**Appunti PD**

Le principali leggi che regolano la programmazione distribuita sono:

* Legge di Sarnoff
  + Il valore di una rete di broadcast è direttamente proporzionale al numero di utenti: V=a\*N
* Legge di Metcalfe
  + Il valore di una rete è proporzionale al quadrato del numero dei suoi utenti
* Legge di Reed
  + Il valore di una rete cresce esponenzialmente con la dimensione della rete

**Reference Model (Open Distributed Processing)**

Si definisce **reference model** l’astrazione utilizzata da produttori, progettisti e sviluppatori. Questa ha il compito proprio di permettere la comunicazione tra queste varie figure, astraendosi dai campi specifici di ogni area tecnica. Difatti la trasparenza è una delle caratteristiche più importanti di un sistema distribuito.

Questo lo rende indipendente da ogni sorta di specifica di implementazione. Dev’essere però sufficientemente informativo da descrivere i meccanismi e i metodi da implementare

Per ottenere un’astrazione **standard**, si utilizza un modello fornito dall’ ISO/IEC. Basato sul modello ISO/OSI, estendendolo.

**Caratteristiche**

Il reference model è:

* Remoto
  + Componenti remote, ovvero risorse collocate in locale o anche su macchine diverse
* Concorrente
  + Sistema concorrente, ovvero, insieme di macchine che provano ad eseguire contemporaneamente, in questi casi non si possono sfruttare sistemi e meccanismi per accesso a regione critica (semafori, operazioni atomiche ecc)
* Assenza di uno stato totale
  + Data dall’alta dinamicità (ed eterogeneità) del sistema totale
* Fallimenti Parziali (partial failures)
  + Ogni componente può fallire indipendentemente, in questo caso il fallimento non porta all’arresto del sistema
  + Il sistema continua a fornire le sue funzionalità (anche se non tutte)
* Eterogeneità
  + Il sistema è eterogeneo nel senso di macchine che utilizzano software, hardware, protocolli, linguaggi differenti.
* Autonome
  + Non esistono punti di controllo che coordinano o gestiscono il sistema
  + Proprio per questo motivo è necessaria collaborazione
* Evoluzione
  + Assecondare l’ambiente tecnologico e aziendale con flessibilità
* Mobilità
  + Non solo di utenti ma anche di dati, nodi e risorse
  + Il sistema può cambiare, non bisogna dare per scontato nulla in un sistema distribuito

**Requisiti non funzionali**

Requisiti non direttamente collegati alle funzionalità, sono indice globale della qualità del sistema

Possono essere:

* Aperti
  + Uso di interfacce e standard noti e riconosciuti
    - Utili per facilitare l’interoperabilità e l’evoluzione
* Integrati
  + Risorse differenti vengono utilizzate senza strumenti ad-hoc
* Flessibili
  + Utilizzo di sistemi legacy (così da garantire l’evoluzione del sistema)
  + Gestire modifiche durante l’esecuzione
* Modulari
  + Ogni componente autonoma (ma leggermente interdipendente verso il sistema) deve poter essere aggiunta, rimossa o modificata
* Supportare la federazione dei sistemi
  + Unione di diversi sistemi (amministrativamente e architetturalmente)
    - Per fornire servizi in maniera congiunta
* Facilmente scalabili
  + Rendere facile l’aumento di gestione di carico
  + Aumentare il throghput aggiungendo risorse senza modificare l’architettura
* Trasparenza
  + Mascherando dettagli e differenze nell’architettura del sistema

**TRASPARENZA**

Il requisito non funzionale più importate è:

* Trasparenza
  + Il sistema distribuito apparirà come un’unica entità all’utente
  + PRO:
    - Maggiore produttività (per via dell’astrazione del modello)
    - Alto riuso di applicazioni riutilizzate

La trasparenza può essere categorizzata in **tre** livelli:

1. **Libello base**
   1. Accesso
   2. Locazione
2. **Livello di funzionalità**
   1. Migrazione
   2. Replica
      1. Backup funzionalità o dati
   3. Persistenza
   4. Transazioni
3. **Livello di Efficienza**
   1. Scalabilità
   2. Prestazioni
   3. Malfunzionamenti

