Documentatie

Assigment 2

NUME STUDENT: Neacă Radu-Sabin

GRUPA: 30222

Cuprins

[1. Obiectivul temei 3](file:///C:\Downloads\FACULTATE\Tehnici%20de%20programare\Assigment1\pt2022_30222_neaca_radu-sabin_assigment_1\PT2022_30222_Neacă_Radu-Sabin_Documentatie_1.docx#_Toc95297885)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 3](file:///C:\Downloads\FACULTATE\Tehnici%20de%20programare\Assigment1\pt2022_30222_neaca_radu-sabin_assigment_1\PT2022_30222_Neacă_Radu-Sabin_Documentatie_1.docx#_Toc95297886)

[3. Proiectare 4](file:///C:\Downloads\FACULTATE\Tehnici%20de%20programare\Assigment1\pt2022_30222_neaca_radu-sabin_assigment_1\PT2022_30222_Neacă_Radu-Sabin_Documentatie_1.docx#_Toc95297887)

[4. Implementare 5](file:///C:\Downloads\FACULTATE\Tehnici%20de%20programare\Assigment1\pt2022_30222_neaca_radu-sabin_assigment_1\PT2022_30222_Neacă_Radu-Sabin_Documentatie_1.docx#_Toc95297888)

[5. Rezultate 7](file:///C:\Downloads\FACULTATE\Tehnici%20de%20programare\Assigment1\pt2022_30222_neaca_radu-sabin_assigment_1\PT2022_30222_Neacă_Radu-Sabin_Documentatie_1.docx#_Toc95297889)

[6. Concluzii 7](file:///C:\Downloads\FACULTATE\Tehnici%20de%20programare\Assigment1\pt2022_30222_neaca_radu-sabin_assigment_1\PT2022_30222_Neacă_Radu-Sabin_Documentatie_1.docx#_Toc95297890)

[7. Bibliografie 7](file:///C:\Downloads\FACULTATE\Tehnici%20de%20programare\Assigment1\pt2022_30222_neaca_radu-sabin_assigment_1\PT2022_30222_Neacă_Radu-Sabin_Documentatie_1.docx#_Toc95297891)

Obiectivul temei

Obiectiv principal:

Scopul lucrarii consta in crearea unei aplicatii ce calculeaza timpul mediu de asteptare al unui client in momentul in care acesta s-a pus la coada. Acest lucru il voi realiza folosindu-ma de o lista de Thread-uri, fiecare Thread fiind o coada de clienti.

Obiective secundare:

* Analiza problemei si identificarea cerintelor.
* Organizarea codului corespunzaor: Precum structura MVC (Model-View-Controller). Fiecare clasa creata nu trebuie sa depaseasca o dimensiune mai mare de 300 de linii, cu exceptia clasei Controller. Fiecare metoda trebuie sa cuprinda maxim 30 de linii. Clasele trebuie impartite in pachete astfel incat sa existe o vizibilitate si organizare a proiectului mult mai buna.
* Testarea aplicatiei, verificarea codului astfel incat metodele folosite pe parcursul programului sa fie corect implementate, fara a prezenta cazuri de eroare.
* Implementarea si simularea unei interfete ce sa dispuna si de o fereastra ce afiseaza rezultatul dorit.
* Folosirea thread-urilor si a mecanicilor de sincronizare pentru a trimite in paralel cu procesul principal, mai multe procese, fiecare reprezentand o coada de clienti.

