SISTEMI DISTRIBUITI

INTRODUZIONE

Definizione: Si definisce sistema distribuito una collezione di computer che appare ai suoi utenti come un singolo sistema coerente.

Caratteristiche:

- **mancanza di memoria condivisa:** le informazioni vengono scambiate tramite messaggi
- **non esiste uno stato globale dell'intero sistema**
- Ogni componente è autonomo
  - c'è bisogno di coordinare **l'esecuzione concorrente** dei diversi componenti del sistema
- non esiste un clock globale
  - impossibilità di un fallimento completo del sistema, ma solo di fallimenti indipendenti dei singoli componenti.

Tipi di architetture:

- Architettura a strati : come i sistemi operativi e i middleware
- Architettura a livelli : applicazioni client-server
- Architettura ad oggetti: Java Remote Method Invocation (RMI)
- Architettura incentrata sui dati: Web come file system condiviso
- Architettura ad eventi

Sistema Operativo di Rete (Detto anche NOS - Network operative system)

- Particolare tipo di sistema distribuito in cui l'applicazione distribuita "gira" su più macchine, ognuna con il suo OS ma con uno strato intermedio che gestisce i servizi di rete.
- Gli utenti sono a conoscenza della molteplicità delle macchine, e l'accesso alle risorse può avvenire in modi diversi, come il remote logging via SSH o TELNET, via desktop remoto o via FTP.
- Diversi client possono accedere alle risorse in posti diversi e attraverso diverse modalità.

Middleware

- Software per migliorare le prestazioni e la trasparenza di distribuzione di un NOS
- nasconde l'eterogeneità delle macchine condivise fornendo un'interfaccia identica a tutti verso i servizi comuni.
- permette la definizione di un sistema di comunicazione che nasconde i dettagli dello scambio dei messaggi
- permette di definire nomi simbolici che possono essere scambiati e/o cercati
- definisce meccanismi per l'esecuzione atomica di operazioni di read/write e livelli di protezione dell'accesso ai dati e dell'integrità della computazione.
- Basati su paradigmi di distribuzione e comunicazione, come le RPC o gli oggetti distribuiti, tramite diversi livelli di astrazione (dal più generico TCP/IP ai livelli dedicati come l'ODBC per i database o l'HTTP per internet o ancora l'RMI per gli oggetti).

Sistema Operativo Distribuito (DOS – Distributed Operative System) (confronto con NOS)

- non interpone i servizi di rete tra le applicazioni distribuite e il client, ma uno strato condiviso di servizi di sistema per il controllo delle risorse siano esse dati o risorse di sistema, come capacità computazionale o memoria.
- In questo sistema gli utenti non sono a conoscenza della molteplicità delle macchine, l'accesso alle risorse condivise è identico all'accesso a risorse locali
- possibilità di migrare processi o parti di essi da una macchina all'altra per migliorare le prestazioni dei processi stessi, tramite politiche di bilanciamento del carico, di efficienza dell'hardware.

Trasparenza (aspetto importante dei sistemi distribuiti): necessità di nascondere all'utente finale i dettagli di come avvengono determinate funzionalità del sistema. (più c'è trasparenza, meno l'utente finale sa come funziona il sistema)

Categorie di trasparenza:

- Trasparenza d'accesso : Permette l'accesso a risorse locali e remote con le stesse operazioni e stesso formato dei dati

- Trasparenza di località : Permette l'accesso a risorse remote senza conoscere la loro locazione fisica nella rete
- Trasparenza di concorrenza : Permette a più processi di operare con risorse condivise senza interferenze
- Trasparenza di replicazione : Permette l'uso di istanze multiple delle risorse senza intaccarne l'affidabilità e l'integrità
- Trasparenza di fallimento : Permette di sanare eventuali fallimenti parziali del sistema senza che l'utente ne abbia evidenza
- Trasparenza di mobilità : Permette il trasferimento di dati, utenti e sw su macchine diverse senza impatto utente
- Trasparenza di performance : Permette la riconfigurazione dinamica (load balancing) senza impatto
- Trasparenza di persistenza : Permette di nascondere una risorsa sia in memoria sia su disco

Scalabilità: capacità di un sistema di mantenere le stesse prestazioni all'aumentare delle richieste di risorse

Tecniche di scalabilità

Gli elementi costruttivi di un Sistema Distribuito sono servizi, dati e algoritmi. Se essi fossero centralizzati su una stessa macchina, si avrebbero problemi di performance/scalabilità del sistema. Esistono quindi diverse tecniche, da poter applicare ad elementi distribuiti, che permettono di aumentare il grado di scalabilità di un sistema

- **comunicazione asincrona:**comunicazione tra i componenti non bloccante (permette quindi un elevato grado di trasparenza di concorrenza)
- **fat client vs. fat server**:delega dei compiti ai nodi meno impegnati
  - fat client=client con molte funzionalità che è meno dipendente dal server
  - fat server=server con molte funzionalità al quale dipendono maggiormente i client collegati
- **ripartizione dei compiti**:parallelizzazione delle attività tramite ripartizione gerarchica come nei DNS o tramite pipeline
- **replicazione:**più nodi forniscono lo stesso servizio(ma ciò introduce problemi di consistenza dei dati).
- **caching** delle informazioni: salvataggio temporaneo di dati recenti nella memoria centrale per velocizzare le prestazioni in caso di richieste multiple della stessa risorsa (può essere visto come una forma di replicazione lato client)

MODELLO CLIENT-SERVER

- L'architettura di base prevede che un client acceda ad un server con una richiesta e che il server risponda con un risultato.
- La richiesta può anche essere "girata" da un server ad un altro (se le informazioni non risiedono sul server contattato, oppure se esso è molto carico), caso in cui si parla di **accesso a server multipli**, oppure può avvenire tramite proxy server, che funge da intermediario per le comunicazioni in entrata/uscita.
- Il modello di interazione tra processi client e processi server è un modello sincrono, in cui il client che ha emesso la richiesta attende che il server la elabori e restituisca una risposta prima di procedere.
- Le comunicazioni tra client e server si basano su **protocolli applicativi** che definiscono il formato, l'ordine di invio e ricezione dei messaggi, i tipi di dati e le azioni da eseguire quando si riceve un messaggio.
- Esempi di protocolli applicativi sono l'HTTP, l'FTP e l'SMTP.

Aspetti critici

- **ripartizione dei compiti fra client e server**: chi-fa-cosa dipende dal tipo di operazione (es. in una banca, tutto è gestito lato server) ma ciò influenza le operazioni in base al carico
- **identificazione e accesso al server**: il client deve avere le informazioni sufficienti all'accesso al server
- **comunicazione tra client e server:** le primitive disponibili e le modalità per la comunicazione.
- **gestione del ciclo di vita di client e server**: bisogna regolamentare l'attivazione e la terminazione del client e del server.

Alternativa valida: architettura a più livelli, in cui esistono dei server che si comportano anche da client, richiedendo informazioni ad ulteriori server posti nella rete. (es: Server di presentazione server applicazione server db).

Come fa il client a conoscere l'indirizzo del server?

Esso può essere espresso come costante (nel caso di applicazioni client es. client di servizio bancario), oppure può essere inserito dall'utente (es: web browser), ottenuto via un name server o un repository da cui il client può ottenere le informazioni necessarie (DNS) o tramite un protocollo diverso per l'individuazione del server (Broadcast per DHCP).

MODELLO P2P

- Nel modello peer-to-peer (P2P) non esiste un server definito, e tutti i client si comportano e partecipano al sistema in egual misura.
- Le comunicazioni avvengono in maniera asincrona e attraverso lo scambio di messaggi, eventualmente basati comunque su un protocollo client/server durante la connessione (TCP).
- I sistemi P2P si organizzano e gestiscono autonomamente, tramite un controllo decentrato grazie all'adattabilità della morfologia di rete,

Aspetti critici

- difficoltà nell'**individuazione dei partecipanti**,
- gestione di **connessione e comunicazione** tra i peer
- gestione di **inserimento e cancellazione** di peer/processi
- **comunicazione multicast e broadcast**.

Fasi principali:

- **BOOT:** permette ad un peer di trovare una rete e di collegarsi ad essa (anche se nessuno ormai fa boot P2P, questa operazione viene vista più come un'operazione client/server tramite i cosiddetti superpeer)
- **LOOKUP:** permette ad un peer di trovare con chi interagire (il proprietario del file cercato / la persona desiderata per chattare). Anche questa operazione viene effettuata tramite Superpeer.
- **INTERAZIONE:** due peer iniziano la loro interazione (chat/scambio file)
  - **pure** se le 3 operazioni avvengono tutte peer-to-peer
  - **P2P** se le fasi di lookup e interazione sono svolte peer-to-peer mentre l fase di boot prevede l'utilizzo di qualche server
  - **P2P ibride** che l'unica fase realmente peer-to-peer è lo scambio, mentre boot utilizza qualche server e il lookup viene svolto da peer particolari detti Superpeer/Hub/MainPeer/Server (principalmente nelle reti di scambio file).

COMUNICAZIONE

Tipologie principali di comunicazione d'informazioni

- **Asincrona**:il mittente non attende una risposta del ricevente prima di procedere
- **Persistente**: il middleware memorizza i dati fino alla consegna del messaggio al ricevente (non è necessario che i processi mittente/destinatario siano in esecuzione prima e dopo l'invio/ricezione dei messaggi)
  - **sincrona:** il mittente spedisce un messaggio e attende la conferma di ricezione da parte del ricevente (se attivo) o del middleware (se il ricevente non è attivo). All'arrivo della conferma il mittente riprende l'elaborazione
  - **asincrona:** il mittente spedisce un messaggio ma non attende la conferma prima di procedere. Si affida quindi totalmente alle capacità di consegna del protocollo di comunicazione scelto.
- **transiente:** se il destinatario non è connesso i dati non vengono scaricati.
  - **Asincrona transiente:** il mittente invia un messaggio ma solo dopo che è consapevole che il destinatario sta girando, dopodiché procede nella normale elaborazione
  - **sincrona transiente receipt-based** (basata su un ritorno): il messaggio viene inviato ma magari il ricevente è impegnato in altro. A quel punto, il mittente si ferma, aspettando un messaggio di ricevuta (ACK) del ricevente (di solito il ricevente si interrompe alla ricezione del messaggio solo per spedire indietro l'ACK oppure la gestione dei messaggi è compiuta in un secondo thread del processo ricevente). Ricevuto l'ACK del ricevente, il mittente continua la sua elaborazione. Nel frattempo il ricevente però non ha ancora iniziato ad elaborare il messaggio
  - **sincrona transiente basata sulla consegna:** molto simile al secondo, solo che l'attesa del mittente si prolunga fin quando il ricevente non inizia ad elaborare il messaggio, momento in cui esso invia l'ACK
  - **sincrona transiente basata sulla risposta:** , l'ACK viene posticipato al momento in cui il ricevente ha completato l'elaborazione del messaggio, prolungando ancora l'attesa del mittente

La comunicazione persistente viene utilizzata per lo più per lo scambio dei messaggi senza richiedere che né il mittente né il ricevente siano attivi durante la trasmissione. Si possono creare quindi 4 combinazioni attivo A e passivo P (mittente-ricevente): A-A A-P P-A P-P

Interfacce basilari per le code in un sistema di scambio messaggi

- **PUT** – inserisce un messaggio nella coda specificata
- **GET** – blocca finchè la coda specificata non è vuota, e rimuove il primo messaggio

- **POLL** – controlla se la coda specificata per i messaggi e rimuove il primo senza bloccare
- **NOTIFY** – installa un handler che viene chiamato quando un messaggio è presente nella coda specificata

Come identificare correttamente un processo con il quale comunicare

- è necessario conoscere l'indirizzo IP dell'host su cui il processo è in esecuzione, e il numero di porta associato al processo.
- Usando delle socket, ossia delle API (application programming interface) del sistema operativo che collegano un processo ad una determinata porta TCP/UDP in modo da poter interfacciarsi

Primitive per l'utilizzo delle socket

- **SOCKET:** crea un nuovo socket (punto di collegamento)
- **BIND:** collega un socket ad un indirizzo locale
- **LISTEN:** annuncia la volontà di accettare connessioni in entrata sul socket
- **ACCEPT:** blocca il processo finchè non arriva una richiesta di connessione
- **CONNECT:** tenta una connessione in uscita
- **WRITE:** scrive sulla connessione
- **READ:** legge dalla connessione
  - Attenzione: La lettura/scrittura su una socket basata su un canale condiviso da più processi può determinare problemi di concorrenza per l'accesso multiplo ad una risorsa condivisa. E' quindi necessario adottare meccanismi di mutua esclusione.
- **CLOSE:** chiude la connessione

Mentre un client è abbastanza semplice da realizzare, un processo server deve tenere conto dell'affidabilità del server stesso e del possibile carico al momento della connessione da parte di più client.

Come definire un server

- **Iterativo:**soddisfa una richiesta alla volta
  - quando arriva una richiesta di connessione il server crea una socket temporanea per stabilire una connessione diretta con il client, mentre altre richieste vengono accodate sulla porta nota per essere soddisfatte successivamente
  - calo delle prestazioni per gli altri client
- **Concorrente:** soddisfa più richieste alla volta
  - **a processo singolo (simulata):** simula la presenza di un server dedicato ad ogni richiesta con select()
    - **select()**: funzione che permette di leggere in modo asincrono dai diversi canali di I/O disponibili sul server, e sospende il processo finchè non esiste almeno un canale libero, ritorna uno dei canali disponibile per la lettura/scrittura
  - **multi processo**: vengono realmente creati nuovi processi server ad ogni richiesta con fork()
    - **fork()**: funzione in C, realizza un esatta copia del processo per prendere cura della connessione in ingresso
  - **multi thread:** vengono creati thread diversi dello stesso processo per soddisfare le richieste, implementandoli in java/C# in modi diversi:
    - **1 thread per client:** Il thread coordinatore rileva un nuovo client e lo connette ad un nuovo thread che si occupa della richiesta e, al momento della disconnessione del client, si chiude
    - **1 thread per richiesta:** il thread coordinatore riceve una richiesta e genera un thread figlio per processarla, che si chiude al termine della richiesta stessa.
    - **1 thread per servente:** ogni servente ha un proprio thread ed una coda, il coordinatore riceve una richiesta e la inserisce nella coda del servente giusto (servente = tipo di richiesta). Ogni thread servente legge ciclicamente dalla propria coda ed esegue le richieste.
    - **Pool di thread:** dato che la creazione dei thread è costosa, il costo di creazione può essere ammortizzato facendo gestire ad un thread diverse richieste. Un pool di thread viene creato all'avvio del sistema e assegnato alle richieste man mano che esse arrivano.

Classi (in Java) delle socket: java.net.Socket o java.net.ServerSocket

Costruttori client:

- public Socket(); //costruisce un socket base, non connesso
- public Socket(String host, int port); //costruisce un socket e lo collega alla porta sull'host
- public Socket(InetAddress address, int port), //simile a sopra, ma con un altro modo per specificare l'indirizzo

Connessioni client:

- public void bind(SocketAddress bindpoint); //collega il socket all'indirizzo locale

- public void connect(SocketAddress endpoint, int timeout); //connette il socket al server con il timeout specificato in millisecond
- public void close(); //chiude il socket

Comunicazioni client:

- Public InputStream getInputStream();
- Ritorna uno stream di input per il socket specificato, da cui leggere i dati
- Public OutputStream getOutputStream();
- Ritorna un output stream per la scrittura di bytes sul socket.

REMOTE PROCEDURE CALL

Definizione: Con RPC (Remote Procedure Call) si riferisce all'attivazione da parte di un programma di una "procedura" o subroutine attivata su un computer *diverso* da quello sul quale il programma viene eseguito .

- L'RPC consente a un programma di eseguire subroutine "a distanza" su computer "remoti" , accessibili attraverso una rete.
- Essenziale al concetto di RPC è l'idea di trasparenza: la chiamata di procedura remota deve essere infatti eseguita in modo il più possibile analogo a quello della chiamata di procedura "locale"; i dettagli della comunicazione su rete devono essere "nascosti" (resi *trasparenti*) all'utilizzatore del meccanismo.

Vantaggi

- sono un'estensione alle normali chiamate di procedure, molto semplici da realizzare in quanto hanno una semantica identica alla chiamata di procedura locale e sono facili da implementare.

Svantaggi

- sono completamente esplicite (cioè sono tutte scritte dal programmatore)
- sono statiche (scritte nel codice del sw)
- bloccanti
  - possibile soluzione: utilizzare delle procedure asincrone, in cui il client chiamante non attende il risultato della procedura prima di procedere, ma soltanto un messaggio di "accettazione" da parte del server. Quando poi il server completerà l'esecuzione della procedura, invierà i risultati al client che subirà quindi un'interruzione rispondendo con un messaggio di ACK al server.

Stub (usati per creare le RPC)

- modelli di procedura (dichiarazioni/signature) presenti sia sul client che sul server
- vengono utilizzati per simulare comportamenti locali per client e server e per realizzare la comunicazione tra processi remoti.

Client StubClient(a,b)StubServer(a,b)Server

- I parametri presenti nella chiamata alla procedura sono passati per valore o reference, e vengono "impacchettati", spediti, "spacchettati" e riprodotti attraverso una tecnica chiamata **marshalling** dei dati.

REMOTE METHOD INTERFACE – OGGETTI DISTRIBUITI

Definizione: La **Java Remote Method Invocation** (**Java RMI**) è un'API java che è l'equivalente versione a oggetti delle Remote Procedure Calls (RPS), con supporto al trasferimento diretto di classi Java serializzate e garbage collection distribuita

Caratteristiche:

- **Struttura a oggetti**
  - Incapsulamento dati (variabili che ne definiscono lo stato)
  - Metodi che definiscono le operazioni sui dati
  - Interfaccia d'accesso.
- **Oggetti compile-time**
  - cioè essere definiti attraverso classi Java/C++,
- **Oggetti a run-time**
  - accessibili attraverso degli adattatori chiamati wrappers
- **oggetti persistenti e transienti**
- **riferimento agli oggetti**:
  - INDIRIZZO IP HOST

    - PORTA SERVER
    - INDIRIZZO OGGETTO (ID)
- RMI è un middleware che fornisce servizi come il garbage collection degli oggetti remoti, il caricamento e controllo di un class loader per gli oggetti compile-time, l'attivazione automatica degli oggetti e così via
- l'invocazione di metodi tra oggetti in macchine virtuali differenti (Ciò significa che se l'oggetto A risiede su una macchina ma ha bisogno dell'oggetto B presente su un'altra macchina per poter funzionare, RMI riesce a mascherare questa invocazione all'utente senza problemi)
- In particolare l'RMI per Java, chiamato JavaRMI, utilizza anch'esso degli stub in maniera simile all'RPC per gestire i parametri a valore, consentendo però anche il passaggio di parametri per reference definendo ulteriori stub specifici per ogni oggetto.

Invocazioni agli oggetti:

- **Statiche**: l'interfaccia dell'oggetto è nota in compilazione, mentre sono
- **Dinamiche**: sfruttano alcune informazioni logiche sull'identità dell'oggetto e del metodo Invoke(obj,method,input_par,output_par)

Riferimenti: Sempre in JavaRMI I riferimenti ad oggetti remoti sono passati per valore per permettere invocazioni remote, sempre se essi sono serializzabili (implementano cioè l'interfaccia Remote e Serializable e ridefinendo i metodi writeObject(ObjectOutputStream out) e readObject(ObjectInputStream in)).

Serializzazione:rappresentazione di un oggetto come uno stream di byte, essenziale per poter memorizzare, trasferire e ricostruire lo stato dell'oggetto.

Identificazione oggetti: Per poter identificare un oggetto una volta passato, vengono utilizzati dei directory service che sono disponibili ad un host e porta conosciuti. Per JavaRMI viene definito quindi un **rmiregistry** che sta su ogni macchina che ospita servizi remoti e risponde alla porta 1099. Esso fornisce funzionalità di Lookup (e bind per il server), e funziona in questo modo:

- Il server pubblica l'oggetto remoto nel Registry con BIND
- Il client ottiene il reference all'oggetto remoto con LOOKUP
- Il client accede all'oggetto remoto

HTTP – HTML5 – CSS

HTTP(HyperText Transfer Protocol): è un protocollo usato come principale sistema per la trasmissione d'informazioni sul web ovvero in un'architettura tipica client-server.

