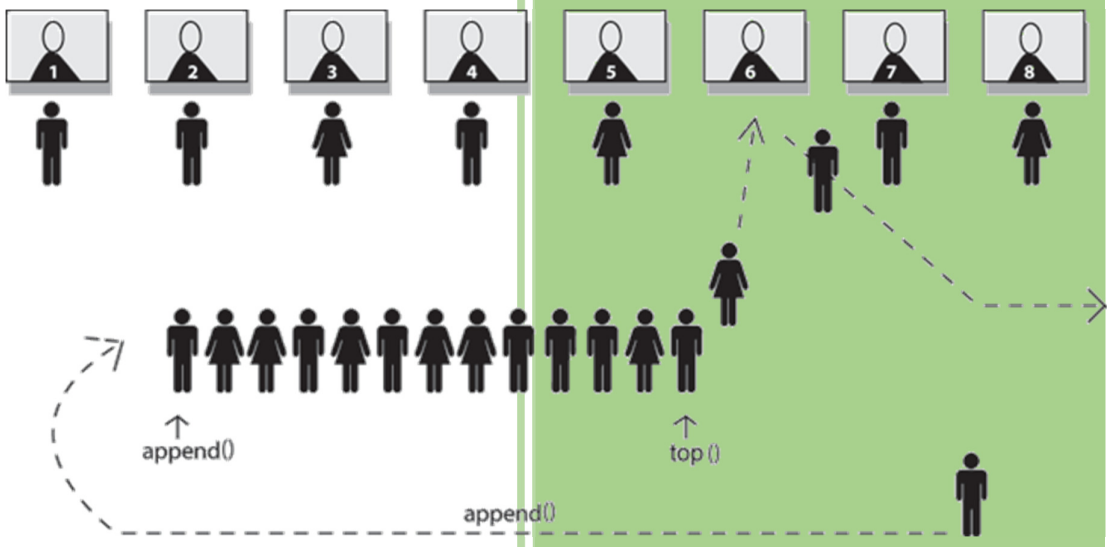
Queuing theory and efficient simulation

O. Senhadji El Rhazi

2013

Queuing theory and efficient simulation



O. Senhadji El Rhazi

Multi-Task Optimisation

Table of Contents

Introduction

In many life situations, there exist queues with different behaviours. The purpose of this paper is to model those queues. These queues are prevalent in several areas. Not only, in the daily life in the administrative branches, or in the banks, but also in computer science and in telephone networks. The sharing of resources via a server, and the stream of commands to the level of the microprocessor or the management of telephone calls are all models for the queue. The present paper is divided into three parts: a basic theory part, an implementation part and a development of the code which will enable us to get some results.

The simulation are based on the following model:

- The durations between two successive arrivals are independent random variables of the same law: exponential of parameter λ(ε) that will be designated by M (for Markov).
- The time of service at the counters are independent variables and of same law: This law is taken exponential in the first place and bimodal in second place noted G. The bimodal case allows to model the possibility of lengthy queries and short ones.
- A single server will be taken.
- Different strategies of service of customers in the queue will be addressed:

  - ✓ First in first out (FIFO) the more natural one!
  - ✓ LIFO (last in, first out).
  - ✓ SIRO (service in random order) or the next customer is taken randomly in the queue,
  - ✓ PS (Processor sharing)
  - ✓ LIFOpreemp (LIFO pre-emptive strike) or the current service is interrupted to immediately take in the new customer arrived.

Simulation Model

To simulate the chains M/M/1 and M/G/1, or the two notation that have been explained previously, we must simulate two laws:

1.1 The exponential law ε(λ)

The exponential law of a parameter λ is simulated by the *dual function va1 (double l)*. We used to this end the result of probability. U being a random variable of uniform law on [0, 1]

$$\frac{-Ln(U)}{\lambda} \sim \mathcal{E}(\lambda)$$

1.2 The bimodal law

A random variable following this law takes two kinds of values, to simulate periods of short and long past over the counter according to the client's request. The density of this law has been taken equal to:

$$p(t) = \alpha \cdot (\lambda exp(-\lambda t) \cdot \mathbf{1}_{[0,s]} + \lambda exp(-\lambda(t-s)))$$

Where α is a normalization constant, λ is the parameter of the exponential and s represents an order of magnitude of the lengthy queries (requete_longue).

