PROLOGO AI SISTEMI DISTRIBUITI

Un sistema distribuito consiste in un insieme di macchine, che sono gestite in maniera autonoma, connesse attraverso una rete. Ogni nodo del sistema è indipendente e coordina il proprio lavoro attraverso uno strato software, definito **middleware**, che fa percepire il sistema all'utente come un'unica rete. Un sistema distribuito, normalmente, risponde sia a motivazioni economiche sia a motivazioni tecnologiche. Nel contesto **economico**, i sistemi distribuiti rispondono a precise esigenze delle aziende, caratterizzate da acquisizioni, integrazioni e fusioni. Per questo, abbiamo bisogno di un sistema versatile e agile. Un sistema distribuito risponde anche a problematiche nel contesto **tecnologico**, dove abbiamo poco tempo per perfezionare il prodotto finale e dobbiamo far fronte ad un progresso continuo. Inoltre, molti sistemi, sono accessibili ad un'enorme platea di utenti e quindi soggetti a picchi di carico. Ovviamente, sappiamo che lo sviluppo dell'informatica è strettamente legato allo sviluppo delle tecnologie **hardware**, che avanzano ancora più velocemente di quelle software. Nel corso della storia, diverse **"leggi"** hanno previsto l'evoluzione di questi sistemi. Scriviamo leggi tra virgolette poiché, per quanto rivelatisi affidabili, non sono dimostrate, quindi puramente **empiriche**. Vediamone alcune:

- **"Legge" di Moore**: afferma che, all'interno di un processore, il numero di transistor raddoppia ogni 18 mesi. Per esempio, nel 1997 la Intel (Moore è uno dei fondatori) ha lanciato il Pentium 2 che aveva 7,5 milioni di transistor. Dopo 18 mesi, ne aveva 15 milioni, dopo 36 ne aveva 30 e dopo 42 ne aveva addirittura 42 milioni, quindi una crescita stimata inferiore rispetto a quella che è davvero avvenuta. Questa "legge" aveva, però, alcuni limiti fisici, infatti aumentando i transistor in maniera così elevata, era facile incorrere nel cosiddetto "thermal noise", un rumore termico generato dall'agitazione dei portatori di carico.

- **"Legge" di Sarnoff**: afferma che il valore di una rete di broadcast è proporzionale al numero di utenti. Una rete di broadcast significa che esiste un servizio raggiungibile ad un numero *n* di utenti, quindi ci riferiamo perlopiù alla diffusione dei programmi televisivi e radiofonici.
- **"Legge" di Metcalfe**: afferma che il valore di una rete di comunicazione cresce con il quadrato del numero di persone collegate. In una rete di comunicazione, un qualsiasi nodo può raggiungere altri nodi, così come accade con gli SMS.
- **"Legge" di Reed**: afferma che una rete sociale cresce in maniera esponenziale se associata a nodi con interessi comuni che condividono idee e obiettivi e che hanno un qualche senso di appartenenza. Più gruppi ci sono, più la rete monetizza. Un esempio lampante di questa legge è il successo ottenuto dai social network, come Facebook e Instagram.

OPEN DISTRIBUTED PROCESS

Per facilitare lo sviluppo dei sistemi distribuiti, è importante condividere un modello comune, chiamato **modello di riferimento**, spesso abbreviato con **RM-ODP**, acronimo di Reference Model of Open Distributed Process. Questo modello è molto utile per gestire i problemi di comunicazione, astraendo e standardizzando la portabilità e la trasparenza. Un sistema distribuito ha alcune caratteristiche fondamentali:

- **remoto**: le componenti devono poter essere localizzate su macchine diverse.
- **concorrenza**: un sistema distribuito deve essere concorrente, quindi supportare la contemporanea esecuzione di due o più processi. Quindi non esistono i lock e semafori che gestiscono la sincronizzazione. Non esiste neanche uno stato globale, poiché non c'è un clock in comune.
- **malfunzionamenti parziali**: ogni componente può tranquillamente smettere di funzionare senza bloccare l'intero sistema, come in un normale sistema centralizzato.
- **eterogeneità**: un sistema distribuito è, per norma, eterogeneo in ogni contesto, cioè può avere realizzazioni e implementazioni differenti in base all'hardware, il sistema operativo o i protocolli di rete.
- **autonomia**: tutte le componenti sono indipendenti e collaborano insieme.
- **evoluzione**: un sistema distribuito deve assecondare l'evoluzione dell'ambiente all'interno del quale è realizzato e implementato.
- **mobilità**: non solo degli utenti ma anche di nodi diversi.

Un sistema distribuito presenta anche alcuni requisiti non funzionali, che non sono collegati alle funzionalità e che specificano la qualità del sistema. Un sistema distribuito deve essere:

- **aperto**: deve usare interfacce e standard noti, per favorirne l'interoperabilità e l'evoluzione.

- **integrato**: deve poter usare risorse diverse senza usare strumenti ad hoc, cioè adibiti proprio a quel determinato scopo.
- **flessibile**: deve essere in grado di integrare al suo interno sistemi legacy, vale a dire sistemi obsoleti che per un motivo o un altro non possono essere rimpiazzati e, inoltre, deve poter gestire le modifiche durante l'evoluzione.
- **modulare**: ogni componente è autonoma e non interdipendente dal sistema.
- **scalabile**: deve poter gestire forti picchi di carico da parte dell'utenza.
- **sicuro**: deve fare in modo che utenti non autorizzati non abbiano accesso ad esso. Si tratta di un requisito complicato da realizzare a causa della modularità e della natura remota.
- **trasparente**: deve permettere all'utente di visualizzare il sistema distribuito come un'unica entità, nascondendo dettagli irrilevanti.

La trasparenza caratterizza fortemente un sistema distribuito. In base alle sua caratteristiche abbiamo diversi tipi di trasparenza:

- **trasparenza di accesso**: nasconde le differenze nella rappresentazione dei dati, fornendo un'unica interfaccia sia da remoto che da locale. Si tratta di una trasparenza fornita di default.
- **trasparenza di locazione**: nasconde le informazioni di posizione circa una componente all'interno del sistema. Anche questa trasparenza è fornita di default.
- **trasparenza di migrazione**: nasconde la possibilità di far migrare oggetti da un nodo all'altro, senza che l'utente ne sia a conoscenza. Dipende dalla trasparenza di accesso (accesso all'oggetto) e di locazione (la posizione dell'oggetto).
- **trasparenza di replica**: un oggetto viene replicato in varie copie su nodi diversi dei sistema. Questa trasparenza si occupa di rendere coerente le copie con lo stato originale. Dipende dalla trasparenza di accesso (accesso alle copie) e di locazione (la posizione delle copie).
- **trasparenza alle transazioni**: nasconde all'utente le normali attività di coordinamento svolte durante il corretto processo di concorrenza.

- **trasparenza alla persistenza**: anticipiamo che, un oggetto, viene reso persistente posizionandolo in memoria secondaria. In un sistema distribuito, questo accade senza che l'utente se ne debba occupare. Si tratta di una procedura che ha un meccanismo di attivazione/disattivazione, per risparmiare risorse su oggetti poco utilizzati (**handle**). Dipende della trasparenza di locazione (permette l'handle anche su nodi diversi).
- **trasparenza alla scalabilità**: un sistema, come detto, è scalabile quando è in grado di gestire alti picchi di carico dell'utenza. Questa trasparenza fa in modo che, in vista di queste circostanze, il sistema non debba modificare la sua architettura, integrando risorse. Essa dipende dalla trasparenza di replica (per l'accesso alle risorse) e di migrazione (per lo spostamento di queste risorse).
- **trasparenza alle prestazioni**: nasconde ai progettisti i meccanismi utilizzati per ottenere grandi prestazioni da parte del sistema. Questa trasparenza dipende dalla trasparenza di replica (accesso a nuove risorse che migliorino le prestazioni), di migrazione (posizione di queste risorse), persistenza (rendere persistenti queste risorse).
- **trasparenza ai malfunzionamenti**: nasconde i malfunzionamenti all'utente, permettendo comunque di interagire col sistema, con prestazioni più bassi. Essa dipende dalla trasparenza di replica (per ripetere eventuali operazioni fallite) e dalla trasparenza alle transazioni (operazioni eseguite con transazioni, in caso di errore, non vengono confermate e possono essere ripetute.

LIVELLO BASE

LIVELLO DI FUNZIONALITÀ

LIVELLO DI EFFICIENZA

MIDDLEWARE A OGGETTI DISTRIBUITI

Come già anticipato, il middleware è uno strato software che si trova tra le applicazioni e il sistema operativo, comprendendo anche protocolli di rete e hardware. Il middleware fu creato per risolvere pesanti problemi di comunicazione tra i nodi di un sistema distribuito. Lo scopo del middleware è, quindi, quello di fornire appropriate astrazioni per i programmatori. Il middleware è costituito da tre parti:

- **middleware di infrastruttura**: aiuta nella comunicazione tra sistemi operativi diversi e nella gestione della concorrenza, evitando di usare meccanismo non portabili.
- ▼ **middleware di distribuzione**: autorizza compiti comuni per la comunicazione, come:
  - **marshalling**: passaggio di parametri tramite invocazioni remote.
  - **multiplexing**: passaggio di parametri tramite lo stesso canale di comunicazione.
  - gestione della **semantica delle invocazioni**.
  - riconoscimento o gestione dei **malfunzionamenti**.
- **middleware per servizi comuni**: per gestire persistenza, transazioni e sicurezza.

