

Sistemi distribuiti

Un sistema distribuito è composto da elementi hardware o software che **interagiscono e si coordinano esclusivamente tramite lo scambio di messaggi**. Questo tipo di sistema si presenta agli utenti come un'entità unica e coerente, nonostante sia costituito da una collezione di elementi di calcolo autonomi.

In un sistema distribuito, ogni computer, dotato di un proprio sistema operativo, opera come un **nodo autonomo**. Abbiamo le seguenti caratteristiche:

- **Gestione della memoria:** non esiste una memoria condivisa tra i nodi, e la comunicazione avviene unicamente attraverso lo scambio di messaggi
- **Gestione dell'esecuzione:** ogni nodo opera in modo autonomo, è l'esecuzione è concorrente. Il coordinamento è cruciale per definire il comportamento complessivo del sistema o dell'applicazione
- **Gestione del tempo:** non c'è un clock globale che sincronizza i nodi, ogni stato è gestito localmente, e il coordinamento avviene solo attraverso la comunicazione
- **Tipi di fallimenti:** i nodi possono fallire indipendentemente l'uno dall'altro, senza causare un fallimento a livello di sistema

Architettura

L'architettura nei sistemi distribuiti descrive come i vari componenti interagiscono e si organizzano. Questi componenti possono essere disposti secondo diversi stili architettonici:

- **Layered:** un modello a strati dove ogni livello nasconde la complessità di quello sottostante
 - o **Pure layered organisation:** segue il modello ISO/OSI con strati distinti
 - o **Mixed layered organisation:** permette di bypassare alcuni strati
 - o **Mixed downcalls and upcalls:** una combinazione di chiamate ascendenti e discendenti, spesso implementate con callback
- **Livelli:** una struttura con dipendenze, simile alla relazione client-server, senza strati intermedi
- **Basate sugli oggetti:** utilizza l'invocazione di metodi remoti, seguendo i principi della programmazione orientata agli oggetti
- **Centrati sui dati:** come il web, che funziona come un file system distribuito definendo risorse accessibili globalmente
- **Basate su eventi:** dove le azioni sono guidate da eventi specifici

Sistemi operativi distribuiti

Per quanto riguarda questo argomento, abbiamo tre diversi tipi:

- **Distributed Operating Systems:** offrono funzionalità che mascherano la presenza di *molteplici macchine*, gestendo anche il bilanciamento del carico e l'accesso ai dati. Richiedono hardware omogeneo, e il sistema operativo può essere condiviso da tutte le macchine appartenenti al cluster
 - **Funzionalità richieste:** *bilanciamento del carico* (distribuisce i processi sulla rete), *accelerazione del calcolo* (i sottoprocessi possono essere eseguiti contemporaneamente su siti diversi), *preferenze hardware* (l'esecuzione del processo potrebbe richiedere un processore specializzato), *preferenze software* (il software richiesto potrebbe essere disponibile solo in un determinato sito), *accesso ai dati* (eseguire il processo in remoto, invece di trasferire tutti i dati localmente)
 - **Caratteristiche particolari:** utenti non consapevoli della molteplicità delle macchine, accesso a risorse remote come l'accesso a risorse locali; migrazione dei calcoli: trasferire il calcolo attraverso il sistema
- **Network Operating Systems:** sono progettati per reti, includono primitive per lo sviluppo di applicazioni distribuite, e supportano hardware eterogeneo
 - Gli utenti sono consapevoli della molteplicità di macchine
 - Fornisce **funzionalità di comunicazione esplicite**: comunicazione diretta tra i processi (socket), esecuzione concorrente dei processi provenienti da un'applicazione distribuita, e i servizi sono gestiti dalle applicazioni
 - L'accesso alle risorse di varie macchine viene effettuato esplicitamente da *accesso remoto alla macchina di destinazione*, *Remote Desktop*, *trasferimento di dati da macchine remote a macchine locali tramite FTP*
- **Middleware:** si posiziona sopra il NOS per fornire servizi globali, evitando la duplicazione di funzionalità nelle applicazioni.
 - Su sistemi DOS, si occupa di realizzare servizi e si basa su macchine omogenee
 - Su sistemi NOS, i servizi sono gestiti in modo esplicito dalle applicazioni, e si basa su macchine eterogenee
 - Ha una serie di **servizi**, che possono risolvere diversi problemi:
 - **Naming** (i nomi simbolici vengono utilizzati per identificare le entità che fanno parte di un DS, possono essere utilizzati dai registri per fornire gli indirizzi reali)
 - **Trasparenza di accesso:** definisce e offre un modello di comunicazione che nasconde i dettagli sul passaggio dei messaggi
 - **Persistenza:** definisce e offre un servizio automatico per l'archiviazione dei dati
 - **Transazioni distribuite:** definisce e offre modelli di persistenza per garantire automaticamente la coerenza di operazioni di lettura/scrittura

- **Sicurezza:** definisce e offre modelli per proteggere l'accesso a dati e servizi, e integrità di calcolo

Modello client-server

Describe un'interazione dove **il client richiede un servizio e il server risponde**, con tempi di inattività alternati tra i due. Le operazioni richiedono tempo, e possono essere gestite da un **cluster di server**, o tramite **proxy**.

I problemi

Qualsiasi sistema distribuito deve affrontare quattro problemi:

- **Identificare la controparte:** *denominazione*
- **Accedere alla controparte:** come posso raggiungere una risorsa remota —> riferimento (*punto di accesso*)
- **Comunicare:** abbiamo bisogno di un *protocollo* e capire *la semantica/sintassi dei dati*

Trasparenza della distribuzione

Con **trasparenza** intendiamo nascondere i dettagli agli utenti, in modo che possano ignorare ciò che accade, e non possano influenzare il servizio fornito. Abbiamo bisogno della trasparenza in una serie di servizi:

- **Denominazione:** i nomi simbolici vengono utilizzati per identificare le risorse che fanno parte di un sistema distribuito
- **Accesso:** nascondere le differenze nella rappresentazione dei dati e il modo in cui si accede a una risorsa locale o remota
- **Posizione:** nascondere dove si trova una risorsa nella rete
- **Trasferimento della mobilità:** nascondere che una risorsa può essere spostata in un'altra posizione mentre è in uso
- **Migrazione:** nascondere una risorsa può spostarsi in un'altra posizione
- **Replica:** nascondere che una risorsa viene replicata
- **Concorrenza:** nascondere che una risorsa può essere condivisa da più utenti indipendenti, garantendo la coerenza dello stato
- **Fallimento:** nascondere l'errore e il ripristino di una risorsa
 - **Omission failure:** Un servente accetta richieste ma fallisce nella ricezione di tale richiesta o nell'elaborazione
- **Persistenza:** nascondere che una risorsa è volatile o memorizzata in modo permanente

Mirare però alla piena trasparenza della distribuzione potrebbe essere troppo:

- Ci sono **latenze di comunicazione** che non possono essere nascoste
- Nascondere completamente i guasti di reti e nodi è **impossibile**: non è possibile distinguere un computer lento da uno guasto, e non si può mai essere sicuri che un server abbia eseguito un'operazione prima di un arresto anomalo
- Una piena trasparenza **costerà le prestazioni**, poiché per esempio richiede tempo per *mantenere le repliche esattamente aggiornate con il master, e svuotare immediatamente le operazioni di scrittura su disco per la tolleranza agli errori*
- L'esposizione della distribuzione può essere buona usando *servizi basati sulla posizione*, quando si ha a che fare con utenti in fusi orari diversi, o quando rende più facile per un utente capire cosa sta succedendo.

Abbiamo anche alcuni termini importanti:

- **Failure transparency**: Un sistema è in grado di portare a termine un compito anche in presenza di fallimenti parziali
- **Migration/Mobility Transparency**: Lo spostamento di software e/o dati non comporta modifiche nei programmi utente
- **Persistence Transparency**: un dato può essere in memoria o disco rigido
- **Access Transparency**: Accesso a risorse locali e remote con le stesse operazioni e con lo stesso formato di dati

Occultamento

L'occultamento delle informazioni è un principio fondamentale nell'ingegneria del software.

Abbiamo una separazione tra:

- **Il cosa** (quale servizio fornisce un componente o sistema)
 - o È definito da un **IDL** (Interface Definition Languages), e da definire invece l'**API** (Application Programming Interface) di componenti o sistemi. Dovrebbe essere annotato semanticamente
- **Il come** (come quel servizio è stato implementato e distribuito)
 - o È implementato con uno strumento (come un framework) adatto a quel problema specifico o ambiente. Dovrebbe essere implementato con algoritmi e tecnologie specifici ed efficaci

Le interfacce invece dovrebbero essere progettate secondo principi condivisi, complete, e indipendenti da un'implementazione o distribuzione specifica; dovrebbe supportare anche l'interoperabilità, la portabilità e l'estendibilità.

Politiche vs meccanismi

Un sistema distribuito dovrebbe essere composto da componenti **indipendenti**, dove ogni componente *fornisce autonomamente un servizio* o esegue un compito (indipendenza logica), e dove ogni componente *usa o collabora con altri componenti per eseguire o fornire servizi più complessi* (composizione).

Questo può essere facilitato da una netta separazione tra **politiche** (come sfruttare le capacità, quindi definire un comportamento) e **meccanismi** (capacità fornite dai componenti). Un esempio nei sistemi operativo è il *cambio di contesto* (meccanismo) e il *Round Robin* (politica).

Protocollo

Per poter capire le richieste e formulare le risposte, due processi devono concordare un **protocollo**, che 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.

Le applicazioni su TCP/IP si **scambiano** (meccanismo) stream di byte di lunghezza infinita, che possono essere segmentati in **messaggi** (politica).

I programmi

Un programma è una sequenza di istruzioni che definisce il comportamento di un'entità computazionale, come un computer o un dispositivo. Un **processo** è un'istanza di un programma in esecuzione, che occupa una certa quantità di memoria e di risorse; può essere visto anche come un **esecutore di programmi**, che non termina finché non ha completato il suo compito, o non riceve un segnale di interruzione. Un processo può comunicare con altri processi tramite canali, che sono flussi di dati in ingresso e in uscita.

Servizio TCP/UDP

Per poter iniziare a parlare di questo argomento, dobbiamo prima parlare del servizio TCP/IP. È il protocollo di comunicazione più usato in internet, e permette di trasmettere dati tra processi remoti. Si basa sull'architettura **client-server**, in cui un client richiede un servizio da un server, che poi esegue l'attività e restituisce i risultati al client. Abbiamo due tipi di servizi:

- **TCP** è orientato alla connessione, cioè richiede che il client e il server stabiliscano una connessione prima di scambiarsi dati; garantisce l'*affidabilità*, il *controllo di flusso* e il *controllo della congestione dei dati*

- UDP è un trasporto non affidabile, cioè non richiede una connessione e non offre garanzie di qualità dei dati, ma può essere più veloce e adatto per applicazioni che tollerano perdite parziali di dati.

La socket

Una socket è un tipo particolare di canale, che permette la **comunicazione tra processi che non condividono memoria**, cioè che si trovano su macchine diverse o parti diverse di una stessa macchina. Una socket usa il servizio TCP/IP per trasmettere dati tra processi remoti, usando le normali system call *read/write*. Una socket richiede che il processo mittente e il processo ricevente conoscano l'indirizzo IP e la porta della macchina che esegue il processo remoto. Può usare sia TCP che UDP, a seconda delle esigenze dell'applicazione.

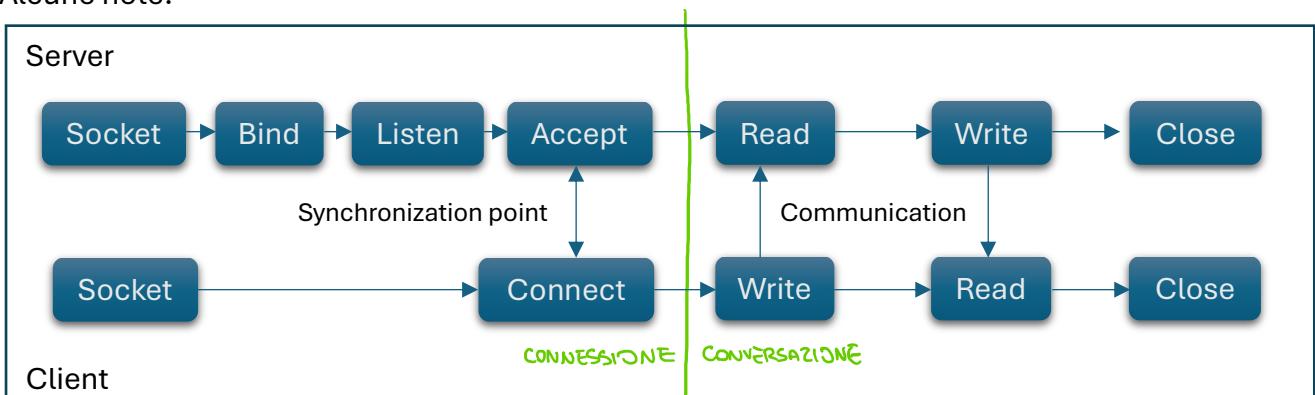
Creazione e gestione

Per creare e gestire una socket, sia il client che il server devono eseguire la **system call socket**, che è sospensiva, ovvero richiede il passaggio del controllo all'OS. Successivamente:

Client	Server
System call connect (IP server: porta server) per stabilire la connessione —>	System call accept (per accettare la richiesta di connessione)
(SE ACCETTATA:	FINE CONNESSIONE, INIZIO CONVERSAZIONE)

Una volta passata alla fase di conversazione, il client e il server possono scambiarsi dati tramite le system call **read/write** in modo bidirezionale (*system call sono bloccanti poiché passano in modalità kernel*), cioè senza un ordine prestabilito. La comunicazione avviene attraverso flussi di byte, che devono essere interpretati secondo il protocollo di alto livello usato, come http o SMTP. La comunicazione può essere anche di tipo **asincrono**, cioè non richiedere una risposta immediata dal destinatario, o di tipo **multicast**, cioè inviare lo stesso messaggio a più destinatari contemporaneamente.

Alcune note:



- Nella fase di **connect**, il formato della richiesta viene stabilito dal livello di trasporto, perché, indipendentemente dall'applicazione, la connessione è la stessa

- Una volta accettata, si usa una **nuova socket** dedicata alla comunicazione con il client. Questo perché *il server non riesce a distinguere* tra una richiesta di connessione o conversazione; se si mandano soltanto byte, non si riesce a distinguere.
- Nella fase di connessione si usa il **TCP protocol**, mentre nella fase di conversazione usiamo l'**application protocol**
- È importante indicare la dimensione dei payload in risposta perché **non è mai possibile usare un carattere terminatore**
- Al fine di leggere i messaggi, possiamo avere potenzialmente un flusso di dati infiniti, ma non possiamo avere memoria infinita. Quindi, bisognerà leggere **pezzo per pezzo**; la dimensione di ogni pezzo è arbitrario. Bisogna sapere inoltre, ogni volta, quanto devo leggere e quando devo finire. Effettivamente, quindi bisogna avere:

socket	byteLetti	buffer	dimBuffer
--------	-----------	--------	-----------

Le socket in Java

Java definisce alcune classi che costituiscono un'interfaccia ad oggetti alle *system call*; le principali sono **java.net.Socket** e **java.net.ServerSocket**. Queste classi accorpano funzionalità, e mascherano alcuni dettagli con il vantaggio di semplificare l'uso.

Come per ogni framework, è necessario conoscerne il modello e il funzionamento per poterlo utilizzare in modo efficace.

Classe Socket

- **Costruttori:**
 - **public Socket():** crea una socket connessa con il tipo di SocketImpl predefinito dal sistema
 - **public Socket(String host, int port) throws UnknownHostException, IOException:** crea una socket di tipo stream, e la connette al numero di porta specificato sull'host nominato. Se l'host è *null*, viene assunto l'indirizzo di loopback
 - **public Socket(InetAddress address, int port) throws IOException:** crea una socket di tipo stream e la connette al numero di porta specificato all'indirizzo IP indicato
- **Metodi per gestire le connessioni:**
 - **public void bind(SocketAddress bindpoint) throws IOException:** associa la socket a un indirizzo locale

- **public void connect(SocketAddress endpoint) throws IOException:** connette questa socket al server
- **public void connect(SocketAddress endpoint, int timeout) throws IOException:** connette questa socket al server con un valore di timeout specificato, in millisecondi
- **public void close():** chiude la socket
- **Metodi per stabilire canali I/O per lo scambio di byte:**
 - **public InputStream getInputStream() throws IOException:** restituisce uno stream di input per questa socket
 - **public OutputStream getOutputStream() throws IOException:** restituisce uno stream di output per scrivere byte su questa socket

Classe ServerSocket

- **Costruttori:**
 - **public ServerSocket() throws IOException:** crea una server socket non vincolata
 - **public ServerSocket(int port) throws IOException:** crea una server socket vincolata alla porta specificata
 - **public ServerSocket(int port, int backlog) throws IOException:** crea una server socket e la vincola al numero di porta locale specificato, con il backlog specificato
- **Metodi per gestire le connessioni:**
 - **public void bind(SocketAddress endpoint) throws IOException:** associa il ServerSocket a un indirizzo specifico (indirizzo IP e numero di porta). Se l'indirizzo è *null*, il sistema sceglierà una porta effimera e un indirizzo locale valido per legare la socket
 - **public void bind(SocketAddress endpoint, int backlog) throws IOException:** associa il ServerSocket a un indirizzo specifico, con un valore di backlog che deve essere un valore positivo maggiore di 0. Se il valore passato è uguale a 0, verrà assunto il valore predefinito
 - **public Socket accept() throws IOException:** ascolta una connessione da realizzare su questa socket e la accetta. Restituisce la nuova socket. Il metodo si blocca fino a quando non viene stabilita una connessione
- **Metodi di utilità:**
 - **public InetAddress getInetAddress():** restituisce l'indirizzo locale di questo server socket, o null se la socket non è vincolata
 - **public int getLocalPort():** restituisce la porta su cui questa socket non sta ascoltando, o -1 se la socket non è ancora vincolata
 - **public SocketAddress getLocalSocketAddress():** restituisce l'indirizzo dell'endpoint a cui questa socket è vincolata, o null se non lo è.

Architettura client-server

L'architettura tra i due cambia:

- **Client:** è più semplice rispetto a quella di un server. Tipicamente, è un'applicazione standard che **stabilisce una connessione di rete tramite una socket**. Questo approccio ha un impatto limitato all'utente del client, e non presenta significative preoccupazioni di sicurezza, poiché il client gestisce solo le proprie risorse e dati.
- **Server:** un server deve essere progettato con maggiore attenzione. La sua architettura include:
 - La creazione di una socket su una porta nota per ricevere richieste di connessione dai client
 - L'invio di stream di bytes non limita il tipo di messaggi da inviare
 - L'ingresso in un ciclo continuo che alterna tra
 - L'attesa e l'accettazione di richieste di connessione dai client
 - Ciclo di lettura ed esecuzione di comandi, seguito dall'invio di risposte, che continua fino al termine della conversazione, spesso determinato dal client stesso
 - La chiusura della connessione una volta completata la comunicazione

La stabilità e l'affidabilità di un server dipendono fortemente dalla qualità della comunicazione con i suoi client. È difficile rilevare le interruzioni nelle connessioni, poiché di solito è il client a monitorare lo stato del server. Ogni conversazione richiede una connessione o socket dedicata, che può aumentare la complessità e le risorse necessarie. La condivisione dei dati e il controllo affidato al client possono introdurre vulnerabilità di sicurezza, richiedendo quindi misure di protezione aggiuntive.

L'identificazione delle risorse, inoltre, **non è risolta** in quanto è necessario conoscere l'indirizzo dei componenti.

Concorrenza

La concorrenza è un concetto fondamentale nei sistemi distribuiti, che si riferisce alla **capacità di eseguire contemporaneamente diverse parti di un programma**, come processi o thread, noti anche come agenti. Questa caratteristica è essenziale nello sviluppo di software efficiente e reattivo.

Interessante notare che due agenti possono operare in *maniera contemporanea*, anche condividendo la stessa unità di elaborazione centrale (CPU).

Scenari di concorrenza

Esistono principalmente due scenari di concorrenza:

- **Concorrenza su singola macchina:** che sia dotata di una sola CPU o di più core, gli agenti possono condividere la stessa memoria e le funzioni del sistema operativo
- **Concorrenza su macchine distinte:** il programma opera su computer collegati attraverso una rete, entrando nel dominio della programmazione distribuita

Gestire la concorrenza

Possiamo utilizzare:

- **Funzionalità dell'ambiente:** meccanismi forniti principalmente dal sistema operativo, che facilitano l'esecuzione e la comunicazione tra gli agenti
- **Funzionalità dei linguaggi di programmazione:** costrutti linguaggi che permettono di espandere la programmazione da un paradigma sequenziale a uno concorrente o distribuito

Concorrenza vs parallelismo

La concorrenza si riferisce alla capacità di far avanzare più attività, come processi o thread, nel tempo. Il parallelismo, invece, descrive la **capacità di eseguire più attività simultaneamente da esecutori differenti**.

Possiamo dividerlo in:

- **Concorrenza senza parallelismo:** si verifica su un singolo core con multiprogrammazione
- **Concorrenza con parallelismo:** si verifica su sistemi multicore con multiprogrammazione
- **Parallelismo dei dati:** diversi core eseguono la stessa operazione su sottoinsieme diversi dei dati
- **Parallelismo delle attività:** diversi core eseguono attività differenti su dati comuni

Legge di Amdahl

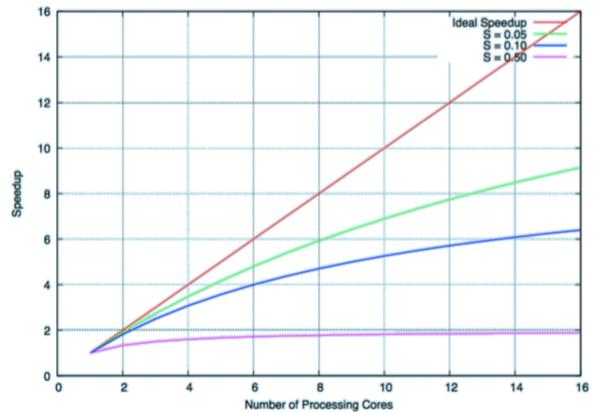
La legge di Amdahl fornisce una **stima del miglioramento delle prestazioni** ottenibile aggiungendo core a un'applicazione con componenti sia sequenziali che parallele.

Il guadagno è limitato dalla formula:

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

Dove:

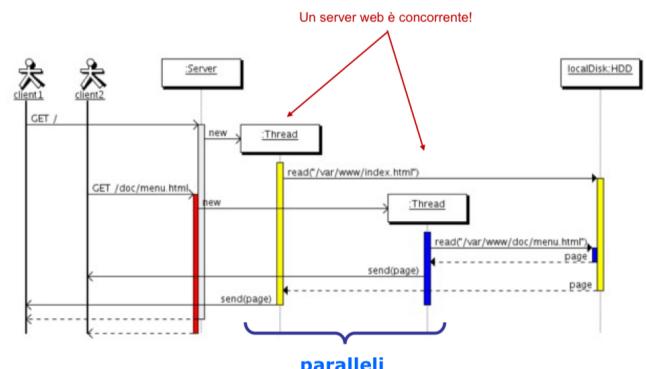
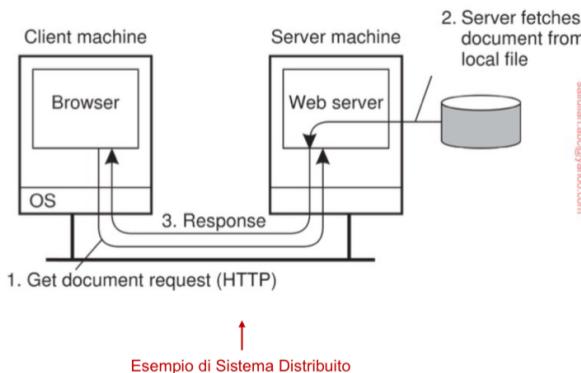
- **S**: porzione dell'applicazione che deve essere eseguita sequenzialmente
- **N**: numero di core



Es. 75% applicazione può essere parallelizzato ($S=25\% = 0,25$), con $N = 2$ core, si può raggiungere una velocità fino a 1,6 volte maggiore rispetto a un singolo core

Programmazione concorrente

La programmazione concorrente implica lo **sviluppo di programmi che includono più flussi di esecuzione**, come processi o thread. Questo approccio è vantaggioso per diversi motivi:



- Esempio di Sistema Distribuito
- Per sfruttare al meglio i processori multi-core moderni
 - Per evitare di bloccare l'esecuzione di un'applicazione durante l'attesa di operazioni I/O
 - Per sfruttare i programmi (interagiscono con ambiente, controllano diverse attività, gestiscono vari tipi di eventi, forniscono funzionalità a un utente umano, etc) in modo più efficace

Processi

Un sistema operativo gestisce l'esecuzione di numerosi programmi su un singolo sistema di elaborazione, un approccio noto come **multiprogrammazione**. Spesso, il numero di programmi in esecuzione supera il numero di CPU disponibili. Per gestire ciò, l'OS usa un'astrazione chiamata *processo*. Un processo è un'entità attiva che il sistema operativo definisce per eseguire un programma. Generalmente, si presume che un processo operi in modo sequenziale.

Programma vs processo

È importante fare questa distinzione.

- Un **programma** è un insieme passivo di istruzioni, solitamente contenute in un file sorgente o eseguibile
 - Un singolo programma può generare più processi, sia da utenti diversi che dallo stesso utente che esegue il programma più volte, anche contemporaneamente
- Un **processo** è un'entità attiva: è l'esecutore di un programma o, in altre parole, un programma in esecuzione. Quando un programma è caricato in memoria ed attivato, diventa un processo che contiene almeno un flusso di controllo attivo.

Multiprogrammazione e multitasking

Gli obiettivi principali dell'OS includono la *massimizzazione dell'utilizzo della CPU*, e la fornitura di un'illusione che *ogni processo abbia una CPU dedicata*. Quest'ultima è un'astrazione utile per chi sviluppa programmi.

Due tecniche adottate negli OS sono queste due:

- **Multiprogrammazione**: mantenere più programmazioni in memoria e impedire che un programma che non è in condizione di proseguire l'esecuzione mantenga la CPU
 - L'OS mantiene in memoria i processi da eseguire, li carica, e gli assegna una memoria e una serie di altre informazioni.
 - Quando una CPU non è impegnata ad eseguire un processo, l'OS 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
 - Richiede che **tutte le immagini di tutti i processi siano in memoria** perché siano eseguibili; se i processi sono troppi, possiamo usare la tecnica dello **swapping** o *memoria virtuale*. Queste tecniche aumentano il **grado di multiprogrammazione**

- **Multitasking:** è un'estensione della multiprogrammazione dove **la CPU viene sottratta periodicamente al programma in esecuzione**, ed assegnata ad un altro programma
 - Tutti i processi progrediscono in maniera continuativa nella propria esecuzione, anziché solo nei momenti in cui il programma che detiene la CPU si mette in attesa
 - Il suo obiettivo è di dare l'illusione del **parallelismo**: si ha una quantità predeterminata (quantum) di tempo di CPU, durante il quale il task può essere eseguito

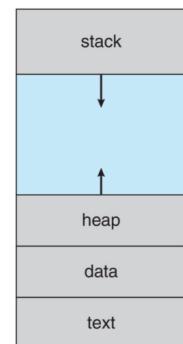
Struttura di un processo

Gli OS forniscono delle chiamate di sistema attraverso le quali i processi possono essere creati, terminati o gestiti.

