Distributed Information Systems

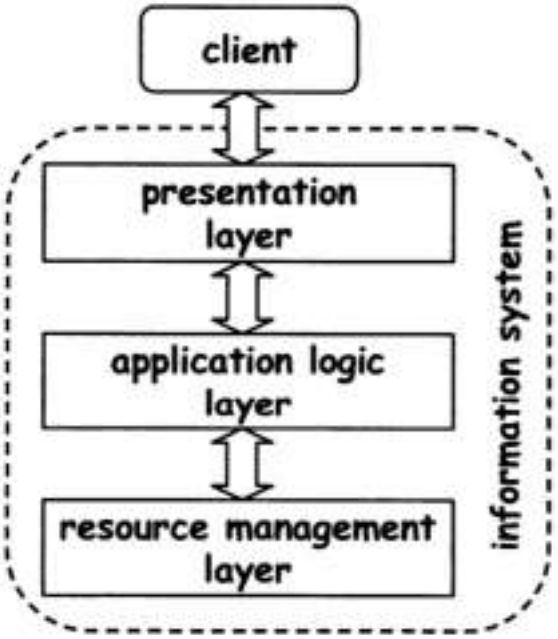
I **Web Services** sono una tipologia di *distributed information systems*. Molti dei problemi che cercano di risolvere, così come il loro design attuale, possono essere capiti meglio considerando come i *distributed information systems* si sono evoluti nel tempo.

In questo capitolo descriveremo gli aspetti chiave dei *distributed information systems*:
- design
- architecture
- communication patterns

Design of IS

Layers of an Information System

Ad un livello concettuale, gli information systems sono progettati su tre layers: presentation, application logic, resource management.



- **Presentation Layer:** tutti gli IS[1] devono comunicare con entità esterne (umani e/o computers). Una grande parte di questa comunicazione include la presentazione delle informazioni a queste entità esterne e permettere l'interazione (richiedere operazioni e ricevere risposte). Questo layer è concettualmente simile alle View del pattern architetturale MVC.
- **Application Logic Layer:** gli IS non si limitano a inviare informazioni e dati. La maggior parte infatti esegue svariate operazioni sui dati prima di recapitarli. Queste operazioni includono un programma che implementa l'operazione richiesta dal client attraverso il "presentation layer". Ci si riferisce spesso a questi programmi come ai "servizi" offerti dal IS (es. prenotazione volo…).
- **Resource Management Layer:** gli IS necessitano di dati su cui lavorare. Da un punto di vista astratto, questo layer include diverse data sources, dai database alle directories fino ad altri IS (ognuno con i propri layer).

---

[1] Information System

Top-Down Design of an IS

L'idea è di partire definendo le funzionalità del sistema dal punto di vista dei client e di come questi interagiranno con il sistema stesso. Una volta che le funzionalità top-level sono state definite, bisogna progettare l'application logic in modo che implementi queste funzionalità. Infine si definiscono le risorse necessarie all'application logic.

Presentation Layer -> Application Logic Layer -> Resource Management Layer
=
Driven by the Functionality

Vantaggi	Svantaggi
Il design enfatizza i goals	Può essere applicato solamente per i sistemi sviluppati from scratch

Bottom-Up Design of an IS

Oggigiorno, gli IS sono costruiti integrando sistemi già esistenti, chiamati *legacy system*[2]. Questa integrazione non può essere top-down poiché non è possibile progettare le funzionalità dei sistemi da integrare. Le funzionalità sono quelle che sono e difficilmente questi sistemi possono essere modificati.
L'approccio bottom-up prevede quindi i seguenti steps:

- Definire le funzionalità (o goals) top-level.
- Studiare il Resorce Management Layer (dove i legacy systems risiedono) cercando di capire come ottenere le funzionalità desiderate a partire da ogni singolo componente.
- Progettare l'Application Logic Layer in modo da usare a dovere le risorse sottostanti al fine di implementare le funzionalità desiderate.

Resource Management Layer -> Application Logic Layer -> Presentation Layer
=
Driven by the Characteristics of the Lower Layers

Una caratteristica importante di un design bottom-up è quella di fornire sistemi loosely-coupled, dove molti componenti possono essere usati come stand-alone systems.

Vantaggi	Svantaggi
Non ha molto senso parlare di vantaggi/svantaggi poiché in molti casi, quando si deve lavorare con applicazioni legacy, il design bottom-up è l'unica opzione.	

[2] Un sistema o un applicazione diventa legacy nel momento in cui viene usato in un contesto diverso da quello per cui inizialmente è stato previsto

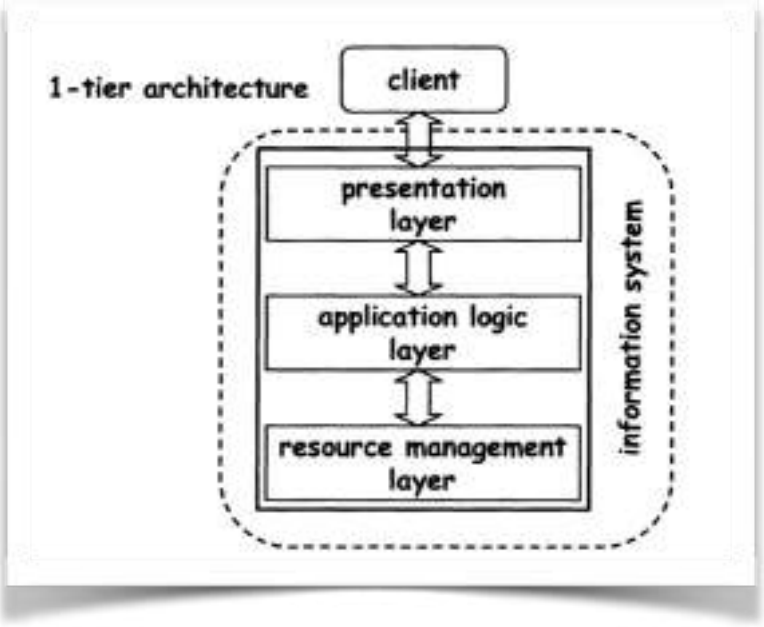
Architecture of IS

I tre layer discussi prima sono costrutti meramente concettuali. Nell'implementazione reale di un IS, questi layer sono combinati e distribuiti in diversi modi e ci si riferisce ad essi con il termine ***tiers***.

In base a come questi tiers sono organizzati, possiamo individuare **quattro tipi di IS:**

- 1-tier
- 2-tiers
- 3-tiers
- N-tiers

One-tier Architectures



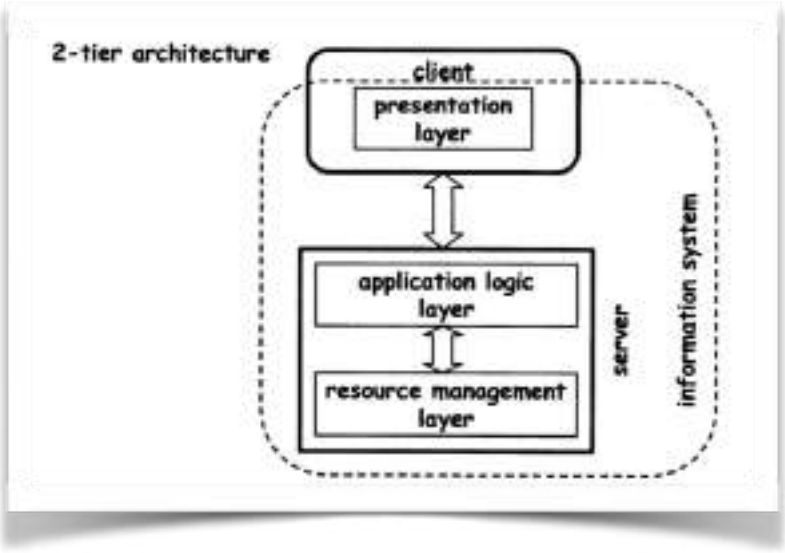
Usi: mainframe-based systems e dumb terminal (visualizzano le informazioni così come vengono inviate dal mainframe).

Caratteristiche: presentation, application logic e resource management layers sono fusi in un unico tier. Da notare che anche il presentation layer risiede nel mainframe, che ne controlla ogni aspetto dell'interazione con il client (il formato e la struttura delle informazioni, come queste vengono visualizzate lato client e come reagire ai vari input del client).

Integrazione: quando questi sistemi devono essere integrati con altri sistemi, il metodo più comune è quello dello "*screen-scraping"* (soluzione né efficiente né tanto meno elegante).

Vantaggi	Svantaggi
I progettisti sono liberi di fondere i layers quanto necessario ad ottimizzare le prestazioni	Difficili e cari da mantenere
Nessun context switch e chiamate tra componenti	Sistemi quasi impossibili da modificare
Nessuna conversione/trasformazione complessa dei dati	integrazione poco efficiente ed elegante
Praticamente nessun client development	
Le prestazioni raggiunte da sistemi con architettura 1-tier rimangono quasi sempre imbattute.	

Two-tiers Architectures



Usi: questa architettura è nata con l'avvento dei PC. Al posto quindi di mainframe e dumb terminals, ci sono grandi computer (mainframes e servers) e piccoli computer (PCs e workstations). Questa architettura è diventata molto popolare soprattutto come architettura client/server.

