SISTEMI DISTRIBUITI

Laurea triennale in scienze informatiche

Andrea Broccoletti

Università degli studi di Milano Bicocca

A.A. 2022/2023

Indice

Capitolo 1

Definizioni, caratteristiche e problematiche

1.1 Sistema distribuito

Sistemi nei quali i componenti software e hardware comunicano e si coordinano solo scambiandosi dei **messaggi**.
Sono elementi di computazione che appaiono come un singolo sistema centrale coerente, ma ogni nodo è indipendente. Questa caratteristica introduce problemi di **sincronizzazione** e **coordinazione** tra i nodi.

Ha delle caratteristiche principali e generiche:

- non c'è una memoria condivisa
- ogni componente è autonomo
- non c'è un clock globale
- i fallimenti sono indipendenti
- l'esecuzione è concorrente

1.1.1 Trasparenza referenziale

Proprio per far vedere il tutto come un unico sistema ad un utente finale, i fallimenti, i ritardi e le locazioni sono trasparenti.

1.2 Architettura software

Definisce la **struttura del sistema**, le interfacce tra i componenti e i pattern di interazione.
I sistemi distribuiti possono essere organizzati secondo diversi stili architetturali: a strati (*layered*), a livelli (*tier*), basato sugli oggetti, centrate sui dati o basate su eventi.

1.2.1 Architetture stratificate

Sono il modello base, che organizzano il software in **strati**, costruiti uno sopra l'altro, e ogni strato è un set di sottosistemi. I livelli più in alto rappresentano applicazioni più specifiche, mentre quelli più in basso sono più generali.

1.3 Sistemi operativi distribuiti

Sono i sistemi operativi che governano i sistemi distribuiti. Esistono tre modelli principali, con diversi obiettivi:

- **DOS** distribuited operating system, è trasparente alle applicazioni, ovvero il sistema decide dove eseguire l'applicazione in maniera autonoma.
- **NOS** networking operating system, è governato dalle applicazioni, ovvero è l'applicazione che decide dove essere eseguita
- **middleware** layer aggiuntivo che aggiunge trasparenza

1.3.1 Migrazione

Essendoci diverse macchine che collaborano, la collaborazione deve essere esplicitata anche nella migrazione, che può avvenire in tre modi diversi:

- **dati** trasferimento di un file o di porzioni di file
- **della computazione** trasferire la computazione piuttosto che i dati
- **del processo** eseguire un processo o parti di un processo in diversi siti (intesi come locazioni, macchine)

1.3.2 Middleware

Implementa servizi per renderli trasparenti alle applicazioni. Ci sono alcuni servizi base quali:

- naming
- trasparenza di accesso
- persistenza, ovvero uno **storage**
- transazioni distribuite, con relativa **consistenza** in lettura e scrittura
- **sicurezza** con modelli per proteggere dati e servizi

1.4 Modello Client Server

Il modello di interazione in cui il client invia una **request**, il server elabora la richiesta e, nello stesso periodo, attende la risposta, che arriva quando il server fornisce una **reply**.

L'architettura di base prevede quindi che un client acceda ad un server con una richiesta e che il server risponda con un risultato.

L'accesso ai server può avvenire **direttamente**, a **server multipli** o attraverso dei **proxy** server.

1.5 Problemi fondamentali

1.5.1 i 4 Problemi

Generalmente, qualunque sistema distribuito deve interfacciarsi con quattro problemi principali:

- **naming** identificare la controparte della comunicazione
- **access point** come raggiungere un processo o una risorsa remota
- **protocol** come vengono scambiati i messaggi tra i partecipanti alla comunicazione
- **content of a message** come comprendere il contenuto di un messaggio, concordando la sintassi e la semantica.

1.5.2 Distribution transparency

La trasparenza è la pratica di **nascondere i dettagli** agli utenti, che possono ignorare cosa succede e soprattutto non possono influenzare il servizio che stanno usufruendo.

CI sono diversi tipi di trasparenza, a seconda della tipologia e del grado di trasparenza che si vuole raggiungere:

- **naming** nomi simbolici sono usati per identificare risorse che non sono parte dei sistemi distribuiti
- **access transparency** nascondere le differenze nella rappresentazione dei dati e su come delle risorse remote o locali sono accessibili
- **location transparency** nascondere dove una risorsa è localizzata nella rete
- **mobility transparency** detta anche relocation, nascondere lo spostamento di una risorsa in altre lcazioni mentre questa è in uso
- **migration transparency** nascondere lo spostamento di una risorsa, intesa come migrazione
- **replication transparency** nascondere la replicazione di una risorsa
- **concurrency transparency** nascondere la condivisione di una risorsa tra più utenti indipendenti
- **failure transparency** nascondere un fallimento e il recovery di una risorsa
- **persistance transparency** nascondere se una risorsa è volatile o permanente

Gradi di trasparenza

Puntare ad una trasparenza completa è troppo, perchè ci sono alcuni problemi:

- ci sono latenze di comunicazione che non possono essere nascoste
- nascondere completamente i fallimenti è teoricamente e praticamente impossibile
- ha un costo in performance

· esporre la distribuzione a volte è conveniente, ad esempio un utente che non riceve risposta da un server per un lungo periodo di tempo, può invece essere avvisato del fallimento a posto di attendere inutilmente

Interfacce

Dovrebbero essere progettate in accordo ai principi di condivisione, neutrali e supportare l'interoperabilità, la portabilità e l'estendibilità.

1.6 Politica e meccanismo

Un sistema distribuito dovrebbe essere composto da componenti indipendenti logicamente, ovvero ogni componente dovrebbe essere in grado di fornire autonomamente un servizio o eseguire un'operazione.
Ogni componente collabora con altri componenti per fornire servizi più complicati ed eseguire operazioni complesse. Questi obiettivi possono essere raggiunti separando le politche dai meccanismi.

Meccanismo

servizi offerti dai componenti.

Politica

Come i servizi possono esser utilizzati, definendone il comportamento.

Maggiore è il grado di separazione tra politica e meccanismo, più dobbiamo fornire meccanismi più complessi, portando ad una gestione più complessa del sistema. Bisogna quindi trovare un giusto bilanciamento.

1.7 Protocolli

Per poter capire le richieste e formulare delle risposte, i due processi devono **concordare** un protocollo, che definisce il formato, l'ordine di invio e di ricezione dei messaggi tra dispositivi, il tipo dei dati e le azioni da eseguire quando si riceve un messsaggio.

Esempio

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

1.7.1 Elementi minimi

Proprio in relazione ai protocolli, per creare un'applicazione, per quanto basilare, serve come minimo definire un protocollo di comunicazione e condividere il protocollo tra gli attori dell'applicazione.

Capitolo 2

Stream-oriented communication

2.1 Servizi di trasporto internet

2.1.1 Servizio TCP

- orientato alla connessione
- trasporto affidabile
- controllo di fusso e di congestione
- non offre garanzia di banda e ritardi minimi

Nel servizio TCP, ogni messaggio viene scomposto in **segmenti** e numerato per garantire il riordinamento dei dati e controllare duplicazioni e perdite. È il protocollo prediletto per i sistemi distribuiti.

2.1.2 Servizio UDP

È al contrario un trasporto **non affidabile** tra processo mittente e ricevente: non offre connessioni, affidabilità, controllo di flusso, controllo di congestione e garanzie di ritardo e banda. Può essere conveniente per le applicaizoni che tollerano perdite parziali a vantaggio delle prestazioni.
Ogni messaggio viene scomposto in un flusso di byte in segmenti e li invia uno per volta.

2.1.3 Problemi fondamentali

In entrambi i servizi, ci sono problemi fondamentali che vanno affrontati per definire una corretta comunicazione.

Gestione del ciclo di vita di client e server

Bisogna gestire il ciclo di vita delle parti comunicanti, ovvero determinare come vengono attivati e terminati il client e il server.

Identificazione e accesso

Il client deve avere determinate informazioni per poter riconoscere e accedere ad un server. L'**indirizzo** del server può essere:

- direttamente fornito nel codice
- fornito dall'utente
- richiesto con un nameserver DNS
- attraverso protocolli di identificazione

Ripartizione dei compiti

Tra client e server, sebbene abbiano ruoli pressochè ben separati, a volte è necessario, a seconda dell'applicazione e dell'influenza sulle prestazioni, definire cosa è necessario che faccia il client e cosa il server.

2.2 Socket

Ogni processo comunica attraverso **canali**, che gestiscono **flussi di dati in ingresso e in uscita**. Le socket sono particolari canali per la comunicazione tra processi che non condividono memoria, per esempio perchè risiedono su macchine diverse.
Per potersi connettere o inviare dati ad un processo, un altro processo deve conoscere l'**host** e la **porta** a cui connettersi.

La comunicazione via socket avviene attraverso **flussi di byte**, dopo una **connessione esplicita**, tramite normali system call **read** e **write**. Sono chiamate sospensive che bloccano il processo finchè il sistema operativo non ha effettuato la lettura o la scrittura. Entrambe le call utilizzano **buffer** per

garantire flessibilità. Oltre a queste due principali, ci sono altre calls che sono definite come standard:

- `Socket` crea un nuovo endpoint di comunicazione
- `Bind` blinda un indirizzo locale a un socket
- `Listen` annuncia che il socket è pronto alla connessione
- `Accept` blocca fintantochè una connessione non viene accettata
- `Connect` tentativo di stabilire una connessione
- `Close` rilascia la connessione

2.2.1 Processi e socket

Il server crea una socket collegata alla well-known port, che identifica il servizio fornito, dedicata a ricevere richieste di connessione. Con la primitva `accept()`, il server crea una nuova socket, cioè un nuovo canale, dedicato alla comunicazione con il client. È quindi un altro processo che elabora le richieste del client che si è connesso, mentre il processo principale continua ad accettare nuove richieste.