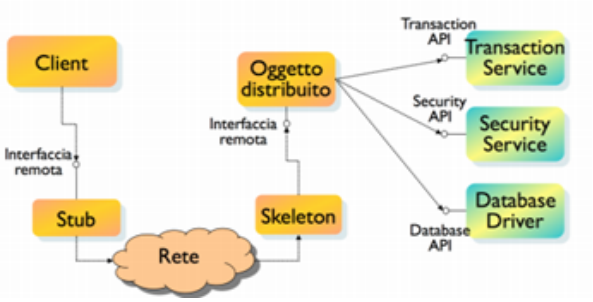
Alla base del middleware e, quindi, di tutte le applicazioni distribuite, ci sono però le Remote Procedure Calls (**RPC**). Esse permettevano ad una procedura di invocarne un'altra, su una macchina diversa. In questo modo, il progettista si concentrava sulle funzionalità delle determinate procedure, senza preoccuparsi del fatto che siano remote o locali. Una RPC richiedeva la traduzione dei tipi di dati invocati, che venivano trasmessi tramite stream di byte su socket, in modo da

codificare e trasmettere i dati in maniera remota (marshalling). Le RPC facevano anche rispettare la **sincronia** dell'invocazione. Tutte queste operazioni venivano implementate tramite uno **stub**, cioè porzioni di codice che simulano il comportamento del codice esistente. Col tempo, però, sono nati alcuni problemi relativi alle RPC, come diversi paradigmi procedurali, tipi di dati elementari e, soprattutto, la mancanza di gestione delle eccezioni. Per questo, si è passati al middleware, capace di gestire sistemi distribuiti molto complessi. Abbiamo diverse implementazioni dei sistemi distribuiti:

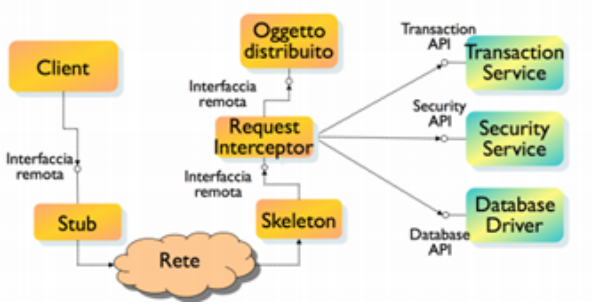
- **CORBA**: acronimo di Common Object Request Broker Architecture, permette ai sistemi distribuiti eterogenei, scritti in linguaggi diversi, di comunicare e collaborare. I linguaggi sono tradotti tramite l'**IDL** (Interface Definition Language). Le invocazioni remote, invece, avvengono attraverso l'**ORB** (Object Request Broker), che effettua l'invocazione in maniera totalmente trasparente al client. Nonostante la sua completezza, CORBA è stato pian piano soppiantato da altri modelli che necessitavano una complessità minore e, soprattutto, un'interoperabilità maggiore.
- **Java RMI**: acronimo di Java Remote Method Invocation, fu la proposta di Sun per le applicazioni distribuite basate sugli oggetti. Esso è realizzato, appunto, in Java e definisce il Middleware all'interno della Java Virtual Machine, in modo che si possa integrare con le altre librerie e possa fornire servizi comuni.
- **Microsoft .Net Framework**: fu la proposta di Microsoft. Essa contiene un ambiente di esecuzione comune, detto CLR (Common Language Runtime), in cui le applicazioni vengono scritte in uno dei linguaggi supportati da Microsoft. All'interno di questa soluzione, Microsoft fornì anche **.Net Remoting**, per favorire la comunicazione tra processi distribuiti.

Abbiamo descritto fino ad ora sistemi che utilizzano un **middleware esplicito**, dove i servizi venivano richiesti dal progettista. Questa procedura presentava alcuni problemi di complessità, portabilità e di controllo sugli errori. Per questo venne presentato il **middleware implicito**, dove i servizi vengono forniti automaticamente dal sistema, sulla base delle richieste specificate. Questa astrazione ha favorito lo sviluppo del Modello a Componenti Distribuite (detto **Enterprise Computing**). Quando parliamo di componenti, ci riferiamo a dei blocchi di software riutilizzabili che vengono combinati con un sistema distribuito per realizzare determinate funzionalità

MIDDLEWARE ESPLICITO



MIDDLEWARE IMPLICITO



PROGRAMMAZIONE CONCORRENTE E THREAD IN JAVA

Come già ribadito, la programmazione distribuita implica la conoscenza e, soprattutto, l'applicazione della programmazione concorrente, in cui più processi vengono eseguiti insieme. Esistono tre tipi fondamentali di programmazione concorrente:

- programmazione concorrente eseguita su calcolatori diversi.
- programmazione concorrente eseguita sulla stessa macchina.
- programmazione concorrente eseguita nello stesso processo.

Prima di proseguire, è necessario fare la netta distinzione tra **multitasking** e **multithread**. Si tratta di una differenza abbastanza banale. Infatti cambia soltanto l'unità elementare di esecuzione: in un sistema multitasking abbiamo il **processo**, mentre in quello multithread, appunto, il **thread**. Infatti, in sistemi operativi multithread, i processi operano attraverso uno o più thread, detti light-weight-process. La differenza tra processo e thread è sostanziale. I processi non condividono lo stesso spazio di indirizzamento e tendono a fare competizione per accaparrarsi dati e periferiche. Invece, thread appartenenti allo stesso processo possono condividere quello spazio di indirizzamento. Un thread migliora l'efficienza di un programma, poiché rende più facile la cooperazione e richiede meno risorse di un normale processo. Per questo che, nella nostra vita quotidiana, troviamo spesso applicazioni multithread, come browser web, server con alti picchi di carico oppure streaming audio. I thread possono essere manipolati in Java. Abbiamo due diversi tipi di implementazione:

```java
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

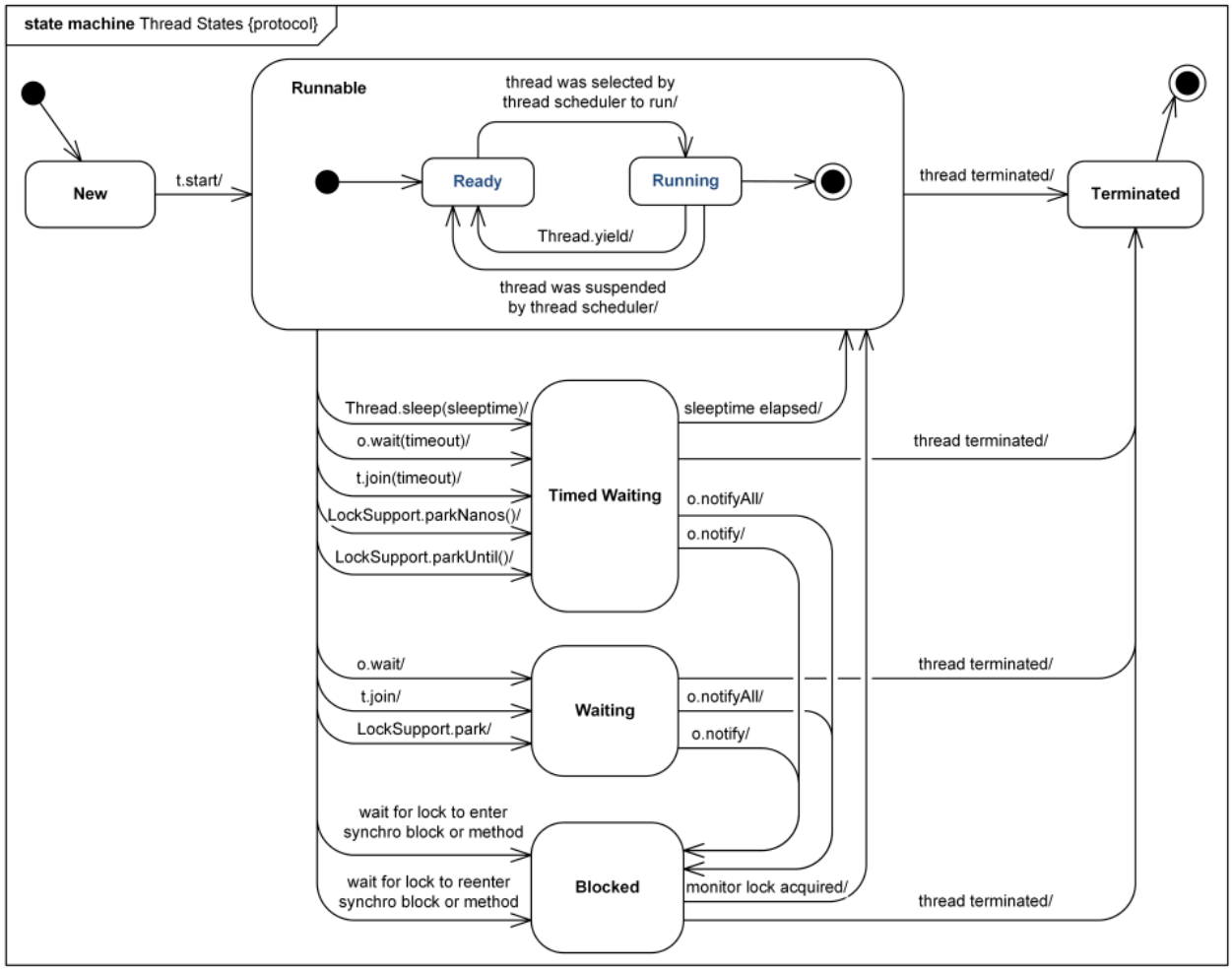
Questa prima implementazione estende la classe `Thread`, lancia il thread con il metodo `run()`. Poi, all'interno del metodo main si istanzia e si lancia il metodo `start()`.