- il client che è rappresentato da un Browser mentre il server è un WebServer.
- **si basa sul protocollo di rete TCP/IP.** Infatti esso inizia una connessione TCP (tramite un socket) sulla porta 80, connessione che viene accettata da un web server che inizia lo scambio di messaggi HTTP con il client che, infine, chiude la connessione.
- **Protocollo transiente** (il server non mantenga informazioni sulle richieste precedenti provenienti dal client)

Ipertesto: insieme di testi o pagine leggibili con l'ausilio di un'interfaccia elettronica, in maniera non sequenziale, tramite particolari parole (hyperlink, link) che costituiscono una rete raggiata o variamente incrociata di informazioni organizzate secondo criteri paritetici o gerarchici

Messaggi http

- tipologie di messaggi HTTP: **request** e **response**.
- **REQUEST**
    - Tutti i messaggi sono in formato ASCII e contengono diverse linee per comando.
    - **Formato**

COMANDO		URL		VERSIONE
NOME_CAMPO_HEADER	:	VALORE		
NOME_CAMPO_HEADER	:	VALORE		
NOME_CAMPO_HEADER	:	VALORE		

DATI

- Alcuni campi dell'header:
- **HOST :** Specifica l'host da cui ottenere la pagina
- **CONNECTION :** Specifica se mantenere aperta o chiudere la connessione dopo questo messaggio (HTTP 1.1)
- **USER-AGENT :** Contiene informazioni sul client, usato per finalità statistiche e di tracciamento dell'uso del protocollo
- **ACCEPT :** Specifica il tipo di dato o di mime type che sono accettabili dal client
- I principali metodi utilizzati:

Nome	Descrizione	Cache	Sicuro	Idempotente
GET	recupera ogni informazione identificata dall'URI di richiesta	X	X	
HEAD	Come il get, ma ottiene solo le informazioni dell'header, infatti non ottiene il body di risposta	X	X	
POST	Richiede che il server accetti l'entità inclusa nei dati come nuova risorsa sottostante all'URI di richiesta			
PUT	Richiede di mettere l'entita chiusa sotto l'URI di richiesta (se l'URI si riferisce a una risorsa già esistente, modifica la risorsa; se l'URI non si riferisca a una risorsa, il server può creare una nuova risorsa con quell'URI)			X
DELETE	Cancella la risorsa specifica			X
OPTIONS	Richiede informazioni sulle opzioni di comunicazione (metodi HTTP) disponibili di uno specifico URI di richiesta (usato per capire le funzionalità di un Webserver)			X
TRACE	Invia indietro (messaggio d'eco/loopback) la richiesta ricevuta (così un client può vedere se ci sono modifiche o aggiunte fatte dai server intermedi)			X

- **Cache:** usa la memoria cache per salvare temporaneamente le risorse più richieste in modo da velocizzare i tempi d'accesso
- **Sicuro**(safe)**:** il metodo è inteso solo per ottenere informazioni e non deve cambiare stato al server (non causa effetti collaterali)
- **Idempotente**(idempotent)**:** gli effetti di richieste multiple sono identici a quelli di una richiesta sola (escludendo gli errori)

- **RESPONSE**
- è presente la stessa struttura della request ma con dei codici al posto dei metodi. (esempio: il codice 200 indica che è tutto ok, il 400 che è stata fatta una richiesta incomprensibile, il 404 che il documento richiesto non è stato trovato, 418 sono una teiera ecc…)

MIME types (Multipurpose Internet Mail Extension):specificano i tipi di dati scambiati in una comunicazione. (esempii: **text/plain text/html image/jpeg audio/basic audio/mpeg3 video/mpeg video/avi application/pdf application/msword.)**

Cookies: file di testo salvati su disco e rilasciati dal server che il browser invia al server per gli accessi successivi, per memorizzare le informazioni di autenticazione e preferenze dell'utente, in modo da non doverle inserire ogni volta (esempio: memorizzazione nome utente e password, memorizzazione articoli in un carrello per acquisti online...)
(Per evitare di inviare oggetti già presenti nella cache del client, è possibile inserire una riga all'interno dell'header con parametro "If-modified-since: <data>". In questo modo, il server può limitarsi a rispondere con "HTTP/1.0 304 not modified" se non è necessario ricaricare la pagina, altrimenti con un semplice "HTTP/1.1 200 OK" contenente i dati nuovi)

HTML(HyperText Mark-up Language): linguaggio di markup descrittivo solitamente usato per la formattazione e l'impaginazione dei documenti ipertestuali su internet sottoforma di pagine web.

- La parte testuale dei documenti web è espressa in linguaggi standard come l'HTML e l'XML, più alcuni linguaggi di scripting per arricchire l'interazione come Javascript o altri tipi di oggetti (di solito multimediali) trasmessi con l'encoding MIME.
- **HTML definisce la struttura logica dei documenti, ma non il loro aspetto finale** che dipende dal motore di rendering presente su ogni browser.
- Esso si definisce un linguaggio di mark-up perché prevede dei marcatori (**tag**) per qualificare ogni elemento della pagina. Ogni tag contiene un elemento, e deve essere aperto e chiuso, es: <tag>elemento</tag> oppure <tag elemento />

Struttura tipica di un documento HTML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
        <title> Nuova pagina </title>
</head>
<body>
</body>
</html>
```

Descrizione tag principali

- **Doctype:** specifica la versione del linguaggio in cui è scritto il documento, cioè come il browser deve interpretarlo
- **head:** contiene informazioni di contesto, come una descrizione della pagina, il titolo, le parole chiave, collegamenti a file esterni da caricare (css)
- **body:** racchiude il contenuto della pagina
- **tutto il resto:** Al suo interno esistono tag per specificare i titoli (<H1>…<H6>) e le immagini <IMG SRC="xxx.jpg" ALT="testo alternativo"/>.
- le tag hanno un **significato logico, non grafico** (esempio: <H1> non dice che dimensione o colore debba avere il testo contenuto nel tag. HTML però prevede alcune eccezioni come il tag <FONT> che permette di specificare alcune informazioni puramente grafiche: <FONT COLOR="#808080" SIZE="+2">Testo colorato + grande di 2</FONT>)
- Altre tag
  - Esistono una serie di tag per la formattazione del testo, come che rende il testo corsivo, che lo rende grassetto, <u>che lo sottolinea</u> e che lo racchiude in paragrafo . Si possono anche creare
  -
  - elenchi numerati
  -
  - Oppure
  -
  - Elenchi puntati
  -
  - **Immagini**:
  -
  - **Link:**
  - Testo visualizzato
  - **Tabelle:**
  - <Table border="dimensione_bordo" summary="Testo evidenziato sul mouse over">
  - <caption>Titolo tabella</caption>

- <tr> //riga 1
- <th scope=”col”>Titolo colonna</th>
- </tr>
- <tr>
- <td>valore cella 2.1</td>
- <td>valore cella 2.2</td>
- </tr>
- </table>
- Con un’attenta scelta tra celle e porzioni di testo, titoli e immagini (utilizzabili anche come sfondo) è possibile creare layout complessi per le pagine web. Tramite i tag <thead> e <tfoot> di HTML5 è possibile creare sottotabelle visualizzabili come intestazione o chiusura delle tabelle

Nuovi tag html5:

<article>	Content from an external source - news article, blog, or forum
<aside>	Content aside from the page content
<audio>	Used to embed sound content
<canvas>	Used to draw graphics
<datalist>	Provides a list of predefined options for input controls
<details>	Used to show or hide contents
<embed>	Embeds an external application or interactive content into page
<figcaption>	Caption for the figure tag
<figure>	Specifies self-contained content
<footer>	The footer of a document or section
<header>	Specifies a group of introductory or navigational elements
<keygen>	Specifies a key-pair generator field used in forms

Tag	Description
<mark>	Represents highlighted text
<meter>	Used for measurements with minimum and maximum values
<nav>	Specifies navigation for a document
<output>	Represents the result of a calculation
<progress>	Specifies the state of work in progress
<section>	Defines sections in a document or article
<source>	Used to specify multiple media resources for media elements
<summary>	Defines a visible heading for the details tag
<time>	Used to specify the date or time in a document
<video>	Specifies video, such as a movie clip or video stream
<!-- -->	Comments in source that are not displayed in the browser

- In particolare, i tag <audio> e <video> permettono di caricare elementi multimediali **in-line**. Es:
- <audio controls>

    - <source src=”file.ogg” type=”audio/ogg”>
    - <source src=”file.mp3” type=”audio/mpeg”>
    - Messaggio di alert per browser non supportante audio
- </audio>
- N.b. I file vengono caricati nell’ordine di script, prima il .ogg e poi il .mp3 solo se il browser non supporta il .ogg
- Con HTML5 è possibile anche controllare il tipo di dato inserito in input, con gli attributi **color, date, email,list,number,range,search,tel,url** usati nel tag <input> es: <input type=”date”/> o <input type=”datetime-local”/> che fa un calendario dropdown.

CSS(Cascading Style Sheet) sono documenti, o porzioni di documento HTML, che impostano la formattazione di documenti HTML, XHTML (ad esempio i siti web e relative pagine web)

Vantaggi:

- Separazione tra contenuto di una pagina web (HTML) e il modo in cui il contenuto deve essere presentato (CSS)
- Tempi di caricamento pagine ridotti (perché i CSS possono essere condivisi tra molte pagine)
- Possibile creazione di più layout per tipo di dispositivo (pc, cellulare, tablet...) con un’unica pagina HTML
- Impaginazione molto più precisa e ricca (molti comandi, ereditarietà dei contenuti...)

Struttura:

- Regole in due parti **proprietà:valore**
- Tag associati a ogni regola

Modello tipico

tag {proprietà1 : “valore1”;
proprietà2: “valore2”;}

Esempi

```
BODY {background-color:white}
A:link {color:red}
A:visited {color:maroon}
A:hover {color:purple}
P {font-family:Verdana,Arial,Helvetica,sans-serif;
font-size:10pt;
color:black}
UL {font-family:Verdana,Arial,Helvetica,sans-serif;
font-size:10pt;
color:black;
font-style:italic}
H1 {font-family:"Comic Sans MS";
font-size:18pt;
color:green;
font-weight:bold;
text-align:center}
H2 {font-family:"Comic Sans MS";
font-size:14pt;
color:olive;
font-weight:bold;
text-align:center}
```

```
#contenitore {
    Width:800px;
    margin:0 auto;
} //scritto cosi si applica ad un div con id “contenitore”
```

APPLICAZIONI WEB

- Le applicazioni web consentono di realizzare delle componenti software invocabili tramite browser utilizzando il protocollo applicazione HTTP e il protocollo di rete TCP/IP come se esse fossero delle normali pagine web.
- Data l'enorme diffusione del web, della sua caratteristica indipendenza dalla piattaforma e dal fatto che internet è un sistema aperto, viene quindi conveniente pubblicare un'applicazione web per aumentare considerevolmente il bacino d'utenza.
- Inoltre le applicazioni web hanno l'enorme vantaggio di potersi integrare con altre applicazioni web, consentendo un'esperienza utente sempre più coinvolgente e completa.

Sistema aperto:sistema in cui ogni componente è autonomo e posso aggiungere/togliere altri componenti senza influenzare gli altri collegati (Esempio: inserire un nuovo server senza spegnere tutti gli altri)(java simile a sistema aperto perché finchè la classe non è caricata, posso riscrivere solo la classe senza richiudere tutto il programma)

Implementazioni:

- **programmi compilati da parte del web server**. Su richiesta del client, il web server invoca un eseguibile che può essere scritto in qualunque linguaggio che supporti l'interazione con il web server.
- **script interpretati** scritti nei cosiddetti linguaggi, appunto, di scripting (PHP,Python e Perl) che non vengono compilati ma interpretati ogni volta dal web server tramite un motore di scripting. Quest'alternativa consente la creazione di applicazioni web portabili su qualsiasi piattaforma, aumentando però i tempi di esecuzione in quanto esse devono essere reinterpretate ad ogni esecuzione.

CGI (Common Gateway Interface)(implementazione di programmi compilati da parte del webserver): è una tecnologia standard usata dai webserver per interfacciarsi con applicazioni esterne generando contenuti web dinamici.

- Una CGI legge una HTTP REQUEST in arrivo al web server ed elabora una HTTP RESPONSE
- molto semplice da scrivere (può essere scritta in ogni linguaggio di programmazione)
- scarsa flessibilità (se viene cambiata la base dati di scambio bisogna riscrivere l'interfaccia)
- lentezza nell'interpretazione dei messaggi HTTP (per il casting del tipo di dato e per l'eventuale controllo di correttezza degli stessi)
- mancanza di uno stato dell'applicazione(ogni form è un messaggio autonomo sotto il punto di vista HTTP ed è quindi necessario l'inserimento di campi nascosti nella pagina o di cookie per il mantenimento delle informazioni di stato delle sessioni)

Servlet (implementazione CGI di java)(Java con codice HTML): oggetti scritti in linguaggio Java che operano all'interno di un server web (es. Tomcat, Jetty) oppure un server per applicazioni (es. JBoss, Glassfish) permettendo la creazione di Web Applications

- L'uso più frequente delle servlet è la generazione di pagine web dinamiche a seconda dei parametri di richiesta inviati dal client browser dell'utente al server
- gestita in maniera automatica da un container (engine) quindi non è vincolata ad un particolare server
- Ogni servlet presenta un'interfaccia che definisce il set di metodi riutilizzabili, apportando però uno svantaggio a livello di rigidità dei metodi esponibili.
- Il contanier controlla le servlet in base alle richiesta dei client.
- Le servlet sono residenti in memoria, consentendo cosi il mantenimento dello stato della sessione.
- **ciclo di vita**
  - Il container crea una servlet tramite il metodo **init()** alla prima richiesta di invocazione della servlet stessa
  - L’istanza creata è condivisa da tutti i client (problemi di condivisione=
  - Ogni richiesta genera un Thread che esegue la doXXX appropriata

- essa verrà poi distrutta con il metodo **destroy()** nel caso essa non sia più richiesta e/o alla scadenza di un timeout predefinito ma con l'inserimento di un metodo di sincronizzazione con i client e tenendo traccia dei thread in esecuzione (siccome allo scadere del timer predefinito la servlet può essere ancora in esecuzione.

- **comandi**
  - **getServletConfig() e getServletInfo():** per interrogare la servlet stessa, per ottenere informazioni di configurazione e parametri
  - **service(ServletRequest req, ServletResponse resp)** metodo principale che lancia l'esecuzione vera e propria della servlet, generando un thread separato per la gestione di ogni richiesta.
- Interfaccia: implementata dalla classe astratta javax.servlet.Servlet, con i metodi javax.servlet.GenericServlet e javax.servlet.http.HTTPServlet.
  - sviluppo delle servlet facilitato poiché basta implementare via override solo i metodi interessati
- **service()** (HTTPServlet) invoca i metodi per gestire le richieste web come doGet(), doPost().
- parametri (HTTPServlet) sono adattati al protocollo HTTP, e sono HTTPServletRequest e HTTPServletResponse.
- Recupero informazioni relativa a richieste/risposte HTTP
  - **String getParameter(String name)**: restituisce il valore dell'argomento indicato
  - **Enumeration getParameterNames()**:ritorna una lista di nomi di parametri
  - **Cookie[] getCookies():** restituisce i cookies del server sul client
  - **void addCookie(Cookie cookie)**: aggiunge un cookie nell'intestazione della risposta
  - **HTTPSession getSession(boolean create)** che crea o recupera una sessione HTTP per un determinato client.

JSP (Java Server Pages)(HTML con codice Java): tecnologia di programmazione Web in Java per lo sviluppo della *logica di presentazione* (tipicamente secondo il pattern MVC) di applicazioni Web, fornendo contenuti dinamici in formato HTML o XML.

- Le servlet possono essere integrate in maniera molto semplice nella costruzione delle pagine web tramite la tecnologia JSP.
- Questa tecnologia specifica l'interazione tra un server/container servlet ed un insieme di pagine che presentano informazioni all'utente.
- Esse sono costituite dai tag HTML tradizione e da particolari tag applicativi che controllano la generazione del contenuto della pagina, facilitando la separazione logica tra strato applicativo e strato di presentazione. (ASP = JSP microsoft con script VB anziché JAVA).
- Il codice JSP all'interno di una pagina viene incluso in tag speciali, delimitati da <% e %>, ad esempio

```
<div id="saluti">
        Sei su <I><% request.getParameter["title"] %></I>
</div>
```

- Le pagine JSP vengono compilate alla prima esecuzione, e mantenute in memoria per rendere più veloce una richiesta successiva della stessa pagina
- le richieste e le risposte HTTP vengono tradotte dal web server in oggetti conosciuti in Java (HTTPServletRequest e HTTPServletResponse)
- esecuzione di una JSP il client invia via HTTP la richiesta della JSP, che viene interpretato ed accede ai componenti lato server per la generazione dei contenuti dinamici. Il risultato viene poi spedito al client come pagina HTML

Tag tipici di una pagina jsp

```
<%-- COMMENTO -->
<%@ DIRETTIVA AL COMPILATORE %>
<tag attributes> azioni nel body in XML </tag>
```

- Possono essere le importazioni delle librerie, o attributi/valori passati alla pagina

```
<%@ page import="java.math.*, java.util.*" %>
```

- Possono includere in compilazione altre pagine JSP o HTML

```
<%@ include file="Copyright.html" %>
```

- Possono dichiarare tag definite dall'utente implentando classi

```
<%@ taglib uri="TableTagLibrary" prefix="table"%>
<table:loop>….</table:loop>
<jsp:azioni>corpo_azione</jsp:azione>
```

- Determinano l'invio della richiesta corrente ad un'altra pagina

```
<jsp:forward page="login.jsp"%>
```

```
        <jsp:param name="user" value="admin" />
    <jsp:param name="pwd" value="password"/>
</jsp:forward>
```

- Invia la richiesta ad una data URL includendone il risultato

```
<jsp:include page="login.jsp"%>
```

- Localizza ed istanzia un javaBean nel contesto specificato (pagina,richiesta,sessione,applicazione)

```
<jsp:useBean id="cart" scope="session" class="Carrello" />
```

- Elementi di scripting: istruzioni nel linguaggio specificato nelle direttive
- Tramite questi elementi è possibile accedere a degli elementi che possono essere condivisi in un determinato scope che sono ad esempio la **Declaration:** Variabili o metodi usati nella pagina

```
<% int[]  v = new int[10]; %>
```

  - **Expression:** Espressione nel linguaggio di scripting valutata e sostituita con il risultato

```
<p>La radice di 2 è <%math.sqrt(2.0) %></p>
```

  - **Scriptlet:** Frammenti di codice che controllano la generazione della pagina