2.2.2 Read e Write

Le socket sono come detto stream, ovvero flussi di bytes, quindi non c'è il concetto di **messaggio**, è un flusso continuo e senza fine. La lettura e la scrittura devono avvenire per **numero abritrario di byte**.
Quindi, si devono prevedere **cilci di lettura** che termineranno in base alla dimensione dei messaggi come stabilito dal formato del protocollo applicativo in uso.

2.2.3 Le socket in Java

Java definisce alcune classi che costituiscono un'interfaccia ad oggetti alle system call precedentemente illustrate. Le principali sono la `java.net.Socket` e la `java.net.ServerSocket`.
Queste classi accorpano funzionalità e mascherano alcuni dettagli con il vantaggio di semplificarne l'uso. come per ogni **framework** è necessario consocere il modello e il funzionamento per poterlo utilizzare in modo efficace.
Queste funzionalità sono esposte di seguito.

Costruttori

- `public Socket(String host, int port)` crea un socket connesso alla specifica porta, in un determinato host
- `public ServerSocket(int port)` crea un server socket, connesso alla porta specificata. La coda massima consentita di default è di 50 richieste di connessione

Gestire le connessioni

- `public void bind(SocketAddress bindpoint)` blinda un socket ad un indirizzo locale
- `public void connect(SocketAddress endpoint, int timeout)` connette il socket al server, con uno specifico valore di timeout
- `public void close()` chiude il socket
- `public Socket accept()throws IOException` rimane in ascolto e accetta una connessione. Restiuisce il nuovo socket. Il metodo è bloccante fintantochè una connessione non viene effettuata

Gestire i flussi

- `public InputStream getInputStream()throws IOExceptionpublic InputStream getInputStream()throws IOException` restituisce l'input stream legato al socket.
- `public OutputStream getOutputStream()throws IOException` Restituscei l'output stream legato al socket per scrivere bytes.

2.2.4 Progettazione di un'applicazione

Per quanto riguarda il **lato client**, l'architettura è concettualmente più semplice, è spesso un'applicazione convenzionale che usa una socket anzichè un altro canale di I/O. Non ci sono particolari problemi di sicurezza, ha effetti solo sul lato client.
Invece, il **lato server**, prevedere un'archiettura generale in cui viene creata una socket con una porta nota per accettare le richieste di connessione. Inoltre, si prevede un ciclo infinito in cui alternare:

1. attesa/accettazione di una richiesta di connessione ad un client

2. ciclo lettura-esecuzione, invio risposta fino al termine della conversazione, stabilita spesso dal client

3. chiusura della connessione

2.3 Architetture dei server

Introducendo le architetture dei server, è doveroso distinguerli in quattro tipologie:

- **iterativi** soddisfano una richiesta alla volta
- **concorrenti processo singolo** simulano la presenza di un server dedicato
- **concorrenti multi-processo** creano server dedicati
- **concorrenti multi-thread** creano thread dedicati

Di seguito saranno trattati in modo più approfondito.

2.3.1 I/O non bloccante

Prima però, è doveroso fare una digressione sulle operazioni di lettura e scrittura, che comportano l'uso di system call **bloccanti**, ovvero si deve attendere la conclusione dell'operazione richiestta prima di restituire il controllo al chiamante.
Per leggere in modo **non bloccante**, serve saper prima di fare un operazione di I/O se il canale è **pronto**, cioè se si effettua un'operazione il controllo viene restituito immediatamente. LA system call `select()` ha proprio questo compito. In Java, questo tipo di I/O viene effettuato grazie ai **channel**, che non fa altro che mettere il chiamante in sleep e di rispondere in modo negativo se non è possibile effettuare un'operazione di I/O. A lato di questi, è presente il `Selector` che permette di gestire dei canali non bloccanti, in particolare può gestire gli eventi di connessione, accettazione, scrittura e lettura.

2.3.2 Server iterativo

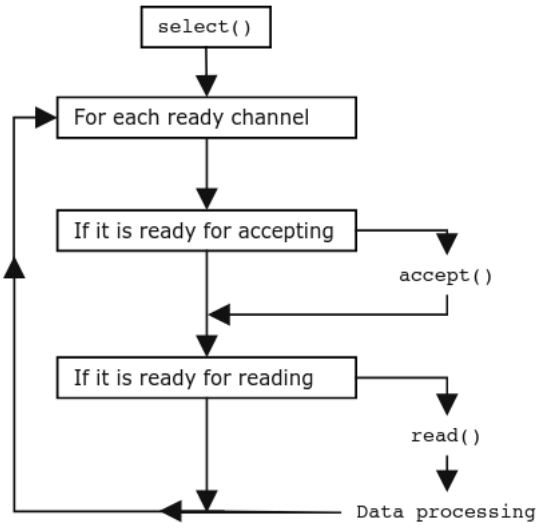
Al momento di una richiesta di connessione, il server crea una socket temporanea per stabilire una connessione diretta con il client. Le eventuali ulteriori richieste per il server verranno **accodate** alla porta nota per essere

successivamente soddisfatte.
È semplice da progettare, ma viene servito un solo client alla volta, e questo può bloccare l'evoluzione di altri client.

2.3.3 Server concorrente

Un server concorrente può gestire più connessioni client, attraverso la possibilità di **simulare più socket** utilizzando un solo processo, oppure creando **processi slave**.
Generalmente, uno schema per questa tipologia di server è la *figura 2.1*.

Figura 2.1: Esempio di server concorrente



2.3.4 Server multiprocesso

Non è altro che un server concorrente che **crea nuovi processi** slave attraverso la primitiva `fork()`, che crea un processo clone del padre che eredita i canali di comunicazione ed esegue lo stesso codice. Il codice deve prevedere che il padre chiuda la socket per la conversazione con il client e che il figlio chiuda a sua volta la socket per l'accettazione di nuove connessioni. La struttura del server è la stessa di quella iterativa in quanto ogni server gestisce un solo client.

2.3.5 Confronto tra modelli

Per confrontare i modelli, ovvero chiarire quando è necessario un server monoprocesso e quando multiprocesso, è più efficace confrontare le loro caratteristiche principali.

Monoprocesso

Sono iterativi e concorrenti, quindi:

- gli utenti condividono lo stesso spazio di lavoro
- adatto ad applicazioni **cooperative** che prevedono la **modifica dello stato**, ovvero può avvenire lettura e scrittura

Multiprocesso

In questo caso invece:

- ogni utente ha uno spazio di lavoro autonomo
- adatto ad applicazioni **coopeartive** che **non modificano** lo stato del server, ovvero si effettua solo lettura
- adatto anche ad applicazioni **autonome** che modificano uno spazio di lavoro proprio, in cui si effettuano lettura e scrittura

Capitolo 3

Concorrenza e programmazione multithreading

3.1 Introduzione alla concorrenza

La **concorrenza** è la contemporaneità di esecuzione di parti diverse di uno stesso programma, ovvero la capacità di far progredire più di una attività simultaneamente. Diverso è il **parallelismo**, che è invece la capacità di eseguire più di una attività simultaneamente.

3.1.1 Legge di Amdhal

Fornisce il guadango in termini di performance derivante dall'aggiunta di core ad un'applicazione che ha componenti sia sequenziali sia parallele.

3.1.2 Programmazione concorrente

Si indica così la pratica di implementare dei programmi che contengano **più flussi di esecuzione**. Viene utilizzato per sfruttare gli attuali processori multicore e per evitare di bloccare l'intera esecuzione di un'applicazione a causa dell'attesa del completamento di un'azione di I/O.

3.1.3 Processi

Un sistema operativo esegue un certo numero di programmi sullo stesso sitema di elaborazione, di solito di molto maggiore rispetto al numero di CPU del sistema. A tal scopo il sistema operativo realizza e mette a disposizione un'**astrazione detta process**, che è un'entità attiva astratta definita

dal sistema operativo allo scopo di eseguire un programma. Si suppone un'esecuzione sequenziale.

Operazioni sui processi

I SO di solito forniscono delle chiamate di sistema con le quali un processo può **creare**, **terminare**, **manipolare** altri processi. Dal momento che solo un processo può creare un altro processo, all'avvio il sistema operativo crea dei processi primordiali dai quali tutti i processi utente e di sistema vengono progressivamente creati.

Struttura di un processo

Un processo è composto da diverse parti:

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

Immagine di un programma

L'immagine è formata da:

- codice del programma
- data section, che ha dimensioni costanti per tutta la vita del processo
- stack e heap

Stato di un processo

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

- **new** il processo è creato, ma non ancora ammesso all'esecuzione
- **ready** il processo può essere eseguito ed è in attesa che gli sia assegnata una CPU

- **waiting** il processo non può essere eseguito perchè è in attesa che si verifichi qualche evento
- **terminated** il processo ha terminato l'esecuzione

Commutazione di contesto

Avviene quando la CPU deve passare dall'esecuzione di un processo a quella di un altro processo, viene effettuata dal **dispatcher**. Più è complesso il sistema operativo, più lo è il contesto.

3.1.4 Multiprogrammazione

Impedire che un programma che non è in condizione di proseguire l'esecuzione mantenga la CPU occupata. Il SO mantiene in memoria i processi da eseguire, li carica e gli assegna memoria ed una serie di altre informazioni. Quando una CPU non è impegnata ad eseguire un processo, il SO seleziona un processo non in esecuzione e gli assegna la CPU.

3.1.5 Multitasking

Far si che un programma interattivo reagisca agli input utente in un tempo accettabile, è quindi un'**estensione della multiprogrammazione** in cui la CPU viene sottratta periodicamente al programma in esecuzione ed assegnata ad un altro programma, in questo modo tutti i programmi progrediscono in maniera continuativa nella propria esecuzione.

3.1.6 Multithreading

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

Thread a livello utente

Sono disponibili nello spazio utente dei processi, sono quindi quelli offerti dalle librerie thread ai processi.