Graficamente i livelli di trasparenza sono rappresentati così

Immagine che contiene schizzo, testo, diagramma, disegno

Descrizione generata automaticamente

**LIVELLO BASE**

**Trasparenza di accesso**:

* Nasconde le differenze nella rappresentazione dei dati e nell’invocazione per l’interoperabilità
  + Rende il sistema astratto (nascondendo dati e struttura) a chi accede
  + L’utente o chiunque acceda passerà per la stessa interfaccia sia da remoto che da locale
    - Un oggetto, quindi, può anche essere spostato, l’importante è che sia ancora collegato al sistema
* Default dei sistemi distribuiti

**Trasparenza di locazione**:

* Impedisce di utilizzare informazioni circa la posizione di un determinato componente
  + Si accede ad una determinata risorsa attraverso un servizio di naming
* Fondamentale per un sistema distribuito
* Default dei sistemi distribuiti

**LIVELLO DI FUNZIONALITÀ**

**Trasparenza di migrazione:**

* Permette al sistema di migrare oggetti da un nodo all’altro, e grazie alla trasparenza di **locazione** e a quella di **accesso**, un utente potrà sempre raggiungere quell’oggetto. Nessun utente saprà di questa migrazione
  + Utile per ottimizzare le prestazioni o per risolvere malfunzionamenti o riconfigurazioni

**Trasparenza di replica**:

* Un oggetto viene duplicato (repliche) in più copie posizionate in altri nodi del sistema
  + È necessario però mantenere le repliche coerenti tra loro
* Usato per le prestazioni
  + Ponendo i servizi dove possono essere facilmente raggiungibili
* Basato su trasparenza di accesso e locazione, poiché l’utente non saprà né dov’è erogato l’oggetto n’è quale degli oggetti delle repliche verrà erogato

**Trasparenza di persistenza:**

* Il sistema rende *persistente* (messo in memoria secondaria), senza che l’utente se ne occupi
  + È un meccanismo che attiva e deattiva per risparmiare risorse, evitando che un oggetto poco utilizzato sia in memoria principale, o viceversa, può portare un elemento molto utilizzato dalla memoria secondaria in memoria principale. Questo può essere identificato dagli handle
* Basato su trasparenza di locazione
  + Poiché l’utente deve poter accedere all’oggetto senza sapere dov’è

**Trasparenza di transazioni**

* Implicitamente concorrente
* Transazioni garantite dal sistema
  + Offre coerenza del comportamento in presenza di accessi concorrenti
* Semplificazione notevole offerta dal sistema
  + Le prestazioni sono possibilmente un problema (sovraccarico di richieste)

**LIVELLO DI EFFICIENZA**

**Trasparenza di scalabilità**

* Il sistema è in grado di servire carichi di lavoro crescenti senza modificare architettura ed organizzazione
  + Il programmatore, o l’utente, non dovranno modificare l’architettura o sapere di queste migrazioni e repliche
* Si basa su Trasparenza di migrazione e di replica
  + Il carico può essere quindi fatto migrare e replicare così da poter essere operato e suddiviso sfruttando in modo ottimale le risorse del sistema, sfruttando quelle che non sono sotto carico o che sono sotto carico in forma minore

**Trasparenza di prestazioni**

* Il progettista/sviluppatori del sistema ottengono alte prestazioni pur non conoscendo i meccanismi utilizzati
  + Bilanciamento del carico (migrazione/replica)
  + Minimizzazione della latenza (migrazione/replica)
  + Ottimizzazione delle risorse (persistenza)

Trasparenza ai malfunzionamenti

* In presenza di malfunzionamenti il sistema è in grado di fornire servizi (anche se in maniera parziale)
* Si basa su:
  + Replica
    - Così da poter continuare ad operare sui dati del componente malfunzionante
  + Transazioni
    - Potenzialità di eseguire un rollback (tornare a prima dell’operazione non complete, che andranno rieseguite con i replicati)