```
<table>
<%for (int i=0;i<v.length;i+){  %>
<tr><td> <%= v[i] %></td></tr>
<% } %>
</table>
```

- **Oggetti impliciti:** elementi delle servlet che sono oggetti condivisibili in un determinato scope: request, la response, out, page, pageContext, session, application, config ed exception.
  - **Creazione degli oggetti:**
    - Implicitamente con le direttive JSP
    - Esplicitamente con le azioni
    - Direttamente con uno script (raro)
  - **Scope degli oggetti:** indicano il livello di visibilità dell'oggetto e quindi da chi può essere utilizzato
    - **Application(massima visibilità):** accessibile da pagine appartenenti alla stessa applicazione (oggetto globale)
    - **Session:** accessibile da pagine appartenenti alla stessa sessione in cui sono stati creati
    - **Request:** accessibile da pagine che hanno processato la stessa richiesta in cui è stato creato
    - **Page(minima visibilità):** accessibile solo dalla pagine in cui è stati creato

MVC (Modern View Controller) **all'architettura JSP/Servlet**

- il client invia la richiesta alla servlet
- la servlet richiama una pagina .jsp e li mette a disposizione della pagina jsp come Java Beans
- la servlet richiama una pagina .jsp che legge i dati dai beans ed organizza la presentazione della pagina che invia poi al client.
- Ottengo una **divisione dei compiti**, così ogni componente può essere costruito da gente specializzata (HTML al grafico, codice al programmatore...)

Java Bean: una classe scritta in un linguaggio ad oggetti che presenta un costruttore vuoto ed un insieme di campi leggibili e scrivibili solo attraverso degli appositi metodi get e set (detti *metodi accessori*) o delle proprietà.

WEB SERVICES

Definizione breve: è un sistema software progettato per supportare l'interoperabilità tra diversi elaboratori su di una medesima rete ovvero in un contesto distribuito

Definizione lunga: Si può definire un web service come "un software applicativo identificato da un URI le cui interfacce e binding sono in grado di essere definite, descritte e scoperte tramite artefatti XML e che supporta interazioni dirette con altri software usando messaggi basati su XML attraverso protocolli internet"

Perché i Web Services?

- Software come un servizio: web services distribuiti ovunque e rapidamente, incapsulabili, isolabili a determinati individui, comportando a un disaccoppiamento dei componenti e a sistemi più flessibili e stabili
- Interoperabilità dinamica di business: nuovi contratti più flessibili e gestibili grazie all'interoperatività dei sistemi
- Accessibilità:decentralizzazione dei servizi e distribuzione in internet, in modo da accederci con molti dispositivi

- Efficienza: ridistribuzione dei compiti, in modo da potersi concentrare sugli aspetti più critici, ignorando tutti i problemi della progettazione software
- Specifiche accordate universalmente
- Integrazione legacy: creare nuovi sistemi senza ricostruire tutto da capo
- Nuove opportunità sul mercato: per gli sviluppo dinamici

Caratteristiche chiave: I servizi sono quindi definibili:

- come componenti indipendenti
- con un'interfaccia ben definita (attraverso linguaggi standard di descrizione come il WSDL)
- un punto di accesso univoco (tramite l'utilizzo di URL che permettono possibili sviluppi di directory di naming chiamate UDDI Directory)
- che effettuano uno scambio dati basato sui documenti (attraverso la rappresentazione standard XML)

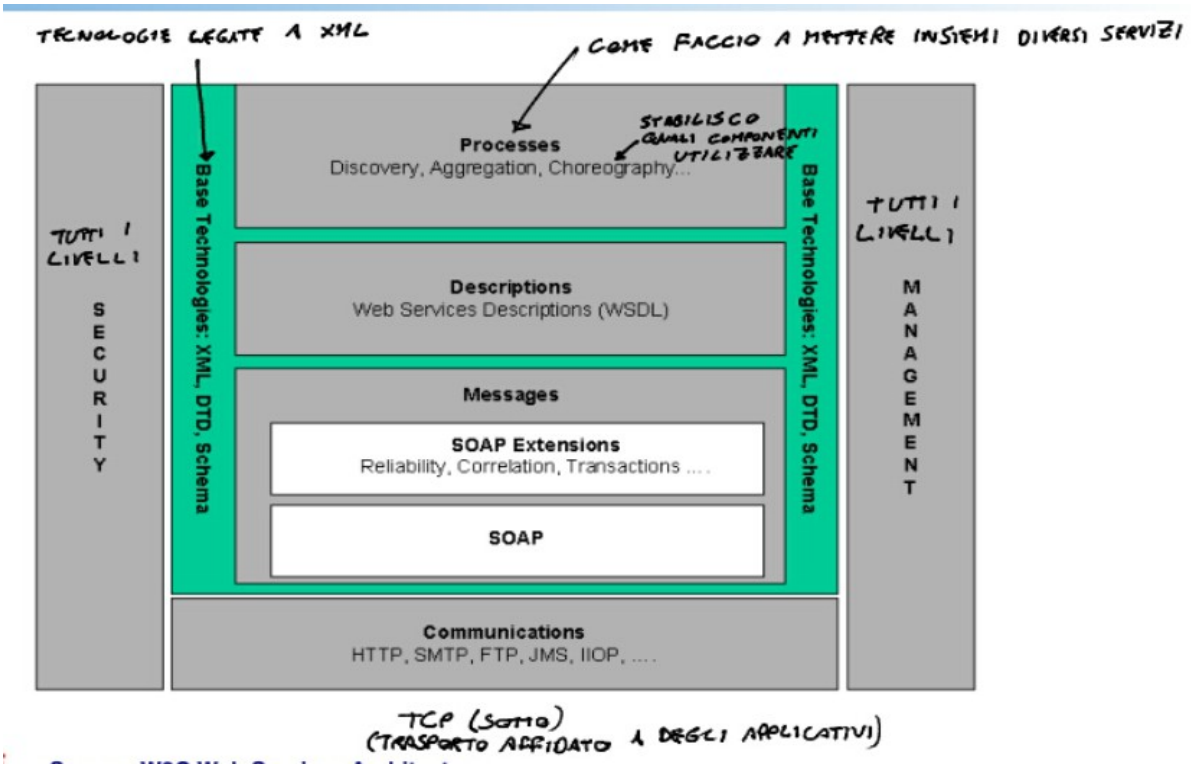
Come è organizzato un sistema

- ogni operatore fornisce servizi di rete
- i servizi devono essere
  - descritti per essere individuabili
  - invocabili in modo noto
  - essere componibili
  - devono soddisfare le esigenze del cliente e del dominio in cui sono sviluppati

SOA (Service Oriented Architecture) modello per strutturare i web service che si basa su 3 principi fondamentali (Minimizzazione della dipendenza fra i servizi, Presenza di contratti per la comunicazione, Riutilizzabilità), e che li suddivide in base a tre principali criteri:

- Componenti
  - Servizio
  - Descrizione del servizio
- Ruoli (attori)
  - Fornitore del servizio
  - Agenzie di discovery
  - Richiedenti del servizio
- Operazioni
  - Pubblicare
  - Trovare
  - Interagire

Stack d'architettura Web Service



Stack concettuale:un'altra architettura possibile è quella che suddivide le funzionalità dei web service in base allo stack dei protocolli utilizzati (a partire dal livello più basso):

- **Network**: I web services devono essere accessibili su rete internet per essere invocati da un richiedente. Essi utilizzano comunemente lo standard http per la comunicazione
- **XML-Based Messaging** : Rappresenta l'uso di XML come protocollo standard per lo scambio dei messaggi. In particolare, i web services usano SOAP per la gestione dell'XML e per l'aggiunta di informazioni. I messaggi vengono scambiati sullo strato network tramite richieste HTTP POST.
- **Service Description**: Strato responsabile della descrizione dell'interfaccia pubblica di un web service. WSDL è lo standard per questo strato, basato su XML per la descrizione del servizio.
- **Service pubblication & discovery**: Strato responsabile delle funzionalità di pubblicazione e ricerca di un servizio. Attualmente lo standard è UDDI basato anch'esso su XML
- **Service Flow** : Definisce come vengono eseguite le comunicazioni tra servizi e i flussi delle informazioni.
  A supporto del linguaggio XML, entrano in gioco anche il DTD (Document Type Definition) che permette di specificare le caratteristiche strutturali di un documento XML, come le relazioni gerarchiche tra gli elementi, l'ordine di apparizione degli stessi nel documento e la loro obbligatorietà, e il linguaggio XSD, che serve a definire la struttura del documento XML come gli attribuiti che possono comparire in un documento , eventuali elementi "figli" di altri e la loro molteplicità.

UDDI (Universal Description Discovery and Integration): un directory service in cui le aziende possono registrare i propri web service pubblici e trovarne altri.

- Un registro UDDI contiene informazioni riguardanti un insieme di web services, tra i quali anche la loro descrizione WSDL e i dati contenuti accessibili via SOAP.
- Sebbene UDDI sia stato proposto come standard, esso è poco utilizzato (microsoft e IBM hanno spento i loro UDDI pubblici nel 2006).
- Servizi forniti da UDDI:
  - Pagine Bianche (informazioni di base sui contatti dell'azienda)
  - Pagine Gialle (contengono informazioni sui servizi web raggruppati per categorie)
  - Pagine Verdi (contengono informazioni tecniche sulle funzioni supportate dai web services esposti)
- Cosa definiscono le specifiche UDDI:
  - Le interfacce Web del servizio
    - Publish API
    - Inquiry API
  - Strutture dati utilizzate
    - Business entity : descrizione di un'azienda che fornisce il servizio
    - Business service: descrive una collezione di WS collegati offerti da una business entity.
    - Binding template: descrive informazioni tecniche necessarie all'uso del WS (1 o +)
    - tModel: tipo di dato che permette la definizione di etichette per la classificazione delle informazioni.

Peculiarità dei Web Services

- **Componenti pubblici**
- **Componibilità**
- **Descrizioni semantiche**
- **QoS (Quality of service)**
- **Organizzazione del sistema**

Semantic Web: trasformazione del World Wide Web in un ambiente dove i documenti pubblicati (pagine HTML, file, immagini, e così via) sono associati ad informazioni e dati (metadati) che ne specificano il contesto semantico in un formato adatto all'interrogazione e l'interpretazione (es. tramite motori di ricerca) e, più in generale, all'elaborazione automatica

SOAP(Simple Object Access Protocol): protocollo basato sull'XML che permette lo scambio di messaggi tramite protocolli internet standard (HTTP/SMTP).

- stateless (senza stato) e "one-way" (tipicamente request-response, può attraversare intermediari, no multicast)
- ignora la semantica dei messaggi scambiati attraverso di esso (essi possono essere RPC o Documenti, ma è l'endpoint che definisce la semantica, non SOAP)
- l'interazione tra due siti deve essere codificata nel documento SOAP

- ogni pattern di comunicazione (esempio: request-response) dev'essere implementato sui livelli più alti dei sistemi
- basato su XML e standard per creare messaggi XML
- non è legato ad una particolare piattaforma e/o linguaggio di programmazione
- semplice ed estensibile
- usato per accesso ai registri e invocazione dei servizi (nel contesto dei Web Services)

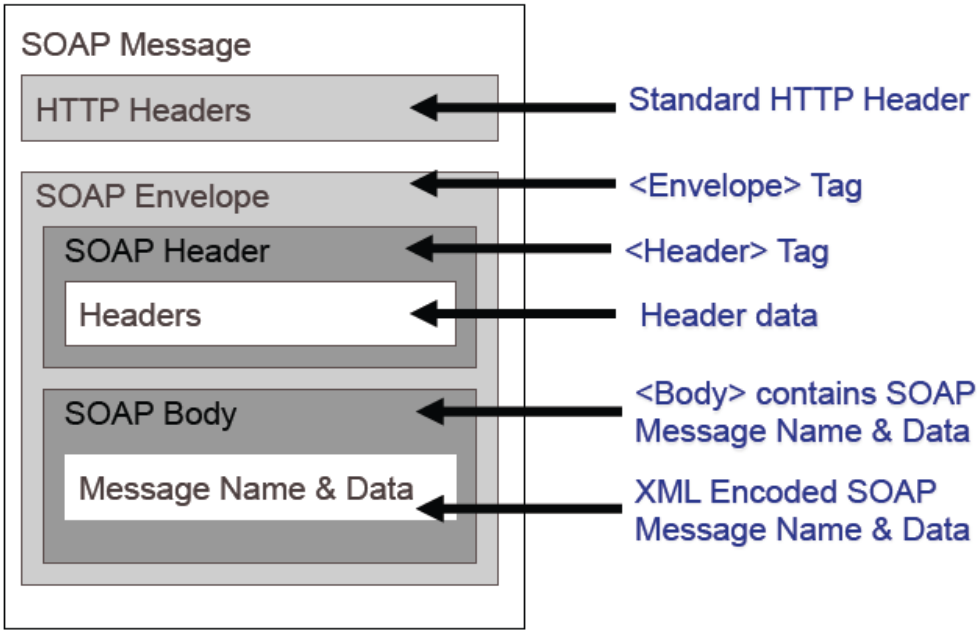
namespace XML: usati da SOAP per definire la struttura dati

- http://schemas.xmlsoap.org/soap /envelope/ che definisce la struttura del tag <envelope>
- http://schemas.xmlsoap.org/soap/encoding/ che definisce la serializzabilità dei messaggi

Struttura di un messaggio SOAP (in tag)

- **<Envelope>** :definisce i contenuti del messaggio, chi se ne dovrebbe occupare e se il messaggio sia obbligatorio o meno
- **<header>:** opzionale può contenere informazioni sul routing dei messaggi, sulle autenticazioni e sul contesto delle transazioni
- **<body>** :contiene il messaggio e informazioni su richiesta e risposta.

Uso SOAP e http come livello di trasporto



Types are either a simple (scalar) type or a compound type constructed as a composite of several parts, each of a simple type.
Simple types Int, float, negativeInteger, string

· · ♦

Compound types structs and arrays

· · ♦ A compound value represents a combination of two or more accessors grouped as children of a single accessor

· · ♦ A struct is a compound value with different accessor names

· · ♦ An array is a compound value with accessors with the same name.

Tipi di dati in SOAP

- **semplice** (scalari)(int,float,negativeInteger, string) **<accessor>value</accessor>**
  - *Accessor:* contiene o permette l'accesso a un valore
  - *Value:* una singola unità di dati o una combinazione di dati
  -
- **complessi** (compositi) (struct e array)
  - costruiti da più parti, ognuna che può essere un tipo complesso o semplice, oppure un array di valori con lo stesso nome
  - **<person>**

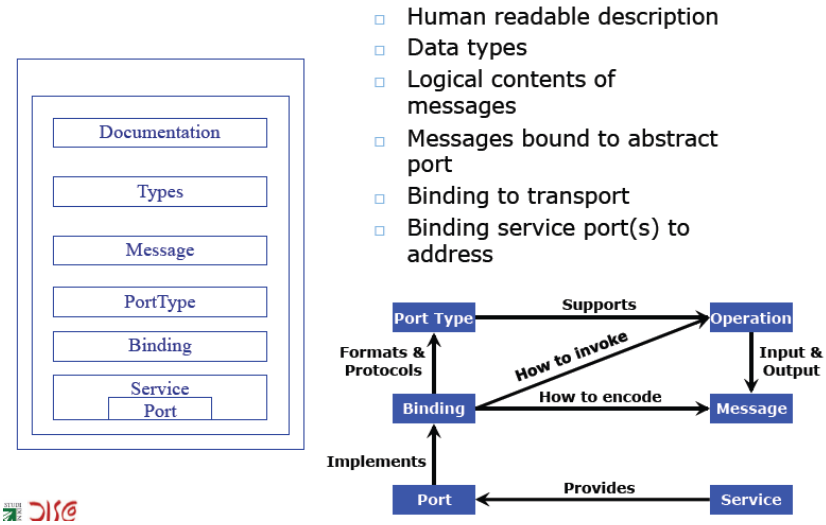
<firstname>Joe</firstname>

<lastname>Smith</lastname>
</person>
L'implementazione del SOAP nei linguaggi di programmazione avviene attraverso delle librerie chiamate SOAP CONTAINER .

WSDL (Web Service Definition Language):linguaggio basato sull'XML per la descrizione delle unzionalità di un web server (interfacce dei WS e del modo per accedervi)
WSDL descrive 4 pezzi critici di dati:

- Informazioni sulle interfacce ad accesso pubblico
- Dichiarazione dei tipi di dato scambiati nelle richieste e risposte
- Informazioni riguardanti il protocollo di trasporto
- Indirizzo per contattare il servizio

Struttura tipica:



- Struttura di definizione

```
<definition>
        <types> definizione dei tipi</types>
        <message> definizione di un messaggio </message>
        <portType> operazione fatta dal WS</portType>
        <binding>Protocollo di comunicazione del WS</binding>
</definition>
```

- Esempio di frammento

```
<message name="getTermRequest">
        <part name="term" type="xs:string"/>
</message>
<message name="getTermResponse">
        <part name="value" type="xs:string"/>
</message>
<portType name="glossaryTerms">
        <operation name="getTerm">
        <input message="getTermRequest"/>
        <output message="getTermResponse"/>
        </operation>
</portType>
```

Elementi principali di WDSL

- **<portType>**: (il più importante)definisce un Web Service, le operazioni che possono essere fatte e i messaggi coinvolti (l'elemento è comparabile a una libreria di funzioni di un linguaggio tradizionale)

- **<message>:** messaggi usati dal Web Service (come arriva il messaggio e come viene codificato), che definiscono gli elementi di dati di ogni operazione; ogni messaggio può consistere di uno o più parti (equiparabili ai parametri di una funzione tradizionale)
- **<types>:** tipi di dati usati dal Web Service (definiti con XML Schema per neutralità)
- **<binding>:** protocolli di comunicazione usati dal Web Service

Esempio binding

```
<binding name=”glossaryBinding type=”glossaryTerms”>
        <soap:binding style=”document” transport=”http://schemas.xmlsoap.org/soap/http”/>
        <operation>
                <soap:operation soapAction="http://example.com/getTerm"/>
                <input>
                        <soap:body use=”literal”/>
                </input>
                <output>
                        < soap:body use=”literal”/>
                </output>
        </operation>
</binding>
```

Ulteriori tag chiamati <service> che corrispondono alla vera descrizione del WS, comprendendo tutti i tag <port> di tipo portType

