DOCUMENTATIE

TEMA 2

QUEUE MANAGEMENT

*Nume student: SZAKACS EMMA-EVELIN*

*Grupa: 30221 An 2 Semestrul 2*

**Cuprins**

[**1.** **Obiectivul temei** 2](#_Toc99277058)

[**2.** **Analiza problemei, modelare, scenarii, cazuri de utilizare** 3](#_Toc99277059)

[**3.** **Proiectare** 4](#_Toc99277060)

[1. Diagrama UML 4](#_Toc99277061)

[3.2 Structuri de date folosite 5](#_Toc99277062)

[**4.** **Implementare** 5](#_Toc99277063)

[4.1.Clase si metode 5](#_Toc99277064)

[4.2. GUI 7](#_Toc99277065)

[**5.** **Rezultate** 8](#_Toc99277066)

[**6.** **Concluzii** 10](#_Toc99277067)

[**7.** **Bibliografie** 10](#_Toc99277068)

# **Obiectivul temei**

**Obiectivul principal** al acestei teme este de a creea o aplicatie care sa gestioneze cozi de clienti astfel incat timpul de asteptare sa fie minim. Scopul unei cozi este de a furniza un loc in care sa astepte clientul inainte de a fi „procesat”. Se vor genera N clienti care si Q cozi, clientii vor astepta pana vor fi serviti, iar dupa ce ajung in coada la care au cel mai putin de asteptat, vor fi serviti. Dupa ce le expira timpul de servire vor fi scosi din coada. Totul se va intampla in cel mai eficient mod.

**Obiectivele secundare** care contribuie la realizarea obiectivului principal sunt:

|  |  |
| --- | --- |
| 1. Analiza si intelegerea problemei. | Capitolul 1 |
| 2. Folosirea conceptelor OOP, cum ar fi incapsularea si definirea unor clase potrivite pentru descompunrea problemei in subprobleme. | Capitolul 3 |
| 3. Implementarea unei clase care va simula aplicatia | Capitolul 4 – Implementare |
| 1. Folosirea eficienta a thred-urilor astfel incat fiecare coada sa aiba un thread. | Capitolul 4 – Implementare |
| 1. Folosirea unei interfete grafice “User Friendly” – folosind Java Swing | Capitolul 4 -Implementare Capitolul 5 - Rezultate |
| 1. Testarea unitara (folosind JUnit) pentru verificarea rezultatelor. | Capitolul 5 - Rezultate |
|  |  |

# [**Analiza problemei, modelare, scenarii, cazuri de utilizare**](file:///C:\Users\Emma\Downloads\PT2021-2022_Documentation_Template.doc#_Toc95297886)

O coada este o structura de date liniara sau o colectie in Java care stocheaza elemente intr-o ordine FIFO(First In, First Out). Cozile sunt utile pentru modelarea unor probleme din lumea reala. Aplicatia ar trebui sa simuleze intr-un timp t o serie de N client care vor intra in Q cozi, care vor astepta, vor fi serviti, iar apoi vor iesi din cozi. Toti clientii vor fi generate in momentul in care incepe simularea si vor fi caracterizati de un id, timp in care ajunge si timp de servire. Aplicatia urmareste timpul total petrecut de fiecare client in coada si calculeaza timpul mediu. Clientii vor fi adaugati in coada cu timpul minim de asteptare in momentul in care timpul lor de sosire va fi egal cu timpul de simulare.

Icon

Description automatically generated

**Cazuri de utilizare**

Diagram

Description automatically generatedActorul principal in acest scenario este utilizatorul. Utilizatorul trebuie sa poata introduce numarul de client, numarul de cozi, timpul de simulare, timpul minim de sosire al clientilor, timpul maxim de sosire al clientilor, timpul minim de servire al clientilor si timpul maxim de servire al clientilor. Utilizatorul apasa pe butonul “START” pentru inceperea simularii. In cazul in care utilizatorul insereaza valori gresite, ar trebui sa poata sa introduca iar datele.

