio nell'immediato può sembrare vantaggioso, perché

abbiamo un solo modo per fare le cose in locale e in remoto. In realtà ciò è dannoso, perché non risultano in modo evidente i problemi della comunicazione in rete (riportati nella sezione 2.3) e - ancora peggio - in questo modo facciamo fatica ad affrontarli. Bisogna quindi trattare in modo diverso le risorse locali e le risorse remote;

- *i confini determinati da chi possiede i servizi che vogliamo usare.* SOA infatti prevede che i servizi che utilizziamo possano essere anche forniti da altri. Dobbiamo essere consci che usare servizi altrui significa rispettare le condizioni a cui sono resi disponibili (vedi quarto principio).

SOA riconosce questi confini prevedendo che client e servizio comunichino tra di loro tramite scambio di messaggi:

- non viene imposta nessuna tecnologia o paradigma per la realizzazione del client e del servizio. A seconda delle tecnologie impiegate l'elaborazione dei messaggi (invio, ricezione, esecuzione logica) saranno implementati nel modo più conveniente;
- è esplicito che comunichiamo tramite rete. Siamo quindi più consci che contattare un servizio impiega più tempo che invocare un metodo locale, che i messaggi possono venire persi e che il servizio che vogliamo contattare potrebbe non rispondere;
- le policy del servizio che vogliamo contattare si riflettono anche su come dobbiamo comporre i messaggi (se devono essere crittati per esempio) e in che modo trasmettiamo i messaggi.

Services are autonomous (i servizi sono autonomi) Un servizio è progettato per funzionare in qualsiasi contesto viene posto, senza assumere (implicitamente o esplicitamente) un particolare scenario in cui verrà usato.

Questo perché in SOA la soluzione si ottiene usando una serie di servizi. I servizi usati possono essere posseduti da differenti entità (alcuni servizi propri ed altri esterni) e possono cambiare nel tempo per adattarsi a nuove necessità. Se un servizio facesse assunzioni sul modo in cui verrà utilizzato (chi lo usa, il contenuto dei messaggi che riceve, ...), le sue possibilità di riuso sarebbero molto basse ed inoltre sarebbe più probabile che vada in errore quando riceve messaggi non semanticamente corretti.

Per esempio, il servizio di trasmissione log non può assumere che tutte le entry che gli arrivano abbiano l'identificativo valido. Questa assunzione potrebbe sembrare lecita dato che l'unico utilizzatore creato (la libreria client) invia sempre entry con identificativi validi. Il servizio deve invece fare il controllo perché in futuro potrebbero essere creati nuovi client che potrebbero usarlo in modo non corretto.

Services share schema and contract, not class (i servizi condividono lo schema e il contratto, non le classi) Il terzo e il quarto principio parlano di come sono fatti i due pezzi di cui è composta la descrizione di un servizio, l'interfaccia e le policy.

Un servizio espone un'*interfaccia* che specifica:

- le funzionalità che offre (contract);
- il formato dei messaggi che può ricevere ed inviare (schema). I messaggi ricevuti dal servizio richiedono l'esecuzione di una particolare funzionalità a partire da alcuni dati (che possiamo considerare come argomenti). In risposta all'esecuzione il servizio può restituire un messaggio con i risultati.

Questa interfaccia è espressa in un linguaggio adatto ad essere processato automaticamente.

L'enunciazione del principio evidenzia uno scoglio con cui si scontrano coloro che stanno imparando SOA. Le prime volte si è infatti portati a pensare che all'interno dei messaggi vengano trasportati oggetti, quindi entità che incapsulano logica e dati. L'approccio sembra funzionare: gli oggetti vengono creati nel client, vengono manipolati in modo corretto tramite i loro metodi, poi vengono trasmessi al servizio per eseguire una data funzionalità.

Si capisce che l'approccio non va bene. Il problema è rappresentato dalle classi: queste devono essere condivise sia dal client che dal servizio (perché contengono logica) e così facendo violiamo il primo tenet, esponendo dettagli di implementazione del servizio. Perdiamo anche l'interoperabilità tra client e servizio, dato che una classe è specifica per un dato linguaggio e/o piattaforma.

