

# SISTEMI DISTRIBUITI

## SECONDO SEMESTRE 2023

PROF. DE PAOLI, CIAVOTTA

Andrea Falbo - Quack

---

### ***CAPITOLO 1: DEFINIZIONI, CARATTERISTICHE, PROBLEMATICHE***

#### **Definizione di Sistema Distribuito**

Definiamo un *sistema distribuito* come un sistema in cui componenti hardware o software situati nei computer in rete comunicano e coordinano le loro azioni attraverso il passaggio di messaggi.

Possiamo anche definirli come una collezione di elementi computazionali autonomi che appaiono all'utente come unico sistema coerente.

Un *nodo* è un elemento computazionale autonomo come dispositivi hardware o processi software. Devono collaborare. Porta a problemi di sincronizzazione e coordinazione.

Una *collezione di nodi* è un raggruppamento di nodi dove questi possono essere aperti o chiusi. Tutti devono operare allo stesso modo. La caratteristica fondamentale è la *trasparenza di distribuzione* ovvero la proprietà per la quale i dettagli interni della distribuzione sono nascosti agli utenti. Deve operare per partial failures.

Caratteristiche fondamentali dei Sistemi Distribuiti:

- *Gestione della memoria*: Non c'è memoria condivisa, comunicazione via scambio messaggi, non c'è stato globale: ogni componente (nodo, processo) conosce solo il proprio stato e può sondare lo stato degli altri.

- 
- *Gestione dell'esecuzione*: ogni componente è autonomo quindi esecuzione concorrente. Il coordinamento delle attività è importante per definire il comportamento di un sistema/applicazione costituita da più componenti
  - *Gestione del tempo* (temporizzazione): non c'è un clock globale, non c'è possibilità di controllo/scheduling globale, solo coordinamento via scambio messaggi.
  - *Tipi di fallimenti*: indipendenti dei singoli nodi, non c'è fallimento globale

## Architetture software

Un'*architettura software* definisce la struttura del sistema, le interfacce tra i componenti e i protocolli. I sistemi distribuiti possono essere organizzati secondo diversi stili architetturali:

- Modello base: Architetture a strati come sistemi operativi, middleware
- Architetture a livelli come le applicazioni client server
- Architetture basate sugli oggetti come Java-Remote Method Invocation (RMI)
- Architetture centrate sui dati come il Web inteso come file system condiviso
- Architetture basate su eventi come applicazioni Web dinamiche basate su callback (AJAX)

Architetture stratificate: software organizzato in strati dove ogni stato è costruito sullo stato inferiore. Possono essere:

- *Puri* come stack ISO/OSI
- *Mix con solo downcall* come organizzazione SO
- *Mix con downcall e upcall* come le applicazioni web

*DOS*: Distributed Operating System ha come obiettivo quello di nascondere e manipolare risorse hardware. Funzioni generiche

*NOS*: Network Operating System è un sistema operativo connesso per multicomputer eterogenei. Ha come obiettivo quello di offrire servizi locali a clienti remoti.

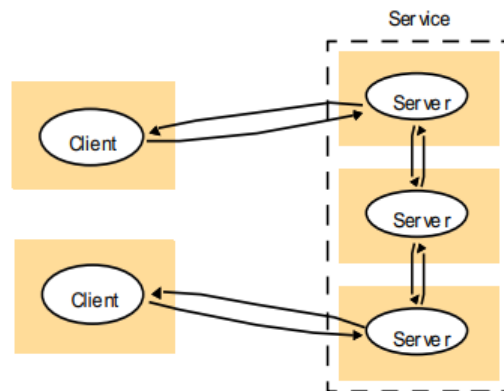
*Middleware*: stato sopra a NOS che implementa servizi a scopo generico; aumenta trasparenza. Servizi

## Il modello client-server

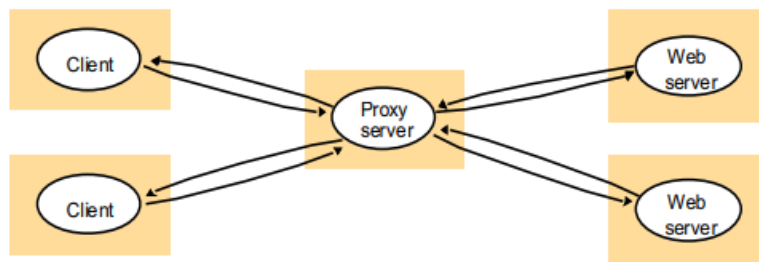
---

Il modello *client-server* è un modello di interazione tra due processi. L'architettura di base prevede che un client acceda ad un server con una richiesta e che il server risponda con un risultato. Abbiamo due configurazioni:

- Accessi a server multipli



- Accesso via proxy



## Proprietà e Caratteristiche Fondamentali

I principali problemi di un sistema distribuito sono:

- *naming*: identificare una controparte
- *access point*: accedere alla controparte, ci serve un riferimento
- *protocollo*: comunicazione, scambio di messaggi, serve un formato
- *semantica*: capire i messaggi

---

Abbiamo diversi gradi di trasparenza:

- Latenze di comunicazione: non possono essere nascoste
- Nascondere guasti dei nodi: è impossibile nascondere
- Completa trasparenza: ha un costo sull'efficienza

L'*Information Hiding* è fondamentale nell'ingegneria del software. Separazione tra quale servizio fornisce e come un servizio è implementato. Quando si parla di *meccanismo* si intende che una certa funzione/servizio è disponibile, quando si parla di *politiche* si intende come una certa funzione/servizio è utilizzata.

*Esempio:* Le applicazioni su TCP/IP si scambiano stream di byte di lunghezza infinita (il meccanismo) che possono essere segmentati in messaggi (la politica) definiti da un protocollo condiviso

Per poter capire le richieste e formulare le risposte i due processi devono concordare un protocollo.

I *protocolli* definiscono il formato, l'ordine di invio e di ricezione dei messaggi tra i dispositivi, il tipo dei dati e le azioni da eseguire quando si riceve un messaggio

## **CAPITOLO 2: STREAM ORIENTED COMMUNICATION**

### **Identificazione dei processi**

Network edge: abbiamo gli *host* che ospitano processi che eseguono le applicazioni. Due modelli: client-server e il peer-to-peer dove abbiamo interazione simmetrica tra host.

Un *processo* è un'entità gestita dal sistema operativo: è un'area di memoria RAM per effettuare operazioni e memorizzare i dati + registro che ricorda la prossima operazione + canali di comunicazione.

---

Ogni processo comunica attraverso *canali* che gestiscono flussi di dati in ingresso e in uscita. Dall'esterno ogni canale è identificato da un numero intero detto "*porta*"

Le *socket* sono particolari canali per la comunicazione tra processi che non condividono memoria. Per potersi connettere o inviare dati ad un processo A, un processo B deve conoscere la macchina (host) che esegue A e la porta cui A è connesso (*well known* port). Le Socket sono API per accedere a TCP e UDP.

Servizio *TCP*:

- Orientato alla connessione: il client invia al server una richiesta di connessione
- Trasporto affidabile (reliable transfer) tra processi mittente e ricevente
- Controllo di flusso (flow control): il mittente rallenta per non sommergere il ricevente
- Controllo della congestione (congestion control): il mittente rallenta quando la rete è sovraccarica
- Non offre garanzie di banda e ritardo minimi

Servizio *UDP*:

- Trasporto non affidabile tra processi mittente e ricevente
- Non offre connessione, affidabilità, controllo di flusso, controllo di congestione, garanzie di ritardo e banda
- Può essere conveniente per le applicazioni che tollerano perdite parziali (es. video e audio) a vantaggio delle prestazioni

Le *politiche* dei servizi UDP sono scomporre il flusso di byte in segmenti e inviarli, uno per volta, ai servizi network .

Le *politiche* dei servizi TCP sono scomporre e inviare come UDP, Ogni segmento viene numerato per garantire riordinamento dei segmenti, controllo duplicati e miss. Utilizza variabili e buffer per realizzare il trasferimento bidirezionale di flussi di bytes ("pipe") tra processi. Prevede ruoli client/server durante la connessione. NON prevede ruoli client/server per la comunicazione. Utilizza i servizi dello strato IP per l'invio dei flussi di bytes.

---

Aspetti *critici*:

- Gestione del ciclo di vita di client e server: Attivazione/terminazione del cliente e del server
- Identificazione e accesso al server: Informazioni che deve conoscere il cliente per accedere al server
- Comunicazione tra client e server: primitive disponibili e le modalità per la comunicazione
- Ripartizione dei compiti tra client e server: Dipende dal tipo di applicazione, influenza le prestazioni in relazione al carico.

Per *identificare* un server possiamo:

- inserire nel codice del client l'indirizzo del server espresso come costante (es. Client Bancario)
- chiedere all'utente l'indirizzo (es. Web browser)
- utilizzare un name server o un repository da cui il client può acquisire le informazioni necessarie (es. DNS)
- adottare un protocollo diverso per l'individuazione del server (es. broadcast per DHCP)

*Risoluzione* dei problemi fondamentali da parte di TCP/IP:

- naming: identificazione a basso livello
- access point: utilizzo di indirizzo IP(host:porta) per accedere ad un processo
- protocollo: stream di bytes
- semantica: non definita

Comunicazione TCP: Il server crea una socket collegata alla well-known port dedicata a ricevere richieste di connessione. Con `accept()` il server crea la socket.

Il prototipo della read è `byteLetti read(socket, buffer, dimBuffer)`

## Le socket in Java

---

Java definisce alcune classi che costituiscono un'interfaccia ad oggetti alle system call illustrate in precedenza. Le principali sono `java.net.Socket` e `java.net.ServerSocket`. Queste classi accorpano funzionalità e mascherano alcuni dettagli con il vantaggio di semplificarne l'uso.

Il client ha una architettura più semplice del server, usa socket ed ha effetti solo su di essa.

Il server a un architettura che prevede che venga creata una socket con porta nota per accettare le richieste di connessione ed entra in un *ciclo* in cui alterna:

- attesa ed accettazione dell'host
- lettura ed esecuzione
- chiusura della connessione

L'affidabilità del server è strettamente dipendente dall'affidabilità della *comunicazione* tra lui e i suoi client

## Architetture del Server

I server possono essere:

- *iterativi*: soddisfano una richiesta alla volta. Al momento di una richiesta di connessione il server crea una socket temporanea per stabilire una connessione diretta con il client. Le eventuali ulteriori richieste per il server verranno accodate alla porta nota per essere successivamente soddisfatte. Vantaggi: Semplice da progettare. Svantaggi: Viene servito un cliente alla volta, gli altri devono attendere, Un client può impedire l'evoluzione di altri client, Non scala.
- *concorrenti processo singolo*: simulano la presenza di un server dedicato
- *concorrenti multi-processo*: creano server dedicati
- *concorrenti multi-thread*: creano thread dedicati

Le operazioni di lettura e scrittura comportano l'uso di system call *bloccanti* ovvero si attende la conclusione dell'operazione richiesta prima di restituire il controllo al chiamante. Per leggere in modo non bloccante serve sapere prima di fare una operazione di lettura o scrittura se il canale è pronto. La system call *select()* ha questo compito

---

Un canale non-bloccante non mette il chiamante in sleep: l'operazione richiesta o viene completata immediatamente o restituisce un risultato che nulla è stato fatto. Solo canali di tipo socket possono essere usati nei due modi.

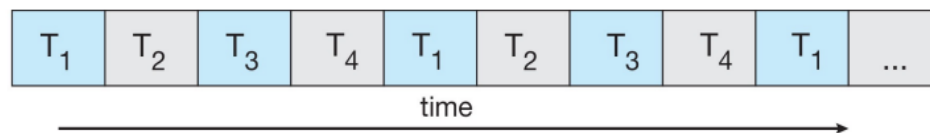
## **CAPITOLO 3: CONCORRENZA E PROGRAMMAZIONE MULTITHREADING**

### **Richiami su concorrenza, parallelismo, sistemi multiprogrammati, processi e thread**

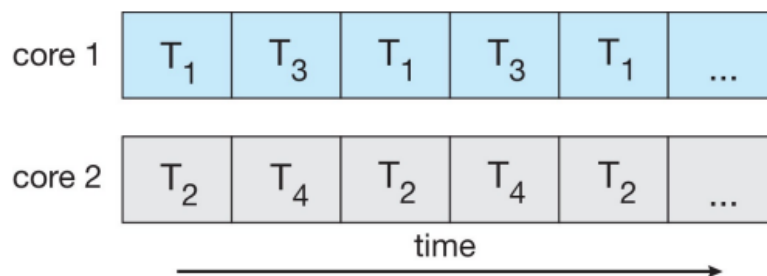
*Concorrenza*: contemporaneità di esecuzione di parti diverse di uno stesso programma.

*Parallelismo*: capacità di eseguire più di un'attività simultaneamente.

- Single core + multiprogrammazione = concorrenza senza parallelismo



- Multicore = concorrenza attraverso il parallelismo



Tipi di parallelismo:

- *Parallelismo dei Dati*: diversi core effettuano la stessa operazione, operando su diversi sottoinsiemi di dati.



- 
- *Parallelismo delle attività*: diversi core effettuano diverse attività

*Legge di Amdahl*: Fornisce il guadagno in termini di performance derivante dall'aggiunta di core ad un'applicazione che ha componenti sia sequenziali che parallele.

$$\text{incremento velocità} \leq 1 / (S + (1 - S / N))$$

Dove S è la porzione di applicazione che deve essere realizzata sequenzialmente, e N è il numero di core.

Con il termine *programmazione concorrente* si indica la pratica di implementare dei programmi che contengano più flussi di esecuzione. Questo si effettua per:

- Per sfruttare gli attuali processori multi-core
- Per evitare di bloccare l'intera esecuzione di un'applicazione a causa dell'attesa del completamento di un'azione di I/O
- Per strutturare in modo più adeguato un programma che interagisce con l'ambiente, controllino diverse attività, gestiscono diversi tipi di eventi, forniscono funzionalità ad utente.

Un sistema operativo esegue un certo numero di programmi sullo stesso sistema di elaborazione. Il numero di programmi da eseguire può essere arbitrariamente elevato, di solito molto maggiore del numero di CPU del sistema. A tale scopo il sistema operativo realizza e mette a disposizione un'astrazione detta *processo*.

Tra gli obiettivi del sistema operativo:

- Massimizzare l'utilizzo della CPU = Mantenere impegnata la (o le) CPU il maggior tempo possibile nell'esecuzione dei programmi
- Dare l'illusione che ogni processo abbia una CPU dedicata. Astrazione utile a chi sviluppa il programma

---

Due tecniche adottate nei sistemi operativi sono la *multiprogrammazione* e il *multitasking*:

- Obiettivo della multiprogrammazione: impedire che un programma che non è in condizione di proseguire l'esecuzione mantenga la CPU
- Obiettivo del multitasking: far sì che un programma interattivo reagisca agli input utente in un tempo accettabile.

*Multiprogrammazione*: Il sistema operativo mantiene in memoria i processi da eseguire. Li carica e gli assegna la memoria e una serie di altre informazioni. Quando una CPU non è impegnata ad eseguire un processo, il sistema operativo seleziona un processo non in esecuzione e gli assegna la CPU. Quando un processo non può proseguire l'esecuzione, la sua CPU viene assegnata ad un altro processo non in esecuzione.

Se i processi sono troppi non possono essere contenuti tutti in memoria: la tecnica dello *swapping* può essere usata per spostare le immagini dentro/fuori dalla memoria.

La *memoria virtuale* è un'ulteriore tecnica che permette di eseguire un processo la cui immagine non è completamente in memoria.

Queste tecniche aumentano il numero di processi che possono essere eseguiti in multiprogrammazione, ossia il *grado di multiprogrammazione*.

*Multitasking*: estensione della multiprogrammazione. La CPU viene sottratta periodicamente al programma in esecuzione ed assegnata ad un altro programma: in questo modo tutti i programmi progrediscono in maniera continuativa nella propria esecuzione.

---

*Creazione di processi:* Di solito nei sistemi operativi i processi sono organizzati in maniera gerarchica. Un processo (padre) può creare altri processi (figli). Questi a loro volta possono essere padri di altri processi figli, creando un albero di processi.

I processi di regola richiedono esplicitamente la propria *terminazione* al sistema operativo. Un processo padre può attendere o meno la terminazione di un figlio e può forzare la terminazione di un figlio.

Un processo è composto da diverse parti:

- Lo stato dei registri del processore che esegue il programma, incluso il program counter
- Lo stato della regione di memoria centrale usata dal programma, o immagine del processo
- Lo stato del processo stesso
- Le risorse del sistema operativo in uso al programma

Durante l'esecuzione, un processo cambia più volte stato. Gli stati possibili di un processo sono:

- *Nuovo (new)*: il processo è creato, ma non ancora ammesso all'esecuzione
- *Pronto (ready)*: il processo può essere eseguito (è in attesa che gli sia assegnata una CPU)
- *In esecuzione (running)*: le sue istruzioni vengono eseguite da qualche CPU
- *In attesa (waiting)*: il processo non può essere eseguito perché è in attesa che si verifichi qualche evento (ad es. il completamento di un'operazione di I/O)
- *Terminato (terminated)*: il processo ha terminato l'esecuzione

Il *Process Control Block* è la struttura dati del kernel che contiene tutte le informazioni relative ad un processo.

---

Lo *scheduler dei processi* sceglie il prossimo processo da eseguire tra quelli in stato ready:

- Ready queue: processi residenti in memoria e in stato ready
- Wait queues: code per i processi che sono residenti in memoria e in stato wait; una coda diversa per ciascun diverso tipo di evento di attesa

I processi possono essere indipendenti o *cooperare*. Un processo coopera se il suo comportamento “influenza” o “è influenzato da” il comportamento di uno o più altri processi. Per permettere ai processi di cooperare il sistema operativo deve mettere a disposizione primitive di *comunicazione inter-processo* (IPC) . Viene eseguito in due modalità:

- Memoria condivisa
- Message passing

IPC tramite *memoria condivisa*: Viene stabilita una zona di memoria condivisa. La comunicazione è controllata dai processi che comunicano. Un problema importante è permettere ai processi che comunicano tramite memoria condivisa di sincronizzarsi.

IPC tramite *message passing*: Permettono ai processi sia di comunicare che di sincronizzarsi, I processi comunicano tra di loro senza condividere memoria. Per comunicare due processi devono: Stabilire un link di comunicazione tra di loro.

*Multithreading*: Fino ad ora abbiamo assunto che un processo abbia un singolo flusso di esecuzione sequenziale. Se supponiamo che un processo possa avvalersi di molti processori virtuali, più istruzioni possono essere eseguite concorrentemente, e quindi il processo può avere più flussi (thread) di esecuzione concorrenti.

---

Un programma sequenziale ha un singolo flusso di controllo.

Un programma multi-threaded ha più flussi di esecuzione.

Un thread termina quando finisce il codice della routine specificata all'atto della creazione del thread stesso, oppure quando, nel codice della routine, chiama la syscall di terminazione

Un **TCB** memorizza il contesto di un thread e le sue informazioni di contabilizzazione.

Thread a livello utente e kernel: I Thread possono essere:

- Thread a **livello utente**: i thread disponibili nello spazio utente dei processi; sono quelli offerti dalle librerie di thread ai processi
- Thread a **livello del kernel**: i thread implementati nativamente dal kernel; sono utilizzati per strutturare il kernel stesso in maniera concorrente.