```
<service name=”Glossario”>
        <documentation>WSDL file per il glossario</documentation>
        <port binding=”glossaryBinding” name=”glossaryPort”>
                <soap:address location=”http://localhost:8080/soap/servlet/rpcrouter”/>
        </port>
</service>
</definitions>
```

Tipi di operazioni Web Service:

- **One-Way**: L’operazione riceve un messaggio ma non risponde (asincrono)
- **Request-Response**: L’operazione riceve una richiesta ed invia una risposta (sincrono)
- **Solicit-Response:** L’operazione manda una richiesta ed attende una risposta
- **Notification:** L’operazione invia un messaggio ma non attende una risposta.

Processi di business(livello di astrazione più alto): una serie di attività logicamente correlate fra loro, automatiche o manuali, di lunghezza variabile, effettuate per ottenere un risultato di business ben definito

Workflow: La sequenza di passi in cui oggetti e/o informazioni vengono passati da uno step produttivo ad un altro.

Operazioni per comporre i web services (per definire il workflow)

- **Orchestrazione:** descrive come i WS possono interagire fra loro a livello di messaggi, includendo la logica di business e l’ordine di esecuzione delle operazioni
- **Coreografia**: descrive la sequenza di messaggi che comprendono più parti e sorgenti compresi nel macroprocesso.

BPEL(Business Process Execution Language): linguaggio basato su XML usato per specificare formalmente i processi business e i protocolli d’interazione di business (come i due aspetti sopra dell’interoperabilità dei Web Services)

- BPEL fornisce molte funzionalità per facilitare la modellazione e l’esecuzione di processi business basati su Web Service
- BPEL usa dei WSDL per specificare le attività che prendono parte ad un processo basato su WS, estendendo WSDL in 3 modi

- Ogni processo BPEL è esposto come un WS usando WSDL, descrivendo i punti d'ingresso e di uscita pubblici di ogni processo
- I tipi di dato di WSDL sono usati all'interno di un BPEL per descrivere le informazioni passate tra le richieste.
- WSDL può essere usato per referenziare servizi esterni richiesti da un processo di business.

Struttura di un BPEL:

- **Message Flow:** Attività basilari come <invoke> <receive> e <reply> (chiamano, ricevono e rispondono ad altri Web Services)
- **Control Flow:** Definiscono stati per scopi di sincronizzazione
- **Data Flow:** comprendono variabili che forniscono il mezzo per contenere messaggi che formano lo stato di un processo di business
- **Process Orchestration:** utilizzano link di partner per stabilire relazioni p2p
- **Fault & Exception Handling:** si occupano di eventuali errori durante le invocazioni dei WS

RESTful Web Service (REpresentational State Transfer): un insieme di principi di architetture di rete, i quali delineano come le risorse sono definite e indirizzate.

- Il termine è spesso usato nel senso di descrivere ogni semplice interfaccia che trasmette dati su HTTP senza un livello opzionale come SOAP o la gestione della sessione tramite i cookie.

Principi di REST:

- Le risorse devono essere identificabili tramite URI (1 Risorsa = 1 URI)
- Le risorse sono accedute attraverso le loro rappresentazioni (MIME types)
  - Risorse desiderate: inserite nell'header della richiesta con il campo accept
  - Risorse restituite: inserite nell'header della risposta con il campo Content-Type
- Ci possono essere più rappresentazioni di una risorsa
- **Lo stato delle applicazioni è guidato dalla manipolazione di risorse**
- Le interfacce devono essere uniformi per tutte le risorse
  - 4 possibili azioni per ogni risorsa : PUT – GET – POST – DELETE
  - Re-implementazione delle applicazioni CRUD
    - Create=POST
    - Read=GET
    - Update=PUT
    - Delete=DELETE
- **I messaggi sono autodescrittivi (tramite i metadata) e stateless**
- Le risorse contengono link=hypermedia (raccolta di informazioni eterogenee, quali grafica, audio, video e testo, connessi tra loro in maniera non sequenziale)(derivato di hypertext)
  - Attraverso gli HyperLink, il server fornisce una serie di link al client, che "sposta" l'applicazione da uno stato all'altro seguendo i link.

SOAP vs. REST

- **Protocollo di trasporto**
  - REST: HTTP
  - SOAP: diversi protocolli (HTTP, TCP, SMTP)
- **Formato messaggi**
  - REST: diversi formati (e.g., XML-SOAP, RSS, JSON)
  - SOAP: XML-SOAP (Envelope, Header, Body)
- **Identificatore dei servizi**
  - REST: URI
  - SOAP: URI e WS-addressing
- **Descrizione servizi**
  - REST: Documentazione Testuale o Web Application Description Language (WADL)
  - SOAP: Web Service Description Language (WSDL)
- **Composizione servizi**
  - REST: Mashup
  - SOAP: BPEL
- **Scoperta dei servizi**
  - REST: no supporto standard
  - SOAP: UDDI

- **Utilizzo di http**
  - REST: linguaggio contenitore
  - SOAP: usato direttamente come formato dei dati (secondo diversi modelli)

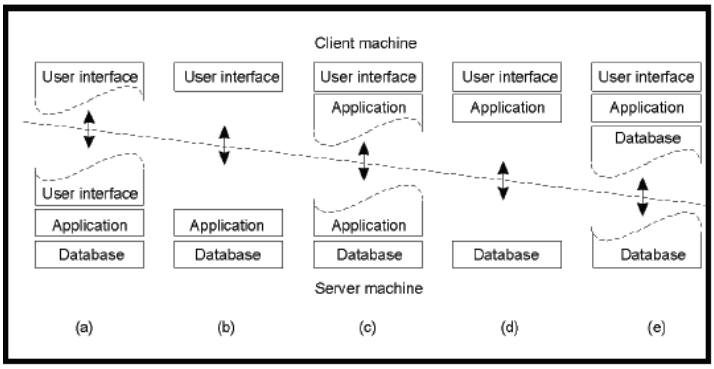
Vantaggi di REST:

- REST è semplice
  - Caching (miglioramento tempi di risposta e riduzione carico di lavoro del server)
  - Meno stati di comunicazione (miglior bilanciamento di carico)
  - Meno software specializzato (semplici tecnologie utilizzate)
  - Identificazione con meccanismi standard
- REST è standard
  - Enfatizzazione sull'utilizzo corretto e completo del protocollo HTTP
  - Meccanismi leggeri e a strati (su http)
  - Piattaforma d'applicazione distribuita e controllata con l'hypermedia

---

AJAX E JSON

AJAX(Asynchronous JavaScript and XML) ::è un gruppo di tecniche di sviluppo Web interconesse usate sul lato client per creare applicazioni Web asincrone. Lo sviluppo di applicazioni HTML con AJAX si basa su uno scambio di dati in background fra web browser e server, che consente l'aggiornamento dinamico di unapagina web senza esplicito ricaricamento da parte dell'utente.

- Ajax permette di programmare il client per decidere il carico di lavoro gestito da lui o dal server
- 

- E' più un nuovo pattern piuttosto che una nuova tecnologia

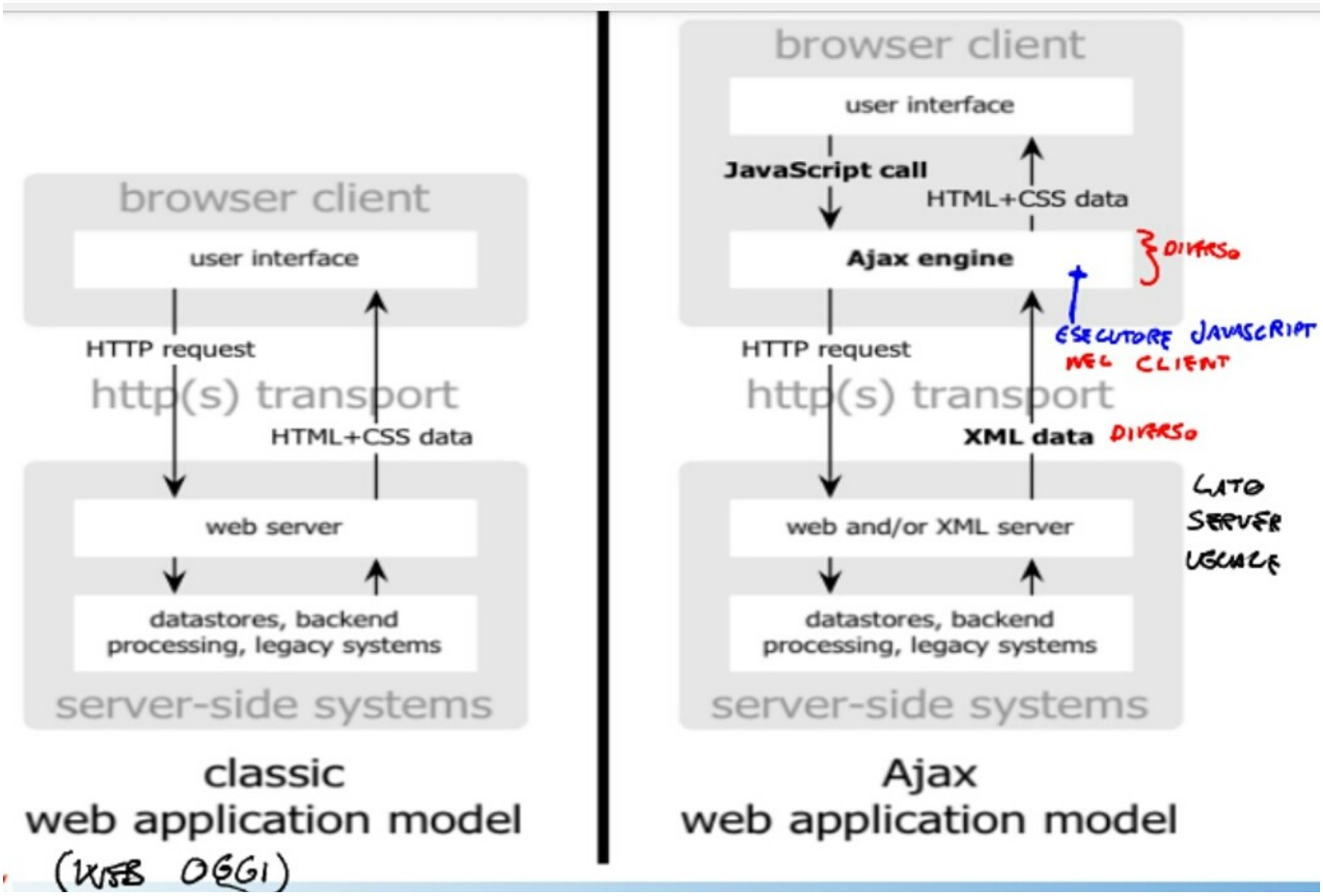
Componenti di AJAX:

- **HTML (o XHTML) e CSS**: presenta l'informazione
- **DOM**: presentazione dinamica e interazione
- **XML e XSLT**: scambio e manipolazione di dati
- **Oggetti XMLHttpRequest**: Recupero di dati IN MODO ASINCRONO dal web server
- **Javascript:** tiene tutto insieme

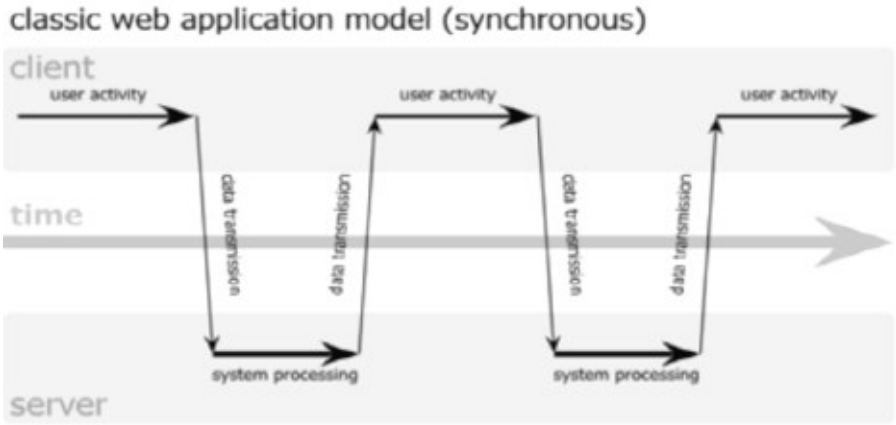
Utilizzo dei paradigmi AJAX

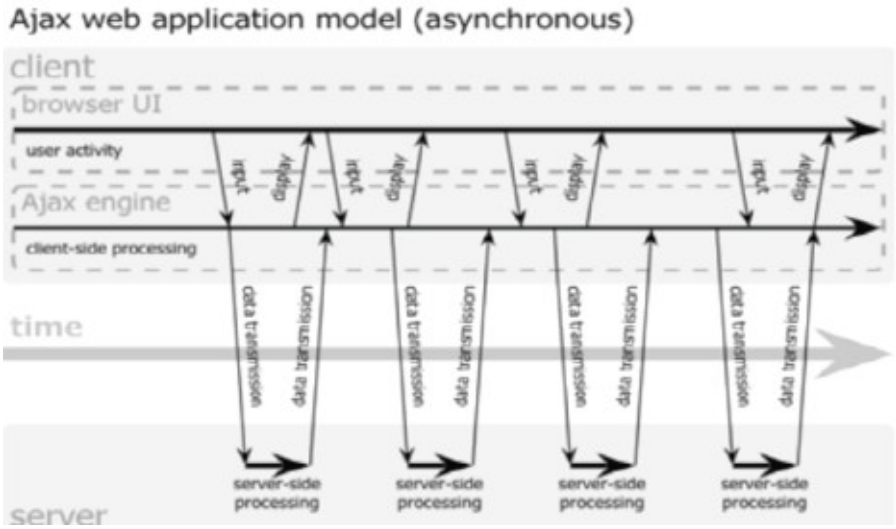
- **Validazione dati in tempo reale:** contenuti di form che dovrebbero essere validati dal server sono invece validati dal form prima di essere inviati
- **Auto completamento**(dei dati inseriti)
- **Operazioni con migliori dettagli**(senza dover riaggiornare la pagina)
- **Controlli UI sofisticati** (controlli a albero, menu e barre del progresso)(senza dover riaggiornare la pagina)

Struttura normali applicazioni web VS Ajax



Processazione delle richieste normali VS Ajax(nei browser, il comportamento è asincrono rispetto alla rete)





Problemi di Ajax

- Il tasto indietro del browser perde di significato
- Cambiamento imprevisto di parti della pagina

- Il segnalibro di un particolare stato della pagina diventa difficile
- Aumento della dimensione del codice su browser (e quindi aumento dei tempi di risposta)
- Difficoltà nel debug
- Sorgente sempre visibile (quindi aperta agli hacker)
- Più carico del server (le richieste asincrone sono un'operazione pesante)

JSON(JavaScript Object Notation): formato di standard aperto che usa testo leggibile naturalmente per trasmettere oggetti di dati consistenti in coppie valore-attributo. E' usato primariamente per trasmettere data fra un server e un'applicazone web, come un'alternativa a XML.
Tipi base: Number, String, Boolean, Array, Object, Null
Proprietà di JSON:

- Formato immediatamente leggibile sia da macchina che da umani
- Supporto dell'Unicode, quindi quasi ogni carattere esistente
- Formato che si auto-documenta
- Sintassi ristretta e requisiti di parsing che permettono ai necessari algoritmi di parsing di rimanere semplici, efficienti e consistenti
- Abilità di rappresentare record, liste e alberi (le strutture dati più generali)

Implementazione JSON in AJAX:

```
!<html>
!...
!<script> var data = JSONdata; </script>
!...
</html> !
```

È usato con XMLHttpRequest e può essere convertito in una struttura JavaScript

RSS(RDF Site Summary oppure anche Really Simple Syndication): famiglia di formati standard di web feed per pubblicare informazione frequentemente aggiornata: aggiornamenti di blog, notizie, audio, video. Un documento RSS (chiamato "feed", "web feed", o "canale") include testo completo o sintetico, e metadata, come la data di pubblicazione e il nome dell'autore

---

JAVASCRIPT & JQUERY

JAVASCRIPT: linguaggio di scripting orientato agli oggetti e agli eventi, comunemente utilizzato nella programmazione Web lato client per la creazione, in siti web e applicazioni web, di effetti dinamici interattivi tramite funzioni di script invocate da *eventi* innescati a loro volta in vari modi dall'utente sulla pagina web in uso (mouse, tastiera ecc...).
Linguaggio di scripting: linguaggio di programmazione per l'automazione di compiti altrimenti eseguibili da un utente umano all'interno di un ambiente software

- Javascript è dinamico, debolmente tipizzato, con sintassi ispirata a C e Java
- Usando javascript, le applicazioni web sono create componendo diversi pezzi di codice creati da terze parti, piuttosto che programmando tutto
- È un linguaggio principalmente su lato client (nello strato di presentazione nell'architettura multitier)
- Tutte le variabili non hanno tipo fino a quando non vengono dichiarate
- Gli operatori sono ancora i soliti se non per === e !== che, contrariamente a == e !=, il controllo viene fatto ma senza effettuare una conversione
- Un'array non necessita di essere inizializzato, è essenzialmente dinamico e i suoi elementi possono essere di tipi diversi
- Strutture di controllo di flusso (if, while, do while, for, switch...) identiche a Java

JQUERY: libreria di funzioni Javascript (framework) open source per le applicazioni web, che si propone come obiettivo quello di semplificare la manipolazione, la gestione degli eventi e l'animazione delle pagine HTML.

- Sintassi specificatamente orientata a permettere una rapida e semplice selezione di elementi del documento HTML
- Sintassi comando: $(selector).action()
  - $ definisce l'accesso a funzionalità offerte da jQuery
  - selector viene utilizzato per specificare una sorta di query per selezionare una parte del documento HTML
  - action specifica un'azione da effettuare sull'elemento stesso

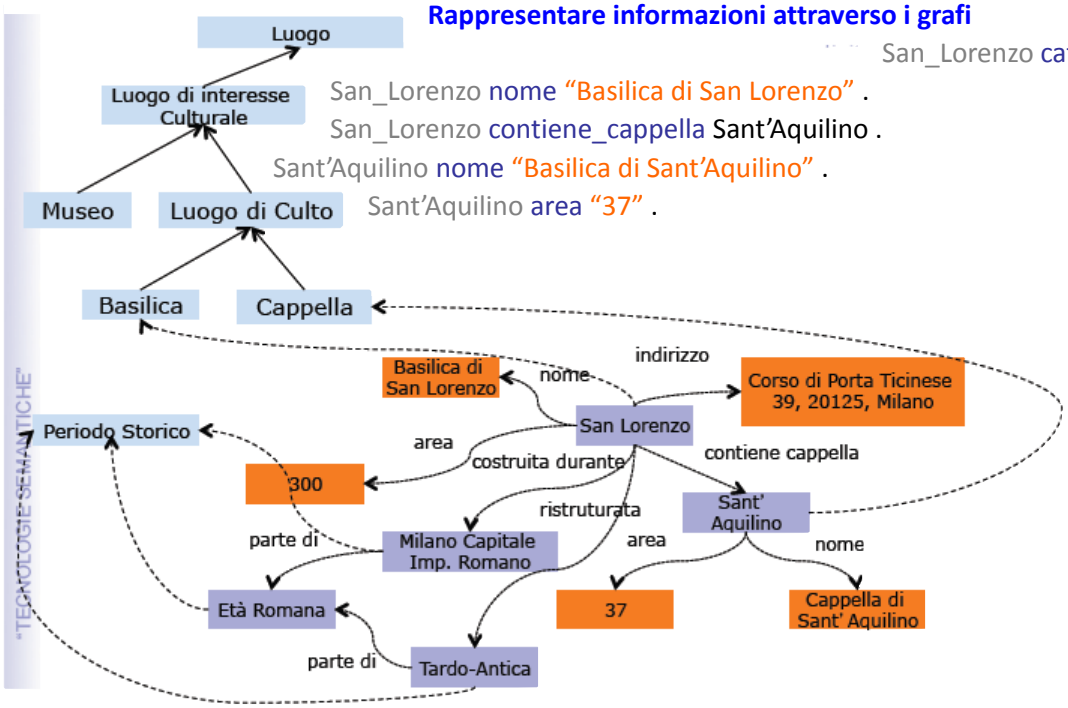
- struttura a base di span.pq (che è anche il selettore di tutti gli span)

RDF – RESOURCE DESCRIPTION FRAMEWORK

RDF è un modello per la rappresentazione dei dati su web, basato su: triple, grafi orientati, URI

- **Triple:** unità (atomo) base di organizzazione delle informazioni
  - può essere semplificata come una relazione <soggetto>-<predicato>-<oggetto>
  - Es. CASA-INDIRIZZO-VIA RICORDI 21 o SAN_LORENZO nome “Basilica di san lorenzo”
- **Grafi orientati:** insieme di triple
  - è una coppia D = (V, A),
  - **V:**insieme di vertici
  - **A:**insieme di archi orientati di D.
    - "arco orientato":arco caratterizzato da una direzione

Rappresentare informazioni attraverso i grafi



San_Lorenzo category Basilica.
San_Lorenzo nome “Basilica di San Lorenzo” .
San_Lorenzo contiene_cappella Sant’Aquilino .
Sant’Aquilino nome “Basilica di Sant’Aquilino” .
Sant’Aquilino area “37” .

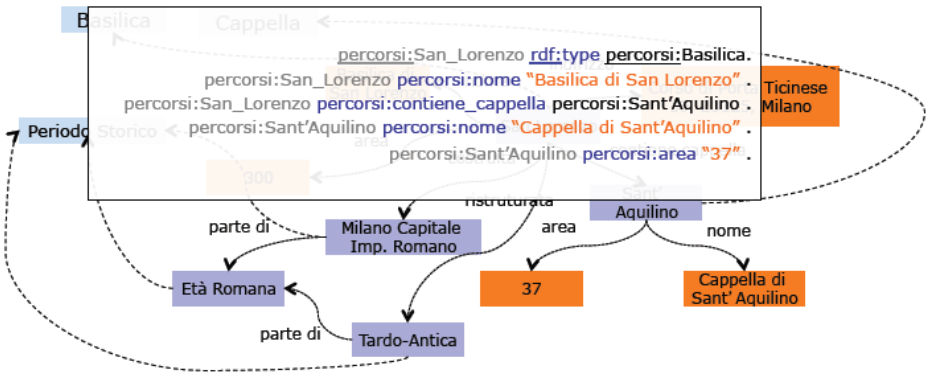
- **URI**(Uniform Resource Identifier):identificazione univoca obbligatoria delle risorse(nomi,predicati, relazioni...)
- **Ogni cosa può essere una risorsa**
- **Sintassi URI:** `<scheme>:<scheme-specific-part>`
- Composto dai sottoinsiemi URL e URN
  - **URL** (Uniform Resource Locator): identifica le risorsa fornendo informazioni sulla locazione di essa in rete
  - **URN** (Uniform Resource Name): identifica le risorse tramite nomi univoci

Rappresentare le informazioni attraverso grafi sul Web

Indirizzo pagina Web in cui descrivo la mia risorsa
http://www.lintar.disco.unimib.it/mediawiki/index.php/Basilica ← URI

Radice comune a tutte le risorse
http://www.lintar.disco.unimib.it/mediawiki/index.php/

Abbreviazione
http://www.lintar.disco.unimib.it/mediawiki/index.php/ => percorsi: ← NAMESPACE



Caratteristiche principali

- **Indipendenza:** Dato che i predicati sono risorse, ogni organizzazione può inventarli
- **Interscambio:** siccome RDF può essere convertito in XML, RDF è serializzabile e condivisibile

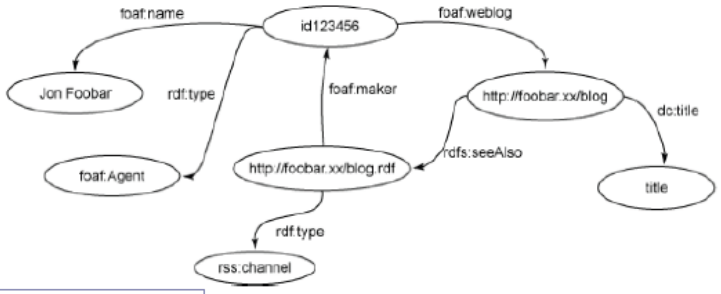
- **Scalabilità:** Le proprietà RDF sono semplici (organizzate in triple), quindi sono facili da gestire e osservare

Es: www.sic-online.it rdf:maker "Sic s.r.l."

La tripla, tradotta in RDF sarebbe:

```
<rdf: RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:au="http://description.org/schema/">
        <rdf:description rdf:about=www.sic-online.it>
                <rdf:author>Sic s.r.l.</rdf:author>
        </rdf:description>
</rdf:RDF>
```

RDF triples and RDF graph structure



<SUBJECT> <PREDICATE> <OBJECT>

Id123456 foaf:name "Jon Foobar" .
Id123456 rdf:type foaf:agent .
http://foobar.xx/blog.rdf foaf:maker Id123456 .
http://foobar.xx/blog.rdf rdf:type rss:channel .
http://foobar.xx/blog dc:title "title" .

PS.
Blank Nodes...

Id123456 foaf:friend _:b .
_:b rdf:type foaf:agent .

Notazione Turtle (Terse RDF Triple Language)**/N3 per rappresentare RDF:**

- Turtle e N3 non usano XML e sono progettati per un codice più human-readable e compatto di XML che è più machine-readable)
- usa le stesse triple di XML (human&machine-readable)

Example — XML Namespaces

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description rdf:about="http://www.archeoserver.it">
    <dc:title>Sharing archaeological knowledge on the web</dc:title>
  </rdf:Description>
</rdf:RDF>
```

may be written in **Turtle** notation like this:

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .

