Domanda 1

Race conditions: Quando più thread accedono contemporaneamente alla stessa risorsa e il risultato dipende dall'ordine di esecuzione. Il comportamento diventa imprevedibile.

Deadlock: Situazione di stallo dove due o più thread si bloccano reciprocamente aspettando risorse possedute dagli altri. Nessuno può procedere.

Indeterminismo: L'esecuzione concorrente è intrinsecamente non deterministica - lo stesso programma può produrre risultati diversi in esecuzioni diverse a causa dell'interleaving dei thread.

Starvation: Un thread non riesce mai ad ottenere le risorse necessarie perché altri thread le monopolizzano continuamente.

Le altre opzioni **non sono problematiche** della concorrenza:

- **Amhdal's law**: È un limite teorico del parallelismo, non un problema
- **Preemption**: È un meccanismo del sistema operativo per gestire i thread
- **Instruction ordering**: È una caratteristica dell'architettura, non necessariamente un problema
- **Clock skew**: È un problema di sincronizzazione temporale nei sistemi distribuiti

Legge di Amdahl (Amdahl's Law):

È un teorema che stabilisce il **limite teorico massimo** del miglioramento delle prestazioni ottenibile parallelizzando un programma.

Formula: Speedup = 1 / (S + P/N)

Risposta 2

Serializzazione/Marshalling in parole semplici:

È il processo di **convertire un oggetto in memoria in una sequenza di byte** che può essere:

- Salvata su disco
- Inviata in rete
- Memorizzata in un database

Analogia: È come **impacchettare** un oggetto complesso per spedirlo via posta.

Processo:

1. **Serializzazione**: Oggetto → Byte array
2. **Trasmissione/Storage**: I byte viaggiano
3. **Deserializzazione**: Byte array → Oggetto ricostruito

Termini:

- **Marshalling** = termine più generale (include metadati)
- **Serializzazione** = conversione specifica in formato binario/testuale

Maschera complessità distribuendo il formato standard attraverso lo scambio di messaggi (scambio asincrono su un canale = simula concorrenza).

Domanda 3

Channels in Java - spiegazione semplice:

Problema tradizionale: Con Socket normali scrivi codice così:

```
while(true) {
    Socket client = server.accept(); // BLOCCA il thread
    // gestisci client...
}
```

Un thread = una connessione. Con 1000 client = 1000 thread = disastro.

Soluzione Channels: Invece di bloccare i thread, **registri callback** che vengono chiamate quando succede qualcosa:

```
// Registri cosa fare QUANDO arriva una connessione
server.accept(attachment, new CompletionHandler() {
    void completed(AsynchronousSocketChannel client, Object attachment) {
        // Connessione arrivata! Gestiscila
        // Poi rimettiti in ascolto per la prossima
        server.accept(attachment, this);
    }
});
```

Nota a margine: Callback = Attendi il risultato asincrono che ti viene passato "ad una certa" come risultato o come funzione.

Differenza chiave:

- **Socket**: "Aspetto finché non arriva qualcosa" (thread bloccato)
- **Channel**: "Dimmi quando arriva qualcosa" (thread libero)

Vantaggio: Un solo thread può gestire migliaia di connessioni perché non si blocca mai.

Svantaggio: Devi ristrutturare tutto il codice in piccoli metodi che reagiscono agli eventi. È più complesso da scrivere.

In pratica: È come la differenza tra stare al telefono aspettando (Socket) vs. lasciare il numero per essere richiamati (Channel).

Differenze informatichesi

(1) Socket vs Datagram

Socket (TCP):

- **Connessione**: Devi "chiamare" e rimanere in linea
- **Affidabilità**: Garantisce che i dati arrivino tutti e nell'ordine giusto
- **Bidirezionale**: Puoi parlare e ascoltare contemporaneamente
- **Stream di byte**: Non sai dove finisce un messaggio
- **Overhead**: Più lento ma più sicuro

Datagram (UDP):

- **Senza connessione**: Mandi "cartoline" senza sapere se arrivano
- **Inaffidabile**: I pacchetti possono perdersi o arrivare disordinati
- **Unidirezionale**: Mandi e basta, se vuoi risposta devi metterti in ascolto
- **Messaggi delimitati**: Ogni pacchetto è un messaggio completo
- **Veloce**: Meno overhead, più performance

Analogia: Socket = telefonata, Datagram = cartolina

---

(2) Thread vs Processi

Thread:

- **Condividono memoria**: Accesso alle stesse variabili (pericoloso!)
- **Leggeri**: Creazione e cambio contesto veloce
- **Comunicazione facile**: Basta leggere/scrivere variabili condivise
- **Un crash = tutti giù**: Se uno va in crash, tutto il processo muore

Processi:

- **Memoria isolata**: Ognuno ha il suo spazio, nessuna interferenza
- **Pesanti**: Creazione e cambio contesto costoso
- **Comunicazione complessa**: Serve IPC (pipe, socket, shared memory)
- **Isolamento**: Se uno va in crash, gli altri continuano

Analogia: Thread = coinquilini (condividono casa), Processi = case separate

Domanda 4

In computer science, a fiber is a particularly lightweight thread of execution.