Possono essere adottate diverse strategie (modelli di multithreading):

- **Molti-a-uno**: i thread a livello utente di un certo processo sono implementati su un solo thread a livello del kernel
- **Uno-a-uno**: ogni thread a livello utente è implementato su un singolo, distinto thread a livello del kernel
- **Molti-a-molti**: i thread a livello utente di un certo processo sono implementati su un insieme di thread a livello del kernel possibilmente inferiore di numero, e l'associazione thread utente / thread kernel è dinamica, stabilita da uno scheduler interno alla libreria di thread

Un **lightweight process** (LWP) è l'interfaccia offerta dal kernel alle librerie dei thread per usare i thread del kernel.

## **Thread in java, modalità di dichiarazione ed esempi base**

**Java threads**: astrazioni offerte della JVM e gestiti dalla stessa.

---

Modalità principali di creazione dei thread in Java:

- Sottoclasse della classe standard `java.lang.Thread`
- Implementazione metodo `run()` interfaccia `java.lang.Runnable`

In Java ogni programma in esecuzione è un thread: Il metodo `main()` è associato al thread "main".

Il modo più semplice per creare ed eseguire un Thread è:

- Estendere la classe `java.lang.Thread`
- Riscrivere (ridefinire, override) il metodo `run()` nella sottoclasse
- Creare un'istanza della sottoclasse
- Richiamare il metodo `start()` su questa istanza: rende il thread `t` pronto (ready) all'esecuzione. È importante notare che non dobbiamo chiamare esplicitamente il metodo `run`, la sua invocazione avviene in maniera automatica

L'ordine con cui ogni thread eseguirà le proprie istruzioni è noto, ma l'ordine globale in cui le istruzioni dei vari thread saranno eseguite effettivamente è indeterminato.

#### *Esempio:* esecuzione Thread

```
public class RunnableExample implements Runnable{
    public void run() {
        System.out.println("Ciao!");
    }
    public static void main(String arg[]){
        RunnableExample re = new RunnableExample();
        Thread t1 = new Thread(re);
        t1.start();
    }
}
```

**Daemon Thread:** processi che eseguono un ciclo infinito di attesa di richieste ed esecuzione delle stesse. Priorità bassa.

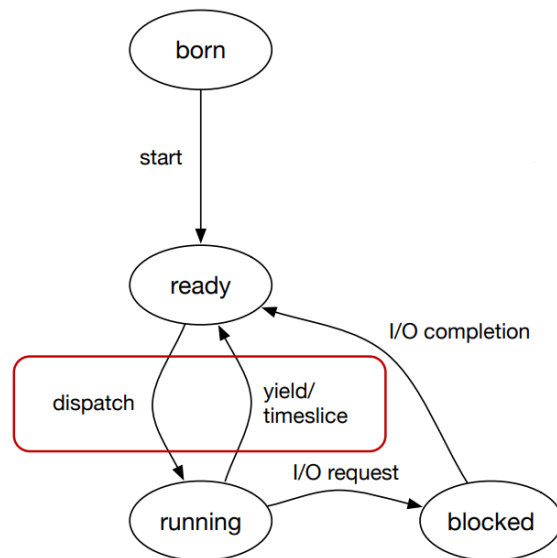
Non si può far partire lo stesso thread (la stessa istanza) più volte. Una seconda chiamata genera l'eccezione `IllegalThreadStateException`.

---

Eccezione: Daemon Thread, come ad esempio il garbage collector.

Stati di un thread:

- Quando invochiamo il metodo `start()` su un thread, il thread non viene eseguito immediatamente, ma passa nello stato di Ready.
- Quando lo scheduler lo seleziona dalla coda, passa nello stato Running, ed esegue il metodo `run()`.
- `yield()` fornisce un meccanismo per informare lo scheduler che il thread corrente è disposto a rinunciare al suo attuale uso del processore ma vorrebbe essere schedulato di nuovo il prima possibile.



Per fermare un Thread in Java si usa il metodo `interrupt()` che setta un flag di interruzione nel thread. Il metodo `interrupt()` non funziona se il thread "interrotto" non esegue mai metodi di attesa.

Il **Threading implicito** è un meccanismo che delega la creazione e gestione dei threads ai compilatori e alle librerie per permettere agli sviluppatori di ragionare in termini di task da compiere. Ce ne sono diversi tipi, vedremo:

- Thread Pools
- Fork-Join

I **thread pools** permettono di creare un certo numero di thread, organizzati in un gruppo, che attendono di rispondere ad una richiesta di lavoro. L'implementazione avviene mediante oggetti generici `Future<>` che permettono di rappresentare dei segnaposto nei quali i risultati dell'operazione saranno resi disponibili.

---

Le **fork-join** sono un metodo che permette di suddividere la task in subtask che verranno poi riunite una volta che tutte sono state risolte.

## Gestione dell'accesso alle risorse condivise

Interazioni tra agenti concorrenti:

- **Cooperazione**: interazioni "prevedibili e desiderate". La loro presenza è necessaria per la logica del programma. Avviene tramite scambio di informazioni (anche semplici come segnali). Sincronizzazione diretta o esplicita
- **Competizione**: gli agenti competono per accedere ad una risorsa condivisa. Politiche di accesso alla risorsa sono necessarie. *Serve mutua esclusione*. Sincronizzazione può essere indiretta o implicita
- **Interferenze**: interazioni "non prevedibili e non desiderate" - errori di programmazione, spesso dipendenti dalle tempistiche (time dependent) e non facilmente riproducibili.

I meccanismi di sincronizzazione permettono di controllare l'ordine relativo delle varie attività dei processi/thread. *Almeno un agente deve aspettare un altro o più.*

Se modello di **memoria condivisa**:

- **mutua esclusione**: dati, regioni critiche del codice, non sono accessibili contemporaneamente a più thread
- **sincronizzazione su condizione**: si sospende l'esecuzione di un thread fino al verificarsi di una opportuna condizione sulle risorse condivise. *Più forte*

Se modello a **scambio di messaggi**:

- di solito impliciti nelle primitive di send e receive: un thread può ricevere un messaggio solo dopo il suo invio.
- **sincronizzazione su eventi**: si sospende l'esecuzione di un thread fino al verificarsi di un evento

**Sincronizzazione su eventi**: Quando si invoca il metodo join() su un thread, il thread



---

chiamante entra in uno stato di attesa (wait). Rimane in tale stato finché il thread chiamato non termina.

Il metodo `join()` può anche ritornare se il thread chiamato viene interrotto.

In questo caso, il metodo lancia una `InterruptedException`.

Se il thread chiamato è già terminato o non è stato avviato, la chiamata al metodo `join()` ritorna immediatamente

**Sincronizzazione su condizione:** I thread vengono messi progressivamente in attesa in attesa che una condizione globale non venga soddisfatta.

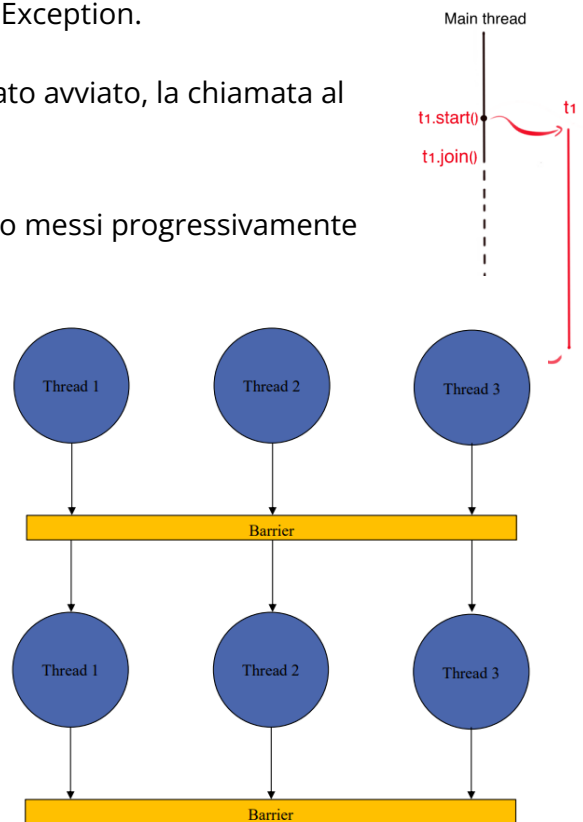
Possibile implementazione: un contatore condiviso conta il numero di processi che devono terminare; il contatore decrementa ogni volta che un thread termina il suo task.

Deve essere aggiornato in maniera *consistente* in quanto è condiviso.

Nei programmi concorrenti molti problemi sono causati dalle cosiddette **Race Conditions**: tutte quelle situazioni in cui thread diversi operano su una risorsa comune, ed in cui il risultato viene a dipendere dall'ordine in cui essi effettuano le loro operazioni.

**Problemi di accesso** a risorse condivise: le azioni che noi eseguiamo non sono atomiche ma sono composte da un insieme molto grande di altre operazioni. Non essendo atomiche, sono interrompibili dallo scheduler e quindi possono inframmezzare l'esecuzione di thread differenti e portare la risorsa condivisa in uno stato inconsistente. Questo problema è detto *inferenza*. Si risolve con accessi mutualmente esclusivi alla risorsa, che la rende logicamente non interrompibile.

La sincronizzazione implica che almeno un thread sia in attesa, abbiamo due possibili implementazioni:



- 
- *busy waiting*: ha senso solo in sistemi multiprocessore, non richiede cambio di contesto, spreca tempo di calcolo sulla CPU
  - *sincronizzazione basata sullo scheduler*: il thread viene risvegliato quando si verifica l'evento atteso

*Protocollo*: ogni thread acceda ad una risorsa condivisa mediante le seguenti operazioni:

1. *Sezione di ingresso* (richiesta entrata nella sezione critica): necessario un'operazione del SO che esegue un'unica operazione macchina (atomica) detta `test_and_set(B)` dove B è una boolean, il metodo ritorna il valore originario della cella di memoria puntata e imposta il valore della cella a true.
2. *Sezione critica*: blocco di codice che può essere eseguito da un solo thread alla volta
3. Eventuale *sezione di uscita* (ad esempio, per notificare gli altri processi)
4. Sezione *non critica*

Il *Lock* è una variabile logica manipolabile atomicamente:

- È una variabile Binaria
- Due metodi `lock()` e `unlock()` permettono di acquisire il lock in maniera atomica
- Una volta che un thread ha acquisito un lock, gli altri thread che richiedono il lock() si bloccano finché il thread che lo detiene non lo rilascia
- Per ogni lock deve essere gestita una coda di thread in attesa

Se vogliamo avere qualcosa che non sia mutualmente esclusivo ma che sia esclusivo solamente per dei gruppi possiamo utilizzare un *Semaforo*. Un semaforo è una variabile intera che può essere manipolata solo attraverso due operazioni atomiche: `acquire()` e `release()`

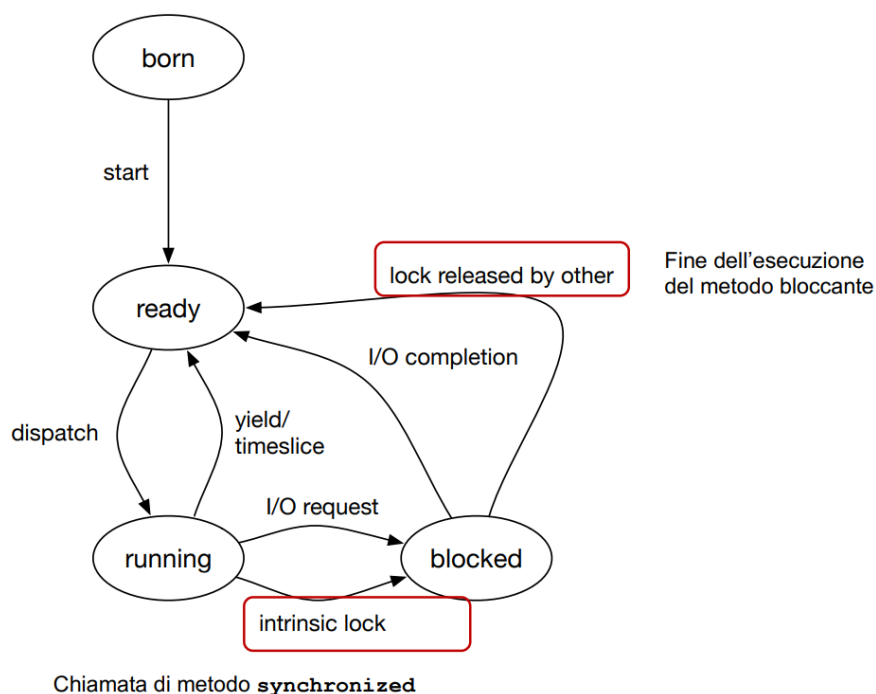
- La prima aspetta che la variabile sia maggiore di zero, e quindi la decrementa
- La seconda incrementa incondizionatamente la variabile
- Per ogni semaforo deve essere gestita una coda di thread in attesa

*Osservazione*: Un lock può essere scritto come un semaforo che può assumere solo i valori 0 e 1

---

L'attivazione di metodi Java può essere resa mutuamente esclusiva tramite la specifica della parola chiave *synchronized* che crea un lock implicito ad ogni oggetto; questo è possibile in quanto derivano dalla classe Object una variabile per il lock.

L'aggiornamento degli stati di un thread con lock è:



Quando un metodo *synchronized* termina, si stabilisce automaticamente una relazione "happens-before" con ogni invocazione successiva di altri metodi *synchronized* sullo stesso oggetto. Ai fini delle problematiche di interferenza, è come se questi metodi fossero stati resi atomici

C'è la possibilità di definire sezioni critiche più piccole di un metodo intero. Si può dire che all'interno di un metodo non sincronizzato c'è una sezione (un blocco formalmente delimitato) che è sincronizzato in relazione a un particolare oggetto (tipicamente *this*).

Una *variabile statica* è una variabile di classe, ovvero ha un valore uguale per tutte le istanze.

Per accedere in modo sincronizzato ai dati statici si deve ottenere il lock su questo oggetto di tipo Class:

- 
- si può dichiarare un metodo come *static synchronized*
  - si può dichiarare un blocco come *synchronized* sull'*oggetto* di tipo *Class*

Il fatto che *synchronized* non faccia parte della segnatura (*firma del metodo*) è molto comodo, perché ci consente di

- Definire classi adatte all'uso sequenziale senza preoccuparci dei problemi della concorrenza
- Poi modificare queste classi derivando sottoclassi che vengono rese adatte all'uso concorrente mediante *synchronized*.

### Precondizioni per metodi *synchronized*

*Monitor*: È una primitiva a più alto livello rispetto a lock e semafori. E' un tipo di dati astratto, le cui variabili interne sono accessibili solo da un insieme di procedure esposte dal monitor. Solo un processo/thread alla volta può essere attivo nel monitor (mutua esclusione). per poter permettere sincronizzazione bisogna definire delle variabili di tipo condizione:

- *x.wait()*: mette in stato di attesa il processo/thread corrente, e lo forza a lasciare il monitor
- *x.signal()*: rende ready un processo/thread che aveva invocato *x.wait()*, se esiste

*Comunicazione asincrona*: invio e ricezione di un messaggio in momenti diversi.

Il problema è assicurare che:

- il produttore non cerchi di inserire nuovi dati quando il *buffer è pieno*
- il consumatore non cerchi di estrarre dati quando il *buffer è vuoto*.

Ci vorrebbe un meccanismo di controllo sulla risorsa che garantisca che certe condizioni siano verificate.

*Esempio*: Abbiamo due metodi: *preleva* e *deposita* che se il buffer è pieno non potranno essere aggiornati (avremmo perdita di informazione). Il thread entra nel metodo e si accorge che non

---

può fare le operazioni. Invece di andare in polling preferiamo usare la **notifica** che può rientrare quando il buffer non sarà più pieno.

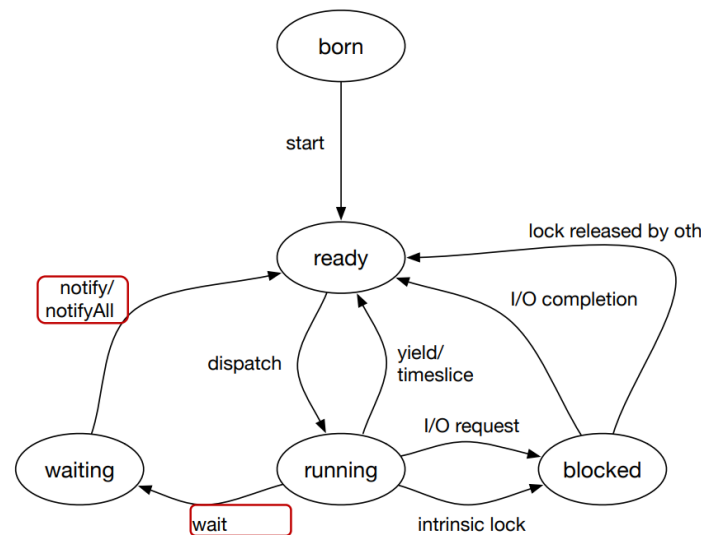
Metodo **wait()**: Rilascia il lock sull'oggetto e sospende il thread fino alla ricezione di un segnale di notifica sull'oggetto stesso da parte di un altro thread.

Metodo **notify()/notifyAll()**: Risveglia un thread (scelta non deterministica) /tutti i thread che si erano messi in attesa volontaria tramite wait. Metodo deve essere synchronized, serve usare il try-catch per i metodi.

Aggiungendo i metodi wait() e notify()/notifyAll() il nostro stato del thread diventa:

- Quando un thread va in attesa (wait) su un oggetto, lo scheduler lo sposta in una "Wait List" associata a quell'oggetto, e rilascia il lock sull'oggetto.
- Quando un thread in attesa viene svegliato (mediante notify), lo scheduler lo sposta nella Ready List.

### Variabili atomiche



Nel package **java.util.concurrent.atomic** sono definite una serie di classi che supportano le operazioni atomiche su singole variabili.

Tipi di atomicità:

- **Reale**: una sola istruzione di CPU viene impiegata per eseguire l'operazione
- **Virtuale**: il thread "crede" di avere accesso atomico alla variabile (synchronized)

Per le applicazioni multi-thread, è necessario garantire due condizioni per un comportamento coerente:

- **Mutua esclusione**: solo un thread esegue una sezione critica alla volta.

- 
- **Visibilità**: le modifiche apportate da un thread ai dati condivisi sono visibili agli altri thread per mantenere la coerenza dei dati.

Due thread sono in esecuzione su due core diversi, perché abbiamo variabili condivise tenute in cache per performance. Code interne in cui accumulano istruzioni e memorie interne per accessi più veloci. Si creano però problematiche di visibilità.