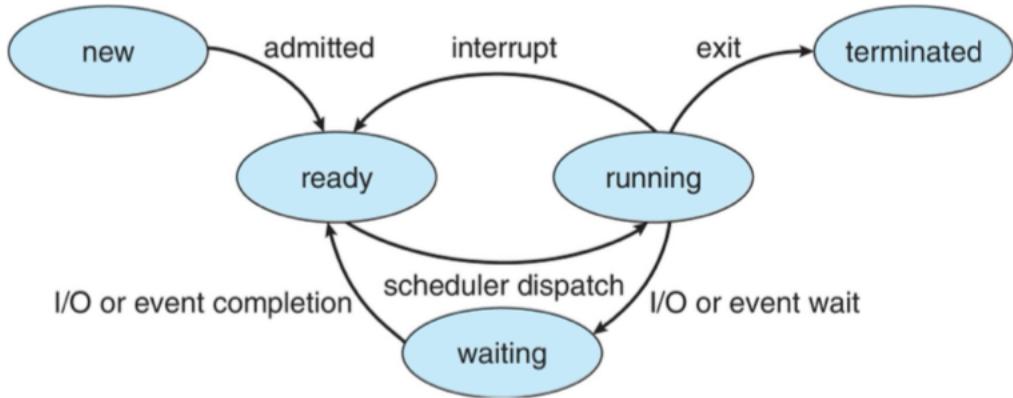
All'avvio, l'OS genera alcuni *processi fondamentali* da cui tutti gli altri processi, sia utente che di sistema, vengono divelti (da cui derivano). Questi processi primordiali sono la base della **gerarchia dei processi**.

Un processo è composto da diverse parti:

- Lo stato dei **registri** del processore, incluso il Program Counter
- Lo stato della **regione di memoria centrale** usato dal programma, o **immagine** del processo
 - Codice del programma
 - Sezione dati con le variabili globali
 - Stack delle chiamate con parametri, variabili locali e indirizzo di ritorno
 - Heap con memoria allocata dinamicamente
 - *Stack e heap hanno una dimensione dinamica durante l'esecuzione*
- Lo **stato del processo** (nuovo, in esecuzione, in attesa, pronto, terminato)
 - **Nuovo:** creato ma non in esecuzione
 - **Pronto:** può essere eseguito e attende una CPU
 - **In esecuzione:** le istruzioni sono eseguite da una CPU
 - **In attesa:** in pausa, in attesa di un evento, come un completamento I/O
 - **Terminato:** ha completato l'esecuzione



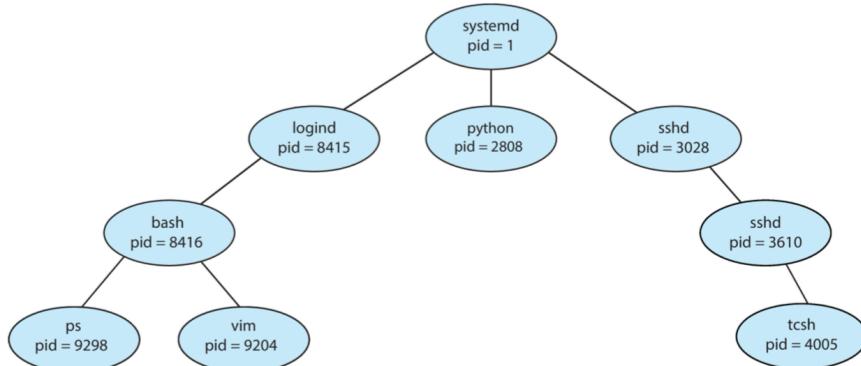
- Le **risorse del sistema operativo in uso** (file, semafori, memoria condivisa)



Gerarchia

Infatti, i processi sono tipicamente organizzati in una struttura gerarchica; un *processo padre* può generare *processi figli*, che a loro volta possono diventare padri di altri processi, formando così un albero di processi.

Le risorse di ciascun processo possono essere condivise *completamente*, *parzialmente* o *per nulla*. Inoltre, quando si crea un processo figlio, può essere un duplicato del padre, o può eseguire un programma differente.



Terminazione

Un processo deve richiedere **esplicitamente** la propria terminazione al sistema operativo. Una volta fatto questo, un processo padre può attendere la terminazione dei figli, o *farzlarla* per vari motivi (uso eccessivo di risorse, la non necessità delle funzionalità del figlio, etc).

Alcuni OS non permettono ai processi figli di continuare dopo la terminazione del padre, portando a una terminazione a cascata.

Process Control Block

Detto anche **task control block**, è la struttura dati del kernel che contiene tutte le informazioni relative ad un processo:

- Process state: ready, running, ...
- Process number: identifica il processo
- Program counter: contenuto del registro “istruzione successiva”
- Registers: contenuto dei registri del processore
- Informazioni relative alla gestione della memoria, I/O, scheduling ed accounting

Scheduling dei processi

Lo scheduler dei processi **sceglie il prossimo processo da eseguire tra quelli in stato ready**. Mantiene diverse code di processi:

- **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

Durante la loro vita, i processi migrano da una coda all’altra, a seconda dello stato del processo stesso.

Commutazione di contesto

Quando la CPU deve passare dall’esecuzione di un processo a quella di un altro processo avviene un **context switch**; la commutazione di contesto viene effettuata dal **dispatcher** che *salva il contesto del processo da interrompere nel suo PCB, e carica il contesto del processo da eseguire dal suo PCB*.

Il tempo necessario per il context switch è **overhead**: non viene eseguito alcun lavoro utile, dove per lavoro utile si intende l’esecuzione di un programma utente. Più complesso è l’OS, più è complesso il contesto, e più tempo occorre per un context switch.

Comunicazione inter-processo

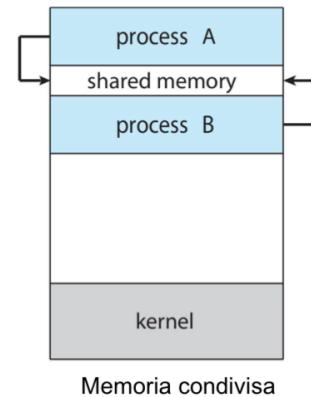
I processi possono essere indipendenti, o cooperare. Un processo **coopera** se il suo comportamento influenza, o è influenzato dal comportamento di uno o più altri processi. Per permettere ai processi di cooperare, il sistema operativo deve mettere a disposizione primitive di **comunicazione inter-processo** (IPC), di cui sono di due tipi:

IPC tramite memoria condivisa

Viene stabilita una zona di **memoria condivisa** tra i processi che intendono comunicare. La comunicazione è controllata dai **processi che comunicano**.

Un problema importante è permettere ai processi che comunicano tramite memoria condivisa di sincronizzarsi, poiché un processo non deve leggere la memoria condivisa mentre l'altro la sta scrivendo.

Allo scopo, i sistemi operativi mettono a disposizione ulteriori primitive per la sincronizzazione.

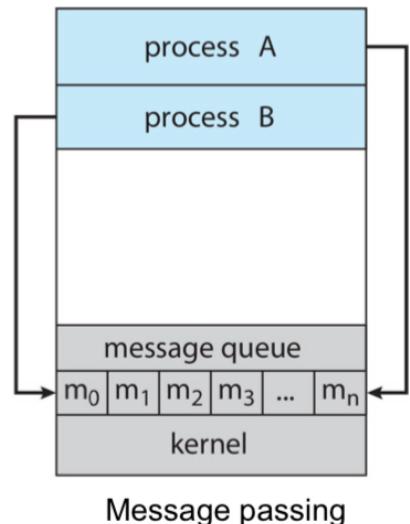


IPC tramite message passing

Permettono ai processi **sia di comunicare che di sincronizzarsi**; i processi comunicano tra di loro senza condividere memoria, attraverso la memoria del sistema operativo, o di un altro processo. Devono essere disponibili:

- Un'operazione *send(message)*
- Un'operazione *receive(message)*

Per comunicare, due processi devono **stabilire un link di comunicazione tra di loro**, e poi scambiarsi messaggi; questo modello di comunicazione è la normalità nel caso di programmazione distribuita.

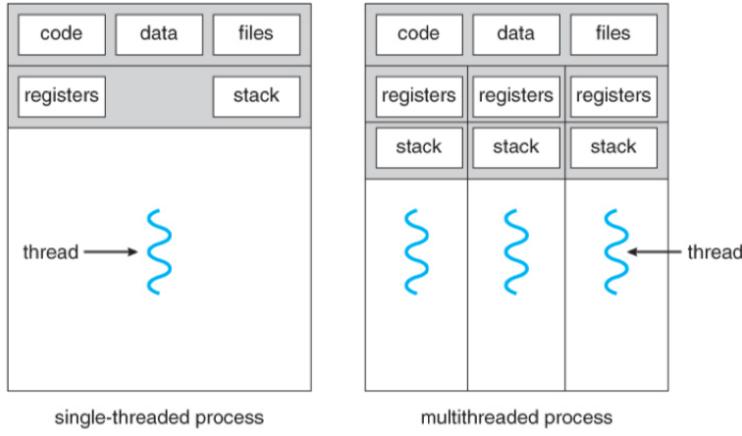


Multithreading

Se supponiamo che un processo possa avvalersi di *molti processori*, *più istruzioni possono essere eseguite concorrentemente*, e quindi il processo può avere più thread di esecuzione concorrenti. Per poter consentire l'esecuzione di diversi flussi di controllo in parallelo, sono stati introdotti i **thread**.

I thread di uno stesso processo **condividono la memoria globale** (data and heap), **la memoria contenente il codice** (code section) e **le risorse ottenute dall'OS**. Ogni thread di uno stesso processo, però, deve avere un proprio stack, altrimenti le chiamate a subroutine di un thread interferirebbero con quelle di un altro thread concorrente.

Un programma **sequenziale** ha un singolo flusso di controllo, mentre un programma **multithreaded** ha più flussi di esecuzione che gli permettono di svolgere diversi compiti allo stesso tempo, e devono spesso coordinarsi.



Abbiamo vari vantaggi:

- **Condivisione di codice e dati:** i thread di un processo condividono lo spazio di memoria del processo
- **Economia:** creare un thread richiede meno risorse che creare un processo
- **Migliori tempi di risposta:** l'applicazione può eseguire un'operazione lunga in un thread a parte e continuare a rispondere agli input utente
- **Scalabilità:** il Multithreading in una macchina con più processori aumenta il grado di parallelismo
- **Ottimizzazione delle risorse:** se un thread è in attesa di una operazione di I/O di un altro thread del medesimo programma può accedere alla cpu evitando il context switch

Abbiamo due tipi di thread:

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

Thread control blocks

Nei kernel multithreaded, il kernel mantiene delle strutture dati analoghe ai PCB chiamate thread control blocks, che **memorizza il contesto di un thread e le sue informazioni di contabilizzazione**. Il PCB contiene solo le informazioni di contesto e contabilizzazione comuni.

Di norma, il PCB è collegato ai TCB dei thread kernel utilizzati dal processo, e viceversa anche.

Modelli di supporto

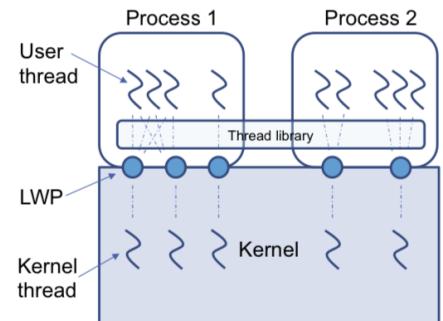
I thread a livello del kernel vengono utilizzati dalle librerie di thread per implementare i thread a livello utente di un certo processo. Abbiamo tre tipi:

- **Molti-a-uno:** i thread a livello utente di un certo processo sono implementati su un solo thread a livello kernel
 - o Usabile su ogni OS, ma se un thread utente fa una chiamata di sistema bloccante blocca tutti i thread utente, e non sfrutta la presenza di più core
- **Uno-a-uno:** ogni thread a livello utente è implementato su un singolo thread a livello kernel
 - o Permette un maggior grado di concorrenza, e di sfruttare il parallelismo nei sistemi multicore, ma ha un maggiore overhead e stress 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, e l'associazione è dinamica (stabilità da uno scheduler interno alla libreria)
 - o È complesso da implementare, e permette agli utenti di creare anche dei thread che hanno un mapping uno-a-uno con un thread a livello del kernel

Lightweight process

Un **Lightweight Process** è l'interfaccia offerta dal kernel alle librerie dei thread per usare i thread del kernel. Ogni LWP è un **oggetto astratto** associato staticamente al kernel ad esattamente un thread del kernel. La libreria dei thread può usare un LWP per *mandare in esecuzione un thread utente* (ma l'LWP può bloccarsi, o essere pre-empted dal kernel).

In più, può essere usato dalla libreria dei thread per aggiornare le *informazioni relative al contesto del processo utente* utili a prendere migliori decisioni di scheduling.



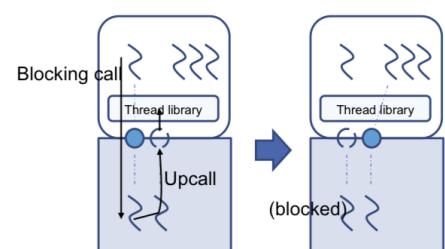
Attivazione dello scheduler

L'attivazione dello scheduler è un **modello di collaborazione tra libreria dei thread e kernel**.

Il kernel comunica alla libreria dei thread l'occorrenza dei principali eventi di scheduling sugli LWP attraverso **upcall**. Le upcall, quindi, chiamano opportune funzioni callback della libreria dei thread.

In risposta ad una upcall, la libreria dei thread può:

- Aggiornare le proprie strutture dati
- Decidere di effettuare operazioni di scheduling



Java Threads

La Java Virtual Machine offre delle astrazioni per la creazione e gestione dei thread, che sono tipicamente implementati sfruttando il modello di **threading offerto dall'OS**. La JVM mette a disposizione una libreria che permette di definire i thread in due modi principali: estendendo la classe standard `java.lang.Thread`, o implementando l'interfaccia `java.lang.Runnable` attraverso il metodo `run()`.

```
public class ThreadMain (extends Thread/implements Runnable){  
    public static void main(String args []) {  
        Thread t = Thread.currentThread();  
        Thread t2 = new ThreadMain();  
        t2.start();  
        t.setName("Thread name");  
  
    }  
    public void run() {  
        //insert here code to run at the start  
        try { Thread.sleep(2000);}   
        catch (InterruptedException e) { }  
    }  
}
```

Ogni programma Java in esecuzione è di per sé un thread, comunemente noto come **main thread**. Il metodo `main()` è il punto di ingresso del programma, e corrisponde al thread principale. Per interagire con le proprietà del thread attualmente in esecuzione, si può ottenere un riferimento utilizzando il metodo `Thread.currentThread()`, che permette anche di modificare il nome del thread ad esempio.

La prassi comune è:

1. Estendere la classe `Thread`
2. Ridefinire il metodo `run()` nella sottoclasse, inserendo il codice che il thread dovrà eseguire
3. Creare un'istanza della sottoclasse
4. Invocare il metodo `start()` sull'istanza creata, affinchè lo scheduler dell'OS riconosca il nuovo thread e lo metta in esecuzione.

Quando si utilizza l'interfaccia `Runnable`, si ha la **possibilità di definire un thread pur mantenendo la libertà di estendere un'altra classe**. Il processo di creazione di un thread tramite `Runnable` include:

1. Definire una classe che implementa `Runnable`, e dotarla di un metodo `run()` significativo
2. Creare un'istanza di questa classe
3. Creare un'istanza di `Thread`, passando al costruttore l'istanza della classe che implementa `Runnable`
4. Chiamare il metodo `start()` sull'istanza di `Thread`

“Vivicità” del thread

Un aspetto cruciale nella gestione dei thread è determinare se un thread è *alive* o *terminated*. Il metodo `isAlive()` verifica se il thread sta ancora eseguendo il metodo `run()`.

Un thread è considerato **vivo dall’invocazione di `start()` fino al completamento di `run()`**. È importante notare che `start()` può essere chiamato una sola volta per thread; un secondo tentativo di invocazione causerà un’eccezione `IllegalThreadStateException`.

Scheduling

I thread in Java sono soggetti a uno **scheduling pre-emptive**, e basato su priorità. Ciò significa che i thread con priorità più alta vengono eseguiti per primi, e lo scheduler può interrompere un thread per assegnare la CPU a un altro processo. Il metodo `yield()` segnala allo scheduler che *il thread corrente è disposto a cedere il suo utilizzo corrente del processore*. Inoltre, un thread può entrare nello stato **blocked** quando effettua una richiesta di I/O, e l’OS lo mette in attesa fino al completamento dell’operazione.

Sospensione e cancellazione

Il metodo **`Thread.sleep()`** permette di sospendere l’esecuzione di un thread per un numero specificato di millisecondi.

Internamente, qualche volta Java potrebbe anche fare semplicemente un loop per un determinato tempo, mentre non fa niente.

La **cancellazione** di un thread si riferisce alla sua terminazione prematura. Esistono due modelli principali:

- *Cancellazione immediata*: il thread viene interrotto immediatamente
- *Cancellazione differita*: il thread controlla periodicamente se deve terminare, permettendo una chiusura più ordinata.

Per interrompere un thread, si utilizza di solito il metodo **`interrupt()`**, che imposta un flag di interruzione nel thread di destinazione. È importante verificare periodicamente lo stato di interruzione con `isInterrupted()`, o `Thread.interrupted()`, e, se necessario, uscire dal metodo `run()`.

Thread pool

Il thread pool è un’importante astrazione; si tratta di un oggetto che **contiene al suo interno una serie di thread**, spesso definiti *thread vuoti*, pronti ad essere utilizzati. Questo approccio è particolarmente intelligente perché permette di assegnare a ciascun thread un lavoro specifico da svolgere, secondo un’interfaccia prestabilita che deve essere implementata.

Ha i seguenti vantaggi:

- Più rapido perché evita il sovraccarico di lavoro associato alla creazione e distruzione di thread
- Il numero di thread che possono esistere contemporaneamente è limitato dalla dimensione del pool, che aiuta a prevenire il consumo eccessivo di risorse e potenziali problemi di astrazioni
- Separazione tra compiti da svolgere e meccanica della loro creazione: permette di adottare diverse strategie di esecuzione, rendendo il sistema più flessibile e scalabile

Programmi concorrenziali

I programmi sequenziali sono noti per le loro proprietà deterministiche. Questo significa che, seguendo il loro flusso di esecuzione, possiamo **prevedere con precisione il risultato, e identificare con chiarezza eventuali anomalie o bug**. Nella programmazione concorrenziale, abbiamo la mancanza di questo determinismo. Questa caratteristica rende difficile tracciare il flusso di esecuzione, e può mascherare la presenza di bug.

Differenziamo il tipo di interazione tra i processi:

- **Cooperazione:** interazioni prevedibili e volute che si verificano in momenti e condizioni specifici, creati da noi. Questo tipo di sincronizzazione è *diretto* o *esplicito*
 - o Avvengono tramite scambio di informazioni, che possono essere anche semplici segnali
- **Competizione:** si verifica quando diversi processi tentano di accedere a una risorsa condivisa, come una porzione di memoria; la sincronizzazione è *indiretta* o *implicita*
 - o Gli agenti competono per l'accesso a una risorsa condivisa, richiedendo politiche di accesso ben definite
- **Interferenze:** interazioni imprevedibili e indesiderate, spesso risultato di errori di programmazione e dipendenti dal timing

Meccanismi di sincronizzazione

Sono essenziali per garantire l'ordine corretto delle operazioni nei sistemi distribuiti. Essi assicurano che determinate azioni siano eseguite in una sequenza specifica.

Quando usiamo un modello basato sulla **memoria condivisa**, è cruciale implementare la **mutua esclusione**; questo si ottiene creando delle **regioni critiche** nel codice, che non possono essere accedute da più thread contemporaneamente, per evitare condizioni di gara.

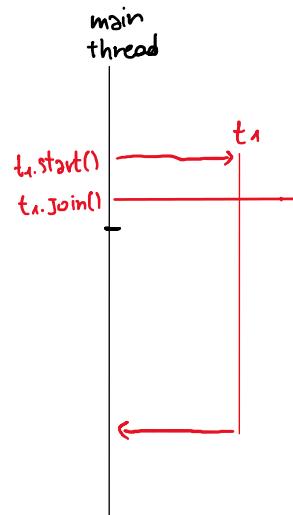
Usiamo anche la **sincronizzazione su condizione**, per sospendere l'esecuzione di un thread fino a quando non si verifica una certa condizione, permettendo così un controllo fine sul flusso di esecuzione.

Nel caso del modello basato su **scambio di messaggi**, la sincronizzazione avviene implicitamente attraverso le primitive `send` e `receive`. A seconda che l'accesso sia bloccante o non bloccante, i thread possono essere *sospesi in attesa che si verifichi una condizione specifica*, prima di poter accedere alla coda dei messaggi.

Join

Questo metodo nella classe `Thread` permette a un **thread chiamante di attendere fino al completamento del thread chiamato**. Durante questo periodo di attesa, il thread *chiamante* non ha accesso alla CPU, e rimane in stato di *wait* fino a quando non si verifica l'evento di terminazione del thread chiamato.

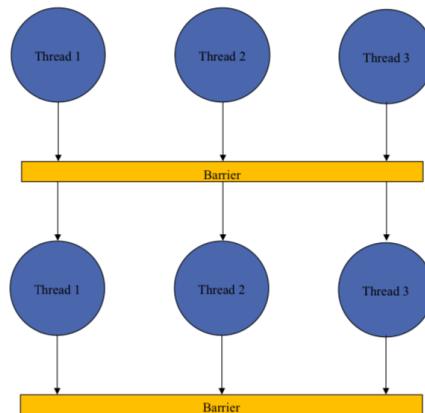
Se il thread chiamato viene interrotto, il metodo lancia un'**InterruptedException**. Se il thread chiamato è già terminato, o non è stato avviato, la chiamata ritorna immediatamente.



Barriere

Sono strumenti di sincronizzazione utilizzati quando abbiamo più thread che lavorano su compiti suddivisi in fasi. È necessario che **tutti i thread completino la prima fase prima di poter procedere alla seconda**. Il primo thread che finisce deve *attendere gli altri*; solo quando l'ultimo thread raggiunge questo punto, la barriera viene rimossa, e possono tutti procedere.

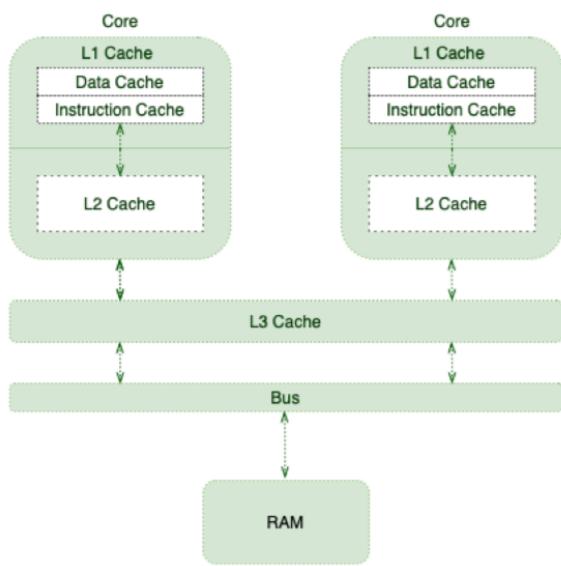
Per implementarla, possiamo utilizzare oggetti come *Barrier*, *Latch*, o un contatore condiviso che tiene traccia del numero di processi. È fondamentale che il **contatore sia atomico**, ovvero non interrompibile, per mantenere l'integrità del processo.



Race condition

Rappresentano un problema significativo nei sistemi concorrenti, in particolare quando più thread **modificano una stessa area di memoria**. Se non gestite correttamente, possono portare a uno stato alterato della risorsa condivisa. Lo scheduler dell'OS può interrompere l'esecuzione di thread diversi, in modo tale da rendere la risorsa condivisa inconsistente. Questo tipo di aggiornamento errato, causato da una combinazione casuale di letture e

scritture è l'**interferenza**. La soluzione a questo problema consiste nel garantire l'accesso mutuamente esclusivo mutuamente esclusivo alla risorsa.



Il problema della visibilità

I problemi di visibilità emergono in sistemi multicore, dove **modifiche effettuate da un thread su un core non potrebbero essere immediatamente visibili ad altri thread su core diversi**. Questo è dovuto alla struttura gerarchica della cache nella CPU, che include:

- **L3 cache:** condivisa tra più core
- **L2 cache:** esclusiva di ogni core
- **Data cache and instruction cache:** anch'esse esclusive.

L'uso della parola chiave **volatile** in Java garantisce che **ogni aggiornamento** a una

variabile **sia propagato attraverso i livelli di cache o alla memoria centrale**, assicurando così che ogni thread lavori con la versione più aggiornata della variabile.

Implementare la sincronizzazione

Esistono due approcci principali:

- **Attesa attiva:** adatto per sistemi multiprocessore/multicore, non richiede un cambio di contesto, ma può sprecare tempo di calcolo sulla CPU
- **Basata sullo scheduler:** prevede l'auto-sospensione dei thread, che vengono risvegliati quando si verifica l'evento atteso.

Sezione critica

Si tratta di una **porzione di codice che deve essere eseguita da un solo thread** alla volta, per prevenire inconsistenze nei dati. L'obiettivo è stabilire un *protocollo di cooperazione tra i processi che assicuri la mutua esclusione*: quando un thread è nella sua sezione critica, nessun altro thread può esserlo.

Il protocollo prevede tre fasi:

- **Sezione di ingresso:** il thread richiede l'entrata nella sezione critica
- **Sezione critica:** il thread esegue le operazioni che richiedono l'accesso esclusivo alla risorsa

- **Sezione di uscita:** il thread può, ad esempio, notificare gli altri processi della sua uscita dalla sezione critica

Per fare questa operazione, i processori moderni offrono un'operazione atomica **test_and_set(B)**, che prende un boolean come parametro. Questa operazione verifica lo stato di una variabile booleana B e, se B è *false*, la ritorna e la imposta a *true* in un ciclo di esecuzione.