<http://www.archeoserver.it> dc:title "Sharing archaeological knowledge on the web" .
```

- gli URI identificano individui, proprietà, classi e datatype
- i soggetti devono essere delle URI, eccetto i nodi vuoti
  - **nodo vuoto =** _:name
- **letterali:** stringhe, numeri e dae
  - **piani:** "2006" "Età del bronzo"
  - **tipizzati:** "2006"^^xsd:integer "Età del bronzo"^^xsd:string
- **abbreviazioni**
  - le triple con lo stesso soggetto possono essere abbreviate con ";"

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .

<http://www.archeoserver.it/> dc:title "Sharing archaeological knowledge on the web" ;
                              dc:publisher "UniBo" ;
```

  - le triple con lo stesso soggetto e predicaco possono essere abbreviate con ","

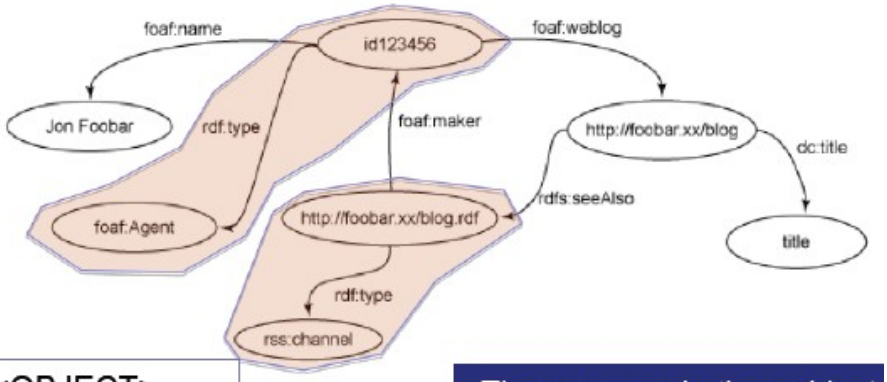
```
@prefix dc: <http://purl.org/dc/elements/1.1/> .

<http://www.archeoserver.it/> dc:title "Sharing archaeological knowledge on the web"@en ,
                                      "Condividi la conoscenza archeologia sul web"@it ;
```

Language selector

-
- **RDF è stato progettato fin dall'inizio per gestire risorse multiple**

rdf:type <SUBJECT> rdf:type <OBJECT>

- **Categorizzazione triple**

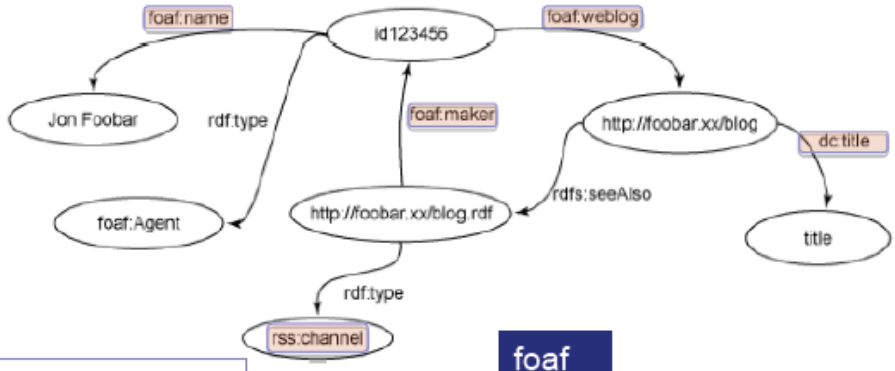


<SUBJECT> rdf:type <OBJECT>

The resource in the *subject* belongs to the **category** specified by the *object*

Id123456 foaf:name "Jon Foobar" .
Id123456 rdf:type foaf:agent .
http://foobar.xx/blog.rdf foaf:maker Id123456 .
http://foobar.xx/blog.rdf rdf:type rss:channel .
http://foobar.xx/blog dc:title "title" .

-
- **Vocabolari condivisi**



<SUBJECT> <PREDICATE>
<OBJECT>
Id123456 foaf:name "Jon Foobar" .
Id123456 rdf:type foaf:agent .
http://foobar.xx/blog.rdf foaf:maker Id123456 .
http://foobar.xx/blog.rdf rdf:type rss:channel .
http://foobar.xx/blog dc:title "title" .

foaf
…
dc …
dc…

Vocabularies:
- Properties
- Categories
…with shared meaning

-

FOAF (Friend Of A Friend): rappresentazione formale in pagine che descrivono persone, I collegamenti tra di loro e le cose che possono creare e fare

- Esempio di FOAF su Dan Brickley:

<foaf:Person rdf:about="#danbri" xmlns:foaf="http://xmlns.com/foaf/0.1/">

<foaf:name>Dan Brickley</foaf:name>
<foaf:homepage rdf:resource="http://danbri.org/" />
<foaf:openid rdf:resource="http://danbri.org/" />
<foaf:img rdf:resource="/images/me.jpg" />
</foaf:Person>

- Altro esempio di FOAF

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

<#JW>
    a foaf:Person ;
    foaf:name "Jimmy Wales" ;
    foaf:mbox <mailto:jwales@bomis.com> ;
    foaf:homepage <http://www.jimmywales.com/> ;
    foaf:nick "Jimbo" ;
    foaf:depiction <http://www.jimmywales.com/aus_img_small.jpg> ;
    foaf:interest <http://www.wikimedia.org> ;
    foaf:knows [
        a foaf:Person ;
        foaf:name "Angela Beesley"
        ] .

<http://www.wikimedia.org> rdfs:label "Wikimedia" .
```

Since rdf:type is such a fundamental relationship, Turtle has a special keyword **a**, to replace the type relationship

DC (Dublin Core): è un sistema di metadati di elementi essenziali e standard usati per descrivere le informazioni di qualunque risorsa sulla rete, a prescindere della sua piattaforma (cross-platform)

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
urn:pp:20070426 dc:creator "Andrea Bonomi" .
urn:pp:20070426 dc:subject "RDF" .
urn:pp:20070426 dc:description "An introduction to RDF" .
urn:pp:20070426 dc:date "26 Apr 2007" .
urn:pp:20070426 dc:dateCopyrighted "13 Apr 2007" .
urn:pp:20070426 cc:License cc:by-sa-1.0 .
```

example of CC metadata for this presentation

Dublin Core Metadata Initiative®

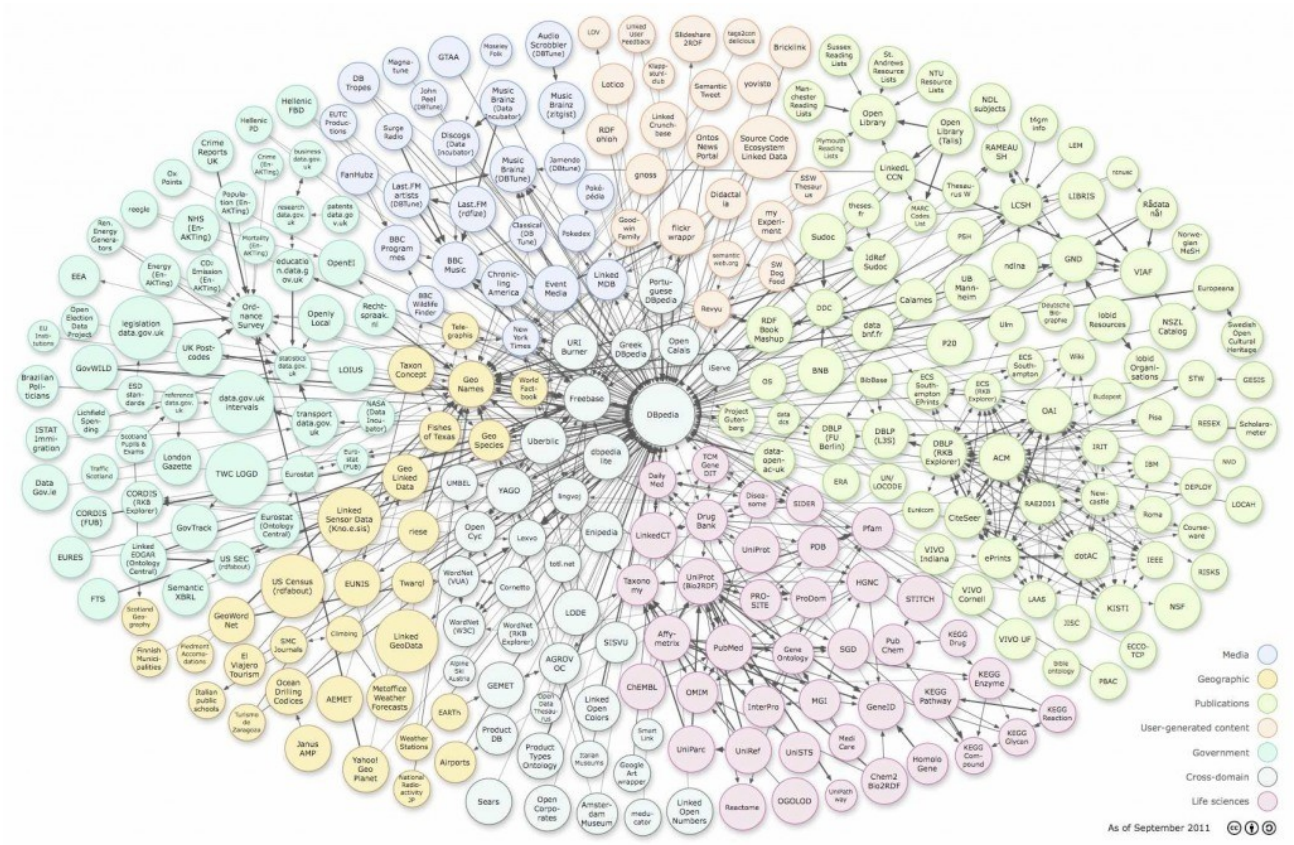
http://dublincore.org/

The most common metadata elements used by Dublin Core are:

- title (the name given the resource)
- creator (the person or organization responsible for the content)
- subject (the topic covered)
- description (a textual outline of the content)
- publisher (those responsible for making the resource available)
- contributor (those who added to the content)
- date (when the resource was made available)
- type (a category for the content)
- format (how the resource is presented)
- identifier (numerical identifier for the content such as a URL)
- source (where the content originally derived from)
- language (in what language the content is written)
- relation (how the content relates to other resources for instance, if it is a chapter in a book)
- coverage (where the resource is physically located)
- rights (a link to a copyright notice)

Linked Data: modalità di pubblicazione di dati strutturati e collegati tra loro basato su tecnologie e standard web aperti come HTTP e URI e ne estende l'applicazione che possano essere lette e comprese dai computer(Semantic Web)

- Pubblico dati strutturati sul Web
- Imposto collegamenti dati tra più sorgenti di dati qualsiasi
- **Principi**
  - Usare URI per identificare oggetti
  - Usare HTTP URI in modo che questi oggetti possano essere referenziati e cercati da persone
  - Fornire informazioni utili sull'oggetto quando la sua URI è deferenziata, usando formati standard come RDF
  - Includere link ad altre URI relative ai dati esposti per migliorare la ricerca di altre informazioni relative nel Web
- Esempio rilevante: **LOD** (Linked Open Data)

- 

SPARQL

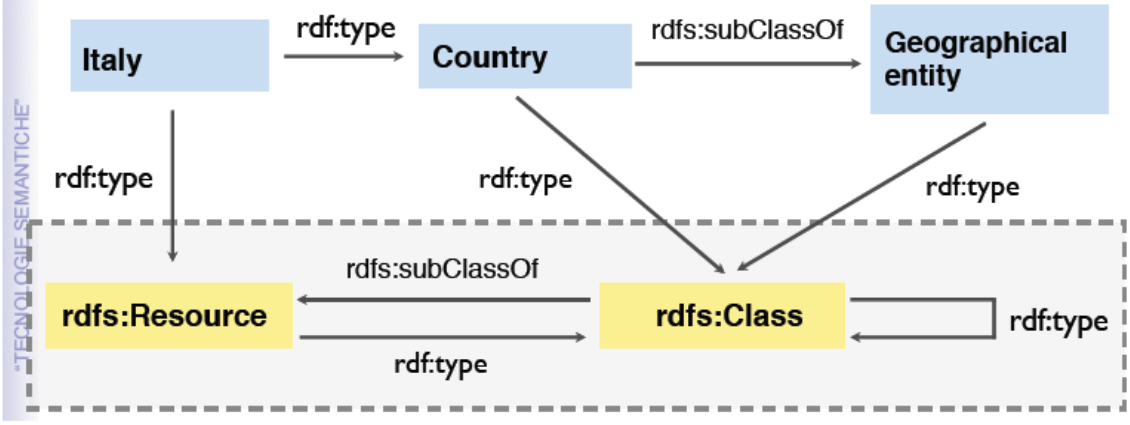
Ontologia:rappresentazione formale, condivisa ed esplicita di una concettualizzazione di un dominio di interesse
Tipi di ontologia:

- **Thesauri** (ontologie lessicali): dizionario che raggruppa termini di uno o più linguaggi in gruppi di sinonimi e termini collegati, inclusi iperonimi (concetto più generale) e iponimi (concetto più specifico)
- **Tassonomie:** classificazione degli oggetti di un dominio rappresentati da una gerarchia con una struttura ad albero
- **Ontologie assiomatiche:** espressione del significato inteso dal vocabolario inteso in termini di categorie primitive e la struttura del dominio d'interesse è impostata da assiomi e linguaggio logico

RDF Schema: set di classi con certe proprietà usando il modello dati estendibile di rappresentazione di conoscenza di RDF, fornendo elementi base per la descrizione di ontologie (vocabolari RDF), intesi per strutturare risorse RDF, salvabili in insiemi di triple alle quali si può accedere con SPARQL

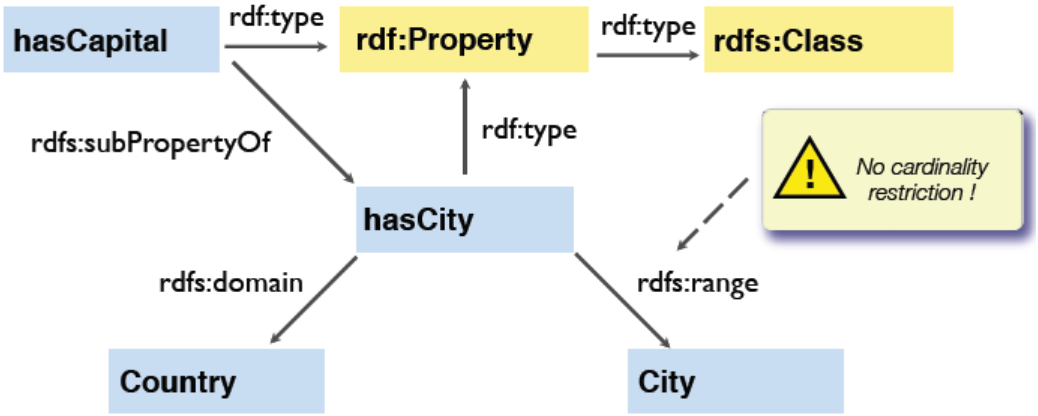
- RDF Schema risolve il problema di RDF che non ha meccanismi per descrivere proprietà e le relazioni
- **Classi principali**
  - **rdfs:Resource=**tutte le cose descritte da RDF sono istanze di questa classe
  - **rdfs:Class=** dichiara una risorsa come classe di un'altra risorsa (rdfs:subClassOf le sottoclassi)
  - **rdfs:Literal=** valori letterali (stringhe, interi...)
  - **rdf:property=** proprietà RDF
- **Classi per la tassonomia dell'RDF Schema**
  - **Gerarchie di classe=** rdfs:Class e rdf:subClassOf
  - **Gerarchie di proprietà=** rdf:property e rdfs:subPropertyOf
  - **Dominio e restrizione su proprietà=** rdfs:domain e rdfs:range

The **rdf:type** property may be used to state that a resource is an instance of a class.
The **rdfs:subClassOf** property may be used to state that one class is a subclass of another

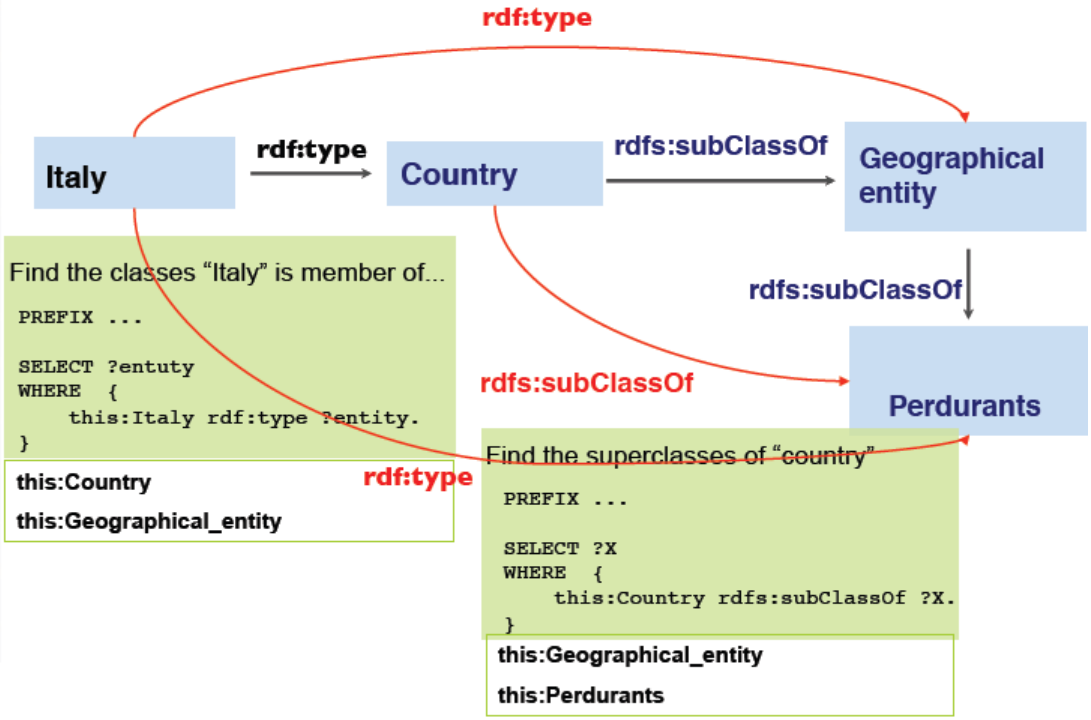


RDF Schema definition

RDF properties are binary relation between subject resources and object resources.
rdf:Property is the class of RDF properties.
rdf:Property is an instance of **rdfs:Class**.



RDFS supports reasoning, SPARQL query answering does not

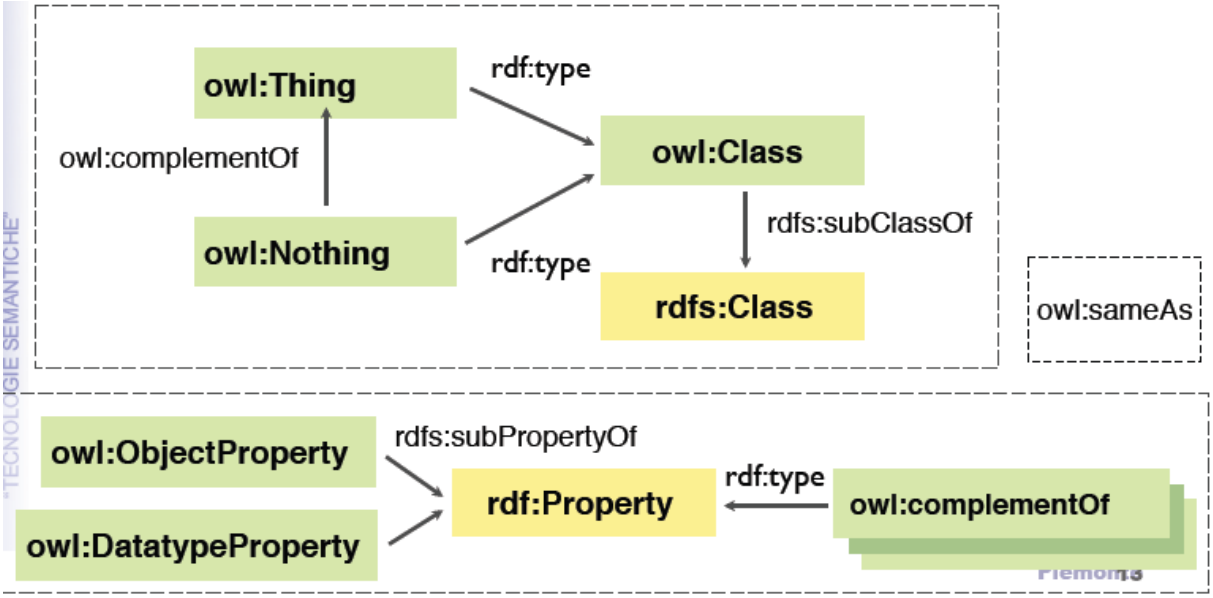


RDF-S semantics (partial)

if	then
x rdfs:subClassOf y . a rdf:type x .	a rdf:type y .
x rdfs:subClassOf y . y rdfs:subClassOf z .	x rdfs:subClassOf z .
x a y . a rdfs:subPropertyOf b .	x b y .
a rdfs:subPropertyOf b . b rdfs:subPropertyOf c .	a rdfs:subPropertyOf c .
x a y . a rdfs:domain z .	x rdf:type z .
x a u . a rdfs:range z .	u rdf:type z .

OWL (Ontology Web Language): linguaggio di *markup* per rappresentare esplicitamente significato e semantica di termini con vocabolari e relazioni tra gli stessi.

- OWL è progettato per avere la massima compatibilità con RDF Schema
- OWL fornisce vocabolari addizionali con la semantica formale



SPARQL(SPARQL Protocol And RDF Query Language): linguaggio di interrogazione per dati rappresentati in RDF

- Struttura simile a SQL
- Linguaggio di query/protocollo standard
- Sola lettura
- RDF

Specifiche principali

- Linguaggio query di SPARQL per RDF
- Protocollo SPARQL per RDF
- Risultati Query di SPARQL in formato RDF
- Casi d'uso e requisiti per l'accesso dati RDF

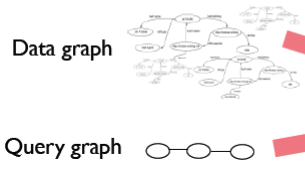
What is a Query ?