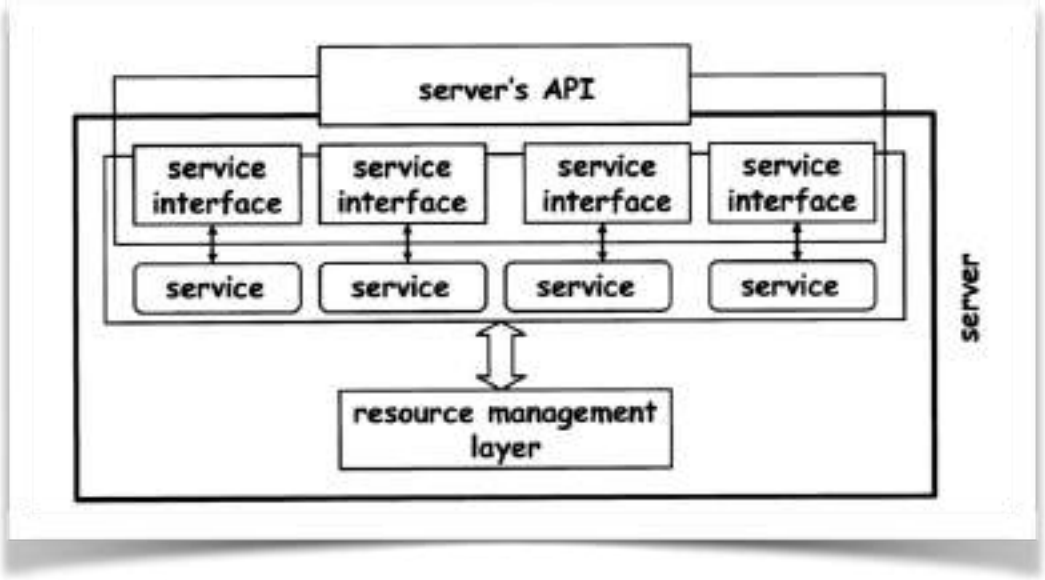
Caratteristiche: application layer e resource management layer risiedono su server mentre il presentation layer viene spostato su client. **Spostare il presentation layer sui client ha due vantaggi principali:**

- il presentation layer può sfruttare la potenza di calcolo dei client, liberando quindi risorse lato server per gli altri due layers;
- è possibile progettare presentation layers ad hoc per differenti client e differenti utilizzi, senza aumentare la complessità del sistema (pannello admins…).

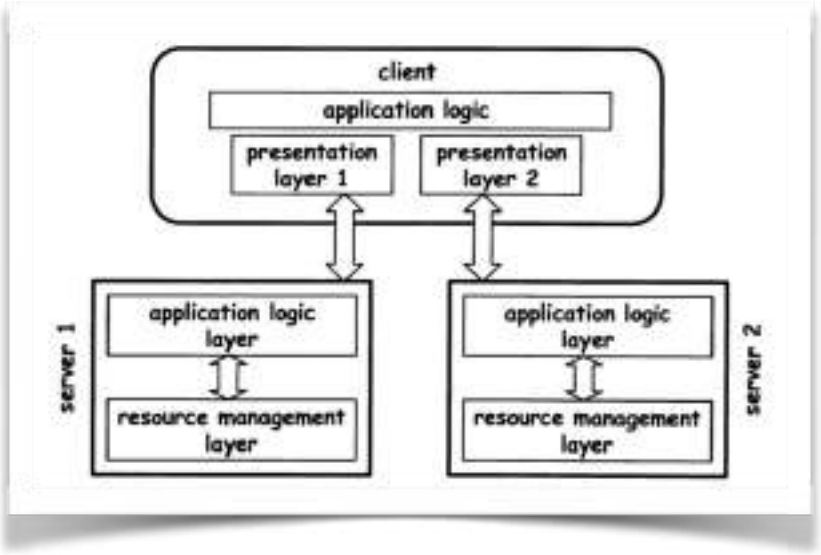
Due tipologie di client (a seconda della complessità):

- thin client: client con funzionalità minime;
- fat client: client con funzionalità estese e complesse.

Le architetture client/server e meccanismi come le RPC (vedere prossimo capitolo) hanno forzato i progettisti di DIS a pensare in termini di **interfacce pubbliche**. Infatti, al fine di sviluppare i clients, il server deve avere una nota e stabile interfaccia (API). I singoli programmi responsabili dell'application logic prendono il nome di *"services"*. Ogni service ha una nota *"service interface"* che definisce come bisogna interagire ed usare il servizio stesso. L'insieme di tutte queste interfacce costituisce il *"server's API"*.

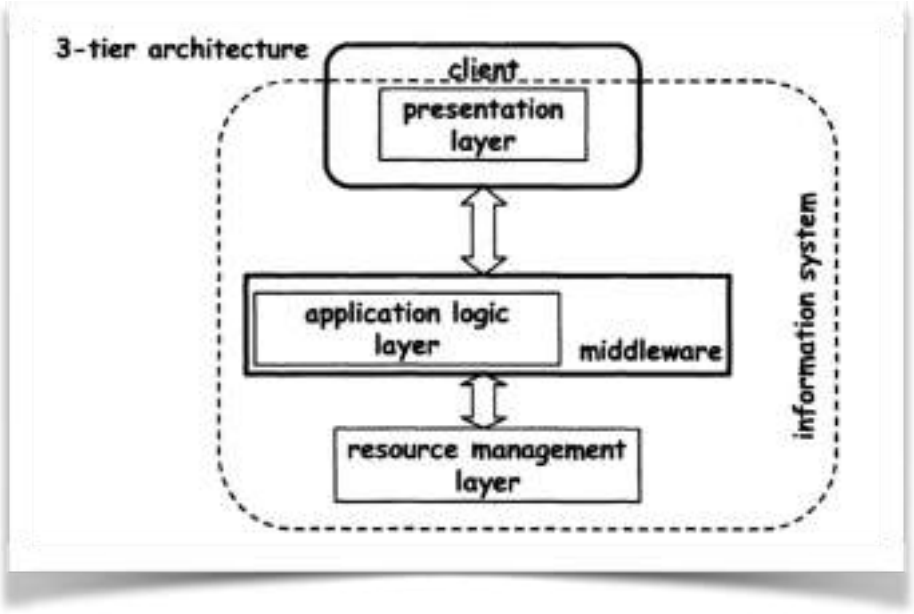


Integrazione: quando questi sistemi devono essere integrati con altri sistemi, tipicamente il compito viene scaricato sui client. In altre parole, viene aggiunto application logic layer aggiuntivo che gestisce la logica dell'integrazione (il problema è che questo layer è incluso nei clients).



Vantaggi	Svantaggi
Mantenendo l'application logic e il resource manager layers insieme (lato server) è ancora possibile eseguire le operazioni chiave in modo molto veloce	Un singolo server può gestire un numero limitato di clients
Sviluppo IS cross-platforms grazie al fatto che il presentation layer è indipendente dal server	Integrazione poco efficiente

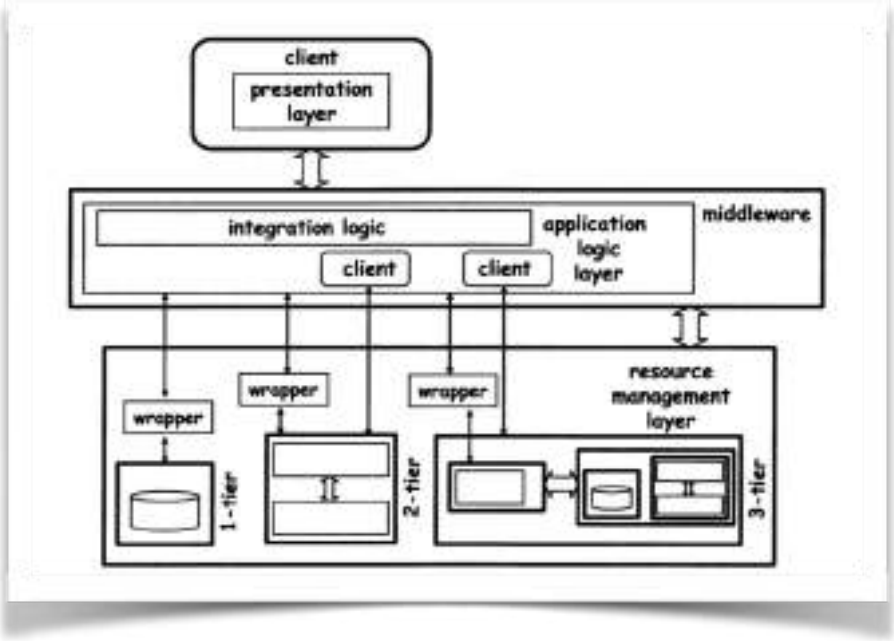
Three-tiers Architectures



Usi: con la proliferazione di servers, ognuno con la propria interfaccia, e l'incremento della banda fornito dalle LANs, le architetture 2-tiers cominciarono a dare problemi (soprattuto legati alla necessità di integrare molteplici server).

Caratteristiche: come abbiamo visto con le architetture 2-tiers, il problema dell'integrazione non può essere scaricato sui clients. L'architettura 3-tiers risolve il problema introducendo un tier intermedio tra il client e il server. Questo nuovo tier contiene l'application logic necessaria a gestire l'integrazione. Da notare che il resource manager layer potrebbe essere costituito da diversi sistemi, ciascuno con i propri application e resource manager layers organizzati secondo una qualsiasi architettura (1-tier, 2-tiers, 3-tiers…). Negli ultimi due casi, si parla di architettura N-tiers (vedere dopo) poiché la 3-tiers non è stata propriamente designata ad integrare sistemi 2/3-tiers. Nell'architettura 3-tiers classica il resource manager layer comprende data sources (db, directories…), non sistemi complessi. Ed è proprio in questo contesto che nascono gli ODBC (come ad esempio il JDBC), interfacce sviluppate al fine di permettere all'application logic del middleware level di accedere ai database in modo standardizzato.

