Axhenti alessandro

TPSI

Web service

RPC e Middleware Correlati

L’RPC è il fondamento della maggior parte delle piattaforme middleware attualmente disponibili. Vediamo nel dettaglio la sua evoluzione.

Contesto Storico

L’RPC è stato introdotto nei primi anni ‘80 da Birrell e Nelson come parte del loro lavoro sull’ambiente di programmazione Cedar [28]. L’idea era quella di permettere la chiamata trasparente di procedure situate su altre macchine. Questo meccanismo è diventato rapidamente la base per la costruzione di sistemi a 2 livelli, ereditando molte convenzioni e assunzioni proprie dell’RPC.

L'RPC ha fornito un approccio chiaro alla distribuzione delle applicazioni, utilizzando il concetto familiare di procedura, evitando la necessità di cambiare il paradigma di programmazione. Con il tempo, si è evoluto da semplici librerie a una vera e propria piattaforma middleware. Oggi, l'RPC è alla base della maggior parte dei sistemi distribuiti, incluso il Web Services e l’integrazione delle applicazioni aziendali.

Come Funziona l'RPC

Lo sviluppo di un'applicazione distribuita con RPC segue una metodologia ben definita. Per semplicità, supponiamo di voler sviluppare un server che implementa una procedura da utilizzare in remoto da un singolo client.

Il primo passo consiste nel definire l'interfaccia della procedura. Questo viene fatto utilizzando un **linguaggio di definizione dell'interfaccia (IDL)**, che fornisce una rappresentazione astratta della procedura in termini di parametri di ingresso e di uscita. Questa descrizione IDL rappresenta la specifica dei servizi offerti dal server. Con questa descrizione in mano, si può procedere allo sviluppo del client e del server (Figura 2.2).

Il secondo passo consiste nella compilazione della descrizione IDL. Qualsiasi implementazione di RPC e qualsiasi middleware che utilizzi RPC o concetti simili fornisce un

compilatore di interfacce. Tipicamente, la compilazione dell'IDL produce:

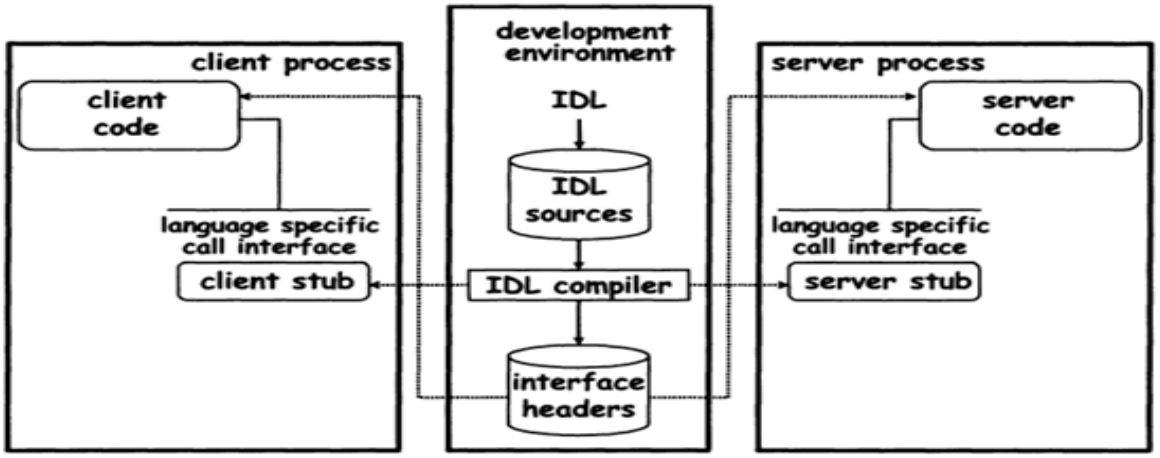


Fig. 2.2. Developing distributed applications with RPC

- **Stub del client**: ogni firma di procedura nell'IDL genera uno stub del client, un pezzo di codice da compilare e collegare al client. Quando il client chiama una procedura remota, in realtà esegue una chiamata locale allo stub, che si occupa di:

  localizzare il server (binding),

  formattare i dati (marshalling e serializzazione),

  comunicare con il server,

  ricevere la risposta e restituirla come risultato della chiamata.
  In altre parole, lo stub è un proxy per la procedura effettiva implementata sul server, facendo apparire la chiamata come una normale procedura locale.