Analiza problemei, modelare, scenarii, cazuri de utilizare

Primul pas inainte de implementarea proiectului ar fi analiza problemei si modul cum ar trebui rezolvata aceasta. Asadar, am ales sa tratez problema client-coada, ca fiind o legatura task-server. Fiecare client reprezinta un task pe care server-ul trebuie sa il adauge, iar mai apoi, sa il execute conform cerintei. Astfel, server-ul reprezinta o coada de task-uri ce se bazeaza pe principiul FIFO (First in first out). Fiecare client are un timp cand de sosire in fata cozilor, si alt timp ce sugereaza timpul pe care il petrece in fata cozii, pe primul lor. Astfel, task-ul va continu un ID ce semnifica numele clientului, un numar TimeArrival ce semnifica momentul cand ajunge la coada si un numar TimeService, ce reprezinta durata de timp pe care o are de asteptat in momentul cand ajunge pe prima pozitie din coada. Aceasta aplicatie poate sa contina mai multe servere pentru a imbunatati sistemul, marind viteza de parcurgere a Task-urilor, in cazul in care numarul acestora ar fi prea mare pentru a fi rezolvate de un numar mic de servere. In momentul in care fiecare Task ajunge in fata si urmeaza a fi pus in unul din server, ajungem sa ne gandim la 2 strategii de rezolvare a acestei probleme de eficienta. Una o reprezinta viteza cu ajutorul careia task-urile trec mai repede prin cozi si se executa mai repede fiecare, aceasta fiind varianta rapida de executie a tuturor taskurilor, pe care o vom numi „Shortest\_time”. Cealalta strategie consta in numarul de Task-uri pe care il are fiecare server, astfel incat, Task-urile sa se imparta in numar egal fiecarui server. Aceasta metoda neeficienta o vom numi „Shortest\_queue”. Diferenta dintre aceste 2 metode se regaseste la implementare, cea eficienta fiind mai greu de implementat fiind nevoiti sa verificam timpul de asteptare maxim din fiecare server, iar la urma adaugand urmatorul Task in server-ul ales. Pentru a 2-a strategie e nevoie doar de numararea fiecarui task din server si alegerea celui cu numar minim de task-uri. Dupa ce m-am familiarizat putin cu conceptul de baza al problemei, am ales sa-mi construiesc fereastra ce tine de interfata programului. Aici am introdus cate un bloc specific pentru fiecare valoare data de la tastatura. Asadar, un prim exemplu ce tine de design-ul aplicatiei ar fi:

Graphical user interface, application

Description automatically generated

Proiectare

Pe partea de proiectare, m-am folosit de thread-uri si de mecanici de sincronizare astfel incat sa trimit in paralel cu procesul principal, mai multe thread-uri ce reprezinta serverele de taskuri. Din thread-ul principal am initializat server-ul de taskuri si am ales strategia pe care o vom utiliza, urmand sa dam start thread-urilor ce adauga cate un task in ele in functie de strategia aleasa. La fiecare secunda dupa ce un task a fost trimis in server, serviceTime-ul lui se decrementeaza in momentul in care ajunge in fata server-ului. Dupa initializare, fiecare thread are un timp ce este dat de serviceTime-ul task-urilor. Acesta trece in asteptare (sleep) cate o secunda, pana cand serviceTime-ul task-ului ajunge la 0. Cu ajutorul mecanicilor de sincronizare, am reusit sa afisez in ordine mesajele in consola, in fisierul text dar si in fereastra de evenimente. Mai jos este prezenta diagrama UML a proiectului cu legaturile intre clase, cu constructorii respectiv variabilele acestora.

Graphical user interface

Description automatically generated

Implementare

In implementarea aplicatiei am urmat structura MVC (Model-View-Controller) care sa respecte conditiile unei buni organizari a codului, astfel incat sa nu am clase ce depasesc de 300 de linii sau metode ce au mai mult de 30 de linii. Clasele au fost impartite in mai multe pachete: “Exceptions”, pentru fiecare exceptie intalnita; “Main”, pachetul cu clasa main de unde se lanseaza proiect si unde se regaseste clasa de controller ; “Methods”, pachetul unde se afla clasele ce distribuie fiecare task in server si locul unde rezultatul este calculat si afisat in consola pentru teste, acesta urmand sa fie trimis catre controller, pentru a fi afisat si in fereastra de evenimente; “Model”, contine cele 2 clase “Task” si “Server”.

Clasa “Task” continue informatiile pe care le are fiecare task, precum: ID, timeArrival, timeService. Acestea sunt initializate prin setere si apelate prin getere.