In SOA i messaggi trasportano quindi solo dati ed indicano quale funzionalità deve essere eseguita, la logica è contenuta solo nel servizio. Sarà cura del servizio controllare che i messaggi siano ben formati e validi prima di eseguire la funzionalità.

L'interfaccia dovrebbe rimanere stabile nel tempo, ma questo nella realtà non è possibile. Grazie al modo di specificare l'interfaccia, è possibile applicare una serie di pattern che ci permettono di progettare un'interfaccia che possa essere aggiornata mantenendo la retrocompatibilità, in modo molto più efficace rispetto ad altri paradigmi. Questo è molto importante, in quanto costringere ad aggiornare i client ad ogni aggiornamento dell'interfaccia minerebbe il riutilizzo dei servizio. Teniamo inoltre presente che non tutti i consumatori del servizio potrebbero essere sotto il nostro diretto controllo.

Service compatibility is based upon policy (la compatibilità dei servizi si basa sulle policy) Un servizio espone insieme all'interfaccia anche una serie di *policy*, che indicano i vincoli sotto i quali il servizio opera. Per esempio, i vincoli possono indicare in che modo il servizio comunica con i consumatori, a che indirizzo si trova il servizio o in che modo effettua l'autenticazione.

Il fatto che queste policy siano esposte nella descrizione del servizio fa si che siano esplicite e che possano essere analizzate in automatico. Ad esempio, tool che analizzano la descrizione di un servizio possono capire se le comunicazioni sono crittate o meno, evitando che il programmatore debba capirlo andando a consultare documenti altrove.

Mentre una interfaccia è progettata con la speranza di rimanere stabile, le policy sono soggette ad essere aggiornate nel tempo.

4.1.1 Web Service

Il modo più comune per implementare un'architettura di tipo SOA è quella di impiegare i Web Service. I Web Service permettono di applicare SOA utilizzando tecnologie standard e che assicurano interoperabilità tra servizi implementati da aziende diverse su piattaforme diverse.

Le tecnologie principali dei Web Service sono:

- *XML Infoset*. E' il linguaggio base con il quale sono definite tutte le altre tecnologie. XML Infoset è più astratto rispetto ad XML e non impone una specifica rappresentazione delle informazioni. Questo permette di scegliere la rappresentazione più adatta a seconda dello scenario;
- *SOAP*, il protocollo attraverso il quale client e servizi si scambiano i messaggi. SOAP è stato progettato in modo tale da essere estensibile senza dover ridefinire lo standard. Il nome SOAP era inizialmente un acronimo che stava per Simple Object Access Protocol. Questo acronimo non rappresenta più lo scopo del protocollo, per cui SOAP non è stato più espanso;
- *WSDL* (Web Service Description Language), un dialetto XML che permette di definire l'interfaccia e le policy del servizio. Anche WSDL è pensato per poter essere esteso ed esprimere cose non previste (nuovi protocolli, nuove modalità di crittazione, ...)

Con il tempo sono fioriti una serie di standard per raggiungere gli obiettivi più disparati, che vengono indicati collettivamente con la sigla WS-* (dato che i loro nomi iniziano appunto con WS-). Questi standard sono creati da diverse aziende ed organizzazioni. Ne citiamo alcuni a titolo di esempio:

- *WS-Addressing*, che indica in che modo devono essere specificate le informazioni sul destinatario e sul mittente di un messaggio;
- *WS-Security*, che aiuta a rendere più sicure le comunicazioni;
- *WS-ReliableMessaging*, che permette di spedire in modo affidabile i messaggi, assicurando per esempio che i messaggi siano processati dal servizio nello stesso ordine in cui sono stati inviati dal client.

Nel 2002 è nato un ente, Web Services Interoperability Organization (WS-I), il cui obiettivo è definire dei profili, ciascuno dei quali indica gli standard che i Web Service dovrebbero usare per raggiungere in modo efficace degli obiettivi (interoperabilità, sicurezza, ...). Nei profili sono contenute anche delle indicazioni di uso che dovrebbero sopperire alle ambiguità degli standard adoperati.