Anche se questa architettura è stata designata principalmente come **piattaforma di integrazione**, può essere utilizzata anche come architettura 2-tiers. In quest'ultimo caso il tier intermedio non contiene la logica di integrazione ma invece contiene le logiche dei singoli sistemi.



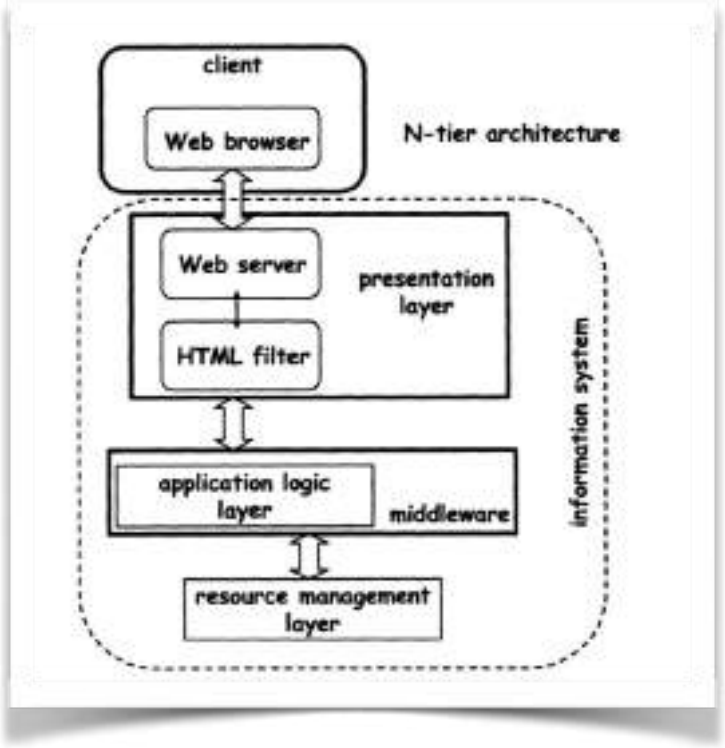
Vantaggi	Svantaggi
La perdita di performance dovuta all'aggiunta del tier intermedio è più che compensata dalla maggior flessibilità del sistema..	Integrazione difficile su Internet (l'architettura non è stata designata per questo scopo) e se bisogna integrare diversi 3-tiers systems (mancanza di standard precisi)
Inoltre la logica del tier intermedio può essere distribuita su più nodi, ne consegue una maggiore affidabilità complessiva del sistema	

N-tiers Architectures

Usi: queste architetture non costituiscono in realtà un'evoluzione delle architetture 3-tiers, bensì sono architetture 3-tiers nella loro forma più generale e perfezionate per essere utilizzate nel contesto di Internet (Internet come canale d'accesso principale).

Caratteristiche
Due configurazioni principali:

- **Linking of different systems:** il resource manager layer non include solamente risorse semplice come databases ma anche sistemi complessi, ognuno con la sua architettura (vedere prima figura architetture 3-tiers).
- **Adding connectivity through the Internet:** i client è un Web Browser e il presentation layer è distribuito tra il Web Browser e il Web Server. L'HTML filter converte i dati da un qualsiasi formato usato ai livelli sottostanti nel formato HTML interpretabile dai Web Browser.



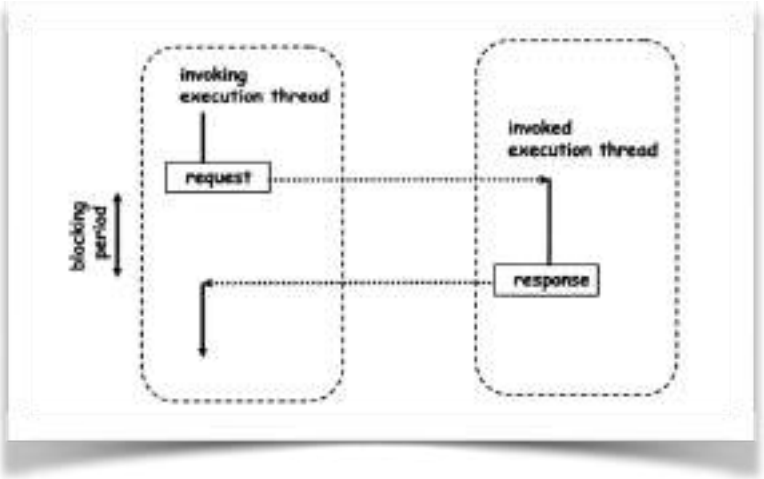
Vantaggi	Svantaggi
Architettura molto flessibile che consente la distribuzione di ciascun tier	Molti strati coinvolti, difficile da sviluppare, mantenere, modificare e spesso con funzionalità ridondanti

Conclusioni sulle architetture

Ogni tier in più aggiunge flessibilità, funzionalità e distribuzione. Al costo di una diminuzione delle performance (la comunicazione delle varie componenti organizzate in tiers diversi è più lenta).

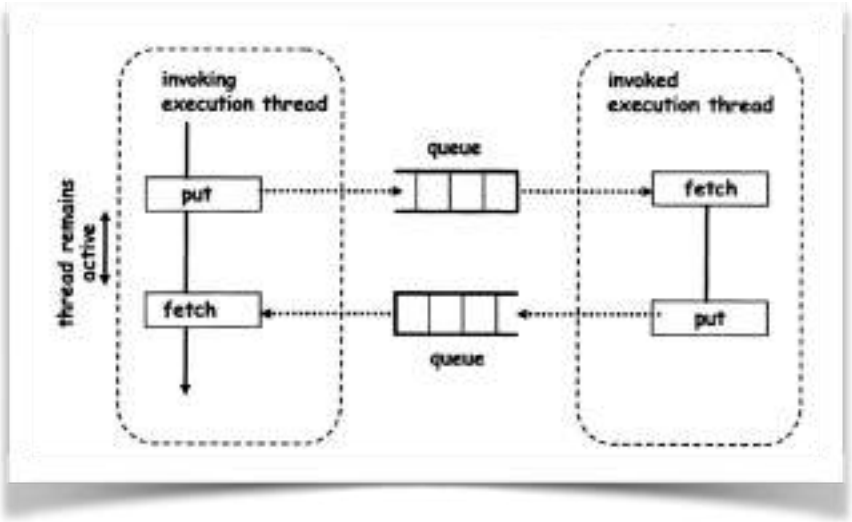
Communication in an IS

Comunicazione Sincrona



Vantaggi	Svantaggi
Design più semplice	Spreco di tempo e risorse

Comunicazione Asincrona



Un'interazione asincrona è più utile quando la comunicazione non è del tipo richiesta-risposta. Ad esempio nelle applicazioni che hanno bisogno di mandare periodicamente informazioni ai clients, o sistemi che fanno uso di event notification, o ancora i **sistemi publish/subscribe** dove le componenti pubblicano (publishing) continuamente le informazioni al sistema e le altre componenti interessate si iscrivono (subscribe) ad esse (a questo punto riceveranno informazioni più dettagliate).

Tipicamente le interazioni asincrone richiedono che i messaggi siano memorizzati in qualche intermediario prima che arrivino al destinatario, in questo modo, ad esempio, le due parti non devono essere online contemporaneamente (si pensi alla posta elettronica). Questi intermediari tipicamente sono **code**.

Middleware

E' la soluzione architetturale al problema dell'integrazione di collezioni di server e applicazioni al di sotto di una comune service interface.

I middleware offrono un buon livello di astrazione tale da nascondere molti dei complessi dettagli implementativi tipici dello sviluppo di applicazioni distribuite. In questo modo il programmatore non dovrà farsi carico di gestire ogni aspetto del sistema ma è il middleware stesso ad occuparsene. Tramite questo livello di astrazione, lo sviluppatore ha accesso a funzionalità che altrimenti sarebbero da sviluppare from scratch (gestione delle socket, persistenza, transazionalità, gestione multi-threading, logging…).

N.B. Bisogna sempre scegliere un middleware che offre quello di cui si ha bisogno e non di più. Infatti più sono le funzionalità offerte, più la complessità del middleware e del suo utilizzo/mantenimento sono alte.

Tipi di Middleware

- **RPC:** rappresentano la forma base di middleware. Fornisce l'infrastruttura necessaria a trasformare "procedure calls" in "remote procedure calls" in modo uniforme e trasparente. Oggigiorno, i sistemi RPC sono alla base di quasi tutti i tipi di middleware (inclusi i web services middleware).
- **TP Monitors:** sono la tipologia più vecchia e conosciuta di middleware. Semplificando, i TP Monitors possono essere visti come RPC con capacità transazionali. A seconda se sono implementati mediante architettura 2-tiers o 3-tiers, di distinguono in TP-lite e TP-heavy.
- **Object Brokers:** gli RPC sono stati sviluppati quando predominavano i linguaggi imperativi. Con la nascita dei linguaggi OO, è nata anche la necessita di supportare l'invocazione di oggetti remoti. Questo è la ragion d'essere degli Object Brokers. Molti di questi middleware usano RPC come meccanismo sottostante per implementare la chiamata di oggetti remoti e non differiscono quindi molto da essi. CORBA è l'esponente di questa categoria di middleware.
- **Object Monitors:** quando i vendors provarono a specificare e a standardizzare le funzionalità delle piattaforme basate su Object Brokers, capirono che molte di queste erano già disponibili per i TP Monitors. Ma i TP Monitors erano basati su linguaggi procedurali. Il risultato è stato quello di adattare i TP Monitors all'OO, dando vita quindi agli Object Monitors (TP Monitors OO).
- **Message-Oriented Middleware (MOM):** le prime forme di RPC fecero emergere i limiti della comunicazione sincrona (non sempre necessaria). Questo limite fu inizialmente superato con le RPC asincrone. Successivamente i TP Monitors furono estesi con sistemi persistenti di code di messaggi che si rivelarono così utili da portare alla nascita di una nuova famiglia di middleware indipendenti: i MOM. Queste piattaforme forniscono accesso transazionale alle code, code persistenti e primitive di lettura/scrittura su code locali e remote.
- **Message Brokers:** tipo di MOM capace di trasformare e filtrare i messaggi direttamente nelle code. Fondamentalmente, l'unica differenza è costituita dalla presenta di una logica applicativa direttamente integrata nelle code che permette interazioni asincrone molto potenti ed efficaci.