- **Stub del server**: simile allo stub del client, ma lato server. Riceve l'invocazione dal client, esegue le operazioni di deserializzazione e unmarshalling, chiama la procedura implementata nel server e restituisce il risultato al client. Deve essere compilato e collegato al codice del server.
- **Template di codice e riferimenti**: in molti linguaggi di programmazione, le procedure devono essere definite in fase di compilazione. Il compilatore IDL genera i file ausiliari necessari (come i file di intestazione `.h` per C). I compilatori moderni generano anche template con la struttura base del server, lasciando agli sviluppatori solo il compito di implementare la logica della procedura.

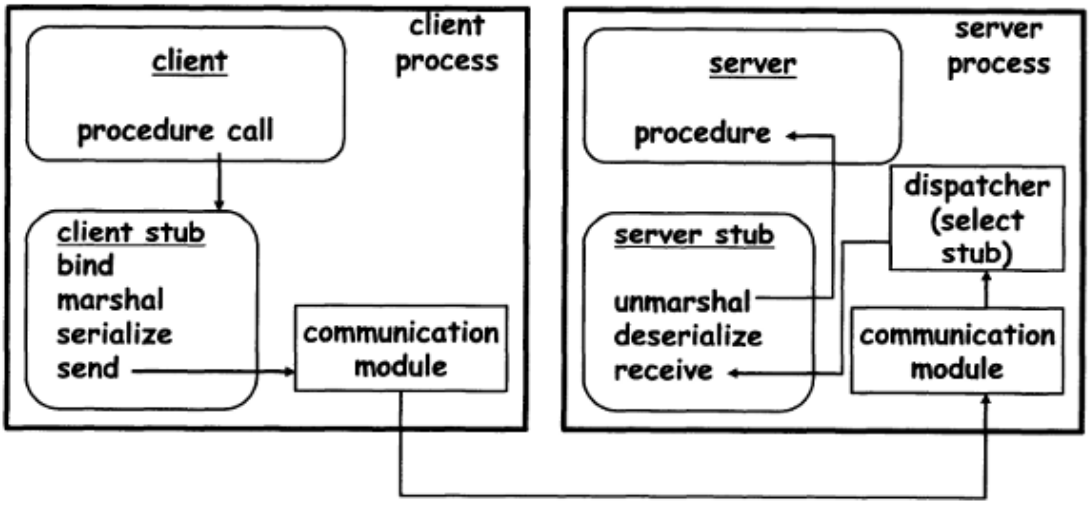


Fig. 2.3. Basic functioning of RPC

Binding nell'RPC

Per effettuare una chiamata RPC, il client deve prima **localizzare e collegarsi (binding) al server** che ospita la procedura remota. Esistono due tipi di binding:

- **Binding statico**: l'indirizzo del server (IP, porta, ecc.) è codificato nel client stub. Questo metodo è semplice ed efficiente, ma ha svantaggi:

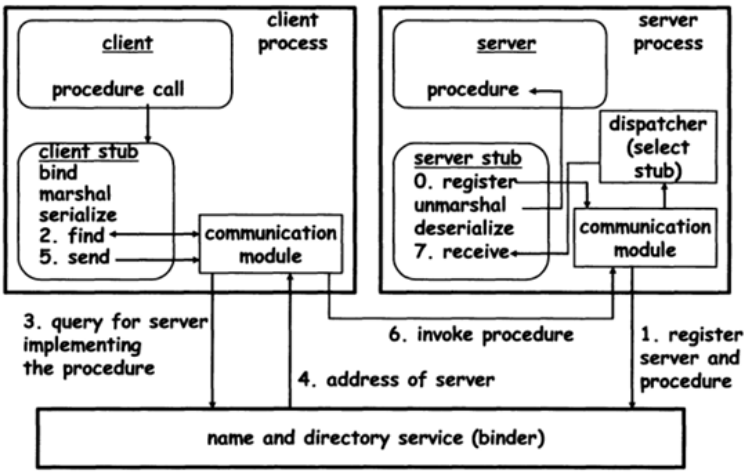
  Se il server cambia indirizzo, il client deve essere ricompilato.

  Non è possibile il bilanciamento del carico tra più server.

- **Binding dinamico**: usa un **server di directory** (detto anche binder) per risolvere gli indirizzi dei server in base alle firme delle procedure. Quando il client invoca una procedura remota, lo stub del client chiede al server di directory un server disponibile. Questo metodo è più flessibile e permette:

  Bilanciamento del carico tra server multipli.

  Ridondanza: se un server fallisce, le richieste vengono inoltrate a un altro server disponibile.