I Web Service hanno un accento molto marcato sull'interoperabilità tra servizi sviluppati su piattaforme differenti e da aziende differenti. Un sistema che copre solo una intranet e non è aperto all'esterno può comunque essere SOA; risulta comunque evidente che SOA si avvantaggia della presenza e dell'utilizzo di standard aperti e riconosciuti da tutti.

Ribadiamo il fatto che i Web Service sono uno dei tanti modi per fare SOA, non l'unico.

4.2 Amazon Simple Queue Service e Software as a Service

Per disaccoppiare la comunicazione tra client e servizio di trasmissione log è stato usato Amazon Simple Queue Service (SQS), un Web Service accessibile da Internet che offre code scalabili, ad alta disponibilità ed economiche. SQS è un servizio peculiare che abbraccia la filosofia Software as a Service (SaaS).

Amazon ha investito notevoli risorse nello sviluppo di una infrastruttura adeguata a supportare le operazioni del suo sito di e-commerce. Dal 2006 Amazon ha cominciato ad offrire a terzi l'uso di questa infrastruttura. Ciò è stato fatto offrendo una serie di Web Service, tra cui SQS, che esponessero all'esterno le capacità dell'infrastruttura. Dato il traffico che Amazon deve sostenere, SQS offre ottime garanzie di alta disponibilità e di scalabilità.

SQS è interoperabile: è infatti un Web Service implementato usando tecnologie e linguaggi standard.

SQS può essere confrontato con i prodotti di Message Oriented Middleware, cioè quei prodotti che permettono lo scambio di messaggi tra applicazioni tramite l'uso di code. Per essere usati, questi prodotti devono essere acquistati ed installati in un server apposito. SQS è invece un servizio già attivo fornito su sottoscrizione. Basta infatti registrarsi presso Amazon ed richiedere l'accesso al servizio, poi possiamo subito cominciare ad usarlo. Ogni inizio mese Amazon ci accrediterà una spesa corrispondente al traffico generato e al numero di messaggi scambiato. Il costo del servizio è molto basso: nel periodo maggio-luglio 2007 l'Autore ha speso circa 20 centesimi di dollaro per aver trasferito circa 1000 messaggi e 3 MB.

SQS è un esempio di software che può essere classificato come Software as a Service (SaaS). SaaS è un paradigma che si contrappone al modello usuale di vendita ed utilizzo delle applicazioni, che prevede l'acquisto delle licenze e l'installazione del prodotto su hardware dell'azienda. SaaS prevede invece che il software venga ospitato in una locazione centrale dal fornitore, venga fornito a consumo o su sottoscrizione e venga usato dagli utenti tramite Internet.

Da notare come nell'acronimo la parola servizio non ha la stessa accezione che ha in SOA, ma indica semplicemente un'attività che viene svolta per conto terzi. SaaS non è legato a SOA ma è invece un paradigma che astrae dalla particolare architettura. Infatti possono essere considerati SaaS servizi online come Gmail, Hotmail, del.icio.us, Google Reader, ... In questa trattazione SaaS verrà discusso per applicazioni destinate alle aziende e non per applicazioni consumer.

Cerchiamo di capire in dettaglio le differenze tra SaaS e il modello classico di concepire il software (già parzialmente delineate quando abbiamo parlato di SQS):

- non è necessario installare il prodotto, in quanto questo è già reso disponibile e funzionante dal fornitore. Non c'è quindi bisogno che l'azienda compri hardware per far girare il prodotto o assuma personale per l'installazione, la configurazione e la manutenzione del prodotto. Queste incombenze e questi costi passano al fornitore.

  Ciò è particolarmente importante per la parte server di un prodotto, che è quella che richiede risorse maggiori per poter funzionare (server dedicato, personale esperto, ...). Questo punto si applica meno ai client che

usano il server, dato che un client non ha bisogno di grandi risorse e gira sul computer del dipendente, che per forza deve essere acquistato. I client possono essere quindi applicazioni client da installare sui computer dell'azienda oppure applicazioni Web.