The function that simulates this law is *double va2 (double l)*. To simulate this law, the following theoretical result was used:

$$F(t) = \int_0^t p(x)dx$$
$$F(X) \sim U \ , \ F^{-1}(U) \sim X$$

- X random variable density equal to p
- U random variable of uniform law on [0, 1]

The following figures represented the histograms generated by these two functions va1 and va2. Note the conformity of the histogram with the exponential laws of va1 and bimodal of va2.

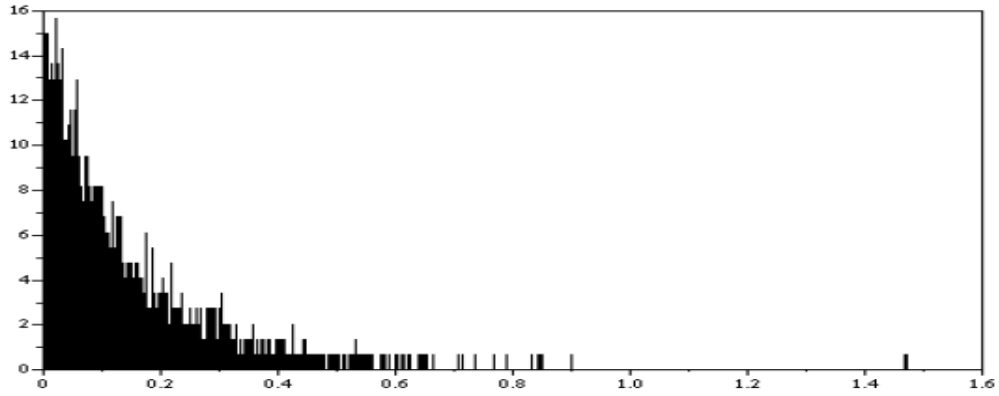


Fig 1 – histogram of function var1 (10000 iterations, λ = 1)

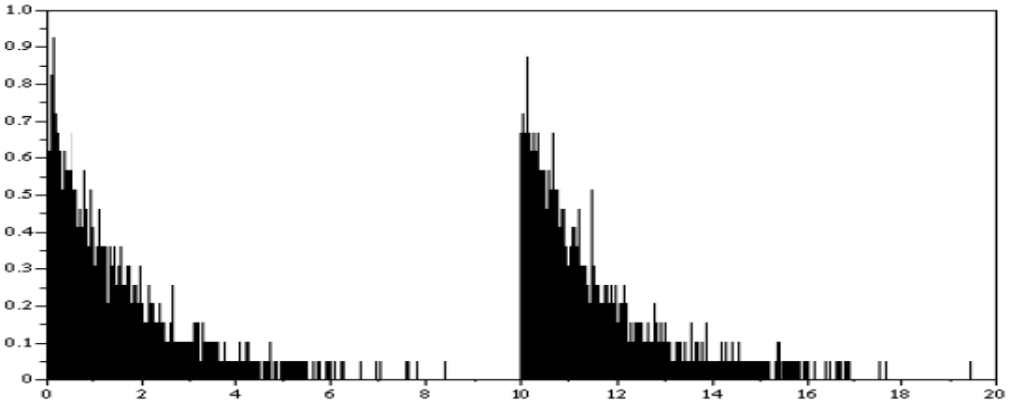


Fig 2 – histogram of function var2 (10000 iterations, λ = 1, s = 10)

Coding Approach

We recall that the strategies involved to designate what client serve -and when (LIFO Preemp)- among the crowd of customers in the queue.

2.1 FIFO (First In First Out)

To implement the **FIFO** strategy a stack **INTVECTOR st** is taken into account. This stack contains (at each moment) all clients who are in the queue except the one who is at the counter. The number j is designated by the variable **ds_guichet**.

