Tehnici de programare-Tema 2

Oprean Andrei

Gr.30228 An II Seria B

1. Obiectivul temei

Creearea unei aplicatii in Java care eficientizeaza distributia clientilor intr-un numar dat de cozi. Afisarea in timp real a starii fiecarei cozi si a starii sistemului .

2. Analiza problemei,modelare,scenarii,cazuri de utilizare

2.1. Analiza problemei

Folosind ca date de intrare numarul de cozi,intervalul de simulare,frecventa cu care ajung clientii la cozi exprimata in secunde si timpul individual de procesare al fiecarui client putem distribui clientii intr-un mod eficient astfel incat sa micsoram timpul de asteptare la coada al fiecarui client .

2.2. Modelare

Fiecare coada la care se va aseza un client este specifica un server care se va ocupa de procesarea efectiva a clientului . Avand ca informatie de intrare numarul de cozi, il putem translate in servere, urmand ca si coada sa fie folosita doar ca un loc de asteptare al clientilor care asteapta sa fie serviti. Fiecare server va lucra individual, deci pot fii in curs de procesare un numar de clienti egal cu numarul de cozi. Tot in timp real, utilizatorul va putea urmari starea fiecarei cozi si jurnalul de activitate al aplicatiei . Avand aceste informatii putem deduce ca vom avea nevoie de o clasa care va modela clientul ca entitate, o clasa care va modela serverul si care va contine o coada de obiecte de tip client, o clasa care va dirija aceste operatii si din care se va lansa executia fiecarui thread si o clasa de test in care se vor lua datele de intrare si care le va transmite catre clasa de procesare. Tot in aceasta clasa vom rula si metoda main().

2.3. Scenarii si cazuri de utilizare

Avem doua scenarii de utilizare care depind de datele de intrare:

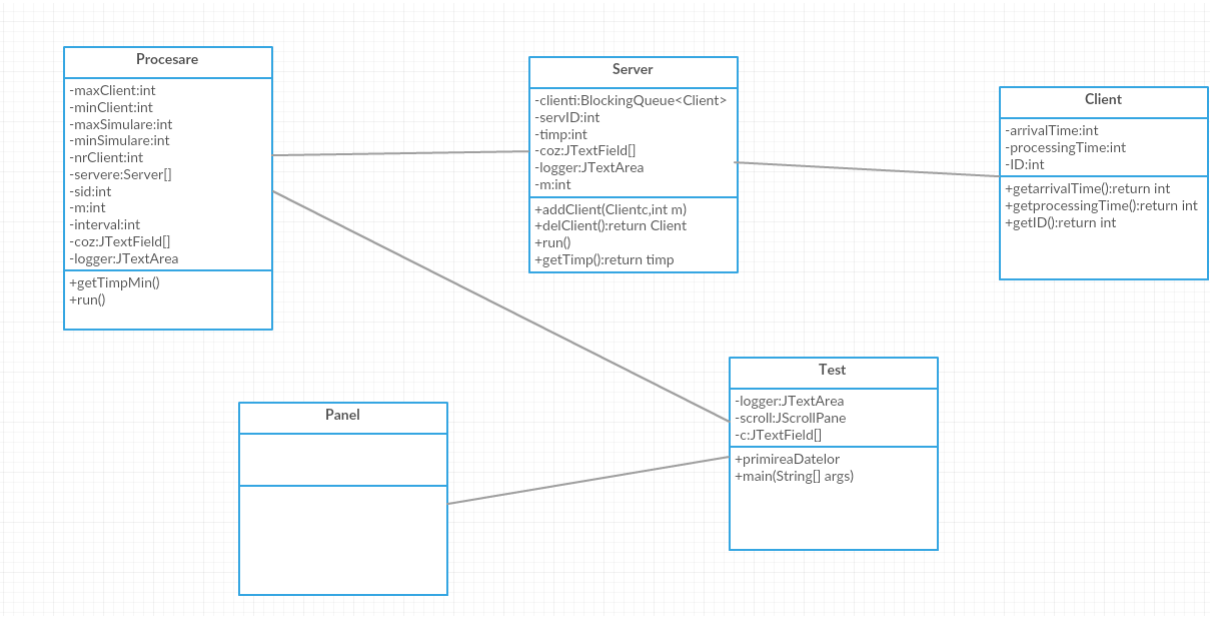
-Intervalul de venire al clientilor este mai mic decat timpul mediu de procesare al fiecaruia, caz in care cozile se vor supra-aglomera din cauza faptului ca serverul nu poate procesa clientii in ritmul in care ei ajung in coada. Acest scenariu este dependent de numarul de cozi din sistem. Este un scenariu mai apropiat de realitate .