Thread a livello kernel

Sono implementati nativamente dal kernel, sono utilzizati per struttura il kernel stesso in maniera concorrente e vengono utilizzati dalle librerie di thread per implementare i thread a livello utente di un certo processo. A tal scopo possono essere adottate diverse strategie:

- **molti a uno** i thread a livello utente di un certo processo sono implementati su un solo thread a livello del kernel, è bloccante ma utilizzabile su ogni SO
- **uno a uno** ogni thread a livello utente è implementato su un singolo, distinto thread a livello kernel, aumenta la concorrenza e il parallelismo, ma anche l'overhead
- **molti a molti** i thread a livello tente di un certo processo sono implementati su un insieme di thread a livello del kernel possibilmente inferiore di numero, e l'associzione thread utente/thread kernel è dinamica, stabilita da uno scheduler interno alla libreria di thread. È complessa da implementare.

Lightweight processo

Un LWP è l'interfaccia offerta dal kernel alle librerie dei thread per usare i thread del kernel. Ogni LWP è un **oggetto astratto** associato statisticamente dal kernel ad esattamente un thread del kernel e può essere usato dalla libreria dei thread per maggiornare le (minime) informazioni relative al contesto del processo utente utili a prendere migliori decisioni di scheduling.

In linux

Linux non distingue tra processi e thread e indica entrambi con il termine **task**.

3.1.7 Comunicazione inter-processo

I processi possono essere indipendenti o cooperare, e coopera se il suo comportamento influenza o è influenzato da il comportamento di uno o più altri processi. Per permettere ai processi di cooperare il sistema operativo deve mettere a disposizione primitive di comunicazione inter-processo.

Memoria condivisa

Viene stabilita una zona di memoria condivisa tra i processi che intendono comunicare e la comunicazione è controllare dai processi che comunicano, che devono anche essere capaci di **sincronizzarsi**.

Message passing

Permettono ai processi sia di comunicare che di sincronizzarsi, infatti i processi comunicano tra di loro senza condivisione di memoria, ma bensì attraverso la mediazione del sistema operativo o di un altro processo. Devono quindi prima stabilire un **link di comunicazione** tra di loro, e poi scambiarsi messaggi per comunicare.

3.2 Multithreading in Java

3.2.1 Java threads

Sono **astrazioni** offerte dalla JVM e gestiti dalla stessa. Tipicamente sono implementati sfruttando il modello di threading offerto dal sistema operativo, offrendo quindi una libreria per la definizione e l'esecuzione dei thread e per l'interazione con il sistema operativo.
Le modalitià principali di creazione dei thread sono con la classe `java.lang.Thread` e l'interfaccia `java.lang.Runnable`.
In Java ogni programma in esecuzione è un thread, anche il metodo `main()`.

3.2.2 Classe principale

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

1. estendere la classe `java.lang.Thread` che contiene un metodo `run()` vuoto
2. riscrivere, ovvero effettuare un **override**, del metodo `run()` nella sottoclasse. Il codice eseguito dal thread è incluso nel metodo e nei metodi invocati direttamente o indirettamente dallo stesso. Inoltre, il codice verrà eseguito in parallelo a quello di altri thread
3. creare un'istanza della sottoclasse
4. richiamare il metodo `start()` che crea l'istanza e fa anche partire il thread

È importante notare che non dobbiamo richiamare esplicitamente il metodo `run`, poichè la sua invocazione avviene in maniera automatica. I due thread, creante e creato, saranno eseguiti in modo concorrente ed indipendente. È importante anche notare che un'applicazione Java **ha almeno un** thread.

L'ordine con cui ogni thread eseguirà le proprie istruzioni è noto, ma l'ordine globale in cui le istruzioni saranno eseguite è effettivamente indeterminato, introducendo quindi **nondeterminismo**.

Esempio

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

3.2.3 Runnable interface

Siccome Java non consente l'ereditarietà multipla, un modo alternativo di realizzare un thread è implementare solamente il metodo `run`, ovvero implementare l'interfaccia `Runnable`, avendo la possibilità di estendere un'altra classe base, introducendo **flessibilità**.

Per creare thread usando l'interfaccia `java.lang.Runnable` bisogna:

1. definire una classe, implementazione di `Runnable`, dotata di un metodo `run()` significativo
2. creare un'istanza di questa classe
3. istanziare un nuovo thread, passando al costruttore l'istanza della classe

Esempio

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

3.2.4 Terminazione di thread

Quando si avvia un thread attraverso il suo **entry point**, il thread entra in stato di **alive**, fintantochè il suo metodo `run` non termina. Quando `run` ritorna, il thread è considerato **terminated**.

Una volta terminato, il thread **non può essere rieseguito**. Quindi, non si può far partire lo stesso thread (ovvero la stessa istanza) più volte.
Si noti che solo la chiamata `start()` crea un nuovo thread. Si può invocare run() direttamente, ma in questo modo il metodo verrà eseguito normalmente, sullo stack del thread corrente, senza che un nuovo thread venga creato: non lo fate!

Is Alive

Esiste il predicato `isAlive()` che può essere usato per valutare se il thread sia stato fatto partire e al contempo se non sia stato terminato.

IllegalThreadStateException

Se si tenta di riavviare o avviare un thead più volte, la chiamata stessa genera l'eccezione `IllegalThreadStateException`.

3.2.5 Busy loop

Per rallentare l'esecuzione dei thread, si può creare un metodo privato che cicli a vuoto allo scopo di perdere del tempo. Generalmente non è però una buona soluzione, perchè si sprecano cicli di processore e, se ci sono molti

thread, è possibile che quando si giunge al momento in cui il thread deve uscire dal ciclo, il thread in questione non abbia accesso alla CPU, con la conseguenza che esce **in ritardo** rispetto al momento desiderato.

Esempio

```
private void BusyLoop() {
    long start = System.currentTimeMills();
    long stop = stratTime + 60000;

    while(System.currentTimeMills() < stop) {
        continue;
    }
}
```

3.2.6 Cancellazione dei thread

Con cancellazione si intende la terminazione prematura di un thread, due possibili modelli per la cancellazione sono:

- **immediata** il thread è terminato immediatamente
- **differita** il thread controlla periodicamente se deve terminare, in modo da effettuare una **terminazione ordinata**, come in java.

Cancellazione in Java

In Java, la cancellazione avviene attraverso il metodo `interrupt()` che setta un **flag di interruzione** nel thread di destinazione e ritorna. Il thread che riceve l'interruzione non viene effettivamente interrotto, ma solo quando viene controllato il flag il thread sa di dover terminare. In particolare, il flag viene controllato quando si invocano metodi quali `sleep`, `join`, etc. Se un thread è già in stato di pausa e riceve una `interrupt()`, la JVM lancia l'eccezione `InterruptedException`.

Esempio

```
public class ThreadSleep extends Thread {
    public ThreadSleep(String s) { super(s); }

    public void run() {
        for (int i=0; i<5; ++i) {
        System.out.println(getName() + ": in esecuzione.");
        try {
            Thread.sleep(200);
        }
        catch (InterruptedException e) {
            System.err.println(getName() + ": interrotto");
            break;
        }
    }
    System.err.println(getName() + ": finito");
    }
}
```

Problematica di interruzione

Il metodo `interrupt` non funziona se il thread interrotto non esegue mai metodi di attesa. I thread devono quindi controllare periodicamente il loro stato di interruzione, attraverso `isInterrupted()` che controlla il flag di interruzione senza resettarlo, o con `Thread.interrupted()` che invece controlla il flag di interruzione e se settato lo resetta.

Esempio

```
public class MyThread extends Thread {
    public void run() {
        while (condizione) {
            if(Thread.interrupted()) {
                System.err.println(getName() + ": ended");
                break;
            }
        }
    }
}
```

3.2.7 Threading implicito

Meccanismo che **delega** la creazione e gestione dei threads ai compilatori e alle librerie. L'obiettivo è di permettere agli sviluppatori di ragionare in termini di **task** da compiere, che vengono poi mappati sui thread in maniera automatica.

Thread pools

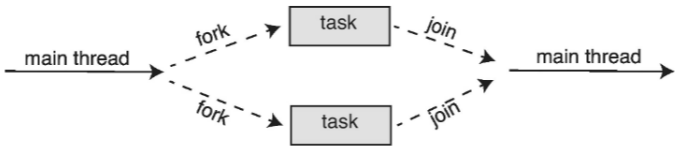
È uno degli approcci impliciti di threading, in cui viene creato un certo numero di thread, organizzati **in gruppo**, che attendono di rispondere ad una richiesta di lavoro. È l'approccio utilizzato per la gestione del ciclo di vita delle servlet. Ha diversi vantaggi:

- più rapido che creare il thread all'arrivo della richiesta
- separa il task da svolgere dalla meccanica della sua creazione, permettendo diverse **strategie di esecuzione**

Fork-join

Un altro metodo di threading implicito, che suddivide un main thread in diverse task.

Figura 3.1: Classico schema fork-join



Esempio

```
Task(problem)
    if problem is small enough
        solve the problem
    else
        subtask = fork(subset of the problem)
        subtask2 = fork(subset of the problem)

        return combined results
```

3.3 Sincronizzazione

3.3.1 Interazione tra agenti concorrenti

- **cooperazione** interazioni prevedibili e desiderate, la loro presenza è necessaria per la logica del programma. Avviene tramite scambio di informazioni, anche semplici come segnali
- **competizione** gli agenti competono per accedere ad una risorsa condivisa, con la necessità di politiche di accesso alla risorsa
- **interferenze** interazioni non prevedibili e non desiderate, principalmente errori di programmazione, spesso dipendenti dalle tempistiche e non facilmente riproducibili

3.3.2 Meccanismi di sincronizzazione

Sono i meccanismi che permettono di **controllare l'ordine relativo delle varie attività** dei processi/thread.
Questi meccanismi vengono messi a disposizione dal sistema operativo nel caso di processi. In Java ritroviamo le stesse solizioni implementate a livello di JVM per i thread.