Each time a new client n arrives in the queue, its number -his rank in the queue, the customers being stowed in an ascending order of their time of arrival- is added to the end of the queue by the instruction **st.push_back(n)**. The empty variable is then set to 0 *empty = 0* to indicate that the stack is not empty. Each time a client leaves the queue, the top customer in the queue can then be directed into the counter-strategy **FIFO** which expressed by the following statement:

- **j=st.front() ;** the highest element in the stack
- **st.erase(begin()) ;** the client leaves the stack
- **ds_guichet=j ;** the variable **ds_guichet** is redefined

2.2 LIFO (Last In, First Out)

The same variables that previously were taken with the same definitions are also used here. Each time a new client n arrives in the queue, its number is added to the end of the queue by the instruction **st.push_back(n)**. The empty variable is then set to 0 *empty = 0* to indicate that the stack is not empty. Each time a client leaves the queue, the last customer in the queue can then be directed into the counter-strategy **LIFO** which expressed by the following statement:

- **j=st.back( );** the lowest element in the stack
- **st.erase(st.end( ))-1 ;** the client leaves the stack, the last element is deleted
- **ds_guichet=j ;** the variable **ds_guichet** is redefined

2.3 SIRO (Service In Random Order)

The same variables that previously are taken with the same definitions are also used here. This strategy involves a new function void **shaker (INTVECTOR &s)** whose role is to randomly swap the elements of the queue. The top item in the stack will be random. In each new arrival, the newcomer is added at the end of the queue by the instruction **st.push_back(n)**. Similarly, as previously, the empty variable is set to 0. In each departure of a customer, when his treatment is completed, a new customer will then randomly taken -strategy **SIRO**- in the queue to go to the ticket counter. This is expressed by these commands:

- **shaker(st) ;** random permutation of the elements of the stack
- **j=st.front() ;** the highest element of the stack is taken
- **st.erase(st.begin()) ;** this element is removed from the stack
- **ds_guichet=j ;** the variable ds_guichet is redefined

2.4 Ps (Processor Sharing)

This strategy is a bit unusual in the sense that all requests are processed at the same time. The explanation of this strategy encroaches on the algorithmic part of the next part which will therefore be more concise.

The variables used are:

- A table **int st[N]** containing customers
- **Size** variable to measure the number of customers in the queue
- A table **RESTE_A_TRAITER** containing the wait times
- Remaining for each clients; the queries being processed at the same time.
- **Next** to designate what client exit the queue.

In each arrival of a new client, its number is stored in **st st[size] =n**, the size of the queue increases, **size=size 1**, the time remaining to him is given by **RESTE_A_TRAITER[n] =Va2 (mu)**. The empty variable is, as previously, upgrade to 0 *empty = 0*. The **next** customer who will leave the queue is the one whose box **RESTE_A_TRAITER[n]** is minimal. Moreover, the variable next is such that **RESTE_A_TRAITER[next]** is minimum. In each departure of a customer, the element **RESTE_A_TRAITER[next]** is crushed by **RESTE_A_TRAITER[size]** and the variable size is decremented **size= size -1 ;**. the variable next is then recalculated according to the same principle.

2.5 LIFOpremp (LIFO pre-emptive strike)

The principle of strategy is to serve immediately the newcomer by interrupting the processing of the current request.

The variables used are:

- **INTVECTOR st** (the same defined previously for **FIFO**)
- A table **RESTE_A_TRAITER**
- A varibale **ds_guichet** (the same defined previously)

In each arrival of a new client, the one at the ticket counter is sent to the queue **st.push_back(ds_guichet)**. The remaining time to satisfy his request is logged **RESTE_A_TRAITER[ds_guichet]**. The variable **ds_guichet** is initialized to n the number of the last client arrival. Finally *empty = 0* to indicate that the stack is not empty.

In each departure of a customer, the last client returned sees the treatment of its query resume which expressed by:

- **j=st.back( );** the lowest element in the stack, the last returned
- **st.erase(st.end( )-1);** the client leaves the stack
- **ds_guichet=j ;** the variable ds_guichet is redefined

The Algorithm

The purpose of this algorithm is to return a history of events which take place between the moment 0 of departure and a time T corresponding to the passage of N client by the service. The events are, the arrivals and departures of the customers. The history must include the

moment of arrival, departure, the waiting time in the queue before the beginning of the processing of the request and the duration of treatment for each client. These data will allow then to calculate averages and make comments on each type of queue characterized by its strategy and its laws.