A Query is question.
Answering requires understanding:

An RDF graph

- The query
- The thing queried
- The relationship between them

SPARQL uses RDF graph for queries



SPARQL Query Types

Ground graphs as queries

Boolean query

Returns TRUE if
Data Graph => Query Graph

```
ASK
WHERE {
 book:0001 book:author author:01 .
}
```

ASK

Other SPARQL query types:
DESCRIBE, CONSTRUCT

Variable

SELECT

```
SELECT ?author
WHERE {
    book:0001 book:author ?author .
}
```

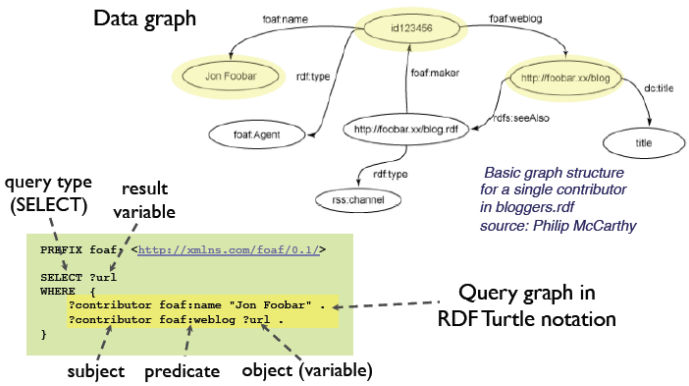
Blank nodes as query variables

There is an answer if
μ (query graph) $\subseteq$ data graph

Results is a Table, not a Graph

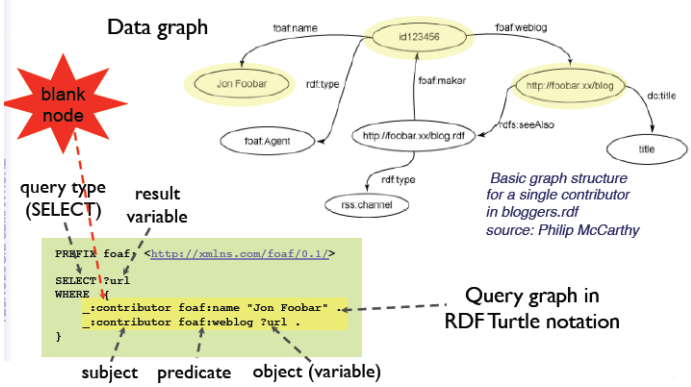
Graph pattern query structure

Find the URL of the blog of the person named "Jon Foobar"



Graph pattern query structure

Find the URL of the blog of the person named "Jon Foobar"



One more example

Data graph

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .

_:author1 <http://www.w3.org/2001/vcard-rdf/3.0#FN> "J.K. Rowling".
<http://example.org/book/book1> dc:title "Harry Potter and the Chamber of Secrets".
<http://example.org/book/book1> dc:creator _:author1 .
<http://example.org/book/book2> dc:title "Harry Potter and the Prisoner Of Azkaban" .
<http://example.org/book/book2> dc:creator _:author1 .
```

SPARQL Query

```
PREFIX books:   <http://example.org/book/>
PREFIX dc:      <http://purl.org/dc/elements/1.1/>
PREFIX vcard:   <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?bookTitle ?authorName
WHERE
  { ?book dc:creator ?author .
    ?book dc:title ?bookTitle .
    ?author vcard:FN ?authorName
}
```

Try this example online:

http://www.sparql.org/query.html

Results

bookTitle	authorName
Harry Potter and the Chamber of Secrets	J.K. Rowling
Harry Potter and the Prisoner Of Azkaban	J.K. Rowling

Value constraints

```
ASK WHERE
  {
    ?author vcard:FN "J.K. Rowling" .
}
```

This query returns true iff there is an author with name "J.K. Rowling"

SPARQL provides an operation to test strings, based on regular expressions.
The syntax is different from the SQL "LIKE" !

```
ASK WHERE
  {
    ?author vcard:FN ?name .
    FILTER regex(?name, "rowling", "i")
}
```

This query returns true iff an author names contains the string "rowling"

Filter flags: "i" means "Case insensitive"

Filter Pattern

The regular expression language is the **XQuery** regular expression language which is codified version of that found in **Perl**.

http://www.perl.org

```
ASK WHERE
  {
    person:0001 info:age ?age .
    FILTER (?age >= 18)
  }
```

Filter on an integer property. This query returns true iff there is a person 0001 has age >= 18

Filter on the value of an integer property.

```
SELECT ?name ?age
WHERE
  {
    ?person info:name ?name .
    ?person info:age ?age .
    FILTER (?age >= 18)
  }
```

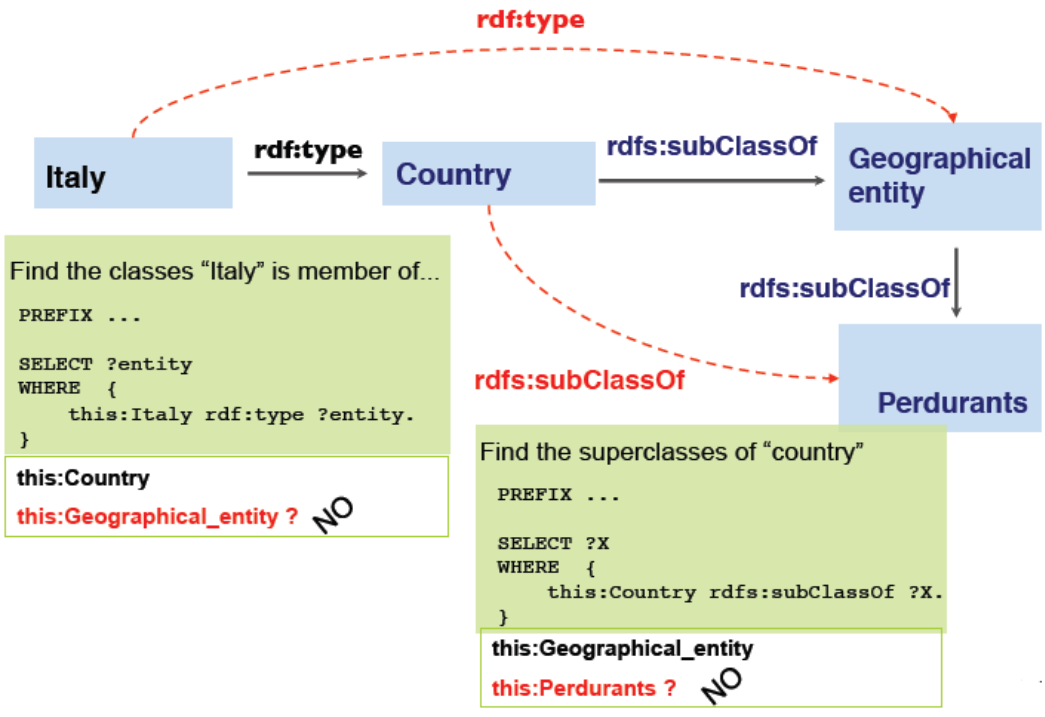
This query returns the name and age of each entity with age >= 18

SPARQL has a the ability to query for data but not to fail query when that data does not exist.

```
SELECT ?name ?age
WHERE
{
    ?person info:name ?name .
    OPTIONAL { ?person info:age ?age . FILTER ( ?age > 18 ) }
}
```

This query returns the name and age of each entity with unknown age or age >= 18

SPARQL query answering is not reasoning!



Find the classes "Italy" is member of...

```
PREFIX ...

SELECT ?entity
WHERE  {
    this:Italy rdf:type ?entity.
}
```

this:Country

this:Geographical_entity ? NO

Find the superclasses of "country"

```
PREFIX ...

SELECT ?X
WHERE  {
    this:Country rdfs:subClassOf ?X.
}
```

this:Geographical_entity

this:Perdurants ? NO

PROGRAMMAZIONE CONCORRENTE

La programmazione concorrente è la programmazione basata su programmi concorrenti

- **Programma sequenziale:** un singolo thread di controllo;

- **Programma concorrente**: più thread di controllo operano assieme condividendo le risorse in modo da svolgere più operazioni in parallelo (quindi più operazioni allo stesso tempo);
  - **Thread:** parte di un processo

Effetti della programmazione concorrente

- Miglioramento delle prestazioni (grazie al parallelismo)
- Aumento del trasferimento dati (throughput) (una chiamata I/O ha bisogno di un blocco per thread)
- Aumento velocità di risposta (i thread di priorità maggiore vengono eseguiti prima)
- Struttura più adatta a interfacciarsi col mondo reale (con richieste mulitiple)

La concorrenza è ormai comune ma è facilmente soggetta ad errori

Sistema reattivo: sistema che funziona indefinitamente nel tempo, che deve auto sostenersi e funziona reagendo a comandi posti dall'esterno (di solito i sistemi concorrenti sono sistemi reattivi)

Modello:rappresentazione semplificata del mondo reale

- Concentrazione su determinati aspetti e comportamenti del sistema
- Vefica meccanica dei processi

Modalità di rappresentazione dei modelli

- **automi a stati finiti**, ossia sistemi di transizione (LTS=Labelled Transition Systems)
- **codice:**FSP (Finite State Processes) utilizzando LTSA

Traccia:sequenza di istruzioni

Numero di transizioni di un automa: (numero elementi)^(numero stati)

- E' la parte più importante del processo perché è la parte che lo fa funzionare
- Un solo processo alla volta nella sezione critica (mutua esclusione)

Cosa contiene il sistema di transizioni

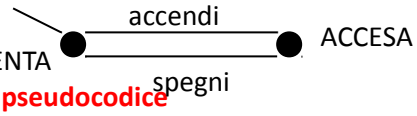
$A=(S,E,T,S_0)$

- S=insieme degli stati finiti
- E=alfabeto
- T=insieme delle transizioni
- S_0=stato iniziale
- Una transizione è determinata da stato iniziale, simbolo dell'alfabeto e stato successivo

Come disegno il sistema di transizioni

Esempio

SPENTA=(accendi->ACCESA).
ACCESA=(spegni->SPENTA).



Struttura processo concorrente in pseudocodice