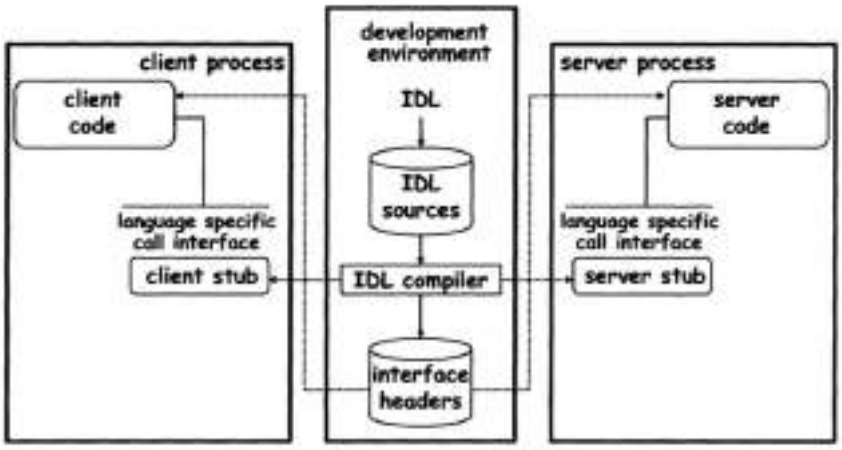
Molti middleware con spesso funzionalità ridondanti. Questa ridondanza non è relativa all'astrazione fornita, bensì all'infrastruttura sottostante (molte volte basata su RPC).

RPC

Con le RPC sono nati i concetti di *client* (il programma che invoca la procedura remota) e di *server* (il programma che implementa le procedure remote invocabili) e molti altri concetti usati ancora oggi come: *interface definition languages* (IDL), *name and directory services*, *dynamic binding*, *service interface*…

Oggigiorno, le RPC rappresentano il cuore di molti DIS. Possiamo ad esempio trovarle nelle *RMI* (invocazione remota di metodi al posto di procedure, ma concettualmente equivalenti) e nelle *stored procedure* (usate per interagire un db).

Funzionamento



- **Definire l'interfaccia della procedura remota:** questo viene fatto tramite un IDL che fornisce un'astrazione della procedura stessa in termini di input (parametri) e output (risultato dell'invocazione).
- **Compilazione dell'interfaccia:** l'interfaccia IDL viene compilata e il risultato della compilazione sono il client/server stub (stesso discorso di progIII - binding to server, marshaling/unmarshaling, handling network details…).

N.B. Binding
Il binding è il processo tramite il quale il client individua il server a cui sottoporre le richieste di invocazione delle procedure remote. Può essere **statico** o **dinamico**.

Binding Statico: lo stub del client viene creato direttamente legato ad uno specifico server. Più semplice ed efficiente ma alto coupling.

Binding Dinamico: viene aggiunto un ulteriore strato, il *name and directory service (binder)*. Questo strato ha il compito di restituire l'opportuno indirizzo del server in base alla firma della procedura invocata. Meno efficiente ma soluzione più flessibile.

Estensioni

- async RPC
- transactional RPC

TP Monitors

I *transaction processing monitors* (TP Monitors), sono una delle forme di middleware più vecchie ed efficienti (nel corso degli anni sono stati studiati ed ottimizzati molto). Oggigiorno, sono dietro a molti dei sistemi N-tiers e la loro architettura/funzionalità sono d'ispirazione per le nuove forme di middleware.

Evoluzione

I TP Monitors inizialmente furono sviluppati per lavorare nel contesto dei mainframes. Vista la natura fortemente multi-threading di questi ultimi, uno dei primi requisiti fu quello della transazionalità.

Col passare degli anni, i TPM si sono evoluti moltissimo, sia da un punto di vista architetturale (1-tier to N-tiers) che da un punto di vista delle funzionalità offerte (nelle versioni più complesse arrivano quasi a raggiungere quelle di un SO). I TPM forniscono diversi livelli di astrazione adatti a risolvere problemi di natura e complessità diversa.

TP-lite monitors
I TPLM sono una forma leggera di TPM. L'idea era di offrire le funzionalità chiave dei TPM (es. transactional RPC) come uno strato extra incluso direttamente nei database management systems. Questo può essere fatto mediante le stored procedures, dove l'applicazion logic può essere scritta ed eseguita direttamente nello scope del db piuttosto che in un layer intermedio come avviene dei TPM convenzionali. TPLM forniscono un'estensione delle funzionalità offerte da un db, caratterizzate da un architettura 2-tiers e prive delle funzionalità sofisticate di un classico TPM (ma con funzionalità sufficienti per tutte quelle applicazioni/sistemi i cui resource manager layers sono costituiti solamente da db).

Servizi di un TP Monitor (slides)

- Gestione dei processi server: attivazione, funnelling, monitoraggio e bilanciamento dei carichi
- Gestione delle transazione: proprietà ACID
- Gestione della comunicazione client/server

Transactional RPC e TP Monitors

Lo scopo principale di un TPM è quello di supportare l'esecuzione di transazioni distribuite.
A questo scopo, i TPM implementano un'astrazione chiamata *transactional RPC* (TRPC).

Le RPC convenzionali, furono inizialmente progettate per permettere ad un client di comunicare con un server, invocando procedure remote. Ma cosa succede se ad esempio un client comunica con più server? Le RPC tratterebbero ogni invocazione remota come indipendente dalle altre, ma non sempre è quello che si desidera. Se ad esempio queste invocazioni multiple sono legate, si potrebbe desiderare che o vanno tutte a buon fine o nessuna di essere deve andare a buon fine.

Ed è proprio a questo punto che entrano in gioco le TRPC. L'dea è quella di impacchettare (*wrap*) una serie di invocazioni di procedure remote in una transazione (in parole povere, estendendo le RPC con le proprietà acide già note nell'ambito dei database). Ma vediamo come funziona:

- Le invocazioni remote sono wrappate all'interno di una transazione mediante due comandi: **BOT** (begin of transaction) e **EOT** (end of transaction). Questo garantisce l'atomicità delle invocazioni.
- Quando si raggiunge il BOT, lo stub del client contatta il ***transaction management*** (TM) per richiedere un identificativo univoco della transazione appena cominciata.
- Ad ogni invocazione, lo stub del client, include l'identificativo della transazione.
- Quando viene raggiunto l'EOT, lo stub del client contatta il TM che inizia la ***two-phase commit*** (2PC).
- La 2PC avviene in due passi:
    - il TM invia ai servers coinvolti un messaggio di *prepare_to_commit*;

- se tutti rispondono positivamente entro il timeout, il TM invia il messaggio *commit*, diversamente se qualcuno risponde negativamente o non risponde viene inviato il messaggio *roleback.*

Dove il transaction management è il modulo che coordina le interazioni tra client/server al fine di garantire il corretto funzionamento delle transazioni.

Object Brokers

Gli OB estendono le RPC con il paradigma OO (RPC + OO) e forniscono tutta una serie di servizi che facilitano lo sviluppo di applicazioni OO distribuite.

Evoluzione

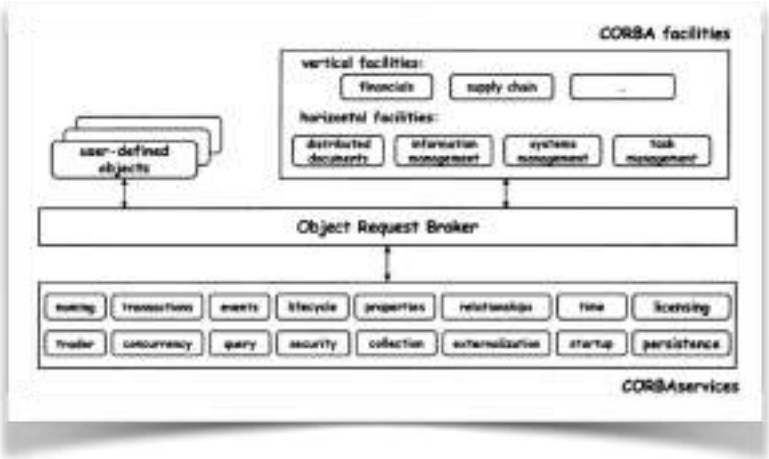
Gli OB rappresentano la naturale evoluzione delle RPC verso il paradigma OO. Concettualmente i due middleware sono molto simili, ma ci sono alcune differenze:

- i client non invocano procedure remote, ma bensì metodi remoti;
- il middleware non fa il binding client/server, ma bensì client/oggetto remoto[3];
- siamo in un contesto OO, di conseguenza troviamo concetti come ereditarietà, polimorfismo…

CORBA

L'esponente di questa famiglia di middleware è sicuramente CORBA (Common Object Request Broker Architecture). CORBA è un'architettura e una specifica per la creazione e la gestione di applicazioni OO distribuite[4].