In practice, the historical data are taken after a moment corresponding to the passage of 1000 clients since the initial moment and before the moment after which will 100 in the queue. This is to help to stabilize the system in a stationary speed. These two moments are expressed respectively by the variables **heats** and **cools**. The historical data are then stored in a file **res.txt**. The history is defined by four tables:

- **Arrival** to store the moment of arrival of each client.
- **DUREE_TRAITEMENT** to store the duration of the processing of the query.
- **OUTPUT** to store the time of output of the customers at the counter.
- **TREATMENT** to store the moment of the beginning of treatment, this variable is unused for the queues **PS** and **LIFOpreemp**.

The flow time is simulated by three variables:

- **time** the current time since the first arrival of a customer.
- **t** the time between the moment current (time) of the arrival of the next customer.
- **u** the time between the moment the current departure next to a client.

The variable t follows, throughout the algorithm, an exponential law. On the other hand u follows, in the first case, an exponential law and, in the second case, a bimodal law. These two variables therefore indicate the closest event. In effect $t \leq u$ designates an arrival then that $t \geq u$ designates a departure. The time variable **time** is accordingly corrected for each event, it is in other words to a sum of several t and u scenarios.

If the nearest event is an arrival of a customer n then:

- The time variable is corrected: **time = time t ;**.
- By definition of u and t: **u = u -t ;**. **t= va1 (λ);.**
- The moment of arrival is recorded in the table **ARRIVAL: ARRIVAL[n] =time ;.**

If the event the nearest is an output from a client n then:

- The time variable is corrected: **time = time u ;**.
- By definition of t: **t = t - u ;**.
- The moment of departure is in the table **OUTPUT**: **OUTPUT[ds_guichet] =time ;**

The next customer j -if the queue is not empty- is taken according to the logic and the algorithm explained in the previous section:

- By definition of u: **u= va2 (mu)**.
- **DUREE_TRAITEMENT[j]= u ;**.
- If the queue is empty **empty = 1 ;**.

If the queue is empty the choice step (previous section) is skipped. It goes directly to the next arrival and the customer is directly supported.

C/C++ Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <math.h>
#include <malloc.h>
#include <string.h>
#include <conio.h>
#include <time.h>
#include <vector>
using namespace std;
typedef vector<int> INTVECTOR;
//*********************************
//PARAMETRAGE: M/M/1 ou M/G/1 ? =0 si la loi des temps de service est exponentielle (M); =1 si elle est bimodale (G):
#define BIMODALE 0
// Stratégie d'utilisation ? (Une seule valeur à 1 !)
#define FIFO 1 //First In, First Out
#define LIFO 0 //Last In, First Out
#define SIRO 0 //Service In Random Order
#define PS 0 //Processor Sharing
#define LIFOpreemp 0 //LIFO préemptive
//(interruption de service et prise en charge immédiate du cl arrivé) Fichier de sortie ?
#define fichier "res.txt"
// Nbre de clients pour faire chauffer le système et atteindre l'état stationnaire
#define chauffe 1000
// Nbre de clients à ne pas prendre en compte à la fin(car non sortis du système)
#define refroidit 100
// Paramètres ?
#define N 10000
#define lambda 0.9
// mu > lambda, sinon la file explose !
#define mu 1.0
//paramètre de la loi bimodale
#define requete_longue 20.0
//********************************* variables globales d'enregistrement :
double ARRIVEE[N+1];
#if PS || LIFOpreemp
#else
double TRAITEMENT[N+1];
#endif
double DUREE_TRAITEMENT[N+1];
double SORTIE[N+1];
//NB : pour simplifier la gestion d'indices, on n'utilisera pas la case 0 de ces tableaux.*********************************
// Les fonctions de tirages aléatoires : Fonction de tirage aléatoire pour les temps d'arrivée entre deux clients (M) :
double va1(double l) {
return -log(double(rand())/RAND_MAX)*(1/l);
}
// Fonction de tirage aléatoire pour les temps de service :
#if BIMODALE
double va2(double l) {
const double alpha=1/(2-exp(-l*requete_longue));
const double interm= alpha*(1-exp(-l*requete_longue));
double U=double(rand())/RAND_MAX;
if (U<interm) return ((-1/l)*log(1-U/alpha));
return