RPC e Eterogeneità

Uno degli obiettivi principali di RPC è gestire **sistemi eterogenei**. Poiché la gestione della distribuzione è delegata agli stub, è possibile sviluppare client e server su piattaforme diverse e con linguaggi diversi.

Per evitare di creare un numero eccessivo di stub specifici per ogni combinazione di piattaforme e linguaggi, si utilizza una **rappresentazione intermedia**. Il linguaggio IDL non solo definisce le interfacce, ma anche il formato intermedio per la comunicazione tra client e server.

Ad esempio, l'IDL di Sun, chiamato **XDR (External Data Representation)**, richiede che le chiamate siano codificate in oggetti da 4 byte, utilizzando codici ASCII. Questo garantisce che i dati possano essere trasferiti tra sistemi con architetture e rappresentazioni dati diverse.

Grazie a queste caratteristiche, RPC ha svolto un ruolo fondamentale nello sviluppo di **architetture client-server**, permettendo la comunicazione tra PC e server più potenti con sistemi operativi differenti (es. DOS per i client e UNIX per i server).

Estensioni di RPC

Il tradizionale RPC condivide molte delle caratteristiche di una chiamata di funzione locale. In primo luogo, utilizza un approccio sincrono e procedurale. L'esecuzione, compresa quella delle sottoprocedure, è sequenziale. Un client può avere una sola chiamata in sospeso alla volta ed è bloccato fino a quando la chiamata non restituisce un risultato. Sul server, possono essere utilizzati più thread per gestire diversi client contemporaneamente, ma ogni thread è dedicato esclusivamente a un singolo client. Fin dall'inizio, i progettisti si resero conto che le limitazioni di questo modello impedivano l'implementazione di tutte le forme di interazione necessarie in un sistema informativo distribuito. Di conseguenza, sono state implementate molte estensioni ai meccanismi di base di RPC. La motivazione dietro queste estensioni

deriva dagli stessi cambiamenti che hanno spinto verso l'adozione di architetture a tre livelli, ossia la crescente disponibilità di server con interfacce stabili e pubblicate, che ha stimolato la necessità di un'architettura e di un'infrastruttura per supportare l'integrazione dei server. Infatti, le estensioni di RPC hanno portato alla creazione di diverse piattaforme middleware, come monitor di transazioni (TP monitors), monitor di oggetti, sistemi di code o broker di messaggi. Qui discuteremo RPC asincrono (che è alla base del middleware orientato ai messaggi e dei broker di messaggi), mentre nel paragrafo 2.3.2 discuteremo RPC transazionale (come base per i TP monitors).

RPC asincrono è stata una delle prime estensioni a RPC per supportare chiamate non bloccanti. Permette a un client di inviare un messaggio di richiesta a un servizio senza aspettare la risposta. Il gestore di comunicazione restituisce immediatamente il controllo al programma client dopo l'invio della richiesta, quindi il thread del client non viene bloccato quando effettua una chiamata. Ciò gli consente di avere più chiamate in sospeso, continuando a fare altre operazioni.

Questa funzionalità è relativamente semplice da implementare cambiando il funzionamento dello stub del client. Invece di un solo punto di ingresso per invocare la procedura, lo stub fornisce due punti di ingresso: uno per invocare la procedura e uno per ottenere i risultati dell'invocazione. Quando il client chiama la procedura remota, lo stub estrae i parametri di input e restituisce immediatamente il controllo al client, che può quindi proseguire con l'elaborazione. Nel frattempo, lo stub esegue la chiamata al server e attende la risposta. Questo comportamento può essere facilmente ottenuto utilizzando thread nel client, senza che il programmatore debba esserne consapevole. Successivamente, il client effettua una seconda chiamata allo stub per ottenere i risultati. Se la chiamata è stata completata, lo stub avrà memorizzato i risultati in una struttura dati condivisa e il client li preleverà da lì. Se la chiamata non è ancora stata completata, il client riceverà un errore che indica che deve riprovare più tardi. Se c'è stato un errore con la chiamata stessa (timeout, errore di comunicazione, ecc.), il client riceverà un valore di ritorno che indica la natura del problema. In sostanza, gli stub eseguono in modo sincrono, ma forniscono l'illusione di un'esecuzione asincrona per i client e i server che li utilizzano.