```java
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```

L'implementazione descritta in precedenza presenta alcuni deficit, infatti per quanto semplice, risulta problematico nel caso uno si volesse estendere un'altra classe. Per questo utilizziamo questa seconda implementazione. In questo caso si implementa un'interfaccia `Runnable`, si lancia sempre il metodo `run()` e, all'interno del main, viene istanziato un oggetto `Thread` al costruttore.
Oltre a questi due metodi descritti sopra, la classe `Thread` fornisce anche altri metodi importanti:

- `sleep()`: passandogli un valore in millisecondi, è possibile ritardare l'esecuzione di qualcosa.
- interrupt: non si tratta di un metodo, ma di un'indicazione che dice al thread di fermarsi e fare qualcos'altro. Di solito, viene catturato il tutto da un'eccezione, ma è possibile verificarne lo stato tramite la chiamata `interrupted()`.
- `join()`: mette in pausa il thread nell'attesa che si verifichi qualcos'altro.

Vediamo adesso, in un diagramma in UML, in cui sono rappresentati gli stati di un thread.



Quando viene creato un nuovo thread, viene eseguito il metodo start() e il thread entra nello stato Runnable. All'interno di Runnable, il thread può entrare nello stato Ready. Sarà lo scheduler, poi, a decidere quale thread passerà per primo allo stato Running. Fino a quando non termina, il thread può passare dallo stato Ready allo stato Running molteplici volte. Ma, durante il suo ciclo di vita, può passare anche in altri stati. Può trovarsi in Timed-Waiting, in cui il thread si trova in uno stato di attesa a tempo, dovuto all'intervento di una sleep e di una join. Il thread, invece, può anche trovarsi in uno stato Waiting, in cui il thread è forzatamente costretto ad aspettare, oppure Blocked, che interviene durante i lock e la sincronizzazione, bloccando immediatamente il thread poiché c'è qualche limitazione legata a differenti esecuzioni.

Abbiamo detto che i thread condividono il proprio spazio di indirizzamento e, quindi, condividono l'accesso ai tipi primitivi, ai riferimenti a oggetti ecc. È una comunicazione molto efficiente, ma che fa nascere diversi problemi ed errori, come interferenza e inconsistenza. Per risolvere questi problemi è necessaria la **sincronizzazione**. Quando parliamo di **interferenza**, intendiamo dire che i thread hanno dei problemi di accesso concorrente e si verifica la famosa *race condition*, cioè quando il risultato dell'esecuzione dipende dall'ordine di esecuzione dei suddetti thread. Abbiamo, invece, problemi di inconsistenza quando, a causa di protocolli di cache e ottimizzazioni hardware, i thread hanno una visione diversa dei dati. Per far fronte a questi problemi sarà necessario utilizzare relazioni di tipo happens-before. Questo tipo di relazione fa in modo che la memoria scritta da un thread sia visibile ad un altro thread. Prima di scendere ulteriormente nel dettaglio, è necessario fare alcune premesse. Cominciamo col dire che lo speedup è un numero che misura le prestazioni di due sistemi che elaborano lo stesso problema. La legge di Ahmdal afferma che:

$$S = 1/1 - p + p/n$$

Lo speedup S di un programma X è il rapporto tra il tempo impiegato dal processore per eseguire X, rispetto al tempo impiegato da n processori per eseguire X. Sia p una parte di X, che è possibile parallelizzare, allora avremo la dicitura descritta sopra.

Fondamentalmente, Ahmdal ci dice che la parte sequenziale rallenta il programma e che, quindi, bisogna impegnarsi a rendere la parte parallela predominante rispetto a quella sequenziale. Tornando a noi, ci sono sempre quei famosi problemi della programmazione concorrente. Fortunatamente, Java ci mette a disposizione alcuni strumenti. Parliamo dei **metodi sincronizzati**, cioè dei costrutti in Java che correggono errori di concorrenza. Per rendere un metodo sincronizzato basta soltanto aggiungere `synchronized` alla sua dichiarazione. Questo fa sì che due esecuzioni sullo stesso oggetto non si interfoglino, infatti i thread che sono in attesa rimangono sospesi fino a quando il primo thread non ha finito. Quando un thread esce dal metodo sincronizzato stabilisce una relazione happens-before con le altre invocazioni. Un costruttore non può essere sincronizzato. È buona norme, al momento della costruzione dell'oggetto condiviso, di evitare di far scappare il riferimento prima che si riferisca ad un altro oggetto. Per far fronte a questo basta chiamare il metodo `instances()`. Introduciamo un nuovo strumento utile alla

sincronizzazione, ossia i **lock intrinseci**. Si tratta di un'entità associata al singolo oggetto, che garantisce accesso esclusivo e consistente, stabilendo una relazione happens-before. Un thread acquisisce il lock di un oggetto e lo rilascia quando ha terminato. A livello di codice dobbiamo semplicemente specificare quale oggetto usa il lock, tramite `synchronized(this)`. Definiamo anche i **metodi statici**, che sono associati alla classe, ma non sono operazioni fatte su un oggetto. Per essere sincronizzato un metodo statico acquisisce il lock dell'oggetto tramite `ClassName.class`, che si riferisce alla classe di appartenenza. Nota bene che i metodi sincronizzati statici garantiscono accesso esclusivo a metodi sincronizzati statici, mentre metodi sincronizzati d'istanza garantiscono accesso esclusivo solo a metodi sincronizzati di quell'istanza. L'ultimo strumento che andremo ad analizzare è l'**accesso atomico**. Un'azione atomica è una procedura non interrompibile fino a quando non è finita. È possibile specificare azioni atomiche in Java per:

- read e write su variabili di riferimento o su tipi primitivi.
- read e write sul variabili `volatile` (archiviate nella memoria principale).

Abbiamo descritto gli strumenti utili alla sincronizzazione, ma questa procedura comporta anche alcuni problemi:

- **deadlock(stallo)**: quando due thread sono bloccati, ognuno in attesa dell'altro. Il programma si blocca e va in crash.
- **starvation**: un thread non riesce ad avere accesso ad una risorsa condivisa e quindi non riesce a fare progressi.
- **livelock**: un caso particolare di deadlock, in cui i thread si occupano di rispondere uno alle azioni dell'altro senza bloccarsi.

Introduciamo, adesso, un nuovo elemento utile alla sincronizzazione. Si tratta del **Singleton**. È un design pattern creazionale che garantisce che, di una determinata classe, venga create una sola e unica istanza e che venga fornito un unico punto di accesso globale a tale istanza. Per implementare una classe `Singleton` dobbiamo rendere privato il costruttore della classe e implementare un metodo statico detto `factory`. La classe Singleton viene usata per la memorizzazione di stato e utilizza la **lazy-allocation**, cioè alloca solo quando viene utilizzato per la prima volta. Dato che istanziamo la classe una sola volta, servirà sincronizzarla solo una volta.

PROGRAMMAZIONE SOCKET TCP

Nella programmazione ad oggetti ci sono degli oggetti, appunto, che contengono uno stato e hanno un comportamento. Per potere estendere questo modello anche in ambito distribuito, è necessario poter invocare un metodo da parte di un oggetto remoto. Per invocare metodi in maniera remota, Java mette a disposizione i socket, che utilizzano strumenti e procedure simili a quelli che verranno poi approfonditi in Java RMI. Le classi per trattare i socket sono inserite all'interno del package `java.net`. Normalmente, la comunicazione su Internet avviene tramite dei protocolli, che usano dei socket per ricevere e trasmettere dati. Abbiamo due protocolli che sono coinvolti nell'implementazione dei socket:

- **TCP** (Trasmission Control Protocol): offre una connessione affidabile tra due punti della comunicazione.
- **UDP** (User Datagram protocol): permette solo di inviare pacchetti di dati da un'applicazione all'altra.

I socket TCP sono endpoint di rete che uniscono due programmi. Ogni socket ha un **numero di porta** che identifica l'applicazione. Normalmente, i due programmi coinvolti si identificano come **client** e **server**. Il server è in esecuzione e attende la richiesta di connessione da parte del client. Il server accetta la connessione e assegna un nuovo socket per la comunicazione. La classe `ServerSocket` implementa un socket di connessione che attende e restituisce l'oggetto `Socket`, usato durante la comunicazione, do po aver accettato la connessione col metodo `accept()`. La comunicazione tra client e server avviene attraverso la scrittura e la lettura di **stream**, ognuno associato al socket e, attraverso un meccanismo di **serializzazione** (trasformazione di un oggetto in memoria in una sequenza di byte trasmessi in rete). Un oggetto viene serializzato attraverso la classe `ObjectOutputStream` e verrà poi deserializzato tramite `ObjectInputStream`. Per la creazione degli stream utilizziamo un meccanismo di **wrapping**, in cui ogni classe specializzata prende il costruttore di un'istanza di classi più alte nella gerarchia. Lo

scopo di una classe wrapper è quello di creare un involucro per contenere un tipo primitivo, rendendolo un oggetto e ornandolo di metodi. Abbiamo affrontato la comunicazione tra socket utilizzando oggetti locali. Adesso, vogliamo rendere il tutto distribuito, spiegando dei concetti che poi ci aiuteranno nella comprensione di Java RMI. Introduciamo uno strato software che rende trasparente l'invocazione remota dei metodi. Questo strato è composto da due parti principali:

- **stub**: oggetto che si trova sul client e rappresenta il server verso il client. Infatti presenta ed espone gli stessi metodi che vengono esposti sul server.
- **skeleton**: lo stub ha il compito principale di comunicare con questo strato, che si trova sul lato server. Esso effettua l'invocazione del metodo richiesto sull'oggetto server e poi comunica il valore restituito allo stub.