-Intervalul de venire al clientilor este mai mare decat timpul mediu de procesare al fiecaruia, caz in care cozile vor fi in majoritatea timpului goale,in functie de diferenta dintre cele doua date de intrare . Acest scenariu ar putea fi apropiat de scenariul unor servere care gestioneaza diferite incercari de citire sau trimitere a datelor .

Un al treilea scenariu ar putea aparea atunci cand numarul de cozi introdus este foarte mare pentru ca desi este putin probabil intr-un mediu din lumea reala, evolutia cozilor ar putea deveni greu de interpretat de catre utilizator .

Aplicatia se va putea utiliza pentru simularea diferitelor scenarii care vor arata evolutia procesarii clientilor in functie de numarul de cozi si timpii de intrare. Spre exemplu se poate vedea diferenta dintre numarul mediu de clienti de la fiecare coada in functie de numarul cozilor. Cu aceste informatii in timp real un utilizator poate sa gaseasca cea mai buna strategie pentru sistemul pe care ar vrea sa il implementeze in realitate. Un alt caz de utilizare ar putea fi eficientizarea accesului utilizatorilor la un site sau o baza de date . Deoarece acestea nu pot procesa decat un anumit numar de interogari la un anumit moment cererile vor trebui puse intr-o coada si procesate in functie de criteriile specific fiecareia .

3.Proiectare



Pentru a simula activitatea sistemului descris am folosit o lista de servere.Serverele sunt modelate cu ajutorul clasei Server care are in componenta ei o coada de clienti.Fiecare server reprezinta un thread separat care opereaza pe propria coada de clienti . La fel ca intr-un scenariu de zi cu zi serverul se ocupa de operatiile de adaugare si scoatere a clientilor din coada.Pentru modelarea cozii am ales sa folosesc colectia BlockingQueue<> care ofera suport automat pentru sincronizarea operatiile de introducere si scoatere in mediul concurent.Din

punct de vedere al eficientizarii am ales sa distribui clientii in functie de timpul de asteptare cumulat al fiecarei cozi . Acest timp se calculeaza inainte de fiecare introducere in coada. Variabila in care se va stoca indexul cozii cu cel mai mic timp de asteptare este folosita si in clasa Server pentru a stii unde trebuie inserat in interfara grafica clientul respectiv .

Pentru interfata grafica cu utilizatorul am ales sa folosesc, initial, o serie de campuri in care utilizatorul sa poata introduce datele de intrare. Apoi in functie de numarul de cozi se vor genera atatea campuri in care utilizatorul poate urmari evolutia in timp real a cozilor. Pe langa acestea evolutia sistemului se poate observa si in loggul de activitate care afiseaza fiecare operatie de introducere si procesare din sistem.Acest logger salveaza toate afisarile astfel ca poate fii urmarita evolutia sistemului de la inceput pana la sfarsit cu ajutorul scrollului. Astfel ca interfata grafica are un aspect placut care face urmarirea datelor usoara prin plasarea acestor informatii in apropierea una de alta fara ca ele sa se intercaleze.