# **Proiectare**

## Diagrama UML

Unified Modeling Language sau UML pe scurt este un limbaj standard pentru descrierea de modele si specificatii pentru software. UML a fost la bază dezvoltat pentru reprezentarea complexității programelor orientate pe obiect, al căror fundament este structurarea programelor pe clase, și instanțele acestora ( numite și obiecte ). Cu toate acestea, datorită eficienței și clarității în reprezentarea unor elemente abstracte, UML este utilizat dincolo de domeniul IT.

Diagrama UML generate pentru proiectul acesta este urmatoarea:

Graphical user interface, text

Description automatically generated with medium confidence

## **Structuri de date folosite**

BlockingQueue reprezinta o structura de coada care pe langa operatiile pe care le suporta o coada, are operatii care asteapta pana cand coada este goala pentru a putea extrage elemente. In plus, asteapta pana cand este spatiu pentru inserarea elementelor, inserandu-le doar in acele momente. Aceste tip de structura de date este folosita pentru a cozi de tipul productor-client, motiv pentru care am ales sa le folosesc in acest proiect.

Cele Q cozi care vor fi creeate vor fi BlockingQueues pentru usurinta implementarii.

In plus, am mai folosit liste obisnuite pentru a retine cei N clienti generatii. Timpul de sosire si timpul de servire al clientilor va fi generat random.

# **Implementare**

## 4.1.Clase si metode

**1.Clasa client**

Aceasta clasa asa cum ii spune si numele este clasa pentru Clienti. Acestea vor avea un numar de identificare, timp de sosire sit imp de servire. Pe langa constructor, gettere si settere, suprascriu metoda toString pentru afisarea unui client.

**2.Interfata Strategy**

Acesta interfata are o singura metoda, cea de adaugare a clientilor. Ea va fi implementata de clasele ConcreteStartegyQueue si ConcreteStrategyTime.

**3.Clasa ConcreteStrategyTime**

Cum spuneam anterior, acesta clasa implementeaza interfata Strategy, astfel trebuie sa contina metoda de adaugare a unui client. Ideea principala a acestei strategi este sa parcurg toate Serverele, adica cozile, si sa gasesc server-ul care are timpul de asteptare minim. Odata ce gasesc id-ul serverului respective, adaug la acel server clientul dat ca parametru.

**4.Clasa ConcreteStategyQueue**

Acesta clasa este similara cu clasa ConcreteStrategyTime, cu diferenta ca se va cauta server-ul care are cei mai putin client, iar in acel server se va adauga clientul dat ca parametru.

**5.Clasa SelectionPolicy**

Enumera cele doua tipuri: shortest\_queue si shortest\_time.

**6.Clasa Server**

Clasa Server implementeaza interfata Runnable. Aceasta interfata este folosita pentru orice clasa care urmeza sa folosesaca un thread. Ca variabile instanta clasa are un BlockingQueue, timpul de asteptare pentru coada, care are tipul AtomicInteger, numarul de identificare a thread-ului, strategia, scheaduler-ul, o variabila booleana pentru a verifica daca coada este deschisa si variabile pentru clientul curent. AtomicInteger-ul joaca un rol important in aplicatie, deoarece este un int care se updateaza automat. In constructor voi seta timpul de asteptare la 0, si voi creea un BlockingQueue de tipul ArrayBlockingQueue cu o capacitate mare astfel incat coada sa nu ajunga sa se umple.

Metoda addClient adauga un nou client in coada si creste timpul de asteptare al cozii respective cu timpul de servire al clientului respectiv. Deoarece implementeaza interfata Runnable, clasa trebuie sa aiba metoda run, fara nici un parametru. Cand un obiect implementeaza interfata Runnable, inceperea unui thread face ca metoda run sa fie apelata separate in acel thread care se executa. In metoda run scot elemental din capul cozii cu metoda take() care il si sterge. Dupa aceea opresc thread-ul pentru un timp egal cu timpul de procesare al clientului. Dupa ce am facut toate astea pot decrementa timpul de asteptare al cozii.