3.3.3 Sincronizzazione su eventi

Un esempio di metodo per la sincronizzazione temporale di 2 thread è il metodo `join`. Quando si invoca, il thread chiamante entra in uno stato di **attesa**. Rimane in tale stato finchè il thread chiamato non termina. Può anche ritornare se il thread chiamato viene interrotto, generando una `InterruptException`. Se invece il thread chiamato è già terminato o non avviato, la chiamata ritorna immediatamente.

3.3.4 Sincronizzazione su condizione

Meccanismo di sincronizzazione detto di **barriera**, prevede che i thread vengano messi progressivamente in attesa che una condizione globale non venga soddisfatta.
Una possibile implementazione di questo tipo è un **contatore condiviso** conta il numero di processi che devono terminare, il contatore in questione decrementa ogni volta che un thread termina il suo task.

3.3.5 Race condition

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

Il problema è dovuto al fatto che le azioni **non sono atomiche**, cioè sono interrompibili dallo scheduler, che può **inframmezzare** l'esecuzione di thread differenti, portando la risorsa condivisa in uno stato inconsistente.

Un aggiornamento incorretto di una risorsa dovuto a una combinazione aleatoria di letture e scritture viene detto **interferenza**.

Una semplice soluzione al problema consisnte nel fornire accesso **mutuamente esclusivo** alla risorsa, logicamente non interrompibile.

3.3.6 Sincronizzazione basata su Sezione Critica

L'obiettivo è realizzare un protocollo di cooperazione tra i processi che faccia si che, quando un thread è nella sua **sezione critica**, gli altri non lo siano, secondo il principio di **mutua esclusione**.

La **sezione critica** è un blocco di codice che può essere esguito da un solo thread alla volta.

Per realizzare questo tipo di sincronizzazione, c'è bisogno di un'operazione atomica `test_and_set(B)` non interrompibile che, supponendo B variabile booleana in memoria, ritorna il valore originario della cella di memoria puntata e imposta il valore della stessa a `true`.

Lock

Il lock è una **variabile logica** booleana manipolabile atomicamente. Quando un thread acquisisce un lock, gli altri thread che lo richiedono si bloccano fintantochè il thread che lo detiene non lo rilascia. Per questo, ogni lock deve avere e gestire una **code di thread in attesa**.

Semaforo

Il semaforo è una variabile intera che può essere manipolata solo attraverso operazioni atomiche `acquire()`, che aspetta che la variabile sia maggiore di

zero, e quindi la decrementa e `release`, che invece incrementa incondizionalmente la variabile. Anche per ogni semaforo è necessaria una **coda di attesa**.

3.3.7 Synchronized

Serve per implementare la mutua esclusione in Java. In particolare, Java associa un **lock intrinseco/implicito**, ad ogni oggetto che abbia **almeno** un metodo synchronized.
Il fatto che un un thread sia in esecuzione all'interno di un metodo (o blocco) synchronized fa sì che altri thread che richiedano l'esecuzione dello stesso o un altro metodo (o blocco) synchronized vengano messi in attesa che T1 completi l'esecuzione del metodo.

Esempio

```
[...]
public synchronized void deposito(
    float cifra){
        saldo += cifra;
    }
public synchronized void prelievo(
    float cifra){
        if(saldo > cifra){
            saldo -= cifra;
            return cifra;
        }
    float prelevati = saldo;
    saldo = 0.0f;
}
[...]
```

Monitor

È una primitiva a più alto livello rispetto a lock e semafori, di tipo **astratto**, le cui variabili interne sono accessibili solo da un insieme di procedure esposte dal monitor stesso. Solo un processo/thread alla volta può essere attivo nel monitor.

All'interno, vi possono essere delle **variabili di tipo condizione**, su cui è possibile effettuare operazioni di:

- `x.wait()` mette in stato di attesa il processo corrente, e lo foza a lasciare il monitor
- `x.signal()` rende ready un processo che aveva invocato il metodo precedente, se esiste

In Java, ogni Object è un monitor, e quindi i metodi `wait`, `notify`, `notifyAll` sono presenti in ogni classe, però la gestione dei meccanismi di accesso condizionato che utilizzano questi metodi sono completamente a carico del programmatore.

3.3.8 Problema Produttore-Consumatore

Supponiamo di avere una risorsa condivisa tra un produttore e un consumatore, caratterizzata da alcuni stati interni significativi. Per quanto la memoria possa essere ingente, il buffer ha **dimensione finita** e, inoltre, se è **vuoto** non è possibile prelevare nulla. Il problema consiste nell'assicurare che:

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

Per risolvere il problema, si deve trovare un meccanismo di controllo sulla risorsa che garantisca che certe condizioni siano verificate, in particolare Java mette a disposizione un insieme di metodi per risovlere questo problema in maniera elegante e lineare.
Dal punto di vista implementativo:

1. **se il buffer è pieno, non inserire nulla e aspetta**, il produttore deve quindi sospendere la propria esecuzione fintantochè il buffer è pieno
2. **se il buffer è vuoto, non provare a cancellare niente e aspettare**, ovvero il consumatore si sospende fintantochè il buffer è vuoto.

BlockingQueue

Nei fatti, si deve costruire una coda bloccante, che in Java è già definita nell'interfaccia generica `BlockingQueue<T>`, che presenta due metodi bloccanti `put()` e `take()`.

3.4 Variabili atomiche

Tip

Un'operazione è atomica quando tutte le sub-operazioni che la compongono verranno eseguite senza possibilità di interruzione da parte di un altro thread

3.4.1 Problema della visibilità

Per le applicazionio multithread, è necessario garantire due condizioni per un comportamento coerente:

- **mutua esclusione** solo un thead esegue una sezione critica alla volta
- **visibilità** le modifiche apportate da un thread ai dati condivisi sono visibili agli altri thread per mantenere la coerenza dei dati.

Problemi di visibilità si creano quando due thread sono in esecuzione su core diversi, perché le variabili condivise vengono tenute in cache per ragioni di performance. Si può ovviare al problema della visibilità utilizzando variabili **volatile**.

3.4.2 Problemi del locking

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

- se un thread non riesce ad acquisire un lock viene sospeso
- context switch, risvegliare un thread presenta un costo
- un thread in attesa non può eseguire nessuna operazione
- un thread a bassa priorità può bloccare thread che ne hanno una più alta (priority inversion), infatti quando un thread acquisisce un lock nessun altro thread che ha bisogno di quel lock può proseguire

3.4.3 Optimistic retrying

Approccio alternatico al locking, che si basa sul principio che sia più efficiente riprovare che chiedere un permesso. In particolare, non è prevista nessuna sincronizzazione in lettura, mentre in scrittura:

1. lettura della variabile, creando una copia locale
2. aggiornamento della copia
3. scrittura della variabile **se non c'è collisione**, ovvero se il valore della variabile non è cambiato, altrimenti si ritenta l'operazione

Questo approccio è implementato dagli strumenti `CompareAndSwap` e dal `compareAndSet`. Permane comunque, seppure in modo molto remoto, la possibilità di **starvation**, eliminando però i problemi legati al locking. Inoltre, nel peggiore dei csi, **l'overhead non è mai superiore a quello dei lock**.

3.4.4 Atomiche in Java

Java dispone di 12 classi contenitori per implementare le variabili atomiche, su cui si possono effettuare operazioni di:

- `int get()` restituise il valore corrente della variabile
- `void set(int newValue)` aggiorna il valore corrente della variabile
- `int getAndSet(int newValue` aggiorna atomicamente il valore corrente della variabile e restituisce il valore contenuto precedentemente
- `boolean compareAndSet(int expected, int update)` aggiorna atomicamente il valore della variabile update se il valore corrente della variabile è uguale a expected
- `int getAndIncrement()` incrementa atomicamente il valore corrente della variabile e restituisce il valore contenuto precedentemente

Oggetti complessi

In Java, si può ricorrere a delle classi di aiuto, in cui salvare oggetti complessi, in particolare la `AtomicReference<T>` può essere usato per verificare l'aggiornamento dell'oggetto in questione usando la CAS.

3.5 Liveness

Nella programmazione concorrente, per liveness si intende il fatto che un programma sia sempre in grado di progredire correttamente evitando problemi. Proprietà che **non è garantita** dal fatto che le componenti concorrenti siano attice e accedano in maniera mutualmente esclusiva alle cosiddette sezioni critiche di un oggetto.
Questa proprietà, è desiderabile, ma per essere definita più precisamente richiede di specificare quali siano i problemi che si vogliono evitare.

Possiamo arrivare a due definizioni di liveness, una debole e una più stringente:

- 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.
- Un sistema, in circostanze analoghe al precedente, è libero da starvation se garantisce che tutti i thread riescano ad accedere alla sezione critica e progredire nella loro esecuzione

3.5.1 Deadlock

Quando ad un processo viene garantito l'accesso esclusivo (ad esempio tramite una mutua esclusione) ad una risorsa, possono crearsi **situazioni di stallo**.
In programmazione concorrente, la situazione di deadlock si verifica quando **ogni membro** di un gruppo di agenti (nel nostro caso i thread) è in **attesa** che qualche altro membro rilasci un lock su di una risorsa.

In pratica si tratta di un'attesa circolare destinata a non terminare mai. Infatti, essendo tutti i thread in attesa, nessuno potrà mai generare l'evento di sblocco, quindi l'attesa si protrae all'infinito.

Condizioni necessarie

Ci sono quattro condizioni necessarie affinché un deadlock si verifichi. Queste sono generalmente espresse in termini di risorse assegnate a un thread. Se esiste nel sistema **una sola istanza per ogni tipo di risorsa** allora tali condizioni sono anche sufficienti per un deadlock.