La motivazione originale per l'interazione asincrona nei sistemi RPC era quella di mantenere la compatibilità con la funzionalità fornita dalle architetture a uno o due livelli. In particolare, le architetture a uno o due livelli tipicamente supportavano operazioni batch, che non possono essere implementate con RPC sincrono; RPC asincrono ha fornito un modo per implementare tali operazioni. In seguito, i progettisti hanno riconosciuto vantaggi aggiuntivi nell'RPC asincrono, ma hanno anche incontrato sfide per i programmatori. Per essere davvero utile, RPC asincrono ha necessitato di un'infrastruttura molto più sofisticata rispetto a quella fornita dagli stub. Ad esempio, poiché vengono mantenute meno informazioni sulla connessione e le richieste sono separate dalle risposte, può essere più difficile recuperare da un fallimento in RPC asincrono rispetto a RPC tradizionale. Questa infrastruttura è stata sviluppata come parte dei sistemi di code utilizzati nei TP monitors e si è successivamente evoluta in quelli che oggi conosciamo come broker di messaggi. Poiché inizialmente RPC non disponeva del supporto necessario, l'uso di RPC asincrono non era tanto diffuso come lo

è oggi. Riprenderemo questa questione in dettaglio quando discuteremo del middleware orientato ai messaggi (sezione 2.5) e dei broker di messaggi (sezione 3.2).

Infrastruttura Middleware di RPC: DCE

Anche senza considerare le estensioni al meccanismo base di RPC, RPC presuppone già un certo grado di infrastruttura sia per lo sviluppo che per l'esecuzione. Alcune di queste infrastrutture sono minime, in particolare nelle implementazioni di base di RPC, come il RPC di Sun. In altri casi, l'infrastruttura è piuttosto estesa, come nel caso del Distributed Computing Environment (DCE) fornito dall'Open Software Foundation (OSF) (Figura 2.5). DCE è ancora in uso oggi in molte piattaforme middleware e prodotti di integrazione aziendale. Poiché rappresenta una fase intermedia tra il client-server di base e le architetture a tre livelli complete, dedichiamo un po' di tempo qui per capire cosa offre DCE.

DCE è stato il risultato di un tentativo di standardizzare RPC—un tentativo che non ha avuto successo. DCE fornisce non solo una specifica completa di come dovrebbe funzionare RPC, ma anche un'implementazione standardizzata. L'obiettivo di DCE era fornire ai fornitori un'implementazione standard che potessero poi usare ed estendere secondo le necessità per i propri prodotti. Utilizzando la stessa implementazione di base di RPC, si sperava che i prodotti risultanti fossero compatibili. Molti ritengono che questa sia la ragione per cui DCE non è diventato uno standard accettato. Quando il gruppo di gestione degli oggetti ha proposto CORBA come standard per piattaforme distribuite (discusso più avanti in questo capitolo), aveva imparato da questo errore e ha imposto solo una specifica, non un'implementazione. Tuttavia, poiché molte implementazioni conformi allo standard CORBA non sono compatibili tra loro, OSF potrebbe aver avuto ragione.

La piattaforma DCE fornisce RPC e una serie di servizi aggiuntivi che sono molto utili quando si sviluppano sistemi informativi distribuiti. Questi servizi includono:

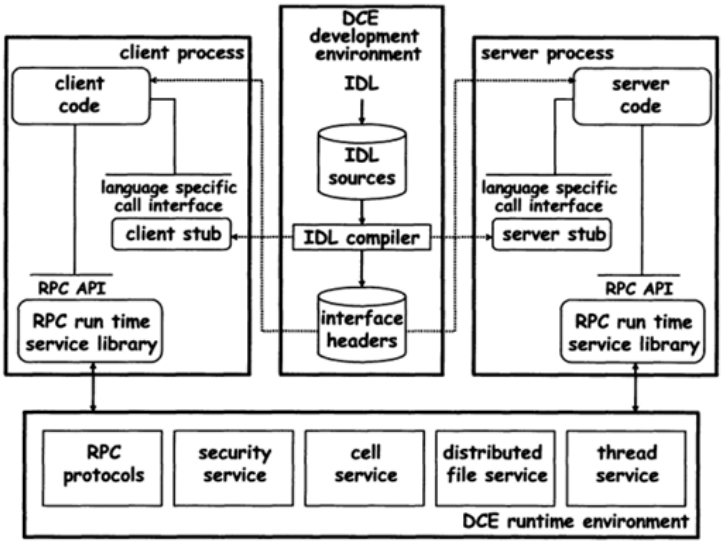


Fig. 2.5. The architecture of DCE

- **Servizio di directory delle celle**: La directory delle celle è un sofisticato server di nomi e directory utilizzato per creare e gestire domini RPC che possono coesistere sulla stessa rete senza interferire tra loro.
- **Servizio del tempo**: Il servizio del tempo fornisce meccanismi per la sincronizzazione degli orologi tra tutti i nodi.
- **Servizio di thread**: Come evidenziato nelle spiegazioni precedenti, il multi-threading è una parte importante dei sistemi RPC. DCE supporta thread e più processori con il servizio di thread.
- **Servizio di file distribuiti**: Il servizio di file consente ai programmi di condividere file dati in un ambiente DCE.
- **Servizio di sicurezza**: In un ambiente condiviso, come quelli gestiti da DCE, non è fattibile accettare chiamate di procedura da client arbitrari. Il servizio di sicurezza risolve questo problema fornendo comunicazioni autenticate e sicure.

Le caratteristiche architetturali principali di DCE sono mostrate nella Figura 2.5. Questa figura è un buon riassunto di come funziona RPC, così come di alcuni dei supporti necessari in ambienti reali. Il lettore attento noterà che questa figura apparirà nuovamente sotto altre forme mentre discuteremo altre forme di middleware. Questo non è un caso. La maggior parte delle piattaforme middleware descritte in questo e nel prossimo capitolo si basa su RPC e, in alcuni casi, sono estensioni e miglioramenti diretti del meccanismo RPC. Infatti, molte di queste piattaforme sono effettivamente costruite sopra piattaforme RPC, comprese quelle che apparentemente utilizzano paradigmi differenti (come broker di oggetti o middleware orientato ai messaggi).

WEB SERVICE PIU' MODERNI:

HTTP (Hypertext Transfer Protocol)

è il protocollo fondamentale per la comunicazione tra client e server nel web. È un protocollo di livello applicativo che definisce come i messaggi devono essere formattati, trasmessi e come il server deve rispondere a determinate richieste. HTTP è **stateless**, il che significa che ogni richiesta da parte di un client è indipendente e il server non conserva alcuna informazione riguardo alle richieste precedenti.

1. Richieste HTTP

Una **richiesta HTTP** è un messaggio inviato dal client (di solito un browser web o un'applicazione) al server per richiedere una risorsa, come una pagina web, un'immagine o una risorsa API. Una richiesta HTTP contiene vari componenti, e ognuno ha un ruolo specifico.

Struttura di una richiesta HTTP

1. **Linea di richiesta**:

La **prima riga** della richiesta è la "request line", che contiene:

- **Metodo HTTP**: Indica l'operazione da eseguire, come GET, POST, PUT, DELETE.
- **URI (Uniform Resource Identifier)**: L'indirizzo della risorsa a cui si sta accedendo, come `/home` o `/users/123`.
- **Versione HTTP**: Specifica la versione del protocollo HTTP, come HTTP/1.1 o HTTP/2.

2. **Header HTTP**:

Contiene metadati sul messaggio, come informazioni su tipo di contenuto, lingua, e altre direttive. Gli header forniscono dettagli aggiuntivi per il server per trattare correttamente la richiesta.

Alcuni esempi di header:

- **Host**: Specifica l'hostname del server (per esempio, `www.example.com`).
- **Accept**: Indica i tipi di contenuto che il client è disposto a ricevere, come `application/json` o `text/html`.
- **User-Agent**: Contiene informazioni sul browser o sull'applicazione che invia la richiesta.
- **Authorization**: Contiene credenziali per l'autenticazione (ad esempio, in base a un token di accesso).

3. **Body (opzionale)**:

Non tutte le richieste hanno un corpo. Il corpo della richiesta è usato per inviare dati al server (ad esempio, nei metodi POST o PUT). I dati possono essere in vari formati, come JSON, XML, o form-data.

Metodi HTTP

HTTP definisce vari metodi che il client può usare per interagire con le risorse:

- **GET**: Recupera una risorsa dal server (ad esempio, una pagina web o un file).
- **POST**: Invia dati al server per creare una nuova risorsa (ad esempio, inviare un modulo o creare un nuovo utente).
- **PUT**: Modifica o sostituisce una risorsa esistente.
- **DELETE**: Elimina una risorsa dal server.
- **PATCH**: Modifica parzialmente una risorsa.
- **OPTIONS**: Restituisce i metodi HTTP supportati dal server per una determinata risorsa.

2. Risposte HTTP

Una **risposta HTTP** è il messaggio che il server invia al client in risposta a una richiesta HTTP. Contiene un codice di stato che indica se la richiesta è stata completata con successo, insieme a un corpo che può contenere i dati richiesti.