- i prodotti SaaS sono solitamente disponibili su sottoscrizione o a consumo. Il costo del software non è più fisso come nel modello tradizionale (il costo della licenza fatto una sola volta e da ammortizzare) ma diventa variabile, dipendente o dal tempo di utilizzo o dal consumo fatto (banda, numero di messaggi trasmessi, ...) e quindi maggiormente prevedibile a priori. Si può dire che con SaaS si paga solo nella misura in cui si usufruisce;

- il costo di un prodotto SaaS è più basso di un prodotto tradizionale. Questo è dovuto a due fattori:

  - l'utente non ha più i costi dell'hardware e del personale per mantenere il prodotto (come detto nel primo punto);
  - l'infrastruttura del fornitore può essere vista come un consolidamento delle varie infrastrutture che avrebbero dovuto avere i vari utenti nel modello tradizionale. Il fornitore dovrà quindi avere un'infrastruttura grossa; i costi per la sua creazione e manutenzione vengono però divisi tra tutti gli utenti del prodotto.

- i prodotti SaaS sono accessibili da un'utenza più ampia. La fornitura centralizzata del prodotto fa si che ci sia economia di scala, per cui all'aumentare degli utenti il costo del software può diminuire o a parità di costo possono migliorare le condizioni a cui viene erogato. Questo fa sì che utenti che prima non avrebbero considerato l'acquisto del prodotto con il modello tradizionale perché troppo costoso (basti pensare ai CRM o agli ERP) ora hanno le risorse per poterseli permettere, allargando il potenziale mercato del fornitore.

SaaS ha anche alcuni aspetti suscettibili di critica:

- se la connettività ad Internet è assente (viene a mancare o lavoriamo in un luogo in cui non possiamo accedere a Internet), non siamo in grado di usare le applicazioni. Questo problema è mitigato tramite l'uso di client che funzionino anche in mancanza di connessione (usando solitamente una cache locale ed accodando le richieste ai servizi). A titolo di esempio, gli smart client e i framework come Google Gears supportano l'uso offline;

- l'adesione a SaaS prevede solitamente la presenza di un unico servizio che è progettato per servire e memorizzare i dati di più utenti (architettura multitenant). Nella progettazione del servizio bisogna quindi garantire che un utente possa accedere sempre e solamente ai suoi dati. Questa rassicurazione può però essere non sufficiente per alcuni utenti particolarmente attenti alla privacy e alla sicurezza dei propri dati;

- nel caso di servizi che memorizzano dati per conto dell'utente (come Amazon Simple Storage Service o il CRM online di Salesforce.com) ci sono situazioni delicate che si verificano qualora l'utente voglia cambiare servizio o nel caso in cui il fornitore cessi l'attività. In entrambi i casi non è sempre

chiaro come i dati memorizzati presso il servizio possano essere recuperati affinché possano essere impiegati in modo opportuno (per esempio per essere gestiti in un altro servizio analogo).

4.3 Windows Communication Foundation

Windows Communication Foundation (WCF) è una tecnologia contenuta nel .NET Framework 3.0 che serve per costruire applicazioni distribuite orientate ai servizi. WCF è stato progettato con i seguenti obiettivi:

- fornire un modo semplice per realizzazione applicazioni SOA;
- essere estensibile in modo da supportare agevolmente nuovi scenari;
- unificare le tecnologie Microsoft esistenti per realizzare applicazioni distribuite;
- utilizzare per la maggior parte tecnologie standard, tra cui WSDL, SOAP e gli standard WS-*;
- essere performante.

Nella nostra trattazione ci concentreremo sui primi due punti.
WCF è composto da due parti, il service model e il channel layer.

Service model Ci permette di implementare i servizi e i client. La caratteristica principale del service model è che non diamo un'implementazione completa e diretta di tutti i meccanismi necessari; forniamo invece un modello che a run time verrà usato per istanziare le classi opportune del channel layer. L'idea di fondo di questo approccio è che il modello è più semplice da manipolare rispetto ad un'implementazione diretta e ci permette di concentrarci sulle cose veramente importanti del servizio, ovvero l'interfaccia, la logica e le policy.