- **mutua esclusione** solo un'attività concorrente (thread o processo) per volta può utilizzare una risorsa (cioè, la risorsa non è condivisibile simultaneamente).
- **Hold and wait** attività 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 aspettando altre risorse da acquisire.
- **No preemption sulle risorse condivise** una risorsa può essere rilasciata solo volontariamente (non tolta con la forza dal SO/Scheduler/JVM) da un'attività concorrente
- **Attesa circolare** deve esistere una possibile catena circolare di attività concorrenti e di richieste di accesso a risorse concorrenti tale che ogni attività mantiene bloccate delle risorse che contemporaneamente vengono richieste dai attività successive.

3.5.2 Evitare il deadlock

Esistono alcuni possibili approcci per affrontare le situazioni di Deadlock.

Deadlock prevention

il Deadlock può essere evitato se si fa in modo che almeno una delle quattro condizioni richieste per deadlock (Mutual exclusion, Hold and wait, No preemption e Circular wait) non si verifichi mai.
Per esempio, evitare la hold and wait facendo in modo che mai si possa detenere una risorsa mentre si è in attesa di un altra risorsa. Oppure evitare l'attesa circolare ordinando le richieste di accesso alle risorse, e si richiede il lock seguendo l'ordine. Se tutti i thread si attengono a questa disciplina, non si possono creare deadlock circolari.

Deadlock removal

non si previene il deadlock, ma lo si risolve quando ci si accorge che è avvenuto.

3.5.3 Starvation

In programmazione concorrente, la situazione di starvation si verifica quando un agente **non riesce ad accedere a una risorsa** che le viene perpetuamente negata. Questo può essere dovuto alle 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. Anche una gestione scorretta della mutua esclusione o una definizione inadeguata dell'**algoritmo delle attività** date le circostanze (esempio della rotonda con traffico sbilanciato) potrebbe causare starvation.

3.5.4 Livelock

Il concetto di livelock è simile a quello di deadlock ma ancora più complicato da caratterizzare formalmente e quindi da poter gestire in modo automatico. Formalmente, in una certa situazione i membri di un gruppo di agenti possono non essere bloccati, ma ciononostante **non progredire effettivamente**. Diversi protocolli distribuiti hanno necessità di effettuare operazioni di sincronizzazione iniziale, anche detta **handshake**, quindi questo genere di problematica è meno caricaturale di quanto possa sembrare

3.5.5 I filosofi a cena

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 (quella alla sua sinistra e quella alla sua destra). Le forchette non sono condivisibili simultaneamente, e dopo aver mangiato un filosofo ripone sul tavolo le forchette usate. Si suppone che il piatto venga riempito da qualche entità esterna non appena diventa vuoto, chiaramente si tratta di un problema giocattolo per illustrare un problema serio.

Il problema consiste nello sviluppo di un algoritmo che impedisca situazioni di deadlock o starvation.

Se tutti i filosofi prendessero la forchetta di sinistra, si raggiungerebbe un deadlock, e la stessa cosa vale per qualsiasi permutazione di questo insieme di azioni.

Soluzione di Dijkstra

Questa soluzione al problema, spezza la simmetria che causa l'**attesa circolare**. In pratica, viene considerato l'ordine tra le forchette e si forza a prendere prima quella con identificativo minore.

Soluzione basata sull'attesa circolare

Se un filosofo ha acquisito una forchetta e non è riuscito a procurarsi anche la seconda, può ipotizzare che si stia creando la condizione per un deadlock, quindi rilascia la forchetta in suo possesso e attende un po' di tempo prima di riprovare a impossessarsi delle forchette. Ma se tutti i filosofi si comportano nello stesso modo, tutti rilasciano la forchetta destra contemporaneamente e poi tutti riprendono contemporaneamente la forchetta sinistra può crearsi una situazione di livelock. Perchè il procedimentro quindi funzioni, è necessaria un' **attesa casuale**.

Soluzione basata su rimozione di hold and wait

Soluzione basata su rimozione di hold and wait § Altra possibilità consiste nell'evitare la condizione "hold and wait": basta che ogni filosofo prenda (con una operazione atomica) entrambi le forchette se disponibili, e aspetti se invece non sono disponibili. Ci vuole un **mediatore** che osservi lo stato delle forchette e agisca di conseguenza.

3.5.6 Problema lettori/scrittori

Abbiamo una risorsa condivisa che può essere letta da un certo numero thread in contemporanea ma il cui accesso in scrittura deve invece essere esclusivo (per questioni di **consistenza del dato**).
La semplice soluzione prevede che:

- la lettura non sia interamente sincronizzata, il controllo che non ci siano scrittori attivi e che il numero di lettori sia inferiore al massimo permesso deve essere sincronizzata.
- la scrittura deve essere invece completamente sincronizzata ma non deve avvenire se anche solo un lettore sta effettuando il suo lavoro.

Nella risorsa condivisa è necessario mantenere due contatori, rispettivamente per il numero di lettori e scrittori attuali.

La soluzione proposta non è **fair** perché in caso ci sia uno sbilanciamento tra il numero di thread reader e quello dei thread writer un thread di tipo writer messo in attesa potrebbe accedere al database dopo thread reader creati dopo. Esiste anche il problema della starvation; se le letture arrivano in continuazione allora le scritture potrebbero rimanere sempre in attesa.

Capitolo 4

Message oriented communication

4.1 Browser

Il browser è l'applicazione per il Web sul lato del client. Un web browser, detto **User-Agent**, è un programma che consente la navigazione nel Web da parte di un utente. La funzione primaria di un browser è quella di **interpretare il codice con cui sono espresse le informazioni** e **visualizzarlo** in formato di ipertesto.

4.1.1 Web Page

È concettualmente un file, cioè una **sequenza di dati** residente in un computer, che è identificato da una **URL**, ovvero un indirizzo univoco di quella risorsa.
Storicamente, una pagina web è un file HTML che definisce la struttura e i contenuti della pagina. Le web page sono servite da dei **web server**.

4.2 Ipertesto

Il modello fondativo del web è l'ipertesto, che è un insieme di testi o pagine che possono essere letti in maniera non sequenziale secondo percorsi definiti detti **hyperlink**, o più semplicemente **link**. Gli ipertesti compongono una reta raggiata o variamente incrociata di informazioni organizzate secondo criteri paritetici o gerarchici.

4.2.1 URL

Identifica univocamente un oggetto nella rete e specifica il protocollo da usare per ricevere e inviare dati. Ha cinque componenti principali:

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

4.3 Protocollo HTTP

È un protocollo **stateless**, ovvero che non mantiene informazione sulle richieste precedenti del client, che serve per il trasferimento di pagine di ipertesto. Concretamente, client e server si scambiano messaggi HTTP.

4.3.1 Metodi

I principali metodi utilizzati sono:

GET

Restituisce una rappresentazione di una risorsa, includendo eventuali parametri in coda alla URL della risorsa. La sua esecuzione non ha effetti sul server, per questo è considerato **safe**.
Include anche eventuali parametri in coda alla URL della risorsa.

POST

Comunica dei dati da elaborare lato server o crea una nuova risorsa subordinata all'URL indicata.
La POST prevede che i dati vengano messi in coda come documento autonomo (**body**).
Ha proprietà di **idempotenza**: gli effetti collaterali di più richieste identiche sono gli stessi di una singola richiesta. Dunque vogliamo che un oggetto se non c'è si crea, se c'è si aggiorna e non se ne crea uno nuovo.

HEAD

Simile al metodo GET ma viene restituito solo l'head della pagine web, spesso usato solo in fase di **debugging**.

4.3.2 Messaggi HTTP

Esistono due tipi di messaggi, **request** e **response**, che hanno però la stessa struttura:

- **start-line** obbligatorio, specifica il protocollo. È necessario spazio tra le parti del protocollo. Serve un **Carriage-Return Line Feed** alla fine.
- **Header lines** opzionale: formato da coppie nome-valore arbitrarie, sono qualificatori di domanda e risposta. CRLF finale per ogni coppia ed empty line finale per terminare l'header line.
- **Payload** opzionale, il contenuto vero e proprio che vogliamo mandare.

Esempio

Start-line: Part_1 Space Part_2 Space Part_3 CRLF
Header lines: Header_field_name:value CRLF
...
Header_field_name:value CRLF
CRLF
Payload: message_body

4.3.3 Codici di stato response

Forniscono informazioni relative alla response:

- **1xx** richiesta ricevuta
- **2xx** richiesta ricevuta, compiuta, capita, accettata e servita
- **3xx** azione aggiuntiva deve essere presa
- **4xx** la richiesta contiene errori e non può essere capita
- **5xx** il server fallisce l'adempimento a una richiesta apparentemente valida.

4.3.4 MIME

Multipurpose Internet Mail Extension, servono per qualificare i dati inviati via internet in 5 sottotipi: text, image, audio, applicazion e video.

4.3.5 Cookie

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

4.4 Comunicazione a flusso di messaggio

Un'applicazione, invia i messaggi come stream di byte al servizio di trasporto, e legge lo stream di byte dal servizio di trasporto e ricostruisce i messaggi.