```
//tutte le altre cose
while(true){ //1
        snc(); // Sezione non critica: istruzioni del processo non indispensabili al funzionamento del processo stesso
        ingresso(); //istruzioni necessare per raggiungere la risorsa condivisa
        sc();//Sezione critica: parte che contiene le istruzioni che utilizzano le risorse in mutua esclusione (un
        processo //per volta) e fa funzionare il processo
        uscita(); //istruzioni che rilasciano la risorsa
}
```

Proprietà da garantire

- **Accesso esclusivo alla risorsa esclusiva** (mutua esclusione, sincronizzazione)
  - **Lock:** quando un processo acquisisce la risorsa deve acquisire il lock, così viene bloccata agi altri processi e n possono usarla perché non hanno il lock
  - **Flag:** quando un processo acquisisce la risorsa, scatta un flag che impedisce l'accesso ad altri processi o permette l'accesso quando ritorna al valore precedente
  - **Di conseguenza, vengono eliminate le interferenze**
    - **INTERFERENZA:** Aggiornamento distruttivo, causato da letture e scritture non controllate
    - Può causare notevoli errori perchè i dati possono essere letti da un processo quando un altro sta scrivendo, quindi il processo lettore non ha i dati più recenti
  - **Monitor:** oggetto utilizzato per regolare gli accessi a una risorsa condivisa, può anche informare tutti gli altri thread se la risorsa è disponibile e occupata (esempio monitor: parcheggio che conta quanti posti sono occupati, blocca le entrate se il parcheggio è vuoto, blocca le uscite se il parcheggio è vuoto)
    - Dato che un monitor è una classe a sé, può contenere metodi, flag, lock e qualunque altra cosa

- **Progresso** (all'inizio, un processo ha la risorsa, poi, col proseguire del tempo, la risorsa è passata a un altro processo in attesa)
  - **I processi che non stanno usando la sezione critica non devono influenzare i processi che stanno usando la sezione critica** (altrimenti l'insieme dei processi si adegua alla velocità del processo più lento)
- **Attesa limitata** (un processo deve entrare nella sezione critica entro un tempo finito (il processo non può aspettare per sempre)

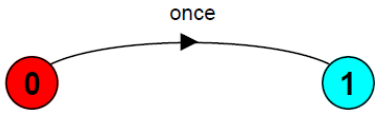
DEADLOCK: situazione in cui due o più processi o azioni si bloccano a vicenda, aspettando che uno esegua una certa azione (es. rilasciare il controllo su una risorsa come un file, una porta input/output ecc.) che serve all'altro e viceversa.

- Non è effettivamente risolvibile (se non spegnendo il computer), ma è prevenibile rispettando le proprietà elencate sopra

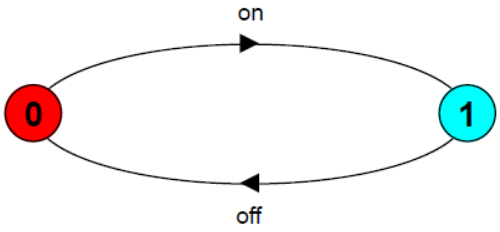
PROBLEMA DA RISOLVERE: INTERFERENZE

ISTRUZIONI FSP (con relative traduzione in automi)

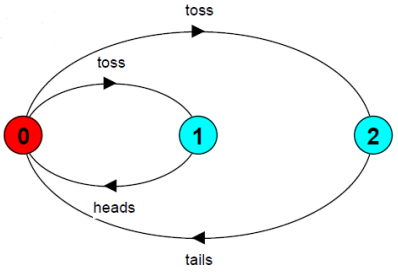
- **convenzione:** azioni scritte in minuscolo, processi scritti con l'iniziale maiuscola
- **azione semplice: (x->P)** dove x è un azione e P è il processo
  - ONESHOT = (once -> STOP).



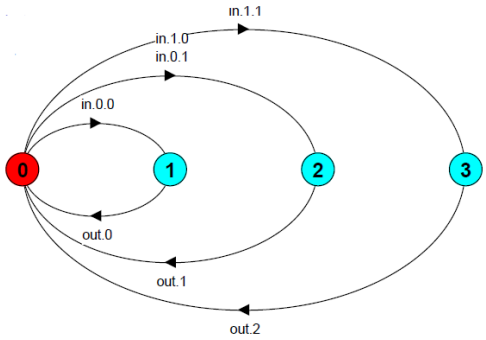
- **Ricorsione:** l'unico modo per ripetere determinati comportamenti
  - SWITCH = OFF,
  - OFF = (on -> ON),
  - ON = (off-> OFF).
  - **Posso compattare**
  - SWITCH = OFF,
  - OFF = (on ->(off->OFF)).
  - **Posso ulteriormente compattare**
  - SWITCH = (on->off->SWITCH).



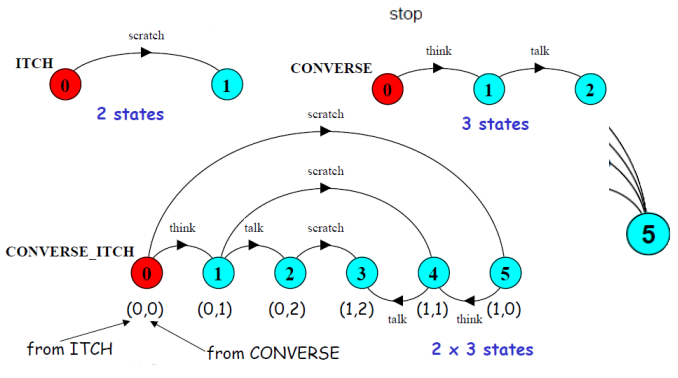
- **Scelta (non deterministica): (x->P | y->Q)** dove x e y sono azioni distinte e P e Q sono processi distinti
  - **Se avviene x eseguo P oppure se avviene y eseguo Q**
  - COIN = (toss->HEADS|toss->TAILS),HEADS= (heads->COIN),TAILS= (tails->COIN).



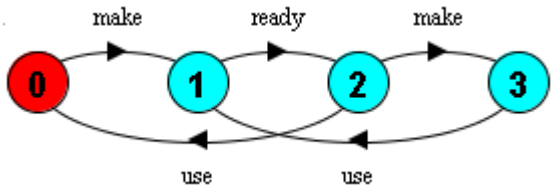
- **Esecuzioni multiple: BUFF= (in[i:0..x]->out[i]-> BUFF).** Dove x è valore massimo
  - **Genero x copie dello stesso processo**
  - BUFF = (in[i:0..3]->out[i]-> BUFF).
  - **Equivale a**
  - BUFF = (in[0]->out[0]->BUFF
  - |in[1]->out[1]->BUFF
  - |in[2]->out[2]->BUFF
  - |in[3]->out[3]->BUFF).
  - **Oppure**
  - BUFF(N=3) = (in[i:0..N]->out[i]-> BUFF).
  - **Notazione compatta (labelling): ||** **NP=([1..N]:P).** dove P è un processo dichiarato in precedenza
    - Const N=10
    - P=(snc->acq->sc->ril->P).
    - ||NP=([1..N]:P).



- **Azioni con guardia: (when B x -> P | y -> Q)** dove B è la condizione, x e y sono le azioni, P e Q sono processi
  - **Se la condizione B è vera, posso eseguire x che porta a P oppure y che porta a Q**
  - COUNTDOWN (N=3) = (start->COUNTDOWN[N]),
  - COUNTDOWN[i:0..N] =
  - (when(i>0) tick->COUNTDOWN[i-1]|
  - when(i==0)beep->STOP|
  - stop->STOP).
- **Azioni parallele: (P||Q)** dove P e Q sono processi
  - **due o più processi possono lavorare contemporaneamente**
    - **concorrenza:** processi multipli **logicamente**

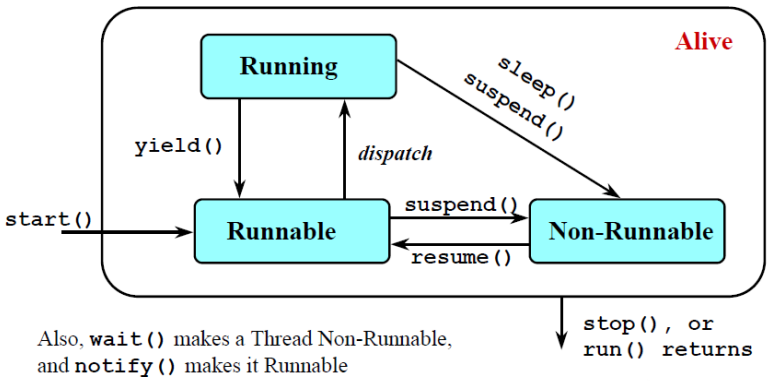


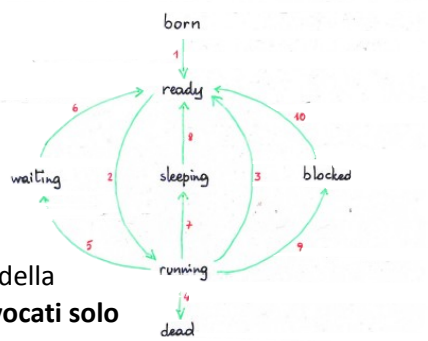
    - **parallelismo:** processi multipli **fisicamente**
  - ITCH = (scratch->STOP).
  - CONVERSE = (think->talk->STOP).
  - ||CONVERSE_ITCH = (ITCH || CONVERSE).
    - **Possibili esiti**
    - think->talk->scratch
    - think->scratch->talk
    - scratch->think->talk
- **proprietà algebriche**
  - **commutativa:** (P||Q)=(Q||P).
  - **associativa:** (P||(Q||R))= ((P||Q)||R) = (P||Q||R).
- **azioni condivise: più processi diversi, stesso simbolo dell'alfabeto**
  - **comunicazione tra processi**
  - **l'azione condivisa dev'essere svolta da tutti i processi che la possiedono per continuare l'esecuzione di tutti**
  - MAKER = (make->**ready**->MAKER).
  - USER = (**ready**->use->USER).
  - ||MAKER_USER = (MAKER || USER).
  - **Grazie alle azioni condivise, ho la mutua esclusione**
  - LOCK = (acquire->release->LOCK).
  - ||LOCKVAR = (LOCK || VAR).
  - set VarAlpha = {value.{read[T],write[T], acquire, release}}
  - e
  - TURNSTILE = (go -> RUN),
  - RUN = (arrive-> INCREMENT
  - |end -> TURNSTILE),
  - INCREMENT = (value.acquire
  - -> value.read[x:T]->value.write[x+1]
  - -> value.release->RUN
  -)+VarAlpha.
  -



IMPLEMENTAZIONE IN JAVA

- **ogni processo è una classe separata**
- bisogna distinguere i tipi di processi:
  - **attivi:** thread metodi con l'attributi synchronized
  - **passivi:** fornitori di servizi ai thread
- **si utilizza la classe Thread**
- **classMyThread extends Thread {**
- **publicvoid run(){**
- **//.....**
- **}**
- **}**
- **Per creare un nuovo thread** Thread a = new MyThread();
- **Metodi rilevanti**
  - **run()/start()** (da riempire):attiva il thread
  - **stop()** (da riempire): termina il thread
  - **notify():**sblocca un thread in attesa
  - **notifyAll():** sblocca tutti i thread in attesa
  - **wait():**mette un thread in attesa, rilasciando il lock della
    - **notify(), notifyAll() e wait() possono essere invocati solo in classi synchronized**
  - (gli altri nelle immagini con il ciclo di vita di un thread)
- **Sincronizzazione in Java**





2: dispatch
3: quantum expiration, yield, interrupt
4: complete
5: wait
6: notify, notifyAll
7: sleep risorsa
8: sleep interval expires
9: issue I/O
10: I/O completion

- **Per garantire la mutua esclusione in java** le classi (solo quelle attive) devono avere la parola chiave syncrhronized (esempio syncronized void method())
- public synchronized void act()
- throws InterruptedException
- {
- while (!cond) wait();
- // modify monitor data
- notifyAll();
- }

- **Risveglio spurio (Java):** un thread in attesa può risvegliarsi dall'attesa della risorsa senza effettivamente essere sbloccato (da comandi come notify() e notifyAll())
  - Per questo il comando wait() è nel ciclo while, in modo da controllare continuamente che il thread sia effettivamente in attesa

ANALISI DEI SISTEMI CONCORRENTI

- **Proprietà:** un sistema S ha proprietà p se p è vero in tutte le sue condizioni
  - **Atomiche:** categorizzate in un solo gruppo
  - **Complesse**: categorizzate in più gruppi
  - Sono espresse in logica temporale (logica booleana che introduce il concetto di evoluzione nel tempo)
- **Safety:** (sicurezza) niente di brutto succederà
  - Non c'è lo stato di STOP/ERRORE (o almeno può esserci ma non viene mai raggiunto)
  - Max safety=non succede nulla di brutto
  - **Fomula base**
  - property SAFE_ACTUATOR = (command->respond->SAFE_ACTUATOR).
  - **Esempi (in una codifica FSP):**
    - property EDUCATO=(bussa->entra->EDUCATO).
- **Liveness:** (essere vivo) qualcosa di buono avverrà
  - Max Liveness: avviene tutto
  - **progress** P = {a1,a2,..an) definisce una proprietà di progresso P che assicura che in una esecuzione infinita del sistema, almeno una delle azioni a1,a2,..an verrà eseguita infinite volte (esecuzione imparziale)
    - **Esecuzione imparziale:** per ogni stato attraversato infinite volte, la computazione sceglie infinite volte tutte le computazioni possibili **(LTSA usa questa politica)**
    - **Esempio applicazione (con resto del codice FSP)**
    - MONETA = (lancia->testa->MONETA | lancia->croce->MONETA).
    - MONETA system: **progress TESTA={testa}**
    - **progress CROCE={croce}**

MOBILE APPS

In un mondo che punta sempre di più alla mobilità, anche nell'accesso alle informazioni, diventa necessario predisporre delle applicazioni in grado di girare su dispositivi mobili in tutta sicurezza e facilità.
Tipi di mobile apps: Web Mobile (WebApp), applicazioni native o applicazioni ibride.

Webapp:

- normali applicazioni web che permettono di simulare l'aspetto delle UI proprie di app native.
- **Vantaggi:**
  - scritte in markup HTML (Di solito HTML5)
  - non devono essere sottoposte al processo di approvazione da parte del vendor per finire sui market digitali (quindi tempi di sviluppo-rilascio più rapidi).
  - Tempi di sviluppo più bassi
- **Svantaggi** :
  - mancanza di interazione con hardware e software del device
  - mancata presenza sullo store del device (mancanza di visibilità per la webapp)

Applicazioni native:

- Sviluppate in una IDE (Integrated Development Environment) dedicate
- Scaricabili (gratis o a pagamento) tramite lo store del device (dopo l'approvazione del vendor).
- **Vantaggi:**

- ottima visibilità sui marketplace,
- possibilità di ampliare le funzionalità dell'app grazie all'accesso fisico al device ed ai suoi dati (posizione GPS, rubrica ecc…)
- interazione con quasi tutte le funzionalità del dispositivo
- di solito, presentano un grado di performance maggiore (se il codice è scritto bene)
- Funzionamento off-line

- **Svantaggi:**
  - Necessarie diverse implementazioni per i diversi S.O (Android, iOS, Windows Phone...)
  - Alti tempi e costi di sviluppo (linguaggi un po' a più basso livello ma potenti, inoltre serve una versione per ogni sistema operativo)

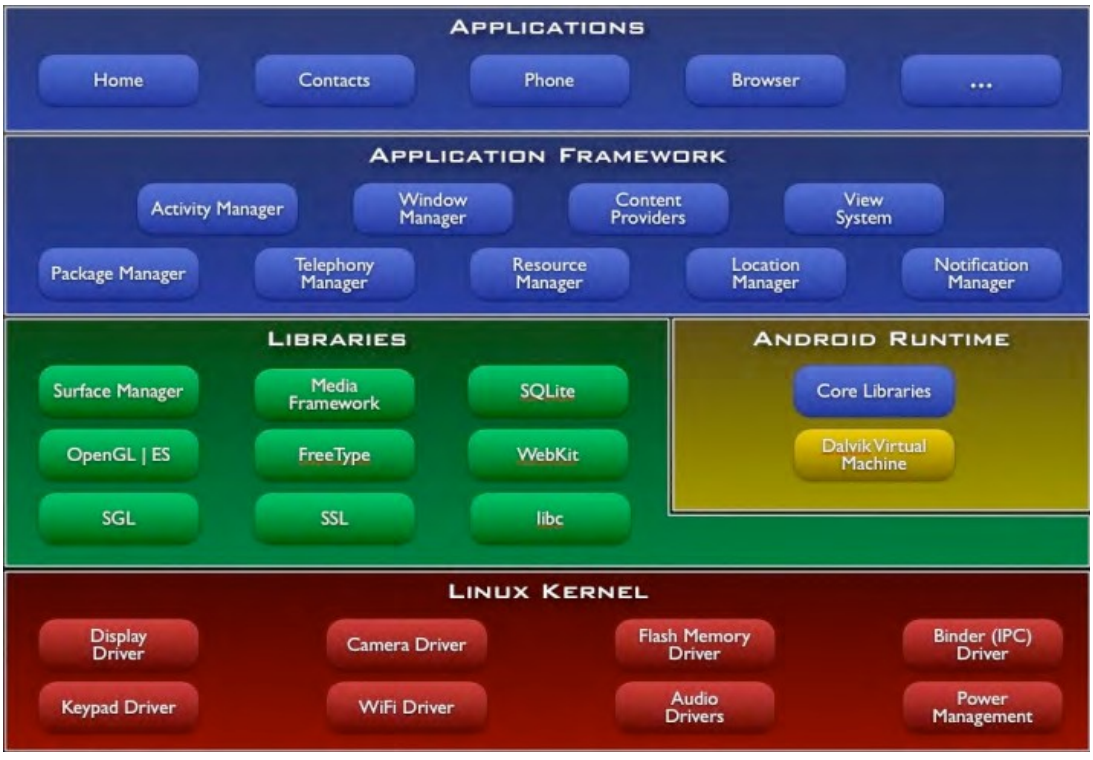
app ibride:

- Via di mezzo tra webapp e applicazioni native, cioè delle app native con strumenti di gestione di interfacce HTML come le web app
- Ingloba gli aspetti positivi delle altre 2 tipologie di app
- **Vantaggi:**
  - Propensione al multi-piattaforma
  - Velocità nella realizzazione
- **Svantaggi:**
  - possibile problemi di performance (nel caso di device offline, alcune parti non funzionano proprio)
- La costruzione di app ibride avviene di solito in HTML5, CSS e JAVASCRIPT, tramite l'uso di frame work come PhoneGap che facilitano l'accesso alle risorse hardware e software presenti sul device.
- Data la capacità di (quasi) tutti i device di interpretare i linguaggi di programmazione citati, il codice viene scritto una volta sola per tutti i mercati di destinazione.

ANDROID

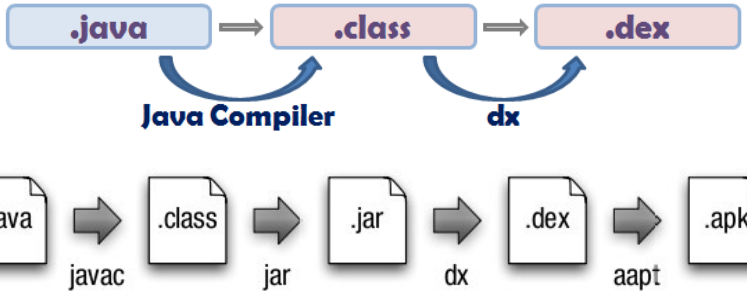
- Android è un sistema operativo mobile basato su linux (quindi open-source) sul quale ogni produttore crea la sua versione (Google ad esempio aggiunge servizi come Maps, Samsung monta le Samsung Apps).
- Esso, data la sua caratteristica di essere open source e quindi la capacità di contenere i costi, è diventato il SO mobile più utilizzato al mondo, ed è presente (in diverse declinazioni), su smartphone, tablet, TV e altro.
- Sviluppare applicazioni per android prevede la conoscenza di Java o Python, in quanto le applicazioni girano su di una virtual machine (Davlik Virtual Machine).

Struttura architettura android



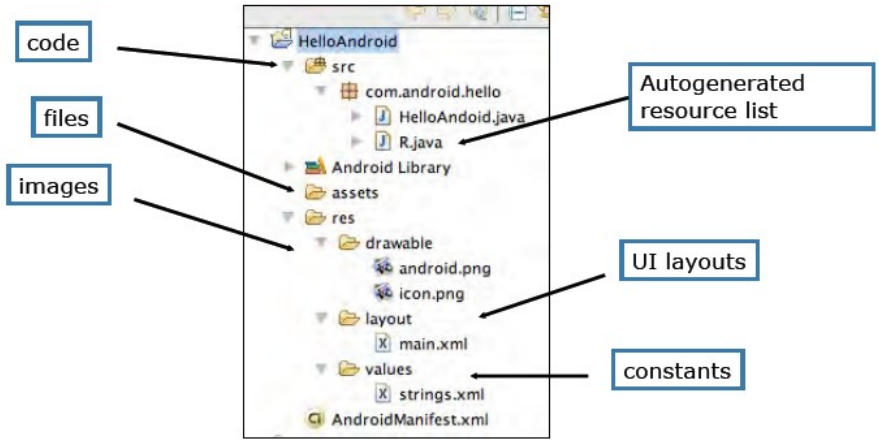
- **Linux kernel:** strato d'astrazione fra hardware e resto del software, occupandosi del mantenimento dei servizi di base (controllo dei processi, della rete, dei driver, della sicurezza...)
  - Presenta un componente dedicato all'IPC (Inter Process Communication) che permette ai componenti di scambiarsi messaggi
  - Il telefono non è più parte del kernel, ma è diventato un'app
- **Librerie:** set di librerie scritte in c/c++ che vengono utilizzate dai componenti del sistema e che vengono esposte agli sviluppatori tramite l'Android Application Framework

- **Alcune librerie**
  - **Surface Manager:** si occupa delle gestione delle view ossia le componenti dell'UI
  - **Open GL ES:** Versione ottimizzata per mobile delle OPEN GL per la grafica 3D
  - **SGL :** (Scalable Graphics Library) – libreria per la grafica in 2D
  - **FreeType :** gestore dei font
  - **Media Framework :** gestore codec multimediali
  - **SQLite:** DBMS relazionale molto leggero
  - **WebKit:** Browser Engine Open Source (come Safari e Chrome)
  - **SSL :** (Secure Socket Layer) Gestione delle connessioni di rete sicure
  - **Libc :** Versione linux embedded della libreria std C
- **Runtime:**librerie che forniscono la maggior parte delle funzionalità principali delle libreria di Java,
  - fornisce API per
    - strutture dati
    - utilities
    - accesso ai file
    - accesso alla rete
    - grafica: ecc...
  - **SDK di Android (Davlik Virtual Machine):** fornisce un ambiente in cui ogni app android viene lanciata



    - ogni applicazione ha il suo processo, con la sua istanza di Dalvik VM.
    - Il compilatore Java della DVM traduce i file .class in file .dex (bytecode), per minimizzare l'impatto sulla memoria (e più si compila, più si diminuisce la dimensione dell'app)
    - Ma la DVM è anche la responsabile per
      - lentezza di risposta
      - elevato consumo di memoria
    - Dalvik utilizza un compilatore JIT (just-in-time), cioè traduce il byte code in linguaggio macchina al momento dell'esecuzione, risultando in un maggior consumo di batteria in quanto usa più cicli di CPU per leggere/interpretare/eseguire il codice.
  - **Android Runtime Virtual Machine (ART)**: Dalla versione attuale, 4.4 (hehe), Android ha sostituito la DVM con ART che utilizza un compilatore AOT (ahead-of-time), compilando il codice al momento dell'installazione, così al momento dell'esecuzione ha la velocità di caricamento come un'app nativa (prestazioni migliorate, minor consumo di batteria perché non servono ulteriori compilazioni)
    - Un grande vantaggio di ART è che permette agli sviluppatori di Android di continuare a scrivere lo stesso codice che avrebbero scritto con Dalvik
- **application framework**: abilita e semplifica il riutilizzo dei componenti
  - gli sviluppatore hanno completo accesso al set di API utilizzate dalle applicazioni di sistema.
  - Gli utenti possono sostituire i componenti
  - Ogni applicazione può pubblicare le sue capacità che le altre app possono usare
  - **Qualche API**
    - **Activity Manager:** gestione del ciclo di vita delle attività (elementi fondamentali della struttura delle app)
    - **Package Manager:** gestione del ciclo di vita delle applicazioni nel dispositivo (installazione/disinstallazione/spazio occupato)
    - **Window Manager:** gestore delle finestre nel dispositivo
    - **Telephony Manager:** gestore delle funzionalità telefoniche
    - **Content Provider:** gestore dati nel sistema
    - **Resource Manager:** gestore e ottimizzatore risorse del dispositivo
    - **View System:** render delle view e gestore dei relativi eventi
    - **Location Manager:** gestore della localizzazione (fondamentale per app delle mappe)
    - **Notification Manager:** gestore delle notifiche del sistema
- **Applicazioni:** android fornisce delle applicazioni di sistema (Mail, SMS, Calendario, Browser, Contatti ecc…)
  - Queste applicazioni sono scritte in Java

Tipica struttura dei file delle Apps



Tutte l
dipenc

sfruttando le gerarchie

Interfaccia grafica

- la GUI di un'app è costruita utilizzando una gerarchia di oggetti View e ViewGroup
- la struttura è un po' rigida ma ottimizzata per cellulari/tablet
- android fornisce un vocabolario XML che corrisponde alle sottoclassi di View e ViewGroup in modo da poter definire gli elementi della GUI

Logica applicativa

- **Intent**: oggetto che fornisce un binding di esecuzione tra due componenti separate, come ad esempio la pressione di un tasto e la comparsa di un messaggio.
  - Rappresenta "l'intento di fare qualcosa" dell'app
  - Si possono usare per molte cose, ma di solito si usano per iniziare un'altra attività
  - Ogni intent viene associato ad un attività (activity) che è una singola azione che l'utente può fare.
- **Activity:** è una singola cosa concentrata che l'utente può fare
  - Di solito, le activity interagiscono con l'utente, quindi la classe Activity crea una nuova window in cui bisogna scrivere la UI.

Componenti applicativi

- Ogni applicazione Android si basa su degli Application Component, i componenti fondamentali di un'applicazione.
- Esistono 4 tipi di componenti:
  - **Activity:**
    - tipicamente è una singola schermata dell'interfaccia utente
    - più activity possono formare l'applicazione
    - inseme delle activity=user experience
    - un activity che compone un app può essere"usata" da altre app (esempio, apri SMS da Contatti)
  - **Service:**
    - Agiscono in background,
    - senza UI
    - si occupano di operazioni a lunga durata (es. riproduzione audio fuori da musica)
  - **Content Provider:**
    - gestisce e condivide i dati tra app
    - Ogni app scrive su storage persistente
    - Con il content provider, un'altra app può leggere i dati e (se concesso dallo stesso) modificarli. (es. Contatti accessibili da altra app)
  - **BroadcastReceiver:**
    - Componenti che rispondono a notifiche provenienti da altre app o da sistema, senza UI.
    - Lanciati dal sistema (esempio: notifica della batteria scarica) o da app (esempio: notifica fine download)
    - ogni notifica viene recapitata con un oggetto di tipo Intent
- **Vantaggi**
  - Alta personalizzazione