Sia lo stub che lo skeleton implementano un'interfaccia comune, detta **interfaccia remota**, dove vengono definiti i metodi che devono essere invocati da remoto. L'interfaccia remota deve essere usata a runtime. Il protocollo di comunicazione è abbastanza chiaro. Ogni invocazione di un metodo remoto sullo stub comporta la scrittura sul socket dell'oggetto da inviare, con i suoi eventuali parametri. È necessario che i metodi remoti dello stub lancino eccezioni poiché, usando la rete, è possibile incorrere in problemi di comunicazione. Rimane l'ultimo problema. Come fa il client a sapere dove si trova il server? In questo caso ci affidiamo al **well-know-service**, che è un servizio che fornisce una parziale trasparenza di locazione, fornendo il servizio di naming necessario.

JAVA REMOTE METHOD INVOCATION

Java RMI è una libreria di Java che permette di sviluppare applicazioni distribuite, effettuando comunicazione remota tra i programmi, scritti in Java. In generale, la proposta di Java RMI, quindi del team dell'informatico Jim Waldo, aveva come obiettivo quello di superare alcuni limiti imposti dal precedente CORBA. Java RMI deve assicurare la semplicità e l'integrazione nell'implementazione, sia per favorirne la rapida diffusione e sia per usare il linguaggio Java.
L'invocazione deve avvenire in maniera totalmente trasparente e l'integrazione con Java deve essere quanto più ampia possibile. In Java, un'applicazione viene eseguita all'interno di una **sandbox**, un ambiente specifico in cui le applicazioni vengono eseguite in sicurezza. Le componenti più importanti a garantire la sicurezza di Java sono:

- **sicurezza intrinseca**: Java è un linguaggio fortemente tipizzato, dove tutti i tipi vengono definiti a tempo di compilazione e ci sono pochissimi casting. Inoltre, la memoria viene gestita automaticamente da un garbage collector e, solo al momento dell'esecuzione. Inoltre, in Java non ci sono puntatori e, quindi, nessun accesso illegale.
- **ClassLoader**: si occupa di caricare la classe al momento dell'esecuzione e, inoltre, fa in modo che ogni classe abbia il proprio namespace identificativo, per prevenire interferenze.
- **Bytecode Verifier**: controlla che la classe sia conforme alle specifiche del linguaggio e che non ci siano violazioni di accesso o di overflow.
- **Security Manager**: definire i confini della sandbox, usando policy fornite dall'utente, applicandole ad operazioni potenzialmente pericolose.

Specifichiamo anche che Java RMI permette l'esistenza di vari tipi di invocazione, fornendo quella **unicast**, da un client ad un server, e quella **multicast**, cioè da un client a più server. L'oggetto server è attivato solo al momento dell'invocazione.

Adesso, però, scendiamo più nel dettaglio, descrivendo il meccanismo vero e proprio di Java RMI. Al suo interno, la descrizione dei servizi offerti da remoto è contenuta all'interno di un'**interfaccia remota**, che è una semplice interfaccia Java. Normalmente, Java RMI è contenuto in cinque package:

- `java.rmi` / `java.rmiserver`: meccanismo per invocazioni remote.
- `java.rmi.activation`: per oggetti attivabili, ossia quelli che forniscono riferimenti persistenti.
- `java.rmi.dgc`: fornire il Distributed Garbage Collector.
- `java.rmi.registry`: servizio di localizzazione.

Definendo l'interfaccia remota, essa è una semplice interfaccia Java, ma deve estendere l'interfaccia `Remote`, vuota. Ogni metodo al suo interno sarà remoto e deve dichiarare esplicitamente l'eccezione `RemoteException`. I suoi parametri vengono dichiarati tramite la propria. Possiamo avere due tipi di implementazione remota:

- riuso dell'implementazione remota, dove si deriva esplicitamente da `UnicastRemoteObject`, ereditando classi da `Remote Object` e `Remote Server`.
- classe di implementazione locale, dove la classe deriva il comportamento da un'altra classe locale e non si preoccupa dell'oggetto.

Nota bene che l'implementazione può contenere anche metodi fuori dall'interfaccia remota, ma che saranno accessibili solo localmente. Per poter invocare il metodo remoto di un oggetto remoto, l'oggetto client deve avere a disposizione il **riferimento remoto**. Può essere ottenuto come risultato di altre invocazioni di metodi oppure ottenuto tramite un meccanismo di **name-server**, che gestisce i riferimenti specificando un ID e metodi per ricerca, registrazione ed elenco. Un metodo remoto può dichiarare solo parametri o valori restituiti che implementino l'interfaccia `Serializable`. Un oggetto locale, passato da un'invocazione, viene passato per copia, cioè il suo contenuto viene copiato prima di essere serializzato. Dobbiamo fare attenzione, poiché quando passiamo parametri con più riferimenti allo stesso oggetto, ci troveremo due riferimenti distinti allo stesso server:

```
Date B = new Date();
Date C = B;
```

```
A.comunicaInizio(B);
A.comunicaInizio(C);
```

Con questo snippet di codice vediamo come vengono creati due oggetti diversi sulla stessa macchina remota, quando sulla macchina locale avevano lo stesso riferimento. Per questo, Java RMI mette a disposizione uno strumento, detto **integrità referenziale**. Quando, nella stessa invocazione, vengono passati più riferimenti allo stesso oggetto, allora anche sulla macchina remota, essi punteranno allo stesso oggetto:

```
Date B = new Date();
Date C = B;
A.comunicaInizio(B, C);
```

Il sistema di Java RMI è costituito da tre layer, o livelli fondamentali, che sono:

- **Stub/Skeleton Layer**:

  Si tratta del livello più alto di Java RMI ed è l'interfaccia tra l'applicazione (classi Java) e il resto del sistema. Questo layer comunica attraverso il livello inferiore tramite uno **stream di marshal**. Come già anticipato, questo livello è costituito da stub e skeleton. Lo stub deve:

  1. Iniziare la connessione con la macchina virtuale remota.
  2. Effettuare il marshalling verso uno stream di marshal.
  3. Effettuare l'unmarshalling dei valori restituiti.
  4. Attendere il risultato dell'invocazione.
  5. Restituire il valore all'oggetto client che l'ha richiesto.

  Lo skeleton, invece, gestisce l'invocazione per conto del server:

  1. Effettuare l'unmarshalling dei parametri di invocazione.
  2. Invocare il metodo di implementazione che si trova nella JVM.
  3. Effettuare il marshalling verso il metodo che l'ha invocato.

  Stub e Skeleton vengono creati dall'`rmic` che, a partire dall'implementazione dell'oggetto remoto, genera i file di stub e skeleton. Lo stub/skeleton, quindi, è presente ma totalmente trasparente.

- **Remote Reference Layer**:

Si occupa di interfacciare il livello di trasporto con quello di stub/skeleton. Esso specifica il comportamento dell'invocazione e la semantica del riferimenti. Specifica se possiamo avere invocazioni:

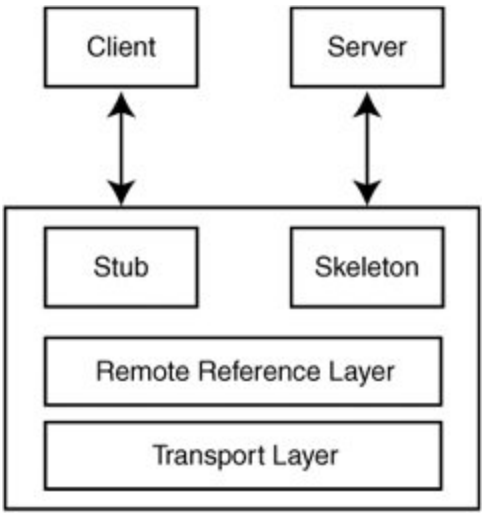
- unicast, da un client ad un server.
- multicast, da un client a tanti server replicati.
- oggetti attivabili, verso oggetti remoti persistenti.
- con riconnessione, tentare connessioni alternative se l'oggetto non risponde all'invocazione.

Questo layer comunica con lo stub/skeleton attraverso il metodo `invoke()`, richiamato dallo stub. Invece, esso comunica con il livello inferiore attraverso uno **stream di connessione**, cioè una connessione orientata a flussi.

▼ **Transport Layer**:

Ha i seguenti compiti:

- Stabilire una connessione verso macchine remote.
- Gestire e monitorare le connessioni.
- Gestire gli oggetti remoti nello spazio di indirizzamento remoto.
- Connettere le chiamate in entrata.
- Identificare il dispatcher (mediatore tra CPU e processi) e inoltrargli la connessione.



La gestione della memoria è un aspetto fondamentale, posto spetto sotto l'attenzione dei progettisti. Esistono due modi per gestire la memoria:

- farla gestire al programmatore, attraverso i metodi di allocazione e deallocazione, come `malloc()` e `free()`.
- farla gestire al sistema, usando il Garbage Collector.