**Oggetti Distribuiti**

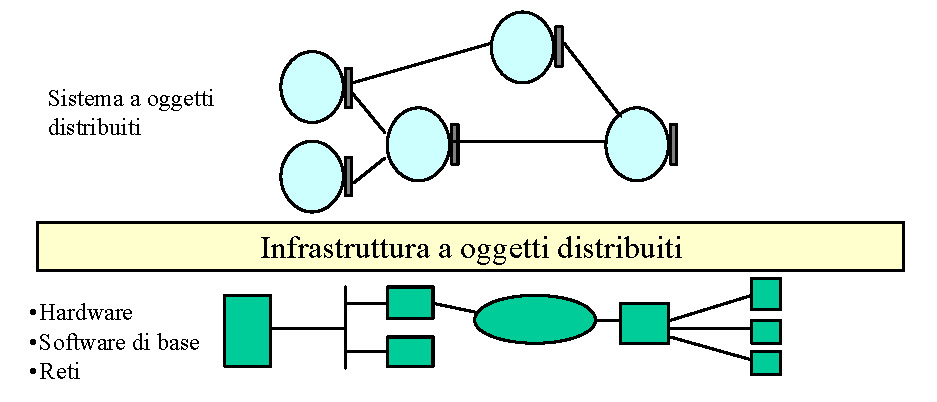
Definiamo innanzitutto gli **oggetti distribuiti,** questi provengono dalla confluenza di due aree:

* Sistemi distribuiti
  + Ovvero l’obiettivo di realizzare un unico sistema basato sulle risorse in rete
* Programmazione ad oggetti
  + Cerca di minimizzare la difficoltà di progettare, sviluppare, testare e manutenere un sistema

Il loro obiettivo è garantire:

* Scalabilità
* Estendibilità
* Facilità di gestione
* Tolleranza ai malfunzionamenti
* Interoperabilità

**Middleware**

Per un sistema distribuito è necessario integrare sistemi con architetture eterogenee, questa integrazione avviene attraverso il **middleware**, che si contrappone tra i sistemi ad oggetti distribuiti e hardware, software di base e reti:

Il middleware è utile per i programmatori, esonerandoli dal conoscere tutta l’architettura sottostante

Il middleware può essere suddiviso in:

* Middleware di infrastuttura
  + Permette la comunicazione tra SO diversi e gestione delle concorrenza (Es. JVM)
* Di distribuzione
  + Automatizza compiti comuni per la comunicazione come per esempio:
    - Marshalling
      * Suddivide i dati in più messaggi
    - Multiplexing
    - Gestione delle invocazioni
    - Riconoscimento e gestione malfunzionamenti
* Per servizi comuni
  + Persistenza, transazioni, sicurezza, ecc…

Il middleware fornisce un’astrazione che ha come vantaggio:

* Riutilizzare le soluzioni adottate
* Facilita l’accesso alle risorse (tramite i tre livelli di middleware)
* Rendono lo sviluppo più facile

**Remote Procedure Call**

Fu la prima tecnologia a fornire la traduzione dei tipi di dato a livello applicazione

* Forniva dati attraverso uno stream di byte su socket in modo che potessero essere codificati e decodificati (marshalling e unmarshalling), superando la differenza di rappresentazione di interi e stringe (ASCII e EBCDIC)

Permette al programmatore di usare un paradigma noto: sincronia della invocazione forzata

Utilizzo di client stub e server stub per forzare mashalling, rappresentazione dei dati e sincronia, creati automaticamente sulla base di IDL (Interface Definition Language)

CONTRO del RPC:

* Paradigma procedurale
  + Mentre il mondo si spinge verso una programmazione ad oggetti
* I tipi di dato sono solo elementari
  + Niente struct o puntatori
* Formato dei dati diverso
* Mancanza della gestione delle eccezioni

**Invocazioni Sincrone (RPC)**

Paradigma delle invocazioni di funzioni (metodi): la funzione chiamante viene vloccata finché il metodo non ha terminato

È la semantica della programmazione non-concorrente (più semplice per sviluppo)

Al contrario, le **invocazioni asincrone**:

* La funzione chiamante continua concorrentemente insieme alla funzione chiamata

RPC per le sue problematiche va superato: **Middleware a oggetti distribuiti**

**Middleware a oggetti distribuiti**

Risulta un’evoluzione di sistemi distribuiti complessi, necessario per la progettazione di sistemi più complessi

**Middleware CORBA**

È composto da:

* ORB (Object Request Broker)
* Servizi di base (Common Object Service)
* Servizi orizzontali per applicazioni
* Servizi verticali per dominii
* Inter-ORB
  + Permette a più ORB di comunicare
* **MANCA SINCRONIA**

**Microsoft .NET**

Basato su:

* Common Language Runtinme
  + Applicazioni (scritte in C#, Visual Basic, F# ecc) eseguite da macchine virtuali
* .NET Remoting eredita il ruolo di meccanismo di comuncazione remota tra oggetti avuto da DCCOM e COM+
* Suddivide i compiti in:

**Enterprise Java**

Ha l’obiettivo di limitare la complessità per la realizzazione di servizi basati su oggetti, questo, sfruttando il riuso. Una delle caratteristiche importanti è la suddivisione tra layer di presentazione e quello di business, così da permettere facilmente politiche di bilanciamento di carico (trasparenza di migrazione)

È un modello a componenti, ovvero, moduli software che può essere ricombinato in binario a differenza di librerie e oggetti tradizionali

Le componenti sono gestite da un server detto container

Ha un middleware implicito, ovvero, non va invocato ma viene gestito automaticamente. E, all’occorrenza può essere modificato tramite un file esterno

**Middleware implicito ed esplicito**

Storicamente i primi sistemi di MW erano **espliciti**:

* Andavano esplicitamente invocati dal progettista all’interno del codice, tramite un’interfaccia remota. Queste sono chiamate alle **Middleware API** (***specifiche per quel middleware***)

Questo causa diversi problemi:

* Complessità
* Mancanza di controllo del sistema su errori
* Mancanza di portabilità (poiché potrebbe non essere standard l’API Middleware chiamata)

Il **middleware implicito** invece esegue le chiamate senza l’intervento del progettista, risolvendo molti dei problemi del middleware esplicito.

Facendo una netta separazione tra logica di business (codice java e logica per la risoluzione del problema o del servizio) e logica di presentazione (logica usata dal middleware = API)

La decisione della logica di presentazione (o dei servizi di Middleware da richiedere), vanno esplicitati in un file esterno (di configurazione detto di **deployment**) come richieste in maniera generica piuttosto che l’invocazione di una API precisa; Questa, funzionando da innesto attraverso il **Request Interceptor**, che rappresenta un nuovo agente che chiama le API richieste nel file di configurazione oltre all’oggetto (logica di business).

Quindi il **Request Interceptor** riesce a fare chiamate alle API Middleware a prescindere dal Middleware utilizzato, rendendo questo sistema “invisibile” al progettista dell’oggetto distribuito

**Programmazione concorrente e thread**

La necessità della programmazione concorrente nasce con la limitatezza della crescita del numero dei transistor e limiti fisici invalicabili. Questi nel caso del numero di transistor sono limitati per via della loro vicinanza e quindi conseguenzialmente il calore prodotto. Poiché aggiungere altri transistor sono possibili da aggiungere ma difficilmente raffreddabili

Ora più che mai è necessario migliorare tecniche e tecnicismi di programmazione sfruttando il più possibile l’architettura del calcolatore.

Sfortunatamente, bilanciare il carico è spesso difficile e può causare vari hotspot. Perciò ormai il miglioramento tecnico si basa sull’uso di più transistor, certo, ma suddivisi in più core, sfruttando la capacità di eseguire concorrenzialmente i calcoli.

La preferenza nell’uso di più core nasce per via del problema in cui più thread potrebbero accedere alla stessa memoria condivisa, causando svariati errori e rallentamenti.

Programmare in maniera concorrente, richiede la conoscenza di programmare in maniera distribuita. I primi si distinguono in tre tipi:

1. Su calcolatori diversi
2. Processi concorrenti sulla stessa macchina (multitask)
   1. Processo padre concorre con i figli generati con fork
3. Programmazione in più thread (multithread)

Il multitasking è solo un’illusione di programmazione concorrente, in quanto a livello di SO, i processi avvengono in maniera sequenziale ma **concorrendo** per l’uso della CPU

Il multithreading invece è l’estensione del multitasking ma basato su un solo processo, che lavora però su più thread

Un esempio del multithreading:

* Un browser contemporaneamente deve:
  + Renderizzare una pagina web
  + Fare richieste alla pagina web
  + Ecc