**7.Clasa Scheduler**

Ideea prinipala a acestei clase este sa trimita client la cozi in functie de strategia stabilita. Are ca variabile instanta o lista de servere, o lista de thread-uri si strategia. In constructor primeste ca parametrii numarul maxim de servere si numarul maxim de client. Pentru numarul maxim de servere se vor crea servere si thread-uri cu acele servere. Dupa ce toate sunt create, dau start la thread-uri. Tot in constructor voi alege strategia. Metoda changeStrategy nu face decat sa schimbe strategia. Metodele deactivateServers si activateServers dezactiveaza, respective activeaza toate serverele. Metoda dispatchTask primeste ca parametru un client, iar cu acel client apeleaza metoda addClient din Strategy. In plus, clasa suprascrie metoda toString pentru afisarea continutului cozilor.

**8.Clasa SortByArrivalTime**

Clasa implementeaza interfata Comparator pentru a sorta clientii crescator dupa timpul de sosire.

**9.Clasa SimulationManager**

Aceasta clasa este cea care simuleaza intreaga aplicatie, legand clasele intre ele. Aici se vor salva datele din interfata, adica timpul de simulare, timpul minim de servire, timpul maxim de servire, timpul minim de sosiire, timpul maxim de sosire, numarul de client si numarul de cozi. In constructor se creeaza o instanta la interfata grafica. Tot aici am pus actionListener pentru butonul de start. In momentul in care utilizatorul apasa butonul de start, se vor lua datele introduse in interfata, se vor genera clientii si se va creea thread-ul principal la care se da start.

In metoda generateNRandomClients, cum am spus si anterior, se genereaza n clienti a caror timp de sosire si timp de asteptare va fi generat random. Dupa ce acestia au fost generati, se vor sorta crescator dupa timpul de sosire.

Metoda cheakIfPeakTime gaseste timpul la care au fost cei mai multi clienti intr-o coada.

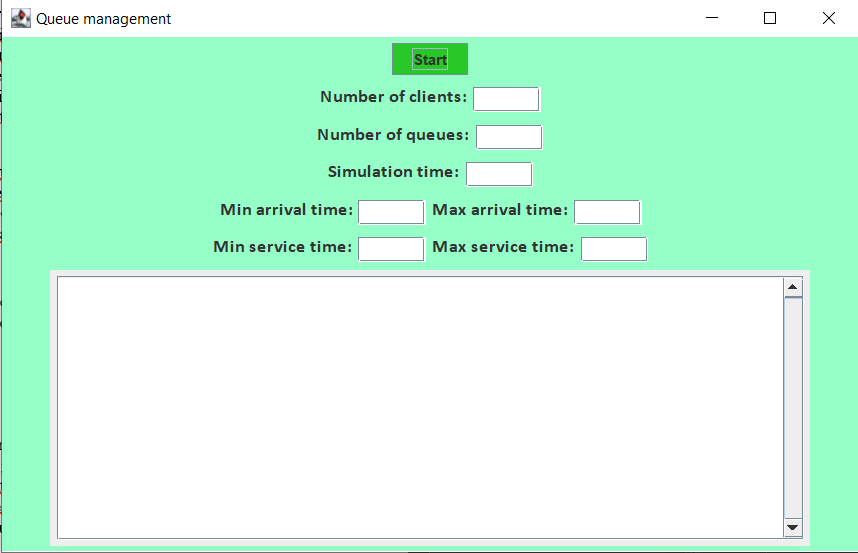
Clasa acesta implementeaza, la fel ca si clasa Server interfata Runnable. Astfel, are metoda run. Inainte de toate initializez scheaduler-ur. Initializez timpul curent la 0, iar daca timpul curent este mai mic decat timpul de simulare se verifica daca este vreun client cu timpul de sosire egal cu timpul curent, iar daca da, se apeleaza metoda dispatchTask din scheaduler, se creste timpul total de servire si se sterge clientul din lista, iar daca nu, se trece la verificarea urmatorului client. Dup ace toate astea au fost realizate, se afiseaza rezulltatul, iar apoi se creste timpul curent. La final se apeleaza metoda pentru dezactivarea serverelor. Average Service Time va fi totalServiceTime impartit la numarul de client.