Clasa “Server” continue o lista de servere de timp “ LinkedBlockingQueue<Task> ” dar si alte variabile precum: “firstTask”, “waitingPeriodOfSimulation”, ” averageTimeArray ”. Tipul de lista “ LinkedBlockingQueue<Task> ” l-am folosit datorita functiilor pe care mi le ofera. Functia de “.peek()” imi scoate primul element din lista, task-ul care urmeaza a fi citit, iar cu ajutorul functiei “.remove()” sterg Task-ul in momentul in care acesta a fost executat cu success. De fapt, task-ul este sters in momentul cand timpul sau ajunge la 0. Asadar, variabila de tip atomicInteger “waitingPeriodOfSimulation” este initializata cu serviceTime-ul fiecari task in momentul in care acesta este trimis din thread-ul principal in server. Variabila “firstTask” are rolul de a retine primul task din server in momentul in care acesta este executat, urmand sa ca “firstTask” sa retina toate datele referitoare la primul task. Acest lucru ne pune la dispozitie modificarea serviceTime-ului, astfel incat sa putem afisa pe ecran serviceTime-ul actual primului task din server.

Clasa “MainQueue” este clasa in care se afla lansarea programului, dar si clasa in care se afla prima fereastra a interfetei, “Queue”. Dupa pornire, comanda de “launch()” apeleaza clasa ce tine de controller, numita “QueueController”.

Clasa “QueueController” continue toate butoanele si casutele de tip text ce se afla in cele 2 ferestre ale aplicatiei: “Queue”(fereastra de initializare a aplicatiei) si “Log of Events”(fereastra de afisare a rezultatului). Aici am folosit cateva metode specifice precum “onClickStart()”, “clickOnBack” ce au loc numai la actiuni de apasare a butoanele.

Metoda “onClickStart()” la apasarea butonului “START” deshide a 2-a fereastra pentru a afisa rezultatul dorit. Metoda “clickOnBack” din a 2-a fereastra te intoarce la prima pentru a insera alte valori. Metodele “LogOfEventsWindow” si “showAverageTime” au rolul de a salva intr-un string rezultatele la fiecare secunda parcursa. Aceste 2 metode salveaza cele 2 string-uri intr-un text de tip TextArea prin metoda de initializare “initialize(URL url, ResourceBundle rb)”. In metoda “onClickStart”, pe langa crearea unei ferestre noi, avem si instantierea simularii, fiind locul unde se trimit informatiile primate referitoare la serverul de task-uri, prin parametrii constructorului clasei “SimulationManager”.

Clasa “SimulationManager” dupa ce primeste si salveaza informatiile din fereastra “Queue”, trimite prin instantierea clasei “Scheduler”, numarul de servere pe care il avem pentru a le initializa. Dupa acest process, se verifica ce strategie urmeaza a fi folosita, aceasta fiind trimisa tot prin intermediul Clasei “Scheduler”. In final se genereaza random toate task-urile dorite. Metoda “showList” afiseaza lista tuturor task-urilor inserate, Metoda “sortedTaskList” sorteaza in functie de timeArrival, fiecare task din lista. Metoda “showTasks” afiseaza fiecare task lista, dar este folosita numai pentru listele din servere. Ultima metoda din aceasta clasa “showQueueList” afiseaza toate serverele, impreuna cu toate task-urile pe care le au in ele. Functia de “run()” specifica thread-urilor, verifica la fiecare secunda, daca se poate adauga sau nu un task nou in server prin coditia de “currentTime <= maxSimulation” unde currentTime reprezinta timpul fiecarui thread, acesta crescand de fiecare data cu cate o secunda si adaugand task-uri noi in server doar daca timeArrival-ul lor este mai mic sau egal cu timeCurrent. “maxSimulation” reprezinta timpul simularii, moment in care thread-ul se opreste iar odata cu el, tot programul. Pe langa faptul ca se adauga cate un task nou in lista, resultatul se salveaza atat in fisierul text, cat si in fereastra de evenimente. La finalul acestui “while” calculam timpul de asteptat pe care il are fiecare task, calculam suma lor, si impartim la numarul de task-uri generate. Astfel rezultatul final “averageTime” se afiseaza atat in consola cat si in fisierul text si fereastra de evenimente.

Clasa “Scheduler” primeste in constructorul sau, numarul maxim de servere, aceastea fiind trimise catre clasa “Server” pentru a fi initializate. Tot aici, odata cu instantierea fiecarui server, are lor si pornirea a cate un thread pentru fiecare server creat. Thread-ul pleaca in paralel cu thread-ul din clasa “SimulationManager”. Metoda changeStrategy are rolul de a instantia strategia pe care urmeaza sa o folosim, in timp ce metoda “dispatchTask” trimite fiecare task primit catre aceasta.