Parlando in termini di Java RMI, esso fornisce un meccanismo di Garbage Collection, che le permette di integrarsi al meglio con Java. Si chiama **Distributed Garbage Collector** e tiene traccia di tutti i riferimenti, che siano **live** (attivi) oppure **weak** (deboli). Per estendere questo meccanismo anche agli oggetti remoti, bisogna introdurre il meccanismo di **lease**, dove ogni riferimento ha un tempo di vita specifico e, al termine di questo periodo, se non è stato invocato, diventa weak e viene eliminato. Java RMI permette il passaggio dei parametri o dei valori attraverso meccanismi di serializzazione che si vanno a scontrare con una caratteristica di Java, che carica le classi al momento dell'esecuzione. In questo modo, l'oggetto remoto potrebbe non conoscere la classe istanziata dall'oggetto passato. Questa problematica viene risolta con il **caricamento dinamico** delle classi. Quando si fa il marshalling per la trasmissione, gli oggetti vengono annotati con il **codebase** (URL) di un server dove è possibile reperire la definizione della classe. Una parte fondamentale del marshalling è la specializzazione del meccanismo di serializzazione effettuato da `ObjectOutputStream`, cioè modificando tre delle sue classi:

- `replaceObject()`: che definisce un metodo alternativo per serializzare un oggetto sullo stream.
- `enableReplaceObject()`: che restituisce un booleano specificando se specializzare il meccanismo di `replaceObject()`.
- `annotateClass()`: specifica il codebase per i caricamento dinamico.

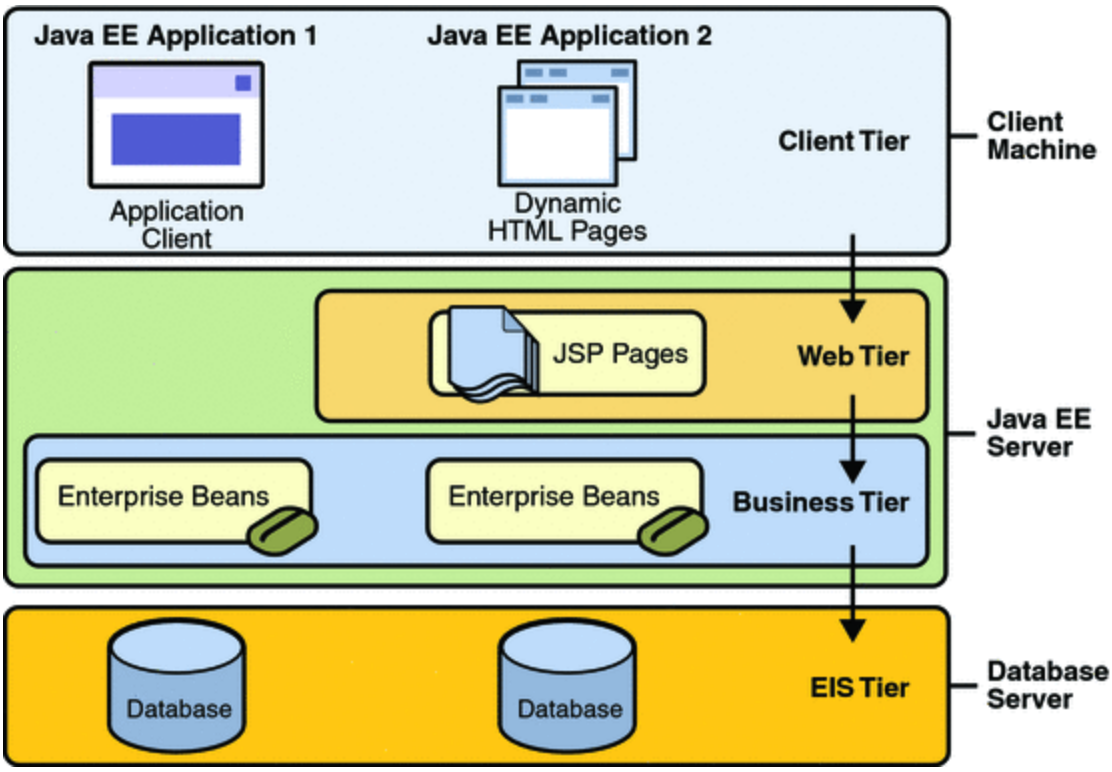
I termini marshalling e serializzazione sono quasi sinonimi. La serializzazione e il marshalling trasformano gli oggetti in byte, ma semanticamente hanno obiettivi diversi, poiché il marshalling tende a dare informazioni in più che descrivono quella determinata chiamata remota.
In Java RMI, dopo aver implementato l'interfaccia remota con i suoi metodi e le sue estensioni, dobbiamo implementare il server. Il server, normalmente, è istanza di una classe che implementa la nostra interfaccia remota e deriva da `UnicastRemoteObject`. Il costruttore deve essere scritto, anche se vuoto. Qui, poi, interviene `rmic`, che utilizza lo **stub compiler** per generare lo stub e lo skeleton. Ora, dobbiamo fare in modo che questo oggetto remoto sia accessibile al client. A tale scopo, Java RMI propone un servizio di naming, detto **rmiregistry**. Esso deve essere lanciato prima del server e, il client che vuole accedere ai servizi remoti, deve fare un'operazione di `lookup()` verso questo registro, in modo da ottenere il riferimento da usare durante l'invocazione. Ora, il server può essere eseguito. Bisogna anche scegliere una **policy**, ossia una politica di sicurezza, che serve a stabilire dei permessi che regolino azioni pericolose. Una volta lanciato il server, esso si registra al rmiregistry, usando metodi della classe `Naming`. Così, abbiamo ottenuto la nostra invocazione remota.

JAVA ENTERPRISE EDITION

Java EE, acronimo di Java Enterprise Edition, è un insieme di specifiche sviluppate in linguaggio Java. Fondamentalmente, le applicazioni Java EE sono applicazioni distribuite, transazionali e portabili, che garantiscono efficienza, sicurezza e affidabilità dal lato server. L'obiettivo principale di Java EE è quello di garantire API che accorcino tempi di sviluppo e investimenti, dimostrando di avere efficienza e scalabilità. Java EE utilizza un modello applicativo che simula un'architettura **multi-tier**, cioè un software di tipo client/server, le cui funzionalità sono suddivise in più livelli comunicanti tra loro. Nel caso di Java EE, esso è diviso in **componenti**, a seconda della loro funzione e la loro applicazione. Abbiamo varie categorie per le componenti di Java EE:

- **componenti per web service**: sia di tipo REST che di tipo SOAP, che sono due approcci per la trasmissione dei dati, specialmente nella comunicazione tra applicazioni, appunto, web.
- **componenti web**: legate soltanto alla visualizzazione delle pagine web, quindi servlet o JSP.
- **componenti enterprise**: blocchi specifici di Java EE come CDI, EJB, JMS, JSA.
- **componenti per interazione con database**: come JTA, JDBC e JPA.

Di solito, normalmente, i tre livelli di Java EE rappresentano un primo livello, che è un'interfaccia utente, quindi normalmente dello **client-machine**, poi abbiamo il **business-layer**, che contiene una serie di moduli per la generazione di contenuti dinamici (JSP o EJB), e infine il **data-layer**, dove i dati acceduti al livello precedente sono resi persistenti in un qualche database.



All'interno di Java EE, introduciamo la figura del **container**. Un container è l'interfaccia tra una componente e la funzionalità di basso livello specifica per la piattaforma. Per settare un container, dobbiamo specificarne le impostazioni per ciascun componente dell'applicazione Java EE. Si tratta di semplici specifiche riguardo la sicurezza, il naming, le transazioni e le connessioni remote. Nonostante ogni container differisca a seconda della piattaforma e del servizio richiesto, ci sono alcuni tipi di container ben definiti:

- **EJB Container**: gestisce l'esecuzione dei bean enterprise, che forniscono logica di business ad un'applicazione Java EE. Questo container gestisce il ciclo di vita della componente, invia le richieste ai componenti dell'applicazione e fornisce interfacce per i dati ambientali.
- **Web Container**: gestisce l'esecuzioni di componenti web.
- **Application Client Containe**r: gestisce l'esecuzione dell'application-client. Si tratta di un'applicazione autonoma, che viene eseguita sul client per gestire attività, librerie ecc..
- **Applet Container**: gestisce l'esecuzione degli applet, piccole applicazioni client, eseguite sulla JVM.

Introduciamo, adesso, anche le **annotazioni**. Le annotazioni sono una parte fondamentale di Java EE. Prima di questo, le abbiamo vista spesso nei nostri piccoli programmi Java, come magari un `@Override`, quando dobbiamo scrivere un metodo ereditato, oppure `@param`, che serve a dare specifiche riguardo uno specifico parametro. In realtà, in Java EE, così come in Java, esistono molti altri tipi di annotazioni. Le annotazioni semplificano il processo di sviluppo dell'applicazione consentendo agli sviluppatori di specificare all'interno della classe Java stessa come si comporta il componente dell'applicazione nel container, le richieste di inserimento delle dipendenze e così via. Varie annotazioni le andremo a specificare in seguito, ma è buono sapere che per annotare qualcosa basta anticipare l'annotazione con `@`.