## 4.2. GUI

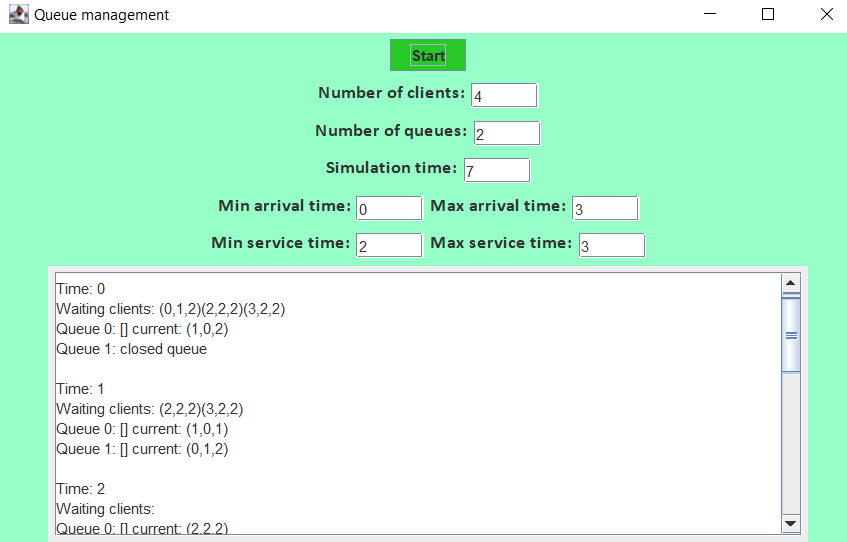
Intergafa grafica am implementat-o in clasa View care extinde clasa Frame. Inainte de toate am dat ca variabile instanta toate butoanele care vor aparea in interfata, textField-urile, JFrame-ul, label-urile si panel-urile. In partea de constructor apelez toate metodele pentru implementarea interfetei. Am impartit interfata in trei panel-uri. Am folosit ca si layout un BorderLayout. La inceput ama asezat panel-ul pentru butonul de start, in centru panel-ul pentru input, iar in partea de jos panel-ul pentru input. Panel-ul pentru input contine cele 7 textField-ri cu label-urile corespunzatoare. Panel-ul pentru butoane contine doar butonul de start, iar panel-ul pentru output contine scroll panel pentru afisarea in timp real a evolutiei cozilor.

O metoda suplimentara pe care am implementat-o pentru imbunatatirea aspectului interfetei este mouseMove. Metoda apeleaza o alta metoda mouseMoveOneButton pentru fiecare buton. mouseMoveOneButton adauga un MouseListener butonului respectiv. Cand mouse-ul este pe buton, I se schimba culoarea butonului, iar cand mouse-ul este mutat, culoarea butonului revine la culoarea initiala.

In plus, in clasa View am pus setter pentru butoanul de start pe care il folosesc in clasa SimulationManeger asttfel incat simularea sa inceapa doar in momentul in care utilizatorul a apasat butonul de start si au fost preluate cu succes datele.



Un exemplu in care se poate vedea cum se afiseaza evolutia cozilor este urmatorul:



# **Rezultate**

Pentru testare am folosit mai multe date de intrare, inclusiv cele specificate in cerinta.