I lock

È un meccanismo che utilizza una variabile logica binaria manipolabile atomicamente per **garantire l'accesso esclusivo** a una sezione critica. I metodi **lock()** e **unlock()** permettono di acquisire e rilasciare il lock in maniera atomica.

Quando un thread acquisisce un lock, tutti gli altri thread che tentano di acquisire lo stesso lock vengono bloccati, fino a quando il lock non viene rilasciato.

Locking pessimistico

Se un thread non riesce ad eseguire un lock, allora viene sospeso, e può avvenire un **context switch** (risvegliare un thread presenta un costo), e comporta altre due conseguenze anche:

- 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; infatti, quando un thread acquisisce un lock, nessun altro thread che ha bisogno di quel lock può eseguire.
- Richiedere un lock costa tempo; in alcuni casi la JVM deve interagire con l'OS, e questo tempo dipende molto anche dallo scheduler.
- Il thread che acquisisce lock può terminare il proprio quanto di tempo **mentre è in possesso di lock**; tutti gli altri thread in attesa, quindi, rimarranno in attesa che un thread terminato si liberi del lock.

Il processo di locking pessimistico aggiunge overhead quando la contesa per le risorse non è frequente; crea una situazione rigida di “*happens before*”, dove blocca l'accesso a una certa variabile condivisa prima, anche se la quantità di thread che fanno uso di questa risorsa sia bassa.

Optimistic Retrying

È un'altra alternativa al locking. Questo approccio è basato sull'idea che sia *più efficiente riprovare un'operazione piuttosto che chiedere permesso*. Non richiede sincronizzazione in

lettura, e utilizza un processo di **read-modify-write** per le scritture, con supporto delle istruzioni dei processori moderni per la collision detection.

I semafori

I semafori sono strumenti di sincronizzazione più generali rispetto ai lock. Un semaforo è rappresentato da una **variabile intera**, e gestisce l'accesso a risorse condivise tramite due operazioni atomiche: *acquire()* e *release()*. L'operazione *acquire()* attende che il valore del semaforo sia maggiore di zero prima di decrementarlo, permettendo così l'accesso alla risorsa.

Al contrario, *release()* incrementa il valore del semaforo, segnalando che la risorsa è nuovamente disponibile. È importante che ogni semaforo mantenga una coda di thread in attesa per gestire l'accesso in modo ordinato.

Synchronized

In Java, la parola chiave **synchronized** viene utilizzata per creare un lock implicito su un oggetto, garantendo che **un solo thread alla volta possa eseguire un metodo** o blocco di codice marcato come *synchronized*. Questo meccanismo assicura che, se un thread è all'interno di un metodo synchronized, altri thread che tentano di eseguire lo stesso metodo o un altro metodo synchronized sull'oggetto verranno messi in attesa fino al completamento del metodo da parte del primo thread. Synchronized non fa parte della segnatura della firma di un metodo, e dunque, **una classe derivata può ridefinire un metodo synchronized e viceversa**.

La sequenzialità introdotta dai blocchi synchronized stabilisce una relazione di tipo *happens-before*, assicurando che le operazioni all'interno della sezione critica di un thread avvengano prima di altre operazioni. I blocchi synchronized consentono di **definire sezioni critiche più piccole all'interno di un metodo**, offrendo una granularità maggiore nel controllo dell'accesso alle risorse.

Lavorando con variabili locali o chiamate a elementi locali, non si verifica il problema delle race condition. Tuttavia, quando si protegge l'accesso a variabili con scope di oggetto, è essenziale usare meccanismi di sincronizzazione per evitare accessi concorrenti non sicuri.

Monitor

Il monitor è una primitiva di sincronizzazione di alto livello, superiore ai lock e ai semafori, che **incapsula il concetto di oggetto**. Questo oggetto specifica cosa deve essere fatto, ma lascia indefinito il *come*. Si tratta di un tipo di dato astratto che racchiude variabili interne, accessibili esclusivamente attraverso un insieme di procedure definite dal monitor stesso.

Il monitor garantisce che **un singolo thread o processo possa accedere** alle risorse condivise in un dato momento, realizzando così la mutua esclusione. In Java, questo oggetto si concretizza nell'oggetto *Object*, che implementa la sincronizzazione attraverso metodi sincronizzati.

Il monitor diventa attivo quando un metodo sincronizzato è in esecuzione all'interno dell'oggetto. Questo permette di controllare l'accesso alle risorse solo quando determinate condizioni sono soddisfatte. Per gestire queste condizioni, il monitor mantiene delle code di thread in attesa che le condizioni si verifichino. Per n metodi synchronized, vengono creati soltanto uno, associato all'oggetto in questione.

Le primitive *wait* e *notify* giocano un ruolo cruciale: permettono ai thread di mettersi in attesa quando una condizione specifica si verifica e di essere notificati quando questa condizione cambi. Questi strumenti offrono al programmatore la capacità di:

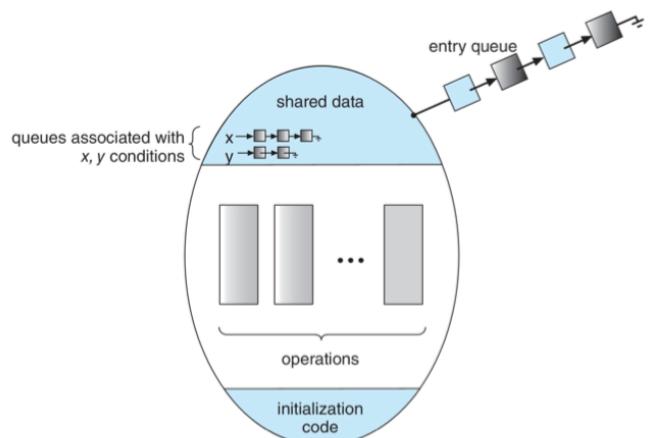
- Assicurare la **mutua esclusione** nell'accesso a risorse specifiche
- Bloccare un **thread in attesa** che una condizione si verifichi
- Notificare i **thread in attesa** che le condizioni relative alla risorsa sono cambiate

Questi due metodi svolgono la seguente funzione:

- **Wait()** mette in stato di attesa il processo/thread corrente, e lo forza a lasciare il monitor. *Rilascia il lock sull'oggetto.*
- **Notify()** rende ready un processo/thread che aveva invocato wait(), se esiste
- **notifyAll()** rende ready tutti i processi/thread che avevano invocato wait() su un monitor
- Questi metodi possono essere eseguiti solo all'interno di un metodo **synchronized**.

Un oggetto non solo implementa la sincronizzazione ma anche i metodi *wait*, *notify* e *notify all*; ogni istanza di classe è implicitamente un monitor se contiene metodi sincronizzati.

Questi metodi sono parte della classe *Object*, e quindi ereditati da ogni classe. Mentre la gestione dei lock intrinseco è trasparente, la gestione dei meccanismi di accesso condizionato alle risorse è responsabilità del programmatore.



Esempio produttore-consumatore

In questo modello, abbiamo agenti/thread che si dividono in due categorie: i **produttori**, che generano dati e li inseriscono in una coda (buffer) di dimensioni limitate, e i **consumatori**, che prelevano i dati dalla coda.

Però, in questo metodo, abbiamo una serie di problemi:

- **Buffer limitato:** il buffer ha una capacità finita: i produttori non possono aggiungere elementi se è pieno, e i consumatori non possono prelevare se è vuoto
- **Comunicazione asincrona:** i messaggi vengono inviati e ricevuti in momenti diversi, richiedendo l'implementazione di una coda di messaggi
- **Send e receive:** operazioni non bloccanti, ma possono diventare bloccanti se la coda è limitata e raggiunge i suoi limiti

```
public class Buffer {  
    public synchronized void insert(char ch){  
        //finchè il buffer è pieno attendi  
        //inserisci nel buffer  
        //avvisa chi è in attesa sul buffer  
  
    public synchronized char delete(){  
        //finchè il buffer è vuoto attendi  
        //cancella dal buffer  
        //avvisa chi è in attesa sul buffer
```

I metodi di inserimento e cancellazione nel buffer possono essere **sincronizzati** per gestire l'accesso concorrente. Possiamo implementarli in questo modo:

- Se il buffer è pieno, il produttore **deve attendere** fino a quando non ci sarà spazio disponibile
- Se il buffer è vuoto, il consumatore **deve attendere** fino a quando non ci sarà qualcosa da consumare

Qua di fianco c'è un esempio di come potrebbe uscire.

Altri approcci

Oltre ai metodi tradizionali, abbiamo due altri approcci:

- **Variabili atomiche:** queste variabili supportano operazioni atomiche, che sono eseguite in un singolo passo indivisibile, garantendo così l'integrità dei dati in ambienti multi thread.
- **Strutture dati thread-safe:** alcune librerie offrono strutture dati progettate per essere thread-safe, ovvero possono essere usate da più thread contemporaneamente senza causare problemi di concorrenza. Queste strutture sono pre-implementate con meccanismi che assicurano l'assenza di interferenze tra i thread.

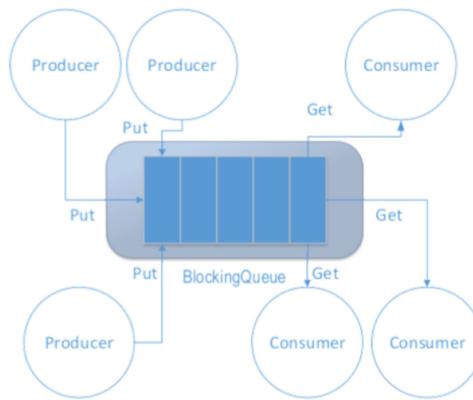
Coda bloccante

Un esempio di oggetto sincronizzato è la **coda bloccante**, una struttura dati fornita dal package `java.util.concurrent`. È una coda che supporta operazioni di inserimento e rimozione in modo thread-safe:

- **Put():** permette di inserire un elemento nella coda; se la coda è piena, il thread che invoca put() viene messo in attesa fino a quando non si libera spazio

- **Take()**: permette di rimuovere e restituire un elemento dalla coda; se la coda è vuota, il thread che invoca take() viene messo in attesa fino a quando non viene inserito un elemento.

Usando questo oggetto, quindi, non è necessario gestire manualmente la sincronizzazione del buffer, poiché la coda stessa si occupa di gestire la mutua esclusione e le condizioni di attesa.



Variabili atomiche

In un ambiente multi-threaded, dove diversi thread operano su una singola risorsa, un altro metodo è l'**accesso atomico alla risorsa**. Nel package `java.util.concurrent.atomic` sono definite una serie di classi che supportano le operazioni atomiche.

Esistono variabili atomiche quali `AtomicInteger`, che offrono un'alternativa alle operazioni di lock tradizionali. Sono “variabili volatile migorate” che **incapsulano lo stato di una variabile**, e forniscono operazioni atomiche per *manipolare e i valori senza richiedere lock*, sfruttando le operazioni atomiche dell’hardware quando disponibili.

L’atomicità può essere:

- **Reale** quando un’operazione è eseguita da una singola istruzione di CPU
- **Virtuale** quando il thread opera come se avesse accesso atomico alla variabile.
Questo è concettualmente simile a un *monitor*, e, sebbene dietro le quinte si utilizzino meccanismi per implementare la mutua esclusione, il thread non percepisce interruzioni.

Alcune classi della libreria `concurrent` hanno delle performance superiori in alcune condizioni alle loro alternative bloccanti. Per esempio, piuttosto di una `HashMap` possiamo usare una `ConcurrentHashMap`, o al posto di una `Queue` possiamo usare una **BlockingQueue** o **ConcurrentLinkedQueue**.

Algoritmi non bloccanti

Gli algoritmi non bloccanti sono meccanismi avanzati che permettono l'accesso **thread-safe** a risorse condivise senza l'uso di lock. Questi algoritmi sono più complessi, ma possono offrire prestazioni superiori in certe condizioni.

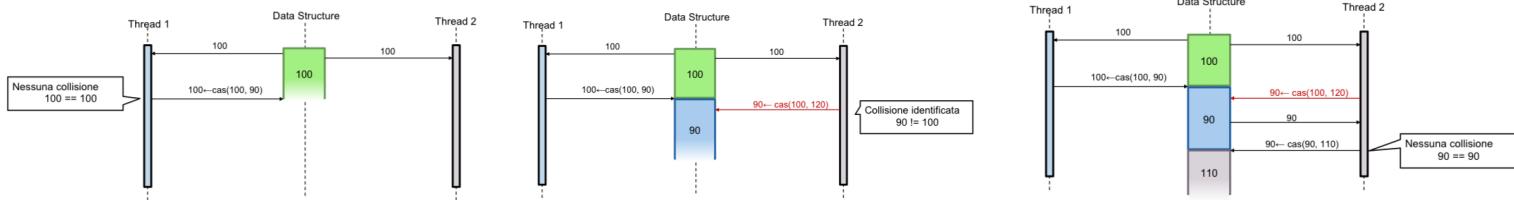
Compare-and-swap

Questo meccanismo è una funzione atomica cruciale per l'implementazione; è disponibile su alcuni processori, e prende in ingresso **V** (posizione di memoria), **E** (un valore atteso), e **N** (un nuovo valore). La funzione CAS aggiorna atomicamente la posizione **V** al nuovo valore **N**, soltanto se **il valore presente in V corrisponde al valore atteso E**.

Se l'aggiornamento ha successo, il valore restituito è uguale al valore atteso; altrimenti, non c'è stato aggiornamento.

```
function cas(v: pointer to int,
            e: int, n: int) returns int{
    o <- &v
    if o == e
        *v <- n
    return o
}
```

La variante **compare-and-set** opera similmente al CAS, ma restituisce un valore booleano indicando il successo dell'operazione.



Il problema di incrementare una variabile intera condivisa `c`, evitando race condition ma con meno overhead rispetto al blocco `synchronized`, può essere affrontato con un approccio **lock-free**. Questo approccio utilizza l'operazione atomica **Compare-and-Swap**, che non può essere interrotta. Se l'operazione CAS non fallisce, la variabile sarà aggiornata visibilmente per tutti i thread, altrimenti l'operazione viene ritentata fino al successo. Tuttavia, esiste un rischio remoto di *starvation*, dove un thread potrebbe non riuscire mai ad incrementare la variabile. Nonostante ciò, il contatore atomico risulta circa **due volte più veloce dell'implementazione con synchronized**.

Aggiornamento di oggetti complessi

Per mantenere una caratteristica invariante tra due campi, come un intervallo tra due numeri interni denotato matematicamente da $[i, u]$, si può usare il CAS in maniera non bloccante.

Se due thread eseguono `setLower` e `setUpper` per impostare l'estremo minore e superiore, entrambi possono avere successo ed invalidare l'invalidante, portando a **lower > upper**. Per

evitare ciò, si ricorre a **helper classes** che salvano entrambi i campi come **AtomicReferences<T>**, per verificare l'aggiornamento dell'oggetto usando compare-and-set.

L'idea è di trasformare l'aggiornamento multiplo in un aggiornamento unico, utilizzando l'**optimistic retrying**. Come esempio, teniamo traccia di una coppia di variabili.

Si crea una classe helper, **NumberRange**, che non ha due variabili atomiche ma una sola che fa riferimento agli oggetti di classe **IntPair**, permettendo di accedere a una coppia di valori. Si verifica che l'invariante sia mantenuto:

- Se *lo* è: si crea un nuovo riferimento atomico a IntPair
- Se il nuovo valore di *lower* è maggiore del vecchio valore: si sta tentando un'operazione non valida
- Altrimenti, si crea un nuovo oggetto IntPair, e si utilizza CompareAndSet per aggiornare il riferimento se non ci sono state modifiche intermedie.

Un esempio di codice potrebbe essere:

```
public class NumberRange{  
    private final AtomicReference<IntPair> pair;  
    public NumberRange (int lower, int upper) {  
        if (lower > upper) { throw new IllegalArgumentException(); }  
        else {  
            pair = new AtomicReference<IntPair>(new IntPair(lower, upper));  
        }  
    }  
  
    Public void setLower (int new lower){  
        IntPair oldPair, newPair;  
        do {  
            oldPair = pair.get();  
            if (new lower > oldPair.getU()) throw new IllegalArgumentException();  
            newPair = new IntPair(newLower, oldPair.getU());  
        }  
        While (!pair.compareAndSet(oldPair, newPair));  
    }  
}  
  
public class IntPair {  
    private final int l; private final int u;  
    IntPair (int l, int u) { this.l = l; this.u = u; }  
    int getL() { return l; }  
    int getU() { return u; }  
}
```

Classi atomiche

Java fornisce **12 classi contenitori** per scalari, vettori e riferimenti, che sono thread-safe e supportano operazioni atomiche di aggiornamento come *compare-and-set*, *get*, *set* e *operazioni di aritmetica*. Le implementazioni delle variabili atomiche dipendono dalla tecnologia disponibile sulla piattaforma. Un esempio di metodi utilizzabili con *AtomicInteger*:

- *get()*: restituisce il valore corrente della variabile
- *set(int newValue)*: aggiorna il valore corrente della variabile

- `compareAndSet(int expected, int update)`: aggiorna atomicamente il valore della variabile se il valore corrente è uguale ad `expected`
- `getAndSet(int newValue)`: aggiorna atomicamente il valore corrente della variabile, e restituisce il valore contenuto precedentemente
- `getAndIncrement()`: incrementa atomicamente il valore corrente della variabile, e restituisce il valore contenuto precedentemente

Deadlock e liveness

Prima di iniziare, distinguiamo i concetti di **safety** (nessuno stato scorretto è raggiungibile dall'applicazione), e **liveness** (gli agenti riescono sempre a progredire nella loro elaborazione). Questa 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**.

Diamo due definizioni di liveness:

- **Debole**: un sistema con diversi thread è libero da deadlock se, nonostante la competizione, **almeno un processo** riuscirà sempre ad accedere alla sezione critica, e riuscirà a progredire nella sua esecuzione
- **Stringente**: un sistema è libero da starvation se garantisce che tutti i thread riescano ad accedere alla sezione critica e progredire nella loro esecuzione

Però, possiamo avere tre problemi che si possono generare.

Deadlock

Quando ad un processo viene garantito l'accesso esclusivo, ad esempio tramite una mutua esclusione, ad una risorsa, possono crearsi situazioni di stallo chiamate **deadlock**.

Il deadlock è da evitare poiché può portare a un ciclo infinito di attese tra processi.

Questa situazione 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 per poter continuare la sua elaborazione. In pratica, si tratta di un'attesa circolare destinata a non terminare mai.

Abbiamo quattro condizioni necessarie affinchè un deadlock si verifichi:

1. **Mutua esclusione**: solo un agente concorrente per volta può utilizzare una risorsa
2. **Hold and wait/accumulo incrementale**: agenti concorrenti che sono in possesso di una risorsa possono *richiederne altre senza rilasciare la prima*. Possono mettersi, quindi, in attesa mantenendo il lock sulle loro risorse. Quindi, un processo A può detenere risorse, e contemporaneamente attendere altre risorse

3. **No preemption sulle risorse condivise:** una risorsa può essere rilasciata solo volontariamente dall'agente che detiene la risorsa, senza l'intervento di entità esterne.
4. **Attesa circolare:** deve esistere una possibile catena circolare di agenti concorrenti e di richieste di accesso a risorse condivise, tale che ogni agente mantiene bloccate delle risorse che contemporaneamente vengono richieste dagli agenti successivi.
 - a. Possiamo risolverlo ordinando le richieste di accesso alle risorse, e si richiede il lock seguendo l'ordine. Se A precede B, bisogna cercare di acquisire B solo se si detiene già A.

Abbiamo due possibili approcci per affrontarla:

- **Deadlock prevention:** può essere evitato se *si fa in modo che almeno una delle quattro condizioni richieste per il deadlock non si verifichi mai*
 - No preemption: si può implementare un meccanismo che permetta a un thread di rilevare se un altro sta mantenendo una risorsa per troppo tempo
 - Hold and wait: richiedere tutte le risorse necessarie simultaneamente, utilizzando tryLock che blocca la risorsa se disponibile, o informa se è già posseduta.
- **Deadlock removal:** lo si risolve quando ci si accorge che è avvenuto.

Starvation

La situazione di starvation si verifica quando un **agente non riesce ad accedere a una risorsa che gli viene perpetuamente negata**.

Questo può essere dovuto alle circostanze associate ad una particolare politica dello scheduler: ad esempio, con uno scheduler a code con priorità statiche, in presenza di molti processi ad alta priorità, un processo a priorità bassa potrebbe non essere mai eseguito.

Livelock

In certe situazioni, i membri di un gruppo di agenti possono non essere bloccati, ma ciononostante non progredire effettivamente, magari per diversi protocolli.

Diversi protocolli distribuiti hanno necessità di effettuare operazioni di sincronizzazione iniziale detta **handshake**.

I filosofi a cena

Mostriamo adesso un tipico problema.

Cinque filosofi sono seduti a un tavolo sul quale sono presenti cinque piatti di spaghetti, e cinque forchette. Ogni filosofo alternativamente *pensa* o *mangia*: per mangiare, un filosofo ha

bisogno di **due forchette**; le forchette **non sono condivisibili simultaneamente**. Dopo aver mangiato, un filosofo ripone sul tavolo le forchette usate. Si suppone che il piatto venga riempito da quante entità esterna non appena diventi vuoto.

Un filosofo ha il seguente comportamento: *pensa finchè la forchetta alla sinistra non diventa disponibile e prendila; pensa finchè la forchetta alla destra non diventa disponibile e prendila;* avendo entrambi le forchette mangia; rimetti sul tavolo la forchetta destra; rimetti sul tavolo la forchetta sinistra; ripeti.

Una volta che abbiamo fatto l'intero giro, tutti i filosofi hanno preso una forchetta, e quindi abbiamo raggiunto una situazione di **deadlock**.

Abbiamo due soluzioni:

- **Spezzare la simmetria:** tutti i filosofi meno uno si comportano alla stessa maniera. Viene considerato l'ordine tra le forchette, e si forza a prendere prima quella con identificativo minore.
- **Attesa casuale:** se un filosofo ha acquisito una forchetta, ma non è riuscito a procurarsi anche la seconda, può ipotizzare che si stia creando la condizione
 - Rilascia la forchetta in suo possesso, e attende un po' di tempo prima di riprovare a impossessarsi delle forchette
 - Questo non basta perché causerebbe un **livelock**: tutti rilasciano quella di destra contemporaneamente, poi tutti riprendono contemporaneamente quella di sinistra. Quindi, l'attesa deve essere casuale.
- **Rimozione di hold and wait:** ogni filosofo prende con un'operazione atomica entrambe le forchette se disponibili, e aspetti se non lo sono. Ci vuole, quindi, un mediatore che osservi lo stato delle forchette.

Message oriented communication

I **protocolli applicativi** si ergono come pilastri fondamentali, operando al di sopra del modello TCP/IP per facilitare l'interazione tra client e server. Vediamo qualche termine fondamentale:

- **Web:** sfrutta il protocollo HTTP per supportare l'interazione dinamica
- **Browser:** applicazione web sul lato client che permette agli utenti di navigare nel Web
La sua funzione principale è *interpretare il codice delle pagine web*, e *visualizzarlo in forma di ipertesto* attraverso un processo chiamato **rendering**. Questo processo è influenzato dai dispositivi utilizzati, inclusi quelli non visivi.
 - **Ipertesto:** rete di testi o pagine leggibili in maniera non sequenziale e interconnessi tramite hyperlink.
- **Pagina web:** file digitale che definisce una risorsa identificata da un URL univoco.

- **Web server:** applicazione che gestisce pagine web su un computer rendendole accessibili ai client
- **Risorse** possono avere diverse rappresentazioni o profili

URI/URL

L'URL identifica un oggetto nella rete, e specifica il protocollo per la trasmissione dei dati, composto da cinque elementi principali:

protocollo://indirizzo_IP[:porta]/cammino/risorsa

Un **URI** è una stringa di caratteri usata per identificare una risorsa; è costituita dall'URL (specifico per identificare le location primarie) e l'**URN** (inteso come una persistente, indipendente dalla location, identificatore delle risorse).

I dati testuali sono espressi in linguaggi standard come HTML (per definire la struttura dei contenuti e la loro impaginazione), e XML/JSON (focalizzati sui dati e la loro struttura). I dati possono essere non testuali, e le pagine web possono contenere del codice fatto in **linguaggi di scripting**, per arricchire l'interazione, e rendere le pagine attive.

Protocollo http

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

Il protocollo di livello applicativi per il web usa il modello client/server:

- **Client:** browser che richiede, riceve e mostra oggetti web
- **Server:** web server che invia oggetti in risposta alle richieste

HTTP usa **TCP**: il client inizia una connessione TCP, creando una socket, verso il server sulla porta 80; il server accetta la connessione TCP dal client. Poi, vengono scambiati i messaggi tra il browser e il web server.

Una proprietà importante di http è che è **stateless**: il server non mantiene informazione sulle richieste precedenti del client; questo vuol dire che ogni richiesta deve contenere tutte le informazioni necessarie per la sua esecuzione.

Formato dei messaggi

I messaggi http hanno due tipi di messaggi: *request* e *response*. Sono codificati in ASCII, e hanno la stessa struttura.

Request message

GET /somedir/page.html HTTP/1.1 → Request line (GET, POST, etc)

Host: www.someschool.edu

Connection: close → Richiede chiusura della connessione alla fine

User-agent: Mozilla/4.0 → Identifica il richiedente

Accept: text/html, image/gif, image/jpeg → Esprime i tipi di dati accettati

Accept-language: fr

Response message

HTTP/1.1 200 OK → Status line (protocol, status code, status phrase)

Connection: close → Connection closed

Date: Thu, 10 Apr 2024 19:43:15 GMT → Data della risposta

Server: Apache/1.3.0 (Unix) → Qualifica il server

Last-Modified: Mon, 22 Jun 2023 → Ultima modifica

Content-Length: 6821

Content-Type: text/html

→ Divisione + tipo di dati restituiti

Abbiamo un paio di linee guida: lo **spazio** viene usato come separatore, la prima linea è la **request line**, per far finire lo header ci aggiungiamo uno spazio di invio, l'invio è per separare le linee.