Si può ovviare al problema della visibilità utilizzando variabili **volatile**

Alcune classi della libreria `java.util.concurrent` hanno delle performance superiori alle loro alternative bloccanti perché hanno:

- variabile **atomiche** `java.util.concurrent.atomic`: sono volatili migliorate, e non richiedono lock. mixano due fattori: l'utilizzo di metodi forniti dal SO che sono atomici e implementare un algoritmo di modifica e valutazione delle collisioni di tipo ottimistico non bloccante.
- usano **algoritmi non bloccanti**: Sono thread-safe senza ricorrere a lock, usati per process scheduling, garbage collection, implementazione dei lock, sono più complessi degli equivalenti basati su lock bloccanti

Il processo di locking viene detto **pessimistico**: se la contesa non è frequente, nella maggior parte dei casi la richiesta e l'esecuzione di un lock non è necessaria ed aggiunge overhead:

- Se un thread non riesce ad acquisire un lock viene sospeso
- Context switch, risvegliare un thread presenta un costo
- Un thread in attesa non può eseguire nessuna operazione
- Priority inversion: un thread a bassa priorità può bloccare thread che ne hanno una più alta

Alternative al Locking: **optimistic retrying**: E' più efficiente riprovare che chiedere permesso:

- Nessuna sincronizzazione in lettura
- Per eseguire una scrittura si esegue una lettura della variabile (copia locale), aggiornamento della copia, scrittura della variabile se non c'è collisione, altrimenti riprovare.

---

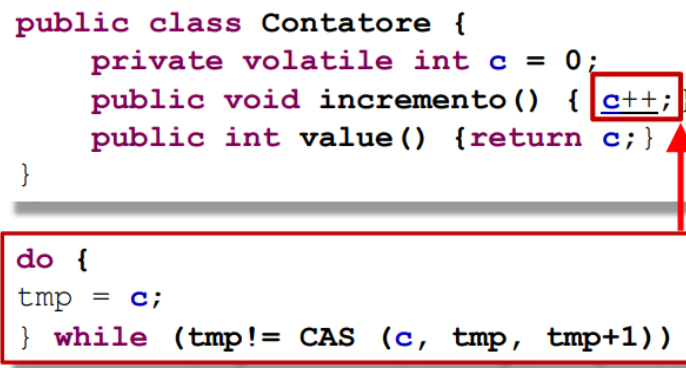
*Compare-and-Swap* (CAS): Prende in ingresso 3 argomenti, 1 posizione di memoria V, 1 valore atteso E, 1 nuovo valore N. Aggiorna atomicamente la posizione V al nuovo valore N, soltanto se il valore presente in V corrisponde al valore atteso E.

*Compare-and-Set*: Come CAS ma restituisce un true se l'operazione si è conclusa con successo, false altrimenti.

Quindi se vogliamo incrementare una variabile intera condivisa c, evitando la race condition ma con meno overhead del blocco synchronized:

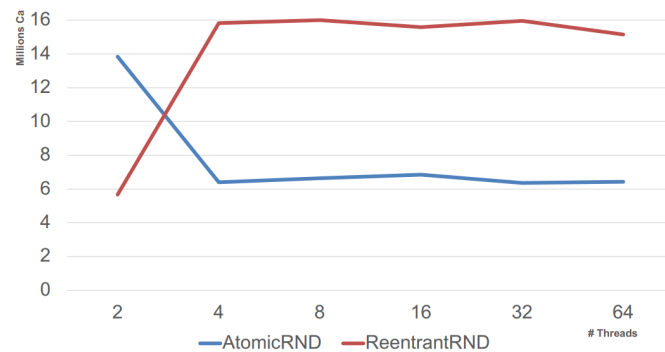
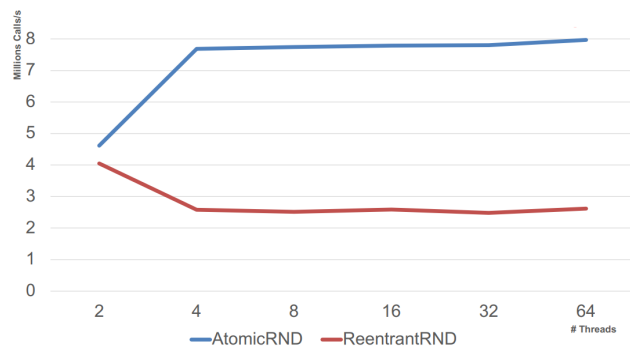
- CAS è una operazione atomica (non può essere interrotta, da 10 a 150 cicli di CPU)
- Dato che il risultato di CAS è visibile (come gli aggiornamenti delle variabili volatile), se non fallisce la variabile risulterà aggiornata per tutti i thread
- Altrimenti l'operazione viene ritentata fino a che l'aggiornamento non ha successo

```
public class Contatore {  
    private volatile int c = 0;  
    public void incremento() { c++; }  
    public int value() { return c; }  
}  
  
do {  
    tmp = c;  
} while (tmp != CAS (c, tmp, tmp+1))
```



Sincronizzazione lock-free: ma c'è il rischio (remoto) di starvation (il thread potrebbe non riuscire mai ad incrementare la variabile)

Se la contesa è elevata, è più efficiente usare un lock, altrimenti si usa atomica.



A Sinistra osserviamo poca contesa e quindi molte più operazioni in atomico. A destra osserviamo molta contesa e, quando si passa a 4 thread, abbiamo performance migliori con lock.

*Esempio:* Immaginiamo di voler mantenere una caratteristica invariante tra due campi:

- Possiamo fare un lock e controllare per ogni aggiornamento che la caratteristica sia mantenuta
- Possiamo farlo in maniera non bloccante usando CAS?

Abbiamo una classe con un valore che contiene il minimo e uno il massimo. In un ambiente multi-thread. Immaginiamo 2 thread uno esegue setLower e l'altro setUpper. Entrambi possono avere successo ed invalidare l'invariante → lower > upper

Trasformiamo l'aggiornamento multiplo in un aggiornamento unico Idea: usiamo la strategia del Optimistic retrying!



---

Per realizzare una versione lock-free e thread-safe della classe `NumberRange`, usiamo una classe helper immutabile che rappresenta una coppia di interi. Classe immutabile, cioè una volta creati gli attributi non possono essere modificati.

In conclusione, le variabili atomiche:

- Sono un sistema di sincronizzazione “leggero” e performante (no context switch, no thread scheduling)
- Sono una generalizzazione delle variabili volatili (garanzie sulla visibilità)
- Permettono operazioni read-modify-write atomiche senza usare lock
- La gestione della contesa è limitata ad una singola variabile
- In generale, le variabili atomiche non offrono il supporto per sequenze atomiche di tipo check-then-act

### Liveness dell'applicazione corrente

Distinguiamo i due concetti:

- *Safety*: nessuno stato scorretto/incoerente è raggiungibile. Garantita dal corretto utilizzo delle risorse.
- *Liveness*: i processi devono progredire nella loro elaborazione

Quest'ultima proprietà non è garantita dal fatto che le componenti concorrenti siano attive, e accedano in maniera mutuamente esclusiva alle cosiddette sezioni critiche di un oggetto.

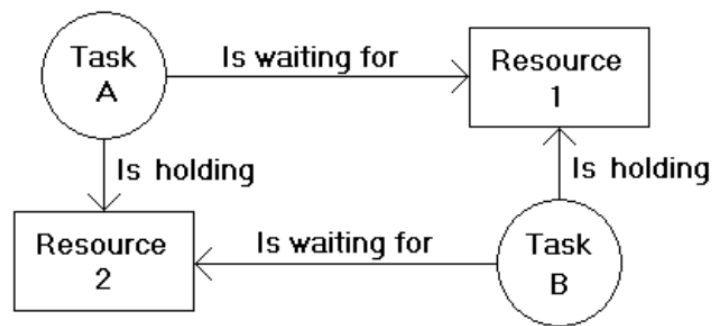
Qui parleremo di tre categorie di problemi di concorrenza:

- *Deadlock*
- *Starvation*
- *Livelock*

Dunque possiamo dire che la Liveness è una proprietà per cui un programma non presenta Deadlock, Starvation e Livelock.

---

In programmazione concorrente, la situazione di *deadlock* si verifica quando ogni membro di un gruppo di agenti (nel nostro caso i thread) è in attesa che qualche altro membro rilasci un lock su di una risorsa. In pratica si tratta di un'attesa circolare destinata a non terminare mai.



Ci sono quattro condizioni necessarie affinché un deadlock si verifichi:

1. *Mutua esclusione*: solo un'attività concorrente per volta può utilizzare una risorsa
2. *Hold and wait*: attività concorrenti che sono in possesso di una risorsa possono richiederne altre senza rilasciare la prima.
3. *No preemption* sulle risorse condivise: una risorsa può essere rilasciata solo volontariamente da un'attività concorrente.
4. *Attesa circolare*: deve esistere una possibile catena circolare di attività concorrenti e di richieste di accesso a risorse concorrenti tale che ogni attività mantiene bloccate delle risorse che contemporaneamente vengono richieste dalle attività successive.

*Osservazione*: Se esiste nel sistema *una sola istanza* per ogni tipo di risorsa allora tali condizioni sono anche *sufficienti* per un deadlock

Esistono alcuni possibili approcci per affrontare le situazioni di Deadlock.

- Deadlock *prevention*: il Deadlock può essere evitato se si fa in modo che almeno una delle quattro condizioni richieste per deadlock non si verifichi mai. Spesso si sacrificano la seconda e la quarta. (estremamente costoso in termini di overhead)

- 
- Deadlock *removal*: non si previene il deadlock, ma lo si risolve quando ci si accorge che è avvenuto.

Esempio di problemi nel Deadlock prevention:

- No hold and wait: non è possibile se devo fare un'elaborazione non interrompibile che coinvolge sia la risorsa a sia la risorsa b
- No attesa circolare: avremo un problema se non esiste un ordinamento assoluto fra le richieste di risorse, né che possa valere per tutti i thread.

*Starvation*: in programmazione concorrente, la situazione di starvation si verifica quando un agente non riesce ad accedere a una risorsa che le viene perpetuamente negata. Questo può essere dovuto alle politiche dello scheduler, a una gestione scorretta della mutua esclusione o una definizione inadeguata dell'algoritmo delle attività.

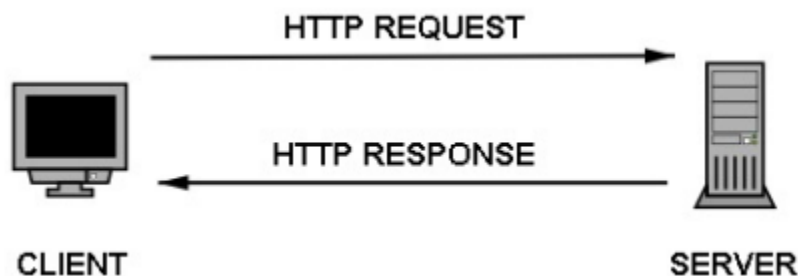
*Livelock*: l'idea è che in una certa situazione i membri di un gruppo di agenti possono non essere bloccati, ma ciò nonostante non progredire effettivamente.

*Esempio*: Due thread che devono "salutarsi" e abbiano due modi di farlo, inchinarsi o stringersi la mano: se l'algoritmo prevede che provino alternativamente un modo o l'altro ciclicamente l'uno o l'altro modo finché l'altro thread non saluta allo stesso modo e se partono da due punti diversi non si troveranno mai ad effettuare lo stesso saluto.

## **CAPITOLO 4: MESSAGE-ORIENTED COMMUNICATION**

### **Introduzione**

*HTTP*: IL web supporta l'interazione tra client e server attraverso il protocollo HTTP:



**Browser:** applicazione web sul lato del client. Si dice **User-Agent**: ovvero è un programma che consente la navigazione nel web da parte di un utente. La funzione di un browser è quella di **interpretare il codice** con cui sono espresse le informazioni e visualizzarle. Una pagina web è costituita da oggetti chiamate **risorse**. La maggior parte delle pagine web sono costituite da un file HTML che definisce la struttura e contenuti della pagina.

**Hypertext:** insieme di testi o pagine leggibili tramite un collegamento.

1. nome protocollo
2. indirizzo host
3. porta del processo
4. percorso dell'host
5. identificatore risorsa

**Dati:** Sono espressi in linguaggi standard:

- ## Protocollo HTTP

28

---

*HTTP*: protocollo a livello applicativo che utilizza il modello client/server. Il client è il browser che chiede, riceve e mostra oggetti web, il server invia oggetti in risposta alle richieste. Il server non mantiene informazioni sulle richieste del client, si definisce stateless. Ci sono due tipi di messaggi: request e response.

*Struttura* request/response:

- *Start-line*: obbligatorio, specifica protocollo. Necessario spazio tra le parti del protocollo. CRLF finale (Carriage-Return Line Feed)
- *Header Lines*: opzionale: formato da coppie nome-valore arbitrarie, sono qualificatori di domanda e risposta. CRLF finale per ogni coppia ed empty line finale per terminare l'header line.
- *Payload*: opzionale, il contenuto vero e proprio che vogliamo mandare.

Start-line:    Part\_1 Space Part\_2 Space Part\_3 CRLF

Header lines: Header\_field\_name:value CRLF

...

Header\_field\_name:value CRLF

CRLF

Payload:    message\_body

*Metodi e applicazione web*: I principali metodi usati sono:

- `get()` : restituisce una rappresentazione di una risorsa. Non ha effetti su server. Viene usato per ottenere informazione in formato HTML, immagini, dati in XML e JSON
- `post()`: comunica dati da elaborare lato server, ogni esecuzione ha un diverso effetto. Non è safe per l'utente.
- `head()` : simile a `get` ma viene restituito solo l'head della pagina web. Viene usato in debugging.

---

*Esempio:* Sono in un sistema bancario e mi interessa il saldo residuo ed effettuare bonifico. Se, per errore, mando la richiesta due volte di leggere il saldo residuo (faccio due get() quindi) rileggo due volte lo stesso valore dunque è safe. Se invece, mando per errore due bonifici (faccio due post() quindi) sto mandando due volte un bonifico, quindi non è safe.

Differenza tra safe e idempotent:

- Safe: i metodi non dovrebbero avere il significato di intraprendere un'azione diversa dal recupero.
- Idempotent: gli effetti collaterali di più richieste identiche sono gli stessi di una singola richiesta. Dunque vogliamo che un oggetto se non c'è si crea, se c'è si aggiorna e non se ne crea uno nuovo.

*Codici di stato response:* ci danno informazioni relative alla response, in quanto non vi è memoria condivisa:"

- 1xx Informational: richiesta ricevuta (101 Switching Protocol)
- 2xx Success : richiesta ricevuta, compiuta, capita, accettata e servita (200 Ok)
- 3xx Redirection: azione aggiuntiva deve essere presa (301 Moved Permanently)
- 4xx Client Error : la richiesta contiene errori non può essere capita (404 Not Found)
- 5xx Server Error : il server fallisce l'adempimento a una richiesta apparentemente valida. (505 HTTP Version Not Supported)

*MIME* (Multipurpose Internet Mail Extension) : qualifica i dati via internet in 5 sottotipi:

1. text: plain, html
2. image: jpeg, gif
3. audio: basic, 32k adpcm
4. application: dati che devono essere processati da un'applicazione prima di essere visibili come ms word, octet-stream
5. video: mpeg, quicktime

*Get condizionale:* non inviare oggetti che il client ha già in cache. Il server manda risposta vuota se l'oggetto in cache è aggiornato

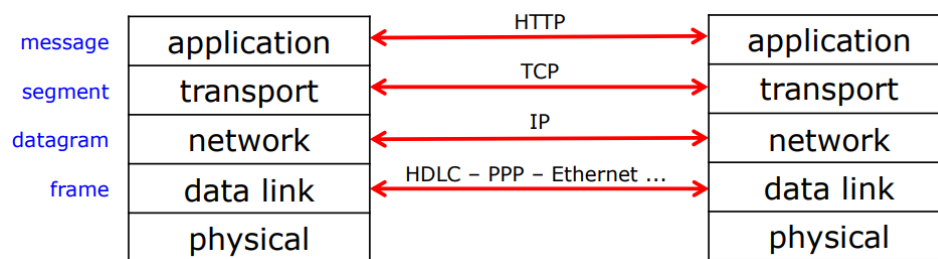
---

**Cookie:** Evita lo stato di stateless, Ha come obiettivo legare più richieste per associare un identificatore alla conversazione. Il Server invia un cookie che l'utente presenta in accessi successivi.

**Autenticazione:** ha come obiettivo controllare l'accesso ai documenti sul server

## Comunicazione a flusso e a messaggio

**Protocollo TCP-IP:**



**Applicazione:**

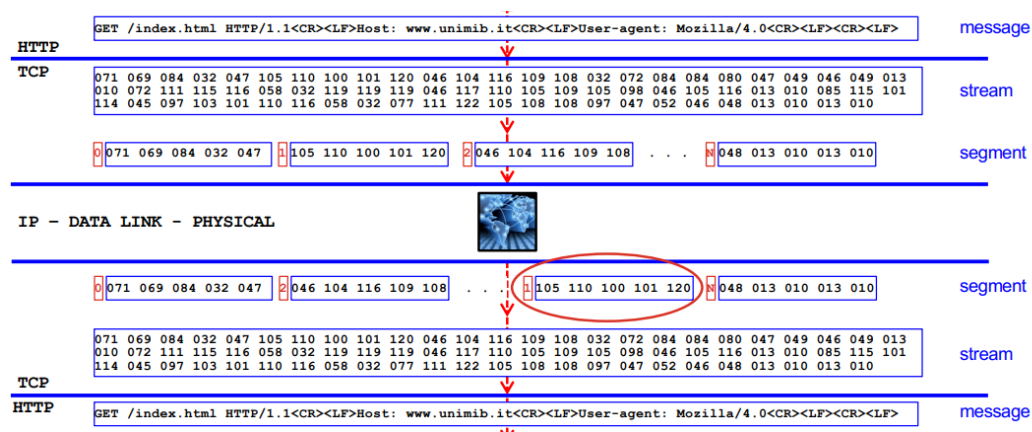
- Invia i messaggi come stream di byte al servizio di trasporto
- Legge lo stream di byte dal servizio di trasporto e ricostruisce i messaggi

Servizio **UDP:**

- Scompone lo stream di byte ricevuto in segmenti
- Invia i segmenti, con una determinata politica, ai servizi network

Servizio **TCP:**

- Scompone e invia come UDP
- Ogni segmento viene numerato per garantire
- Riordinamento dei segmenti arrivati
- Controllo delle duplicazioni (scarto dei segmenti con ugual numero d'ordine)
- Controllo delle perdite (rinvio dei segmenti mancanti)



## Tipi di comunicazione

**Tipi di connessione:** una connessione può essere:

- **sincrona:** due o più interlocutori sono collegati contemporaneamente.
- **asincrona:** non richiede il collegamento contemporaneo degli interlocutori alla rete di comunicazione.

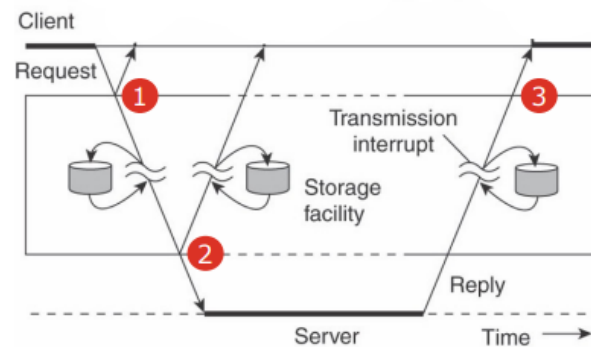
La sincronizzazione può avvenire in 3 punti:

- request submission: Il client invia un messaggio che viene preso in carico dal middleware
- request delivery: viene poi trasferito al middleware ricevente
- request processing: lo passa al server che tratta il messaggio ricevuto. Questa si divide in 3 fasi: 3.1 ricevuto, 3.2 letto, 3.3 processato

I buffer tengono traccia dei messaggi ai vari passaggi.