4. Implementare si testare

4.1 Interfta grafica cu ultilaztorul

Este impartita in 3 panouri:

-Pentru intrare , in care pentru fiecare informatie folosesc un JTextField care sa permita introducerea si stergerea usoara a datelor. Fiecare dintre aceste textfield-uri este indentificabila cu ajutorul unor JLabel-uri puse inainte fiecareia. Astfel utilizatorul stie exact cum trebuie introduse datele. Pentru a valida datele, am mai adaugat un buton “ok”, care atunci cand este apasat transmite datele aplicatiei si porneste simularea.

-Pentru urmarirea in timp real a evolutiei cozilor am generat dinamic un array de JTextField-uri care are dimensiunea data de utilizator la field-ul pentru numarul de cozi.Dupa apasarea butonului de validare a datelor interfara grafica este reactualizata. Aceste field-uri corespund fiecarei cozi a fiecarui server. Clientii sunt adaugati si stersi de metodele respective ale aplicatiei, astfel ca textul din fiecare field este controlat de thread-ul serverului din care face parte. Din nou, fiecare field este etichetat corespunzator astfel ca utilizatorul poate urmari usor evolutia fiecarei cozi. La fiecare adaugare programul sterge mai intai tot textul pentru ca mai apoi sa deseneze din nou clientii in functie de dimensiunea noua a cozii .

-Pentru urmarirea evenimentelor de adaugare si procesare din tot sistemul am folosit un JTextArea incapsulat intr-un JScrollPane. Acesta permite urmarirea evolutiei sistemului de la inceput pana la final cu ajutorul barelor de scroll. Si acesta este actualizat dinamic de catre fiecare thread atunci cand efectueaza o operatie pe coada proprie.

4.2 Logica programului

Dupa ce datele de intrare sunt preluate de la interfata grafica cu utilizatorul , ele sunt transmise catre clasa Procesare care se ocupa de gestiunea sistemului si care tine in

component sa lista de servere si implicit de threaduri. In aceasta clasa se pornesc un numar de threaduri egale cu numarul de cozi citit de la intrare. In metoda run() a clasei Procesare se efectueaza adaugarea clientilor si asigurarea rularii programului doar in intervalul de simulare citit de la intrare. Tot in aceasta clasa avem metoda getTimpMin() care trece prin lista de servere si calculeaza timpul minim de asteptare al unui client nou. Algoritmul folosit de mine nu este unul extrem de eficient, dar avand in vedere ca este putin probabil sa se foloseasca date extrem de mari si luand in considerare intarzierile impuse de timpul de procesare si timpul de sosire al fiecarui client care sunt semnificativ mai mari decat cel mai rau caz de executare al algoritmului, solutia aceasta nu influenteaza notabil viteza de rulare a programului. Metoda returneaza indicele cozii serverului la care va trebui adaugat clientul pentru a sta cat mai putin la coada.

Clasa Server care modeleaza un singur server si coada lui este component principala a aplicatiei. Coada serverului este implementata cu ajutorul colectiei BlockingQueue care faciliteaza operatiile concurente pe cozi. Astfel ca metodele care adauga si sterg clientii se rezuma la introducerea si scoaterea din coada a cate unui client si “adormirea” threadului in functie de timpii fiecarui client. Metoda .take() din colectia BlockingQueue asteapta introducerea unui client in coada inainte de a incepe procesarea acestuia astfel ca nu mai este nevoie de sincronizare efectiva. In metoda run() exista un singur apel la metoda de scoatere a unui client astfel incat imediat cum un client intra in coada el este procesat cu conditia sa nu existe alt client in fata lui. Tot aceste metode controleaza fluxul datelor de iesire din interfata grafica astfel eliminandu-se posibile intarzieri din program. In aceste metode se face redesenare partii interfetei grafice care arata starea fiecarei cozi in timp real. Dup ace tot textul este sters se redeseneaza coada de la 0 pana la lungimea noua a cozii. Asta se intampla doar in cazul stergerii. In cazul introducerii, clientul doar este afisat ca o continuare a textului din campul de text respectiv.