Architettura di CORBA



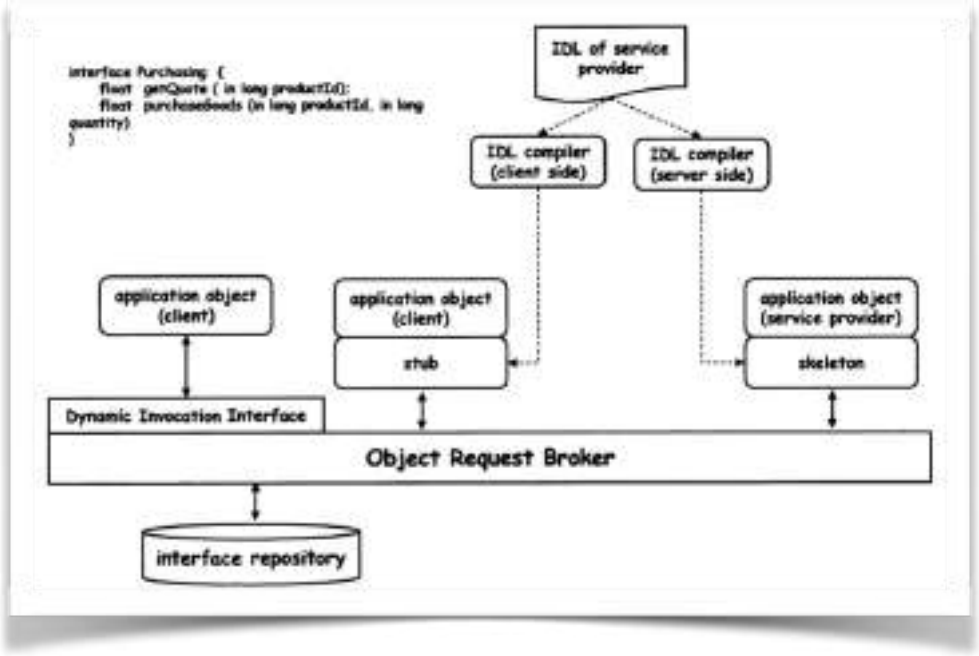
- **Object Request Broker(ORB):** fornisce le funzionalità base dell'interoperabilità degli oggetti.
- **CORBA services:** servizi base necessari alla maggior parte degli oggetti (persistenza, lifecycle management, sicurezza…).
- **CORBA facilities:** servizi base necessari alla maggior parte delle applicazioni (gestione dei documenti, internazionalizzazione, supporto per agenti mobili).

N.B. l'ORB rappresenta il cuore dell'architettura, ogni interazione client/service (user-defined, CORBA services, CORBA facilities) passa attraverso esso.

[3] Oggetti di classi diverse potrebbero definire metodi con la stessa firma ma con comportamenti e implementazioni diverse, di conseguenza il client deve essere collegato ad oggetti specifici in modo da eliminare tutte le possibili ambiguità.

[4] CORBA offre delle specifiche standardizzate più che una concreata implementazione.

Funzionamento di CORBA



Funzionamento molto simile alle RPC:

- Ogni oggetto per essere accessibile attraverso un ORB deve dichiarare la propria interfaccia (IDL + concetti OO).
- Il compilatore di IDL di CORBA prese queste interfacce come input genera:
  - **Stub:** oggetto proxy che nasconde la distribuzione e da l'impressione al client di invocare un metodo di un oggetto locale;
  - **Skeleton:** svincola l'oggetto reale lato server dai problemi derivanti dalla distribuzione.

CORBA Dynamic Service Selection and Invocation

- **Binding statico:** il client è staticamente legato ad una specifica interfaccia di uno specifico oggetto (lo stub generato è legato ad una specifica service interface).
- **Binding dinamico:** permette all'applicazione client di scoprire dinamicamente nuovi oggetti, ottenere le relative interfacce e costruire l'invocazione dell'oggetto on the fly. Questo è possibile grazie a:
  - **Interface repository:** salva tutte le interfacce IDL di tutti gli oggetti noti all'ORB.
  - **Dynamic invocation interface:** fornisce operazioni come *get_interface* e *create_request* che permettono al client di invocare dinamicamente i metodi relativi alle interfacce appena scoperte.

Object Monitors

Quando si cerco di estendere gli Object Brokers con alcune delle funzionalità tipiche dei TPM (come la transazionalità), i risultati furono deludenti. L'*Object Transactional Service* (OTS) di CORBA ne è un chiaro esempio. Il problema era fondamentalmente legato alla scarsa efficienza.

Dunque, visto che molte di queste funzionalità erano state a lungo studiate ed ottimizzate nei TPM, si preferì adattare i TPM all'OO. Il risultato furono gli Object Monitors (TPM + OO).

Message-Oriented Middleware (MOM)

I MOM tipicamente sono presentati come una tecnologia rivoluzionaria. In realtà non è così, molte delle features dei MOM erano già state impiegate in passato:

- **Interazione asincrona:** RPC
- **Code di messaggi:** TPM (per implementare interazioni message-based)

Oggigiorno, la maggior parte dei sistemi di integrazione sono sviluppati come message-oriented middleware.

Message-Based Interoperability

La classe di middleware che supportano il message-based interoperability prende il nome di Message-Oriented Middleware (MOM).

Il termine message-based interoperability si riferisce al paradigma di interazione dove client e service provider comunicano scambiandosi messaggi.

Ogni messaggio ha una specifica struttura:

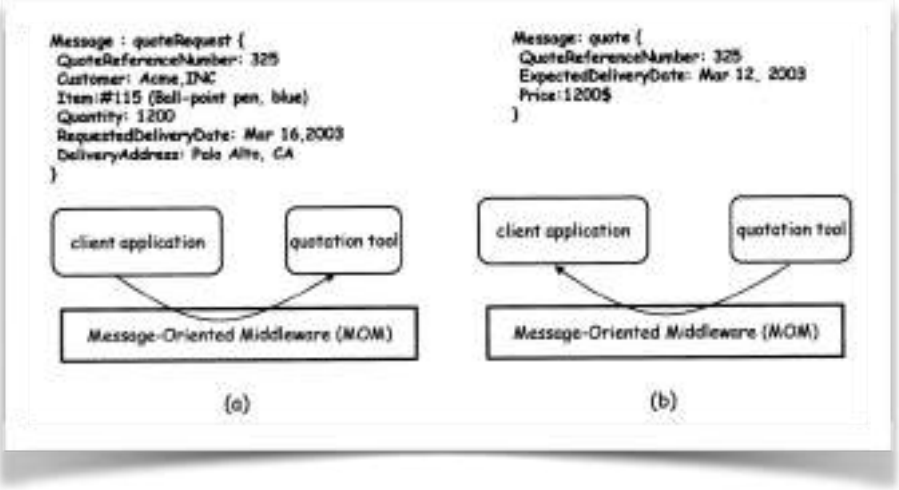
- **Tipo:** tipo di messaggio (quoteRequest)
- **Parametri:** insieme di coppie <nome, valore>

Il linguaggio usato per definire i messaggio dipende dalla piattaforma adottata.

```
Message quoteRequest {
 QuoteReferenceNumber: Integer
 Customer: String
 Item: String
 Quantity: Integer
 RequestedDeliveryDate: Timestamp
 DeliveryAddress: String
}
```

Funzionamento:[5]

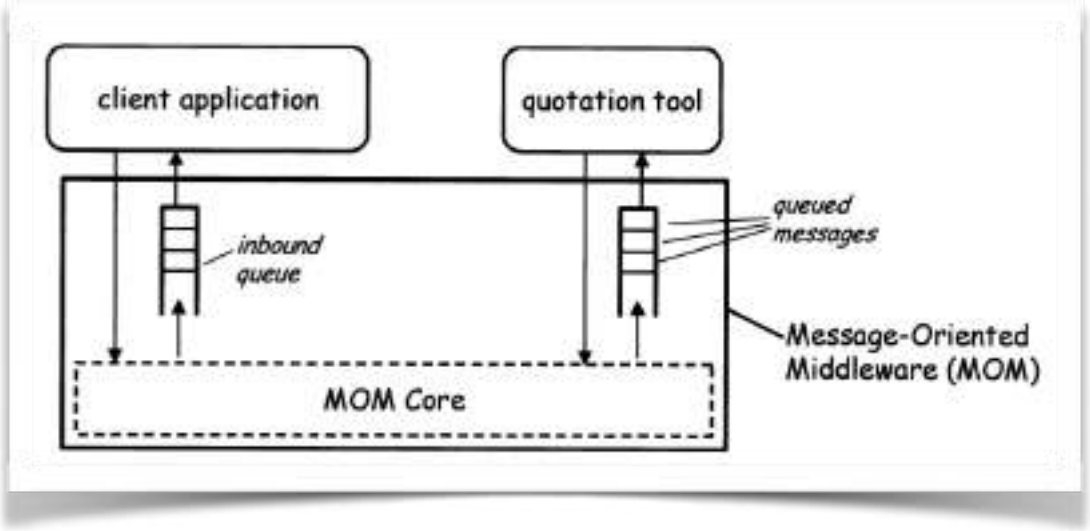
- Client e service provider si mettono d'accordo sulla tipologia dei messaggi che si scambieranno.
- Per richiedere un servizio, il client invia un messaggio (es. quoteRequest) al service provider desiderato.
- Il service provider riceve il messaggio, esegue delle operazioni specifiche per il tipo di messaggio e manda al client un messaggio di risposta contenente le informazioni richieste (es. quote).



[5] Client e service provider sono concetti meramente concettuali, per il MOM tutti i messaggi sono uguali, non tratta diversamente i messaggi di uno piuttosto che dell'altro.