**Thread in Java**

Processo = ambiente di esecuzione con spazio di memoria privato

* La cooperazione tra processi avviene attraverso InterProcess Communication:
  + Pipe
  + Socket

Thread = esistono all’interno di un processo condividendo la memoria

In java un thread è un oggetto di una classe Thread. Con l’evoluzione di Java, sono nate due modalità di gestione dei thread:

1. Istanziare un oggetto thread ogni volta che serve compiere un task asincrono
   1. Il programmatore ha il compito di creare e gestire il thread creato, questi possono essere creati in due modi:
      1. Estendendo la classe Thread
      2. Implementando l’interfaccia Runnable
2. Astrarre la gestione, passando un task ad un executor

I Thread in java invocano il metodo run() che rappresenta il core che viene richiamato per l’uso di questi

Con i thread è possibile invocare il metodo sleep (della classe Thread) per causare una pausa nel thread che contiene l’esecuzione (corrente); anche se in sleep il processo può lanciare un’eccezione di InterruptedException quando viene terminato bruscamente da altri processi, da un master o per qualsiasi causa che non sia la normale terminazione delle istruzioni del thread.

Quindi l’InterruptedException si verifica quando per un Interrupt il thread viene interrotto, un Interrupt è un’indicazione che controlla l’esecuzione del thread con il potere di fermarne l’esecuzione

Quando un thread ha necessità di attendere il completamento di un altro thread si può eseguire il metodo join() che mette il thread corrente in attesa finché il thread atteso non termina (è possibile anche specificare un periodo di attesa come parametro, quello di default è deciso dal SO) (crea una relazione di sequenzialità)

È possibile anche interrompere l’esecuzione di un thread tramite il metodo interrupt()

**Errori nei thread**

I thread comunicano tra loro condividendo:

* Campi (tipi primitivi)
* Campi (che contengono riderimenti ad oggetti)

È un sistema di comunicazione molto efficiente che però da luce a due tipi di errori:

* Interferenza di thread
* Inconsistenza della memoria

Per entrambi è possibile la stessa soluzione, la sincronizzazione. Questa però genera altri tipi di problemi di contesa o **race condition**(quando più thread vogliono accedere alla stessa risorsa):

* Deadlock (starvation)
  + Un thread aspetta troppo
* Livelock
  + I thread aspettano l’altro thread che li sta aspettando

Altri problemi di contesa sono la modifica della stessa risorsa che porta a comportamenti inaspettati (causati dall’**interleaving dei thread**), questo tipo di problema è uno dei tipici **errori di consistenza**.

In questi casi è necessario definire una relazione ***happens-before*** poiché i thread potrebbero essere schedulati in modo diverso ogni volta.

**Relazione happens-before**

Uno dei metodi di definire una happens-before è tramite la sincronizzazione:

* Il thread creato dev’essere coerente con il thread padre, ovvero il nuovo thread creato è in grado di vedere gli effetti del codice precedenti alla sua creazione
* Un altro metodo è quello di utilizzare una variabile volatile, ovvero scrivendo ad un campo happens-before ogni successiva lettura della variabile (da parte di qualsiasi thread)

**Legge di Amdahl**

Usando un esempio:

5 amici vogliono dipingere una casa con 5 stanze. Consideriamo cge le 5 stanze siano uguali e i 5 amici siano ugualmente produttivi, allora finiranno in 1/5 del tempo che ci avrebbe impiegato una sola persona.

Consideriamo poi il caso in cui una stanza è grande il doppio delle altre, il risultato sarà diverso. Questo risultato sarà dato dalla sincronizzazione che avranno i 5 che dipingono l’ultima stanza più grande.

Questo concetto è chiamato **speedup** **S**, si definisce di un programma X: il rapporto tra il tempo impiegato da un processore per eseguire X rispetto al tempo impiegato da n processori per eseguire X. Sia p la parte del programma che è possibile parallelizzare ed n il numero di processori

Legge di Amdhal:

La legge di amdhal assume quindi che la parte sequenziale del programma rallenta significaticvamente qualsiasi speedup che possiamo pensare di ottenre, allora per rendere più efficiente un programma non è esclusivamente necessario migliorare l’hardware, quanto rendere la parte parallela predominante rispetto a quella sequenziale.