Osservazione: Si è sincroni o asincroni *rispetto ad un client*.

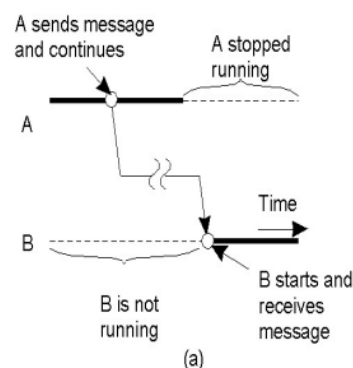




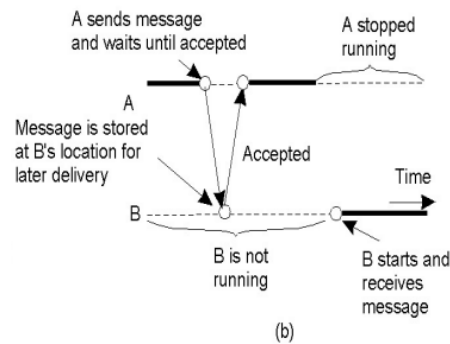
Inoltre può essere:

- *transiente*: il destinatario non è connesso e i dati vengono scartati
- *persistente*: il middleware memorizza i dati fino alla consegna del messaggio al destinatario. Non è necessario che i processi siano in esecuzione prima e dopo l'invio/ricezione dei messaggi

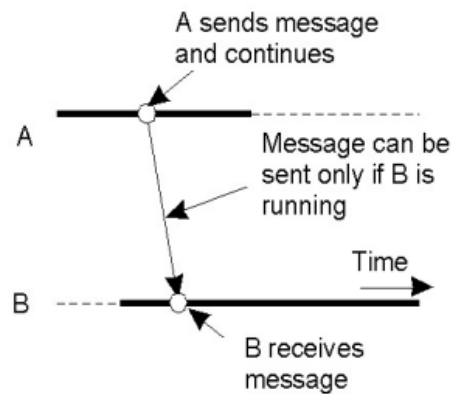
*Comunicazione persistente asincrona*: il middleware mantiene il messaggio fin quando il receiver non si connette e riceve il messaggio; B può anche non essere immediatamente attivo. Ritorna il controllo prima che ogni altra cosa sia successa, quindi A invia e continua.



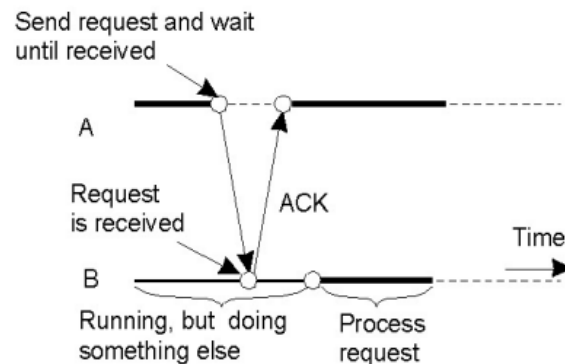
*Comunicazione persistente sincrona*: A attende l'accettazione da parte del middleware di B.



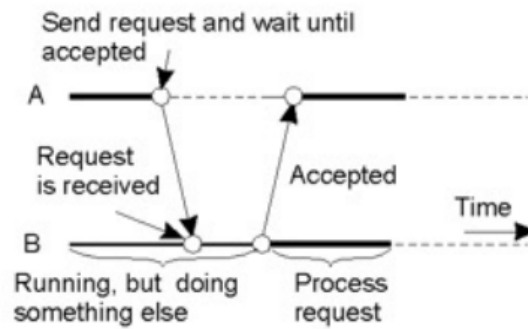
*Comunicazione transiente asincrona*: A può inviare solo se B è in run e pronto a ricevere.



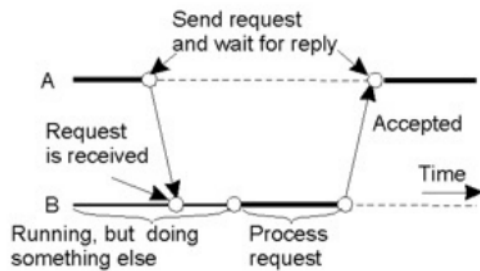
*Comunicazione transiente sincrona basata su ricevuta*: B non processa subito, legge solo inviando un ack, e processa più avanti. Fa diventare persistente la comunicazione, se il sistema non lo fa, è il programma che mantiene il dato non un middleware.



*Comunicazione transiente sincrona basata su consegna*: uguale al modello precedente, ma deve riceverla e poi leggerla anche prima di inviare un accept.



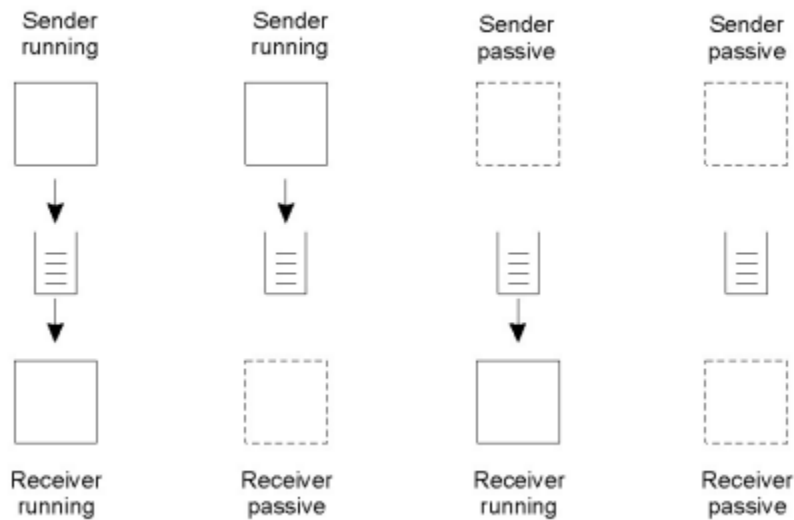
*Comunicazione transiente sincrona basata su risposta:* dopo aver mandato la richiesta aspetta la risposta.



## Comunicazione Persistente

*Message-queueing model:* offrono capacità di archiviazione senza richiedere che il mittente o destinatario siano attivi durante la trasmissione di messaggi. Possiamo averla in 4 modalità:

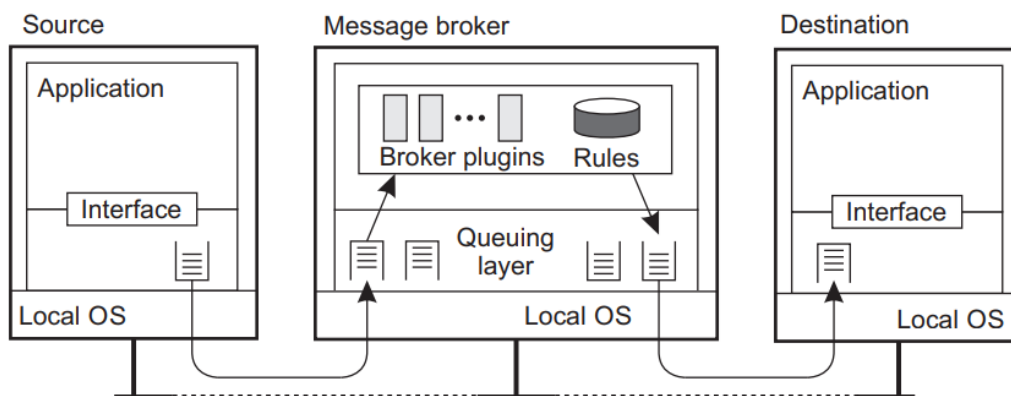
1. Sender running → Receiver running
2. Sender running → Receiver passive
3. Sender passive → Receiver running
4. Sender passive → Receiver passive



*Primitive:* Ci sono 4 primitive:

- *Put*: aggiunge un messaggio ad una coda
- *Get*: blocca fino a quando la coda specificata non è vuota e rimuove il primo messaggio
- *Poll*: controlla una coda specifica e rimuove il primo messaggio senza bloccare
- *Notify*: installa un gestore da chiamare quando un messaggio viene inserito in una coda.

*Message brokers:* usando delle code posso realizzare dei broker che permettono il pattern pubblica e sovrascrive protocolli. Ha il compito di ricevere dati da una sorgente e condividerli a tutti quelli che ascoltano su un dato canale.



---

*PaS Pattern*: Disaccoppiamento tra publisher e subscriber:

- I *publisher* si registrano dichiarando “voglio inviare messaggi su un certo argomento”
- I *subscriber* si registrano dichiarando “voglio ricevere messaggi su un certo argomento”

Il publisher e subscriber non comunicano direttamente: abbiamo disaccoppiamento (quindi indipendenza) e concorrenza (quindi scalabilità). Si possono realizzare comunicazioni N a N e persistenti.

Se nessun subscriber è collegato per un argomento, il broker mantiene i dati secondo diverse politiche.

MQTT: Implementazione più usata di PaS Pattern.

## ***CAPITOLO 5: WEB APP, SERVLET E SERVIZI***

### ***Applicazioni Web***

Le *applicazioni web* sono le applicazioni che usano le funzionalità del web per funzionare. Usano Internet come meccanismo per essere disponibili.

Il *web* è un insieme di applicativi (protocollo HTTP) che funzionano su Internet:

- Permettono di definire un API
- C'è interazione attraverso protocollo HTTP
- L'infrastruttura è completa ed aperta, possiamo creare applicativi di ogni tipo e collegati *dinamicamente* tra loro.

Le applicazioni che vengono eseguite lato client vengono chiamate Web App.

Caratteristiche protocollo *HTTP*:

- scambio di messaggi formati da caratteri, quindi lento e pesante
- usato dal linguaggio HTML per I/O attraverso form, pagine e pagine dinamiche con implementazione di JavaScript
- utilizzo di payload di tipo MIME

- 
- conversazioni tra client e server prive di stato, ovvero senza memoria tra due richieste
  - per creare sessioni di lavoro servono cookie e campi nascosti.

Il cliente è diventato più intelligente perchè oltre a HTML supporta JavaScript, ma anche il Server: adesso è diventato Web Server, in quanto interagisce col cliente con il protocollo HTTP ed esegue un'applicazione tramite un Application Server.

La computazione avviene lato server e può avvenire tramite programmi (*compilati*) o script (*interpretati*):

- Nel primo caso il web server invoca un eseguibile (C++)
- Nel secondo caso il web server ha un engine interno in grado di interpretare il linguaggio: si perde velocità ma si guadagna scrittura dei programmi (Java, Python, Nodejs)

#### *Applicazione Compilata:*

- URL: Uniform Resource Locator: definisce un naming globale
- HTTP: HyperText Transfer Protocol: permette di invocare i programmi sul server come se fossero risorse
- CGI: Protocollo che deve essere gestito dalle applicazioni che permette al server di:
  - attivare un programma
  - passargli le richieste e parametri provenienti dal client
  - recuperare la risposta

*Applicazione Interpretata:* il protocollo CGI viene gestito dall'interprete, quindi le applicazioni non devono più gestire il protocollo. Perdiamo velocità ma è più semplice, portabile e mantenibili.

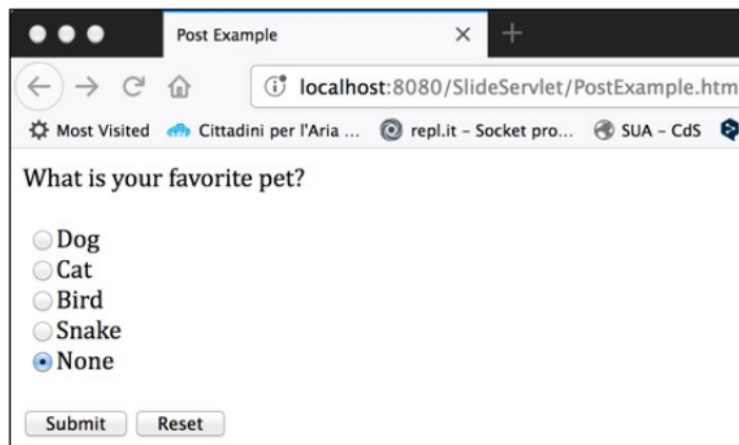
### **Client Side: HTML**

Un *link* in un documento HTML può essere usato per puntare ad una risorsa remota

Anche il *parametro action* di un Form può essere usato per puntare ad una risorsa remota.

---

*Esempio:* Un Form può essere usato per mandare dati al server.



Post Example

localhost:8080/SlideServlet/PostExample.html

What is your favorite pet?

☐ Dog  
☐ Cat  
☐ Bird  
☐ Snake  
☒ None

Submit Reset

Il browser invia una richiesta del tipo:

```
POST /SlideServlet/PostHTTPServlet HTTP/1.1
Content-Length: 11
Content-Type: application/x-www-form-urlencoded
animal=none
```

Dove Content-length e Content-Type sono Headers e animal=none è il payload

## Server Side: Java Servlet

Le *servlet* sono classi Java presenti su un server che devono implementare l'interfaccia: `HTTPServlet`, definisce il set di metodi. Gestito in modo automatico da un container o engine.

Il *container* controlla le servlet in base alle richieste del client attraverso l'interfaccia. mantengono uno stato e consentono interazione con altre servlet.

Mantenere lo stato della conversazione è compito dell'applicazione:

- cookies: informazioni memorizzate livello client che gestiscono sessioni di lavoro

- 
- `HttpSession`: oggetto gestito automaticamente dal container, le servlet vi accedono per memorizzare informazioni

Ogni server implementa l'interfaccia `jakarta.servlet.Servlet` con 5 metodi:

- `void init(ServletConfig config)` Inizializza la servlet, viene invocato dopo la creazione della stessa
- `void destroy()` Chiamata quando la servlet termina (es: per chiudere un file o una connessione con un database)
- `void service(ServletRequest request, ServletResponse response)` Invocato per gestire le richieste dei client (la risposta è nel `response` per questo è `void`)
- `ServletConfig getServletConfig()` Restituisce i parametri di inizializzazione e il `ServletContext` che dà accesso all'ambiente
- `String getServletInfo()` Restituisce informazioni tipo autore e versione

L'interfaccia è solo la dichiarazione dei metodi che, per essere utilizzabili, devono essere implementati in una classe. Sono presenti due classi astratte:

- *`jakarta.servlet.GenericServlet`*: Definisce metodi indipendenti dal protocollo
- *`jakarta.servlet.http.HttpServlet`*: Definisce metodi per l'uso in ambiente web

*`HttpServlet`*: implementa `service()` in modo da invocare metodi per servire le richieste dal web: *Pattern route*. Ha dei metodi chiamati *DoX* che servono per effettuare diverse operazioni:

- `doGet()`: questo metodo è progettato per ottenere il contesto della risposta dalla risorsa Web inviando una quantità limitata di dati di input, questa risposta contiene l'intestazione della risposta, il corpo della risposta.
- `doPost()`: questo metodo è progettato per inviare quantità illimitate di dati insieme alla richiesta alla risorsa web.

*I metodi `doGet()` e `doPost()` devono essere `protected` perché devono essere ereditati.*

Anche i parametri sono stati adattati al protocollo HTTP, cioè consentono di ricevere (inviare) messaggi HTTP leggendo (scrivendo) i dati nell'header e nel body di un messaggio.



---

Interfaccia *HttpServletRequest*:

- Viene passato un oggetto da service
- Contiene la richiesta del client
- Estende ServletRequest

Interfaccia *HttpServletResponse*:

- Viene passato un oggetto da service
- Contiene la risposta destinata al client
- Estende ServletResponse

*Metodi* principali utilizzati:

- String getParameter(String name)
- String[] getParametersValues(String name)
- Enumeration getParameterNames()
- void setContentType(String type)
- ServletOutputStream getOutputStream()
- PrintWriter getWriter()
- Cookie[] getCookies()
- void addCookie(Cookie cookie)
- HttpSession getSession(boolean create)

*Ciclo di vita* di una servlet:

- Una servlet viene creata dal container quando viene effettuata la prima chiamata.
- La servlet viene condivisa da tutti i client
- Ogni richiesta genera un Thread che esegue i vari metodi doX.
- Il container usa init() per inizializzazioni specifiche
- Una servlet viene distrutta dall'engine all'occorrenza di uno dei due eventi:
  - Quando non ci sono thread in esecuzione su quella servlet
  - Quando è scaduto un timeout predefinito
- Container e richieste dei client devono sincronizzarsi sulla terminazione, in quanto alla scadenza del timeout potremmo avere dei thread in esecuzione in service().

- Viene invocato il metodo `destroy()` per terminare correttamente la servlet quando `service()` non ha all'interno dei thread in esecuzione

Init → Service → Do → Destroy

Più richieste vengono servite dalla stessa servlet, questo vuol dire che bisogna far attenzione a come la servlet implementa l'*accesso alle sue risorse*. Spesso sono condivise a livello di database per:

- Utilizzare meno memoria
- Ridurre il costo di gestione
- Abilita la persistenza di risorse condivisibili tra diverse richieste

*Esempio:* Ci sono framework che gestiscono i metodi HTTP per realizzazione di Servlet:

- Spring@MVC dalla versione 2.5 gestisce tutti i metodi HTTP utilizzando delle annotazioni per assegnare i metodi

```
@Controller
public class BookingsController {

    @RequestMapping(value="/bookings", method=RequestMethod.GET)
    public String getBookings() {
        return "someView";
    }

    @RequestMapping(value="/bookings", method=RequestMethod.POST)
    public void addBooking(HttpServletRequest request) {
        // read request
    }
}
```

- JAX-RS: framework REST basato anch'esso su annotazioni

```
@Path("/hotels")
public class HotelsResource {

    @GET
    @ProducesMime( "application/xml")
    public HotelList getHotels(
        @QueryParam("searchString") String searchString) {
        // do something with query parameter
    }

    @Path("/{hotelId}")
    public HotelResource getHotel() {
        return new HotelResource();
    }
}
```

---

## Server Side: JSP

Le *Java Server Pages* (JSP) sono delle tecnologie per la creazione di applicazioni web. Specificano interazione tra container ed un insieme di pagine che presentano informazioni all'utente.

Le pagine sono costituite da tag *tradizionali* (HTML) e tag *applicativi* che controllano la generazione del contenuto (generazione server-side).

Facilitano la separazione tra logica applicativa e presentazione.

Viene incluso in tag delimitati da "<% %>"

La pagina viene convertita in una servlet java la prima volta che viene richiesta.