((-1/l)*log((2-exp(-l*requete_longue))-U/alpha)+ requete_longue);
}
#else
double va2(double l)
{
return -log(double(rand())/RAND_MAX)*(1/l);
}
#endif
//Fonction auxiliaire
void shaker(INTVECTOR &s) {
int temp, val, incr;
for (incr=0 ; incr < s.size() ; incr++)
{
temp = int(double(rand())/RAND_MAX*s.size());
if (temp!=incr) {
val = s[temp];
s[temp] = s[incr];
s[incr] = val;
}}}
//*********************************
void main(void) {
//*********************************
// Initialisation
srand(time(0));

// numéro du prochain client qui arrive.
int n = 1;
// =1 si la file d'attente (hors guichet) est vide ; =0 sinon.
int vide = 1;
// t est la durée qu'il reste d'ici à l'arrivée du prochain client.
double t;
// u est la durée qu'il reste d'ici à la sortie du prochain client.
double u;
// temps de la chronologie "absolue".
double temps;
// numéro du client courant du guichet.
int ds_guichet;
// variables temporaires (incréments...)
int i, j;
FILE* fp;
// On peut supposer que la chronologie débute avec l'arrivée du 1er client ; celui-ci est immédiatement traité.
temps = 0;
u = va2(mu);
ds_guichet = n;
ARRIVEE[n] = temps;
#if PS || LIFOpreemp
#else
TRAITEMENT[n] = temps;
#endif
DUREE_TRAITEMENT[n] = u;
// On ne touche donc pas à la pile, qui reste vide. On tire t pour le prochain client.
t = va1(lambda);
n = n+1;
//*********************************
// Début de l'algorithme : Seule la partie FIFO est commentée. Le reste fonctionne dans la même logique.
#if FIFO
// Création de la pile de type FIFO.
INTVECTOR st;
while (n < N) {
/*Logique de l'algo: On étudie QUEL événement surviendra en premier dans la suite de la chronologie. Le temps courant est défini par la variable temps ; cette variable est mise à jour au fur et à mesure qu'on avance dans l'algorithme. Dans chaque section correspondant à l'événement qui survient, on se place au moment du dit événement et on met à jour les variables. Les événements - et leur "portée" - sont choisis de telle manière que t et u soient bien définis à chaque fois qu'on revient au début de la boucle while. En particulier, l'état "guichet vide", qui est temporaire puisque se terminant à l'arrivée du client suivant, est traité au sein de la section de l'événement 3.*/
if (t<=u)
{
// Le prochain événement est l'arrivée d'un client.
temps = temps+t; u = u-t;
ARRIVEE[n] = temps;
st.push_back(n);
t = va1(lambda);
```

```
n = n+1;
vide = 0;
}
else {
// Le prochain événement est la sortie d'un client.
if (vide==0){
temps = temps+u; t = t-u;
SORTIE[ds_guichet] = temps;
j = st.front(); st.erase(st.begin());
ds_guichet = j;
TRAITEMENT[j] = temps;
u = va2(mu);
DUREE_TRAITEMENT[j] = u;
if (st.size()==0) vide = 1;
}
else {
temps = temps+u; t = t-u;
SORTIE[ds_guichet] = temps;
//La file est vide, le guichet attend l'arrivée du prochain client.
temps = temps+t;
ARRIVEE[n] = temps;
//Le client est tout de suite pris en charge.
u = va2(mu); ds_guichet = n;
TRAITEMENT[n] = temps;
DUREE_TRAITEMENT[n] = u;
//On attend le prochain client.
t = va1(lambda);
n = n+1;
} } } //fin
//*******************************
// Fonctions d'écriture dans un fichier
// des variables globales d'enregistrement.
fp = fopen(fichier,"w");
for (i=chauffe;i<=N-refroidit;i++)
fprintf(fp,"%d\t%8f\t%8f\t%8f\t%8f\n",
i, ARRIVEE[i], TRAITEMENT[i], DUREE_TRAITEMENT[i],
SORTIE[i]);
fclose(fp);
#endif
#if LIFO
// Création de la pile de type LIFO.