Per implementare o consumare un servizio ci servono tre cose, indicate nella terminologia WCF con l'acronimo ABC:

- *Address*, ovvero l'indirizzo al quale il servizio è disponibile. Questo è l'aspetto più semplice da specificare. Sebbene si possa specificare da codice, il modo più conveniente di farlo è tramite il file di configurazione che ogni applicazione .NET prevede. Il file di configurazione è utile per influenzare il comportamento di un'applicazione senza ricompilare quest'ultima; un amministratore che voglia cambiare la locazione del servizio potrà farlo alterando il file di configurazione, senza l'intervento di un programmatore;
- *Binding*, ovvero come il servizio comunica con l'esterno. WCF mette a disposizioni varie classi, dette appunto binding, che implementano vari protocolli di comunicazione, a partire da binding interoperabili secondo i profili WS-I a binding proprietari che comunicano tramite TCP o named pipe. Come per l'indirizzo, il binding per un servizio può essere specificato da codice o da file di configurazione;

- *Contract*, ovvero l'insieme di funzionalità esposte dal servizio e il formato dei messaggi che il servizio può inviare e ricevere. WCF prevede vari modi per modellare il contratto: per semplicità, in questa sede parleremo del modo consigliato, impiegato per la realizzazione del prodotto dello stage.

  L'idea è quella di modellare il contratto del servizio usando le interfacce e le classi. Bisogna naturalmente avere un modo per indicare come tradurre classi ed interfacce in funzionalità e formato dei messaggi. A tal scopo le interfacce e le classi vengono decorati con attributi. Notiamo come la traduzione non venga fatta in modo implicito, ma venga guidata esplicitamente dagli attributi.

  Gli attributi in .NET non sono altro che classi che derivano da una classe chiamata Attribute e che servono per associare metadati a varie parti di un programma. Un attributo può essere associato ad un intero programma, ad una classe o ad un metodo. Questi attributi possono essere ispezionati a runtime mediante reflection. In Visual Basic .NET, un attributo si indica mettendo il nome dell'attributo tra parentesi angolari, ad esempio <Deprecated>, mentre in C♯ il nome va tra parentesi quadre, ad esempio [Deprecated].

  Supponiamo di voler definire le funzionalità di un servizio. Si comincia con il creare un'interfaccia, che va decorata con l'attributo <ServiceContract> per indicare che è il contratto di un servizio WCF. All'interno dell'interfaccia si definiscono vari metodi. Quelli che vogliamo esporre come funzionalità all'esterno devono essere decorati con l'attributo <OperationContract>.

  Ogni funzionalità ha, nel caso più generale, un messaggio con la quale la si invoca ed un messaggio che contiene l'esito dell'invocazione. Il messaggio di invocazione viene modellato in base agli argomenti previsti dal metodo marcato come <OperationContract>, mentre il messaggio restituito è modellato in base al risultato restituito dal metodo.

  Gli argomenti ed il risultato possono usare sia tipi predefiniti di .NET, che WCF sa come tradurre in automatico, sia classi definite da noi. Queste ultime devono essere decorate opportunamente per indicare a WCF come devono essere serializzate nei messaggi. In particolare, le classi devono essere decorate con l'attributo <DataContract> e all'interno di queste classi ogni campo o proprietà (anche privato) che deve essere trasportato deve essere marcato con <DataMember>. Notiamo che quest'ultimo attributo non può essere applicato a metodi: i messaggi sono infatti solo dati e non contengono logica. Il discorso fatto è naturalmente ricorsivo a partire dai campi e proprietà <DataMember> di una classe <DataContract>.

  Ognuno degli attributi citati ha delle proprietà che permette di configurare come deve avvenire la traduzione. Per esempio, è possibile cambiare il nome delle operazioni o il loro namespace. Sono inoltre presenti altri attributi che aiutano a modellare altri aspetti del contratto.