Elementi:

- **Template** text: le parti statiche della pagina HTML
- **Commenti**: <%-- --%>
- **Direttive**: non influenzano la gestione di una singola richiesta HTTP ma influenzano le proprietà generali della JSP e come questa deve essere tradotta in una servlet. <% %>
  - **page**: liste di attributi/valore, valgono per la pagina in cui sono inseriti
  - **include**: include in compilazione pagine HTML o JSP
  - **taglib**: dichiara tag definiti dall'utente implementando opportune classi
- **Azioni**: permettono di supportare diversi comportamenti della pagina JSP. Vengono processati ad ogni invocazione della pagina JSP. <jsp:XXX attributes> </jsp:XXX>
  - **forward**: determina l'invio della richiesta corrente, eventualmente aggiornata con ulteriori parametri, alla JSP indicata
  - **include**: invia dinamicamente la richiesta ad una data URL e ne include il risultato
  - **useBean**: localizza ed istanzia un javaBean, un oggetto che risponde ad una particolare convenzione, nel contesto specificato

- 
- **Elementi di scripting**: istruzioni nel linguaggio specificato nelle direttive. Devono interagire con oggetti java e altre servlet, oltre a gestire le eccezioni. Può utilizzare anche oggetti impliciti che non devono essere creati
    - **Declaration**: variabili o metodi statici usati nella pagina `<%! declaration [declaration] ... %>`
    - **Expression**: una espressione nel linguaggio di scripting (Java) che viene valutata e sostituita al tag `<%= expression %>`
    - **Scriptlet**: frammenti di codice che controllano la generazione del codice HTML: `<% codice %>`

Gli **oggetti** possono essere creati:

- implicitamente usando le direttive JSP
- esplicitamente con le azioni
- direttamente usando uno script (raro)

Gli oggetti hanno un attributo che ne definisce lo "scope".

**JavaBean**: classe che segue regole precise:

- costruttori senza parametri
- campi privati
- metodi di accesso ai campi sono get/set/is

Lo scope determina la vita e visibilità del bean:

- **page**: scope di default, messo in `pageContext` e acceduto con `getAttribute`
- **request**: messo in `ServletRequest` e acceduto con `getAttribute`
- **session/application**: se non esiste un bean con lo stesso id ne crea uno nuovo

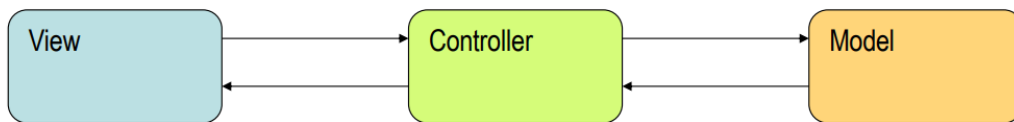
Una **sessione** è una mappa chiave valore che accede ad un oggetto `HTTPSession`: se vogliamo iniziare una sessione, automaticamente viene inviato un cookie al browser del client con le informazioni necessarie, ovvero il `sessionID`.

## Pattern Model View Control

---

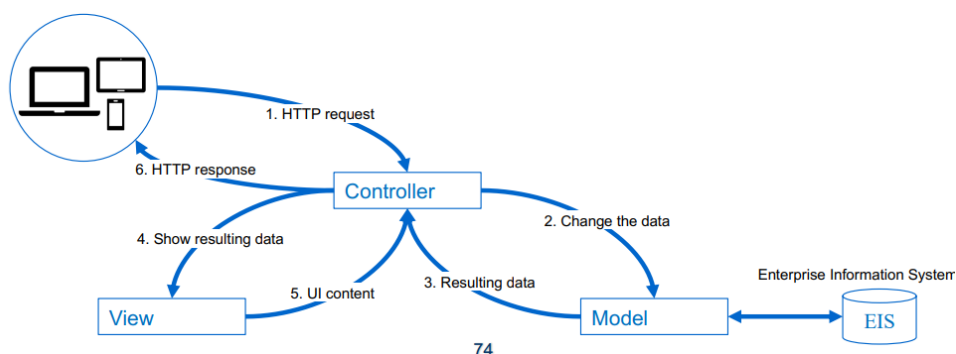
Il **Model-View-Controller** (MVC) è un pattern architetturale che separa data model, user interface, e control logic in tre componenti distinte:

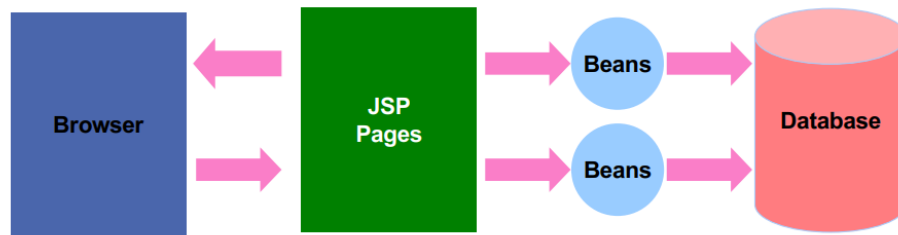
- **Model**: I dati (gli oggetti) trattati dall'applicazione, e le operazioni su di essi
- **View**: La struttura dei dati restituiti al richiedente (e.g., la pagina HTML/CSS)
- **Control**: Definisce le azioni da eseguire a fronte di una richiesta ed interagisce con il Model per modificare i dati e con la View per generare la risposta



Ci sono due tipi di MVC:

- **MVC Standard**: quando siamo nella Vista, e abbiamo visto i dati, dobbiamo mostrare UI Content al Controller per poi poter mandare un HTTP response al client. Gli step eseguiti da MVC Standard sono:
  - Il browser invia una richiesta per la pagina JSP
  - JSP accede a Java Bean e invoca la logica di business
  - Java Bean si connette al database e ottiene/salva i dati
  - La risposta, generata da JSP, viene inviata al browser

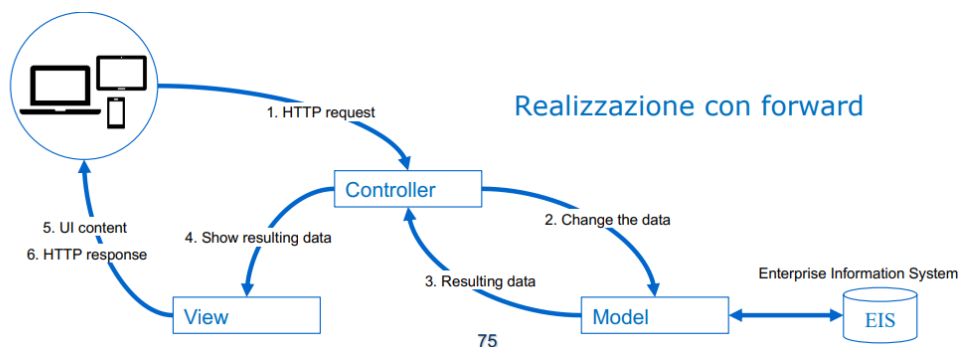


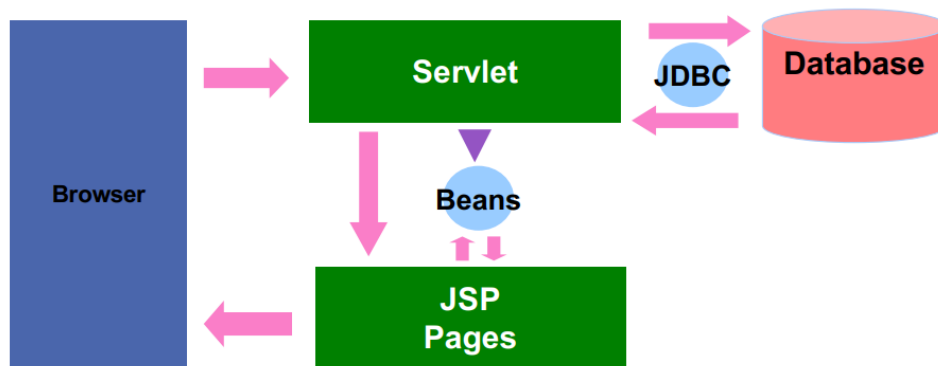


- MVC *Forward*, la Vista è abbastanza intelligente per fare i due passaggi insieme senza dover passare dal Controller. La richiesta viene inviata ad una Java Servlet che:
  - genera i dati dinamici richiesti dall'utente
  - li mette a disposizione della pagina jsp come Java Beans

La Servlet richiama una pagina .jsp, che:

- legge i dati dai beans
- organizza la presentazione in HTML che invia all'utente





#### *Vantaggi:*

- Chiara separazione tra logica di *business* e logica di *presentazione*:
  - poter cambiare la view senza modificare il control e viceversa
  - poter arricchire la view senza appesantire il codice del controller
  - poter definire e scegliere a run-time la view da usare a seconda dell'interazione, dei device utilizzati, dello stato dei dati o delle preferenze dell'utente
- Chiara separazione tra logica di *business* e modello dei *dati* e quindi poter definire diversi mapping tra le azioni degli utenti sul controller e le funzioni sul model,
- Ogni componente ha una *responsabilità* ben definita,
  - poter sviluppare in parallelo,
  - poter mantenere e far evolvere ogni componente in modo indipendente, quindi semplificando la gestione
- Ogni parte può essere affidata a *esperti*:
  - poter assegnare lo sviluppo della view ad esperti di interfaccia e interazione (e.g., pagine jsp o asp)
  - uso tecnologie adatte allo sviluppo delle singole componenti

#### *Svantaggi:*

- Aumento della *complessità* dovuta alla concorrenza (è un sistema distribuito)
- *Inefficienza* nel passaggio dei dati alla view (un elemento in più tra cliente e controller)

---

## Internet of Things

*Internet of Things* è un sistema di oggetti fisici che hanno controparte digitale, con cui è possibile monitorare controllare ed *interagire* attraverso una rete (Internet).

I dispositivi elettronici sono in grado di *servire* il mondo reale.

Le cose diventano servizi ed interagiscono tra loro per aumentare l'intelligenza dell'*ecosistema*.

I sistemi tradizionali distribuiti sono singoli sistemi sviluppati dall'alto verso il basso per decomposizione, client-server e *accoppiamento stretto*: una singola organizzazione possiede il sistema.

I sistemi distribuiti contemporanei includono sottosistemi di terze parti, basandosi sul sistema di *accoppiamento debole*: ogni componente viene mantenuto indipendentemente dagli altri.

Architettura orientata ai servizi (*SOA*) è uno stile architettonico che si concentra su servizi discreti riutilizzabili. I servizi possono chiamarsi tra di loro o essere chiamati da terze parti.

Pro:

- Riutilizzabile
- *Agile* e orientato al processo di sviluppo
- Economia dei servizi
- Scalabilità
- Ottimizzazione e riduzione dei costi

Contro:

- Gestione ingombrante del *ciclo di vita*
- *Dependency hell*
- Integrazione con soluzioni legacy

Ogni servizio dovrebbe:



- 
- Fornire una *descrizione* delle sue *funzionalità* per poter essere *scoperto* e selezionato
  - Fornire l'*accesso* alle sue funzionalità tramite *protocolli* di rete *noti*
  - Sostenere la *composizione* con altri servizi per fornire soluzioni complesse
  - Rispondere alle *esigenze* aziendali e al dominio dei clienti requisiti
  - Garantire un certo livello di Qualità del Servizio (QoS)

Un *servizio Web* è un'entità software indipendente che può essere *scoperta e richiamata* da altri sistemi software su una rete.

I servizi sono componenti indipendenti:

- *Interfaccia* ben nota:
  - Linguaggio di descrizione standard come Web Services Description Language (WSDL)
  - Possibile gestione automatica tramite middleware
- *Punto di accesso unico*
  - Uso dell'URI (URL/URN)
  - Possiamo scoprirlo tramite il significato dei nomi dei servizi (directory UDDI)
- Scambio di dati basato su *documenti*: uso del formato di rappresentazione standard (ad es. XML, JSON)

I componenti sono il servizio e la descrizione del servizio. I ruoli sono 3:

- *Provider*: offre servizi e funzionalità
- *Brokers*: gestisce il catalogo dei servizi
- *Requestors*: vuole usufruire di un servizio ed interagisce con il provider.

SLA (Accordo sul Livello di Servizio) definisce le caratteristiche non funzionali garantite dal servizio. Uno SLA include diversi SLO (Service Level Objectives) che definiscono la qualità del servizio da garantire attraverso metriche specifiche.

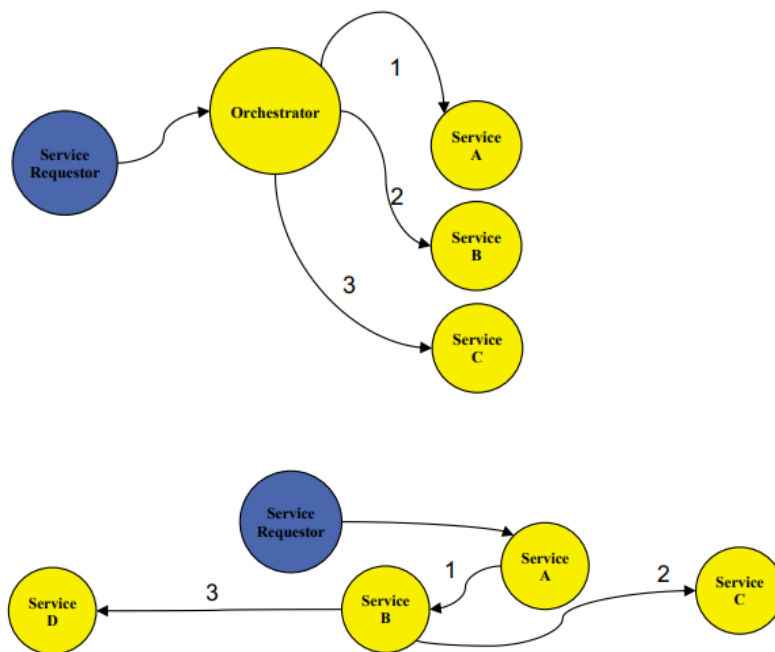
Una composizione è costituita da un insieme di servizi interconnessi, che possono essere utilizzati come un nuovo servizio in altre composizioni. Due servizi sono interconnessi se almeno uno dei due richiede la funzionalità esposta dell'altro.

---

**Processo aziendale:** un insieme di attività correlate eseguite da persone e applicazioni per raggiungere un esito aziendale ben definito.

**Orchestrazione:** descrive come i servizi interagiscono tra loro, inclusa la logica di business e l'ordine di esecuzione delle interazioni dal punto di vista e sotto il controllo di un singolo servizio. Richiede un controllo attivo di un orchestratore. Ingombrante ma più facile da monitorare.

**Coreografia:** descrive la sequenza di interazioni tra più parti coinvolte nel processo dal punto di vista di tutte le parti. Definisce lo stato condiviso delle interazioni tra entità aziendali.



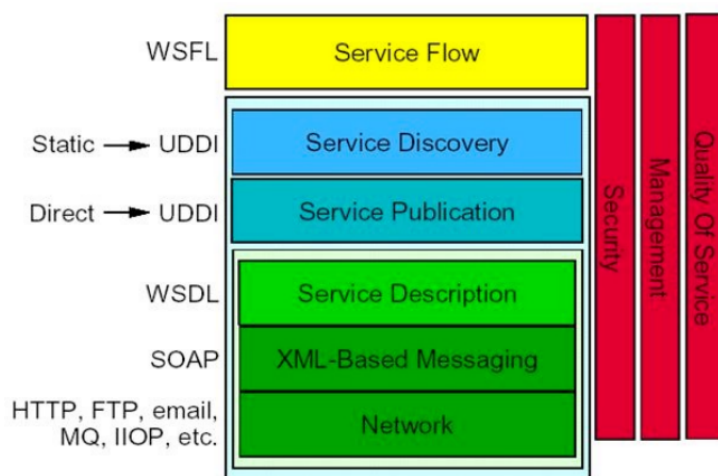
L'Enterprise Service Bus (**ESB**) è un sistema di comunicazione per supportare l'interazione e la comunicazione tra i componenti di un sistema informativo.

- Esempio di approccio coreografico
- Solitamente implementa il modello di publish/subscribe

## SOAP Services

---

## Web Service Stack:



- **Network-Transport:** Rappresenta i protocolli internet utilizzabili in un processo SOAP per far comunicare due servizi
- **Messaging:** SOAP è il linguaggio che definisce la struttura di un messaggio che si può inviare
- **Description:** WSDL descrive il messaggio, quali sono le funzionalità
- **Search and Find:** permette di pubblicarli attraverso dei cataloghi
- **Business Process:** BPEL/WSFL: permette di integrarli in maniera automatizzata

**SOAP** è un protocollo basato su XML che consente ai componenti software e alle applicazioni di comunicare utilizzando messaggi XML.

### SOAP envelope:

- Fa il wrapping del contenuto del messaggio

### SOAP header (opzionale)

- Maggiore flessibilità, può essere elaborato da nodi tra source e dest
- Contiene blocchi di informazioni su come elaborare il messaggio

### SOAP body:

- 
- Messaggio effettivo da consegnare ed elaborare sia per informazioni di richiesta che di risposta

Esistono due modelli per lo scambio di messaggi SOAP via HTTP:

- il modello "SOAP request-response" in cui viene utilizzato il metodo **POST** per portare i messaggi SOAP nel body delle richieste/risposte http;
- il modello " SOAP response" in cui nelle richieste http viene utilizzato il metodo **GET** per ottenere il messaggio SOAP e inserirlo nel body.

**WSDL** è un linguaggio basato su XML per descrivere i servizi Web, i messaggi e come richiamarli. Permette di descrivere quattro dati principali per un servizio:

- Informazioni sull'interfaccia che descrivono tutte le operazioni pubblicamente disponibili di un servizio
- Dichiarazioni del tipo di dati per tutti i messaggi. I tipi complessi possono essere dichiarati (utilizzando SOAP) e utilizzati
- Informazioni di binding sul protocollo di trasporto (HTTP, SMTP, UDP)
- Informazioni sull'indirizzo per localizzare il servizio (URI)

Ci sono servizi *astratti* e *concreti*

**Astratti:**

- Descrizione di un servizio web in termini di messaggi che invia e riceve
- Un'operazione associa *modelli di scambio di messaggi* con uno o più messaggi, definiscono la sequenza e cardinalità dei messaggi scambiati tra servizi
- Un'interfaccia raggruppa queste operazioni in maniera indipendente

**Concreti:**

- I binding specificano il protocollo di trasporto per le interfacce
- Un endpoint associa un URI a un binding
- Un servizio raggruppa gli endpoint che implementano un comune interfaccia

---

WSDL e SOAP sono caduti sotto il loro stesso peso perché erano difficili da capire e complessi. Si è ritenuto necessario un approccio più semplice e leggibile con semantica concordata e utilizzo di JSON.

## REST

In *REST*, torniamo ai principi di *HTTP* eliminando le ridondanze di SOAP e assegnando la semantica ai verbi e *URL*. Formato JSON.

REST si basa su “considerazioni di default”, senza specifica:

1. Concetto di *risorsa* con nome definite da *URI*
2. I *verbi* sono le operazioni che possono essere applicate sulle risorse
3. Il tipo di contenuto definisce le *informazioni* rappresentate
4. abbiamo sempre a che fare solo con risorse, ovvero con dei *dati*

Caratteristiche restful:

1. risorse devono avere identificatore che è parte del pattern (*path parameter*)
2. l'interfaccia ad una risorsa deve essere *uniforme*, con i soliti 4 verbi.
3. architettura client/server
4. *manipolazione* delle risorse avviene lato *client*
5. le risorse *dovrebbero* avere associati dei link
6. le risorse sono *prive di stato*, non sanno chi gli fa richiesta (no sessione attive né stato)
7. messaggi *autodescrittivi*
8. da punto 2, 6,7 ottengo *cache*: spazio di memoria tra client e server. riduce latenza e quantità dati: implemento cache locale e quindi aggiornamento di solo modifiche, richiesta condizionale.