4.4.1 Servizio UDP

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

4.4.2 Servizio TCP

- scompone e invia come UDP
- ogni segmento viene numerato per garantire
- riordinamento dei segmenti arrivati
- controllo delle duplicazioni (scarto dei segmenti con ugual numero d'ordine)
- controllo delle perdite (rinvio dei segmenti mancanti)

4.5 Comunicazione

4.5.1 Tipi di connessione

Una connessione può essere:

- **sincrona** due o più interlocutori sono collegati contemporaneamente

· **asincrona** non richiede il collegamento contemporaneo degli interlocutori alla rete di comunicazione

4.5.2 Sincronizzazione

Può avvenire in 3 punti:

· **request submission** il client invia un messaggio che viene preso in carico dal middleware

· **request delivery** viene poi trasferito al middleware ricevente

· **request processing** lo passa al server che tratta il messaggio ricevuto. Questa si divide in 3 ulteriori fasi:

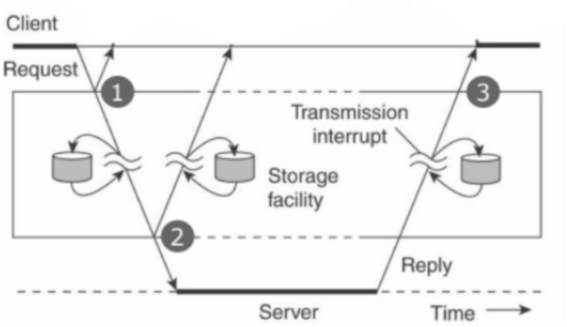
  – ricevuto

  – letto

  – processato

Nei vari passaggi, dei **buffer** tengono traccia dei messaggi, che permettono la sincronizzazione. Quest'ultima, è da considerarsi in relazione al client o al server, si è sincroni rispetto a qualcosa.

Figura 4.1: Punti di sincronizzazione



Caratteritiche di sincronia

La Sincronizzazione può essere:

· **transiente** il destinatario non è connesso e i dati vengono scartati

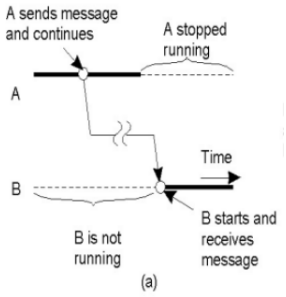
· **persistente** il middleware memorizza i dati fino alla consegna del messaggio al destinatario. Non è necessario che i processi siano in esecuzione prima e dopo l'invio/ricezione dei messaggi

4.6 Tipi di comunicazione

4.6.1 Comunicazione persistente asincrona

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

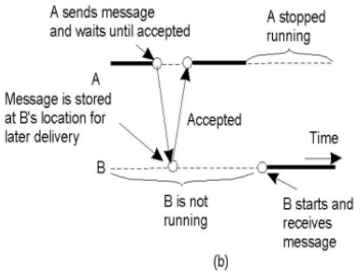
Figura 4.2: Comunicazione persistente asincrona



4.6.2 Comunicazione persistente sincrona

A attende l'accettazione da parte del middleware di B.

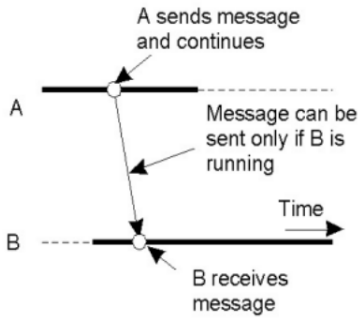
Figura 4.3: Comunicazione persistente sincrona



4.6.3 Comunicazione transiente asincrona

A può inviare solo se B è in run e pronto a ricevere.

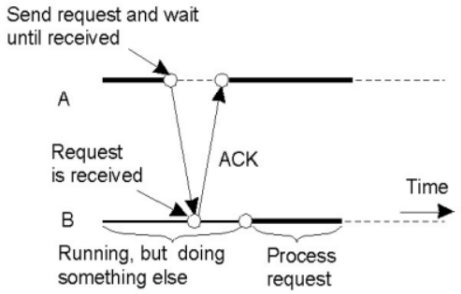
Figura 4.4: Comunicazione transiente asincrona



4.6.4 Comunicazione transiente sincrona basata su ricevuta

B non processa subito, legge solo inviando un ack, e processa più avanti. Fa diventare persistente la comunicazione, se il sistema non lo fa, è il programma che mantiene il dato non un middleware.

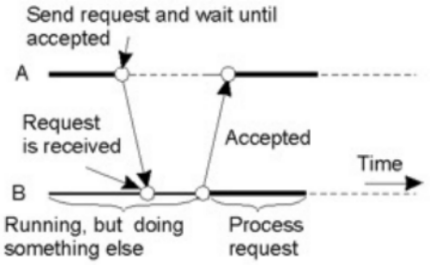
Figura 4.5: Comunicazione transiente sincrona basata su ricevuta



4.6.5 Comunicazione transiente sincrona basata su consegna

uguale al modello precedente, ma deve riceverla e poi leggerla anche prima di inviare un accept.

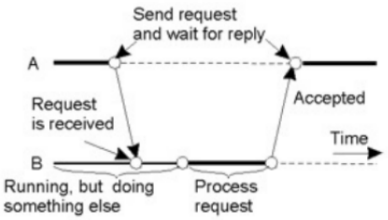
Figura 4.6: Comunicazione transiente sincrona basata su consegna



4.6.6 Comunicazione transiente sincrona basata su risposta

dopo aver mandato la richiesta aspetta la risposta.

Figura 4.7: Comunicazione transiente sincrona basata su risposta



4.7 Comunicazione persistente

4.7.1 Message-queueing model

offrono capacità di archiviazione senza richiedere che il mittente o destinatario siano attivi durante la trasmissione di messaggi. Si può avere in 4 modalità, a seconda dello stato di sender e receiver, che può essere **running** o **passive.**

4.7.2 Primitive

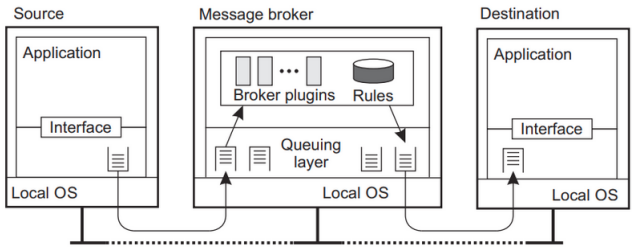
Ci sono 4 primitive:

- **put** aggiunge un messaggio ad una coda
- **get** blocca fino a quando la coda specificata non è vuota e rimuove il primo messaggio
- **poll** controlla una coda specifica e rimuove il primo messaggio senza bloccare
- **notify** installa un gestore da chiamare quando un messaggio viene inserito in una coda

4.7.3 Message brokers

Usando delle code si possono realizzare dei **broker** che permettono il pattern **publish e subscribe**. Ha il compito di ricevere dati da una sorgente e condividerli a tutti quelli che ascoltano su un dato canale.

Figura 4.8: Archiettura di un broker



4.7.4 PaS Pattern

È un disaccoppiamento tra publisher e subscriber:

- **publisher** si registrano dichiarando di voler inviare messaggi su un determinato argomento
- **subscriber** si registrano dichiarando di voler ricevere messaggi su un determinato argomento

Il publisher e il subscriber non comunicano direttamente, si ha infatti un **disaccoppiamento**, quindi indipendenza e concorrenza, con conseguente **scalabilità**. Il broker può mantenere i dati nel tempo, a seconda di diverse politiche.

LA **MQTT** è l'implementazione più usata di Pas Pattern.

Capitolo 5

Web App, Servlet e Servizi

5.1 Applicazioni Web

5.1.1 Caratteristiche web

Il web è utilizzato per la creazione di applicazioni perchè rappresenta uno standard diffuso, che offre semplicità e uniformità di interazione e un'infrastruttura completa di sistemi aperti, completi e potenti.

5.1.2 Tecnologia server side

la computazione in questo tipo di sistema avviene **lato server**, attraverso programmi interpretati o compilati, come gli script.

nel caso di programmi compilati, il web server si limita ad invocare, su richiesta del client, un eseguibile. Nel caso invece debba eseguire degli script, il web server ha al uso interno un **motore** in grado di interpretare il linguaggio di scripting usato. Con il secondo sistema si perde in velocità di esecuzione, ma si guadagna in termini di **scrittura del codice**.

Architettura di un'applicazione compilata

Sono architetture che richiedono la presenza di una **Common gateway Interface**, protocollo che permette al server di attivare un programma, passargli le richieste e i paramentri provenienti dal client e recuperare la rispsosta elaborata. Ogni applicazione di questo tipo deve quindi implementare l'interprete del protocollo.

Architettura di un'applicazione interpetata

In questo caso, il protocollo CGI viene gestito dall'interprete del linguaggio utilizzato e non dalle applicazioni.
Rispetto all'approccio precedente, ha una serie di vantaggi:

- serve programmare solo le logiche delle applicazioni
- modello delle applicazioni conforme al modello del linguaggio utilizzato
- semplice, portabile e manutenibile

5.2 Client side HTML

Nel lato client, ovvero dal punto di vista del **client side**, l'elaborazione è trasparente e viene presentato solo il risultato, in formato HTML.
Il risultato viene elaborato da un browser, che può effettuare diversi tipi di richieste.

5.2.1 Richieste basate su link

Un link in un documento HTML può essere usato per puntare ad una risorsa remota.
Il server invierà richieste del tipo `GET /Servlet/GetHTTP HTTP/1.1`.

Esempio

```
<p>Click the link</p>
<a href="http://localhost:8080/Servlet/GetHTTP">
    Get HTML Document
</a>
```

5.2.2 Richieste per mezzo di Form

Anche il parametro `action` di un form può essere usato per puntare ad una risorsa remota, ovvero ad un'applicazione.
Il server invierà richieste del tipo `GET /Servlet/GetHTTP HTTP/1.1`.