Metodi

I principali metodi usati sono:

- **Get:** restituisce una rappresentazione di una risorsa. Include eventuali parametri in coda all'URL della risorsa
 - È safe: l'esecuzione non ha effetti sul server, e la risposta può essere gestita con una cache dal client

- È idempotente: l'effetto di più richieste identiche di ottenere una risorsa è lo stesso di quello di una sola risorsa.
- Se tanti client inviano una richiesta GET alla stessa risorsa, allora **una sola servlet che le gestisce vengono attivate**, accedendo in concorrenza
- GET resource[?key=value{&key=value}] HTTP1.1
- **Post:** comunica dei dati da elaborare lato server, o crea una nuova risorsa subordinata all'URL indicata
 - L'input segue come documento autonomo
 - Non è idempotente: ogni esecuzione ha un effetto diverso, e la risposta non può essere gestita con una cache dal client
 - POST resource HTTP1.1
- **Head:** simile al metodo GET, ma viene restituito solo l'Head della pagina web

Richiesta GET

La richiesta GET recupera dati da un server. I parametri, noti come query, vengono inclusi nell'URL della risorsa:

```
GET /myCompany/orders?item="Q234"&date="2022/03" HTTP/1.1
Host: example.com
```

Richiesta POST

La richiesta POST invia dati al server per creare o aggiornare una risorsa. I dati vengono inclusi nel corpo della richiesta, detto body:

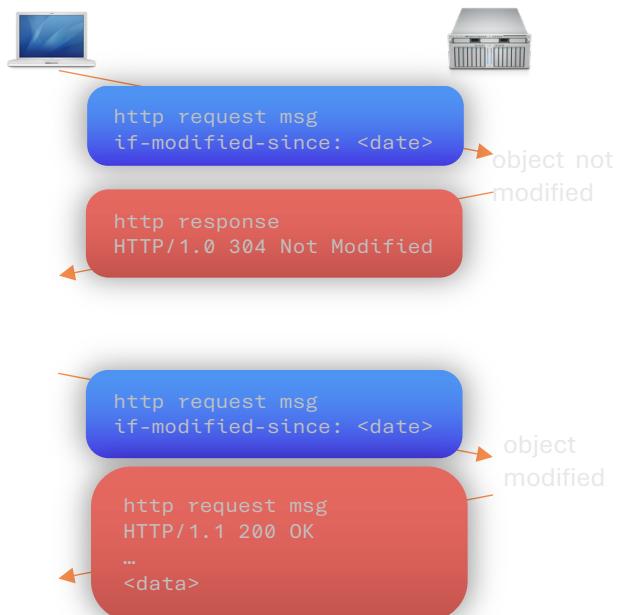
```
POST /myCompany/orders HTTP/1.1
Host: example.com
Content-Length: 59
Content-Type: application/x-www-form-urlencoded
Update=true&oldItem="Q2345"&newItem="Q6825"&date="2022/04"
```

Per entrambi le richieste, possono essere arricchite con dati semantici per migliorare la comprensione da parte delle macchine, e possiamo usare **JSON-LD** che permette di esprimere dati strutturati e collegati.

GET condizionale

È una funzionalità del protocollo di HTTP che permette di ottimizzare l'uso della banda, e ridurre il carico sul server. L'obiettivo è **evitare di inviare nuovamente oggetti che il client ha già memorizzato nella cache**.

1. Il **client** invia una richiesta http con un header che indica la data dell'ultimo oggetto memorizzato nella cache
2. Il **server**, ricevendo la richiesta, controlla se l'oggetto richiesto è stato modificato dopo la data specificata nell'header *If-Modified-Since*
3. Se l'oggetto **non è stato modificato**, il server risponde con un codice di stato **304 Not Modified**, e non include il body nella risposta
4. Se l'oggetto **è stato modificato**, il server invia una risposta con il codice di stato **200 OK**, e include il nuovo contenuto nel body della risposta



Cookie

I cookie http sono **piccoli blocchi di dati** inviati dal client al server. Il browser li memorizza, e li rinvie con richieste successive. I cookie sono spesso utilizzati per mantenere un utente loggato, o per personalizzare l'esperienza di navigazione.

L'obiettivo di associare un identificatore alla conversazione http è quello di creare uno **stato in un protocollo che è per natura stateless**, ovvero senza stato. Questo può essere gestito attraverso i cookie:

- Il **server**, dopo aver processato una richiesta http, può decidere di inviare un cookie al client all'interno della risposta http
- Il **client**, ricevendo il cookie, lo memorizza e lo include in tutti i messaggi successivi inviati al server
- Il **server**, ricevendo la richiesta con il cookie, controlla l'identificatore per l'*autenticazione* (verifica l'identità dell'utente), e *sessione di lavoro* (mantiene traccia dello stato della sessione dell'utente)

L'uso dei cookie viola il principio stateless del protocollo http, ma permette di implementare funzionalità importanti come le sessioni utente e il mantenimento dello stato attraverso le richieste.

Autenticazione

È nel controllo dell'accesso alle risorse in un protocollo http, che è per natura stateless.

Questo significa che **ogni richiesta del client deve essere autenticata** indipendentemente dalle precedenti.

- Il **client** invia una richiesta HTTP al server per accedere a una risorsa protetta
- Se il **server** richiede l'autenticazione, rifiuta la connessione e invia una risposta con il codice di stato **401 Authorization Required** e l'header *WWW-Authenticate*
- Il **client**, ricevendo il codice 401, deve includere le credenziali di login e password nell'header *Authorization* del messaggio di richiesta
- Il **server**, dopo aver ricevuto la richiesta con l'header *Authorization*, verifica le credenziali e, se sono corrette, concede l'accesso alla risorsa

Codici di stato

I codici di stato vengono utilizzati per indicare la risposta del server alle richieste inviate dai client. Possono essere:

- **1xx** – informativi: indicano che la richiesta è stata ricevuta e il processo è in continuazione
- **2xx** – successo: significano che la richiesta è stata ricevuta con successo, compresa, accettata e servita
- **3xx** – reindirizzamento: richiedono un'ulteriore azione per completare la richiesta
- **4xx** – errore del client: la richiesta contiene una sintassi errata o non può essere compresa
- **5xx** – errore del server: il server non è riuscito a soddisfare una richiesta apparentemente valida

MIME

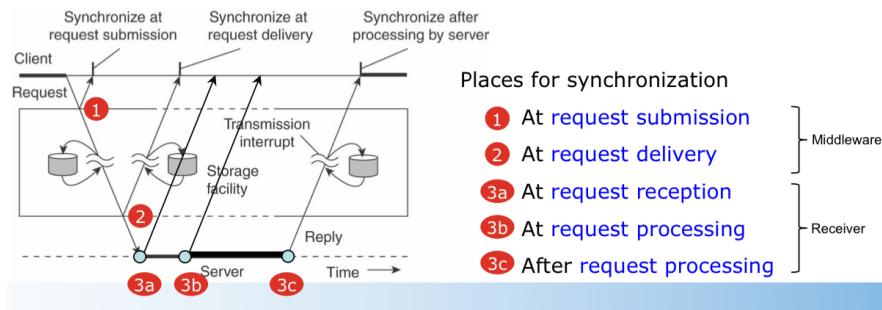
Il MIME (Multipurpose Internet Mail Extensions) è un'estensione del protocollo SMTP che viene utilizzata per **qualificare i dati inviati via internet**. Qualifica il tipo dei dati contenuti nel body di una richiesta o risposta. Definito nelle RFC 2045 e 2056, è suddiviso in cinque categorie principali:

- **Text**: plain, html, etc.
- **Image**: jpeg, gif, etc.
- **Audio**: basic (8 bit), 32kadpcm (32 kbps), etc.
- **Video**: mpeg, QuickTime, etc.
- **Application**: necessitano di essere processati da un'applicazione specifica prima di poter essere visualizzati – msword, octet-stream, etc.

Tipi di comunicazione

Esistono diversi tipi di comunicazione che possono essere classificati in base alla loro sincronicità e persistenza.

- **Comunicazione sincrona:** il mittente e il destinatario devono essere entrambi attivi al momento della comunicazione. Il mittente attende una risposta immediata dal destinatario
 - o Possiamo avere sincronizzazioni a *request submission*, a *request delivery*, a *request processing*
 - o Persistent synchronous, receipt-based transient synchronous, delivery-based transient synchronous communication at message delivery, response-based transient synchronous communication



- **Comunicazione asincrona:** il mittente e il destinatario non devono essere necessariamente attivi contemporaneamente. Il mittente può inviare un messaggio senza attendere una risposta immediata
 - o Persistent asynchronous, transient asynchronous
- **Comunicazione transiente:** se il destinatario non è connesso o disponibile, i dati inviati vengono scartati. Questo tipo di comunicazione richiede che entrambe le parti siano presenti contemporaneamente
- **Comunicazione persistente:** i dati vengono memorizzati da un middleware fino a quando non possono essere consegnati al destinatario. Questo permette ai processi di non dover essere attivi contemporaneamente

Message queuing model

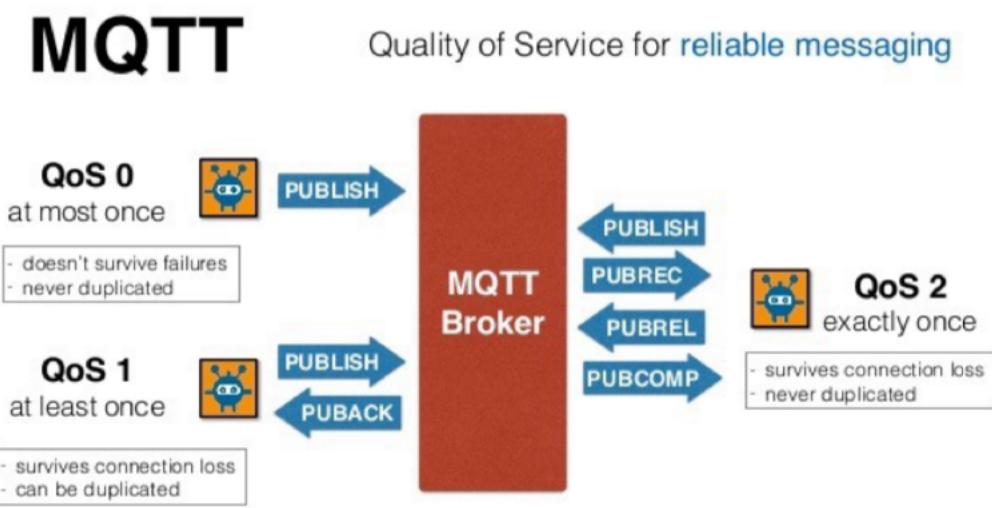
Offre una capacità di memorizzazione a medio termine per i messaggi, **senza richiedere** che il mittente o il destinatario siano attivi durante la trasmissione del messaggio. Utilizza **code** per comunicazioni *loosely-coupled*, permettendo una maggiore flessibilità e scalabilità.

L'architettura stabilisce una relazione tra l'indirizzamento a livello di coda e quello a livello di rete. Organizza un broker di messaggi che agisce come intermediario nella gestione delle code.

I protocolli di publish e subscribe **creano un disaccoppiamento tra mittente** (publisher) e **destinatario** (subscriber). Un broker gestisce il routing dei messaggi basati su argomenti specifici. Si possono realizzare comunicazioni multi-a-molti, con comunicazioni persistenti. Piuttosto che avere una connessione diretta tra sistemi, si ha le connessioni tra le code di ciascun sistema in modo da poter gestire i messaggi via code.

MQTT

MQTT è l'implementazione più utilizzata del pattern *publish-and-subscribe* nel mondo dell'Internet of Things. È un **protocollo di messaggistica client-server publish/subscribe**; è leggero, aperto, semplice, ed è fatto per essere facile da implementare.



Quality of service

Abbiamo tre tipi di quality of service in MQTT:

- **At most once:** fire and forget – il publisher non riceve notifica della ricezione dal broker, né il broker dal subscriber
- **At least once**
 - o Il publisher riceverà notifica (PUBACK) della ricezione dal broker, il broker dal subscriber
 - o La non ricezione del puback in tempo provoca un rinvio del messaggio fino alla ricezione della notifica
 - o I messaggi reinviati contengono un flag, DUP, di duplicazione, che non viene considerato dal broker e inviato al subscriber che dovrà decidere cosa fare
- **Exactly once**
 - o Un messaggio inviato viene ricevuto una sola volta dal subscriber
 - o Vengono scambiati messaggi per certificare la correttezza della ricezione

Proprietà di MQTT

Sessions persistenti

Può essere mantenuto uno stato delle sessioni anche quando le connessioni sono terminate. Se richiesto alla connessione, il broker **mantiene tutti i messaggi** che saranno consegnati al momento della riconnessione.

Retained message

In ogni caso, il broker mantiene uno e un solo messaggio, che viene inviato ad ogni nuovo subscriber al momento della connessione.

Formalismo flessibile

Nel mondo digitale, la gestione dei contenuti richiede un approccio strutturato e flessibile.

Per affrontare questa sfida, si utilizzano **linguaggi di marcatura dei documenti**.

La **marcatura** è l'etichettatura o il tagging di parti di un testo per rendere esplicita un'interpretazione specifica. Possiamo quindi definire la **struttura logica** di un documento, separando i diversi aspetti:

- **Contenuto:** ciò che il testo vuole comunicare
- **Presentazione:** come il testo dovrebbe apparire (formattazione, stili, layout)
- **Navigazione:** come gli utenti possono accedere alle diverse sezioni del documento.

La marcatura rende possibile l'**indipendenza dalla piattaforma**, e il **supporto multilingue**.

Abbiamo due tipi di marcatura:

- **Marcatura procedurale:** descrive come processare il documento
 - Esempi di formati procedurali includono *PostScript, PDF, RTF e Word*
 - Questi formati specificano le istruzioni per la formattazione e la visualizzazione
- **Marcatura descrittiva:** descrive la struttura logica del documento
 - Esempi di formati descrittivi includono *HTML, SGML o XML*
 - Questi formati definiscono la gerarchia dei contenuti, i collegamenti tra le sezioni e altre informazioni semantiche

Architettura a 3 tier

È un modello di progettazione del software che suddivide un'applicazione in tre livelli logici e fisici distinti. I tre livelli principali sono:

- **Presentation Tier:** responsabile dell’interfaccia dell’utente e dell’interazione con l’utente finale. Comprende tutto ciò che l’utente vede e interagisce. Come tecnologie comuni abbiamo *HTML, CSS, JavaScript, React, etc.*
- **Application Logic Tier:** conosciuto anche come “tier di business logic”, contiene la logica di business dell’applicazione, esegue l’elaborazione dei dati e implementa le regole di business. Funziona come intermediario tra il tier di presentazione e il tier di dati, coordinando le operazioni e la logica necessaria. Come tecnologie comuni abbiamo *Java EE, Spring Framework, .NET o Node.js.*
 - o **Servlet** possono gestire parte della logica applicativa leggera, agendo come controller che orchestrano le chiamate alla logica di business e alla persistenza dei dati.
- **Data Tier:** responsabile della gestione e memorizzazione dei dati. Interagisce con il database o altri sistemi di memorizzazione dei dati per creare, leggere, aggiornare e cancellare i dati; comprende i database relazionali e non relazionali, file system, servizi di storage cloud. Come tecnologie comuni abbiamo *SQL databases, NoSQL databases, e JPA/Hibernate.*

XML

È un linguaggio di marcatura che si è imposto come **standard de facto** per lo scambio di informazioni semi-strutturate.

Inizialmente, XML fu pensato come il successore di HTML come linguaggio per il Web. XML è diventato un linguaggio generico, di cui HTML5 rappresenta una specifica particolare.

Abbiamo varie caratteristiche:

- **Marcatura descrittiva e non procedurale**
 - o A differenza di linguaggi procedurali, XML si concentra sulla struttura logica dei documenti
 - o I tag in XML servono a definire la struttura e il significato dei contenuti
- **Flessibilità**
 - o Le etichette possono variare in base all’applicazione o al contesto
 - o Questa flessibilità rende XML adatto a una vasta gamma di utilizzi
- **Indipendenza della piattaforma e supporto multilingue**
 - o XML permette di creare documenti che non dipendono da una specifica piattaforma o linguaggio
 - o È adatto per contenuti multilingue
- **Document Type Declaration (DTD)**
 - o La DTD è parte dello standard XML
 - Contiene o punta a dichiarazioni di marcatura che forniscono una grammatica per una classe di documenti

- Può puntare a un **subset esterno** (speciale entità esterna) contenente dichiarazioni di marcatura, può contenere direttamente le dichiarazioni di marcatura in un **subset interno**, o addirittura fare entrambe le cose
 - Le dichiarazioni di marcatura possono riguardare **tipi di elemento**, **liste di attributi**, **dichiarazioni di entità** o **dichiarazioni di notazione**.
- Definisce la struttura della marcatura ammessa per un tipo di documento specifico
- Con la DTD, possiamo stabilire regole per gli elementi, gli attributi e le entità
- Abbiamo varie limitazioni:
 - **Vincoli sui tipi di valore:** non possono esprimere vincoli sul tipo di valore di un elemento o attributo
 - **Vincoli di co-occorrenza:** non possono specificare che un elemento “unit” è consentito solo quando è presente un elemento “amount”
 - **Flessibilità dello schema:** non supportano la flessibilità per l’ordine degli elementi o la loro posizione nel documento
 - **Mancanza di sottotipizzazione/inheritance:** non consentono la definizione di gerarchie di tipi di elemento
 - **Chiavi composite:** non supportano chiavi composite per garantire l’integrità referenziale
- **Sintassi di XML**
 - Un documento XML è costituito da un insieme di **elementi annidati**
 - Ogni elemento ha una coppia di **etichette di apertura e chiusura**
 - La struttura del documento può essere arbitraria, ma deve rispettare le regole della DTD
 - **XML Schema:** linguaggio di schema più sofisticato che affronta le limitazioni delle DTD
 - **Tipizzazione dei valori:** possiamo specificare il tipo di valore (intero, stringa), e vincoli su valori minimi e massimi
 - **Tipi complessi definiti dall’utente:** possiamo creare tipi di elemento personalizzati
 - **Vincoli di unicità e chiavi esterne:** supporta vincoli di unicità e chiavi esterne
 - **Ereditarietà:** possiamo definire gerarchie di tipi di elemento
- **Validità e ben formato**
 - Un documento XML è **ben formato** se rispetta le regole sintattiche di XML
 - Un documento può essere valido se soddisfa ulteriori vincoli specificati dalla DTD

HTML

HTML è il linguaggio di markup utilizzato per strutturare i contenuti sul web. Permette di **definire elementi** come paragrafi, titoli, immagini e link, che formano la struttura base di una pagina web.

Utilizza una serie di elementi per strutturare contenuti e documenti, che sono rappresentati da **tag**, i quali possono essere di due tipi: **pair tags**, che hanno un tag di apertura e uno di chiusura, e **empty tags**, che non hanno contenuto e si chiudono in se stessi.

```
<nome> -apertura  
...  
<a href="url">link</a>  
</nome> -chiusura
```

È accompagnato da **CSS** per la definizione dello stile, e da JavaScript per aggiungere interattività alle pagine.

La struttura di base di un documento HTML include il tag `<html>`, che racchiude l'intero documento, il tag `<head>` per i metadati come titolo e collegamenti a fogli di stile CSS, e il tag `<body>` che contiene il contenuto effettivo della pagina.

Elementi principali di HTML

Elemento	Tag	Descrizione
Paragrafo	<code><p></code>	Paragrafo di testo
Intestazione	<code><h1>...<h6></code>	Intestazioni, da principali (h1) a secondarie (h6)
Immagine	<code>src="url"</code> <code>alt=desc</code>	Incorpora immagine nel documento
Collegamento	<code><a>href="url">Text</code>	Crea un collegamento a pagina o risorsa
Elenco non ordinato	<code>Primo</code>	Crea un elenco di punti non ordinati
Elenco ordinato	<code>Primo</code>	Crea un elenco numerato
Voce di elenco	<code>Text</code>	Definisce una voce di un elenco

Tabella	<code><table><tr><td>text</td></tr></table></code>	Crea una tabella di dati
Riga della tabella	<code><tr><td>Cella</td></tr></code>	Definisce una riga in una tabella
Cella della tabella	<code><td>Text</td></code>	Definisce una cella in una tabella
Form	<code><form action="submit.php"></form></code>	Crea un modulo per l' input dell'utente
Input	<code><input type="text" name="nome" /></code>	Crea un campo di input (testo, checkbox, radio, button, etc)
Sezione	<code><section><h2>Sezione</h2></section></code>	Definisce una sezione in un documento
Articolo	<code><article><p>Articolo</p></article></code>	Definisce un contenuto indipendente o autocontenuto

Separazione dei contenuti e preparazione

HTML e CSS separano nettamente il **contenuto** (HTML) dalla **modalità di visualizzazione** (CSS). Questo principio, noto come **separation of concerns**, è essenziale per un design informatico efficace.

Front-end frameworks

Per chi non ha senso estetico o tempo per definire un CSS personalizzato, esistono **framework front-end**. Esempi includono *Bootstrap*, *Foundation* e *Skeleton*. Questi framework semplificano la creazione di pagine web con stili predefiniti e componenti riutilizzabili.

Controlli di interfaccia utente

Quando si richiedono dati all'utente, è possibile utilizzare i **controlli di interfaccia utente dedicati**: *checkbox*, *textboxes*, *radio buttons*, *select*, *buttons*, etc..

CSS

CSS è un linguaggio che definisce lo stile di presentazione dei contenuti web, e **possiamo controllare l'aspetto visivo** di una pagina, come colori, font e layout.

DOM

Il Document Object Model è un’interfaccia di programmazione neutrale rispetto al linguaggio e alla piattaforma. Consente ai programmi di accedere e modificare dinamicamente il contenuto, la struttura e lo stile di un documento web. Ogni elemento nel DOM può essere identificato e modificato utilizzando attributi come **id** (identificatore univoco) e **class** (che indica un gruppo di elementi). Rappresenta la struttura e il contenuto di una pagina web con nodi. I nodi corrispondono ad elementi di HTML o XML.

Media query e stili

Le media query sono selettori che **valutano le capacità del dispositivo di accesso alla pagina**, come larghezza, altezza e orientamento dello schermo. Possono applicare stili specifici in base alle condizioni, come la larghezza della pagina o il tipo di dispositivo. Specifica regole di **attività** dei contenuti, tanto in termini di stile che di struttura dell’HTML,

Gestione degli stili e conflitti

Esistono diversi fogli di stile: quelli definiti dall’autore della pagina, dal browser e dall’utente. Per risolvere i conflitti tra stili, si utilizza un algoritmo che considera l’**importanza** (*!important*), la **specificità** *id > class > tag*, e l’**ordine nel sorgente** (il più recente prevale).

CSS

È utilizzato per controllare l’aspetto estetico delle pagine web, permettendo ai designer di *applicare stili uniformi a elementi multipli* con un singolo set di regole. CSS è fondamentale per la creazione di layout responsivi e design adattivi, che garantiscono che le pagine web funzionino bene su una varietà di dispositivi e dimensioni di schermo.

Si scrivono **regole di stile** composte da selettori e dichiarazioni; questi puntano agli elementi HTML che si desidera stilizzare, mentre le **dichiarazioni** sono blocchi di codice che indicano come gli elementi devono essere visualizzati. Qua un esempio:

```
p {  
    color: blue;  
    font-size: 14px;  
}
```

Può essere incluso in una pagina web come **inline** (direttamente negli attributi di stile degli elementi HTML), **interno** (all’interno di `<style>` nel `<head>` della pagina), o **esterno** (in un file separato, collegato tramite `<link>`).

Abbiamo alcune peculiarità importanti di CSS:

- **Cascading:** quando ci sono stili in conflitto, l'ultimo stile definito nel codice avrà la priorità, a meno che non sia sovrascritto da uno stile con una specificità maggiore
- **Specificità:** sistema di punteggio che determina quali regole CSS hanno la precedenza su altre. Per esempio, *stili definiti con ID hanno una priorità maggiore rispetto a quelli definiti con una classe.*
- **Ereditarietà:** alcune proprietà sono ereditate dai genitori ai figli. Questo significa che definendo uno stile, come il colore del testo su un elemento genitore, questo viene applicato anche agli elementi figli, a meno che non venga sovrascritto.
- **Box Model:** ogni elemento ha un modello di riguardo che include *margini, bordi, padding e contenuto effettivo.*
- **Layouts:** offre diversi modelli di layout come *Flexbox* e *Grid*, che permettono di creare layout complessivi e reattivi.
- **Media Queries:** permettono di applicare stili diversi in base alle caratteristiche del dispositivo, come la larghezza dello schermo
- **Variabili CSS:** consentono di definire valori che possono essere riutilizzati in tutto il foglio di stile
- **!important:** sovrascrive qualsiasi altro stile, indipendentemente dalla sua specificità. Un uso eccessivo di questo può rendere il codice difficile da mantenere, e dovrebbe essere evitato quando possibile.

JSON

JSON, acronimo di *JavaScript Object Notation*, è un formato di scambio dati leggero e facilmente leggibile sia dagli esseri umani che dalle macchine. È basato su un sottoinsieme del linguaggio di programmazione JavaScript, ma è **completamente indipendente dal linguaggio**, il che lo rende utilizzabile con molti linguaggi di programmazioni familiari alla famiglia C.

JSON si basa su due strutture dati universali:

- Una **collezione di coppie nome/valore**, simile agli oggetti, record, strutture, dizionari o tabelle hash in vari linguaggi di programmazione
- Un **elenco ordinato di valori**, che corrisponde agli array, vettori, liste o sequenze

La sintassi di JSON prevede che i dati siano separati da virgolette, con le parentesi graffe {} che racchiudono gli oggetti e le parentesi quadre [] che racchiudono gli array.

I tipi di base includono **numeri** (interi, reali o a virgola mobile), **stringhe** (testo Unicode), **booleani** (true/false), **array** (sequenze ordinate di valori separati da virgolette), **oggetti** (collezioni di coppie chiave:valore separate da virgolette), e **null**.

Gli **array** sono utilizzati quando le chiavi sono numeri interi sequenziali, mentre gli **oggetti** sono usati quando le chiavi sono stringhe arbitrarie. Inoltre, *JSON non ha numero di versione*. Inoltre, le strutture di oggetti e array possono essere annidate l'una dentro l'altra.

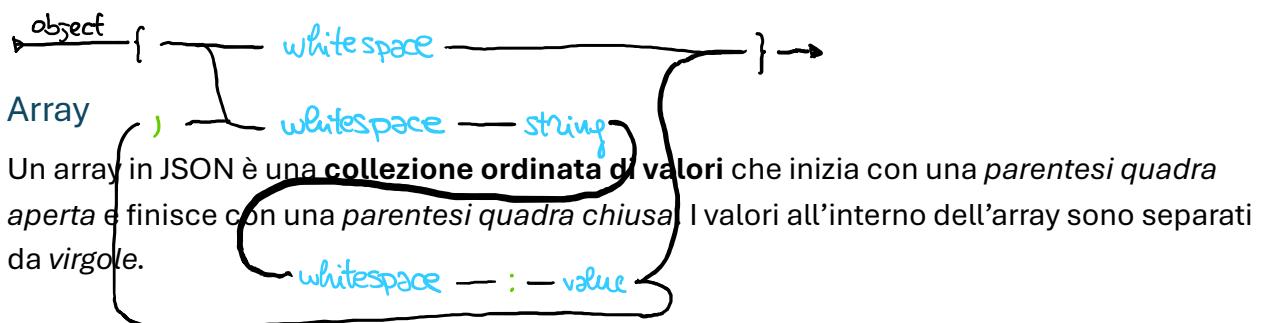
Regole

Un decodificatore JSON deve accettare tutto il testo JSON ben formato, e **può accettare anche testo non JSON**. Un codificatore JSON, invece, deve produrre solo testo JSON ben formato.