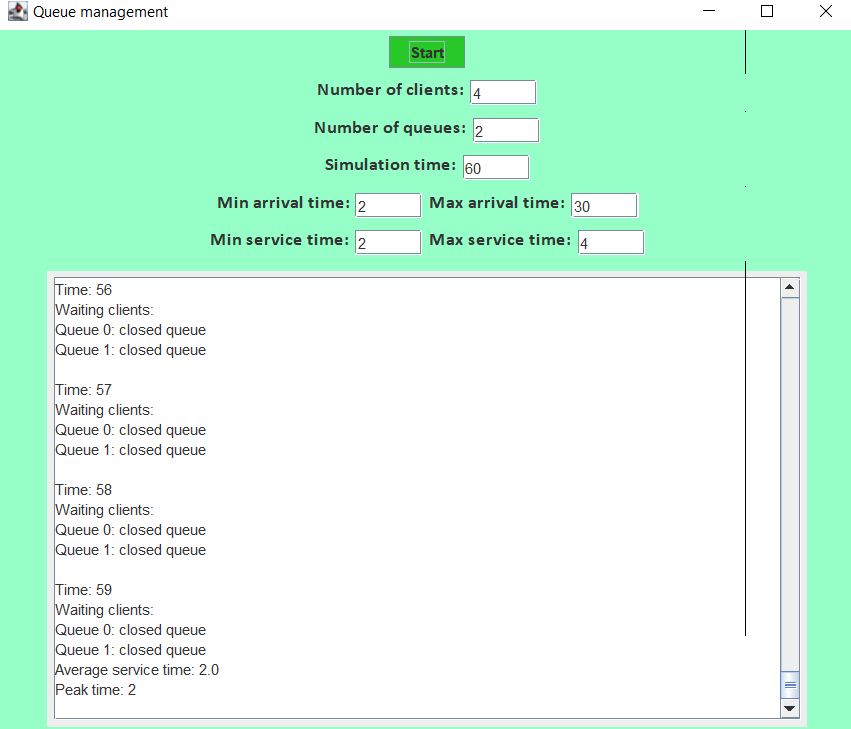
Testul 1: N = 4

Q = 2

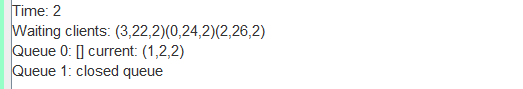
𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛 𝑀𝐴𝑋 = 60 seconds

[𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 𝑀𝐼𝑁, 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 𝑀𝐴𝑋] = [2, 30]

[𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 𝑀𝐼𝑁, 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 𝑀𝐴𝑋] = [2, 4]



Dupa cum se poate observa simularea s-a finalizat cu success dupa 60 de seecunde. La final s-au afisat average service time si peak time. Timpul mediu petrecut de un client la coada(average service time) este de 2 secunde, iar momentul in care coada a fost cea mai plina (peak time) este la timpul 2. In exemplul acesta nu vor fi mai multi clienti la aceasi coada la acelasi moment de timp.