La ricetta di REST:

1. Trovare tutti i *nomi*:
  - a. URI per ogni risorsa.
  - b. URI descrittive, *opache* (non lasciare metadati), persistenti. L'evoluzione avviene tramite *versionamento* delle API.

- 
- c. Evitare query, gerarchia, uso dei verbi, preferire *sostantivi*.
  2. Definire i *formati*:
    - a. evitare di creare rappresentazioni personalizzate
    - b. entrata = uscita
  3. Scegli le *operazioni*
    - a. Spesso le 4 operazioni GET, POST, PUT, DELETE sono sufficienti ma ci sono altri metodi
  4. Evidenziare codici di stato eccezionali

Esiste *overloaded POST* per quando vogliamo usare POST che manipola invece che riceve/da. Si implementa con una chiamata di procedura remota.

Rapporto *asincrono* alle operazioni esponendo uno stato Accettato ma elaborando dopo. La coda diventa risorsa.

*Esempio*: Metodo di pagamento, stampiamo accettato ma in un secondo momento avverrà il pagamento e in un altro secondo pagamento verrà controllato se sono arrivati i soldi del pagamento

Anche lo *stato* diventa una *risorsa*. Se i servizi sono privi di stato possiamo effettuare caricamento del carico in maniera facile e quindi partizionare, scalare, cacheare. Per farlo non devo usare sessioni e quindi cookie o chiavi.

## **CAPITOLO 12: CLOUD COMPUTING**

L'*IoT* è un sistema di oggetti fisici che possono essere scoperti, monitorati, controllati o con cui si può interagire tramite dispositivi elettronici che comunicano su varie interfacce di rete.

*Edge computing*: necessario per limitare il trasferimento di grandi quantità di dati, migliorare la latency dei sistemi, considerare temi di mobilità e località dei sistemi.

Il *cloud computing* è uno stile di computing che fornisce capacità scalabili ed elastiche relative all'IT come servizio a clienti esterni utilizzando tecnologie internet. Servizi:

- Self-service su richiesta: un consumatore può fornire unilateralmente capacità di calcolo senza necessità di interazione umana.

- 
- Ampio accesso alla rete: le capacità sono disponibili in rete e vi si accede attraverso meccanismi standard
  - Pooling delle risorse: le risorse di calcolo del fornitore sono messe in comune per servire più consumatori
  - Rapida elasticità: le capacità possono essere fornite rapidamente ed elasticamente per scalare rapidamente
  - Servizio misurato: l'uso delle risorse può essere monitorato, controllato e riportato

La funzione più importante della *virtualizzazione* è la capacità di eseguire più sistemi operativi e applicazioni contemporaneamente, indipendentemente dalla piattaforma o hardware sottostante:

- aiuta a isolare i guasti causati da errori o problemi di sicurezza;
- permette l'introduzione di nuove capacità di sistema senza aggiungere complessità a hardware e software complessi già esistenti;
- può solitamente migliorare le prestazioni complessive delle applicazioni grazie alla tecnologia che può bilanciare le risorse e fornire solo ciò di cui l'utente ha bisogno

Due tipi di virtualizzazione:

1. *Process Virtual Machine*: virtualizzazione attraverso l'interpretazione o l'emulazione: fornisce un set astratto di istruzioni della macchina; i programmi sono compilati in codice "macchina codice che viene poi interpretato (ad esempio, Java) o emulato (ad esempio, Windows)
2. *Monitoraggio della macchina virtuale*: in grado di fornire una macchina virtuale a molti programmi diversi simultaneamente.

## **CAPITOLO 6: PROCEDURE-CALL COMMUNICATION**

### **Remote Procedure Call**

Le *procedure remote* sono una estensione al distribuito del normale protocollo di chiamata di procedura.

Vantaggi:

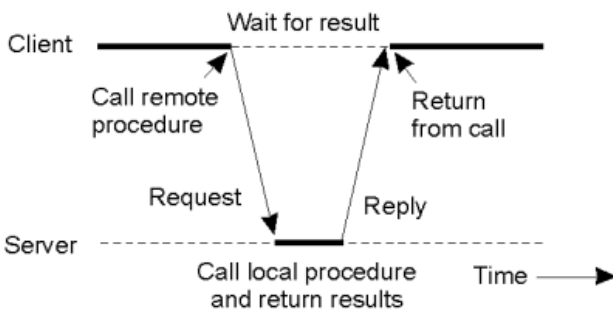
- hanno una semantica nota: chiamata di procedura
- sono facili da implementare: vicine al modello client-server

Svantaggi:

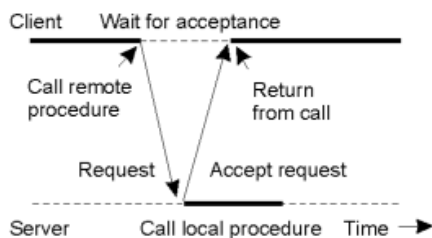
- realizzate dal programmatore: tutto è esplicito
- sono statiche: scritte nel codice dei programmi
- non c'è concorrenza: sono bloccanti

Tutti gli svantaggi sono ovviabili.

*RPC*: cliente effettua chiamata di procedura bloccante e riceve il controllo quando il server ritorna il risultato. Unica differenza con Client-Server è il tempo, infatti abbiamo le frecce.

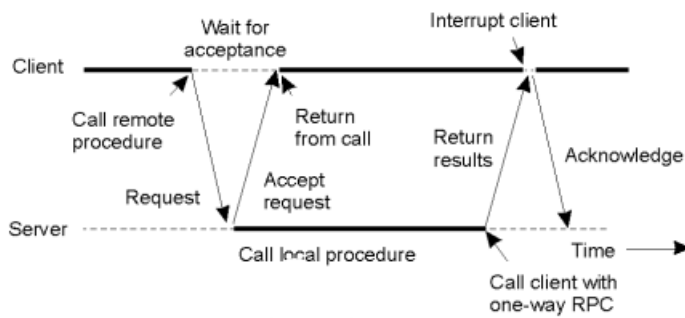


*RPC asincrono senza risposta*: invoco una procedura e restituisce il controllo prima di eseguire.

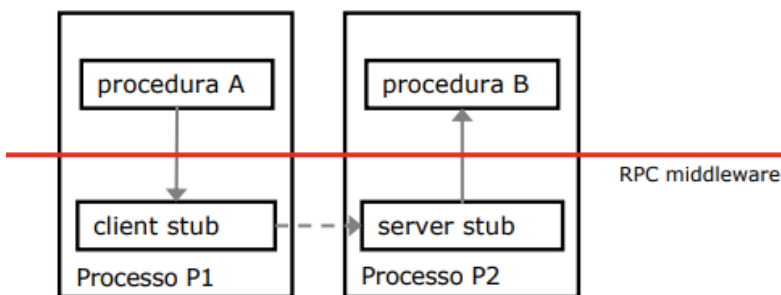


*RPC asincrono con risposta*: come la versione di prima ma quando finisce la task causa un interrupt con call-back

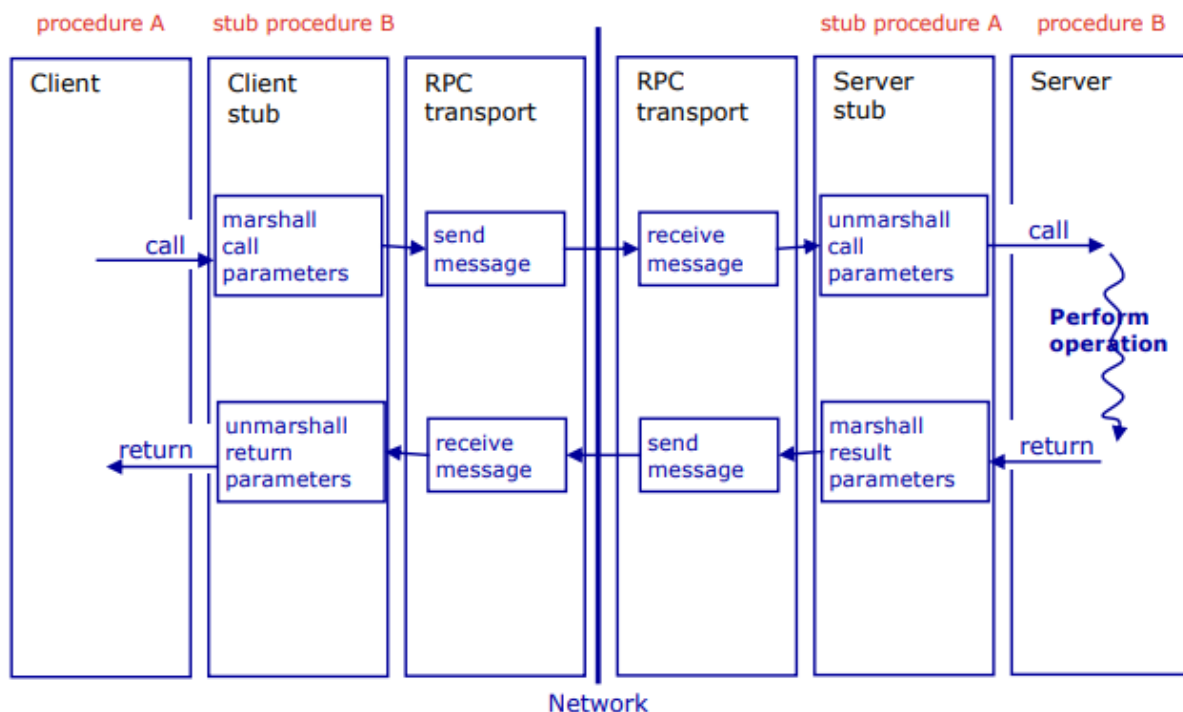




Architettura RPC: utilizzo un client *stub* e un server stub per effettuare la chiamata remota. La comunicazione tra procedure e stub avviene in modalità locale. Il middleware separa gli stub dalle procedure.



Il passaggio dei parametri avviene tramite procedure call remota. Si replica lo stack nel processo remoto: si esegue *marshalling* dei dati, ovvero vengono impacchettati e dopo vengono spaccettati (*unmarshalling*) per creare copie locali.



## Distributed Objects

Il cliente invoca un metodo e passa al *proxy* che funziona come rappresentante dell'oggetto remoto. Questo effettua marshall remoto e passa le informazioni allo *stub* del server, che invocherà lo stesso metodo come oggetto.

Gli oggetti possono essere a *compile-time*, ovvero definiti attraverso interfacce e classi (es. java) oppure a *run-time*, ovvero accessibili attraverso wrappers.

Gli oggetti svolgono le seguenti funzioni:

- Incapsulano lo stato: I valori dei campi o variabili
- Definiscono operazioni sui dati: metodi delle classi
- Definiscono l'accesso: metodi delle interfacce

Un *puntatore* (C++):

- è una variabile che contiene un indirizzo di memoria
- può essere modificato in ogni momento

- 
- non è tipizzato, può riferirsi ad un oggetto o ad altro
  - non può essere distribuito, non possiamo accedere all'area di memoria di un altro processo.

Un *riferimento* (Java):

- è una variabile che contiene informazioni logiche per
- è immutabile, inizializzato alla creazione dell'oggetto
- è strongly typed, la classe dell'oggetto definisce il tipo
- può essere distribuito

## Java RMI (Remote Method Invocation)

*RMI* è un middleware che:

- estende l'approccio OO al distribuito: supporta *l'ereditarietà*
- supporta l'invocazione di metodi tra oggetti su macchine virtuali distinti
- si basa sulla *portabilità* del bytecode e sulla macchina virtuale: più sicuro ed efficiente perché non si deve tradurre nulla

Dunque Java RMI oltre al passaggio di parametri per valore come RPC, permette di passare i parametri per *reference*, definendo degli stub specifici per ogni oggetto.

Le invocazioni sono statiche con interfaccia nota a compile-time oppure dinamiche con informazioni logiche.

I tipi primitivi sono passati per valore. Gli oggetti non remoti vengono passati per valore se *serializzabili*.

Il passaggio per reference permette invocazioni remote:

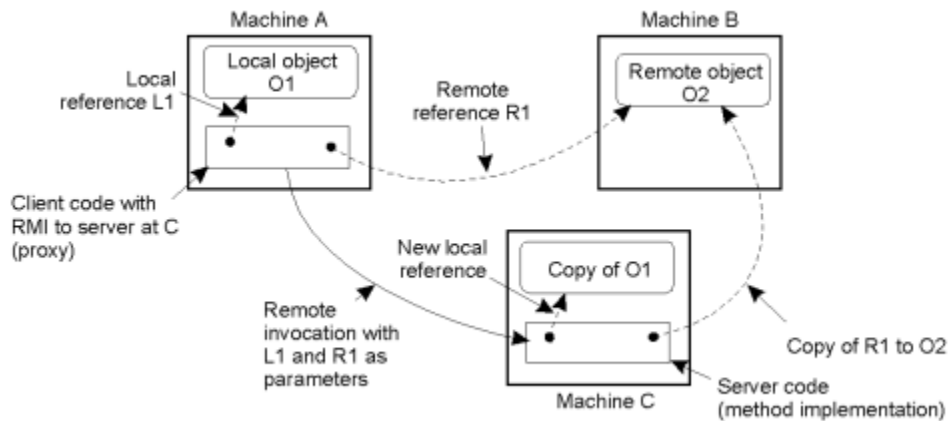
- la classe `java.rmi.server.UnicastRemoteObject` è una reference class che serve allo scopo
- implementa le interfacce `Remote` e `Serializable`

La *serializzazione* rappresenta lo stato di un oggetto come stream di byte. È essenziale per poter memorizzare e ricostruire lo stato degli oggetti:

- per trasferire oggetti via rete
- per definire oggetti persistenti

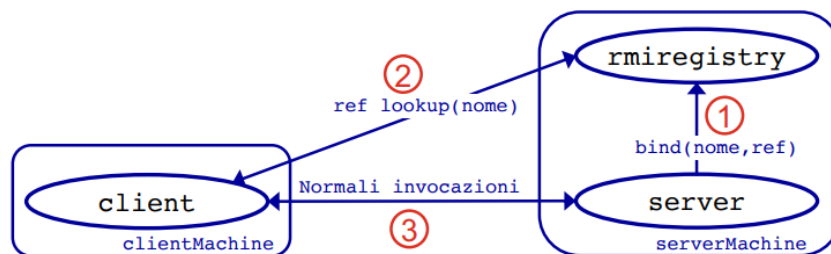
I tipi base sono serializzabili in modo nativo.

Le classi sono serializzabili implementando l'interfaccia `Serializable` ridefinendo i metodi `writeObject` e `readObject`.



Una macchina riesce ad avere la reference remota di un oggetto remoto attraverso:

- Utilizza nomi assegnati dall'utente e un directory service per convertirli in reference operativi
- I directory service devono essere disponibile ad un host e porta nota
- RMI definisce un `rmiregistry`: sta su ogni oggetto remoto (`localhost:1099`) ed attiva listen.



La classe *Naming* dà accesso diretto alle funzionalità del RMI registry:

- I metodi sono static

- 
- I parametri sono stringhe in formato URL riferiti al registry e all'oggetto remoto considerato
  - Default host: localhost, Default port: 1099

I metodi principali della classe Naming sono:

- lookup= restituisce riferimento a oggetto associato al nome specificato
- bind= collega il nome specificato all'oggetto remoto
- rebind= bind + cancella i collegamenti esistenti.
- list = restituisce nomi degli oggetti della registry
- unbind = distrugge collegamento

Java/RMI rispetto ai 4 problemi principali dei sistemi distribuiti:

1. Naming: nome host + nome oggetto
2. Access Point: nome
3. Formato: oggetti
4. Semantica: tipo dei dati

Java definisce un'interfaccia per implementare oggetti remoti: interface: java.rmi.Remote

Per creare una classe remota (server):

1. definire l'interfaccia della classe remota
2. implementare la nuova interfaccia
3. Implementare un server che crei e registri l'oggetto al Registry

Il server

- crea da programma di un oggetto locale che realizza il servizio desiderato
- lo registra presso l'RMI registry sullo stesso host con un nome pubblico

## ***CAPITOLO 7: WEB-APP CLIENT DINAMICI***

### **Multi-Layer vs Multi-Tier**

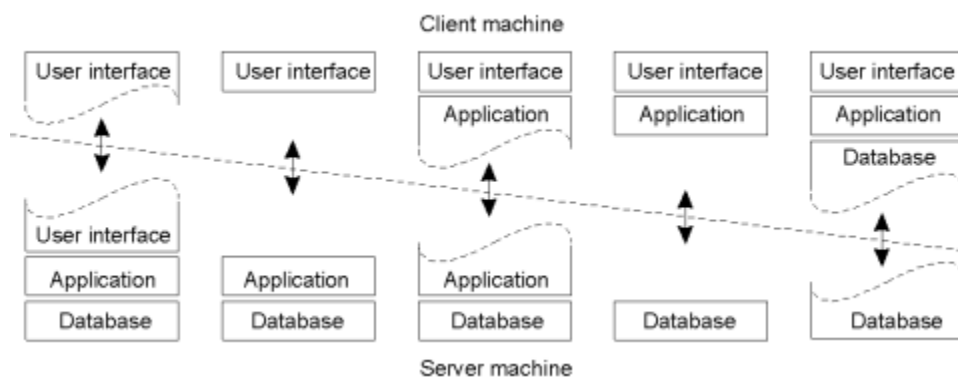
---

I modelli multi-layer avremo ogni layer che si basa sul layer precedente. Es. middleware, comunicazione RPC/RMI, stream-based communication

I modelli multi-tier separano il compito in più moduli. Es. MVC

La nuova idea è far *evolvere i moduli*:

- Lato server tutta la computazione, lato client solo il rendering
- L'applicazione produce lato server e il client ha il compito di visualizzarli
- Database a liv. server, applicativo sia server che client, visualizzazione lato client. vedremo questo
- Caso DBMS, interrogazione database per ottenere i dati
- Lavoro i dati livello client, che sono immagazzinati lato server



## Rich Internet Application

### Caratteristiche Web

- Le pagine Web si ricaricano sempre e non vengono mai aggiornate
- Gli utenti attendono il caricamento dell'intera pagina anche se è necessario un singolo dato
- Restrizioni di richiesta/risposta singola

*Ajax*: Asynchronous JavaScript And XML

- Tecnica di sviluppo per la creazione di applicazioni web interattive
- Non una nuova tecnologia, ma più di un *modello*

- 
- Assiste nelle interfacce utente
  - Meno pagine web leggibili/collegabili dalla macchina

Ajax è composto da:

- HTML e CSS per presentare informazioni
- DOM (Document Object Model): dynamic display and interaction
- XML manipolazione dati (JSON in app moderne)
- XMLHttpRequest object per ricevere dati in modo *asincrono* dal web server
- JavaScript per combinare tutto

*JavaScript* è un linguaggio di scripting a oggetti, non tipizzato.

- Interpretato da un *engine*, viene eseguito dal browser.
- Node.js permette l'esecuzione sul server.

JavaScript permette di rendere le pagine html *dinamiche*, cioè di inserire dei programmi che modificano il comportamento e le visualizzazioni.