Clasa “Strategy” continue atat 2 subclase prin relatia de mostenire : “ConcreteStrategyTime” si “ConcreteStrategyQueue” dar si o clasa pentru a enumera strategiile puse la dispozitie. In “ConcreteStrategyTime” afisam rezultatul pentru cazul in care adaugam task-uri noi in server-ul cu cel mai mic timp de asteptare. In “ConcreteStrategyQueue” spre deosebire de cazul precedent, aici adaugam task-ul in server-ul cu cele mai putin task-uri. La ambele strategii ma folosesc de diverse variabile pentru a calcula atat timpul minim, cat si numarul de cozi minim, urmand sa retinem serverul ales, iar la final, sa adaugam in acesta task-ul droit.

In clasa “Server” m-am folosit de o variabila de timp “LinkedBlockingQueue” pentru a salva fiecare task. Aceasta functie pune la dispozitie mai multe comenzi pe principal unei cozi FIFO, precum cea de adaugare add() dar si cea de stergere remove(). O alta comanda importanta este cea de peek(), ce scoate din coada primul element, astfel incat sa putem modifica fiecare task ce ajunge in fata server-ului. Astfel, am folosit variabila “firstTask” pentru a salva primul task. Variabila de tip waitingPeriodOfSimulation de timp atomicInteger salveaza timpul fiecarui task pe care il are de asteptat (serviceTime-ul). Constructorul acestei clase initializeaza fiecare server de tip “linkedBlokingQueue” dar si timpul acestuia de tip “AtomicInteger”. Metoda “addTask()” este implementata cu mecanici de sincronizare si are ca scop adaugarea fiecarui task in server. Metoda “averagePerTime” calculeaza timpul total al task-urilor dintr-un server. Metoda “removeFirstElement()” sterge task-ul de pe prima pozitie. Metoda “showTaskList()” afiseaza task-urile din coada actuala, dar tine cont de timpul primului task, decrementand serviceTime-ul primului task la fiecare apelare a metodei. Metoda “run()” are scopul de a retine primul task din server, si stergerea acestuia in momentul cand timpul sau a ajuns la 0. Acest lucru este posibil datorita functiei de “sleep” a thread-ului care permite decrementarea fiecarui prim task din servere, in momentul in care un task s-a inserat in metoda de “run()” din clasa “SimulationManager”.

Clasa “Task” salveaza fiecare informatie ce tine de client, acestea fiind : ID (numele), arrivalTime(Timpul sosirii), serviceTime (timpul de asteptare individual al task-ului). Acestea sunt salvate si apelate prin sett-re si gett-re.

Rezultate

Legat de rezultate, avem 2 cazuri in care putem vorbi despre ele. Cazul in care se cauta cea mai scurta coada, cazul neeficient, dar si cazul in care se cauta cel mai scurt timp, cazul eficient. Am testat ambele strategii pentru cazurile date ca exemplu, astfel pentru: Tasks = 50, Servers = 5, SimulationTime = 60, minArrivalTime = 2, maxArrivalTime = 40, minServiceTime = 1, maxServiceTime=7 avem un averageTime la “SHORTEST\_TIME” egal cu 4,92, iar la “SHORTEST\_QUEUE” averageTime este egal cu 9,28. Din aceste 2 rezultate avem si concluzia finala, cazul eficient este cel ce cauta cel mai scurt itmp din server.

Concluzii

Aplicatia realizeaza cu succes rezultatul dorit, timpul mediu pe care clientii il asteapta la coada in functie de strategia aleasa. Din aceasta tema am invatat cum sa folosesc thread-urile intr-un program, modul lor cum functioneaza in paralel cu procesul principal. O metoda de imbunatatire a acestei aplicatii o constituie verificarea fiecarei informatii date de la tastatura, in momentul actual aplicatia functionand numai cand primeste numere naturale de timp int unsigned.

Bibliografie

1. <https://dsrl.eu/courses/pt/materials/PT2021-2022_Assignment_2.pdf>
2. <https://dsrl.eu/courses/pt/materials/A2_Support_Presentation.pdf>
3. <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html>
4. <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html>
5. <https://www.w3schools.com/java/java_threads.asp>