Clasa Client tine doar informatii despre client. Constructorul acestei clase primeste intervalele citite din interfata grafica si generaza un client nou cu timpii specifici aleator. Pentru aceasta am folosit clasa Random cu metoda .nextInt(). Pe langa constructor clasa Client mai are doar metode de intoarcere a datelor. Fiecare client este identificat dupa un ID unic, acesta fiind generat in functie de ordinea in care este introdus intr-o coada, de exemplu primul client va avea ID-ul 0 .

Clasa Test primeste datele din interfata grafica la apasarea butonului. Pentru aceasta, in aceasta clasa este implementat action listener-ul butonului. Tot in aceasta clasa este pornit threadul asociat clasei Procesare si sunt generate cozile de clienti din interfata grafica. Mai departe, datele citite din aceasta clasa cu ajutorul action listener-ului vor fi transmise mai departe catre constructorul clasei Procesare .

5. Rezultate

Rezutatul obtinut in urma executarii acestei aplicatii este analiza in detaliu a evolutiei unui sistem format din clienti si o serie de servere capabile sa sustina o coada de asteptare si proceseze clientii adaugati la anumite momente in acea coada. Aceasta analiza este posibila in

timpul simularii cu ajutorul campurilor care afiseaza starea fiecarei cozi in timp real, iar dupa terminarea simularii, cu ajutorul campului in care este salvata toata activitatea sistemului pe parcursul simularii . Per total, rezultatul este o aplicatie usor de folosit si de citit care ofera niste informatii exacte intr-un mod direct fara a fi nevoie de interpretari successive din partea utilizatorului.

6. Concluzii, dezvoltari ulterioare

In concluzie, managementul clientilor si procesarea acestora cu ajutorul unor cozi intr-un timp eficient este, de exemplu, o metoda foarte buna pentru a creste productivitatea unei afaceri care se bazeaza pe servirea clientilor la o casa sau procesarea diferitelor informatii venite din diferite surse si centralizate intr-un server. In societatea actuala, care se bazeaza pe viteza si eficienta, un astfel de sistem ar fi indispensabil pentru orice aplicatie sau intreprindere care este folosita de un numar mare de utilizatori.

Din aceasta tema, am invatat ca necesitatea folosirii thread-urilor in aplicatii care gestioneaza diferite tipuri de date pentru a creste viteza de procesare a acestora. In acelasi timp am invatat dificultatea sincronizarii operatilor dintre threaduri, care, din fericire sunt simplificate intr-o anumita masura de diferitele colectii si tool-uri oferite de Java. In anumite cazuri comportamentul threadurilor pe anumite date commune este imprevizibil si deci este indicate folosirea metodelor sincronizate sau in cazul de fata folosirea colectiei BlockingQueue. Threadurile sunt o unealta puternica dar care aduc cu ele si multe responsabilitati care trebuie gestionate de catre programator.

Ca si dezvoltari ulterioara, un sistem de acest tip ar putea fi implementat intr-o aplicatie care se ocupa cu gestionarea informatiilor dintr-o baza de date, remarcand in acest caz necesitatea folosirii thread-urilor atunci cand dorim ca aplicatia noastra sa fie folosita de mai mult de un utilizator la un moment dat, utilizatori care probabil vor dori scrierea si citirea datelor din respective baza de date simultan. Necesitatea folosirii unei abordari de felul acesta este evidentiata mai tare cu cat numarul de utilizatori simultani creste.

Ca o schimbare in structura programului, s-ar putea pe viitor implementa inca un camp de intrare in care utilizatorul ar putea selecta unitatea de timp in care sa se desfasoare simularea. O alta dezvoltare ulterioara ar putea fi legarea acestei aplicatii la un sistem fizic, de exemplu o serie de case de marcat intr-un super-market care sa poata transmite datele in timp real aplicatiei iar aceasta, la randul ei, sa poata directiona clientii spre cel mai optim traseu pentru a fi procesati.

7. Bibliografie

www.cs.unicam.it/culmone/?download=java_concurrency_in_practice.pdf

http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

http://www.tutorialspoint.com/java/util/timer_schedule_period.htm

http://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-andthreadpoolexecutor.html

http://javahash.com/java-concurrency-future-callable-executor-example/