YAML è un superset di JSON, il che significa che un decodificatore YAML è anche un decodificatore JSON. Anche Javascript è un superset, quindi **un compilatore JavaScript funge da decodificatore JSON**.

Oggetti

Un oggetto in JSON è un **insieme non ordinato di coppie nome/valore**, iniziando con una *parentesi graffa aperta* e terminando con una *parentesi graffa chiusa*. Ogni nome è seguito da *due punti*, e le coppie sono separate da una *virgola*.



Valori

Un valore in JSON può essere una stringa racchiusa tra virgolette doppie, un numero, un valore booleano come *true* o *false*, *null*, oppure può essere un oggetto o un array.

Una **stringa** in JSON è una sequenza di zero o più caratteri Unicode, racchiusi tra *virgolette doppie*, e utilizzando come carattere escape un *backslash*.

Web app

Le applicazioni web sono strumenti “potenti” che ci permettono di eseguire complesse operazioni direttamente dal browser.

Sono programmi che funzionano all’interno di un browser, e a differenza delle applicazioni native, non richiedono l’installazione su un dispositivo specifico. Questo offre la possibilità di **eseguire la stessa app su diversi dispositivi**, ma un grandissimo svantaggio sono il fatto che sono **bloated** e **consumano molte risorse di sistema**.

HTTP e web

Il web è diventato sinonimo di internet; quest'ultimo è l'**ecosistema che permette ad HTTP di funzionare**, e lo si usa per vari motivi:

- **Semplicità e uniformità**: il successo del web si basa sulla sua struttura semplice e uniforme. Gli sviluppatori possono creare applicazioni senza dover reinventare la ruota ogni volta
- **Potenza degli strumenti**: nel corso degli anni, gli strumenti per lo sviluppo web sono diventati sempre più potenti, anche se molte volte, gli sviluppatori credono che più potenza = “bisogna usare più memoria”, cosa più che sbagliata.
- **Infrastruttura matura**: l’infrastruttura del web è estesa, e supporta sistemi aperti. Attraverso il browser, possiamo invocare diversi servizi web.
- **Interfaccia grafica**: i browser fungono da interfaccia grafica per le applicazioni web

Il browser

Il browser potrebbe essere paragonato a una sorta di *virtual machine*, capace di visualizzare pagine ed eseguire codice interpretato o compilato. È quindi un ambiente con **thread** e **library** che esegue codice JavaScript e HTML.

Questo approccio ha portato a una serie di vantaggi e svantaggi. Ad esempio, la flessibilità del browser consente di *creare pagine dinamiche*, ma può anche *comportare problemi di sicurezza e di performance*.

Il protocollo HTTP

HTTP (Hypertext Transfer Protocol) è alla base del web, e alcune delle sue caratteristiche includono:

- **Formato a caratteri**: i dati vengono tradotti da testo a numeri e viceversa
- **Header e body**: ogni messaggio HTTP ha un **header** (metadati) e un **body** (corpo del messaggio)
- **Linguaggio HTML**: le pagine web utilizzano HTML per input e output; possiamo inviare dati al server tramite form, e ricevere risposte in formato HTML
- **Payload di tipo MIME**: per generalizzare oltre l’HTML, possiamo usare formati come XML, JSON, ZIP e JPEG
- **Conversazioni senza stato**: ogni richiesta HTTP è indipendente dalle altre. Per creare sessioni di lavoro, dobbiamo utilizzare informazioni esplicite come i cookie

Interazione client-server

Il protocollo HTTP ha reso possibile l'interazione tra client e server, consentendo la richiesta e la restituzione di risorse. Con l'avvento delle applicazioni remote, il server ha dovuto espandersi, dando vita a due entità distinte: il **Web Server** (interagisce con l'utente tramite HTTP), e l'**Application Server** (esegue il codice necessario per soddisfare le richieste).

Il web server è un **software** che **fornisce contenuto statico**, come http, immagini, etc., mentre l'application server è quello che **fornisce contenuto dinamico**.

Computazione distribuita

La computazione può avvenire sia sul client che sul server, attraverso l'esecuzione di *programmi compilati o interpretati*.

Architettura di un app web compilata

Abbiamo un **Common Gateway Interface** (CGI), che è un protocollo che permette al server di *attivare un programma* (creare un processo), *passargli le richieste e i parametri provenienti dal client*, e *recuperare la risposta*. Ogni applicazione CGI deve quindi **implementare l'interprete** del protocollo. **CGI** è la comunicazione tra web server e application server.

Quando un client richiede al server un URL corrispondente a un documento HTML, il server restituisce il documento stesso come un file di testo; ciò significa che *il documento viene generato una volta per tutte*, e poi viene inviato al client senza ulteriori elaborazioni. Quando l'URL richiesto corrisponde a un'applicazione CGI, il server esegue il programma in tempo reale, generando dinamicamente la risposta per l'utente. Lo sviluppatore, inoltre, è per forza, obbligato ad usare le librerie e interfacce **specifiche al programma** che permette la comunicazione tra applicazioni.

Architettura di un'applicazione web interpretata

Il protocollo CGI **viene gestito dall'interprete** per il linguaggio usato (le applicazioni, quindi, non devono più gestire il protocollo). I vantaggi sono che *serve programmare solo le logiche delle applicazioni*, il *modello delle applicazioni è conforme al modello del linguaggio utilizzato*, ed è *semplice, portatile e mantenibile*.

Client side (HTML)

Ogni elemento su una pagina web, come un link o un bottone all'interno di un form, può essere utilizzato per **identificare univocamente un endpoint remoto**. I link, ad esempio, possono puntare a risorse remote e, quando cliccati, il browser **invia** una richiesta di tipo **GET** al server.

Possiamo interagire anche tramite **form**: sono elementi fondamentali per la raccolta di dati dall'utente, e possono includere *radio buttons*, *checkboxes* e *altri input*. Il parametro *action* di un form definisce la risorsa remota a cui i dati verranno inviati.

Quando l'utente invia il form, il browser può inviare una richiesta **POST** al server, includendo i dati forniti dall'utente. Le richieste POST includono informazioni come la lunghezza del contenuto (*Content-Length*) e il tipo (*Content-Type*); questo formato permette di inviare dati strutturati che il server può poi elaborare.

Oltre al metodo POST, i dati possono essere inviati anche tramite il metodo GET; questo metodo è meno sicuro per dati sensibili, ma può essere utile per richieste di ricerca o filtraggio.

Server side (servlet)

Le servlet Java sono componenti chiave nell'architettura di un application server. Sono **piccole applicazioni java** che risiedono sul server, e vengono gestite automaticamente da un container, noto anche come *engine*. Questi componenti devono implementare un'interfaccia prestabilita che definisce un set di metodi essenziali per il loro funzionamento. Grazie a questo, si può gestire il ciclo di vita di un *HTTPServlet*, poiché tutte le interfacce contengono un set di metodi standard:

- `init()` è la funzione chiamata dal container una sola volta, e viene utilizzato per eseguire qualsiasi inizializzazione necessaria per la servlet
- `service()` determina il tipo di richiesta http, e chiama il metodo corrispondente. Questo viene chiamato per ogni richiesta dal container.
- Quando il container deve rimuovere la servlet, chiama il metodo `destroy()` per permettere alla servlet di liberare e risorse

Le servlet offrono semplicità e standardizzazione, rendendo più agevole lo sviluppo di applicazioni web; la rigidità del modello può, però, limitare la flessibilità nello sviluppo di soluzioni personalizzate.

I thread e le servlet in Java condividono il fatto di **definire un'interfaccia nota** che specifica il comportamento atteso; permette agli sviluppatori di comprendere e utilizzare queste componenti in modo standardizzato e prevedibile.

Gestione delle servlet

Il container ha il compito di controllare le servlet, **attivandole o disattivandole** in base alle richieste dei client. Sono oggetti Java che risiedono in memoria, e il loro codice viene eseguito

da thread gestiti dall'application server. Possono interagire tra loro, creando un sistema dinamico e reattivo alle richieste degli utenti.

Persistenza e stato

HTTP è un protocollo stateless, il che significa che non mantiene informazioni tra un messaggio e il successivo, e non identifica i client. Tuttavia, possono gestire lo stato della conversazione attraverso l'uso di cookies e oggetti *HTTPSession*.

Interfaccia e metodi

Ogni servlet implementa l'interfaccia *jakarta.servlet.Servlet*, che include metodi come:

- **Void init(ServletConfig config)**: inizializza la servlet, invocato dopo la creazione
- **Void destroy()**: chiamata quando la servlet termina
- **Void service(ServletRequest request, ServletResponse response)**: invocato per gestire le richieste dei client
 - o Questo vuol dire che **ogni volta che una richiesta viene inviata alla servlet**, il container chiama questo metodo per determinare il tipo di richiesta
- **ServletConfig getServletConfig()**: restituisce i parametri di inizializzazione e il *ServletContext* che da accesso all'ambiente
- **String getServletInfo()**: restituisce informazioni tipo autore e versione.

Poiché il motore servlet conosce l'interfaccia delle servlet, è in grado di chiamare i metodi appropriati durante le varie fasi del ciclo di vita, come l'inizializzazione, la gestione delle richieste e la distruzione.

Classi astratte

L'interfaccia è solo la dichiarazione dei metodi che, per essere utilizzabili, devono essere implementati in una classe. Sono presenti **due classi astratte**, cioè che implementano i metodi dell'interfaccia in modo che non facciano nulla. Abbiamo *jakarta.servlet.GenericServlet* che definisce i metodi indipendenti dal protocollo, e *jakarta.servlet.http.HttpServlet* che definisce i metodi per l'uso in ambiente web.

Questo semplifica l'implementazione delle servlet vere e proprie, in quanto basta implementare, ridefinendo li, solo i metodi che interessano.

All'interno di un progetto jakarta abbiamo bisogno di un file **resource** che includa:

- Se deve includere una struttura dati o una variabile di classe, allora deve essere statica
- Un inizializzatore statico
- I vari metodi all'interno della sezione *Ciclo di vita*

HTTPServlet

Implementa **service()** in modo da invocare i metodi per servire le richieste dal web, e include i metodi **doX**, il quale è un metodo HTTP (doGet, doPost, ...). Possono gestire le richieste http e generare risposte appropriate, rendendo il processo di interazione con i client web semplice e diretto.

Abbiamo questi parametri/interfacce:

- **HttpServletRequest**: viene passato un oggetto da service, contiene la richiesta del client, ed estende ServletRequest
- **HttpServletResponse**: viene passato un oggetto da service, contiene la risposta destinata al client, ed estende ServletResponse.

I metodi principali per manipolare le richieste sono:

- **String getParameter(String name)**: restituisce il valore del query parameter dato il nome (valore singolo)
- **Enumeration getParameterNames()**: restituisce l'elenco dei nomi degli argomenti
- **String[] getParametersValues(String name)**: restituisce i valori dell'argomento *name* (valore multiplo)

I metodi principali per manipolare le risposte sono:

- **Void setContentType(String type)**: specifica il tipo MIME della risposta per dire al browser come visualizzare la risposta
- **ServletOutputStream getOutputStream()**: restituisce lo stream di byte per scrivere la risposta
- **PrintWriter getWriter()**: restituisce lo stream di caratteri per scrivere la risposta

Altri metodi inclusi sono:

- **Cookie[] getCookies()**: restituisce i cookies del server sul client
- **Void addCookie(Cookie cookie)**: aggiunge un cookie nell'intestazione della rispostaOs
- **HttpSession getSession(boolean create)**: una HttpSession identifica il client, viene creata se create = true.

Ciclo di vita

L'esistenza di una servlet inizia quando il container, o *engine*, riceve la prima richiesta: è in questo momento che la servlet viene creata. Questa istanza unica della servlet diventa un punto di riferimento condiviso per tutti i client che interagiscono con essa.

Per ogni richiesta in arrivo, il container **genera un nuovo thread** che si occupa di eseguire il metodo *do appropriato*, a seconda del tipo di richiesta HTTP. Prima di entrare in azione, però,

la servlet deve essere inizializzata, e questo compito è affidato al metodo *init*, che il container invoca per eseguire tutte le inizializzazioni specifiche necessarie.

La servlet continua a servire le richieste fino a quando non si verifica uno dei due eventi che ne determinano la **distruzione**: *o non ci sono più thread attivi che la stanno utilizzando*, o *è scaduto un timeout predefinito*. A questo punto, il container chiama il metodo **destroy**, che segnala la fine del ciclo di vita della servlet.

Il framework **JSR (JET SET RADIO)**, **Java Specification Request**, offre una soluzione a questo problema: grazie a una serie di **annotazioni** è possibile **trasformare** una classe qualsiasi in una servlet, facilitando così la gestione delle risorse condivise e riducendo il costo di gestione di oggetti che altrimenti sarebbero spesso identici.

Annotazioni

Sono istruzioni speciali aggiunte al codice sorgente della servlet per fornire informazioni aggiuntive al container servlet, e sono utilizzate per **mappare i metodi delle servlet ai corrispondenti metodi HTTP**, alla fine definendo il comportamento della servlet in risposta a una specifica richiesta http.

Ecco una serie di annotazioni:

Annotation	Description
@PATH(your_path)	Sets the path to base URL + /your_path. The base URL is based on your application name, the servlet and the URL pattern from the web.xml configuration file
@POST	Indicates that the following method will answer to an HTTP POST request
@GET	Indicates that the following method will answer to a HTTP GET request
@PUT	Indicates that the following method will answer to a HTTP PUT request
@DELETE	Indicates that the following method will answer to a HTTP DELETE request
@Produces(MediaType.TEXT_PLAIN[, types])	@Produces defines which MIME type is delivered by a method annotated with @GET. <i>Di solito produce MediaType.APPLICATION_JSON</i>
@Consumes(type[, types])	@Consumes defines which MIME type is consumed by this method
@Pathparam	Used to inject values from the URL into a method parameter

La **terminazione** di una servlet richiede una sincronizzazione accurata tra il container e le richieste dei client. Anche se il timeout è scaduto, potrebbero esserci ancora dei thread che eseguono `service()`. È quindi essenziale **tenere traccia dei thread in esecuzione**, e progettare il metodo `destroy` in modo che notifichi lo shutdown, e attenda il completamento del metodo `service`. È importante inoltre che i metodi che potrebbero richiedere tempi lunghi siano progettati per verificare periodicamente se è in corso uno shutdown e, in caso affermativo, comportarsi di conseguenza.

Java Server Pages

Attraverso l'uso di JSP, è possibile creare pagine web che **rispondono in modo interattivo** alle azioni dell'utente, presentando contenuti generati dinamicamente. Questa tecnologia specifica come un contenitore o server web interagisce con una serie di pagine, note come *viste*, che presentano informazioni all'utente. Le pagine sono composte da tag tradizionali, come *HTML, XML, WML*, e da tag applicativi che gestiscono la generazione del contenuto lato server.

Ciclo di vita

Il ciclo di vita è simile a quello di una servlet, dato che *una JSP viene compilata in un servlet dal container JSP* prima di essere eseguita. È suddivisa nelle seguenti fasi:

1. **Traduzione della JSP in una Servlet:** quando una JSP viene richiesta per la prima volta, il container JSP la traduce in un file Java che estende la classe `HttpServlet`
2. **Compilazione:** il file Java generato viene poi compilato in un file `.class` che può essere eseguito dalla JVM
3. **Caricamento della classe:** il file viene caricato nella JVM del server
4. **Istanziazione:** viene creata un'istanza del servlet generato dalla JSP
5. **Inizializzazione `jsplInit()`:** viene chiamato `jsplInit()`; viene eseguito una sola volta quando il servlet viene caricato per la prima volta, e può essere utilizzato per eseguire operazioni di inizializzazione
6. **Elaborazione della richiesta:** ogni richiesta alla JSP viene gestita dal metodo `jspService()`; questo metodo viene chiamato ogni volta che una richiesta viene fatta alla JSP, e contiene la logica per generare la risposta http
7. **Distruzione:** quando il servlet generato dalla JSP viene rimosso dalla memoria, il metodo `jspDestroy()` viene chiamato; questo metodo può essere utilizzato per rilasciare risorse o eseguire operazioni di pulizia

Vantaggi

Le JSP offrono un vantaggio significativo rispetto alle servlet: *facilitano la separazione tra la logica applicativa e la presentazione dei contenuti*. Questo è analogo alla tecnologia **Microsoft Active Server Page**, che al posto di contenere script Java contengono VBScript.

Separazione tra dinamica e template HTML statico

Le JSP separano efficacemente la parte dinamica delle pagine dal template HTML statico. Il codice JSP è incluso in tag delimitati da <% e %>. Vediamo un esempio:

```
Titolo <i> <%= request.getParameter("title") %> </i>
```

Questa pagina viene poi **convertita automaticamente in una servlet Java** la prima volta che viene richiesta.

doGet e doPost

Il doGet e doPost sono due metodi da inserire all'interno del file java della servlet, che permette di fare le richieste get o post all'URL.

```
protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
    request.setAttribute("name", name);
    var sc = this.getServletContext();
    sc.getRequestDispatcher("path-to-
jsp").include(request, response);
```



```
protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
    var name = request.getParameter("name");
    var int =
        Integer.parseInt(request.getParameter("int"));
    var bool =
        Boolean.parseBoolean(request.getParameter("bool"))
    ;
    var object = new Object();
    object.setName(name);
    objects.add(object);
    doGet(request, response);
```

Elementi costitutivi delle JSP

- **Template text:** le parti statiche della pagina HTML
- **Commenti:** delimitati da <%-- commento --%>
- **Direttive:** istruzioni per il compilatore, come <%@ direttiva %>, che guidano la creazione della servlet
- **Azioni:** tag come <jsp:XXX attributes>body </jsp:XXX>, che definiscono azioni specifiche da convertire in Java.

Elementi di scripting

Sono istruzioni nel linguaggio specificato nelle direttive, e si dividono in tre tipi: **scriptlet, declaration, expression**.

Le direttive page elencano attributi/valori validi per la pagina in cui inseriti, come <%@ page import="java.util.*" buffer="16k" %>. L'azione **forward** determina l'invio della richiesta corrente a una JSP indicata, mentre **include** invia dinamicamente la richiesta ad una data URL, e ne include il risultato. L'elemento **useBean** localizza ed istanza un javaBean nel contesto specificato.

Gestione degli oggetti e scope

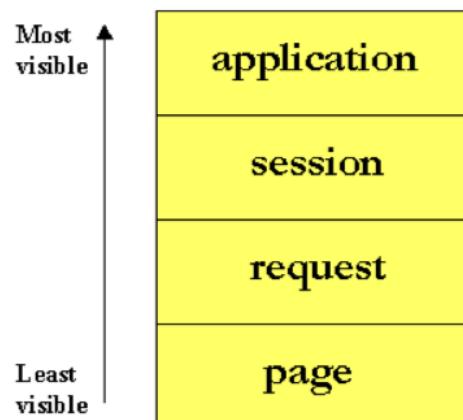
Gli oggetti in JSP possono essere creati implicitamente usando le direttive JSP, esplicitamente con le azioni, o direttamente usando uno script. Hanno un attributo **scope** che ne definisce l'ambito; questo può essere la pagina, la richiesta, la sessione o l'applicazione.

Il linguaggio di script in JSP serve per interagire con oggetti Java e altre servlet, gestire le eccezioni Java, e utilizzare oggetti impliciti che non devono essere creati. Questi includono *request*, *response*, *out* (System.out), *page* (riferimento alla JSP stessa), *pageContext*, *session*, *application* (totalità delle JSP che fanno riferimento ad una particolare applicazione).

Tutti gli oggetti sono identificati con un certo livello di accessibilità, ovvero lo **scope**, che permette di scoprire quali pagine possono accedere a quella risorsa, e quando.

Questi scope hanno le seguenti caratteristiche:

- **Page scope:** limitato alla singola pagina JSP. Gli oggetti archiviati nello scope della pagina sono accessibili solo dalla pagina.
 - o L'attributo della pagina può essere definito usando la direttiva <%@ page %>
 - o Lo scope della pagina termina quando la risposta viene inviata al client
- **Request scope:** limitato alla singola richiesta http. Gli oggetti archiviati nello scope della richiesta sono accessibili solo durante la durata della richiesta http.
 - o Le informazioni archiviate in questo scope possono essere passate tra servlet e JSP durante il processo di inoltro della richiesta
 - o I dati nello scope della richiesta possono essere utili per passare informazioni tra più servlet, o per conservare dati durante la durata di una singola richiesta
- **Session scope:** legato alla sessione utente. Gli oggetti archiviati nello scope della sessione sono accessibili a tutte le richieste dello stesso utente durante la sua sessione sul sito web.
 - o Le informazioni archiviate nello scope della sessione possono essere utilizzate per mantenere lo stato dell'utente tra più pagine durante la sua sessione
 - o Importante notare che le sessioni possono essere impostate per terminare dopo un certo periodo di inattività, o quando l'utente effettua il logout, a seconda della configurazione del server
- **Application scope:** globale per tutta l'applicazione web. Gli oggetti archiviati nello scope dell'applicazione sono accessibili da qualsiasi parte dell'applicazione web, da qualsiasi utente.
 - o Le informazioni archiviate nello scope dell'applicazione sono utilizzate per conservare dati, che devono essere condivisi tra più sessioni utente, o che devono essere persistenti per tutta la durata dell'applicazione



- Le informazioni archiviate in questo scope sono generalmente utilizzate per la configurazione dell'applicazione, le risorse condivise o i dati di cache

JavaBean

Un javaBean è una classe Java che segue una convenzione di nomenclatura standard per i suoi metodi e proprietà, e include:

- **Costruttore:** ogni javaBean deve avere almeno un costruttore vuoto, permettendo così la sua istanziazione senza specificare parametri
- **Proprietà private:** tutti i campi, o proprietà, devono essere dichiarati come privati per encapsulare i dati e proteggerli dall'accesso diretto
- **Metodi di accesso:** per ogni proprietà deve esistere un metodo *getter* e *setter* seguendo la convenzione *setX()* e *getX()* o *isX()* per le proprietà booleane, dove X è il nome della proprietà con la prima lettera maiuscola

Sessioni

Le sessioni in JSP sono utilizzate per memorizzare informazioni relative a un utente durante la sua interazione con l'applicazione web. Abbiamo:

- **Scope session:** utilizzando lo scope *session*, le informazioni possono essere memorizzate e rese disponibili attraverso diverse pagine JSP
- **Oggetto HttpSession:** attraverso l'oggetto è possibile un approccio più programmatico, utilizzando i metodi *putValue()* e *getValue()*, per inserire e recuperare dati dalla sessione

Pattern MVC

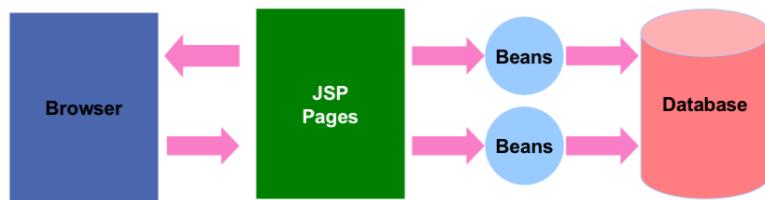
Il **Model View Controller** è un pattern architetturale che separa un'applicazione in tre componenti principali:

- **Model:** rappresenta i dati e i metodi principali per la loro manipolazione
- **View:** gestisce la presentazione dei dati all'utente
- **Controller:** coordina l'interazione tra l'interfaccia utente e il modello di dati, gestisce la logica dell'applicazione

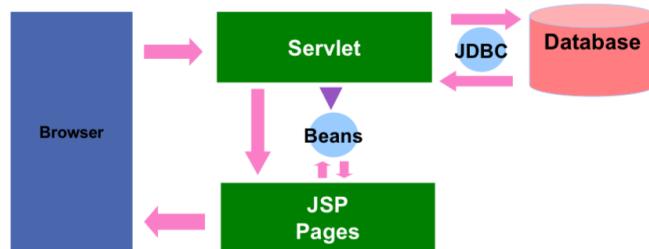
Nel pattern MVC la separazione tra Controller e Model favorisce l'integrazione tra i dati e la logica dell'applicazione.

Abbiamo due modelli:

- **Model 1:** implementa JSP dove la logica di business e la visualizzazione sono accorpate nella stessa JSP, mentre i dati sono gestiti tramite JavaBeans.
 - o **JavaBeans** seguono una specifica precisa, con *costruttore senza parametri* e *metodi di accesso alle proprietà private*
 - o **Scope** determina la visibilità e la durata del bean, che può essere *page*, *request*, *session* o *application*
 - o **Azioni:** permettono di inizializzare e accedere alle proprietà del bean



- **Model 2:** prevede che le richieste siano gestite da una servite Java, che poi interagisce con le pagine JSP per la presentazione dei dati
 - o **Servlet:** genera i dati dinamici e li mette a disposizione della pagina JSP come JavaBeans
 - o **JSP:** legge i dati dai beans e organizza la presentazione in HTML da inviare all'utente.



Internet Of Things (IoT)

L'evoluzione tecnologica ha portato alla nascita dell'**Internet of Things**, un ecosistema in cui non solo i computer, ma anche dispositivi più piccoli e meno potenti, possono connettersi e comunicare. Questi dispositivi, spesso privi di un sistema operativo completo, possono utilizzare connessioni alternative come il Bluetooth LE per interagire con altri dispositivi elettronici o persone.

L'IoT ha introdotto una nuova dimensione nella nostra quotidianità: la **servitizzazione del mondo reale**; gli oggetti diventano fornitori di servizi, creando un ponte tra il mondo fisico e quello dei dati. Questi servizi possono essere eseguiti da dispositivi che, pur essendo autonomi, sono raggiungibili, identificabili in maniera univoca e integrabili in sistemi più ampi.

L'adozione dell'IoT e dei servizi correlati offre numerosi vantaggi, come la **riutilizzabilità**, **l'agilità nello sviluppo**, e la **scalabilità**. Però, presenta anche sfide come la **gestione**

complessa del ciclo di vita e la **dependency hell**, ovvero la difficoltà di gestire le dipendenze tra i vari servizi.

Mattoncini

I servizi IoT possono essere paragonati a **mattoncini software**, elementi standardizzati che, sebbene possano esistere singolarmente, trovano il loro vero potenziale quando combinati per realizzare soluzioni complesse.

Abbiamo la differenza tra due tipi di sistemi distribuiti:

- **Sistemi tradizionali:** sistemi singoli sviluppati in modo *top-down* mediante decomposizione. Quasi sempre implementano il modello *client-server*, e presentano un **accoppiamento forte** (un'unica organizzazione mantiene e/o possiede il sistema)
- **Sistemi contemporanei:** includono sottosistemi di terze parti. Questo favorisce un **accoppiamento debole**, dove ogni componente viene mantenuto indipendentemente dagli altri, e dove ogni componente software deve essere in grado di *servire sistemi diversi in modi diversi*.