Code di messaggi

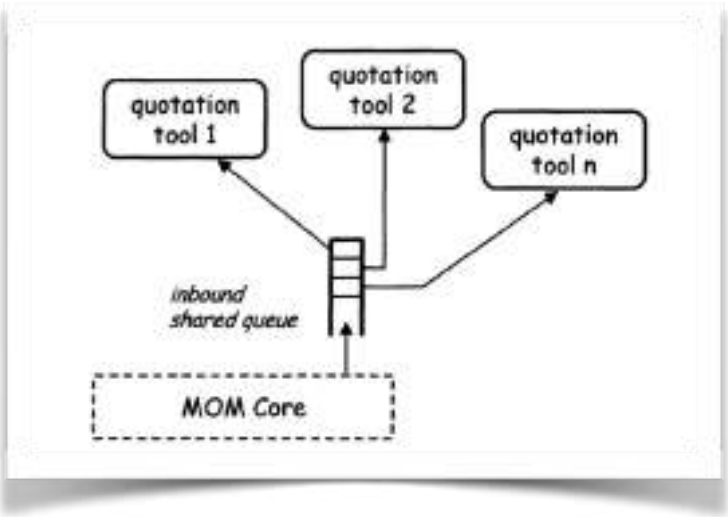
In un modello caratterizzato da code di messaggi, i messaggi inviati dal client sono salvati in una coda (ogni destinatario ha la sua coda - coda sulla destra nell'immagine qua sotto). Quando il destinatario è pronto a ricevere nuovi messaggi, invocherà una specifica funzione del MOM per ricevere il primo messaggio in coda.



N.B. la coda sulla sinistra non è la coda in cui vengono salvati i messaggi inviati dal client che sono in attesa di essere processati dal MOM (il client invia direttamente questi messaggi al MOM), ma è la coda nella quale vengono salvati i messaggi inviati dal "quotation tool" (service provider) destinati al client.

Vantaggi code di messaggi:

- I destinatari possono processori i messaggi quando vogliono
- Mittente e destinatario non devono essere online nello stesso momento
- I messaggi in coda possono essere decorati da priorità e expiration date
- Le code possono essere condivise da più applicazioni (efficenza maggiore)



Interazione con le code di messaggi

I sistemi basati su code, forniscono API che definiscono le modalità di invio/ricezione dei messaggi. L'invio è tipicamente asincrono, mentre la ricezione può essere sia sincrona che asincrona:

- sincrona: il destinatario si mette in attesa di nuovi messaggi, quando arriva un messaggio viene gestito tipicamente su un nuovo thread cosi che il thread principale possa rimettersi in attesa di altri messaggi (funzionamento classico anche dei server);
- asincrona: il destinatario definisce una funzione di callback che sarà invocata dal MOM ogni volte che arriva un nuovo messaggio.

Code transazionali (non ho capito bene - p.81 [64 reale])

Enterprise Application Integration

Problema: quando si ha la necessità di integrare sistemi compatibili e comparabili rispetto alle funzionalità offerte e quando le piattaforme coinvolte non sono molte, i middleware possono essere tranquillamente usati senza ulteriori accorgimenti. Diversamente, per integrazioni di sistemi molto eterogenei, i middleware da soli non sono sufficienti. Il problema principale è dato dal fatto che tutti i middleware fanno assunzioni in merito alla natura dei sistemi sottostanti e quando questi sono molto diversi, l'integrazione diventa difficile se non impossibile.

Soluzione: EAI estendono le capacità dei middleware in modo da renderli adatti all'integrazione di sistemi anche molto eterogenei.

Proliferazione di servizi

I middleware integrando diversi sistemi che, singolarmente forniscono determinati servizi, generano a loro volta nuovi servizi. Questo a portato ad una crescita esponenziale dei servizi. Per l'integrazione dei server si è speso molto tempo e denaro cercando di sviluppare standard che facilitassero la loro integrazione, questo sfortunatamente non si può dire anche per l'integrazione dei servizi. Soprattutto se sono servizi offerti da diverse piattaforme.

EAI Middleware: Message Brokers

I middleware tradizionali basati su RPC e MOM, creano point2point link tra le applicazioni, e di conseguenza sono poco flessibili nell'integrazione di diversi sistemi. Nello specifico i MOM non forniscono un supporto per definire sofisticate logiche di routing dei messaggi tra diversi sistemi e non aiutano gli sviluppatori ad affrontare efficacemente la l'eterogeneità.

In risposta a queste necessità, i MB estendono i MOM consentendo la definizione di logiche di business direttamente a livello del middleware. Nei MOM l'obiettivo del middleware è di spostare messaggi da un punto ad un'altro (offrendo certe garanzie). I MB oltre a questo forniscono tutta una serie di funzionalità aggiuntive come il **routing**, **filtering** e anche la possibilità di **processare i messaggi direttamente a livello del middleware**. Inoltre, la maggior parte di MB forniscono **adapter** che consentono di mascherare l'eterogeneità consentendo di accedere a tutti i sistemi mediante lo stesso *modello di programmazione* e lo stesso *data exchange format*.

Disaccoppiare mittente/destinatario

Una delle caratteristiche fondamentali dei MB è quella di disaccoppiare il mittente dal destinatario. Nei MOM il client deve specificare esplicitamente il destinatario del messaggio, nei MB è possibili definire logiche di routing dei messaggi direttamente al livello del middleware (logica che può essere inclusa a livello del middleware stesso o direttamente nelle code). Questo svincola il client dal dover esplicitare il destinatario, è la logica di routing che in base al **tipo di messaggio** o all'**identità del mittente che** individua l'appropriato destinatario del messaggio.

Modello di interazione Publish/Subscribe

Grazie alla possibilità di definire logiche di routing dei messaggi, i MB possono supportare diversi modelli di interazione message-based.

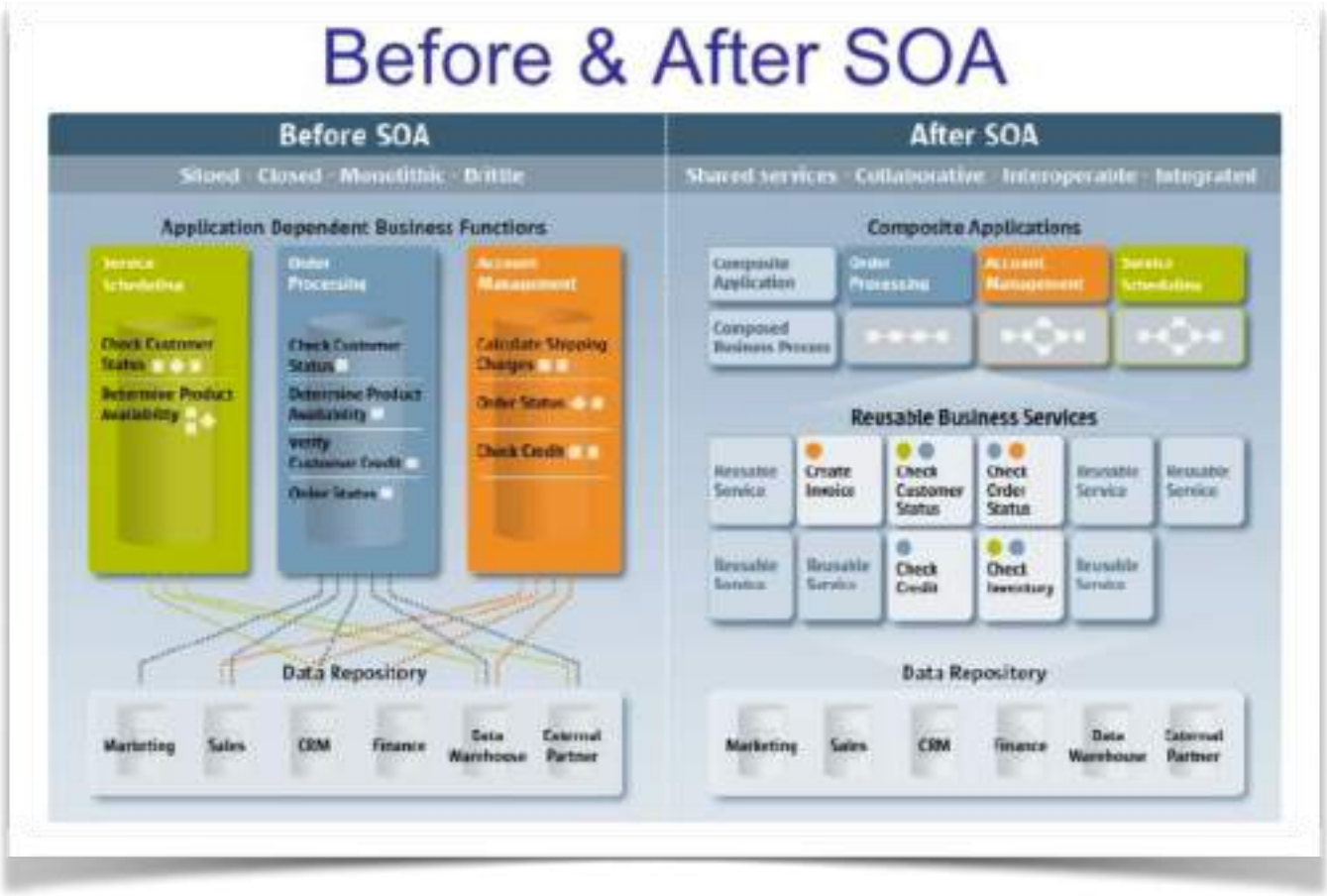
Quello più comune è il **publish/subscribe**. In questo modello, le applicazioni che mandano messaggi non specificano il destinatario esplicitamente ma si limitano a pubblicarlo (*publish*) al middleware che controlla l'interazione. Dall'altra parte, se un'applicazione è interessata a ricevere messaggi di specifici tipi, deve registrare (*subscribe*) i propri interessi (tipi dei messaggi desiderati) al middleware. Tutte le volte che il middleware riceve un messaggio, lo inoltra a tutte le applicazioni che sono interessate a quel tipo di messaggio.

