POLITECNICO DI TORINO

Management and content delivery for Smart Networks

– Components of the team –

"I declare that what is written in this relation is the result of my work"

Avalle Giorgio **S241834**

Contents

Python code explanation

Introduction

Written code is intended as something more general of what actual requirements are: the basic idea is to start developing something that could be used by myself in the future, if needed. For this reason, you can see that part of the code is not implemented yet (there is just a skeleton) and you can notice that I've tried to keep it as modular as possible, sometimes defining functions who just call other ones, more internally. Anyway, this does not impact on the performances required by the exercises of the laboratories: the code is also well commented, to improve its readability as much as possible.

The code can be divided in **two parts**: the first one refers to classes used to model a generic queue (needed to solve the first laboratory), while the second part refers to classes that model the Cloud storage environment (the other laboratory).

Generally speaking, the code is based on functionalities provided by external libraries (that must be installed in order to test and execute my code):

- **simpy**, to create and run the simulation environment
- **numpy** and **scipy**, to compute particular statistics (such as confidence intervals)
- **matplotlib**, to draw plots on the screen

Queues

We will now see briefly what are the involved python files and what are the main classes and methods we can use to implement functionalities of my library

queue_system.py - QueueSystem

This is the main class, used to define a general queue according to the **Kendall's notation**: a textual log file can be specified, in order to save a log of the arrivals/services timings, and a callback can be defined, if we want to execute a specified function with certain parameters when a request is correctly served by the system (this functionality is useful, for example, when we chain several queues).

Idle servers and whole system capacity are modeled as simpy **Resources**: the population, instead, is modeled as a **Container**. Another useful Container is "**_LockedQueue**": unless if it is infinite, it represents a wall that external arriving requests has to pass in order to be served. The goal of its implementation is to force an arrival by the external, when the arrival rate is not defined using a pdf (for example, in the case of chained queues, where the services of a queue have to force an arrival, with probability p, in the following one).

Processes are scheduled into the simulation environment thanks to the "prepare" method: the main scheduled process is "**start**", in which:

- the **batch** of arriving requests is generated, according to its possible max/min values
- we wait to **unlock the queue** (it may be always unlocked)
- accordingly to the value of p, it is decided if the requests **actually reach** the system
- **arrivals** are simulated, only when we actually have them in the **population**
- out-of-buffer arrivals are **discarded**
- accepted ones are **served** by idle servers (in a parallel process) **or queued** in the buffer

At every time, needed statistics are computed in order to have, at the end of the simulation, results about the current batch or their aggregate.

The parallel process launched to serve a client request is "**process_customer**": what the code do is to simulate, using a timeout, its execution and tune opportunely the value of the simpy shared resources (the number of users into the system and the available population).

The method used to generate the arriving batch of requests is "**new_batch**": as you can see, its value is generates according to uniform pdf and the "b" value depends also on the population current value.

The "run" method is used to run, for a certain period of time, the simulation over the queue (simulation batch): once the simulation is completed, statistics and plots are shown to the user

queue_system.py - ChainedQueueSystem

This class allow us to model a **system composed by several queues**, chained each other: input parameters of the constructor are two arrays, containing the queue and its probability that an incoming request (arriving from the previous queue) actually reaches it.

The first queue of the system has p = 1: queues share the same simulation environment and have, as callback, a function able to unlock the queue they are forwarding the served request to. As before, once the simulation is done statistics are shown to the user.

qs_stats.py – ComputedStats

This class contains statistics computed for a simulation batch, or the global simulation, of a queue. From the constructor you can see what are the stored information: methods are mainly getters and setters (or methods used to increment fields' values.

The method "**compute_stats**" is used to compute batch simulation results: in other words, average values for the number of users in the system (queuing line, servers and both) and for the timings (traversing, waiting and service time for requests). Please note that these timings can be retrieved only for a subset of the arrived requests: for example, only served requests are considered when we speak about the traversing times or the service times.

qs_stats.py – QS_Stats_Manager

This class is used as a statistic manager for a queue: every time a request reaches the system, starts to be served, is served or is lost statistics are updated. At the end of simulation, they are given to the user. Statistics are stored as global and about every single batch of the simulation.

As you can see, in the constructor we use "**TimedArrays**" in order to store, for every request ID or for occupancy values, the timestamp associated to them or the time interval they last for. "TimedArray" class has been defined by me for this specific reason, it will be explained better later.

The method "**accept**" is launched when a group of requests is accepted by the system: they are declared as "pending request" and the occupancy values for the system and the queuing line are updated (please, note that more events can occur at the same timestamp, because the simulation environment time is discrete).