**Sincronizzazione e comunicazione tra thread**

Tipicamente i thread:

* Condividono campi
  + Tipi primitivi
  + Riferimenti ad oggetti
* Comunicano efficientemente rispetto all’uso della rete

La comunicazione tra thread può causare errori:

* Inferenza di thread (race condition che porta all’interruzione dei vari thread)
* Inconsistenza della memoria (race condition porta a memoria inconsistente)

Questi problemi possono essere risolti tramite la sincronizzazione:

* Può portare ad altri errori
  + Starvation
  + Deadlock
  + Livelock

Noti i problemi, abbiamo gli strumenti offerti dal linguaggio Java, questi con un trade-off:

* Rendono più facile l’identificazione dei dati condivisi
* Efficienza: gli stumenti possono fare da bottle-neck

**Sincronizzazione in Java**

In java è possibile ottenere la sincronizzazione, si aggiunge la caratteristica **synchronized** alla sua dichiarazione. Questo rende **facile gestire** la sincronizzazione (a costo dell’**efficienza**)

L’uso di un metodo sincronizzato, implica che non possono avvenire due esecuzioni dello stesso metodo interfogliandosi:

* Questo avviene perché, quando viene eseguito un metodo sincronizzato, gli altri thread che eseguono lo stesso metodo sullo stesso oggetto vengono messi in pausa
* Alla fine dell’esecuzione di un metodo sincronizzato, vengono stabilite relazioni di *happens-before* con le successive invocazioni dello stesso metodo sullo stesso oggetto
  + I cambi eseguiti sono visibili a tutti gli altri thread (evita inconsistenza)
* I costruttori non possono essere sincronizzati

**EVITARE I LEAKAGE:**

Quando si costruisce un oggetto considivo tra più thread, non deve assolutamente essere **leaked** il riferimento prima che questo venga costruito, per esempio:

* Dato un oggetto, nel suo costruttore aggiungiamo una chiamata instances.add(this), l’oggetto viene aggiunto ad instances (che è una lista) prima che l’oggetto stesso sia finito di essere costruito
  + Altri thread possono accedere ad instances anche se l’oggetto non è ancora stato costruito del tutto

**Sincronizzando i metodi in Java abbiamo la sicurezza che non si possano interfogliare più esecuzioni di un metodo sincronizzato sullo stesso oggetto**

**Lock intrinseci (monitor lock)**

È un’entità associata ad ogni oggetto, questo garantisce sia accesso esclusivo sia accesso consistente.

Il suo compito è quindi:

* Garantire accesso esclusivo
  + Modificare l’oggetto
* Garantire accesso consistente
  + Relazione *happens-before*

Quando il lock che possedeva viene rilasciato, viene stabilita la relazione *happens-before*.

Durante l’esecuzione di un metodo sincronizzato di un oggetto con un lock, questo viene assunto dal thread, e viene rilasciato solamente alla fine dell’esecuzione del metodo.

**Sincronizzazione di metodi statici**

Un metodo statico synchronized previene l’esecuzione interfogliata di tutti gli altri metodi statici sincronizzati, praticamente si acquisisce il lock dell’oggetto **ClassName.class**

**Differenza tra metodi statici sincronizzati e metodi sincronizzati di istanza:**

* Metodi **sincronizzati** **statici** garantiscono accesso in **mutua esclusione** a metodi **sincronizzati statici**, mentre metodi **sincronizzati di istanza** garantiscono accesso in mutua esclusione ai **metodi sincronizzati** di **quella** istanza.
* ***Entrambi rendono facile utilizzare i metodi sincronizzati a scapito dell’efficienza***

**Accesso atomico**

Un’azione atomica si definisce tale quando sono azioni **interrompibili** e che o si completano del tutto o non lo fanno per niente

In java è possibile specificarle per:

* Read e write su variabili di riferimento e su tipi primitivi (tranne long e double)
* Read e write sono tutte le variabili **volatile**

Write a variabili volatile stabiliscono una relazione *happens-before* con letture successive (**si scrive necessariamente prima che si legga**). Le operazioni atomiche in java sono definite in *java.util.concurrent.atomic*

I metodi atomici non hanno bisogno di essere sincronizzati per la natura stessa del funzionamento atomico

**Problemi della sincronizzazione thread**

Quando due thread sono bloccati uno in attesa dell’altro si dice **deadlock**, in questa maniera il programma concorrente si blocca senza possibilità di sbloccarsi.