CONTEXT AND DEPENDENCY INJECTION

In Java EE si è fatto sempre più avanti un design pattern molto particolare, chiamato IOC (Inversion of Control). Questo pattern si pone l'obiettivo di rendere le componenti sempre più indipendenti, in modo che sia lo stesso container a offrire servizi al nostro codice. Come lo fa? Configurando l'ambiente (**Context**) e risolvendo le dipendenze (**Dependency Injection**). Gli attori principali di Java EE sono di solito dei **Java Beans**, dei **POJO** con caratteristiche particolari, come getters e setters. POJO è acronimo di Plain Old Java Object, ossia una classe Java normalissima eseguita sulla JVM. Questi beans, detti **Managed Beans**, chiamati così proprio perché sono gestiti da CDI. Il concetto alla base di CDI è *loose coupling-strong typing*, cioè debole accoppiamento e forte tipizzazione. La prima dicitura afferma che bisogna tenere una bassa dipendenza tra le componenti, mentre la seconda è un paradigma tipico di Java, poiché Java stesso è un linguaggio fortemente tipizzato. Possiamo dire che sono due mondi diametralmente opposti, poiché la stessa tipizzazione è una dipendenza. L'unione di questi due mondi permette il corretto funzionamento di un'applicazione Java EE. Sappiamo che i MB sono affidati al server e offrono determinati servizi. Tra questi servizi abbiamo gli interceptors, i decorators e gli eventi, che poi approfondiremo. La DI, come detto, è un altro design pattern che permette di disaccoppiare componenti dipendenti. È molto facile dichiarare una dipendenza, infatti basterà aggiungere l'annotazione `@Inject` sopra di essa. In parole povere andiamo a iniettare una dipendenza di beans all'interno di un altro bean, in modo totalmente sicuro. Con questa annotazione viene definito un injection-point e può essere effettuato sia su attributi, metodi e costruttori. Ricordiamo che è importante inserire sempre l'annotazione `@PostConstruct` su un metodo che deve essere eseguito dopo l'iniezione di dipendenza e `@PreDestroy` su un metodo che verrà chiamato prima di rimuovere il bean dal container. Come detto, CDI non comprende soltanto la Dependency Injection, ma anche il Context, il contesto, l'ambiente. Mentre con un normale POJO si crea un'istanza con new e poi si affida

il resto al Garbage Collector, un bean è invece legato al proprio contesto e rimane quello fino a quando non viene distrutto dal container. Per questo, dobbiamo definire degli **scope**. In Java EE ce ne sono vari tipi:

- **Application Scope** (`@ApplicationScoped`): si estende per tutta la durata dell'applicazione.
- **Session Scope** (`@SessionScoped`): si estende per tutta la durata di un'ipotetica sessione HTTP.
- **Request Scope** (`@RequestScoped`): si estende per tutta la durata dell'invocazione del metodo.
- **Conversation Scope** (`@ConversationScoped`): si estende per tutta la durata di più invocazioni all'interno della stessa sessione e, in caso di conversazioni troppo lunghe, esse vengono divise in più fasi.
- **Dependent Pseudo-Scope** (`@Dependent`): si estende per tutta la durata del client e viene attivato ogni qualvolta si effettua un'iniezione.

Come abbiamo già anticipato, l'IOC offre determinati servizi per i beans. Uno di questi, sono gli **interceptors**. L'introduzione di un interceptor parte da un problema molto comune, cioè la manutenzione evolutiva. Quando si ha a che fare con software di grandi dimensioni, un committente può variare le sue richieste e andare a modificare le classi implementate richiederebbe ore di test in più oltre che la possibilità che si generino altri bug. Per questo introduciamo gli interceptor. Un interceptor è un modulo di controllo che incapsula della nuova logica di business e si frappone tra client e moduli già esistenti. Normalmente, un interceptor è una semplice classe Java, che contiene diversi metodi ma, soltanto uno, sarà annotato con `@AroundInvoke`, quindi sarà il metodo di intercettazione. Questo metodo deve anche implementare l'interfaccia `InvocationContext` e restituire un oggetto `Object`. Un metodo con `@AroundInvoke` non deve essere `static` o `final`. Quando vogliamo associare un interceptor al suo bean di destinazione dobbiamo usare l'**interceptor-binding**, un'annotazione separata, in cui viene indicato dove l'interceptor deve intervenire, la durata e altre informazioni.
Un altro dei servizi offerti da IOC sono i **decorators**. Partiamo dal fatto che gli interceptors non conoscono veramente il contesto che intercettano. Svolgono determinate operazioni prima del metodo di business, senza modificarne la logica. Bene, i decorators servono proprio a ovviare questo problema. Per utilizzare una

classe bean decorator, bisogna annotarla con `@Decorator`. Non basta perché, all'interno della classe decorata, avremo bisogno di un injection point delegato, che sia un campo, un parametro, aggiungendo `@Inject` e `@Delegate`. Non è possibile intercettare un decorator e gli interceptors hanno sempre la priorità. Ovviamente, IOC offre anche altri servizi, come le **alternative**, in cui un bean annotato con `@Alternative`, dichiara un'implementazione alternativa. Oppure ci sono i **producers**, che servono a iniettare dipendenze tra diversi MB. Basterà annotare la classe con `@Produces`. Complementare ai producers, ci sono i **disposers**, che permettono di realizzare una pulizia personalizzata di un oggetto che era stato restituito da un producer. Utilizziamo un disposer con l'annotazione `@Disposes`.
L'ultimo dei servizi offerti da IOC, sono gli **eventi**. Gli eventi consentono ai bean di comunicare senza alcuna dipendenza in fase di compilazione. Bisogna definire l'oggetto `Event`, che prende con sé un oggetto Java e possiamo chiamarlo in un qualsiasi punto del nostro bean tramite il metodo `fire()`. Per disambiguare gli eventi, dobbiamo usare i **qualifiers**. In generale, i qualifiers si usano per disambiguare due diverse istanze della stessa interfaccia. Un qualificatore va creato con l'annotazione `@Qualifier`, a cui vengono aggiunte altre annotazione come tipo e durata. Infine, si chiamerà questo qualificatore sopra l'evento usando un'annotazione. Per far combaciare eventi con i beans si utilizzano dei metodi osservatori, annotati con `@Observes`.

JAVA PERSISTENCE API

Java Persistence API, **JPA**, esiste per un motivo ben preciso. Si occupa della gestione della persistenza dei dati all'interno di un DBMS relazionale, presente in applicazioni Java EE. In Java, manipoliamo oggetti che sono istanze di classi. Ma se volessimo che alcuni oggetti siano persistenti? Introduciamo l'**ORM**, acronimo di Object-Relational Mapping, che si occupa di unire il mondo dei database a quello degli oggetti. Inoltre, quando parliamo di oggetti persistenti, è meglio introdurre il concetto di **entità**, piuttosto che di oggetto. Un oggetto vive solo nella memoria, un'entità vive per poco tempo nella memoria prima di essere salvato all'interno di un database. Un'entità è un POJO, quindi deve essere dichiarata e istanziata come una qualsiasi classe Java. Dobbiamo però prima annotare questa classe con `@Entity` e poi dobbiamo annotare una variabile con `@ID`, poiché in un database relazione ogni tabella deve avere il proprio identificativo. Inoltre l'entità deve avere il costruttore vuoto e, poi, è possibile implementare query, getters e setters o anche un costruttore pieno. Una volta definita l'entità, l'ORM delega ad un tool esterno il compito di creare corrispondenza tra tabelle e oggetti. In questo caso, il nostro tool è proprio JPA. JPA abbiamo detto che permette di mappare le entità all'interno di un database. Come lo fa? Utilizzando la classe `EntityManager`, che gestisce le entità, scrivendole o leggendole e permettendo sia operazioni CRUD (create, read, update, delete) che query più sofisticate. Quando creiamo un oggetto `EntityManager`, dobbiamo creare anche una Persistence Unit, abbreviata **PU**, che ci permette di connetterci al database. Il tipo di database e i vari parametri di connessione vengono definiti all'interno di un file di configurazione, detto `persistence.xml`. Una volta create l'`EntityManager`, si rende persistente un oggetto chiamando il metodo `persist()`. In JPA, la maggior parte delle entità deve far riferimento e avere relazioni con altre entità, cioè l'equivalente delle **chiavi esterne** all'interno di un database. Per questo è possibile mappare queste associazioni:

- `@OneToOne`: definisce una relazione unidirezionale o bidirezionale, cioè l'oggetto A punta all'oggetto B oppure entrambi gli oggetti si riferiscono all'altro.

- `@OneToMany`: definisce una relazione bidirezionale in cui oggetto abbia riferimenti ad una serie di oggetti.
- `@ManyToMany`: definisce una relazione bidirezionale in cui entrambi gli oggetti fanno riferimento a più istanze di essi (un cantante ha fatto molti album e molti album sono stati scritti da quel cantante).

Oltre `persist()`, ci sono anche altri metodi che permettono di manipolare le nostre entità, come `remove()`, che appunto rimuove, `merge()`, per modificare, `refresh()`, sovrascrivere modifiche, `detach()`, rimuovere l'entità dal contesto persistente e `all()`, che effettua tutte queste operazioni a cascata. Introduciamo, adesso, **JPQL**, acronimo di Java Persistence Query Language, in linguaggio di query che si basa sul famoso linguaggio per l'interrogazione del database, l'SQL. SQL restituisce solo righe o colonne, mentre JPQL restituisce un'entità o una lista di entità. JPA permette l'utilizzo di cinque tipi di query JPQL:

- **Dynamic Queries**: stringhe di query, specificate dinamicamente a runtime. La query viene trasformata in SQL ogni qualvolta essa viene eseguita.
- **Named Queries**: stringhe di query, statiche, che non possono essere modificate dato che vengono trasformate in SQL a tempo di deployment. Una Named Query si crea con l'annotazione `@NamedQuery` e prendere come parametri un nome con la stringa di query.
- **Criteria API**: sono particolari query, di nuovo tipo, che supportano tutti i meccanismi di JPQL con una sintassi orientata ad oggetti. Ogni parola chiave di SQL diventa un metodo che può essere chiamato da un oggetto di tipo `CriteriaBuilder`.
- **Native Queries**: esse utilizzano solo istruzioni SQL native (`SELECT`, `UPDATE`, `DELETE`) e restituiscono un'esecuzione SQL. Queste query vengono utilizzate quando è necessario convertire automaticamente il risultato in un'entità.
- **Stored Procedure Queries**: le query descritte fino ad ora inviano una query dall'applicazione al dabatase, la eseguono e restituiscono un risultato. Le SPQ sono eseguite direttamente sul database. Nonostante le migliori prestazioni e i migliori controlli di sicurezza, sono query poco utilizzate, perché non portabili.