Specifichiamo meglio l'implementazione di un servizio e di un client:

- la logica del servizio va inserita all'interno di una classe che implementa l'interfaccia decorata con <ServiceContract>. I metodi manipoleranno direttamente le classi marcate come <DataContract>. Per rendere attivo il servizio, è sufficiente istanziare la classe tramite alcune classi helper di WCF. Verrà letta la configurazione ed in base agli attributi verranno messe in piedi le classi del channel layer necessarie;
- l'utilizzo di un servizio è ancora più banale. E' sufficiente avere la descrizione del servizio in linguaggio WSDL e darla in pasto ad un tool apposito fornito con WCF. Questo tool genererà un modello secondo l'ABC descritto sopra, creando quindi una serie di classi che possiamo usare per comunicare con il servizio. Il modello generato dal tool si indica con il termine proxy.

Channel layer E' composto da una serie di classi che permettono la comunicazione con il servizio. Queste classi non vengono utilizzate direttamente dal programmatore, ma vengono istanziate automaticamente in base a quanto modellato nel service model.

Il channel layer assolve alle seguenti funzioni:

- serializza e deserializza opportunamente i messaggi in base alla specifica data nel service model;
- trasporta i messaggi tra client e servizio nel modo specificato da codice o da configurazione;
- per ogni messaggio ricevuto lato servizio, seleziona il servizio destinatario del messaggio ed esegue l'operazione associata.

Si può notare come il programmatore si occupa del contratto e della logica del servizio, mentre tutti i dettagli su come interagiscono client e servizio vengono risolti da WCF.

Sia il service model che il channel layer possono essere estesi:

- si può intervenire su come i messaggi vengono serializzati e deserializzati (a vari livelli);
- si può cambiare il modo in cui viene gestito il ciclo di vita del servizio;
- si può agire sul meccanismo con il quale vengono selezionati i servizi e le operazioni a partire da un messaggio;
- si possono gestire nuovi protocolli;
- si possono introdurre nuovi trasporti. Un trasporto è un componente che stabilisce come i messaggi vengono effettivamente trasportati tra client e servizio.

Glossario

Alta disponibilità Caratteristica di un sistema software che garantisce che quest'ultimo, in un dato periodo di tempo, possa essere non operativo per al massimo una percentuale di tempo fissata a priori.

Amazon Simple Queue Service (SQS) Web Service offerto da Amazon che offre un servizio di code scalabile e ad alta disponibilità.

Attributo In .NET insieme di metadati che possono essere associati ad un elemento del codice (solitamente un metodo, una classe o un programma) e che possono essere recuperati a run time.

Binding Classe di WCF che specifica in che modo un client comunica con un servizio.

Binding element Parte di un binding che si occupa di implementare un particolare aspetto della comunicazione tra client e servizio. Svolge il suo compito tramite la creazione di due factory, il channel factory e il channel listener.

Canale In WCF classe che si occupa concretamente di applicare un protocollo ad un messaggio o di trasmettere un messaggio.

Channel factory Classe di WCF deputata a creare i canali con cui i client possono comunicare con un dato servizio.

Channel layer In WCF insieme di classi che implementano effettivamente la comunicazione con un servizio, instanziate automaticamente a partire dal service model.

Channel listener Classe di WCF deputata a creare i canali con cui un servizio può rispondere ai suoi client.

Coda (di messaggi) Nell'ambito delle architetture software, componente atto allo scambio di messaggi tra una serie di consumatori ed una serie di produttori che garantisce che nessun messaggio trasmesso venga perso.

Contratto Nella terminologia del service model di WCF, elenco di funzionalità offerte da un servizio.

Entry Informazione riguardante un evento. Solitamente fa parte di un log.

Globally Unique Identifier (GUID) Valore usato per identificare univocamente una entità in un dato contesto. Un GUID è generato in modo tale che sia estremamente improbabile che lo stesso valore venga generato due volte, anche su macchine diverse.

Log Collezione di informazioni relative ad una serie di eventi accaduti nel tempo. E' composto da una serie di entry.

Message Oriented Middleware Strato software che abilita la comunicazione tra applicazioni tramite l'uso di code di messaggi.

One way Detto di una comunicazione tra un client e un servizio in cui è solo il client a trasmettere messaggi al servizio. Quest'ultimo si limita ad eseguire la funzionalità richiesta senza restituire alcun messaggio al client.