Esempio

```
<form action="http://localhost:8080/Servlet/GetHTTP"
    method="GET">
    <p>Click the button</p>
    <input type="submit" value="Get HTML Document">
</form>
```

Richieste con dati

Un form può essere usato anche per mandare dati al server.
In questo caso, la richiesta inviata dal browser sarà più complicata, del tipo:

```
POST /SlideServlet/PostHTTPServlet HTTP/1.1
Content-Length: 11
Content-Type: application/x-www-form-urlencoded
param=value
```

Da notare che la richiesta, in questo caso, è stata effettuata tramite il metodo POST; è possibile utilizzare anche il metodo GET, che accoda i parametri all'URL della richiesta effettuata, elaborando una richiesta del tipo:

```
GET /Servlet/GetPostHTTP?animal=none HTTP/1.1
```

Esempio

```
<form action="http://localhost:8080/Servlet/PostHTTP"
    method="POST">
    What is your favorite pet?<br> <br>
    <input type="radio" name="animal" value="dog">Dog<br>
    <input type="radio" name="animal" value="cat">Cat<br>
    <input type="radio" name="animal" value="bird">Bird<br>
    <br> <input type="submit" value="Submit">
    <input type="reset">
</form>
```

5.3 Server side Java Servlet

Le Java Servlet sono **piccole applicazioni Java** residenti sul server. Una servlet non è altro che una **componente** gestita in modo automatico da un **container** o **engine**. Quest'ultimo controlla la servlet, ovvero ne controlla il ciclo di vita, in base alle richieste dei client.
Questo è possibile perchè le servlet implementano una interfaccia nota al server, che definisce il set di metodi.

5.3.1 Stateful

HTTP non prevede la persistenza (**stateless**), quindi non si possono mantenere informazioni tra un messaggio e i successivi e non si possono identificare i client.
Queste caratteristiche sono da ricercare nelle applicazioni, che le implementano in due modi:

- **cookies** sono informazioni memorizzate a livello di client, che permettono di gestire sessioni di lavoro
- **HTTPSession** è un oggetto gestito automaticamente dal container/engine, con cookie o riscrittura delle URL. Le servlet possono accedervi per immagazzinare informazioni.

5.3.2 Interfaccia e classi Servlet

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

- `void init(servletConfig config)` inizializza la servlet, viene invocato dopo la creazione della stessa
- `void destroy()` chiamata quando la servlet termina
- `void service(ServletRequest request, ServletResponse response)` invocato per gestire le richieste dei client
- `ServletConfig getServletConfig()` restituisce i parametri di inizializzazione e il `ServletContext` che da accesso all'ambiente
- `String getServletInfo()` restituisce informazioni sulla servlet, come autore e versione

L'interfaccia è solo la dichiarazione dei metodi che, per essere utilizzabili, devono essere implementati in una classe. Per questo sono presenti due **classi astratte**, cioè che implementano i metodi dell'interfaccia in modo che non facciano nulla:

- `jakarta.servlet.GenericServlet` definisce metodi indipendenti dal protocollo
- `jaakrta.servlet.http.HTTPServlet` definisce metodi per l'uso in ambiente web

Questo semplifica limplementazione delle servlet vere e proprie in quanto basta implementare, ridefinendoli, solo i metodi che interessano.

Classe HTTPServlet

Implementa il metodo `service()` che invoca i metodi per servire le richieste dal web.
Implementa anche diversi metodi del tipo `doX`, per effettuare get, post...
Oltre a queste, sono presenti anche metodi per gestire le richieste e le risposte:

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

5.3.3 Ciclo di vita di una Servlet

Una servlet viene creata dai container quando viene effettuata la **prima chiamata**, la servlet viene condivisa da tutti i client e ogni richiesta genera un **thread** che esegue la `doXXX` appropriata.
Il container invoca il metodo `init()` per inizializzazioni specifiche.

Una servlet viene invece distrutta, sempre dal container, quando non ci sono thread in esecuzione su quella servlet, oppure quando è scaduto un **tiemout** predefinito.
La terminazione avviene invocando il metodo `destroy()`.
Alla scadenza del timeout potrebbe essere ancora che dei thread siano in

esecuzione in `service()`. Bisogna progettare il metodo `destoy()` in modo da notificare lo shutdown e attendere il completamento del meotodo `service()`. Anche i metodi devono implementare tecniche per verificare periodicamente se è in corso uno shutdown e comportarsi di conseguenza.

Thread

Le servlet implementano il modello a thread di cui sopra, per ragioni di utilizzo di meno memoria e riduzione del costo di gestione. Però, è necessario porre attenzione alle risorse condivise e come avviene l'accesso. Spesso i modelli utilizzati sono a livello di **database**, dato che già implementano meccanismi di mutua esclusione.

5.3.4 Realizzazioni di applicazioni

Le Servlet classiche fanno utilizzo solo dell'ereditarietà, supportano tutti i metodi ma sono lunghi. Si possono utilizzare dei **framework** per semplificare il processo di realizzazione di applicazioni.

Spring@MVC

Gestisce tutti i metodi HTTTP, attraverso l'uso di annotazioni per assegnare i metodi

JAX-RS

Framework REST, basato sempre sulle annotazioni.

5.4 Server side JSP

La **Java Server Pages** è una tecnologia per la creazione di applicazioni web. Specifica l'interazione tra un contenitore/server ed un insieme di pagine che presentano informazioni all'utente.
Le pagine sono costituite da tag tradizionali e da **tag applicativi** che controllano la generazione del contenuto, generato server-side. Questi tag sono <% e %>.
Nella pratica, una Java Server Page chiama un programma Java eseguito sul Web Server. La pagina viene convertita **automaticamente** in una servlet java la prima volta che viene richiesta.

5.4.1 Elementi di una JSP

Gli elementi, considerando anche i tag JSP, sono:

- **template text** le parti statiche della pagina HTML
- **commenti** del tipo <%-- commento --%>
- **direttive** non influenzano la gestione di una singola richiesta HTTP ma influenzano le proprietà generali della JSP e come questa deve essere tradotta in una servlet. Sono del tipo <%@ dir %>
- **azioni** permettono di supportare diversi comportamenti e vengono processati ad ogni invocazione della pagina, inoltre permettono di trasferire il controllo da una JSP all'altra e di interagire con i Beans. Sono del tipo <jsp:XXX attributes>body </jsp:XXX>

5.4.2 Direttive

Page

Liste di attributi/valore, che valgono per la pagina in cui sono inseriti.

Esempio

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

Include

Include in compilazione pagine HTML o JSP.

Esempio

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

Taglib

Dichiara tag definiti dall'utente implementando opportune classi.

Esempio

```
<%@ taglib uri="TableTagLibrary" prefix="table" %>
```

5.4.3 Azioni

Forward

Determina l'invio della richiesta corrente, eventualmente aggiornata con ulteriori paramentri, alla JSP indicata.

Esempio

```
<jsp:forward page="login.jsp" >
<jsp:param name="username" value="user" />
<jsp:param name="password" value="pass" />
</jsp:forward>
```

Include

Invia dinamicamente la richiesta ad una data URL e ne include il risultato.

Esempio

```
<jsp:include page="shoppingCart.jsp" />
```

useBean

localizza ed instanzia, se necessario, un **JavaBean** nel contesto specificato, che può essere la pagina, la richiesta, la sessione o l'applicazione.

Esempio

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

5.4.4 Elementi di scripting

Declaration

Variabili o metodi statici usati nella pagina, sono nella forma `<%! declaration [declaration] ... %>`.

Expression

Una espressione nel linguaggio di scripting (Java) che viene valutata e sostituita al tag. Sono nella forma `<%= expression %>`.

Scriptlet

Frammenti di codice che controllano la generazione del codice HTML, valutati alla richiesta. Tale codice diventerà parte dei metodi doXXX della servlet che viene associata alla JSP.

Esempio

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

5.4.5 Oggetti

Gli oggetti possono essere creati implicitamente usando le direttive JSP, esplicitamente con le azioni o direttamente usando uno script. GLi oggetti hanno un attributo che ne definisce lo **scope**, ovvero il livello di visibilità dell'oggetto stesso.

Livelli di visibilità

- **application** accessibili da pagine che appartengono alla stessa applicazione
- **sessione** accessibili da pagine che appartengono alla stessa sessione
- **request** accessibili solo dalle pagine processate da richieste provenienti da dove sono stati creati

- **page** accessibili solo dalla pagine in cui sono stati creati

5.4.6 JavaBean

Un JavaBean è una classe che esegue regole precise, ovvero:

- deve avere costruttori senza parametri
- dovrebbe avere campi privati
- i metodi di accesso ai campi sono nella forma `setXXX`, `getXXX`, `isXXX`.

Esempio

```
public class CounterBean {
    private int count;
    public int getCount() {
        return count;
    }
    public void increaseCount() {
        count++;
    }
}
```

5.5 Pattern MVC

Il pattern architettuale MVC, ovvero **Model View Controller**, ha lo scopo di separare data model, user interface e control logic in tre componenti distinte:

- **model** i dati e i metodi principali per la loro manipolazione
- **view** la presentazione, cioè l'interfaccia
- **controller** il coordinamento dell'interazione tra interfaccia, ovvero le azioni degli utenti, e i dati

Figura 5.1: Il pattern

5.5.1 Vantaggi e svantaggi

Ha vari vantaggi:

- chiara separazione tra logica di business e logica di presentazione
- chiara separazione tra logica di business e modello dei dati
- ogni compomente ha una responsabilità ben definita
- ogni parte può essere affidata a esperti

Però, dalla separazione nascono anche vantaggi:

- aumento della complessità dovuta alla concorrenza, ricordiamo infatti che è un **sistema distribuito**
- inefficienza nel passaggio dei dati alla view, ovvero è presente un elemento in più tra cliente e controller

5.5.2 Pattern Model

Pattern Model 1

Ha come scopo la separazione completa tra dati, logica di business e visualizzazione, prevedendo 4 passaggi:

1. il browser invia una richiesta per la pagina JSP
2. JSP accede a Java Bean e invoca la logica di business
3. Java Bean si connette al database e ottiene/salva i dati
4. la risposta, generata da JSP, viene inviata al browser

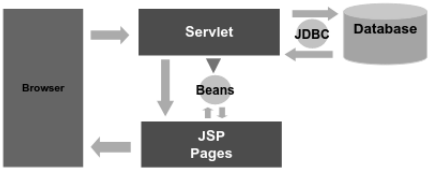
Ha però diversi svantaggi, infatti il controllo della navigazione è **decentralizzato**, poichè ogni pagine contiene la logica per determinare la pagina successiva. Inoltre, è necessaria la creazione di molti **tag ad hoc** per evitare di inserire troppo codice Java all'interno del codice HTML.