Struttura di una risposta HTTP

1. **Linea di stato**:

   La **prima riga** della risposta è la "status line", che contiene:

   - **Versione HTTP**: La versione del protocollo, come HTTP/1.1 o HTTP/2.
   - **Codice di stato**: Un numero che indica l'esito della richiesta.
   - **Descrizione del codice di stato**: Una breve spiegazione del codice di stato.

2. A**lcuni esempi di codici di stato:**

   200 OK: La richiesta è andata a buon fine e la risposta contiene i dati richiesti.

   404 Not Found: La risorsa richiesta non è stata trovata.

500 Internal Server Error: Il server ha riscontrato un errore interno.

201 Created: La risorsa è stata creata con successo (usato spesso in risposta a una richiesta POST).

403 Forbidden: Il client non ha i permessi per accedere alla risorsa.

3. **Header HTTP**:

   Simile alla richiesta, la risposta contiene una serie di **header** che forniscono metadati sul messaggio, come il tipo di contenuto, la data di creazione, la lingua, e altreinformazioni.

   Alcuni esempi di header:

   - **Content-Type**: Indica il tipo di contenuto nel corpo della risposta (ad esempio, `text/html`, `application/json`).
   - **Content-Length**: La lunghezza del corpo del messaggio in byte.
   - **Cache-Control**: Specifica le politiche di caching per il contenuto.
   - **Location**: Indica l'URL di una risorsa creata in seguito a una richiesta POST (spesso usato con il codice 201).

4. **Body (opzionale)**:

   Il corpo della risposta contiene il contenuto richiesto dal client, come il codice HTML di una pagina, i dati JSON di una risorsa API, o una semplice immagine.

   Ad esempio, se una richiesta GET ha richiesto una pagina web, il corpo della risposta conterrà l'HTML della pagina.

3. Codici di Stato HTTP

I **codici di stato** nelle risposte HTTP sono suddivisi in cinque categorie principali, ognuna identificata dal primo numero del codice (es. 2xx, 4xx, ecc.):

- **1xx (Informational)**: Codici di stato informativi che indicano che la richiesta è stata ricevuta e il processo continua (ad esempio, 100 Continue).
- **2xx (Success)**: Indicano che la richiesta è stata elaborata con successo (ad esempio, 200 OK, 201 Created).
- **3xx (Redirection)**: Indicano che il client deve fare un'ulteriore azione (ad esempio, 301 Moved Permanently, 302 Found).

- **4xx (Client Error)**: Indicano errori causati dal client, come richieste malformate o risorse non trovate (ad esempio, 400 Bad Request, 404 Not Found).
- **5xx (Server Error)**: Indicano errori causati dal server (ad esempio, 500 Internal Server Error, 502 Bad Gateway).

SOAP

SOAP (Simple Object Access Protocol) è un protocollo di comunicazione che utilizza XML per inviare messaggi tra client e server. Ecco come funziona, concentrandoci sugli aspetti principali senza entrare in dettagli inutili:

1. Messaggi SOAP

Un messaggio SOAP è un documento XML che contiene tutte le informazioni necessarie per la comunicazione tra client e server.

Un messaggio SOAP è composto da tre elementi principali:

> Envelope (Involucro): Definisce i limiti del messaggio. È l'elemento radice del messaggio.
>
> Header (Intestazione): Contiene informazioni opzionali per il trattamento del messaggio, come credenziali o informazioni di routing.
>
> Body (Corpo): Contiene i dati reali del messaggio, ossia la richiesta o la risposta del servizio.

2. Invio di un messaggio SOAP

- Un client invia un messaggio SOAP al server tramite un protocollo di trasporto come HTTP o SMTP.
- Il messaggio è un documento XML che può contenere un'operazione da eseguire e i dati necessari (parametri) per completare l'operazione.
- L'intestazione del messaggio può includere informazioni supplementari, come autenticazione, o dettagli specifici di routing.

3. Elaborazione del messaggio da parte del server

- Il server riceve il messaggio SOAP e lo interpreta.
- Se c'è un header, il server lo esamina prima di passare al corpo del messaggio. L'header può includere informazioni utili per decidere come trattare il messaggio.
- Il server esegue l'operazione specificata nel corpo del messaggio (per esempio, può fare una query al database o calcolare qualcosa).