INTVECTOR st;
while (n < N) {
if (t<=u) {
// Le prochain événement est
//l'arrivée d'un client.
temps = temps+t; u = u-t;
ARRIVEE[n] = temps;
st.push_back(n);
t = va1(lambda);
n = n+1;
vide = 0;
}
else {
// Le prochain événement est la sortie d'un client.
if (vide==0){
temps = temps+u; t = t-u;
SORTIE[ds_guichet] = temps;
j = st.back(); st.erase(st.end()-1);
ds_guichet = j;
TRAITEMENT[j] = temps;
u = va2(mu);
DUREE_TRAITEMENT[j] = u;
if (st.size()==0) vide = 1;
}
else {
temps = temps+u; t = t-u;
SORTIE[ds_guichet] = temps;
//La file est vide, le guichet attend
//l'arrivée du prochain client.
temps = temps+t;
ARRIVEE[n] = temps;
//Le client est tout de suite pris en charge.
u = va2(mu); ds_guichet = n;
TRAITEMENT[n] = temps;
DUREE_TRAITEMENT[n] = u;
//On attend le prochain client.
t = va1(lambda);
n = n+1;
} } } //fin
//*******************************
// Fonctions d'écriture dans un fichier des variables globales d'enregistrement.
fp = fopen(fichier,"w");
for (i=chauffe;i<=N-refroidit;i++)
fprintf(fp,"%d\t%8f\t%8f\t%8f\t%8f\n", i,
ARRIVEE[i], TRAITEMENT[i],
DUREE_TRAITEMENT[i], SORTIE[i]);
fclose(fp);
#endif
#if SIRO
// Création de l'ensemble st (pas d'ordre particulier).
INTVECTOR st;
while (n < N) {
if (t<=u) {
// Le prochain événement est l'arrivée d'un client.
temps = temps+t; u = u-t;
ARRIVEE[n] = temps;
// n est placé dans le set st.
st.push_back(n);
t = va1(lambda);
n = n+1;
vide = 0;
}
else {
// Le prochain événement est la sortie d'un client.
if (vide==0){
temps = temps+u; t = t-u;
SORTIE[ds_guichet] = temps;
// On mélange bien st, pour que le tirage soit aléatoire...
shaker(st);
j = st.front(); st.erase(st.begin());
ds_guichet = j; // j est pris aléatoirement dans le set st.
TRAITEMENT[j] = temps;
u = va2(mu);
DUREE_TRAITEMENT[j] = u;
if (st.size()==0) vide = 1;
}
else {
temps = temps+u; t = t-u;
SORTIE[ds_guichet] = temps;
//La file est vide, le guichet attend l'arrivée du prochain client.
temps = temps+t;
ARRIVEE[n] = temps;
//Le client est tout de suite pris en charge.
u = va2(mu); ds_guichet = n;
TRAITEMENT[n] = temps;
DUREE_TRAITEMENT[n] = u;
//On attend le prochain client.
t = va1(lambda);
n = n+1;
} } } //fin
//*******************************
// Fonctions d'écriture dans un fichier des variables globales d'enregistrement.
fp = fopen(fichier,"w");
for (i=chauffe;i<=N-refroidit;i++)
fprintf(fp,"%d\t%8f\t%8f\t%8f\t%8f\n", i
, ARRIVEE[i], TRAITEMENT[i], DUREE_TRAITEMENT[i]
, SORTIE[i]);
fclose(fp);
#endif
#if PS
// Il n'a plus de guichet, tous sont traités en même temps, avec une vitesse moindre. Dans ces conditions, u désigne la durée d'ici à la sortie du premier client dont on aura fini le traitement. Création du tableau contenant la file;la file ne
```