Proxy In WCF modello di un servizio che viene usato dal client per la comunicazione. Viene solitamente generato in automatico a partire dalla descrizione del servizio.

Scalabilità Caratteristica di un sistema software che assicura che all'aumentare del carico la reattività del sistema non cambi.

Service model In WCF insieme di classi, attributi e configurazione tramite i quali è possibile implementare servizi e client che consumano servizi.

Service Oriented Architecture (SOA) Paradigma per l'organizzazione e l'utilizzo di capacità distribuite che possono essere sotto il controllo di diverse entità (definizione presa da [1]).

Servizio Nell'ambito di SOA, entità che fornisce l'accesso a determinate capacità attraverso una determinata interfaccia e determinati vincoli (definizione liberamente presa da [1]).

Smart client Un'applicazione desktop che usa servizi per svolgere la maggior parte delle sue funzioni. E' in grado di operare in modalità disconnessa ed è pensato per essere semplice da installare ed aggiornare.

Software as a Service (SaaS) Paradigma per il quale il software non viene concepito come prodotto da comprare e installare, ma un servizio reso disponibile dal fornitore tramite Internet su sottoscrizione.

Trasporto Componente software che permette di estendere WCF introducendo una nuova modalità di trasmissione dei messaggi tra client e server.

Web Service Sistema software progettato per supportare interazioni interoperabili tra macchine che comunicano in rete (definizione presa da [5]).

Windows Communication Foundation (WCF) Tecnologia Microsoft per realizzare applicazione distribuite orientate ai servizi. Fa parte del .NET Framework 3.0.

Sigle e acronimi

ABC Address, Binding, Contract

CRM Customer Relationship Management

ERP Enterprise Resource Planning

GUID Globally Unique Identifier

HTTP Hypertext Transfer Protocol

OASIS Organization for the Advancement of Structured Information Standards

SaaS Software as a Service

SOA Service Oriented Architecture

SQS Amazon Simple Queue Service

TCP Transmission Control Protocol

UTC Coordinated Universal Time

W3C World Wide Web Consortium

WCF Windows Communication Foundation

WS-I Web Services Interoperability Organization

WSDL Web Service Description Language

XML Extensible Markup Language

Bibliografia

[1] OASIS. Reference Model for Service Oriented Architecture. 2006. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm. Recuperato il 02/09/2007

[2] Don Box. Code Name Indigo: A Guide to Developing and Running Connected Systems with Indigo. 2004. http://msdn.microsoft.com/msdnmag/issues/04/01/Indigo/default.aspx. Recuperato il 28/07/2007

[3] Jim Waldo, Geoff Wyant, Ann Wollrath, Sam Kendall. A Note on Distributed Computing. 1994. http://research.sun.com/techrep/1994/abstract-29.html. Recuperato il 10/08/2007

[4] Microsoft. SOA in the Real World. 2007. http://tinyurl.com/3cqv5p. Recuperato il 10/08/2007

[5] W3C. Web Service Architecture. 2004. http://www.w3.org/TR/ws-arch. Recuperato il 01/08/2007

[6] Home page di Amazon SQS. http://aws.amazon.com/sqs. Recuperato il 01/08/2007

[7] Frederick Chong e Gianpaolo Carraro. Architecture Strategies for Catching the Long Tail. 2006. http://msdn2.microsoft.com/en-us/library/aa479069.aspx. Recuperato il 13/07/2007

[8] Ron Jacobs, Frederick Chong e Gianpaolo Carraro. ARCast - Software as a Service. http://channel9.msdn.com/ShowPost.aspx?PostID=217250. Recuperato il 16/07/2007

[9] Craig McMurtry, Marc Mercuri, Nigle Watling e Matt Winkler. Windows Communication Foundation Unleashed. 2007. Sams Publishing. ISBN 0-672-32948-4

[10] MSDN Library. Documentazione di Windows Communication Foundation. http://msdn2.microsoft.com/en-us/library/ms735119.aspx. Recuperato il 06/09/2007