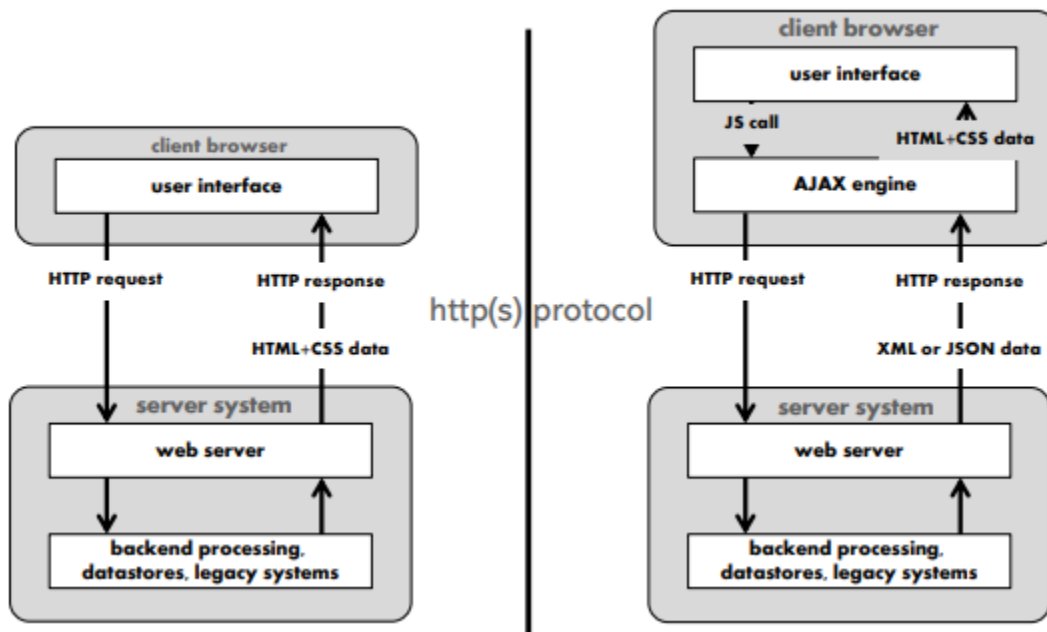
JavaScript è importante perché ha

- la capacità di effettuare richieste in HTTP al server, in maniera trasparente all'utente
- la funzione di rendere asincrona la comunicazione tra browser e web server

JavaScript può richiedere dati in formato testo puro e XML, il formato più diffuso è JSON.

Confronto Architettura Web App classica vs dinamica:

- La risposta alla XMLHttpRequest nella classica è una HTTPResponse (HTML + CSS ) mentre nella dinamica è XML. HTML vengono comunicati da AJAX Engine che fa da *tramite* tra il server e User Interface.



La trasmissione dei dati tra server e application RIA è un aspetto critico. Esistono diverse alternative tecnologiche: SOAP,XML-RPC, JSON, AMF.

La scelta del formato influenza la struttura dell'applicazione e le performance

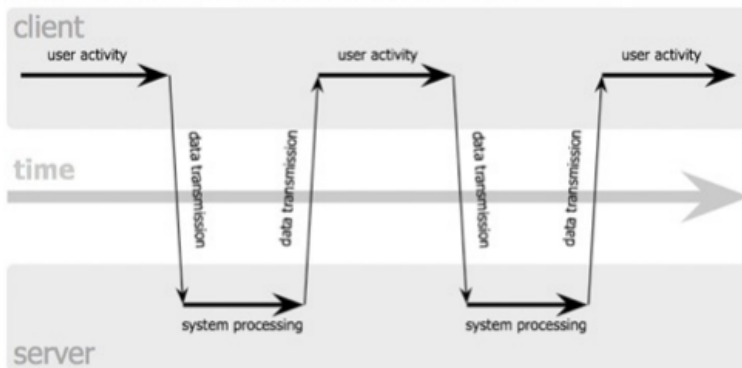
Un formato richiede tempo per:

- predisporre i dati lato server
- trasferire i dati
- fare il parsing dei dati lato client
- fare il rendering dell'interfaccia

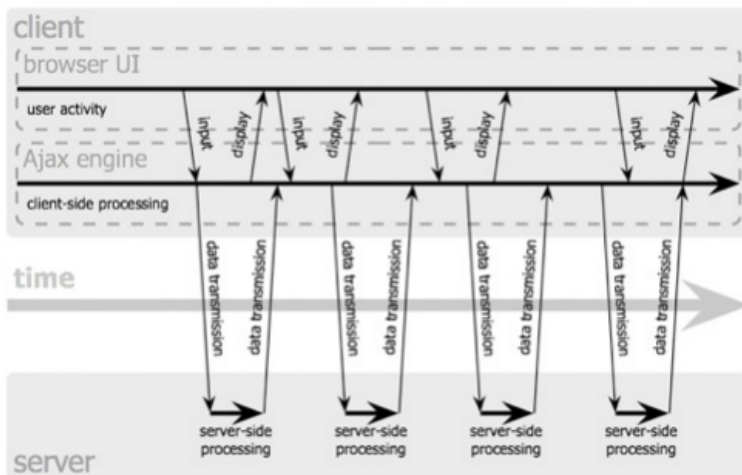
## Ajax Revealed



### classic web application model (synchronous)



### Ajax web application model (asynchronous)



**Function Invocation:** il codice all'interno della funzione verrà eseguito quando "qualcosa" invoca la funzione:

- Quando si verifica un evento
- Quando viene invocato dal codice JavaScript
- Automaticamente

Javascript reagisce agli eventi HTML

Si può associare uno o più gestori di eventi ad ogni elemento del DOM HTML che genera eventi: si utilizza la funzione `AddEventListener`.

Esempio:

```
<!DOCTYPE html>
<html>
<body>
  <p>Click the button to display the date.</p>
  <input type="button" value="The time is?" onclick="displayDate()" />
  <p id="demo"></p>

  <script>
    function displayDate() {
      document.getElementById("demo").innerHTML = Date();
    }
  </script>
</body>
</html>
```

## Programmazione JavaScript

In JavaScript:

- le variabili sono dichiarate **var** ed hanno global scope
- la parola riservata **let** permette di definire variabili valide nel block scope
- La parola riservata **const** permette di definire variabili let con un valore costante

Osservazione: le const sono imm modificabili nelle variabili semplici, ma per variabili composte si può modificare il contenuto

Le **funzioni** possono essere indicate con:

- function f(x,y)
- var f= function(x,y)
- const f= (x,y) =>

La keyword **return** e le parentesi graffe { } possono essere omesse solo quando la funzione ha una sola istruzione

Le **stringhe** sono segnalate con i doppi apici e sono concatenate con +. Hanno vari metodi:

1. length()
2. indexOf(), lastIndexOf()
3. startsWith(), endsWith()
4. slice(x), slice (x,y)

---

Osservazione: I metodi al numero 2. e 3. possono avere un parametro che segnala in che punto iniziare

Metodi per manipolare gli *array*:

- push() : aggiunge un elemento
- pop(): rimuove un elemento
- shift(): rimuove un elemento e shifta tutti di -1
- unshift(): aggiunge un elemento e shifta tutti di +1
- splice(int a, int b, x, ..., z) indica che vogliamo inserire nella a posizione b elementi: x,...,z

Tutti i valori non primitivi in JavaScript sono *oggetti*. Si crea un nuovo oggetto con new

Esistono, oltre al ciclo for normale, due *cicli* for specifici:

- for/of considera tutti i valori di una struttura iterativa (array, stringhe, ...)
- for/in considera tutte le proprietà di un oggetto

Esistono dei *metodi* per lavorare iterativamente su un array.

- Il metodo forEach() chiama una funzione per ogni elemento di un array.
- Il metodo map() crea un nuovo array chiamando una funzione su ogni elemento di un array
- Il metodo filter() crea un nuovo array che contiene gli elementi di un array che passano un test
- Il metodo reduce() esegue una funzione su un array e ritorna un singolo elemento

## RIA: Interazione con il Server

Si può associare uno o più gestori di eventi ad ogni elemento del DOM HTML che genera eventi: si utilizza la funzione *AddEventListener*(event, function)

```
1. <!DOCTYPE html>
2. <html>
3. <body>
4. <p>Click the button to display the date.</p>
5. <button id="button">The time is?</button>
6. <p id="demo"></p>
7. <script>
8. document.getElementById('button').addEventListener('click', displayDate);
```

---

```
9. function displayDate() {
10. document.getElementById("demo").innerHTML = Date();
11. }
12. </script>
13. </body>
14. </html>
```

## La funzione loadDoc()

1. carica il file ajax\_text.txt
2. se la nuova risorsa è caricata correttamente (status codice 200)
3. Allora il testo ricevuto viene visualizzato come contenuto di div demo

```
1. <body>
2. <div id="demo">
3. <h1>First Example</h1>
4. <p>This is a first example of a page including some javascript code to replace the content of the
   page.</p>
5. <p>Just click on the button to change this text.</p>
6. </div>
7. <br>
8. <button type="button" onclick="loadDoc()">Replace Text</button>
9. <script>
10. function loadDoc() ...
11. </script>
12. </body>
```

La classe *XMLHttpRequest()* definisce funzioni che nascondono le chiamate alla API di Sistema. Di solito si esegue:

- Predispone l'ambiente e crea la socket() (riga 3)
- Utilizza onreadystatechange per segnalare un nuovo stato del ciclo di scrittura/lettura che il programma può conoscere attraverso la variabile readyState (riga 5)
- Apre la connessione con la connect() e crea la prima riga di richiesta (obbligatoria) in formato HTTP (riga 9)
- Può popolare l'header del messaggio con funzioni dedicate (riga 9)
- Chiude il messaggio, lo invia utilizzando write(), e avvia la lettura della risposta con read(), riga(10)

```
1. <script>
2. function loadDoc() {
3. const xhttp = new XMLHttpRequest();
4. xhttp.onreadystatechange = function() {
5. if (xhttp.readyState == 4 && xhttp.status == 200) {
6. document.getElementById("demo").innerHTML = xhttp.responseText;
7. }
8. };

```

---

```
9. xhttp.open("GET", "ajax_text.txt", true);
10. xhttp.send();
11. }
12. </script>
```

## JSON objects

*JSON* (JavaScript Object Notation) è

- un formato di interscambio dati leggero
- facile da leggere e scrivere per gli esseri umani e per le macchine da analizzare e generare
- basato su un sottoinsieme di JavaScript
- un formato di testo completamente indipendente dalla lingua

JSON è costruito su due strutture dati universali:

- collezione di coppie nome/valore
- lista ordinata di valori

## RIA: PROGRAMMAZIONE LATO SERVER

*Node.js* è una piattaforma che permette di realizzare applicazioni web veloci e scalabili.

Approccio *asincrono*: Node.js usa un modello ad eventi e un sistema di I/O non bloccante che lo rende leggero ed efficiente.

L'esecuzione dei programmi in Node si basa su un *Single Event Loop*, che

- preleva un evento da una singola coda
- lo serve eseguendo le operazioni previste

Il modulo *file system* (fs) fornisce funzioni CRUD di gestione file

Node permette di realizzare un *Web server* che può gestire pagine html e applicazioni scritte in JavaScript.

Web app per leggere un file con Node: file server che

1. riceve un messaggio http

- 
2. estrae il nome della risorsa e
  3. restituisce la sua rappresentazione

Il modulo *express.js* è un framework web che fornisce gli strumenti per realizzare un server che può ospitare sia risorse statiche, sia applicazioni che generano rappresentazioni dinamiche

## Generazione di applicazioni MVC

La possibilità di associare dei percorsi ai metodi HTTP e associare a questi delle funzioni permette di realizzare *applicazioni REST*.

- bin: il file all'interno di bin chiamato www è il file di configurazione principale della nostra app.
- public: la cartella public contiene i file che devono essere resi pubblici
- route: la cartella route contiene file che contengono metodi per aiutare nella navigazione diverse aree della mappa. Contiene vari file js.
- views: la cartella delle views contiene vari file che costituiscono la parte delle viste del file applicazione.
- app.js: il file app.js è il file principale che è l'intestazione di tutti gli altri file. I vari i pacchetti installati devono essere "richiesti" qui.
- package.json: il file package.json è il file manifest di qualsiasi progetto Node.js e applicazione express.js.

## CAPITOLO 8: SEMANTICA DEI DATI

### Dati semantici

Problema: Affinché una macchina o un essere umano capiscano il significato dei dati necessari per aggiungere i *metadati*, ovvero informazioni extra che qualificano il contenuto dei file.

Un approccio diffuso consiste nell'affidarsi a lightweight collaborative repository collaborativi:

- offrono schemi semplici per specifiche descrizioni semantiche

- 
- forniscono modi per descrivere concetti semplici come cose, persone e posizioni.

Schema.org è diventato il più popolare di questi repository collaborativi.

*JSON-LD* (Linking Data) offre un modo semplice per migliorare semanticamente i documenti JSON:

- aggiunta di informazioni di contesto e collegamenti ipertestuali per descrivere la semantica del file diversi elementi di un oggetto JSON.
- inteso principalmente come un modo per utilizzare i dati collegati in base al Web ambienti di programmazione, per creare servizi Web interoperabili e per archiviare
- Dati collegati in motori di archiviazione basati su JSON

JSON-LD *estende* il linguaggio JSON con diverse keyword con il segno @.

Le tre principali keyword riservate che il linguaggio JSON-LD aggiunge a JSON sono:

- @*context*: reference URL ad uno schema
- @*id* di solito un URI
- @*type* reference URL al tipo di un valore

## Knowledge Graphs

Il *Semantic Web* si riferisce a un'estensione del web che promuove dati comuni formati per facilitare lo scambio di dati con significato tra le macchine

- Quando i motori di ricerca trovano contenuti dal web, le pagine non sono strutturate
- La specifica HTML da sola non definisce un vocabolario condiviso

I *Linked Data* sono un insieme di best practice per la pubblicazione e la connessione di dati strutturati sul Web, in modo che le risorse Web possano essere interconnesse in un modo che consenta ai computer di comprendere automaticamente il tipo e i dati di ciascuna risorsa

Fasi del Web:

- PC Era (80-90) : Desktop

- 
- Web 1.0 - connettere informazioni (90 - 00): WWW
  - Web 2.0 - connettere persone (00 - 10): Social Web
  - Web 3.0 - connettere conoscenze: Semantic Web
  - Web 4.0 - connettere intelligenze: MetaWeb

Machine-processable: supporto per query complesse su fatti. Si passa da dati a fatti.

## Resource Description Framework

**RDF** è un modello di dati per la rappresentazione dei dati sul Web basato su:

- Triple: unità di base per organizzare le informazioni
- Grafici orientati etichettati
- URI (URN + URL): identificatori univoci di risorsa

Un general purpose language per rappresentare i fatti nel Web:

- XML syntax
- Statements: soggetto predicato oggetto
- URI che identificato tutte le risorse
- tutto diventa risorsa

**IRI**: Internationalized Resource Identifiers sono dei superset di URI: si passa da ASCII ad UNICODE

In un grafo RDF un nodo vuoto `_:aid` si riferisce all'indirizzo

Caratteristiche di RDF:

- Indipendenza: poiché i predicati sono risorse, qualsiasi organizzazione indipendente può inventarli.
- Interchange: possono essere convertite in XML
- Scalabilità: le proprietà RDF sono semplici <soggetto, predicato, oggetto> triple, quindi sono facili da identificare
- Le proprietà sono risorse
- Soggetto e oggetto possono essere risorse, quindi possono avere anche delle proprietà.



---

## CAPITOLO 9: ALGORITMI DISTRIBUITI

### Sincronizzazione degli Orologi

Ci sono due modi per creare sincronizzazione tra orologi:

- orologi *fisici*: computer con un tic che fa avanzare il tempo. H interrupts al secondo
- orologi *logici*: ordinamento relativo tra gli eventi. Può essere a sua volta:
  - Ordinamento totale: orologi di Lamport
  - Ordinamento causale: vector timestamp

Lamport timestamps:

- Introduce la relazione happens before:  $a \rightarrow b$  vuol dire a happens before b
- Per rappresentarle si usano valori di orologio: Se  $a \rightarrow b$  allora  $C(a) < C(b)$
- Osservazione: I valori di C possono solo crescere: se clock ricevuto è maggiore allora aggiornano il clock locale aggiungendo 1 al clock ricevuto

Algoritmo:

- Un processo manda un messaggio a molti
- Chi riceve lo mette in coda secondo il timestamp associato
- Ciascuno manda un ack a tutti (con un timestamp corretto)
- Solo quando ciascuno ha ricevuto tutti gli ack il messaggio può essere processato

### Algoritmi di mutua esclusione

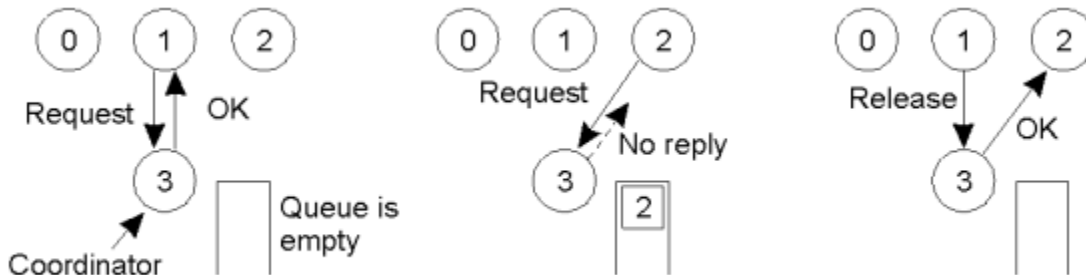
Gli *algoritmi di mutua esclusione* possono essere ottenuti in 3 diversi modi:

1. Coordinatore centralizzato
2. Sincronizzazione distribuita
3. Turni alternati: soluzione basata su token

*Coordinatore centralizzato*: Simula in ambiente distribuito le soluzioni per sistemi monoprocesso:

- Non ci sono deadlock o starvation

- 3 messaggi (request, grant, release)
- Singolo punto di fallimento (failure)
- Coordinatore è un *bottleneck*

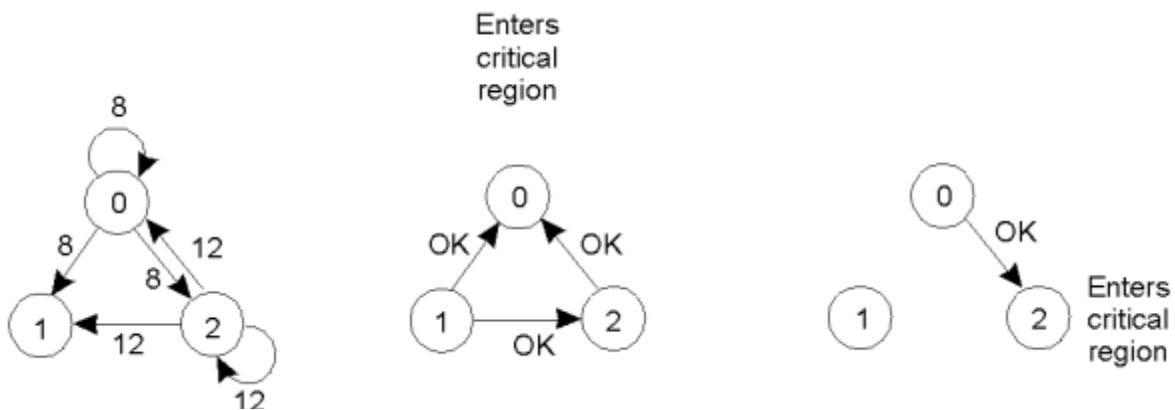


*Sincronizzazione distribuita:* Un processo A vuole entrare in una regione critica, allora invia la richiesta a tutti gli altri processi e aspetta un OK da ciascun processo. Chi riceve il messaggio si trova in una delle seguenti condizioni:

- Non interessato => manda ok
- Nella regione critica => postpone la risposta
- In attesa di ok => confronta il timestamp della propria richiesta e quello di A, se maggiore manda ok, se minore postpone la risposta

Osservazioni:

- Non ci sono deadlock o starvation
- $2(n-1)$  messaggi con  $n$  processi
- $n$  punti di fallimento, soluzione: si risponde sempre con OK o NO
- Bisogna conoscere l'entità dei partecipanti e il loro numero o avere meccanismi di gestione di gruppo
- Performance: se un processo non riesce a rispondere viene rallentato l'intero sistema



**Turni Alternati con Token:** A turno i partecipanti si passano un token: chi lo possiede può accedere alla risorsa condivisa:

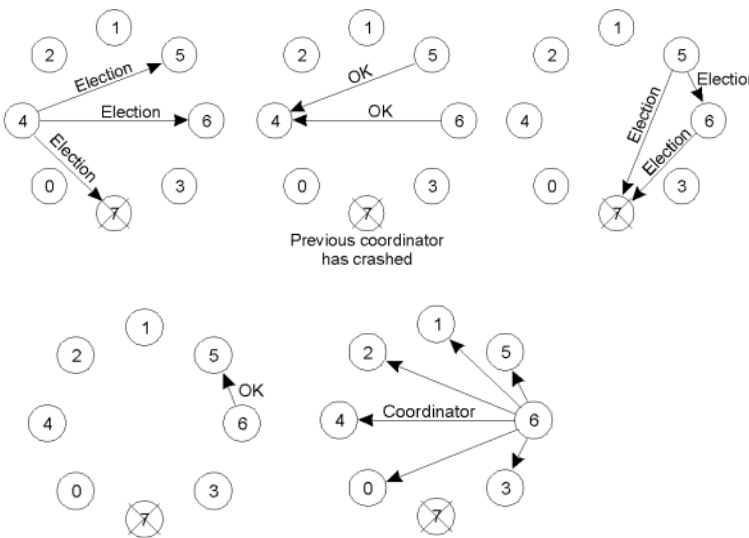
- Vantaggi:
  - Semplicità
  - Evita starvation
  - Permette di evitare deadlock progettando opportunamente i processi
- Svantaggi:
  - In caso di *fault* il token si potrebbe perdere: servono algoritmi appositi per rilevare il problema e generare un nuovo token, ovvero algoritmi di elezione

## Algoritmi di elezione

Talvolta serve un *coordinatore*: ogni processo ha ID univoco, si elegge il processo con ID maggiore. Ci sono due algoritmi

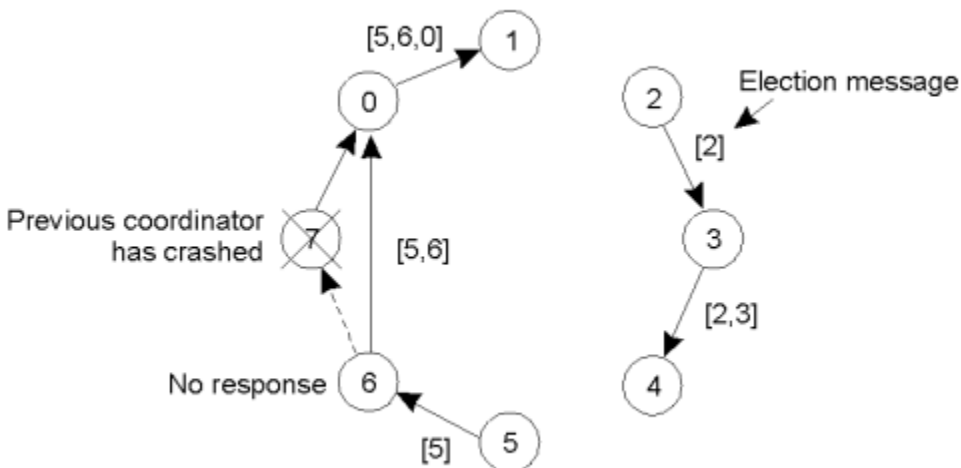
**Bully Algorithm:**

1. Il processo 7, precedente coordinatore, si è arrestato in modo anomalo.
2. Il processo 4 tiene un'elezione inviando un messaggio a tutti i processi con numeri più alti
3. I processi 5 e 6 rispondono, dicendo a 4 di fermarsi
4. Ora 5 e 6 tengono ciascuno un'elezione
5. Processore 6 dice a 5 di fermarsi
6. Processore 6 vince e notifica tutti



### *Ring algorithm:*

1. Due processi, 2 e 5, scoprono contemporaneamente che il precedente coordinatore, processo 7, si è arrestato in modo anomalo.
2. Ognuno di questi crea un messaggio di elezione e inizia a farlo circolare
3. Alla fine, sia il 2 che il 5 convertono il messaggio in un messaggio coordinatore, con esattamente gli stessi membri e nello stesso ordine
4. Quando entrambi saranno andati di nuovo in giro, entrambi verranno rimossi
5. Non è un problema avere extra messaggi circolanti; nel peggiore dei casi consuma un po' di larghezza di banda, ma questo non è considerato uno spreco.



---

## Fault Tolerance

Concetti base per permettere tolleranza ai fallimenti:

- **Disponibilità:**
  - un sistema è pronto per essere utilizzato subito.
  - il sistema *funzioni* correttamente in ogni momento ed è disponibile a svolgere le sue funzioni per conto dei propri utenti.
- **Affidabilità:**
  - un sistema può funzionare continuamente senza fallimento
  - contrariamente alla disponibilità, l'affidabilità è definita in *termini di intervallo* anziché un istante nel tempo
  - un sistema altamente affidabile è quello che molto probabilmente continuerà a funzionare senza interruzione per un periodo di tempo relativamente lungo
- **Sicurezza:**
  - un sistema non funziona temporaneamente correttamente, non accade nulla di *catastrofico*.
  - molti sistemi di controllo di processo, come quelli utilizzati per controllare le centrali nucleari o inviare persone nello spazio, lo sono per fornire un elevato grado di sicurezza.
- **Manutenibilità:**
  - un sistema guasto può essere riparato.

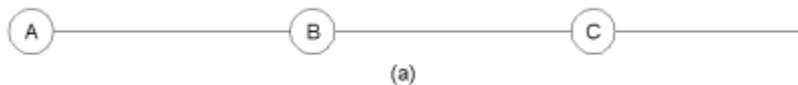
*Tipi di fallimenti:*

- Crash failure: Un server si *arresta*, ma funziona correttamente finché non si arresta
- Omission failure: Un server *non riesce a rispondere* alle richieste in arrivo
  - Receive omission: Un server non riesce a ricevere i messaggi in arrivo
  - Send omission: Un server non riesce a inviare messaggi
- Timing failure: La *risposta* di un server *non rientra* nell'intervallo di tempo specificato
- Response failure: La *risposta* del server *non è corretta*
  - Value failure: Il valore della risposta è errato
  - State transition failure: Il server devia dal corretto flusso di controllo

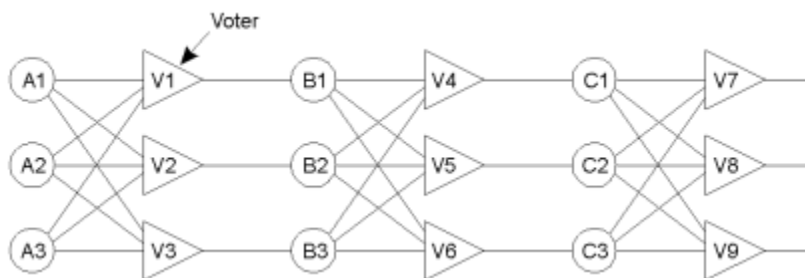
- Arbitrary failure: Un server può produrre *risposte arbitrarie* in tempi arbitrari

## Tripla ridondanza modulare

I segnali passano attraverso i dispositivi A, B, e C, in sequenza. Se uno di loro è difettoso, il risultato finale sarà probabilmente errato



Ogni dispositivo è replicato *tre volte*. Dopo ogni fase ogni elettore è triplicato con tre ingressi e un'uscita. Se due o tre degli ingressi sono uguali, l'output è uguale a quello dell'ingresso. Se tutti e tre gli input sono diversi, l'output non è definito.



L'approccio chiave per tollerare un processo difettoso è l'organizzazione di diversi processi identici in un gruppo:

- Comunicazione in un gruppo piatto
- Comunicazione in un gruppo gerarchico semplice

Accordo nei sistemi difettosi:

1. *Sistemi sincroni contro sistemi asincroni:*
  - a. Un sistema è sincrono se e solo se è noto che i processi operano in una modalità lock-step
  - b. Formalmente, questo significa che ci dovrebbe essere una costante  $c$  tale che se un processore ha eseguito  $c + 1$  passaggi, ogni altro processo ha eseguito a almeno 1 passaggio
  - c. Un sistema che non è sincrono è detto asincrono.

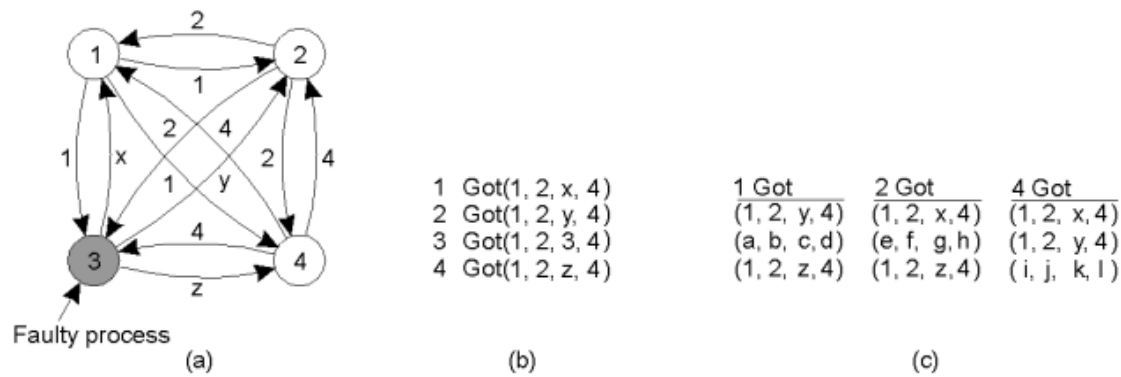
2. Il *ritardo di comunicazione è limitato* o meno
  - a. Il ritardo è limitato se e solo se sappiamo che ogni messaggio viene consegnato con un tempo massimo globale e predeterminato
3. La *consegna del messaggio è ordinata* o meno
  - a. Distinguiamo la situazione in cui i messaggi dello stesso mittente vengono consegnati nell'ordine in cui sono stati inviati, dalla situazione in cui noi non hanno tali garanzie
4. La *trasmissione dei messaggi* avviene tramite unicasting o multicasting

		Message ordering				Communication delay
		Unordered		Ordered		
		Unicast	Multicast	Unicast	Multicast	
Process behavior	Synchronous			X		Bounded
				X		Unbounded
	Asynchronous	X	X	X	X	Bounded
				X	X	Unbounded

Partiamo dal presupposto che i processi sono sincroni, mentre i messaggi sono unicast, preservando l'ordine e il ritardo di comunicazione è limitato.

*Byzantine generals' problem:* abbiamo 3 generali leali e 1 traditore:

1. I generali annunciano la forza delle loro truppe (in unità di 1 chilosoldato)
2. i risultati vengono raccolti da ciascun generale in vettori
3. i vettori vengono inviati a tutti gli altri generali
4. Ogni generale esamina l'i-esimo elemento di ciascuno dei vettori ricevuti, se un valore ha la maggioranza, quel valore o unknown viene inserito nel vettore dei risultati



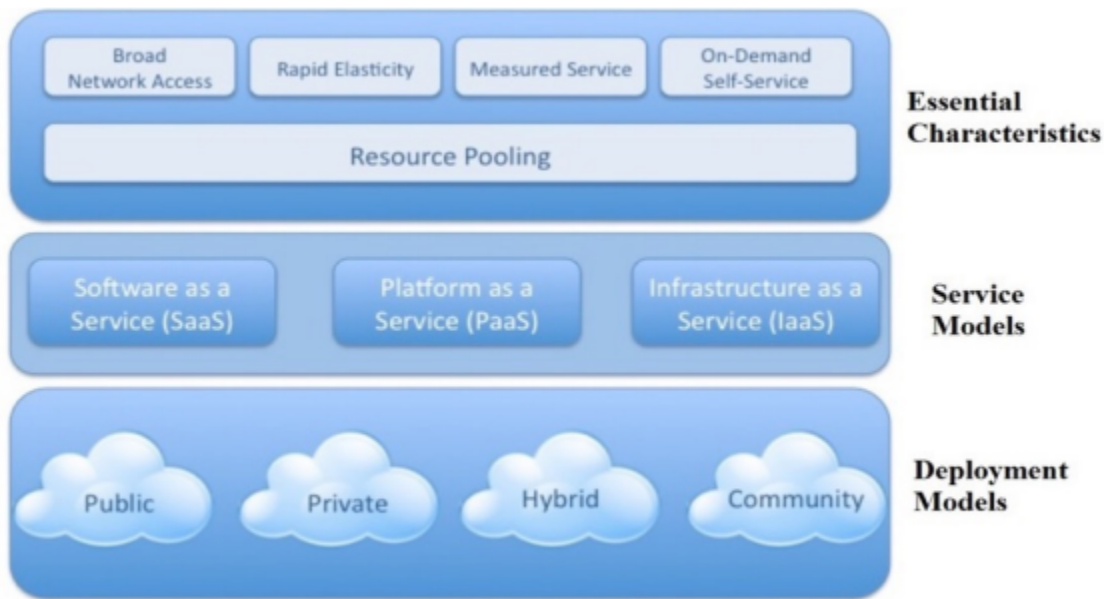
## CAPITOLO 10: CLOUD COMPUTING

*Cloud computing* è un modello di elaborazione che fornisce capacità IT scalabili ed elastiche come servizio ai clienti esterni utilizzando tecnologie Internet:

- è un'esperienza dell'utente e un modello business
- è una metodologia di manipolazione di un'infrastruttura

Il modello cloud del NIST (National Institute of Standards and Technology) è garantito da 5 caratteristiche, 3 modelli di servizio e 4 modelli di distribuzione.





#### *Caratteristiche:*

- On-demand self-service: Un consumatore può fornire unilateralmente funzionalità di elaborazione senza richiesta di interazione umana
- Broad network access: Le capacità sono disponibili sulla rete e vi si accede tramite meccanismi standard
- **Resource pooling:** Le risorse informatiche del provider sono raggruppate per servire più consumatori utilizzando un modello **multi-tenant**
- Rapid elasticity: le risorse devono sembrare illimitate
- Measured service: l'utilizzo di risorse può essere monitorato e controllato

#### *Modelli di servizio:*

- Software as a Service (**SaaS**)
- Platform as a Service (**PaaS**)
- Infrastructure as a Service (**IaaS**)

#### *Modelli di distribuzione:*

- 
- **Public:** L'infrastruttura cloud viene messa a disposizione del pubblico (community) o una grande industria (dedicated) ed è di proprietà di un'organizzazione che vende servizi cloud. Anche conosciuta come cloud esterno o multi-tenant cloud.
  - **Private:** L'infrastruttura cloud è gestita esclusivamente per un'organizzazione. Potrebbe essere gestito dall'organizzazione o da una terza parte in sede o fuori sede. Definito anche come cloud interno o cloud on-premise
  - Ibridi

## Virtualizzazione

La funzione più importante della virtualizzazione è la possibilità di eseguire più sistemi operativi e applicazioni contemporaneamente, indipendentemente dalla piattaforma o hardware:

- aiuta a isolare guasti causati da errori o problemi di sicurezza;
- consente l'introduzione di nuove funzionalità del sistema senza aggiungere complessità hardware e software complessi già esistenti;
- di solito può migliorare le prestazioni complessive dell'applicazione fornendo solo ciò di cui l'utente ha bisogno

Due tipi di virtualizzazioni:

- Process Virtual Machine: virtualizzazione tramite interpretazione o emulazione: fatto per un solo processo, fornisce un set di istruzioni macchina astratto; i programmi sono compilati in codice macchina che viene quindi interpretato (ad es. Java) o emulato (ad es. Windows)
- Virtual Machine Monitor: in grado di fornire una macchina virtuale a molti diversi programmi contemporaneamente; come se più CPU fossero in esecuzione su una singola piattaforma

## Microservices

Un'architettura di **microservizi** inserisce ogni elemento di funzionalità in un servizio separato e rende ogni servizio un'unità di distribuzione indipendente.

---

I microservizi promuovono i principi e le pratiche SE:

- Basso accoppiamento e alta coesione
- Asincrono > sincrono
- Coreografia > orchestrazione

Le applicazioni basate su microservizi preferiscono protocolli più semplici e leggeri come REST.

Applicazioni *monolitiche*:

- Al centro dell'applicazione c'è la logica di business che è implementata dai moduli che definiscono servizi, oggetti di dominio ed eventi.
- Intorno al nucleo ci sono gli adattatori che si interfacciano con il mondo esterno.
- Pur avendo un'architettura logicamente modulare, l'applicazione è impacchettata e distribuita come un monolito.
- Il formato effettivo dipende dal linguaggio e framework dell'applicazione.

Le applicazioni monolitiche se grosse possono portare a molti problemi: mantenibilità, dependency hell, lock-in, reliability, costosi ecc..

Invece di creare un'unica applicazione grande e monolitica, l'idea è di dividere l'applicazione in un insieme di più piccoli servizi interconnessi. Un servizio in genere implementa un insieme di caratteristiche distinte o funzionalità. I microservizi garantiscono:

- risoluzione di problemi di complessità
- ogni servizio è sviluppato, distribuito e scalato indipendentemente

## Containers

I *containers* sono un modo standard per impacchettare un'applicazione e tutte le sue dipendenze in modo che possa essere spostata da un ambiente all'altro ed eseguita senza modifiche

---

*Docker* è una piattaforma aperta per costruire applicazioni distribuite per sviluppatori e amministratori di sistema. *Docker Engine* è un'applicazione client-server con questi componenti principali:

- Un server, tipo di programma di lunga durata chiamato processo demone
- Una API REST che specifica le interfacce che i programmi possono usare per parlare con il demone e istruirlo su cosa fare.
- Un client di interfaccia a riga di comando

Un'immagine *Docker* è composta da file system sovrapposti l'uno sull'altro. Utilizza la tecnica del montaggio congiunto che permette a diversi filesystem di essere montati allo stesso tempo ma apparire come un unico filesystem.

Il modello "copy on write" è una delle caratteristiche che rende Docker così potente: quando Docker avvia per la prima volta un contenitore, lo strato iniziale di lettura-scrittura è vuoto. Man mano che i cambiamenti avvengono, vengono applicati a questo strato.