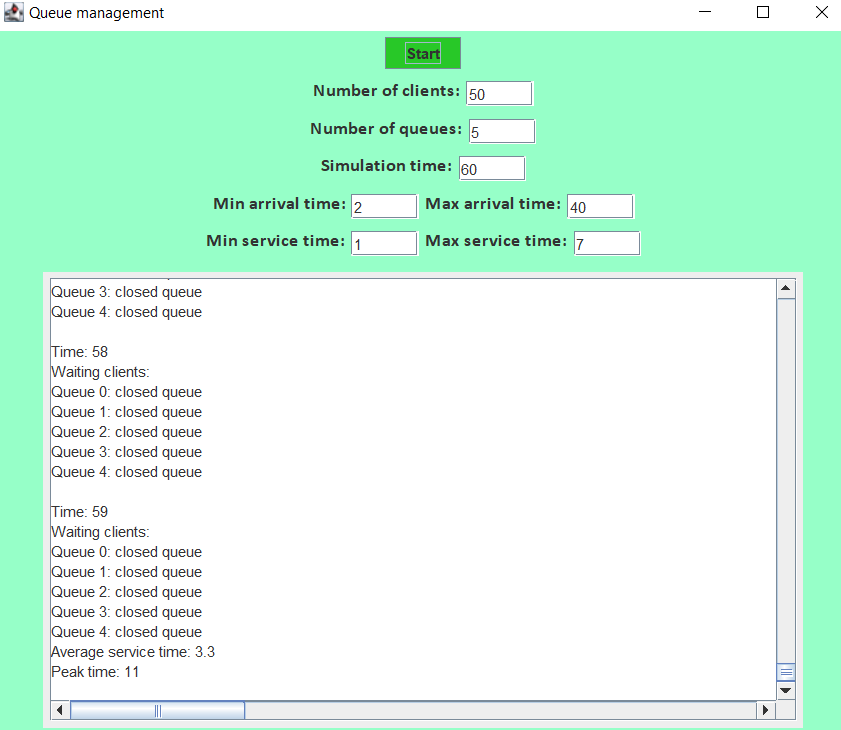
Testul 2: N = 50

Q = 5

t𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛 𝑀𝐴𝑋 = 60 seconds

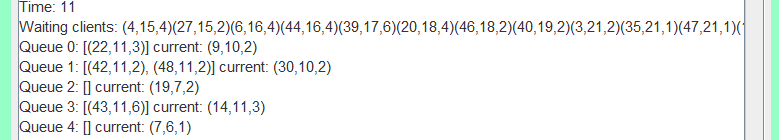
[𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 𝑀𝐼𝑁, 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 𝑀𝐴𝑋] = [2, 40]

[𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 𝑀𝐼𝑁, 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 𝑀𝐴𝑋] = [1, 7]



Pentru acest test se poate observa ca timpul de asteptare mediu si peak time-ul este mai mare decat la testul anterior.

La timpul 11 sunt cele mai aglomerate cozile, aici fiind peak time-ul cel mai mare. Timpul mediu de asteptare este de 3.3.