Si dice **starvation** il caso in cui un thread non riesce ad acquisire in accesso ad una risorsa condivisa

* Questo avviene per via di processi **greedy** che occupano finché possibile

**Livelock** è quando dati due processi a e b, questi due **perdono tempo** risponendo all’altro. Generalmente si differenzia dal deadlock poiché entrambi i processi provano a dare precedenza all’altro, mentre nel deadlock entrambi aspettano di avere la precedenza dall’altro

**Efficienza dei thread**

Consideriamo un array di grandi dimensioni (10k per esempio) che dev’essere inizializzato, nel caso del singolo thread questo dovrà fare 10k iiterazioni.

Nel caso di un uso concorrente di thread, ognuno dovrà gestire solo una porzione della dimensione totale. Una volta suddivisi in n parti lo spazio dell’array, e assegnate agli n thread, il main aspetta in join tutti i thread la fine della computazione.

**Singleton e Double-checked Locking**

Si definisce Singleton un design pattern tipico per la programmazione ad oggetti, in cui, si restringe la istanziazione da parte di una classe ad un solo oggetto.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamenteSi usa solitamente per una **lazy allocation**, ovvero, quando l’allocazione avviene solo quando utilizzato per la prima volta:

Nel caso di un’implementazione in java della tecnica lazy allocation con un singleton senza utilizzare **synchronized**, questo, può causare errori tipo quello di creare **più singleton** per la stessa istanza (per via di un incorretto ***interleaving*** [*interfogliamento*])

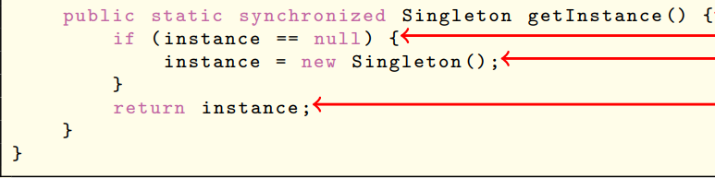
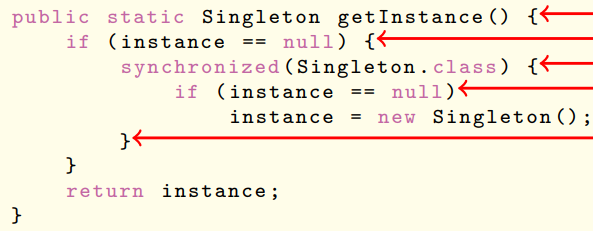
Correggendo il metodo per creare un’istanza aggiungendo il modificatore synchronized si risolve l’interfogliamento, però si causerà una grande perdita in efficienza poiché il synchronized viene considerato solo la prima volta, il resto delle altre le istanze non verrebbero mai create (si controlla che questa esista già o meno) allora si sprecherebbe tempo di attesa degli altri thread in attese inutili:

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamenteSpostando invece il lock ad una grana più fine solo sulla creazione dell’istanza (dopo l’if), ci troveremo nel primo caso in cui thread diversi creerebbero istanze diverse:

Per risolvere questo problema possiamo utilizzare il **double-checked locking**, ovvero, dopo una verifica preliminare che l’istanza sia stata creata, si utilizza un lock per sequenzializzare (situazione precedente), e in seguito al lock utilizziamo un secondo controllo così da evitare il problema precedente. Così rendendo unico il singleton:

Il double-checked locking **potrebbe non funzionare**: poiché il controllo instance == null avviene prima che instance venga inizializzata

Si può risolvere il problema del double-checked locking, definento la variabile instance come *volatile* (standard dopo Java 5)

Alternativamente, sono consigliate classi statiche con l’obiettivo di farle inizializzare ad un’altra classe secondo l’idea *Initialization-on-deman holder* in cui la JVM inizializza l’istanza alla prima invocazione getInstance(), essendo statico viene eseguito solo una volta