- Like threads, fibers share address space
- However, fibers use cooperative multitasking while threads use preemptive multitasking
- Threads often depend on the kernel's thread scheduler to preempt a busy thread and resume another thread; fibers yield themselves to run another fiber while executing

Domanda 5

Cos'è un Framework

Un **framework** è come una **casa già costruita** dove tu devi solo arredare le stanze.

Ti fornisce:

- **Struttura base**: L'architettura principale è già decisa
- **Componenti comuni**: Soluzioni già pronte per problemi tipici
- **Regole del gioco**: Come devi organizzare il tuo codice
- **Infrastruttura**: Gestisce automaticamente le parti noiose/complesse

Differenza libreria vs framework:

- **Libreria**: Tu chiami le sue funzioni quando ti serve
- **Framework**: Lui chiama il tuo codice quando serve a lui

Vantaggi Framework per Applicazioni Distribuite

✅ **Vantaggi**:

Facilità di realizzazione:

- Protocolli di rete già gestiti (HTTP, TCP, serializzazione)
- Parti strutturali già implementate (routing, load balancing)

Maggiore sicurezza:

- Gestito da esperti che sanno i trabocchetti
- Vulnerabilità comuni già risolte

Aggiornamento automatico:

- Patch di sicurezza e miglioramenti senza che tu faccia nulla
- Benefici a tutta l'applicazione quasi gratis

❌ **Svantaggi**:

- **Casi particolari**: Se il tuo uso non è "standard", diventa complicatissimo
- **Dipendenza**: Sei in balia delle loro scelte e tempistiche
- **Black box**: Quando si rompe, non sai perché
- **Lock-in**: Difficile cambiare framework dopo

Esempio: Spring vs. Socket puri per un web server. Spring gestisce tutto automaticamente, ma se vuoi fare qualcosa di "strano" diventa un incubo.

Domanda 6

Perché Java ha avuto successo

Contesto storico (anni '90):

- **Problema**: Ogni sistema aveva linguaggi/architetture diverse
- **Costo**: Riscrivere software per ogni piattaforma = disastro economico
- **Internet emergente**: Servivano programmi che girassero ovunque

Promessa di Java: **"Write Once, Run Anywhere"**

Come funziona il Bytecode/JVM

Processo tradizionale:

```
C/C++ → Compilatore → Codice macchina specifico (x86, ARM, etc.)
```

Risultato: Un eseguibile per ogni architettura.

Processo Java:

```
Java → Compilatore javac → Bytecode (universale)
Bytecode → JVM → Codice macchina specifico
```

Bytecode = "linguaggio macchina virtuale":

- Formato intermedio, indipendente dall'architettura
- La JVM lo traduce nel codice macchina del sistema ospite

Vantaggi strategici

Per le aziende:

- **Un codice, tutti i sistemi**: Windows, Linux, mainframe
- **Costi ridotti**: Non serve riscrivere tutto
- **Mercato più ampio**: Il software gira ovunque

Tecnici:

- **Memory management automatico**: Garbage Collection
- **Sicurezza**: Sandboxing, controllo accessi
- **Librerie ricche**: Tutto incluso nella standard library
- **Performance accettabile**: JIT compilation migliora nel tempo

Ecosistema:

- Supporto enterprise (IBM, Oracle)
- Community enorme
- Strumenti di sviluppo maturi

Risultato: Java è diventato lo standard per applicazioni enterprise e server-side, dove la portabilità e robustezza contano più della performance pura.

Domanda 7

Il Problema del Consenso Distribuito

Scenario: Hai più server che devono **concordare su un dato** (es. saldo di un conto bancario).

Il Dilemma: Come fanno i nodi a mettersi d'accordo quando:

- La rete può perdere messaggi
- I nodi possono guastarsi
- Non c'è un "orologio globale"
- Non sai se un nodo è lento o morto

Esempio Concreto

Banca con 3 server:

- Server A: "Saldo = 1000€"
- Server B: "Saldo = 1000€"
- Server C: "Saldo = 1000€"

Cliente fa bonifico di 100€:

- Server A riceve: "Saldo = 900€"
- Server B: messaggio perso, rimane "1000€"
- Server C: guasto, offline

Problema: Che saldo è quello giusto? 900€ o 1000€?

Soluzioni: Algoritmi di Consenso

PAXOS:

- Protocollo a 3 fasi con ruoli precisi (leader, votanti, ascoltatori)
- Garantisce accordo anche con alcuni nodi guasti
- Complesso da implementare

RAFT:

- Più semplice di Paxos
- Elezione di un leader che coordina tutto
- Se il leader muore, se ne elegge un altro

Principio base: Serve una **maggioranza** (quorum) per decidere. Con 5 nodi, almeno 3 devono essere d'accordo.

Perché è Difficile

Teorema CAP: In caso di partizione di rete, scegli:

- **Consistenza**: Tutti hanno lo stesso dato (ma alcuni nodi non rispondono)
- **Disponibilità**: Tutti rispondono (ma con dati potenzialmente diversi)

Problema dei Generali Bizantini: Come accordarsi quando alcuni "tradiscono" (inviano informazioni sbagliate)?

Bottom line: Il consenso distribuito è **matematicamente complesso** ma necessario per sistemi affidabili.