  - Servizi in background
  - Notifiche per gli eventi in background
  - Multitasking
  - Widget
  - Possibilità di installare ROM modificate
  - Facile accesso a migliaia di App con il play store
  - Google Maniac
- **Svantaggi**
  - Pubblicità
  - Continua connessione a internet (per accedere ai servizi)

FILOSOFIA DI ANDROID

- **Le applicazioni possono comunicare tra di loro per condividere i servizi a patto che l'app lo dichiari**
  - Esempio: se un app vuole fare una foto, l'app delle foto viene usata per fare la foto
  - Problema: le app dipendono fra di loro quindi ci sono problemi di affidabilità e sicurezza delle altre app che forniscono i servizi)
  - le app android non hanno un entry point definito (main), ma diversi entry point separati, uno per activity
  - la comunicazione fra le app non necessita la scrittura di ulteriore codice e l'utente non se ne accorge
- **Quando il sistema attiva un componente, attiva il processo per quell'applicazione e istanzia le classi necessari per quel componente**
  - Esempio: se l'applicazione che si sta codificando attiva il componente che scatta le foto appartenente all'applicazione della fotocamera, il componente sarà attivato nel processo dell'app fotocamera
  - Questo implica che le app android hanno diversi entry point (non esiste il metodo main())
- **Il sistema esegue le app in processi separati, ognuno con un file di permessi che ne regolamenta le potenzialità.**
  - un'app non può direttamente attivare un componente di un'altra app,
  - ma il sistema può.
  - l'app invia un messaggio di sistema richiedendo di attivare il componente di un'altra app, ed è il sistema ad attivarla.
  - Per attivare funzioni esterne, l'app invia un messaggio al sistema di tipo startActivity(Intent i) o startActivityForResult().
- Per lanciare un service si usano i comandi start Service() o bindService(),
- per attivare un broadcast di notifica l'intent definisce il messaggio da inviare, tramite l'uso di sendBroadcast(), sendOrderedBroadcast(), sendStickyBroadcast().
- L'unica eccezione è il Content Provider, che non è attivato da un intent ma bensì da una richiesta ContentResolver, che si occupa di gestire le transazioni con il content provider.
- E' possibile eseguire una query ad un ContentProvider invocando sul ContentResolver il metodo query().
- **le azioni devono essere attivate via Intent**, in due modi:
  - Esplicitando il nome del componente desiderato
  - tramite un Intent Action.
    - Questa struttura dati descrive il tipo di azione ed eventualmente i dati necessari, ma non da chi sia richiesta.
    - Il sistema troverà un componente sul device in grado di eseguire l'azione e lo attiverà (nel caso ne esista più di uno, viene posta la scelta all'utente).
    - Il sistema identifica i componenti in grado di rispondere ad un determinato intent grazie ad un intent filter che si trova nel file manifest.xml dell'applicazione.
- **Android deve conoscere l'esistenza di un'applicazione e di come sia fatta**
  - ogni applicazione dichiara nel file manifest.xml la propria descrizione, i permessi che l'utente deve avere per eseguire l'applicazione, il livello minimo di api richiesto, le risorse api necessarie e un particolare tag <intent-filter> che indica le abilità dell'applicazione.
  - Le applicazioni sono poi composte da file resource, cioè caratteristiche che non cambiano durante l'esecuzione del programma (label, file audio, imgs), e file code che contengono le parti programmative e la gestione degli eventi.
  - Le resource hanno un ID univoco in modo da poter essere utilizzate all'interno del codice o all'interno di altre resource.

CLOUD COMPUTING

Cloud Computing:un modello per abilitare, tramite l'utilizzo della rete, l'accesso diffuso, agevole e a richiesta ad un insieme condiviso e configurabile di risorse di elaborazione (reti, server, spazio su disco, applicazioni e servizi) che possono essere acquisite e rilasciate rapidamente e con uno sforzo minimo di gestione o interazione del fornitore delle risorse.

Caratteristiche principali:

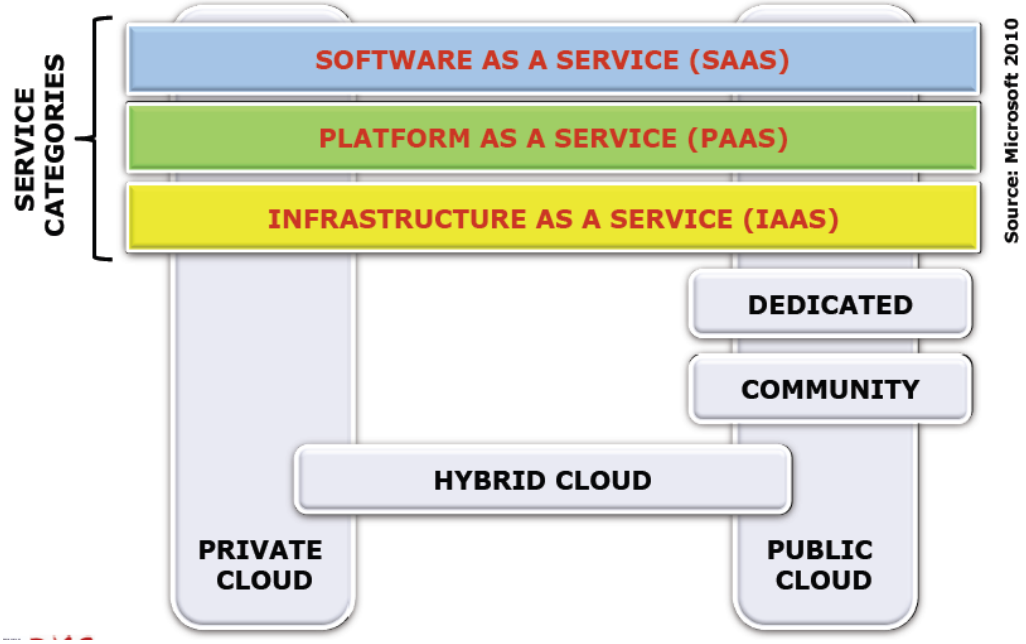
- **Self-Service su richiesta:** Un consumatore può acquisire autonomamente e automaticamente le risorse necessarie senza interazione umana con il fornitore di tali risorse

- **Ampio accesso in rete:** Le risorse sono disponibili in rete e accessibili tramite metodi standard e attraverso piattaforme eterogenee
- **Condivisione risorse:** Le risorse di calcolo del fornitore sono messe in comune a molteplici consumatori tramite un modello condiviso (multi-tenant) tra le diverse risorse fisiche e virtuali assegnate e riassegnate dinamicamente in base alla richieste
- **Elasticità rapida:** Le risorse possono essere acquisite e rilasciate automaticamente per scalare verso l'interno o verso l'esterno in base alla domanda.
  - Al consumatore, le risorse appaiono illimitate e disponibili in ogni momento
- **Servizio misurato:** I sistemi cloud controllano e ottimizzano l'uso delle risorse con capacità di misurazione appropriate per il servizio offerto (memoria,CPU,banda,utenti, disco ecc…)

Concetti fondamentali alla base del cloud computing

- **standardizzazione dell'infrastruttura tecnologica**
- **automatizzazione dei controlli** (elasticità e servizio)
  - Avere una fornitura automatizzata delle risorse permette, dopo il monitoraggio delle stesse, di assegnarle senza alcuna interazione umana e di poter scalare le risorse stesse per ottenerne una ottimizzazione.
- **virtualizzazione delle risorse**.
  - consiste nell'avere risorse virtuali e non più fisiche, consentendo la loro condivisione tra diverse applicazioni.

Architettura del cloud computing



Modelli di servizio

- **SAAS** (Software as a service) (Software delivery model)
  - Il servizio non risiede sulla macchina fisica del consumatore
  - I consumatori non gestiscono hardware o software a parte delle possibili personalizzazioni a lui dedicate (parametri visuali)
  - I servizi sono accessibili attraverso interfacce leggere da qualsiasi dispositivo utilizzando un browser
  - I consumatori usano il servizio a richiesta
  - Scalabilità istantanea
- **PAAS** (Platform as a service) (Platform delivery model)
  - I servizi forniti dal fornitore o acquistati da terzi sono distribuiti su delle piattaforme software in remoto, utilizzando linguaggi, librerie e servizi supportati dal fornitore
  - I consumatori non hanno accesso alla configurazione del sistema, ma solo a quella delle sue applicazioni e dell'ambiente che le ospita
  - Il fornitore deve stimare la richiesta dei servizi e gestire la piattaforma
- **IAAS** (Infrastructure as a Service)
  - I consumatori possono acquisire risorse hardware in remoto su cui hanno il pieno controllo (non sulla struttura cloud, ma solo su quelle acquisite)

Modelli di utilizzo delle risorse (tipi di cloud)

- **Cloud Pubblico**

- Disponibile a chiunque
- Il fornitore del servizio non è gestito dai consumatori
- Accesso con sottoscrizione (pagamento in base all'utilizzo)

- **Cloud Privato**
  - infrastruttura cloud è fornita ad uso esclusivo ad una singola organizzazione con più consumatori,
  - il fornitore del servizio è gestito dal consumatore che può possederlo, dirigerlo e gestirlo
- **Cloud Comunitario**
  - l'infrastruttura è fornita ad una comunità di consumatori
- **Cloud Ibrido**
  - composizione di due o più infrastrutture cloud anche di diverso tipo che rimangono distinte ma unite attraverso tecnologie per la portabilità di dati e applicazioni
  - cloud privati resi disponibili a terze parti

Suddivisione dell'architettura cloud:

- **Cloud service consumer: (consumatore)** Interfaccia per cliente che contiene i servizi e l'eventuale IT in casa del cliente
- **Cloud service developer: (sviluppatore)**Interfaccia per sviluppatori con gli strumenti per lo sviluppo dei servizi
- **Cloud service provider: (fornitore)** Interfaccia per i fornitori che, oltre a fare da collegamento tra sviluppatori e clienti contiene i servizi (siano essi Saas,Paas o Iaas) mostrati ai clienti, la virtualizzazione dei server e le piattaforme di gestione cloud, con due sottolivelli

Problemi del cloud computing

- **Sicurezza e privacy**
  - **Elementi fondamentali del cloud computing**
  - Dati e applicazioni sono salvati in posti all'infuori del controllo del consumatore/azienda quindi:
    - Maggiore rischio di accessi non autorizzati (hackers) ed esposizione dati
    - I requisiti di sicurezza per l'autenticazione, autorizzazione, disponibilità, gestione delle identità, audit, risposta agli incidenti e politiche diventano sempre più importanti
    - **Il cloud deve garantire un alto livello di trasparenza nelle operazioni**
- **Prestazioni e disponibilità (Quality of Service)**
  - Più utenti utilizzano la stessa macchina (più utenti più prestazioni necessarie) (multi-tenancy)
  - I servizi dovrebbero essere SEMPRE disponibili perché una mancanza può causare perdite di denaro
    - Risolvo utilizzando più macchine (sia per aumentare le prestazioni che la ridondanza in modo da sostituire le macchine malfunzionanti)
- **Mancanza di standard**
  - Anche se il cloud ignora dove sia salvato fisicamente, le macchine possono essere completamente diverse e quindi hanno standard diversi e hanno difficoltà a comunicare
- **Eredità (legacy)** (problemi per i nuovi componenti hardware del cloud che devono adattarsi)
- **Basso livello di personalizzazione** (perché tutti hanno lo stesso servizio)
- **Problemi legali (responsabilità)**
  - Problemi di proprietà dei dati
  - Siccome il cloud astrae dalla posizione fisica delle macchine, è difficile capire chi o cosa ha causato il problema

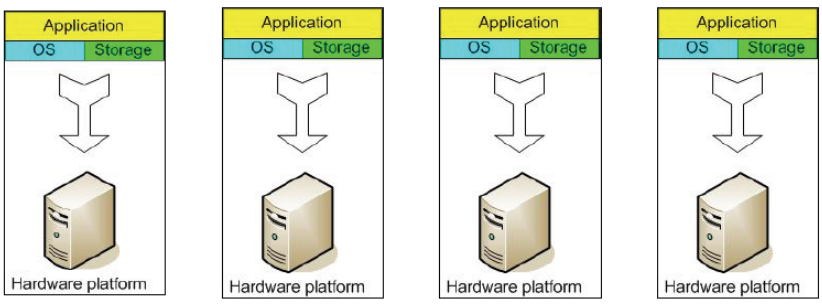
VIRTUALIZZAZIONE

VIRTUALIZZAZIONE: framework o metodologia di divisione delle risorse di un computer in più ambienti di esecuzione, applicando uno o più concetti o tecnologie come:

- Partizionamento hardware e software
- Condivisione del tempo (time-sharing)
- Simulazione totale e completa della macchina, emulazione
- Quality of Service
- Ecc...

CONCETTO TRADIZIONALE DI SERVER (pre-cloud):

- molti server, ognuno gestisce un servizio diverso



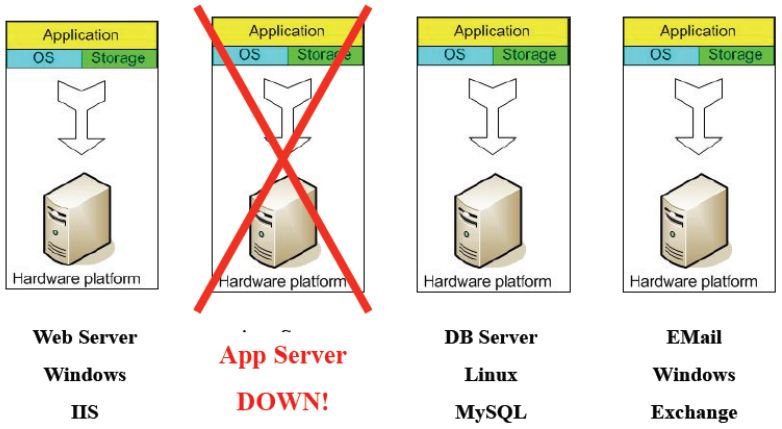
Web Server **Windows** **IIS**

App Server **Linux** **Glassfish**

DB Server **Linux** **MySQL**

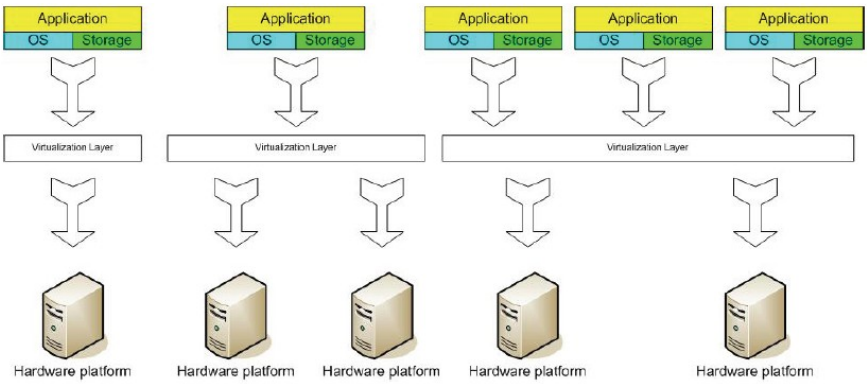
EMail **Windows** **Exchange**

- ogni macchina ha installato il suo S.O., ha la sua interfaccia e i servizi che offre
- **Vantaggi**
  - Facile da concettualizzare (perché ho un applicazione per macchina)
  - Quasi del tutto facile da costruire
    - Sistema nuovo: compro la macchina, installo il software
    - Nuova macchina in un sistema con S.O. diverso
  - Backup facili
  - Virtualmente ogni applicazione/programma può essere eseguita (no limitazioni)
- **Svantaggi**
  - Costi per acquisizione e manutenzione dell'hardware
    - Bisogna comprare una macchina per servizio
    - in caso di S.O. diversi servono tecnici specializzati in entrambi i S.O.
  - non molto scalabile
  - difficile da replicare
  - la ridondanza è difficile da implementare (servirebbero macchine fisiche uguali)
  - vulnerabile a interruzioni di hardware (se una macchina si rompe, perdo il servizio fornito dalla macchina)
  - in molti casi, il processore è poco utilizzato (perché i processori moderni sono troppo evoluti)



NUOVO CONCETTO DI SERVER(virtualizzato):

- molte applicazioni
- strato di virtualizzazione fra applicazioni software e macchine hardware
- numero variabile di macchine che possono essere di più o di meno del numero dei servizi forniti
- **posso aggiungere e/o rimuovere macchine senza causare problemi** (se l'ambiente è costruito correttamente)



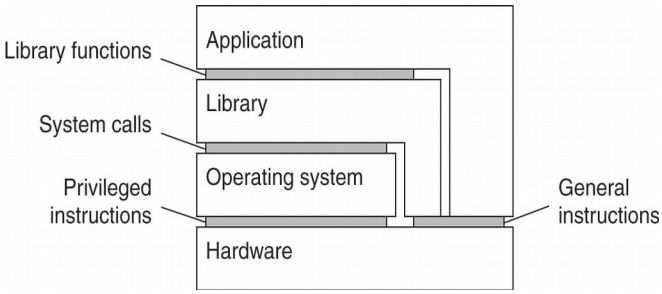
Virtual Machine Monitor (VMM) layer between *Guest OS* and hardware

- i server virtuali possono ancora essere referenziati dalle loro funzioni (server email, server database...)
- possono essere creati dei template di server nell'ambiente virtuale per poter creare più server virtuali identici
- i server virtuali possono essere migrati da un host all'altro quasi immediatamente
- **la virtualizzazione può:**
  - tradurre fra strutture equivalenti (architettura del set d'istruzioni, librerie, chiamate di sistema...)
  - cambiare il livello di astrazione (garbage collection, funzioni virtuali, strumenti di prestazioni, tool di debug...)
  - multiplex/demultiplex delle risorse (nascondi il loro numero fisico)
- **Vantaggi**
  - Pool di risorse
    - **separazione hardware da software**,
    - visione uniforme dell'hardware sottostante
  - Alta ridondanza
  - Alta disponibilità
  - Rapido piazzamento di nuovi server
  - Facile da piazzare
  - Riconfigurabile quando i servizi sono in esecuzione
    - Grazie al completo incapsulamento dello stato dei programmi della macchina virtuale
      - Il bilanciamento del carico di lavoro tra le macchine virtuali non chiaro
      - Serve un modello robusto per gestire i guasti e la scalabilità
  - Ottimizzazione risorse fisiche facendo di più usando meno

- (esempio: macchina che può gestire 100 richiesteuso 1 macchina anziché 100 macchine da 1 richiesta)
- Ottenuto dalla totale mediazione delle interazioni fra le macchine virtuali e l'hardware
  - Aumento sicurezza e affidabilità (Esempio: se un'applicazione crasha su una macchina virtuale, le altre non sono influenzate perché sono isolate, stessa cosa per gli attacchi hacker, limitati solo alla macchina virtuale vittima)

Livelli d'interfaccia nella virtualizzazione

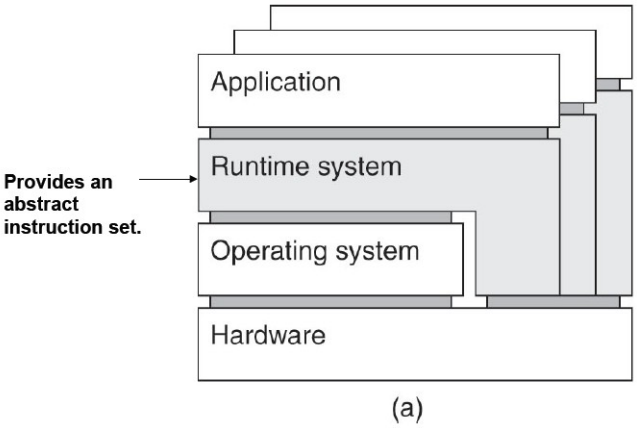
- **Istruzioni macchina non privilegiate:** interfaccia diretta all'hardware disponibile al programma
- **Istruzioni privilegiate:** interfaccia all'hardware per il S.O.
- **Chiamate di sistema:** interfaccia al S.O. per le applicazioni
- **API:** interfaccia del S.O. tramite chiamate di funzioni (di solito usate per nascondere le chiamate di sistema)



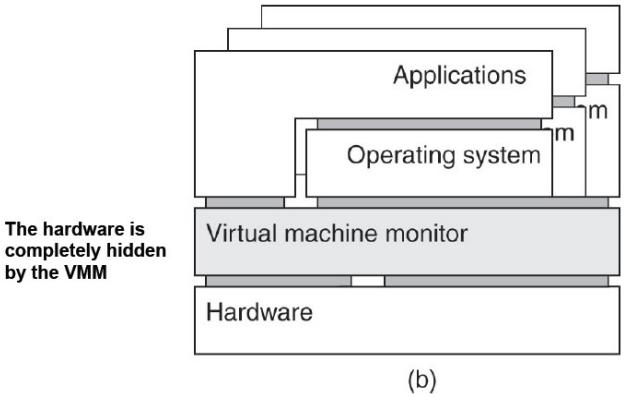
Tipi di virtualizzazione

- **Process Virtual Machine:** virtualizzazione attraverso interpretazione o emulazione
  - Fatto essenzialmente per un processo
  - Fornisce un set astratto di istruzioni macchina
  - Programmi compilati in codice "macchina" che è interpretato (Java) o emulato (Windows)
- **Virtual Machine Monitor:** permette di fornire una macchina virtuale a più programmi diversi simultaneamente (come se ci fossero più CPU in esecuzione su una singola piattaforma) (esempio: VMware e Xen)
  - Particolarmente importante per i sistemi distribuiti perché le applicazioni sono isolate e gli errori sono ristretti a una singola macchina virtuale

Architettura delle macchine virtuali



(a) A process virtual machine, with multiple instances of (application, runtime) combinations.



(b) A virtual machine monitor, with multiple instances of (applications/operating system) combinations.

Obbiettivi principali di sviluppo da rispettare

- **Compatibilità**
- **Semplicità**
- **Prestazioni**

Virtualizzazione CPU

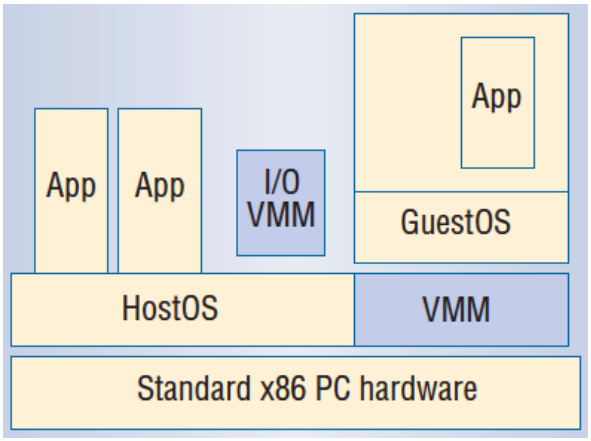
- Un'architettura CPU è virtualizzabile se supporta la tecnica base delle macchine virtuali dell'esecuzione diretta
  - **Esecuzione diretta:** la macchina virtuale è eseguita sulla macchina vera e la macchina virtuale ha il diretto controllo sulla CPU
- **Come si svolge:**
  - Il codice non privilegiato e privilegiato della macchina virtuale viene svolto nella modalità non privilegiata della CPU, mentre il gestore delle macchine virtuali (VMM) è in modalità privilegiata
  - Quando la macchina virtuale tenta di eseguire un'operazione privilegiata, la CPU la blocca nel VMM, il quale emula l'operazione privilegiata sullo stato della macchina virtuale che il VMM gestisce
  - Serve della semantica trappola che permette al VMM di usare direttamente la CPU per eseguire con sicurezza e trasparenza la macchina virtuale
- La virtualizzazione della CPU serve per una maggiore compatibilità (esempio: eseguire Mac OS su windows)
- **Tecniche virtualizzazione**
  - **Paravirtualizzazione:** il VMM definisce l'interfaccia della macchina virtuale sostituendo la porzioni non virtualizzabili del set di istruzioni originale con delle equivalenti e più efficienti versioni virtualizzabili

    - I S.O. devono essere portati per essere eseguiti sulla macchina virtuale
    - La maggioranza delle applicazioni comuni funziona invariata
- **Esecuzione diretta combinata con traduzione binaria veloce:** (es. VMware): le modalità del processore che eseguono normali programmi sono virtualizzate (quindi si può usar l'esecuzione diretta), mentre un traduttore binario può eseguire le modalità privilegiate non virtualizzabili, patchando le istruzioni non virtualizzabili
    - Macchine virtuali ad alte prestazioni che corrispondono all'hardware in modo da massimizzare la compatibilità software
    - Alto controllo sul codice tradotto per ridurre l'overhead della virtualizzazione

Virtualizzazione della memoria

- Il VMM mantiene un'ombra della struttura dati che gestisce la memoria della macchina virtuale, chiamata shadow page table, permettendo al VMM di controllare precisamente quali pagine della memoria sono disponibili alla macchina virtuale
    - **Problemi:**
        - **Paging**
        - **Codice condiviso** (spreco di memoria perché si memorizzano troppe copie del codice)

Virtualizzazione dell'I/O



- Per gestire tutti i dispositivi I/O, il VMM utilizza un processo ombra (I/O VMM in figura) che astrae dall'implementazione fisica dei dispositivi alla quale viene passata ogni richiesta I/O che poi viene gestita dal S.O. ospitante
    - **Vantaggi:**
        - Il VMM è facile da installare perché può essere installato come una normale applicazione
        - L'architettura ospitante supporta comodamente la diversità dei dispositivi I/O attuali
    - **Svantaggi:**
        - Eccessivio overhead perché ogni richiesta I/O passa per il processo ombra
        - I S.O. moderni, essendo best-effort, non possono gestire la distribuzione delle risorse, quindi i server hanno bisogno di accesso diretto all'hardware

Virtualizzazione dei processi: eseguo un processo sotto il controllo di uno strato di software

Virtualizzazione di sistema: esegue un sistema operativo sotto il controllo di uno strato di software