Figura 5.2: Architettura tipica del model 1

Pattern Model 2

La richiesta viene inviata ad una Java Servlet che genera i dati dinamici richiesti dall'utente e li mette a disposizione della pagine jsp come **Java Beans**. La servlet richiama quindi una pagina JSP che legge i dati dai beans e organizza la presentazione in HTML che invia all'utente.

Figura 5.3: Architettura tipica del model 2



5.6 Services Computing

5.6.1 IoT

L' **internet of Things** è un sistema di oggetti fisici che possono essere individuati, monitorati e controllati da dispositivi elettronici in grado di comunicare attraverso diverse interfacce e, eventualmente, in grado di comunicare con reti esterne.

5.6.2 Service oriented Architecture

Tradizionalmente, i sistemi distribuiti sono singoli sistemi, sviluppati in **top-down**, generalmente client-server e chiusi.
nei sistemi contemporanei, i sistemi includono anche **sistemi di terze parti** e **sotto-sistemi**, ognuno indipendente dagli altri.

SI introduce quindi lo stile architetturale **SOA**, che si focalizza sull'uso di pezzi discreti riusabili, invece di design monolitici. Quindi, un **servizio** non è altro che un insieme di funzionalità che offre a dei richiedenti.
Ha come vantaggi la riusabilità, l'adattamento al modello Agile, l'economia e la scalabilità ma, d'altra parte, ha delle difficoltà in termini di **dipendenze** e integrazione con **legacy solutions**.

Servizio

Ogni servizio dovrebbe:

- fornire una **descrizione** delle sue funzionalità per poter essere scoperto e selezionato
- fornire l'accesso ae sue funzionalità tramite protocolli di rete noti
- sostenere la composizione con altri servizi per fornire soluzioni complesse
- rispondere alle esigenze aziendali e al dominio dei clienti requisiti
- garantire un certo livello di qualità del servizio (**QoS**)

5.6.3 Web Service

un servizio web è un'entità software indipendente che può essere scoperta e richiamata da altri sistemi software su una rete, sono quindi componenti indipendenti che:

- hanno un'**interfaccia** ben nota, usano un linguaggio di descrizione standard e implementano **middleware** di gestione automatica
- hanno un **punto di accesso unico**, usano un URI (URL o URN) attraverso il quale possiamo scoprirlo. Lo standard UDDI definisce proprio degli standard per la pubblicazione e la scoperta dei **name services**
- scambiano dati basandosi su **documenti**, con dei formati di rappresentazione standard come JSON e XML

Ruoli dei servizi

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

5.6.4 Service Level Agreement

Uno SLA definisce le caratteristiche non funzionali garantite dal servizio. Include diversi SLO, ovvero **Service Level Objectives** che definiscono la qualità del servizio da garantire attraverso metriche specifiche.

5.6.5 Composing Service

Una composizione di servizi è costituita da un insieme di servizi **interconnessi**, che possono essere utilizzati come un nuovo servizio in altre composizioni. Due servizi sono interconnessi se almeno uno dei due richiede la funzionalità esposta dall'altro.
Inoltre, devono essere **compatibili**, devono usare un protocollo comune e presentare delle **published interface**.

5.6.6 Processo Aziendale

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

Orchestrazione

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

Coreografia

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

Enterpirse Service Bus

L'ESB è un sistema di comuniczione per supportare l'interazione e la comunicazione tra i componenti di un sistema informativo. È un esempio di approccio coreografico e solitamente implementa il modello publish/subscribe.

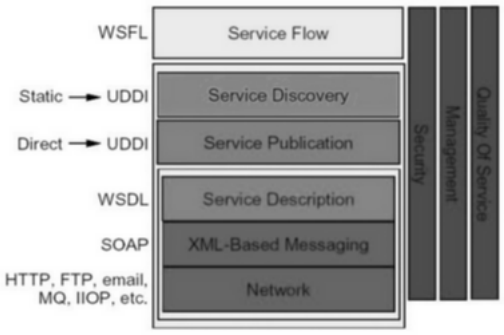
5.7 SOAP Services

5.7.1 Web Service Stack

Il seguente stack si riferisce esclusivamente ad un servizio SOAP. Si suddivide in:

- **Network-Transport** rappresenta i protocolli internet utilizzabili in un processo SOAP per far comunicaionre due servizi
- **messagging** SOAP è il linguaggio che definsce la struttura di un messaggeio che si può inviare
- **Description** WSDL descrive il messaggio, quali sono le funzionalità
- **search and find** permette di pubblicarli attraverso dei cataloghi
- **Business Process** peremmte di integrarli in maniera automatizzata

Figura 5.4: SOAP Stack



5.7.2 SOAP

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

5.7.3 Componenti SOAP

SOAP Envelope

Fa il wrapping del contenuto del messaggio.

SOAP Header

È opzionale, prevede una maggiore flessibilità e può essere elaborato da nodi tra sorgente e destinazione. Contiene blocchi di informazioni su come elaborare il messaggio.

SOAP Body

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

5.7.4 Modelli SOAP con HTTP

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

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

5.7.5 Web Service Description Language

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

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

La descrizione può essere fatta attraverso due modelli concettuali.

Astratti

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

Concreti

I **binding** specificano il protocollo di trasporto per le interfacce, un **endpoint** associa un URI a un binding e un **servizio** raggruppa gli endpoint che implementano una comune interfaccia.

5.7.6 Attualità

WSDL e SOAP sono caduti sotto il loro stesso peso, perchè erano difficili da capire e complessi. Si è ritenuto necessario un approccio più semplice e leggibile con **semantica concordata** e utilizzo del formato standard JSON.

5.8 REST Services

5.8.1 Modello comune

Il principio REST si basa sulla **composizione**, ovvero lo stabilre un modello comune. I sistemi distribuiti lavorano sulla base di un **modello condiviso**, mentre REST è costruito sull'idea di **semplificare l'agreement**, ovvero si basa su delle **considerazioni di default**:

- sono necessari **nomi** per nomenclare le risorse
- **verbi** che sono le operazioni applicabili sulle risorse
- **content types** che definiscono quali sono le rappresentazioni delle informazioni disponibili

5.8.2 Principi REST

REST vuol dire **REpresentational State Transfer**, e si basa sugli stessi principi dell'HTTP, eliminando le ridondande di SOAp e assegnando la semantica a verbi e URL.

5.8.3 Caratteriscihe Restful

1. le risorse devono avere identificatore che è parte del pattern (**path parameter)**
2. l'interfaccia ad una risorsa deve essere **uniforme**, con i 4 verbi HTTP: GET, POST, PUT, DELETE
3. l'architettura è client/server

4. la **manipolazione** avviene lato client
5. le risorse dovrebbero avere associati dei link
6. Le risorse sono **prive di stato**, non sanno chi esegue la richiesta, non essendoci sessioni attive nè uno stato

5.8.4 Caching

Dai punti *2*, *6*, *7* delle caratteristiche restful, si ottiene la necessità di implementare una **cache**: uno spazio di memoria intermedio, tra client e server, che riduce la latenza e la quantità di dati.

Si implementa con una cache locale che si aggiorna solo quando avvengono delle modifiche; le richieste sono quindi **condizionali**, e la cache viene usata grazie ai tag **Cache-Control** e **e-tag** che verificano proprio se la risorsa è stat modificata o no, e quindi se è possibile o meno utilizzare la cache.

5.8.5 Costruzione di applicazioni REST

Si seguono 4 passi consecutivi:

1. Trovare tutti i nomi: bisogna definire URI descrittive, **opache** e persistenti, che si possano evolvere solo attraverso il **versionamento**. Bisogna evitare query, gerarchie, uso di verbi e preferire i sostantivi
2. definire i formati, evitando di creare rappresentazioni personalizzate
3. scegliere le operazioni possibili, che spesso sono le 4 operazioni GET, POST, PUT, DELETE che sono sufficienti, ma esistono comunque altri metodi HTTP
4. evidenziare i codici di stato eccezionali

5.8.6 Overloaded POST

Solitamente, si può mascherare una **Remote Procedure Call** attraverso l'overloading del metodo POST: il POST fa qualcosa e restituisce qualcos'altro, seguendo questa logica si può invocare una procedura remota tramite una richiesta POST.
È un'operazione da evitare se possibile, e comunque che bisogna utilizzare con attenzione.

5.8.7 Operazioni asincrone

Si permettono operazioni esponendo un certo stato di accettazione, ma elaborando poi la richiesta. La **coda** diventa una risorsa.

Esempio

Se ipotizziamo di implementare un metodo di pagamento, si può restituire lo stato di accettato rispetto alla richiesta di un pagamento, ma il pagamento avverrà in un secondo momento e in un altro momento ulteriore verrà controllato se il pagamento è andato effettivamente a buon fine.

5.8.8 Statelessness

Ogni richiesta avviene in modo isolato, sono i **client che guidano lo stato dell'applicazione**, tutte le informazioni necessarie all'esecuzione sono presenti nel body o nell'header della richiesta, e il server non usa informazioni da richieste precedenti, non vi è quindi un **contesto**.

In questo senso, anche uno **stato** diviene una **risorsa**, con un proprio URI, ricordando erò che deve comunque essere tutto gestito dal client.

Grazie alla caratteristica di essere senza stato, le applicazioni sono bilanciate e si può partizionare, scalare e aggiugnere cache in maniera facile, senza l'uso di sessioni, cookie o chiavi.

5.8.9 Hypermedia control

L'HATEOAS, ovvero l'**Hypermedia As The Engine Of Application State**, consente di vedere un link come uno **stato di transizione** nell'applicazione, il server guida il client attraverso diversi stati fonrendo i link relativi ad una certa risorsa richiesta.

Questo consente di scoprire meglio le risorse e di avere una sorta di **ordine di interazione**.