Documentatie Sistem Programari Clinica Medicala

1. Documentatie

Descriere Proiect

Sistemul implementeaza o aplicatie de tip Client-Server pentru gestionarea programarilor intr-o retea de clinici medicale. Aplicatia utilizeaza protocoale TCP/IP pentru comunicare si mecanisme avansate de multithreading pentru a gestiona accesul concurent la resurse partajate (locuri disponibile, plati).

Mediu de lucru

Limbaj: Java (JDK 23+)

Tehnologii: Java Networking (Sockets), Java Concurrency API (ExecutorService, Atomic variables, Locks)

Format date: Mesaje text structurate prin pipe (|)

Parametri de rulare

Parametrii sunt configurati in clasa `ClinicServer.java` si `Main.java`:

Port: 8888

Thread Pool Size: 10 (numar maxim de clienti procesati simultan)

Numar Locatii: 5

Numar Tratamente: 5

Interval verificare periodica: Hardcodat la 5 sau 10 secunde (variabila VERIFICATION_INTERVAL)

Timeout Plata: 30 secunde

Durata rulare server: 180 secunde (3 minute)

Cum se ruleaza aplicatia

Compilare:

```
javac org/example/*.java
```

Executie Server si Clienti (prin Main):"

```
java org.example.Main
```

Clasa Main va porni automat serverul intr-un thread separat, apoi va instantia si conecta cei 10 clienti.

2. Raport Tehnic: Arhitectura si Performanta

Arhitectura Sistemului

Sistemul adopta un model Client-Server Distribuit:

- **ClinicServer**: Centralizeaza logica de business. Utilizeaza un ServerSocket pentru a asculta conexiuni si un FixedThreadPool pentru a delega fiecare client unui handler separat.
- **ClientHandler** (Runnable): Gestioneaza ciclul de viata al unei sesiuni: citeste cererea, acceseaza resursele sincronizate, returneaza raspunsul.
- **ClinicClient**: Simuleaza comportamentul pacientilor. Genereaza cereri de programare (BOOK) la fiecare 2 secunde. Daca programarea este acceptata, trimite imediat o cerere de plata (PAY).

Decizii de Sincronizare si Concurenta

- Pentru a preveni coruperea datelor la accesul simultan al celor 10 thread-uri de client si al thread-ului de verificare:
- **Sincronizarea Listelor**: S-a utilizat Collections.synchronizedList pentru appointments si payments. Aceasta asigura thread-safety la operatiile de baza de adaugare/eliminare.
- **Lacate Explicite (Locks)**: S-au folosit obiecte de tip Object (appointmentLock, paymentLock) in blocuri synchronized in timpul procesului de verificare. Acest lucru garanteaza ca bilantul financiar si lista de programari nu se modifica in timp ce raportul este generat (snapshot consistency).
- **Atomicitate**: Generarea ID-urilor pentru plati si programari se face prin AtomicInteger, eliminand necesitatea lock-urilor greoaie pentru o simpla incrementare.

Utilizarea Thread Pool si Futures

- FixedThreadPool (10 thread-uri): Alegerea unui numar fix egal cu numarul de clienti asigura ca fiecare client are un handler dedicat fara a supraincarca procesorul.
- ScheduledExecutorService: - In **Server**: Folosit pentru `VERIFICATION_INTERVAL` (rularea logicii de verificare la 5s/10s) si pentru monitorizarea timeout-ului de plata (invalidarea rezervarilor dupa 30s).
    - In **Client**: Folosit pentru a trimite cereri la intervale exacte de 2 secunde.
- **Futures**: Desi executia este in mare parte de tip "fire-and-forget", mecanismul submit() permite serverului sa gestioneze asincron fluxurile de date.

Analiza Performantei (Bazata pe Output-uri)

Scenariul 1: Verificare la 5 secunde (`run_5s.txt`) **Mecanism Timeout**: Log-urile arata ca la ora 12:33:13 au fost detectate programari neplatite. Dupa 30 de secunde, sistemul a eliberat locurile (ID-urile 6, 7, 8, 9 au disparut din liste), demonstrand ca thread-ul de monitorizare a timeout-ului functioneaza corect.

Consistenta: Bilantul pe locatii (ex: Locatia 4: 140.0 RON) ramane constant in verificarile succesive daca nu apar tranzactii noi, ceea ce indica absenta "lost updates".

Scenariul 2: Verificare la 10 secunde (`run_10s.txt`) Throughput: In acest mod, serverul a procesat un volum mare de date initial (9 programari confirmate si platite rapid).

Eficienta: Toti cei 10 clienti au fost deserviti fara erori de conexiune sau "Connection Refused", indicand faptul ca pool-ul de 10 thread-uri este dimensionat optim pentru sarcina curenta.

Concluzii Performanta Timp mediu de raspuns: Din log-uri se observa ca raspunsurile sunt trimise in aceeasi secunda cu cererea (latenta sub 100ms), ceea ce este ideal pentru o aplicatie de rezervari.

Capacitate: Sistemul sustine 5 cereri/secunda (10 clienti la 2s interval) cu un consum minim de resurse. Punctul critic ar fi lock-ul de verificare, insa acesta dureaza foarte putin (doar iterare si printare), neafectand experienta utilizatorilor.