4. Risposta del server

- Una volta che il server ha elaborato la richiesta, invia una risposta SOAP.
- La risposta è anch'essa un messaggio XML che contiene i dati richiesti nel corpo del messaggio.
- Se si verifica un errore, la risposta includerà un elemento Fault che descrive l'errore (ad esempio, un errore di sistema o di elaborazione).

5. Struttura di un messaggio SOAP (esempio)

```
  <soapenv:Envelope                    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:web="http://www.example.com/webservice">
     <soapenv:Header/>
     <soapenv:Body>
        <web:RequestData>
           <web:Parameter1>value1</web:Parameter1>
           <web:Parameter2>value2</web:Parameter2>
        </web:RequestData>
     </soapenv:Body>
  </soapenv:Envelope>
```

In questo esempio:

- Envelope è l'elemento radice che racchiude tutto.
- Header è vuoto in questo caso, ma sarebbe usato per informazioni come l'autenticazione.
- Body contiene la parte centrale del messaggio con i dati da inviare, come i parametri da elaborare.

6. WSDL (Web Services Description Language)

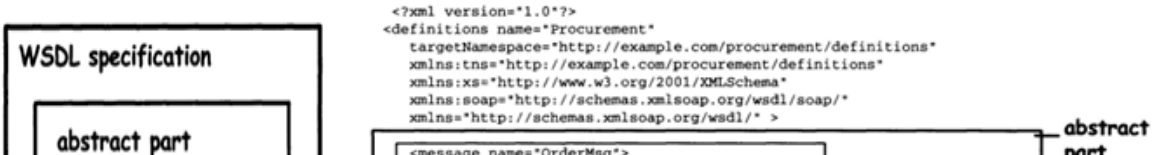
- WSDL è un file XML che descrive come un servizio SOAP può essere utilizzato. Include:

    Le operazioni disponibili

    I parametri necessari per ciascuna operazione

    L'URL a cui il servizio è accessibile

- Un client utilizza il WSDL per sapere come interagire con il servizio, generando automaticamente il codice necessario per inviare richieste SOAP.



Funzionamento sintetico

1. Client invia un messaggio SOAP: Contiene un'operazione da eseguire (ad esempio, chiamare un metodo o passare dati).
2. Server riceve il messaggio, interpreta l'intestazione e il corpo, esegue l'operazione richiesta.
3. Server invia una risposta SOAP, con i risultati o un errore.

Svantaggi di SOAP

1. **Complessità**: La struttura XML di SOAP è complessa, il che rende i messaggi verbosi e difficili da gestire rispetto ad altri formati più leggeri come JSON. La creazione e l'analisi dei messaggi possono risultare costose in termini di risorse.

2. **Overhead**: L'uso di XML aggiunge un overhead significativo rispetto a soluzioni più leggere come REST. L'elaborazione XML richiede risorse di calcolo più elevate, che potrebbero non essere ideali per applicazioni ad alte prestazioni.

3. **Difficoltà di utilizzo nelle applicazioni web moderne**: SOAP è più difficile da integrare in applicazioni moderne rispetto a REST, che è molto più semplice e compatto, particolarmente adatto per applicazioni web e mobili.

REST e l'Evoluzione da SOAP

Nel panorama delle architetture web, **SOAP** e **REST** rappresentano due approcci distinti, con SOAP che precede REST. La nascita e l'evoluzione di REST sono state guidate da alcune criticità e limiti di **SOAP** che hanno spinto gli sviluppatori e le aziende a cercare soluzioni più leggere e semplici per le API.

SOAP: Il Periodo di Dominio (1998-2000)

Nel tardo anni '90, con la crescita delle applicazioni distribuite e la necessità di comunicare tra sistemi eterogenei, nacque **SOAP** (Simple Object Access Protocol). SOAP fu progettato come una soluzione standardizzata per la comunicazione tra sistemi, usando **XML** per definire i messaggi e permettere la comunicazione via **HTTP**, **SMTP**, o altri protocolli. SOAP era quindi molto più robusto e complesso rispetto ad altre soluzioni, come l'uso diretto di HTTP.

Svantaggi di SOAP:

- **Complessità**: La struttura XML di SOAP è complessa, il che rende i messaggi verbosi e difficili da gestire rispetto ad altri formati più leggeri come JSON. La creazione e l'analisi dei messaggi possono risultare costose in termini di risorse. L'uso di XML aggiunge un overhead significativo rispetto a soluzioni più leggere come REST.

L'elaborazione XML richiede risorse di calcolo più elevate, che potrebbero non essere ideali per applicazioni ad alte prestazioni.