SOA

SOA è uno **stile architettonico** che si concentra su elementi discreti riutilizzabili, chiamati **servizi**, invece che su un design monolitico, per costruire le applicazioni. Un servizio fornisce **funzionalità ai richiedenti**, i quali potrebbero essere anche altri servizi, ed è fornito anche attraverso la rete. Lo stato di un servizio non dipende dallo stato di un altro servizio, ovvero è **stateless**.

Ogni servizio deve:

- Fornire una **descrizione** delle sue **funzionalità** da **scoprire** e **selezionare**
- Fornire l'**accesso** alle sue funzionalità attraverso **protocolli di rete** noti
- Supporto alla **composizione** con altri servizi per fornire soluzioni complesse
- Rispondere alle **esigenze** aziendali dei clienti e ai **requisiti** del dominio
- Garantire un livello di **qualità di servizio**

Composizione

La composizione dei servizi nell'IoT può avvenire attraverso due modalità principali:

- **Orchestrazione:** un servizio principale gestisce il flusso di lavoro, richiedendo funzionalità a vari servizi. Descrive come i servizi **interagiscono tra loro**, compresa la logica di business e l'ordine di esecuzioni dal punto di vista e sotto il controllo di un singolo attore. In un'**architettura orchestrata**, quindi, è possibile creare una rete dinamica di servizi, in cui i singoli servizi possono essere offerti da organizzazioni

diverse. Un'organizzazione assume il ruolo di orchestratore, e si fa carico di *implementare il servizio controller per altre organizzazioni* (di solito esistono **broker di servizi**: trovano il miglior allineamento tra i servizi richiesti dal Service Requestor e quelli offerti dai Service Providers).

- **Coreografia:** vede ogni servizio responsabile del proprio compito, interagendo con gli altri in una sequenza di azioni condivise. Descrive la sequenza di interazioni tra più parti coinvolte nel processo dal punto di vista di tutte le parti, e definisce lo stato condiviso delle interazioni tra le entità. Ogni servizio è responsabile di portare a termine il proprio compito, invocando anche altri servizi.

Descrizione e contratti

Ogni servizio nell'IoT dovrebbe fornire una **descrizione chiara delle proprie funzionalità**, accessibili attraverso protocolli di rete noti. Queste descrizioni formano la base di un **contratto** tra cliente e fornitore, che stabilisce le aspettative, e le responsabilità di entrambe le parti.

Servizio

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

Servizio web

Un **servizio web** è un'applicazione software indipendente identificata da un *URI*, che può essere **scoperta e invocata** da altri sistemi software attraverso una rete. Le sue interfacce sono definite, descritte e scoperte. Supporta interazioni dirette con altri software tramite messaggi e protocolli basati su internet. I servizi web utilizzano **tecniche web**, in particolare HTTP, per fornire funzionalità. Possiamo avere inoltre altre due definizioni:

- **Autonomia:** i servizi web sono autonomi e modulari, pubblicati, localizzati e invocati attraverso il web
- **Incapsulamento:** i servizi web sono componenti liberamente accoppiati che possono legarsi dinamicamente tra di loro
- **Auto descrittivo**
- Applicazioni modulari che possono essere **pubblicate, localizzate e invocate** attraverso il Web
- Componenti web **liberamente accoppiati che possono legarsi dinamicamente** l'uno all'altro

Ha i seguenti componenti:

- **Interfaccia:** descrizione delle funzionalità del servizio

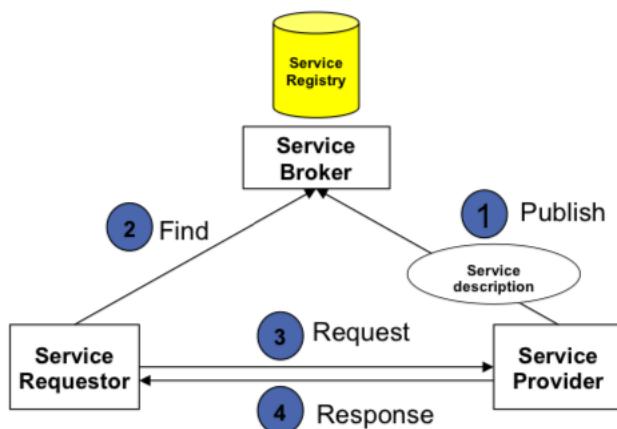
- Ben nota: usa un linguaggio di descrizione standard, e una possibile gestione automatica da parte del middleware
- **Punto di accesso unico:** usa URI (URL/URN) per la scoperta
 - È scopribile tramite servizi di nomi
- **Scambio di dati basato su documenti:** utilizzo di formati standard come XML o JSON

Ha le seguenti peculiarità:

- I componenti sono **pubblici**: sono scopribili (naming, registri, accesso), e le *interfacce sono pubbliche* (protocolli standard gestiti dalla macchina)
- **Componibilità:** se i servizi sono composti si parla di *orchestrazione*; se invece c'è bisogno di coordinamento si parla di *coreografia*
- **Descrizioni semantiche:** parliamo di scoperta, composizione, recommending system, etc.
- **Quality of Service:** di base, abbiamo la *sicurezza, disponibilità e prestazioni*, ma abbiamo bisogno anche di context awareness.
- **Organizzazione del sistema:** p2p (qualsiasi organizzazione, applicazione gestita), ESB (qualsiasi organizzazione, middleware gestito), Grid (modello dato)

SOA

SOA ha come componenti il **servizio** e la **descrizione del servizio**. Può effettuare un *publish* (servizio), *find* (servizio/endpoint), o *interact* (richiesta-risposta). I vari ruoli di ciascun elemento sono:



- **Service broker:** agisce come intermediario tra i **service provider** e i **service requestor**. Il suo ruolo è quello di **gestire il Service Registry**, facilitando la pubblicazione, la ricerca e la selezione dei servizi. Il broker può anche offrire funzionalità aggiuntive come la *valutazione della qualità dei servizi*, la *gestione delle policy* e la *negoziazione degli SLA*
- **Service Requestor:** entità che *cerca e utilizza i servizi disponibili nel Service Registry*. Può essere un'applicazione, un modulo software, o un altro servizio che ha bisogno di

una funzionalità specifica offerta da un *service provider*. Il requestor usa il Broker per trovare il servizio più adatto alle proprie esigenze, e stabilire una comunicazione con esso.

- **Service Provider:** entità che *crea e offre servizi* agli altri componenti dell'architettura SOA. Pubblica le informazioni relative ai propri servizi nel Service Registry tramite il Service Broker, rendendoli disponibili per l'utilizzo da parte dei service requestor. Il provider è responsabile della *manutenzione, aggiornamento e garanzia della qualità* dei servizi offerti.

Accordo sul livello di servizio

Lo **SLA** (Service Level Agreement) è un **contratto** tra il provider e l'utente, e assicura che la funzionalità sia fornita correttamente, e garantisce *proprietà non funzionali*.

Lo SLA, inoltre, definisce le **caratteristiche non funzionali** garantite dal servizio, e comprende diversi **SLO** (Service Level Objectives), che definiscono la qualità del servizio da garantire attraverso metriche specifiche.

Composizione di servizi

Una composizione è un **insieme di servizi interconnessi**, che possono essere utilizzati come nuovi servizi, e lo sono se almeno uno richiede la funzionalità esposta dall'altro. La compatibilità tra i servizi è essenziale per il successo della composizione: devono parlare la **stessa lingua** sintatticamente e semanticamente, e un fornitore di servizi deve presentare un'interfaccia pubblicata.

Business Processes

Un Business Process è un **insieme di attività correlate** eseguite da persone e applicazioni, per ottenere un risultato ben definito. Questi sono creati dalla composizione di servizi.

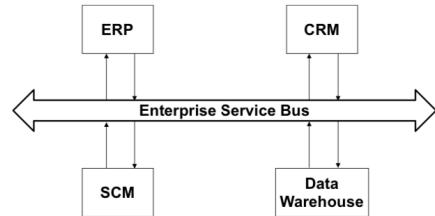
Può contenere condizioni definite che ne determinano l'avvio e uscite definite al suo completamento. Può comportare interazioni formali, o relativamente informali tra i partecipanti (sia umani che software); può contenere una serie di attività automatizzate e/o manuali. È distribuito e personalizzato al di là dei confini all'interno delle organizzazioni e tra di esse, e spesso si estende a più applicazioni con piattaforme tecnologiche diverse. Ha una durata che può variare notevolmente, e di solito è di lunga durata: una singola istanza può durare mesi o addirittura anni.

Enterprise Service Bus (ESB)

L'ESB semplifica l'integrazione tra i componenti di un sistema informativo, ed implementa il modello **publish/subscribe** con funzionalità di *instradamento, trasformazione dei messaggi* e

comunicazione sicura. È un esempio di **approccio coreografico**, con le seguenti caratteristiche principali:

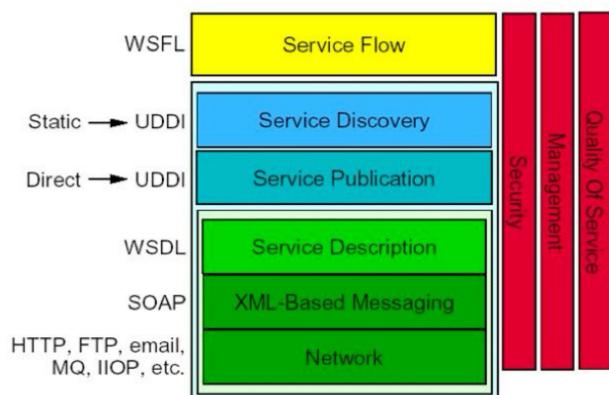
- Instradamento dei messaggi tra applicazioni e servizi
- Trasformazione del messaggio
- Comunicazione sicura
- Architettura estensibile



SOAP

L'acronimo SOAP sta per *Simple Object Access Protocol*. È un protocollo **basato su XML** che consente alle applicazioni software di comunicare usando messaggi XML; gli spazi dei nomi XML forniscono una semantica ai dati. È agnostico rispetto al sottosistema di trasporto, è utilizzato per sfruttare i messaggi XML, ed è *estensibile ed indipendente* dalla piattaforma di programmazione.

Si compone dei seguenti componenti:



- **Business Process:** BPEL/WSFL
Linguaggio di esecuzione dei processi aziendali
- **Search and Find:** UDDI
Universal Discovery Description and Integration, per i registri dei servizi Web.
- **Description:** WSDL
Web Services Description Language, per descrivere servizi in rete basati su XML.
- **Messaggi:** SOAP
Simple Object Access Protocol, per definire un modo uniforme di passare dati codificati in XML.
- **Network:** Protocolli Internet
HTTP, SMTP, ecc.

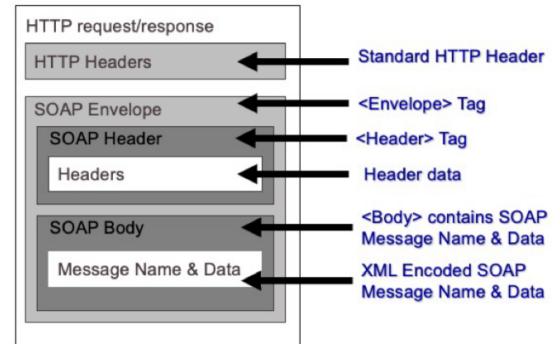
- **Business Process:** BPEL/WSFL: linguaggio di esecuzione dei processi aziendali
- **Search and Find:** UDDI (Universal Discovery Description and Integration), per i registri dei servizi web
- **Description:** WSDL (Web Services Description Language), per descrivere servizi in rete basati su XML
- **Messaggi:** SOAP (Simple Object Access Protocol), per definire un modo uniforme di passare dati codificati in XML
- **Network:** protocolli internet: HTTP, SMTP, etc.

Messaggio SOAP

Un messaggio SOAP ha i seguenti componenti:

- **SOAP Envelope:** ingloba il contenuto del messaggio

- **SOAP Header (opzionale):** può essere elaborato dai nodi tra l'origine e la destinazione. Contiene blocchi di informazioni su come elaborare il messaggio: impostazioni di instradamento e di consegna, asserzioni di autenticazione/autorizzazione, contesti di transazione
- **SOAP Body:** messaggio effettivo da consegnare ed elaborare, sia per le informazioni di richiesta che per quelle di risposta



SOAP con HTTP

Sebbene SOAP possa utilizzare diversi protocolli di trasporto, HTTP è quello più comunemente utilizzato.

Le richieste HTTP sono costituite da un **metodo** (GET/POST), seguito dall'**URL richiesto** e dalla **versione del protocollo**.

Le risposte seguono la semantica di HTTP per fornire il codice di stato della risposta. Esistono due modelli per lo scambio di messaggi:

- **SOAP request-response:** il metodo POST viene utilizzato per portare i messaggi SOAP nel corpo delle richieste/risposte HTTP
- **SOAP response:** nelle richieste HTTP viene utilizzato il GET per ottenere il messaggio SOAP nel corpo della risposta.
- Bisogna ricordarsi di impostare l'intestazione Content-Type come application/SOAP+xml.

Abbiamo qualche esempio:

Richiesta http GET	Risposta http GET	Richiesta http POST
GET /travel.example.org/reservations? code=F3T http/1.1 Host: travelcompany.example.org Accept: text/html;q=0.5, application/soap+xml	http/1.1 200 OK Content-Type: application/soap+xml; charset="utf-8" Content-Length: 200 <?xml version='1.0' ?> <env: Envelope xmlns:env= http://w3.org/2003/05/soap-envelope > <env:Header>...</env:Header> <env:Body>...</env:Body> </env:Envelope>	POST /Reservations HTTP/1.1 Host: travelcompany.example.org Content-Type: application/soap+xml; charset="utf-8" Content-Length: 230 <?xml version='1.0' ?> <env: Envelope xmlns:env= http://www.w3.org/2003/05/soap-envelope > <env:Header>...</env:Header> <env:Body>...</env:Body> </env:Envelope>

WSDL

L'acronimo WSDL sta per *Web Services Description Language*, che è un **linguaggio di descrizione dei servizi web**. È utilizzato per definire *come e dove accedere al servizio*.

È basato su XML per descrivere i servizi web, i messaggi e le modalità di invocazione; consente di descrivere quattro dati principali per un servizio:

- Informazioni sull'**interfaccia** che descrivono tutte le operazioni pubblicamente disponibili di un servizio
- Dichiarazioni di **tipi di dati** per tutti i messaggi; i tipi complessi possono essere dichiarati, utilizzando SOAP, e utilizzati
- Informazioni di **binding** sul protocollo di trasporto
- Informazioni sull'**indirizzo** per la localizzazione del servizio (URI)

Ha le seguenti parti:

- **Parte astratta:** descrive le operazioni e i messaggi. Le interfacce sono indipendenti dal canale di comunicazione
 - o Un'operazione associa modelli di scambio di messaggi a uno o più messaggi
 - o I modelli di scambio di messaggi definiscono la sequenza e la cardinalità dei messaggi scambiati tra i nodi
 - o Un'interfaccia raggruppa queste operazioni in modo indipendente dal canale di comunicazione
- **Parte concreta:** il *binding* specifica il protocollo di trasporto. L'*endpoint* associa un URI a un binding. Un servizio raggruppa gli endpoint che implementano un'interfaccia comune.
 - o **Binding:** attributo *name* definisce il nome del binding, ogni nome deve essere univoco all'interno dello spazio dei nomi del target WSDL 2.0.
 - **Interface** contiene il nome di una delle interfacce definite

- **Type** definisce il formato del messaggio da utilizzare, in questo caso è SOAP
- **Wsoap:protocol** definisce il protocollo di trasporto.
- **Operation**
 - **Ref:** fa riferimento a una specifica operazione
 - **wsoap:meo:** definisce lo schema di scambio dei messaggi per SOAP
- **Fault**
 - **Ref:** definisce quale errore ci si riferisce per questo binding
 - **wsoap:code:** definisce il codice di errore che determina l'invio di questo messaggio
- **Service:**
 - **Name:** definisce il nome del servizio. Ogni nome di servizio deve essere unico all'interno dello spazio dei nomi di destinazione
 - **Interface:** definisce il nome dell'interfaccia (precedentemente definita nella sezione interface) implementata dal servizio
- **Endpoint**
 - **Name:** definisce il nome dell'endpoint, che deve essere unico all'interno dei servizi
 - **Binding:** specifica quale binding già definito verrà utilizzato da questo endpoint
 - **Address:** specifica l'indirizzo fisico dove il servizio sarà disponibile.

REST

Con REST si torna ai principi del protocollo HTTP, eliminando le ridondanze e assegnando una semantica ai verbi e agli URI. Rende centrale il concetto di risorsa, ed è un insieme di guidelines e best practices. REST è meno sicuro rispetto a SOAP.

	REST	SOAP
Protocollo di trasporto	http in modo nativo	Vari protocolli (HTTP, SMTP, TCP)
Formato del messaggio	Vari formati (XML, JSON)	XML-SOAP (envelope, header, body)
Identifieri di servizio	URI	URI e indirizzamento WS
Descrizione del servizio	Documentazione testuale, linguaggio di descrizione delle applicazioni web (WADL)	Web Services Description Language (WSDL)
Service Discovery	Nessun supporto standard	UDDI

I sistemi distribuiti per funzionare devono essere basati su un **modello condiviso**: con WSDL/SOAP i sistemi SOA devono concordare un API comune e messaggi, è molto ben definito ma risulta eccessivamente complesso.

REST, invece, è costruito intorno all'idea di semplificare l'accordo:

- **Nonums** sono necessari per nominare le risorse di cui si può parlare
- **Verbs** sono le operazioni che possono applicate alle risorse nominate
- **Content types** definiscono quali rappresentazioni delle informazioni sono disponibili

REST sta per *Representational State Transfer*, ed è uno stile architettonico per i sistemi distribuiti:

- Le risorse sono identificate da **URI**
- Le risorse vengono manipolate attraverso le loro **rappresentazioni**; quindi possono esistere più rappresentazioni
- Comunicazione tramite messaggi, che sono **autodescrittivi e senza stato**
- Lo stato dell'applicazione evolve per mezzo della manipolazione delle risorse da parte del client
- Hypermedia è il modo per che ha il server per guidare il comportamento del client.
 - o Includere link all'interno delle risposte dell'API, che permettono di navigare attraverso le risorse dell'applicazione in modo dinamico.
 - o Il server fornisce i link necessari per il client per navigare e modificare lo stato dell'applicazione

La risorsa è una qualsiasi informazione che può essere nominata; sono lo **stato dell'applicazione** che possono cambiare nel tempo, e di solito cambiano con l'interazione con il client. Hanno **identificatori**, ed espongono un'**interfaccia uniforme** (GET, POST, PUT, DELETE + identificatori in URI).

Su richiesta, un servizio può trasferire una rappresentazione della risorsa a un client: richiede un'**architettura client-server (vincolo)**. Un client può trasferire una rappresentazione modificata di una risorsa, manipolando le risorse attraverso rappresentazioni, avendo nessuna invocazione di azioni. Le rappresentazioni restituite dal server **devono collegarsi ad altre rappresentazioni**, e i client possono seguire un collegamento proposto e ricevere/modificare risorse. Hypermedia viene usato come motore dello stato dell'applicazioni. L'hypermedia control è un concetto chiave, ed è il concetto di controllare il flusso di navigazione e interazione tra i servizi esposti come risorse. Hypermedia invece è un sistema di informazione che utilizza collegamenti ipertestuali per connettere e navigare tra diversi contenuti correlati.

Le **interazioni sono prive di stato**: ogni richiesta dal client al server deve contenere tutte le informazioni necessarie a comprendere la richiesta, e non può sfruttare alcun contesto memorizzato sul server. Grazie a questo, c'è bisogno di **messaggi autodescrittivi**, con tipi di media standard, metadati e dati di controllo.

REST mette al centro la semplicità:

- La **cache** migliora i tempi di risposta, e riduce il carico del server
- L'**assenza di stato** e la **riduzione delle comunicazioni** facilitano il bilanciamento del carico tra i server
- **Software meno specializzato**, perché le tecnologie sottostanti sono ben note e semplici
- L'**identificazione delle risorse** avviene tramite meccanismi **standard**, non sono necessari nomi aggiuntivi.
- I principi REST enfatizzano l'uso corretto e completo del protocollo HTTP per pubblicare servizi sul web; c'è un meccanismo leggero e stratificato per l'integrazione di dati e servizi, e fornisce una piattaforma applicativa distribuita e guidata dagli ipermedia.

Cacheability

L'avere un'interfaccia uniforme, una mancanza di stato, e i messaggi autodescrittivi significa che c'è bisogno di avere *cacheability*, la quale richiede un sistema a livelli:

- Ci deve essere almeno un livello di cache
- Nella cache è memorizzata l'ultima rappresentazione di una risorsa
- Una cache viene invalidata quando viene modificato lo stato della risorsa (PUT, DELETE)

La cache riduce la latenza e il traffico di rete; vengono memorizzati nella cache solo le risposte ai GET, non i POST etc.... Le cache possono essere **lato client** o **proxy/server**.

Caching locale

È una tecnica che permette di memorizzare copie di dati frequentemente accessibili in diversi punti lungo il percorso di richiesta-risposta. Quando un client richiede una rappresentazione di una risorsa, la richiesta può passare attraverso una cache locale prima di raggiungere il servizio che ospita la risorsa.

Se una delle cache lungo il percorso di richiesta possiede *una copia aggiornata della rappresentazione richiesta*, la utilizzerà per soddisfare la richiesta. Se nessuna cache può soddisfare la richiesta, allora questa prosegue verso il servizio, o server di origine.

Utilizzare HTTP per costruire applicazioni REST

Abbiamo certi header importanti per il cache-control, nella risposta del server:

Intestazione	Proprietà
Expires	Data HTTP, mantenere in cache fino alla scadenza
Cache-control: max-age	Un secondi. Mantenere in cache per il tempo indicato
Cache-control: s-maxage	Secondi. Come sopra, ma solo per i proxy
Cache-control: public	La risorsa può essere salvata in una cache condivisa
Cache-control: no-cache	La risorsa può essere memorizzata nella cache, ma deve essere convalidata dal server di origine prima di ogni riutilizzo
Cache-control: no-store	Non può memorizzare nella cache
Cache-control: must-revalidate	La risorsa può essere memorizzata in cache e riutilizzata quando è fresca. Se diventa stantia, deve essere convalidata con il server di origine prima di essere riutilizzata
Cache-control: proxy-revalidate	Come sopra, ma solo per i proxy

Nel costruire un'applicazione, è importante **definire i nomi** (URI): tutto in un sistema REST è una risorsa, un nome. Ogni risorsa ha un **URI**, e devono essere descrittivi; non dovrebbero essere **opachi**, e dovrebbero inoltre essere mantenuti **invariati** e **persistenti**. È importante usare le *path variables* per codificare la gerarchia, usano altri segni di punteggiatura per evitare di sottointendere una gerarchia, e usa le query variables per implicare le condizioni di filtraggio.

Successivamente, bisogna **definire i formati**: né HTTP né REST impongono un'unica rappresentazione dei dati. Una risorsa può avere diverse rappresentazioni. È importante **evitare di creare rappresentazioni personalizzate**; dobbiamo utilizzare tipi di media ben noti, poiché rende i contenuti più accessibili, e parte del vincolo di messaggistica autodescrittiva. Le rappresentazioni **in entrata devono essere uguali a quelle in uscita**: un client deve essere in grado di accedere e modificare un documento, e dove il server deve *scartare tutti i dati estranei*.

Poi **scegliamo le operazioni** che possono essere applicate alle risorse. Possiamo usare i seguenti metodi HTTP:

		Cache	Safe	Idempotente
OPTIONS	Rappresenta una richiesta di informazioni sulle opzioni di comunicazione disponibili sulla catena richiesta/risposta identificata dal Request-URI	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

GET	Recuperare qualsiasi risorsa identificata dall'indirizzo Request-URI	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
HEAD	Identico a GET, tranne per il fatto che il server non deve restituire un message-body nella risposta	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
POST	È utilizzato per richiedere che il server di origine accetti l'entità contenuta nella richiesta come nuovo subordinato della risorsa identificata dal Request-URI nella Request-Line			
PUT	Richiede che la risorsa allegata sia memorizzata nell'ambito del Request-URI fornito			<input checked="" type="checkbox"/>
DELETE	Richiede che il server di origine cancelli la risorsa identificata dal Request-URI			<input checked="" type="checkbox"/>
TRACE	È usato per invocare un loop-back remoto, a livello di applicazione, del messaggio di richiesta		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Patch	Viene utilizzato per eseguire aggiornamenti parziali su una risorsa			Può essere

Ogni richiesta avviene in modo isolato: il client guida lo stato dell'applicazione, tutte le informazioni necessarie per l'esecuzione di una richiesta sono presenti nel body/header della richiesta, e il server non utilizza le informazioni delle richieste precedenti. I possibili stati di un server sono anch'essi risorse e hanno URI, devono essere gestite dal client.

Il carico delle applicazioni prive di stato può essere *bilanciato* e *memorizzato in cache*; le applicazioni possono essere partizionate e scalate. Le sessioni sono da evitare, e dovrebbe dimenticarsi dei client tra una richiesta e l'altra.

Codice di stato della risposta

Il codice di stato della risposta viene generato dal server per indicare l'esito di una richiesta; è un numero di 3 cifre:

- **1xx** (informativo); richiesta ricevuta; il server sta continuando il processo
- **2xx** (successo); richiesta ricevuta; compresa, accettata, e servita

- **3xx** (reindirizzamento): è necessario intraprendere ulteriori azioni per completare la richiesta
- **4xx** (errore del cliente): la richiesta contiene una sintassi errata o non può essere compresa
- **5xx** (errore del server): il server non è riuscito a soddisfare una richiesta apparentemente valida

PUT vs POST

Quando si creano nuove risorse, si utilizza **POST** se il server sceglie l'URI mentre si utilizza **PUT** se il cliente sceglie l'URI; POST può essere qualcosa come “Process this”, fa qualcosa e restituisce qualcosa. Di solito maschera una chiamata di procedura remota, un’invocazione di funzione.

Remote Procedure Call

Le **Procedure Remote Call** rappresentano un’evoluzione del tradizionale protocollo di chiamata di procedura, adattato per un ambiente distribuito. Questo meccanismo consente di **invocare procedure su un computer remoto** con una semantica familiare e una facile implementazione, grazie alla sua affinità con il modello client-server. I vantaggi sono:

- **Semantica nota:** l’utilizzo delle RPC è intuitivo perché si basa sulla chiamata di procedura
- **Facilità di implementazione:** le RPC si integrano agevolmente nel modello client-server, rendendo il loro utilizzo relativamente semplice.