Per usare queste query possiamo utilizzare tre tipi di interfacce di `EntityManager`, che sono `Query`, utilizzata quando bisogna restituire un oggetto, `TypedQuery`,

quando deve essere restituito un risultato tipizzato e `StoredProcedureQuery` per le SPQ. Una query può opzionalmente avere dei parametri, che saranno settati con `setParameter()` e può restituire due metodi a seconda del risultato, `getResultList()`, per restituire un elenco, `getSingleResult()`, per restituire un singolo risultato.

ENTERPRISE JAVA BEANS

Abbiamo visto fino ad ora il layer di persistenza di un'applicazione Java EE, ma dove viene implementata la logica di business, cioè quella logica di elaborazione che rende operativa un'applicazione. A occuparsi di questo sono gli **EJB**, acronimo di Enterprise Java Beans. Gli EJB sono delle componenti lato server che si occupano di logica di business, sicurezza e transazioni, interagendo liberamento anche con altre componenti. Un EJB è un POJO, di cui verrà fatto il deploy in un container. Java EE fornisce diversi tipi di EJB:

- **Stateless**: il bean non ha uno stato e i metodi e l'istanza possono essere utilizzati per qualsiasi client. Qualsiasi operazione deve essere conclusa con una singola invocazione di metodo.
- **Stateful**: il bean ha uno stato che deve essere mantenuto tra i metodi, utile quando c'è da eseguire diversi passaggi e quando è necessario mantenere uno stato. Questo bean non può essere riutilizzato da altri client
- **Singleton**: un singolo bean condiviso tra i client, che supporta l'accesso concorrente. Il container fa in modo che ne esista una sola istanza e che non si debba specificare all'interno del codice. Il costruttore deve essere privato

Un EJB è strettamente legato al suo container, poiché esso fornisce tutta una serie di servizi fondamentali, come comunicazione remota, dependency injection, pooling (condivisione tra più client), ciclo di vita, messaggi asincroni, transazioni, interceptor ecc. Il container, poi, è quello che interagisce con gli altri container all'interno di un'applicazione Java EE. Come abbiamo detto, un EJB è una classe Java e va intesa come tale. Bisogna soltanto aggiungere un'annotazione, `@Stateless`, `@Singleton`, `@Stateful`, che ne specifica il tipo. Infine, sarà necessario implementare un'interfaccia di business, che sia remota o locale (`@Remote` `@Local`). In un EJB, né il client, né il bean sono responsabili di creazione o distruzione, ma è il container ad occuparsene. Per quanto concerne i bean Stateless e Singleton, essi non mantengono lo stato e hanno un solo processo di creazione e distruzione. Nel Singleton è necessario annotare anche `@Startup`, per far partire il ciclo di vita. Il

bean Stateful ha un ciclo di vita diverso poiché, dovendo mantenere uno stato, passa per uno stato di passivazione (in cui rimane inattivo per molto tempo), annotato con `@PrePassivate` e attivazione (quando viene riattivato) annotato con `@PostActivate`. Per quanto concerne la sicurezza degli EJB, il container si occupa delle autorizzazioni. Possiamo avere un'**autorizzazione dichiarativa**, che viene definita con le annotazioni, o l'**autorizzazione programmatica**, usando metodi della classe `SessionContext`.
Come abbiamo già ampiamente detto, EJB si occupano anche delle **transazioni**. Le transazioni, banalmente, permettono di avere dati coerenti che possono essere processati in maniera affidabile, il tutto seguendo la politica di **ACID** (Atomicity, Consistency, Isolation, Durability). Una delle API più importanti in Java EE per gestire le transazioni è **JTA**, acronimo di Java Transaction API. Normalmente, una transazione è composta da un **Transaction Manager**, che crea e gestisce una transazione globale e il **Resource Manager**, che interagisce con il TM e media con una delle risorse preponderate, come un database o delle code di messaggi. Gli EJB, per natura, supportano le transazioni e ad occuparsene è JTA, che media con il container.

JAVA MESSAGE SERVICE

Fino ad ora sono state affrontate comunicazioni sincrone tra componenti. Quando invece parliamo di messaggistica, intendiamo comunicazione **asincrone**. A permettere lo scambio asincrono tra componenti è uno strato software, detto **MOM**, acronimo di Message-Oriented Middleware. Si tratta sostanzialmente di un buffer, in cui si producono e consumano messaggi. In Java EE, l'API a occuparsi di messagistica asincrona è **JMS**, acronimo di Java Message Service. Quando viene inviato un messaggio, il mittente è chiamato **Producer**, il percorso in cui è archiviato è detto **destinazione** e il destinatario è detto **Consumer**. I messaggi vengono memorizzati e inviati all'interno di uno strato software, detto **Provider**. JMS ha una serie di interfacce che le permettono di connettersi ad un Provider, di scrivere, inviare o ricevere un messaggio. Un'architettura di messaggistica è composta anche da un **Client**, che è una componente Java che consuma il messaggio all'interno del provider. Esistono poi i **Messaggi**, trasformati in oggetti, e gli **oggetti amministrati**, che forniscono connessioni e destinazioni al client, tramite ricerche di naming o dependency injection. In JMS abbiamo due tipi di destinazione:

- **Modello Point-to-Point (P2P)**: in cui la destinazione è una **coda**, in cui si accumulano i messaggi. Si viaggia da un singolo produttore ad un singolo consumatore
- **Modello Publish-Subscribe (Pub-Sub)**: in cui la destinazione è detta topic e in cui un client pubblica un messaggio su un **topic** e chi vuole ricevere il messaggio di iscrive a questo topic. Si viaggia da un singolo produttore a più consumatori. I produttori qui sono detti publishers, mentre i consumatori subscribers.

  In un contesto EJB, gli **MDB** (Message-Driven Beans), sono consumatori di messaggi asincroni, eseguiti all'interno di un container EJB, che si occupa di gestire i soliti compiti transazionali e di sicurezza. Gli MDB ascoltano solo una destinazione che sia una coda o un topic.
  Un elemento molto importante all'interno di un sistema di messaggistica, è la

`ConnectionFactory`. Si tratta di oggetti amministrati che permettono la connessione ad un Provider. Per usare una `ConnectionFactory` è necessario creare un oggetto `Context` e fare un `lookup()`. Lo stesso `lookup()` deve essere effettuato per la creazione di una `Destination`. Infine si usa il metodo `createConnection()` per creare la connessione al Provider. Un messaggio, normalmente, contiene un **header** (identificazione e instradamento), alcune **proprietà** (filtering) e un **corpo** (contiene il messaggio). L'applicazione che invia i messaggi usa il metodo `createProducer()` per creare un `JMSProducer`. Tutto questo è fornito dall'interfaccia `JMSContext`. JMS definisce anche alcuni meccanismi di affidabilità, per far sì che il messaggio venga consegnato correttamente:

- **filtering dei messaggi**: crea dei meccanismi per permettere solo ad alcuni di ricevere il messaggio. Possiamo inserire queste opzioni di filtraggio o all'interno dell'header o all'interno delle proprietà. Un'opzione di filtraggio è detto **message-selector** ed è una stringa.
- **time-to-live**: viene settato un determinato quanto di tempo e il Provider si occupa di eliminare i messaggio quando diventano obsoleti, quindi quando scade il tempo.
- **gestione della persistenza**: JMS supporta la spedizione di messaggi sia persistenti che non persistenti. Inviare messaggi persistenti è più affidabile, poiché non c'è rischio di far fallire l'invio. Per specificarlo, usiamo `setDeliveryMode()`.
- **controllo degli acknowledgement**: l'ACK è un segnale che notifica il corretto ricevimento del messaggio. È possibile farlo con JMS specificando alcuni metodi di `JMSContext`, come `AUTO_ACKNOWLEDGE` (automatico), `CLIENT_ACKNOWLEDGE` (effettuato dal client) e `DUPS_OK_ACKNOWLEDGE` (in caso di fallimento, crea duplicati).
- **durable-consumers**: si crea un Consumer durevole che tiene i messaggi in topic fino a quando ogni iscritto non li riceve.

WEB SERVICES

Banalmente, quando parliamo di un web service, ci riferiamo a qualcosa sul web che offre un determinato servizio. Usando la parola **SOA**, ci riferiamo alla Service-Oriented Architecture, cioè un'architettura che supporta servizi web e che garantisce interoperabilità tra le componenti. Il concetto alla base di un Web Service è il *loosely coupled*, debolmente accoppiati, un concetto che abbiamo già abbondantemente approfondito. Oltre che debolmente accoppiato, un Web Service deve anche mantenere un certo atteggiamento neutrale rispetto alla tecnologia, vale a dire che deve essere sempre disponibile su ogni piattaforma. Inoltre deve garantire la trasparenza di locazione, ossia la definizione e la locazione devono essere accessibili tramite registri pubblici. In parole povere, un Web Service è un sistema progettato per supportare le interazioni interoperabili tra computer su una rete. L'interazione avviene tramite un protocollo, che poi andremo ad approfondire, tramite una richiesta HTTP con serializzazione in XML. Attenzione, perché non bisogna mai confondere un Web Service con un servizio accessibile via web, poiché questi ultimi utilizzano un'interfaccia universale, un browser, ma non necessariamente sono Web Services e, quindi, offrono quei determinati servizi. Per questo è necessario rivedere il meccanismo **SAAS**, ossia Software as a Service, cioè un modello di software distribuito in cui un Producer sviluppa e gestisce un'applicazione web, mettendola a disposizione per i propri clienti. Un Web Service ha alcuni standard ben definiti. Innanzitutto diciamo che l'**XML** assicura la base su cui vengono definiti dei protocolli ben precisi. Oltre ad HTTP, il più usato, possiamo usare anche SMTP (email) o anche JMS.
Introduciamo adesso il **WSDL**, acronimo di Web Service Description Language. Si tratta formalmente di un file in XML, in cui viene descritto il Web Service, attraverso dei tag, come definizioni, tipi di messaggi e porte. Il protocollo più comune per lo scambio dei messaggi è detto **SOAP**, acronimo di Simple Object Access Protocol. Può operare su diversi protocolli di rete, ma utilizza soprattutto HTTP, ed è basato su XML, cioè con i messaggi strutturati secondo uno schema header-body. SOAP richiede anche l'attributo `Envelope`, che definisce il messaggio e il suo namespace. Parliamo ora dell'**UDDI**, acronimo di Universal Description

Discovery and Integration. Si tratta di un registro, un database indicizzato, per la locazione di un servizio. Anche UDDI si basa su XML e può essere scoperto e scaricato. È ormai in disuso. Per i Web Services, l'implementazione di riferimento è **Metro**, un OS che riguarda il mondo Java. Da qui, possiamo poi seguire due linee guida:

- **contract-first**: dove si parte dalla definizione in WSDL e si finisce con la generazione automatica delle classi di quest'ultimo, attraverso `wsimport`.
- **program-first**: dove si parte dall'implementazione in Java e si finisce col generare automaticamente il WSDL con `wsgen`.