- **Difficoltà di utilizzo nelle applicazioni web moderne**: SOAP usava **solo il metodo HTTP POST** per inviare messaggi, il che lo rendeva meno flessibile in quanto non sfruttava pienamente le capacità di HTTP (ad esempio, metodi come GET, PUT, DELETE).

- **Eccessivo formalismo**: SOAP si basa su una struttura molto formale che richiede definizioni precise, come i **WSDL (Web Services Description Language)**, che descrivono i servizi web, ma ciò rendeva l'implementazione più complicata e complicava l'interoperabilità.SOAP è più difficile da integrare in applicazioni moderne rispetto a REST, che è molto più semplice e compatto, particolarmente adatto per applicazioni web e mobili.

La Nascita di REST: Un Approccio Leggero (2000-2005)

Nel 2000, **Roy Fielding**, uno degli architetti del protocollo HTTP, propose una nuova architettura per le comunicazioni tra sistemi chiamata **REST (Representational State Transfer)**, descritta nel suo dottorato. REST nasce come un'alternativa più leggera e semplice a SOAP, e sfrutta al massimo il **protocollo HTTP**.

Motivazioni principali per la nascita di REST:

1. **Problemi di performance di SOAP**: SOAP, con la sua struttura XML complessa e l'overhead, risultava inefficiente per applicazioni che richiedevano alte performance, come quelle web o mobili.
2. **HTTP non sfruttato appieno**: SOAP usava **solo il metodo POST** per tutte le operazioni, mentre HTTP ha una serie di metodi che possono essere utilizzati in modo semantico (ad esempio, GET per leggere dati, POST per creare, PUT per aggiornare, DELETE per eliminare). REST sfruttava questi metodi per mappare facilmente le operazioni **CRUD** (Create, Read, Update, Delete) sulle risorse.

HTTP e il Mappaggio con i Metodi CRUD

Una delle intuizioni chiave di REST è che i metodi HTTP possiedono una semantica naturale che si può mappare direttamente sulle operazioni CRUD:

- **GET**: Recupera una risorsa (equivalente a **Read**).
- **POST**: Crea una nuova risorsa (equivalente a **Create**).
- **PUT**: Aggiorna una risorsa esistente (equivalente a **Update**).
- **DELETE**: Elimina una risorsa (equivalente a **Delete**).

A differenza di SOAP, che richiedeva l'invio di messaggi XML complessi e non sfruttava la semantica dei metodi HTTP, **REST utilizzava HTTP in modo naturale**, rendendo l'architettura più semplice, comprensibile e, soprattutto, meno costosa in termini di risorse.

REST (Representational State Transfer) è un'architettura leggera per la comunicazione tra client e server, utilizzata principalmente nelle applicazioni web. Funziona in modo semplice e diretto, sfruttando i metodi HTTP standard come GET, POST, PUT e DELETE. Ecco come funziona REST concentrandoci sugli aspetti essenziali:

1. Risorse

- **Risorse** sono gli oggetti o i dati che il client vuole ottenere o manipolare (ad esempio, utenti, articoli, ordini, etc.).
- Ogni risorsa ha un **URI (Uniform Resource Identifier)** che la identifica in modo univoco. Ad esempio, un'applicazione potrebbe avere una risorsa "utente" identificata da un URL come `http://example.com/users/123`.

2. Struttura di una richiesta REST

- Una richiesta REST inviata dal client consiste principalmente di:

  URL (URI): L'indirizzo della risorsa, come ad esempio `http://example.com/users/123`.

  Metodo HTTP: Determina l'operazione da eseguire (GET, POST, PUT, DELETE).

  Header HTTP: Contiene metadati, come il tipo di contenuto (ad esempio `application/json` per JSON) o l'autenticazione.

  Body: Quando necessario, contiene i dati da inviare al server (ad esempio, per creare una nuova risorsa con POST).

Endpoint URL	Metodo HTTP	Esito Atteso	Status Code
/products/	GET	Elenco di tutti i prodotti (JSON)	200 OK
/products/17/ (questo prodotto esiste)	GET	Dettagli del prodotto con pk 17 (JSON)	200 OK
/products/21/ (questo prodotto non esiste)	GET	Nessun prodotto e messaggio d'errore	404 Not Found
/products/	POST	Creazione di un nuovo prodotto	201 Created
/products/15/	PUT	Aggiornamento del prodotto con pk 15	200 OK
/products/15/	DELETE	Cancellazione del prodotto con pk 15	204 No Content