Politiche di subscribe:

- Per tipo di messaggio (possibile gerarchia)
- Per parametri

Web Services

SOA



Con SOA si indica generalmente un'architettura software adatta a supportare l'uso di servizi Web per **garantire l'interoperabilità tra diversi sistemi** così da consentire l'utilizzo delle **singole applicazioni** come componenti del processo di business e soddisfare le richieste degli utenti in modo integrato e trasparente.

Il manifesto

L'orientamento ai servizi è il paradigma che circoscrive quello che fai. L'architettura orientata ai servizi (SOA) è il tipo di architettura fondata sull'applicazione dell'orientamento ai servizi. Applichiamo l'orientamento ai servizi per favorire in un modo consistente le organizzazioni nel **fornire prestazioni di business sostenibile, con maggiore agilità di impiego ed efficienza nei costi, adattandosi alle mutevoli esigenze aziendali.** La nostra esperienza ci induce a dare priorità:

- **Al valore di business** rispetto all'aspetto tecnico
- **Agli obiettivi strategici** rispetto ai benefici specifici di un progetto
- **All'interoperabilità** intrinseca rispetto all'integrazione personalizzata
- **Ai servizi condivisi** rispetto alle implementazioni particolari
- **Alla flessibilità** rispetto all'ottimizzazione
- **Al miglioramento evolutivo** rispetto alla ricerca della perfezione iniziale

Descrizione
Nell'ambito di un'architettura Service-Oriented Architecture è quindi possibile **modificare**, in maniera relativamente più semplice, le modalità di interazione tra i servizi, oppure la combinazione nella quale i servizi vengono utilizzati nel processo. Inoltre, risulta più agevole **aggiungere nuovi servizi** e **modificare i processi per rispondere alle specifiche esigenze di business**. Così facendo, il processo di business non è più vincolato da una specifica piattaforma o da un'applicazione; ma può essere considerato come un componente di un processo più ampio e quindi riutilizzato o modificato.[6]

L'architettura orientata ai servizi è particolarmente adatta per le aziende che presentano una discreta complessità di processi e applicazioni. Infatti, viene agevolata l'interazione tra le diverse realtà aziendali. Le attività di business ora possono sviluppare processi efficienti sia internamente che esternamente. Parallelamente aumenta la flessibilità e l'adattabilità dei processi.

La chiave sta nella totale assenza di business logic sul client SOA, il quale è totalmente agnostico rispetto alla piattaforma di implementazione, riguardo ai protocolli, al binding, al tipo di dati, alle policy con cui il servizio produrrà l'informazione richiesta. Tutto a beneficio dell'indipendenza dei servizi, che possono essere chiamati per eseguire i propri compiti in un modo standard, senza che il servizio abbia conoscenza dell'applicazione chiamante e senza che l'applicazione abbia conoscenza, o necessiti di averne, del servizio che effettivamente eseguirà l'operazione.
Service-Oriented Architecture può anche essere vista come uno stile dell'architettura dei sistemi informatici che permetta la creazione delle applicazioni sviluppate, combinando servizi debolmente accoppiati e interoperabilità degli stessi. Questi servizi interagiscono secondo una definizione formale, detta protocollo o contratto, come per i **Web Services Description Language** indipendente dalla piattaforma sottostante e dalle tecnologie di sviluppo (come Java, .NET, ecc.). Per esempio, i servizi scritti in Java usando la piattaforma Java EE e quelli in C# con .NET possono essere utilizzati dall'applicazione sovrastante. Le applicazioni in esecuzione su una piattaforma possono anche utilizzare servizi in esecuzione su altre, come con i Web services, facilitando quindi la riusabilità.
Service-Oriented Architecture **può supportare l'integrazione** e la consolidazione di attività all'interno di complessi sistemi aziendali (sistemi di EAI) ma non specifica o fornisce la metodologia o il framework per documentare capacità e potenzialità dei servizi.

Web Services

Cosa sono: è un sistema software progettato per supportare l'interoperabilità tra diversi elaboratori su di una medesima rete ovvero in un contesto distribuito. Tre aspetti chiave:
- progettati per l'interazione macchina-macchina (o applicazione-applicazione);
  es. un'applicazione web normale che restituisce semplicemente pagine html è designata per l'interazione uomo-macchina, infatti, un'altra applicazione non può consumare l'html (a meno che non faccia web scraping)
- progettati per essere interoperabili (non *platform dependent, Windows/Linux, Java/C#…*);
- devono permettere la comunicazione via web (in modo che siano facilmente consumabili da altre applicazioni).

Panoramica
Tale caratteristica si ottiene associando all'applicazione un'**interfaccia software** (descritta in un formato automaticamente elaborabile quale, ad es., il **Web Services Description Language**) che espone all'esterno il servizio/i associato/i e utilizzando la quale altri sistemi possono interagire con l'applicazione stessa attivando le operazioni descritte nell'interfaccia (servizi o richieste di procedure remote) tramite appositi "messaggi" di richiesta: tali messaggi di richiesta sono inclusi in una "busta" (la più famosa è **SOAP**), formattati secondo lo **standard XML**, incapsulati e trasportati tramite i protocolli del Web (solitamente **HTTP**), da cui appunto il nome web service.

---

[6] Paragone con una singola grande funzione vs tante piccole funzioni

Proprio grazie all'utilizzo di standard basati su XML, tramite un'architettura basata sui Web Service (chiamata, con terminologia inglese, Service oriented Architecture - **SOA**) applicazioni software scritte in diversi linguaggi di programmazione e implementate su diverse piattaforme hardware possono quindi essere utilizzate, tramite le interfacce che queste "espongono" pubblicamente e mediante l'utilizzo delle funzioni che sono in grado di effettuare (i "servizi" che mettono a disposizione) per lo scambio di informazioni e l'effettuazione di operazioni complesse (quali, ad esempio, la realizzazione di processi di business che coinvolgono più aree di una medesima azienda) sia su reti aziendali come anche su Internet: la possibilità dell'interoperabilità fra diversi linguaggi di programmazione (ad esempio, tra Java e Python) e diversi sistemi operativi (come Windows e Linux) è resa possibile dall'**uso di standard "aperti"**.

Caratteristiche

- **Encapsulated:** espone le interfacce ma i dettagli implementativi vengono nascosti
- **Loosely Coupled:** service e consumer sono indipendenti
- **Interfaccia pubblica (contracted software objects):** tutti i web services hanno un'interfaccia pubblica che espone appunto i servizi offerti

Pro

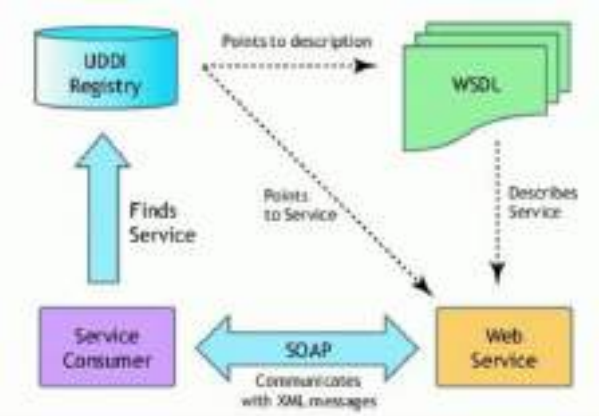
- permettono l'interoperabilità tra diverse applicazioni software su diverse piattaforme hardware
- utilizzano standard e protocolli "open"; i protocolli ed il formato dei dati è, ove possibile, in formato testuale, cosa che li rende di più facile comprensione ed utilizzo da parte degli sviluppatori
- mediante l'uso di HTTP per il trasporto dei messaggi, i Web Service normalmente non necessitano di modifiche alle regole di sicurezza utilizzate come filtro sui firewall
- possono essere facilmente utilizzati, in combinazione l'uno con l'altro (indipendentemente da chi li fornisce e da dove vengono resi disponibili) per formare servizi "integrati" e complessi
- consentono il riutilizzo di infrastrutture ed applicazioni già sviluppate e sono (relativamente) indipendenti da eventuali modifiche delle stesse

Contro

- le performance legate all'utilizzo dei Web Service possono essere minori di quelle riscontrabili utilizzando approcci alternativi di distributed computing quali Java RMI, CORBA, o DCOM
- l'uso dell'HTTP permette ai Web Service di evitare le misure di sicurezza dei firewall (le cui regole sono stabilite spesso proprio per evitare le comunicazioni fra programmi "esterni" ed "interni" al firewall).

Modello

- **SOAP (Simple Object Access Protocol):** il protocollo di richiamo di procedure remote come web services.
- **WSDL (Web Services Description Language):** il linguaggio di definizione dei web services.
- **UDDI (Universal Description, Discovery and Integration):** il protocollo per ricercare i web services, una sorta di "elenco telefonico" o "pagine gialle" dei web services.



Protocolli: TCP/IP, HTTP, XML, SOAP, WSDL, UDDI

Java-WS

JAX-WS (Java API for XML Web Services) è un **insieme di interfacce (API) del linguaggio di programmazione Java dedicate allo sviluppo di servizi web.** L'insieme fa parte della piattaforma Java EE. Come altre API della Java EE, JAX-WS usa annotazioni, introdotte nella Java SE 5, **per semplificare lo sviluppo e implementazioni di client e terminali di servizi web**. JAX-WS fa parte del kit di sviluppo Java per web services (Java Web Service Development Pack – JWSDP) e **include Java Architecture for XML Binding (JAXB) e SOAP**. Uno dei grandi vantaggi di questa implementazione è quella di poter riutilizzare gran parte delle funzionalità, senza toccare server enterprise come il progetto Glassfish.