Normalmente, attraverso un'annotazione, un qualsiasi POJO può diventare un Web Service. Basta aggiungere `@WebService`. Spesso si tende a rendere un EJB un Web Service, per avere transazioni e sicurezza in automatico, flessibilità ed efficienza e la gestione degli interceptor. Con un servizio, basta definire l'XML, ma in Java dobbiamo tradurre in oggetti. Lo facciamo attraverso due tipi di annotazioni:

- **annotazioni di mapping WSDL**: modificano il WSDL e permettono di personalizzare i metodi, le classi e le interfacce.
- **annotazioni di binding SOAP**: consentono il collegamento fra un'entità e il suo valore. Un binding SOAP, che sia un documento o una chiama RPC, deve selezionare due formati di serializzazione, che sia letterale (secondo schema XML) oppure codificato (oggetti).

Mettendo insieme queste annotazioni, otterremo un Web Service in Java ottimale. Per gestire le eccezioni si usa un **errore SOAP** all'interno del messaggio, in modo da convertire le eccezioni. Ogni Web Service ha un contesto, a cui è possibile accedervi tramite `@Resource`.

CLOUD COMPUTING

Partiamo da un concetto base, la **scalabilità**. Possiamo definire un sistema scalabile quando è capace di adattarsi ai forti picchi di carico d'utenza. In maniera più generale, diciamo che un sistema è scalabile quando è capace di aumentare e ridurre le proprie prestazioni in base alle necessità. Il calcolo distribuito ha introdotto nuove tipologie di calcolo:

- **High Performance Computing (HPC)**: un insieme di tecnologie di calcolo, utilizzate da computer cluster (insieme di computer connessi tramite una rete telematica), che servono a fornire altissime prestazioni, utilizzate soprattutto nel calcolo scientifico.
- **High Throghput Computing (HTC)**: un insieme di tecnologie di calcolo che servono a fornire altissime prestazioni per lunghi periodi di tempo.
- **Computation and Data Grid**: un insieme di tecnologie di calcolo che servono a manipolare grosse quantità di dati, utilizzando un grande numero di risorse.

In base a queste tipologie di calcolo, nascono altre classi:

- **calcolo monolitico**: un calcolo centralizzato con un unico nodo elaborativo.
- **calcolo parallelo**: un calcolo strettamente accoppiato con l'esecuzione di più programmi su più microprocessori.
- **calcolo distribuito**: un calcolo debolmente accoppiato con nodi autonomi collegati attraverso una rete.
- **calcolo cloud**: un calcolo che viene servito in base alle necessità dell'utente.

Ed è qui che entriamo nel merito. C'è una differenza fondamentale tra un cloud e un sistema distribuito. Un cloud è in grado di scalare all'infinito, adattandosi ad ogni tipologia di utente. Il cloud computing è un modello di riferimento che eroga servizi su richiesta da un fornitore ad un cliente, a partire da una serie di risorse preesistenti disponibili sotto forma di un'architettura distribuita. All'interno del cloud computing possiamo individuare vari tipi di servizi forniti:

- **IAAS (Infrastructure as a Service)**: oltre alla risorse virtuali da remoto, vengono messe a disposizione anche le risorse hardware, come sistemi di memoria e archivi.
- **PAAS (Platform as a Service)**: invece che uno o più programmi, viene resa disponibile da remoto una piattaforma software che li racchiude.
- **SAAS (Software as a Service)**: ne abbiamo già parlato e consiste nell'utilizzo di programmi su server da remoto, fuori dalla macchina fisica, all'interno di un server web. Questo servizio condivide molti aspetti con un termine ormai in disuso, ASP, acronimo di Application Service Provider.

Le motivazioni al cloud computing sono molteplici. Un cloud fornisce spazi specificatamente designati per il calcolo, è possibile effettuare la condivisione tra molti più utenti, ridurre i costi, avere ambienti scalabili all'infinito e modelli di costo e business ritagliati per le necessità. Molto di questo lavoro lo fa l'**elasticity**, una caratteristica distintiva dei cloud computing, ed è una sorta di sovradimensione alle risorse. In un cloud computing i costi vengono stabiliti in base alla tecnica *pay-as-you-go* (quanto si utilizza, tanto si paga), utilizzando una tecnica di tipo **OPEX** (Operational Expenses), con costi variabili che dipendono dal numero di utenti, a differenza della tecnica CAPEX (Capital Expenses), che ha costi fissi con alti investimenti.
Il cloud computing presenta anche alcune problematiche. Con il cloud computing e, quindi, con il salvataggio dei dati su un server, è possibile incorrere in problemi relativi alla sicurezza informatica e alla privacy. Inoltre l'utente si trova limitato su quei servizi che non sono gestiti direttamente dal cloud ma da agenti esterni, come per esempio la memorizzazione dei dati all'interno di archivi privati e questo non solo non garantisce una continuità del servizio ma filtra in maniera abbastanza netta gli accessi, aumentando i divari tra le tipologie di utente.

MICROSERVIZI E SERVERLESS COMPUTING

Partiamo da un concetto fondamentale. Il **Big Software** è un software di grandi dimensioni, progettato da una grande azienda, uno dei banchi di prova più importanti nel design e nello sviluppo tecnologico. Bene, il Big Software è morto. È morto perché le aziende non hanno più l'esigenza di usarli e, soprattutto, perché non vengono più prodotti da loro. Dalla morte del Big Software nasce l'idea di un software small. L'idea di base è che ci siano diversi servizi che sono sviluppati in maniera indipendente, a differenza quindi di un Web Service. Questi **microservizi** sono indipendentemente sviluppati e mantenuti senza creare dipendenze, comunicando attraverso meccanismi estranei alla piattaforma e senza alcuna infrastruttura centralizzata. I microservizi hanno molti vantaggi. È possibile, infatti, fare un deployment continuo, senza aver bisogno di sincronizzare tutte le modifiche. È possibile il loro delivery (consegna) sul container. La gestione dell'applicazione è in mano al cliente e non al fornitore. L'idea è quella di sviluppare l'applicazione con tanti piccoli servizi, che comunicano tramite HTTP. Con i microservizi aumenta anche la produttività, infatti aumentano i costi. Ci sono alcune caratteristiche ben precise dei microservizi:

- **autonomia**: ogni servizio può essere sviluppato, distribuito, eseguito o modificato indipendentemente da un altro. Inoltre seguono il paradigma di smart endpoints-dumb pipes, vale a dire un'implementazione molto diretta e un'infrastruttura stupida, elementare.
- **specializzati**: ciascun servizio viene progettato per una serie di capacità e si concentra sulla risoluzione specifica di un problema. A sua volta un servizio può essere integrato con del codice sorgente che lo renda più complesso. Inoltre, ogni servizio è progettato per evolversi e resistere ai malfunzionamenti parziali.

Fino a qua, abbiamo visto come i microservizi risultino una specie di manna dal cielo. Invece, se scendiamo nel dettaglio, ci accorgiamo che ci sono anche diverse problematiche, due principalmente: la complessità e la produttività. Ci sono problemi sia a livello di compilazione che a livello di testing, poiché serviranno molte compilazioni e molti test per verificare che ogni servizio funzioni correttamente. Inoltre non esiste una soluzione vera e propria al debugging. Introduciamo, adesso, il concetto di **serverless computing**, cioè un'interpretazione cloud dei microservizi. Lo facciamo introducendo un nuovo servizio di cloud computing, ossia il **FAAS**, acronimo di Function as a Service. Esso ti permette di eseguire codice di risposta a eventi senza la complessità dell'infrastruttura tipicamente associata ai microservizi. Facciamo attenzione, perché dire FAAS e serverless computing non è proprio la stessa cosa. Un'architettura serverless si concentra su quei servizi dove la configurazione è completamente nascosta all'utente, mentre FAAS si concentra propriamente sugli eventi, come abbiamo detto, usando un **gateway** che smista i compiti e che lavora su un database indipendente. La fusione tra un'architettura serverless e FAAS rende l'idea di una corretta implementazione di un servizio cloud per i microservizi. Quest'architettura non è adatta per le lunghe procedure e può verificarsi un **cold-start**, in cui una funzione inattiva viene deattivata e passivata e, al momento dell'utilizzo, avrò un ritardo.