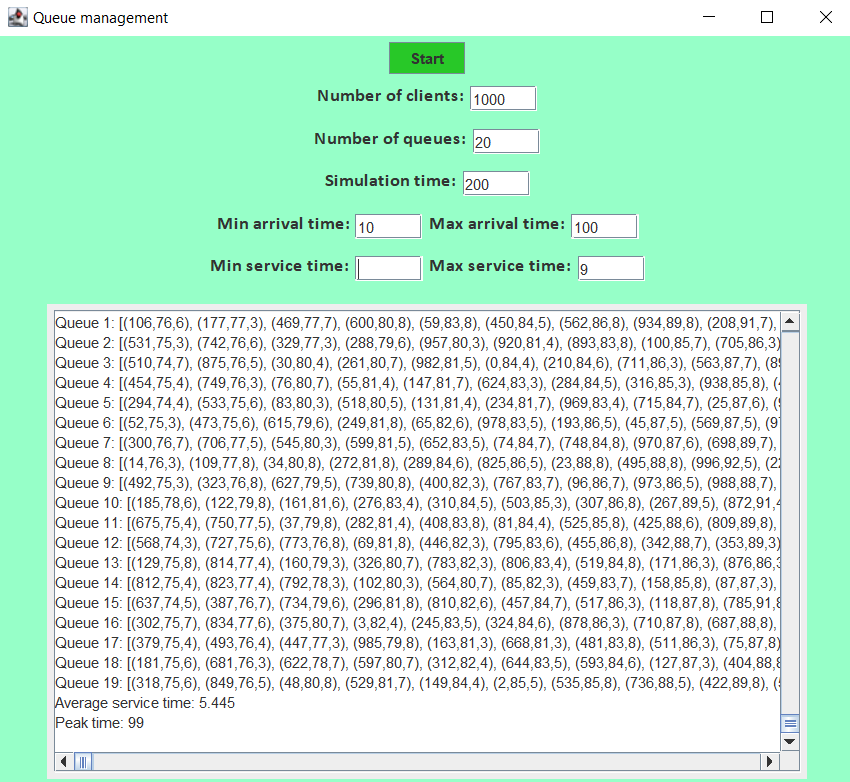
Testul 3: N = 1000

Q = 20

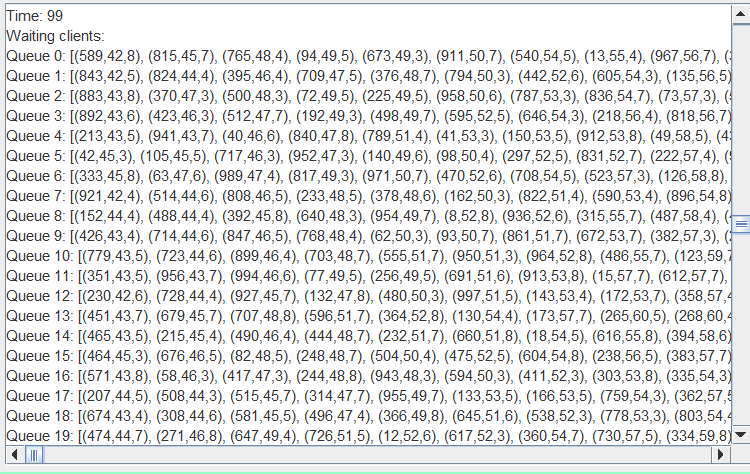
𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛 𝑀𝐴𝑋 = 200 seconds

[𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 𝑀𝐼𝑁, 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 𝑀𝐴𝑋] = [10, 100]

[𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 𝑀𝐼𝑁, 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 𝑀𝐴𝑋] = [3, 9]



Dupa cum se poate observa, average service time si peak time au crescut semnificativ. Pentru un numar de 1000 de clienti distribuiti la 20 de cozi, average service time este de 5.445, iar peak time este 99. La timpul 99 au fost cele mai pline cozile. Se poate observa mai jos:



# **Concluzii**

Prin urmare, acest proiect mi-a adus la cunostinta noi concepe din limbajul de programare Java. Realizarea lui a necesitat documentare foarte amanuntita si lunga deoarece desi conceptul de thread-uri nu este nou pentru mine, este prima data cand chiar le utilizez la ceva. Am invatat mai multe concepte pe langa cel de thread-uri, cum ar fi diferenta dintre Integer, volatile si AtomicInteger, diferenta dintre Queue si BlockingQueue.

Din punctul de vedere al dezvoltarilor ulterioare, se pot face multe legat de interfata grafica. Eu am facut o interfata foarte simpla, dar usor de urmarit. Utilizatorul doar introduce datele si apasa pe butonul start. S-ar fi putut face coloane pentru fiecare queue, inclusiv pentru coada cu clientii in asteptare. De fiacre data cand se adauga un client in coada sa se schimbe ceva legat de coada respectiva, cum ar fi culoarea. Din punctul de vedere al implementarii s-ar putea adauga un text field in interfata grafica pentru a cauta la un moment introdus de utilizator situatia cozilor. De exemplu, in ultimul test in care am avut timpul de simulare 200, am obtinut peak time-ul 99. Pentru utilizator ar fi mult mai usor sa poata cauta cum arata cozile la timpul respectiv.

# **Bibliografie**

* Java Swing: <https://docs.oracle.com/javase/tutorial/uiswing/>
* [1] Cursul de OOP din semestrul 1
* [2]<https://docs.oracle.com/javase/7/docs/api/overview-summary.html>
* [3] https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
* [4] <https://www.baeldung.com/java-synchronized-collections>
* [5] <https://www.baeldung.com/java-synchronized>
* [6] <https://www.baeldung.com/java-runnable-callable>
* [7] <https://docs.oracle.com/javase/8/docs/api/java/util/Timer.html>