SOAP - Simple Object Access Protocol

Cos'è: è un **protocollo per lo scambio di messaggi** tra componenti software. La parola object manifesta che l'uso del protocollo dovrebbe effettuarsi secondo il paradigma della programmazione orientata agli oggetti.

Descrizione

SOAP è un framework estensibile e decentralizzato che può operare sopra varie pile protocollari per reti di computer fornendo tramite messaggi **richieste di procedure remote**. I richiami di procedure remote possono essere infatti modellati come interazione di parecchi messaggi SOAP. SOAP dunque è uno dei protocolli che abilitano i servizi Web.

SOAP può operare su differenti protocolli di rete, ma HTTP è il più comunemente utilizzato e l'unico ad essere stato standardizzato dal W3C, su cui è incapsulato (embedded) il relativo messaggio. SOAP si basa sul metalinguaggio **XML** e la sua struttura segue la configurazione **Header-Body**, analogamente ad HTML:

- **Header (opzionale):** meta-informazioni come quelle che riguardano il routing, la sicurezza (autenticazione/autorizzazione), le transazioni e parametri per l'Orchestration.
- **Body:** trasporta il payload, contiene i dati delle chiamate e/o i risultati di ritorno (deve seguire uno schema definito dal linguaggio XML Schema).

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <ns2:getCourseDetailsResponse xmlns:ns2="http://in28mi
            <ns2:course>
                <ns2:id>Course1</ns2:id>
                <ns2:name>Spring</ns2:name>
                <ns2:description>10 Steps</ns2:description>
            </ns2:course>
        </ns2:getCourseDetailsResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Esempio di SOAP response

WSDL

Cos'è: è un linguaggio formale in formato XML utilizzato per la creazione di "documenti" per la descrizione di Web Service.

Descrizione

Mediante WSDL può essere infatti **descritta l'interfaccia pubblica di un Web Service** (ovvero una descrizione basata su XML) che indica come interagire con un determinato servizio: un "documento" WSDL contiene infatti, relativamente al Web Service descritto, informazioni su:

- **cosa** può essere utilizzato (le "operazioni" messe a disposizione dal servizio)
- **come** utilizzarlo (il protocollo di comunicazione da utilizzare per accedere al servizio, il formato dei messaggi accettati in input e restituiti in output dal servizio ed i dati correlati) ovvero i "vincoli" (bindings in inglese) del servizio;
- **dove** utilizzare il servizio (cosiddetto endpoint del servizio che solitamente corrisponde all'indirizzo - in formato URI - che rende disponibile il Web Service)

Le operazioni supportate dal Web Service ed i messaggi che è possibile scambiare con lo stesso sono descritti in maniera astratta e quindi non collegati ad uno specifico protocollo di rete e ad uno specifico formato.

Il WSDL è solitamente utilizzato in combinazione con SOAP e XML Schema per rendere disponibili Web Services su reti aziendali o su internet: un programma client può, infatti, "leggere" il documento WSDL relativo ad un Web Service per determinare quali siano le funzioni messe a disposizione sul server e quindi utilizzare il protocollo SOAP per utilizzare una o più delle funzioni elencate dal WSDL.

Spiegazione delle componenti: https://www.w3schools.com/xml/xml_wsdl.asp

An WSDL document describes a web service. It specifies the location of the service, and the methods of the service, using these major elements:

Element	Description
<types>	Defines the (XML Schema) data types used by the web service
<message>	Defines the data elements for each operation
<portType>	Describes the operations that can be performed and the messages involved.
<binding>	Defines the protocol and data format for each port type

The main structure of a WSDL document looks like this:

```
<definitions>

<types>
  data type definitions.....
</types>

<message>
  definition of the data being communicated....
</message>

<portType>
  set of operations.....
</portType>

<binding>
  protocol and data format specification....
</binding>

</definitions>
```

UDDI

Cos'è: è un registro (base di dati ordinata e indicizzata), basato su XML e indipendente dalla piattaforma hardware, che permette di descrivere, pubblicare e cercare WS (e i relativi suppliers).

Cosa si può registrare

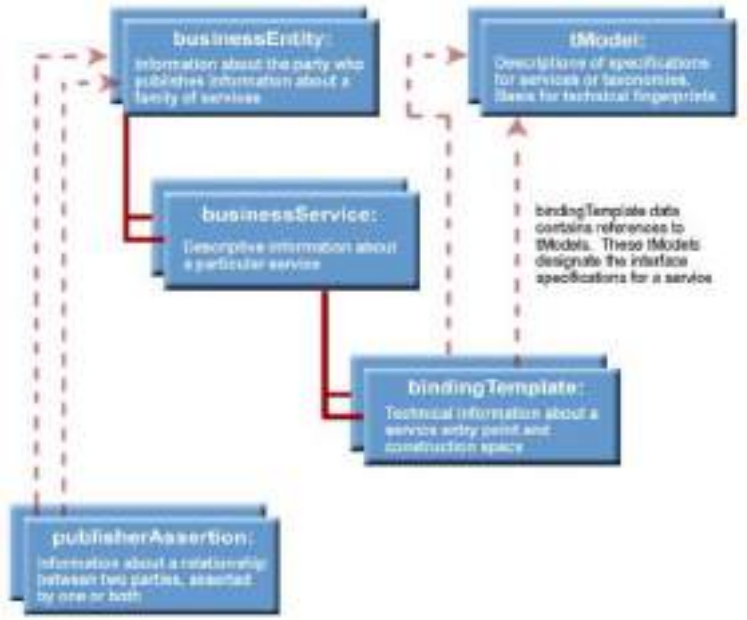
Una compagnia può registrare tre tipi di informazioni in un registro UDDI:

- **Pagine bianche:** informazioni riguardanti l'azienda (ID univoco della compagnia, nome, indirizzo, contatti…).
- **Pagine gialle:** categorizzazione dei servizi basata su tassonomie standardizzate.
- **Pagine verdi:** informazioni (tecniche) dei servizi forniti dall'azienda (interfacce, locazione…).

Architettura

L'architettura di UDDI prevede tre componenti:

- **UDDI Data Model**[7]**:** XML Schema che descrive i web services.
- **UDDI API:** SOAP based API per pubblicare e cercare UDDI data.
- **UDDI Cloud Services:** siti che implementano UDDI e che fanno parte del UBR (UDDI Business Registry) anche noto come Public Cloud, un sistema composto da diversi nodi aventi i dati sincronizzati tramite replication (ogni 24h). E' anche possibile creare registri privati che non fanno quindi parte di questo cloud.



Data Model

Data Structure

UDDI con WSDL: https://www.tutorialspoint.com/uddi/uddi_with_wsdl.htm

Quando UDDI viene utilizzato per archiviare informazioni WSDL o puntatori ai file WSDL, *tModel* deve essere denominato, per convenzione, di tipo *wsdlSpec*, il che significa che l'elemento *overviewDoc* è chiaramente identificato come riferimento a una definizione dell'interfaccia di servizio WSDL.

[7] https://www.tutorialspoint.com/uddi/uddi_data_model.htm

RESTful Web Services

https://it.wikipedia.org/wiki/Representational_State_Transfer

Vedere video udemy su SOAP e prendere nota della differenza tra SOAP e REST.

Microservizi

https://www.salvatorecordiano.it/che-cosa-sono-i-microservizi/

https://en.wikipedia.org/wiki/Microservices

Differences between SOAs and Microservices

- SOAs are Stateful while Microservices are Stateless
- SOAs tend to use Enterprise Service Bus (ESB) while Microservices use a less elaborate and simple messaging system
- SOAs are composed by more lines of code than Microservices (even less than 100)
- SOAs put more emphasis in reusability whereas Microservices focus on decoupling as much as possible
- A change in SOAs require a whole change in the monolithic application
- SOAs use more often relational DB whereas Microservices gravitate more towards non-relational DB

Docker: https://it.wikipedia.org/wiki/Docker

Cloud Computing

Cos'è: paradigma di erogazione di risorse informatiche, come **l'archiviazione**, **l'elaborazione** o la **trasmissione** di dati, caratterizzato dalla disponibilità **on demand** attraverso **Internet** a partire da un insieme di **risorse preesistenti, configurabili e condivise**.

Caratteristiche:

- **On demand services:** servizi utilizzati solamente quando servono.
- **Network access:** Internet usato come canale di accesso.
- **Shared resources:** risorse raggruppate e condivise da più utenti.
- **Flessibilità:** le risorse vengono istanziate su richiesta solamente quando servono.

Si possono distinguere tre tipologie fondamentali di servizi cloud computing:

- **Saas (Software as a Service):** servizio per l'utilizzo di programmi remoti pagando per singolo utilizzo. L'istanza è unica e condivisa da più utenti, che vi accedono tipicamente da browser.
  Chi lo usa? Utenti finali.
  Esempi: Gmail, Google Docs, Google Drive, Microsoft Office 365…
- **PaaS (Platform as a Service):** servizio di ambienti di sviluppo (linguaggi, SO, DB, web server…).
  Chi lo usa? Sviluppatori.
  Esempi: Google App Engine, Microsoft Azure, AWS…
- **IaaS (Infrastructure as a Service):** servizi che mettono a disposizione risorse e infrastrutture hardware come server, capacità di rete, sistemi di memoria, archivio e backup. Le risorse sono virtualizzate per poter essere condivise tra più utenti e inoltre vengono stanziate solamente quando richieste.
  Chi lo usa? System Admins.

https://www.youtube.com/watch?v=36zducUX16w

https://www.youtube.com/watch?v=uroryFU78gM