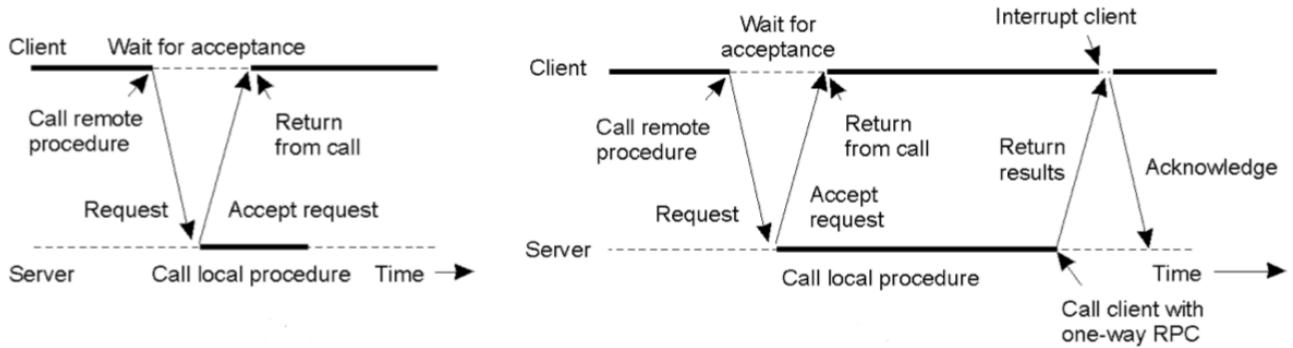
Gli svantaggi invece sono:

- **Implementazione esplicita:** le RPC richiedono che il programmatore gestisca tutti gli aspetti dell’interazione, il che può essere mitigato attraverso l’uso di modelli a componenti più avanzati
- **Natura statica:** le RPC sono codificate direttamente nel codice sorgente dei programmi, ma è possibile superare questa limitazione impiegando chiamate indirette
- **Assenza di concorrenza:** le RPC sono bloccanti, ma si può ovviare a questo problema utilizzando thread o chiamate asincrone per gestire le operazioni in parallelo.

Il modello RPC asincrona prevede che il chiamato possa restituire il risultato della chiamata **chiamando una procedura che è stata passata dal chiamante**, mentre prevede che il controllo venga restituito al chiamante dopo che il **middleware del chiamato ha ricevuto il messaggio di invocazione**. Il server restituisce il controllo alla ricezione della richiesta.

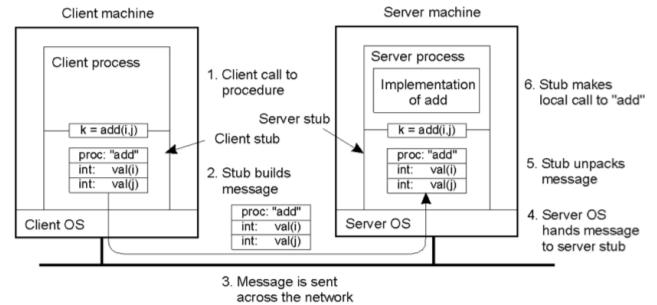
Il modello RPC nasconde la complessità del modello client-server, differenziandosi principalmente nella gestione delle informazioni scambiate, che avviene tramite parametri, con due tipi di interazione:

- **RPC asincrona senza risposta**, dove il client invoca una procedura sul server senza attendere una risposta immediata
- **RPC asincrona con risposta posticipata**, dove il client riceve una risposta tramite una seconda RPC asincrona nota come *call-back*



Si basano sullo scambio di messaggi, un concetto fondamentale già esplorato nel modello di comunicazione a messaggi. Per poter effettuare una RPC, è necessario l'intervento di un **middleware** che trasforma le chiamate locali in chiamate remote; coinvolge sia il lato client che il lato server.

Lato Client	<p>Programma: determina quale procedura chiamare</p> <p>Middleware: gestisce l'interazione del programma cliente</p> <p>Middleware: localizza la macchina server</p> <p>Middleware: processa i parametri per la trasmissione</p>
Lato Server	<p>Middleware: identifica la procedura da invocare</p> <p>Middleware: trasferisce i parametri alla procedura</p> <p>Middleware: elabora il risultato della chiamata</p>



Stub

Gli stub sono utilizzati per **simulare comportamenti locali** sia per il chiamante che per il chiamato, e per gestire la comunicazione tra processi remoti. Il processo di computazione remota attraverso RPC segue questi passaggi:

1. Il client invoca una procedura
2. Lo stub lato client costruisce il messaggio
3. Il messaggio viene inviato attraverso la rete
4. Il sistema operativo del server consegna il messaggio allo stub lato server
5. Lo stub lato server estrae il messaggio
6. Lo stub lato server effettua una chiamata locale alla procedura desiderata

Nel processo remoto avviene una **replica dello stack**: i dati vengono impacchettati, inviati e poi spacciati per creare copie locali, con un'operazione nota come **mm/unmarshalling** dei dati. Con l'unmarshalling, si traducono i dati ricevuti in un formato adeguato alla chiamata locale. Con il marshalling, si permette il passaggio di parametri tra web client e web server.

Oggetti distribuiti

Gli oggetti nel contesto della programmazione orientata agli oggetti sono strutture che **incapsulano dati** e **definiscono operazioni** su questi dati, e specificano i metodi di accesso attraverso le interfacce.

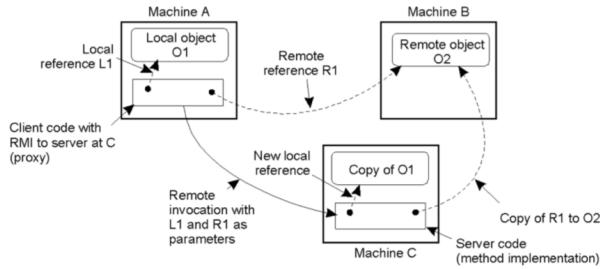
A compile-time, gli oggetti sono definiti tramite interfacce e classi in linguaggi come Java e C++. A run-time, diventano accessibili attraverso i *wrappers*; gli oggetti possono essere **persistenti**, ovvero salvati e recuperati da una memoria non volatile, o **transienti**, esistendo solo temporaneamente durante l'esecuzione del programma; il loro valore non viene salvato durante la sua serializzazione/deserializzazioni, e che una variabile con questo attributo non verrà inizializzata.

Una volta che una variabile transient è deserializzata, in modo da averla inizializzata, deve essere inizializzata manualmente nel codice, altrimenti il suo valore rimarrà *null*. Quindi, i valori transient e static non possono essere serializzati.

Il riferimento agli oggetti remoti è un concetto chiave nella programmazione distribuita. Per esempio, in C++, la differenza è significativa:

- Un **puntatore** è una variabile che contiene un indirizzo di memoria, e può essere inizializzato in qualsiasi momento; può puntare a diversi oggetti nel corso della sua vita, e può avere un valore NULL
- Un **riferimento** è come un alias per una variabile esistente, e deve essere inizializzato al momento della definizione; non può essere riassegnato, e non può essere NULL
 - In Java invece è una variabile che contiene informazioni logiche per accedere a un oggetto, è immutabile e fortemente tipizzato. Un **riferimento distribuito** include l'indirizzo IP della macchina, la porta del server e l'identificazione dell'oggetto (ID, metodo e argomenti).

Il passaggio di parametri per reference o per valore succede come:

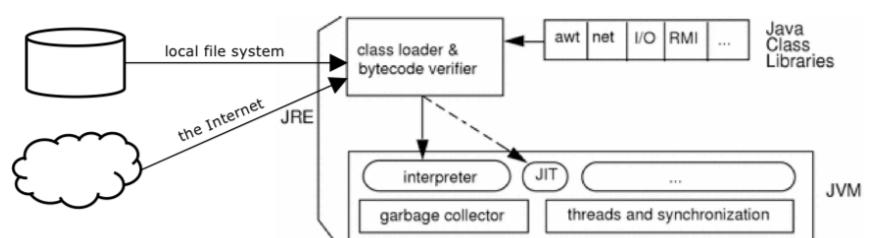


Remote Method Invocation

È un middleware che estende l'approccio orientato agli oggetti al contesto distribuito, supportando l'ereditarietà e l'invocazione di metodi tra oggetti su macchine virtuali distinte. RMI si basa sulla portabilità del *bytecode* e sulla macchina virtuale Java, offrendo servizi come il caricamento dinamico delle classi, la gestione di oggetti remoti, e il garbage collection di oggetti remoti. Non si usano i puntatori, perché non si conosce l'area di memoria che l'altro processo occuperà.

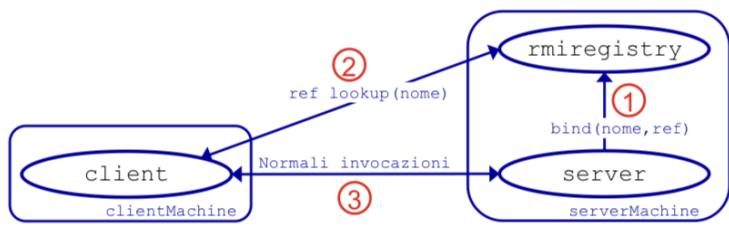
Class loader

È responsabile del caricamento delle classi, e il formato bytecode consente il trasferimento dei file *.class* da librerie, sistemi di file locali o domini di rete. Il caricamento avviene dinamicamente quando viene eseguito un comando *new* o quando un oggetto RMI viene trasferito.



Questo può avvenire **per valore** o **per riferimento**. I tipi primitivi e gli oggetti non remoti serializzabili sono passati per valore, mentre i riferimenti ad oggetti remoti sono passati per valore per consentire invocazioni remote. La serializzazione è fondamentale per memorizzare e trasferire lo stato degli oggetti, e il meccanismo di loading dinamico di Java consente di passare solo le informazioni essenziali sullo stato.

RMI inoltre utilizza un **rmiregistry** per la gestione dei nomi e dei riferimenti agli oggetti remoti, il server pubblica il riferimento e il nome dell'oggetto remoto nel registry, mentre il client ottiene il riferimento all'oggetto e accede ai metodi remoti tramite **lookup**; questo sistema consente la comunicazione e l'interazione tra oggetti distribuiti in una rete.



Object serialization

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, e per definire oggetti persistenti. Converte l'oggetto in una sequenza di byte, che può essere inviata sulla rete. Il meccanismo di loading dinamico di Java permette di passare solo le informazioni essenziali sullo stato: la descrizione della classe può essere caricata a parte. La serializzazione usa il metodo `writeObject(Object obj)` della classe `Object` che attraversa tutti i riferimenti contenuti in `obj` per costruire una rappresentazione completa del grafo.

I tipi base sono serializzabili in modo nativo, ed è possibile creare classi serializzabili implementando l'interfaccia **Serializable**; occorre però ridefinire i metodi.

```

private void writeObject(java.io.ObjectOutputStream out) throws
IOException;

private void readObject(java.io.ObjectInputStream in) throws
IOException, ClassNotFoundException;

```

Quindi, quando si implementa una classe `Serializable`, stai dicendo alla JVM che gli oggetti di questa classe possono essere serializzati, e quindi non è obbligatorio implementare alcun metodo. È una classe **marker**, ovvero senza metodi. Possiamo implementare quei due metodi in modo che si possa controllare il processo.

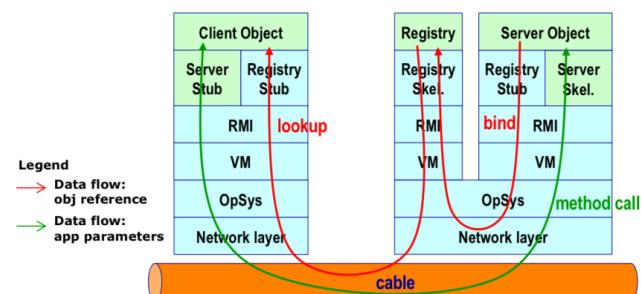
La classe Naming

La classe `Naming` dà accesso diretto alle funzionalità del RMI registry: i metodi sono statici, i parametri sono stringhe in formato URL riferiti al registry e all'oggetto remoto considerato, con default host `localhost` e default port `1099`:

- **Public static Remote lookup(String name) throws NotBoundException, MalformedURLException, RemoteException:** restituisce un riferimento, uno stub, all'oggetto associato al nome specificato
- **Public static void bind(String name, Remote obj) throws AlreadyBoundException, MalformedURLException, RemoteException:** collega (bind) il nome specificato all'oggetto remoto

- **Public static void rebind(String name, Remote obj) throws RemoteException,**
MalformedURLException: collega (bind) il nome specificato all'oggetto remoto, cancellando i collegamenti esistenti
- **Public static String[] list(String name) throws RemoteException,**
MalformedURLException: restituisce i nomi, in formato URL, degli oggetti del registry
- **Public static void unbind(String name) throws RemoteException,**
NotBoundException, MalformedURLException: distrugge il collegamento (bind al nome specificato)
 - o Parametri: **name** (a name in URL format), **obj** (a reference for the remote object)
 - o Eccezioni: **AlreadyBoundException, NotBoundException, RemoteException** (if registry could not be contacted), **AccessException** (if this operation is not permitted), **MalformedURLException** (if the name is not an appropriately formatted URL)

Il server pubblica il reference e il nome dell'oggetto remoto nel Registry invocando il metodo bind; il client ottiene il reference all'oggetto invocando il metodo lookup, e il client accede all'oggetto remoto il reference e i nomi dei metodi remoti.



Creazione di oggetti remoti

Per implementare oggetti remoti si utilizza *java.rmi.Remote*, che è un'interfaccia fondamentale per la creazione di classi remote, comunemente note come server. La procedura per creare una classe remota include tre passaggi chiave:

1. **Definizione dell'interfaccia della classe remota:** stabilisce le fondamenta per la comunicazione remota
2. **Implementazione dell'Interfaccia:** implementarla concretamente
3. **Creazione e registrazione di un Server:** si sviluppa un server che crea l'oggetto remoto e lo registra nel Registry.
 - a. Quindi, crea un oggetto locale che fornisce il servizio desiderato
 - b. Registra questo oggetto nel RMI registry presente sullo stesso host, assegnandogli un nome pubblico

```
MyRemoteObject myObj = new
MyRemoteObject();
Registry localRegistry =
LocateRegistry.getRegistry();
localRegistry.bind("public
name", myObj);
```

Per applicazioni reali, è essenziale considerare le regole di visibilità ed accesso gestite tramite il *SecurityManager*. Questo aspetto è cruciale per la sicurezza.

Architettura multilivello

Un'applicazione è composta da componenti che collaborano per svolgere un compito.

Questa decomposizione di solito avviene su tre livelli seguendo il livello **MVC**, *Model-View-Controller*:

- **Presentazione:** il livello che l'utente vede e con cui interagisce
- **Logica:** il cuore dell'applicazione, dove risiede la logica di business
- **Dati:** il livello che gestisce la persistenza e il recupero dei dati

Divisione del carico

Una questione importante è come organizzare il sistema, ovvero come distribuire i componenti sulle macchine client/server, e cosa può essere implementato lato client e lato server.

Esistono varie configurazioni:

- **Metà interfaccia utente sul client e metà sul server, con applicazione e database sul server:** distribuisce il carico di lavoro tra client e server. Mentre il server gestisce la *logica dell'applicazione* e il *database*, il client si occupa di una *parte dell'interfaccia utente*, alleggerendo il carico sul server e migliorando la reattività
- **Interfaccia utente sul client, con applicazione e database sul server:** il client si occupa esclusivamente della presentazione, offrendo agli utenti un'interfaccia ricca e reattiva. Il server si concentra sulla *logica dell'applicazione* e sulla *gestione dei dati*, garantendo che le operazioni complesse siano gestite efficacemente
- **Interfaccia utente sul client, metà applicazione sul client e metà sul server, con database sul server:** bilancia il carico; alcune funzionalità vengono eseguite *localmente* per una *risposta rapida*, mentre il server si occupa delle operazioni più pesanti e della gestione dei dati
- **Interfaccia utente sul client, applicazione sul client, database sul server:** il client assume la maggior parte delle responsabilità, eseguendo *l'intera applicazione localmente*. Il server si limita a fornire dati, il che può ridurre il traffico di rete e migliorare le prestazioni
- **Interfaccia utente e applicazione sul client, metà database sul client e metà sul server:** configurazione ibrida in cui il database è diviso tra client e server, e può essere utile per ottimizzare l'accesso ai dati e distribuire il carico di lavoro in modo più equilibrato

Caratteristiche del web e Ajax

Le pagine web si **ricaricano sempre** e non vengono **mai aggiornate dinamicamente**; gli utenti devono attendere il ricaricamento completo della pagina anche se è necessario un solo

pezzo di dati, a causa delle restrizioni della singola richiesta/risposta. È possibile, quindi, rendere una pagina reattiva ad eventi generati dall'utente, ma anche eseguire *ciclicamente funzioni JavaScript* indipendentemente da stimoli da parte dell'utente.

Ajax è una tecnica di sviluppo per creare applicazioni web interattive. JavaScript è usato per il **tier di presentazione**, ma può realizzare almeno parte della logica applicativa. L'utilizzo di servizi di terze parti tramite Ajax richiede la **conoscenza delle specifiche API** e formati di interscambio prescritto dal servizio utilizzato. Non si tratta di una nuova tecnologia, ma piuttosto di un modello che assiste nelle interfacce utente e rende le pagine web meno leggibili e collegabili dalle macchine. **Permette di effettuare richieste al server asincrone.** I componenti riguardano:

- **HTML (o XHTML) e CSS** per la presentazione delle informazioni
- **Document Object Model (DOM)** per la visualizzazione dinamica e l'interazione
- **XML e XSLT** per lo scambio e la manipolazione dei dati, anche se **JSON** è spesso preferito
- **Oggetto XMLHttpRequest** per il recupero asincrono dei dati dal server web
- **JavaScript** che lega insieme tutti gli elementi e può essere sostituito da TypeScript (variante a tipizzazione forte)

Abbiamo anche, per esempio, le applicazioni **Ajax-Json**, le quali sono più efficienti di quelle di Ajax-XML, perché i dati JSON sono in formato compatibile JavaScript, ed è più leggero di XML.

JavaScript

JavaScript è un linguaggio di scripting orientato agli oggetti e non tipizzato, interpretato da un motore di esecuzione, solitamente il browser (se si vuole fare esecuzione lato server, bisogna usare Node.js). JavaScript è fondamentale perché **permette di rendere le pagine HTML dinamiche** e di **effettuare richieste HTTP al server in modo trasparente** per l'utente, rendendo **asincrona** la comunicazione tra browser e web server.

Questo comporta alcuni problemi, come la rottura del pulsante “indietro” del browser, cambiamenti inaspettati di parti della pagina, difficoltà nel segnalibro di uno stato particolare, aumento delle dimensioni del codice sul browser che influisce sul tempo di risposta, difficoltà nel debug, sorgente visibile aperta a hacker o plagio, e carico sul server a causa delle richieste asincrone che possono essere operazioni pesanti.

Trasmissione dei dati nelle applicazioni RIA

La trasmissione dei dati tra server e applicazioni RIA può avere diverse tecnologie alternative per gestire questa comunicazione:

- **SOAP (Simple Object Access Protocol):** protocollo standard per lo scambio di messaggi strutturati basato su XML

- **XML-RPC** (Remote Procedure Call con XML per l'encoding): protocollo che permette di eseguire chiamate a procedure remote utilizzando XML come formato di codifica
- **JSON (JavaScript Object Notation)**: formato leggero per lo scambio di dati, facile da leggere e scrivere per gli umani, e semplice da interpretare e generare per le macchine
- **AMF (Action Message Format basato su SOAP)**: formato binario utilizzato per serializzare oggetti ActionScript

Struttura di una pagina web

La struttura di base è definita dal seguente scheletro:

```
<!DOCTYPE html>
<html>
<body>
<!-- Incluso il contenuto da visualizzare, l'interfaccia utente, spazio di
input/output -&gt;
&lt;script&gt;
// Ospita il codice JavaScript
&lt;/script&gt;
&lt;/body&gt;
&lt;/html&gt;</pre>

```

JavaScript non dispone di funzioni incorporate per la stampa o la visualizzazione di dati.

Tuttavia, può visualizzare i dati in diversi modi:

- **Utilizzando innerHTML**, per scrivere all'interno di un elemento HTML, sostituendo il suo contenuto.
- **Utilizzando window.alert()** per aprire una finestra di dialogo con il contenuto specificato
- **Utilizzando console.log()** per scrivere messaggi di debug nella console del browser
- **Utilizzando document.write()** per sostituire l'intera pagina con un nuovo contenuto

```
<p id="demo"></p>
<script>
Document.getElementById("demo").inne
rHTML = "Hello world!";
</script>

window.alert("Hello world!");

console.log("Letti i primi "
+ count + " valori.");
```

Controllo e input dei dati

L'invocazione di una funzione avviene quando “qualcosa” la chiama, ovvero *quando si verifica un evento*, o quando viene *invocata dal codice JavaScript*, o *automaticamente*.

Gli eventi JavaScript sono **cose** che accadono agli elementi HTML: quando JavaScript viene utilizzato nelle pagine HTML può reagire a questi eventi; un'applicazione RIA è tipicamente reattiva, nel senso che non è possibile identificare staticamente un flusso di controllo unitario. Il programma principale si limita a inizializzare l'applicazione, stanziando gli osservatori e associandovi gli opportuni handler.

Si può associare uno o più gestori di eventi ad ogni elemento del DOM HTML che genera eventi utilizzando la funzione `addEventListener`:

```
element.addEventListener(event, function);
```

tipi di eventi:
click, mouseDown, etc

La funzione che deve essere chiamata
può essere anche anonima

Variabili

In ES5 le variabili venivano dichiarate utilizzando la parola chiave **var**, che definisce la variabile con un **ambito globale o di funzione**. È debolmente tipizzato, e il tipo viene stabilito ogni volta a runtime. In ES6 si possono poi usare due nuovi modi:

```
var x = 10; {  
    const x = 2;  
    x = 4; }
```

- **Let**: definisce variabili con **ambito di blocco**, limitando la loro visibilità all'interno del blocco in cui sono dichiarate
- **Const**: simile a let in termini di **ambito di blocco**, ma utilizzata per definire variabili con un valore costante che non può essere modificato. Importante la differenza tra variabili composte e singole: le variabili semplici sono **immodificabili**, mentre i valori delle variabili composte possono cambiare.

Una buona pratica è utilizzare `var` solo per le variabili globali, mentre `let` e `const` dovrebbero essere usati in tutti gli altri casi. In particolare, `const` dovrebbe essere usata il più possibile per evitare cambiamenti indesiderati del valore delle variabili.

Funzioni

In ES5 le funzioni possono essere definite in due modi:

```
function f(x,y) { return x; }  
var f = function(x,y) {  
    return x; };
```

ES6 invece introduce le **arrow functions**, che permettono di scrivere funzioni in modo più conciso.

La keyword `return` e le parentesi graffe `{}` possono essere omesse solo quando la funzione ha una sola istruzione; tuttavia, è buona abitudine utilizzarle sempre per mantenere la chiarezza del codice.

```
const f = (x, y) => x * y;
```

Manipolazione delle stringhe

Le stringhe in JavaScript sono utilizzate per memorizzare e manipolare testo. Possono essere scritte tra virgolette singole o doppie, e hanno diverse proprietà e metodi per lavorare con esse.

Nome	Metodo
Definizione	<code>let carName1 = "Volvo XC60"; let carName2 = 'Volvo XC60';</code>
Lunghezza	<code>let sln = carName1.length;</code>
Concatenazione	<code>let txt1 = "John"; let txt2 = "Doe"; let tx3 = txt1 + " " + txt2;</code>
IndexOf	<code>let position = string.indexOf(string, startingPosition);</code>
LastIndexOf	<code>let pos = str.lastIndexOf(string, startingPosition);</code>
Inizia con	<code>const.startsWith(string, startPosition);</code>
Finisce con	<code>const.endsWith(string, endPosition);</code>
Dividi	<code>let pos = str.slice(startPosition, endPosition);</code>

Array

Nome	Metodo
Aggiungi	<code>fruits.push("Lemon");</code>
Rimuovi ultimo elemento	<code>let y = fruits.pop();</code>
Rimuovi primo elemento e shifta	<code>let z = fruits.shift();</code>
Aggiungi primo elemento e shifta	<code>fruits.unshift("Banana");</code>
Dividi un array	<code>const.splice(whereNewElement, howManyRemoved, whatNewElement);</code>

In JavaScript è valido assegnare un valore a un *indice che supera la lunghezza attuale* dell'array; l'array viene espanso automaticamente, e gli indici intermedi vengono riempiti con *undefined*.

Classi

JavaScript ha introdotto il concetto di classi, una caratteristica fondamentale per la definizione di **prototipi di oggetti**. Le classi servono a dichiarare più entità dello stesso tipo, semplificando la creazione di nuove istanze.

Tutti i valori in JavaScript sono oggetti. La creazione di un oggetto avviene con una new seguita dal nome della classe, ed eventuali parametri. *GetElementById* seleziona un singolo elemento DOM con un ID specifico.

Iterazione su variabili multivalore

Esistono forme più compatte per scandire array e oggetti rispetto al ciclo for.

For/of considera tutti i valori di una struttura iterativa	<pre>const numbers = [1, 2, 3, 4, 5, 6]; let sum = 0; for (let x of numbers) { sum = sum + x; }</pre>
For/in considera tutte le proprietà di un oggetto	<pre>let str = ""; for (let x in person) { str += x + ": " + person[x] + "
"; }</pre>

Elementi di programmazione

filter() crea un nuovo array con gli elementi che passano un test. Non esegue la funzione per gli elementi senza valori, e non cambia l'array originale	<pre>const over18 = numbers.filter(function); function function(value, index, array) { return value > 18; }</pre>
reduce() esegue una funzione su ogni elemento dell'array per ridurlo a un singolo valore. Fa da sinistra a destra, non riduce l'array originale	<pre>let sum = numbers.reduce(myFunction); function myFunction(total, value, index, array) { return total + value; }</pre>
forEach() chiama una funzione una volta per ogni elemento dell'array	<pre>numbers.forEach(myFunction); function myFunction(value, index, array) { txt += value + "
"; }</pre>
map() crea un nuovo array eseguendo una funzione su ogni elemento dell'array	<pre>const numbers2 = numbers1.map(myFunction); function myFunction(value, index, array) { return value * 2; }</pre>

Node.js

Node.js è una piattaforma basata su V8, il *motore JavaScript di Chrome*, che permette di sviluppare applicazioni web veloci, e scalabilità. Ha certe caratteristiche chiave:

- **V8** è un esecutore open source altamente efficiente che occupa poche risorse
- **Modelli di I/O, non bloccante, e ad eventi**, dove Node.js usa un *modello di I/O asincrono*. Invece di bloccare l'esecuzione del programma durante le operazioni di I/O,

come la lettura di file o le richieste di rate, Node.js assegna **callback** agli eventi, e continua l'esecuzione. Quando l'operazione I/O è completata, viene eseguita la callback.

- **NPM (Node Package Manager)** è il gestore dei moduli per Node.js; posso installare, gestire e condividere moduli di software.

Approccio asincrono

Node.js sfrutta l'approccio asincrono per massimizzare l'efficienza: invece di attendere il completamento di un'operazione prima di passare alla sufficienza, Node.js assegna **callback agli eventi**, e continua l'esecuzione.