```
peut pas être plus grande que le nombre max de personnes
qui arrivent !
int st[N];
int taille = 1; //Taille de la file
//(initialement = 1)
// On crée un nouveau tableau.
int RESTE_A_TRAITER[N+1];
int prochain; // indice dans st du prochain
// client à voir se terminer son service.
j = 0;
// pour éviter le warning (j ne sert à rien ds la suite de PS).
st[0] = 1;
RESTE_A_TRAITER[1] = u;
while (n < N) {
if (t<=u) {
// Le prochain événement est l'arrivée d'un client.
temps = temps+t;
// On met à jour les temps qui restent pour le traitement des
clients dans st.
for (i=0;i<taille;i++)
RESTE_A_TRAITER[st[i]] =
RESTE_A_TRAITER[st[i]] - t/taille;
ARRIVEE[n] = temps;
DUREE_TRAITEMENT[n] = va2(mu);
RESTE_A_TRAITER[n] = DUREE_TRAITEMENT[n];
st[taille] = n;
taille = taille+1;
u = RESTE_A_TRAITER[st[0]]*taille;
// ici, prochain stocke l'indice de l'élément de st qui réalise
le minimum.
prochain=0;
for (i=1;i<taille;i++) {
if (RESTE_A_TRAITER[st[i]]*taille<u) {
u = RESTE_A_TRAITER[st[i]]*taille;
prochain = i;
}}
t = va1(lambda);
n = n+1;
vide = 0;
}
else {
// Le prochain événement est la sortie d'un client.
if (vide==0){
temps = temps+u; t = t-u;
for (i=0;i<taille;i++)
RESTE_A_TRAITER[st[i]] =
RESTE_A_TRAITER[st[i]] - u/taille;
SORTIE[st[prochain]] = temps;
// Réorganisation du tableau :
if (prochain==taille-1) {
//prochain est le dernier élt du tableau.
taille = taille - 1;
}
else {
st[prochain] = st[taille-1];
taille = taille - 1;
}
if (taille == 1) vide = 1;
u = RESTE_A_TRAITER[st[0]]*taille;
// ici, prochain stocke l'indice de l'élément de st qui réalise
le minimum.
prochain=0;
for (i=1;i<taille;i++) {
if (RESTE_A_TRAITER[st[i]]*taille<u) {
u = RESTE_A_TRAITER[st[i]]*taille;
prochain = i;
}}}
else {
// ici, vide=1 donc taille=1 et u<t, ie l'unique client va sortir.
prochain=0 normalement.
temps = temps+u; t = t-u;
SORTIE[st[prochain]] = temps;
//La file est vide ; on attend le prochain client
temps = temps+t;
ARRIVEE[n] = temps;
//Le client est tout de suite pris en charge.
u = va2(mu);
RESTE_A_TRAITER[n] = u;
DUREE_TRAITEMENT[n] = u;
//Introduction dans la file (dans le tableau) :
taille = 1;
st[0]=n;
//On attend le prochain client.
t = va1(lambda);
n = n+1;
} } } //fin
//*******************************
// Fonctions d'écriture dans un fichier
// des variables globales d'enregistrement.
fp = fopen(fichier,"w");
for (i=chauffe;i<=N-refroidit;i++)
fprintf(fp,"%d\t%8f\t%8f\t%8f\n", i,
ARRIVEE[i], DUREE_TRAITEMENT[i], SORTIE[i]);
fclose(fp);
#endif
#if LIFOpreemp
INTVECTOR st; // Création de la pile de type FIFO.
int RESTE_A_TRAITER[N+1]; // On crée un nouveau
tableau.
while (n < N) {
if (t<=u) {
// Le prochain événement est l'arrivée d'un client.
temps = temps+t; u = u-t;
RESTE_A_TRAITER[ds_guichet] = u;

st.push_back(ds_guichet);
ds_guichet = n;
u = va2(mu);
ARRIVEE[n] = temps;
DUREE_TRAITEMENT[n] = u;
t = va1(lambda);
n = n+1;
vide = 0;
}
else {
// Le prochain événement est la sortie
// d'un client.
if (vide==0){
temps = temps+u; t = t-u;
SORTIE[ds_guichet] = temps;
j = st.back(); st.erase(st.end()-1);
ds_guichet = j;
u = RESTE_A_TRAITER[j];
if (st.size()==0) vide = 1;
}
else {
temps = temps+u; t = t-u;
SORTIE[ds_guichet] = temps;
//La file est vide, le guichet attend l'arrivée du prochain
client.
temps = temps+t;
ARRIVEE[n] = temps;
//Le client est tout de suite pris en charge.
u = va2(mu); ds_guichet = n;
DUREE_TRAITEMENT[n] = u;
RESTE_A_TRAITER[n] = u;
//On attend le prochain client.
t = va1(lambda);
n = n+1;
} } } //fin du while
//*******************************
// Fonctions d'écriture dans un fichier
// des variables globales d'enregistrement.
fp = fopen(fichier,"w");
for (i=chauffe;i<=N-refroidit;i++)
fprintf(fp,"%d\t%8f\t%8f\t%8f\n", i,
ARRIVEE[i], DUREE_TRAITEMENT[i], SORTIE[i]);
fclose(fp);
#endif
}
```