Nell'approccio **sincrono**, l'esecuzione attende il completamento di un'operazione prima di proseguire.

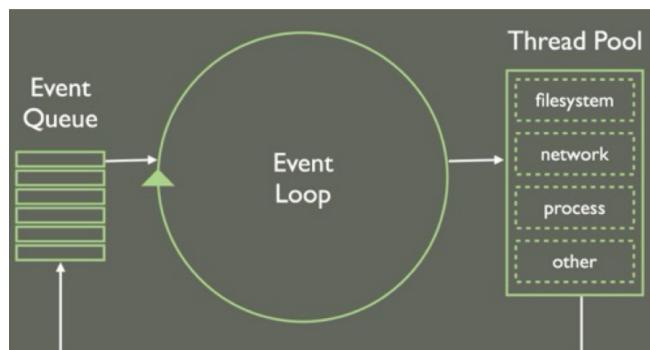
Nell'approccio **asincrono** l'esecuzione non attende il completamento di un'operazione; viene invece definita una callback da eseguire quando l'operazione è terminata.

```
const dato = getRepresentationSync(url);
alert(dato);
```

```
getRepresentationAsync(url, function(dato) {
  alert(dato);
});
```

Event model

Node.js utilizza un singolo thread con un event loop; ogni evento viene eseguito **completamente** prima del successivo, e la risposta a una richiesta Ajax non è sempre immediata. Con questo metodo, *non è necessario preoccuparsi della mutua esclusione*, ma *un evento lungo può bloccare l'interfaccia utente*.



Esecuzione dei programmi

L'esecuzione dei programmi si basa su un **Single Event Loop**:

1. L'applicazione inserisce una **richiesta** (evento) nella coda
2. L'event loop **estrae la richiesta** dalla coda
3. L'event loop **assegna un esecutore** (thread) alla richiesta e alla callback
4. L'esecutore **esegue il compito**, e inserisce la richiesta di callback

5. L'event loop **estrae la richiesta** di callback
6. L'event loop **assegna un esecutore** alla callback
7. L'applicazione percepisce l'effetto dell'esecuzione della richiesta originale.

Moduli

I moduli in Node.js sono unità di codice autonomi che consentono di organizzare, suddividere e riusare il codice in maniera efficiente. Ha alcuni punti chiave:

- **Incapsulamento:** i moduli consentono di incapsulare funzionalità specifiche in file separati; questo aiuta a mantenere il codice pulito, e a evitare l'ingombro di un unico file enorme.
- **Riuso del codice:** i moduli permettono di scrivere una volta e utilizzare ovunque.
- **Esportazione e importazione:** in node.js, si possono esportare parti di un file come moduli utilizzando *module.exports*, e poi importarli con *require*.

Ha i seguenti vantaggi:

- **Organizzazione:** i moduli aiutano a organizzare il codice in unità logiche e indipendenti
- **Manutenzione semplificata:** se una parte del codice cambia, devi modificarla solo nel modulo corrispondente
- **Riuso:** si possono condividere moduli tra progetti o utilizzarli in più parti dello stesso progetto

Creazione di un modulo

Per creare un modulo, si seguono i seguenti passaggi:

1. **Crea un file:** crea un nuovo file per il tuo modulo
2. **Definisci le funzionalità:** si definiscono le funzionalità che vuoi esportare; ecco un esempio.
3. **Importa il modulo:** in un altro file, puoi importare il modulo utilizzando *require*

```
//mioModulo.js
function saluta(nome) {
    return 'Ciao';
}
Module.exports = { saluta };

//main.js
const mioModulo =
require('./mioModulo');
```

Modulo File System

Il modulo *fs* fornisce funzioni per la gestione dei file, come *lettura di file*, *creazione di file*, *aggiornamento di file*, *eliminazione di file*, e *rinomina di file*. Mette a disposizione una modalità asincrona e sincrona di lettura del file.

Ecco un esempio:

```
ASINCRONA:
fs.readFile('file.txt', 'utf8', function(err, nome){
    if (err) throw err;
    console.log('Callback ~ ' + aut + nome);
```

```

const fs = require('fs');
let out = 'Il nome letto è: ';
let nome;
nome = fs.readFile('file.txt', 'utf8');
console.log('Dopo la funzione - ' + out + nome);
Output: Dopo la funzione - il nome letto è: Mario Rossi

```

Modulo Express.js

Express.js è un framework web per Node.js; esso semplifica la creazione di server web, consentendo di gestire rotte, richieste http e risposte in modo efficiente.

Per iniziare, crea una cartella vuota per il tuo server; all'interno, puoi installare Express come dipendenza utilizzando `npm install express`.

Permette di definire rotte per gestire richieste HTTP, ad esempio associando una determinata URL a una funzione che elabora la richiesta e restituisce una risposta. Utilizza **middleware** per eseguire operazioni tra la ricezione di una richiesta e l'invio di una risposta.

Express **semplifica** la creazione di server web e la gestione delle rotte, è ampiamente utilizzato e ha una vasta comunità di sviluppatori che contribuiscono con moduli e risorse. Si possono anche aggiungere middleware personalizzati per estendere le funzionalità di Express.

Creiamo un file **server1express.js** con il seguente codice:

```

var express = require('express');
var app = express();
app.get('/', function(req,res){ console.log("Got a GET request for home");
                                res.send('Hello GET') });

//This responds a POST request for the homepage
app.post('/', function (req, res){ console.log("Got a POST request");
                                    res.send('Hello POST') });

//This responds a DELETE request for the /users/# page
App.delete('/users/:d' function (req, res){
    console.log("Got a DELETE request for /user/#");
    res.send('Hello DELETE') });

//This responds a GET request for the /users page
app.get('/users', function(req, res){
    console.log("Got a GET request for /users"); res.send('Page Listing') });

//This responds a GET request for back, abxcd, ab123cd, and so on
app.get('/ab*cd', function(req, res) {
    console.log("Got a GET request for /ab*cd");
    res.send('Page Pattern Match') });

var server = app.listen(8081, function() {
    var host = server.address().address

```

```
var port = server.address().port
console.log("Example app listening at http://%s:%s", host, port) })
```

Dati semantici

Il web semantico rappresenta un'estensione del web tradizionale che promuove l'uso di formati di dati comuni per facilitare uno scambio di dati significativo tra macchine. Quando i motori di ricerca trovano e indicizzano i contenuti del web, la maggior parte dei dati presenti nelle pagine web è non strutturata. La specifica HTML da sola **non definisce un vocabolario condiviso** che ci permetta di descrivere gli elementi di una pagina, e le loro relazioni, in modo standard e non ambiguo.

Dati collegati

Sono un insieme di buone pratiche per pubblicare e collegare dati strutturati sul web, in modo che le risorse web possano essere interconnesse in un modo che permetta ai computer di comprendere automaticamente il tipo e i dati di ciascuna risorsa. È particolarmente attraente perché qualsiasi applicazione che comprenda il tipo di una risorsa può quindi *raccogliere, elaborare e aggregare* dati da diverse fonti in modo uniforme, indipendentemente dal luogo di pubblicazione. Possono essere rappresentati con **fatti** (relazioni tra soggetto e oggetto), **proprietà delle entità** (caratteristiche specifiche), **elaborazione complessa**, e **integrazione di testi, immagini e video**.

Arricchimento del JSON con metadati

Il JSON può essere arricchito con metadati per migliorarne la chiarezza e l'usabilità; questi sono **informazioni aggiuntive** che qualificano il contenuto. Arricchire un JSON comporta, quindi, un *associare nomi con URI, associare valori letterali con tipi che si riferiscono a un contesto noto, collegare nomi e valori attraverso relazioni*.

JSON-LD

JSON-LD offre un modo semplice per aumentare semanticamente i documenti JSON aggiungendo informazioni di contesto e collegamenti ipertestuali per descrivere la semantica dei diversi elementi di un oggetto JSON. Può essere inoltre espresso in diversi formati, tra cui *compattato, tabulare, visuale e RDF*.

È come un modo per *utilizzare i dati collegati in ambienti di programmazione basati sul web*, per costruire servizi Web interoperabili, e per memorizzare dati collegati in motori di archiviazione basati su JSON. Estende quindi il linguaggio con diverse parole chiave rappresentate dai **nomi speciali** delle proprietà JSON che iniziano con il segno @:

- **Context**: un URL che fa riferimento a uno schema particolare
- **Id**: un identifier univoco, di solito un URI
- **Type**: un URL che fa riferimento al tipo di un valore

Rappresentazione della conoscenza in formati di dati strutturati

La conoscenza può essere rappresentata in formati di dati strutturati come:

- **Vocabolario**: un elenco di termini, ognuno con una definizione che include i loro tipi, che diventa un insieme di URI (es. *FOAF* e *Dublin Core*)
- **Grafo**: una rete dove i termini sono collegati da proprietà, ognuna con una definizione, che diventa un altro insieme di URI
- **Ontologia**: una specificazione formale di un insieme di concetti all'interno di un dominio e delle relazioni tra quei concetti

Chiunque può creare una rappresentazione della conoscenza che descrive le entità. Il problema con questo approccio è che **solo le applicazioni nello stesso ecosistema comprenderebbero questa rappresentazione**.

Interoperabilità e repository collaborativi

Ecosistemi diversi dovrebbero condividere la stessa rappresentazione della conoscenza per diventare **interoperabili**; un approccio popolare è di fare affidamento su repository collaborativi leggeri come:

- **Schema.org**: progetto collaborativo che fornisce una raccolta di schemi condivisi per dati strutturati. Ha lo scopo di *standardizzare la marcatura dei dati* per migliorare l'ottimizzazione dei motori di ricerca, facilitare l'integrazione dei dati e l'interoperabilità tra diverse piattaforme, e migliorare l'usabilità e la chiarezza dei dati web
- **Wikidata**: conoscenza collaborativa gratuita che *raccoglie dati strutturati da Wikipedia*, e *mantiene un grafo* della conoscenza che organizza entità, tipi e proprietà, utilizzando identificatori unici per una gestione precisa dei dati.
 - Utilizzando il prefisso *wd* possiamo creare collegamenti alle entità di Wikidata.

RDF

È un modello di dati basato sulle **triplette** (l'unità base per organizzare le informazioni), sui **grafi diretti orientati ed etichettati** (insiemi di triplette), **URI e IRI** (identificatori unici per

risorse, con IRI che permettono l'identificazione di risorse nelle lingue native degli utenti). Con etichettato intende dire che *ha i nomi sugli archi, orientato perché ha le frecce*. I vantaggi sono:

- **Indipendenza**: organizzazioni indipendenti possono inventare predicatori
- **Scalabilità**: le proprietà RDF sono semplici triplets, facili da gestire e ricercare
- **Interoperabilità**: le proprietà RDF possono essere convertite in XML, facilitando lo scambio di dati.

In RDF, i dati primitivi, o literal, sono utilizzati per rappresentare valori di dati semplici, e **non sono risorse identificabili di per sé**. Sono rappresentati senza un URI, ma sono parte integrante della tripletta.

Turtle è una sintassi testuale per RDF che consente di **scrivere un grafo RDF in una forma testuale** compatta e naturale, con abbreviazioni per i modelli d'uso comune e i tipi di dati. Viene utilizzato per **serializzare i dati RDF**, facilitando la distribuzione e la gestione dei dati tra più macchine. I sistemi RDF nativi applicano **schemi di partizionamento differenziati** per abbracciare sia la *località* (per i vertici normali) sia il *parallelismo* (per i vertici di indice) durante l'elaborazione delle query.

Ogni tupla è composta da tre componenti: **soggetto** (l'entità di cui si sta parlando), **predicato** (la proprietà o relazione dell'entità), **oggetto** (il valore della proprietà o un'altra entità correlata).

Cloud Computing

L'**Internet of Things** è un Sistema di oggetti fisici che possono essere scoperti, monitorati, controllati, o con cui è possibile interagire tramite dispositivi elettronici che comunicano attraverso varie interfacce di rete, e possono essere collegati a internet.

Sono oggetti fisici che sono stati digitalmente arricchiti con alcune caratteristiche specifiche, come **sensori, attuatori, capacità di calcolo, e interfacce di comunicazione**. La sfida principale è di *creare un ecosistema di sistemi integrati e senza soluzione di continuità*; questo implica, dunque, di utilizzare il web come piattaforma programmabile, integrare oggetti fisici tramite gemelli digitali, e sfruttare il cloud computing come paradigma per costruire sistemi complessi.

Edge Computing

L'Edge Computing è una tecnologia emergente che **sposta l'elaborazione dei dati e i servizi di analisi più vicino alla fonte dei dati stessi**, piuttosto che dipendere esclusivamente dai

data center centrali o dal cloud. Questo approccio riduce la *latenza*, diminuisce i costi di trasferimento dati, e affronta preoccupazioni di sicurezza e privacy, migliorando l'efficienza complessiva dei sistemi IoT.

Abbiamo varie sfide:

- **Aumento delle capacità dei nodi edge:** essenziale incrementare le capacità e le risorse dei nodi edge per gestire grandi volumi di dati, e processi complessi direttamente sul campo
- **Costo di trasferimento dati:** il trasferimento di grandi quantità di dati verso e dai data center centrali è costoso. Ridurre la dipendenza dai data center può comportare significativi risparmi economici
- **Sicurezza e privacy:** trasferire dati sensibili attraverso la rete può esporli a rischi di sicurezza e violazioni della privacy. Elaborare i dati ai margini riduce la necessità di trasferimenti di dati sensibili, migliorando la sicurezza
- **Essenzialità dei dati:** non tutti i dati raccolti ai margini sono cruciali; filtrare e analizzare i dati vicino alla fonte consente di inviare al cloud solo le informazioni essenziali, ottimizzando l'uso delle risorse di rete.

Gli obiettivi sono invece:

- **Analisi dei dati vicino alla fonte:** comprendere quando e quali dati possono essere analizzati vicino al luogo di raccolta è fondamentale: questo implica l'**orchestrazione** di servizi e funzioni direttamente ai margini della rete, migliorando la reattività e l'efficienza dei sistemi.
- **Definizione dei nodi edge:** identificare cosa sia un nodo edge, e come si connetta all'IoT e cloud è cruciale per costruire una rete **efficiente e interoperabile**.
- **Sfruttamento efficace delle risorse:** include bilanciare l'elaborazione tra nodi edge e data center centrali in modo da ottimizzare le prestazioni, e minimizzare i costi.

Cloud computing

Il cloud computing è un paradigma che offre risorse IT scalabili ed elastiche, come servizio a clienti esterni utilizzando tecnologie internet. Gli utenti possono accedere a queste risorse senza necessità di conoscenze approfondite sull'infrastruttura sottostante.

Il cloud computing permette l'**accesso on-demand** a una serie di risorse informatiche condivise (reti, server, storage, applicazioni, servizi, etc.). Queste risorse possono essere rapidamente fornite, e rilasciate con un minimo sforzo di gestione o interazione con il fornitore di servizi. Si basa quindi su cinque caratteristiche essenziali:

- **Autoservizio on-demand:** gli utenti possono ottenere risorse computazionali, come *tempo di server e storage di rete autonomamente e automaticamente*.
- **Accesso su rete ampia:** le risorse sono accessibili tramite meccanismi standard che permettono l'uso da parte di varie piattaforme client.
- **Pooling delle risorse:** le risorse del fornitore sono condivise tra più utenti, con assegnazioni dinamiche basate sulla domanda.
- **Elasticità rapida:** le risorse possono essere rapidamente aumentate o ridotte, permettendo una scalabilità flessibile.
- **Servizio misurato:** i sistemi cloud monitorano e ottimizzano automaticamente l'uso delle risorse, fornendo trasparenza per il fornitore e l'utente.

Abbiamo **due tipi di cloud computing**:

- **Cloud pubblico:** accessibile al pubblico generale/ampio gruppo industriale, e di solito è di proprietà di un'*organizzazione che vende servizi cloud*. Offre un ambiente cloud aperto e accessibile con infrastruttura omogenea, politiche comuni e risorse condivise.
- **Cloud privato:** riservato a un'organizzazione, che può gestirlo internamente o tramite una terza parte. Limita l'accesso alle risorse ai membri dell'organizzazione proprietaria, offrendo un'*infrastruttura eterogenea, politiche personalizzate e controllo end-to-end*.

Abbiamo tre modelli di cloud computing:

- **SaaS, Software as a Service:** fornisce applicazioni software complete su Internet. Gli utenti accedono alle applicazioni tramite un browser web, e non devono preoccuparsi dell'installazione, gestione o manutenzione del software
- **PaaS, Platform as a Service:** fornisce una piattaforma che permette agli sviluppatori di costruire, testare e distribuire applicazioni. Include strumenti di sviluppo, Middlesbrough, sistemi di gestione database e altre risorse necessarie
- **IaaS, Infrastructure as a Service:** fornisce risorse di computing virtualizzate su internet. Gli utenti possono affittare server, storage e networking, e hanno il controllo totale sull'infrastruttura, che possono configurare e gestire secondo le loro necessità.

Virtualizzazione

La virtualizzazione è una tecnologia fondamentale che permette di eseguire più sistemi operativi e applicazioni contemporaneamente, indipendentemente dalla piattaforma o dall'hardware sottostante, con vantaggi tra cui *isolamento dei fallimenti, introduzione di nuove capacità, e miglioramento delle prestazioni*.

La virtualizzazione può essere organizzata in vari modi per migliorare l'efficienza e la gestione dei sistemi: possiamo avere un'**organizzazione generale** (struttura che definisce come un programma, interfaccia e sistema interagiscono tra loro), e la **virtualizzazione di un sistema su un altro** (esegue un sistema A sopra un sistema B, creando un ambiente virtuale in cui possono coesistere diversi sistemi operativi).

Abbiamo quattro livelli di interfacce:

- **Istruzioni di macchina non privilegiate:** un'interfaccia diretta all'hardware disponibile per qualsiasi programma
- **Istruzioni privilegiate:** forniscono un'interfaccia all'hardware specificamente per il sistema operativo
- **Chiamate di sistema:** offrono un'interfaccia al sistema operativo per le applicazioni
- **API:** interfaccia dell'OS attraverso chiamate di funzione, spesso utilizzata per nascondere l'interfaccia delle chiamate di sistema

Mentre abbiamo due tipi di virtualizzazione:

- **Macchina virtuale di processo**
 - Avviene tramite interpretazione, o emulazione, e riguarda un singolo processo
 - Fornisce un set di istruzioni di macchina astratto; i programmi vengono compilati in codice macchina, che poi viene interpretato o emulato
- **Monitor di macchina virtuale**
 - Capace di fornire una macchina virtuale a molti programmi contemporaneamente, come se su una singola piattaforma funzionassero più CPU
 - Particolarmente importante per i sistemi distribuiti, poiché *le applicazioni sono isolate, e gli errori sono limitati a una singola VM*. Esempi sono VMWare e Xen.

Microservizi

Questo tipo di architettura suddivide ogni elemento funzionale in un servizio separato, rendendo ogni servizio un'unità indipendente di distribuzione; promuove principi e pratiche dell'ingegneria del software, come *basso accoppiamento* e *alta coesione*, preferendo l'interazione asincrona a quella sincrona, e la coreografia all'orchestrazione.

L'architettura a microservizi è simile alla SOA, ma senza la commercializzazione e le specifiche percepite dei servizi web e middleware simili. Le applicazioni basate su microservizi favoriscono protocolli più semplici e leggeri come REST, e sono progettate secondo il principio di responsabilità singola, dove ogni modulo o classe ha la responsabilità su una singola parte della funzionalità fornita dal software, *completamente encapsulata dalla classe*.

Abbiamo vari principi:

- **Organizzazione intorno alle capacità di business:** un'alta coesione facilita la gestione e la manutenzione, mentre un basso accoppiamento facilita la coordinazione
- **Prodotti e non progetti**
- **Coordinazione, non controllo centralizzato:** governance decentralizzata tramite coreografia ed eventi collaborativi, gestione dei dati decentralizzata
- **Automazione dell'infrastruttura:** l'integrazione di pratiche DevOps per uno sviluppo e operazioni agili
- **Design evolutivo:** il design evolutivo riconosce le difficoltà nel definire correttamente i confini e l'importanza di poterli rifactorizzare facilmente

Invece di costruire un'unica applicazione monolitica, l'idea è di dividere l'applicazione in una serie di servizi più piccoli e interconnessi. Ogni servizio implementa un insieme di funzionalità distinte, e può essere considerato una mini-applicazione. Vari vantaggi sono:

- **Distribuzione ed esecuzione:** a runtime, ogni istanza è spesso una *VM cloud* o un *container docker*. Alcune API REST sono esposte alle app mobili utilizzate da driver e passeggeri. La comunicazione è mediata da un **API Gateway**, responsabile di bilanciamento del carico, *caching*, controllo degli accessi, misurazione delle API e monitoraggio.
- **Indipendenza dei servizi:** ogni servizio ha il proprio schema di database, evitando un modello di dati aziendale unico, garantendo un *basso accoppiamento*, ma con possibili duplicazioni di dati
- **Gestione della complessità:** l'app è suddivisa in servizi gestibili, ciascuno con un'API ben definita. I singoli servizi sono più veloci da sviluppare, più facili da comprendere e mantenere
- **Sviluppo indipendente:** ogni servizio può essere scalato indipendentemente, utilizzando l'hardware che meglio soddisfa i requisiti del servizio.

Applicazioni monolitiche

Sono caratterizzate da una **struttura centrale di logica di business**, implementata attraverso moduli che definiscono servizi, oggetti di dominio ed eventi.

Attorno a questo nucleo centrale si trovano **adattatori**, che interagiscono con il mondo esterno, come *componenti per l'accesso ai database, messaggistica e interfacce web che espongono API o implementano UI*.

Sebbene abbiano un'architettura logica modulare, queste applicazioni vengono pacchettizzate e distribuite come un unico monolite. Il formato di distribuzione dipende dal linguaggio e dal framework utilizzati. Queste applicazioni però hanno vari problemi:

- **Manutenzione ed evoluzione difficili:** a causa della loro complessità, le applicazioni monolitiche sono difficili da mantenere e aggiornare
- **Dependency Hell:** l'aggiunta o l'aggiornamento di librerie può provare a sistemi inconsistenti
- **Reboot necessario per modifiche:** qualsiasi cambiamento in un modulo richiede il riavvio dell'intera applicazione
- **Blocco tecnologico:** gli sviluppatori sono legati alla tecnologia scelta inizialmente
- **Problemi di affidabilità:** un bug in un modulo può compromettere l'intero sistema
- **Distribuzione costosa o subottimale:** la distribuzione di moduli con requisiti diversi su un'unica piattaforma può essere *inefficiente*.

Scalabilità

È una caratteristica cruciale nei sistemi informatici moderni, che permette di **adattare le risorse e le capacità** di un sistema in base alle esigenze variabili degli utenti e dei carichi di lavoro. Non solo consente di gestire un numero crescente di utenti e transazioni, ma anche di migliorare l'affidabilità e la disponibilità dei servizi.

La **scalabilità orizzontale** coinvolge l'aggiunta di più istanze di un'applicazione dietro un bilanciatore di carico, migliorando la capacità di gestione del traffico. La **scalabilità verticale** suddivide un'applicazione in servizi distinti, ciascuno con una funzionalità specifica. La **scalabilità per segmentazione** assegna a ciascun server la responsabilità di un sottoinsieme dei dati, garantendo una distribuzione equilibrata del carico di lavoro.

Contenitori

Le applicazioni funzionali spesso necessitano di componenti aggiuntivi, come strumenti e librerie, per soddisfare le proprietà e le qualità richieste. Per semplificare la gestione e garantire un comportamento coerente, è fondamentale che applicazioni, librerie e strumenti vengano distribuiti insieme.

I **contenitori** rappresentano una soluzione efficace per questo scopo: essi forniscono un modo standard per *impacchettare un'applicazione e tutte le sue dipendenze*, consentendo il trasferimento tra diversi ambienti senza richiedere modifiche. I contenitori funzionano

isolando le differenze tra le applicazioni all'interno del contenitore stesso, permettendo di standardizzare tutto ciò che si trova all'esterno.

Rendono il flusso di lavoro di sviluppo locale e di build più veloce, efficiente e leggero. Consentono l'esecuzione coerente di servizi e applicazioni stand-alone attraverso vari ambienti, l'uso di Docker per creare istanze isolate per test, la costruzione e il test di applicazioni complesse su host locali prima della distribuzione in ambienti di produzione, e la creazione di infrastrutture **Platform-as-a-Service** multiutente. Forniscono inoltre ambienti sandbox leggeri per lo sviluppo, il test e l'insegnamento di tecnologie.

Docker

Docker è una piattaforma aperta per la costruzione di applicazioni distribuite destinata a sviluppatori e amministrazioni di sistemi. È un'applicazione client-server che comprende i seguenti componenti principali:

- **Server:** un programma a lungo termine chiamato processo daemon
- **REST API:** specifica le interfacce che i programmi possono utilizzare per comunicare con il daemon e impartire istruzioni
- **Interfaccia a riga di comando:** Client che utilizza la REST API di Docker per controllare, o interagire con il daemon attraverso script o comandi diretti

Docker image

Un'immagine docker è costituita da filesystem sovrapposti l'uno all'altro.

Alla base c'è un filesystem di avvio, *bootfs*; Docker utilizza la tecnica dell'union mount, che consente di montare diversi filesystem contemporaneamente ma farli apparire come un unico filesystem.

Una delle caratteristiche importanti è il pattern **copy on write**: quando Docker avvia un contenitore, il layer iniziale di lettura-scrittura è vuoto; con le modifiche, queste vengono applicate a questo layer. Se si desidera modificare un file, questo viene copiato dal layer di sola lettura sottostante nel layer di lettura-scrittura. La versione di sola lettura del file esisterà ancora, ma sarà nascosta sotto la copia.

Architettura client-server

Docker adotta questa architettura, che permette una gestione flessibile e scalabile delle applicazioni containerizzate. Questa architettura è essenziale per consentire una vasta gamma di operazioni, dalla creazione alla distribuzione dei contenitori.

Il client e il daemon possono essere eseguiti sullo stesso sistema, oppure un client Docker può connettersi a un daemon Docker remoto. La comunicazione tra il client e il daemon avviene tramite un'API rest, utilizzando socket UNIX o un'interfaccia di rete.

Registry Docker

Un registry Docker è il luogo in cui vengono *memorizzate le immagini Docker*. Docker Hub e Docker Cloud sono registri pubblici, e Docker è configurato per cercare le immagini su Docker Hub per impostazione predefinita. Tuttavia, è possibile anche eseguire un **registry privato**, garantendo maggiore controllo e sicurezza sulle immagini utilizzate. Per le aziende che utilizzano Docker Datacenter, è incluso il Docker Trusted Registry, che offre ulteriori funzionalità di sicurezza e gestione per le immagini Docker.

Microservizi, host OS, hypervisor, infrastrutture as a service