The method "**reject**" is launched when a group of requests is rejected by the system, because the buffer is full: it simply declares them as rejected.

The method "**start_to_serve**" is used to declare that some requests are going to be served: buffer occupancy is then updated, a "waiting_for" time is assigned to requests and they are declared as "pending services", instead of "pending requests". They will be served for sure, because we do not expect to have servers failures.

The method "**end_to_serve**" is used to declare that some requests have been correctly served by the system: occupancy value for the system is updated and requests are marked as "served", attaching them the service time required by the server to finish its work.

The method "**results**" is called when the simulation ends: several statistics are then computed and everything is given to the user. Also some graphs can be shown, thanks to the "**plot**" method and functionalities of the "qs_plots.py" file (based on the "numpy" library).

The static method "**integral_mean**" is used to compute the mean for values characterized by a temporal duration: in other words, "TimedArray" whose timestamp is set to False.

qs_stats.py – CQS_Stats_Manager

This class represents the statistic manager module for "ChainedQueueSystems". In the method "**resume**", it computes statistics relative to the whole system

- First of all, **every single queue** is analyzed independently
- **Average occupancies** are evaluated as the sum of the ones of the internal queues
- **Average times** are evaluated in a more difficult way: it is necessary to define, per each internal queue, the number of requests who leave the system without reaching the end (because served in advance: for example, by an HTTP Accelerator). Storing into "X" the number of correctly served user requests by the i-th queue of the system, we can use it to compute actual waiting, service and traversing time for the single request, according to the effective number of internal queues it has travelled through.

qs_plots.py

In this file, we simply find methods that, based on the matplotlib library, allow the user to **print graphs**. In particular, we can find standard plots, step ones, bar ones and plots specifically made to print pdfs or cdfs.

qs_utilities.py

In this file, we can find methods to work on the possible **arrival and service distribution pdfs** that are supported by the library. As you can see, at the moment only "Exponential" and "Chained" distribution are fully supported (the last one actually is not a pdf, but a way to indicate that the queue receives, in input, requests which are served by another one).

In particular, method "**get_pdf_params**" is used to retrieve as keyboard input parameters needed by the pdf, while "**pdf_random**" is used to generate an instance of a random variable whose pdf type and pdf parameters are the ones specified as input.

qs_exceptions.py

It simply defines a new type of exception, raised when a pdf type specified by the user is not valid.

input_controls.py

This file contains several methods that can be used to control the validity of input arguments

timed_structures.py – TimedData

This data structure refers to data (“data” field) which are characterized also by a **temporal information** (“time” field): according to the “timestamp” Boolean flag, “time” can be a timestamp or a temporal interval.

timed_structures.py – TimedArray

This class represents **arrays formed by “TimedData” objects**, having same “timestamp” value associated (i.e.: it is not possible to mix timestamps with temporal intervals).

Several methods are implemented on this class, in order to retrieve, add, filter, search or update the content of the array.

First laboratory

The purpose of this lab is to model a web server, under specific conditions: M/M/1 queue, M/M/1/B queue with batch arrivals and chained queue system. Arrivals are users' HTTP requests, which are elaborated by the server after a certain time (queuing time + server execution time).

Hypothesis: no failures occur and FIFO scheduling policy is adopted.

Exercise 01

As required, the two queues ("q1" = M/M/1 and "q2" = M/M/1/B, with batch arrivals) are created as instances of the class "QueueSystem", described above. Input parameters are controlled, in order to be sure the code will work properly.

The single batch has a time period which is indicated by "SIM_TIME", while arrival and services parameters are prompted by the user. For each queue, after the simulation we get final results and several graphs. **Five tests are run** in total, in order to observe how the queues' parameters can alter the final results.

The problem of the simulation warm-up

When we analyze a modeled system, we need that it is in steady-state in order to retrieve correct results (**steady-state simulation**). In fact, when we compute a confidence interval we are assuming that the system is stable, the process is stationary and that the observations are independent each other: as a result, we have to identify and remove the warm-up transient.

This operation is often very difficult to perform: one idea can be to implement the "**initial data removal**" technique described in the course's slides. It is based on the fact that we can identify the transient when the average performed removing first "k" samples does not change very much: this happens when we start to remove steady-state sample, because of their property regularity.

In the file "simulation_warm_up.py" you can find a test done in order to determine the duration of the transient, for several types of queues: a long simulation is done and values for Rk are printed as a graph. Actually, what happens is that the **transitory is negligible**: queues under test are always ergodic (arrivals rate = 0.2, service rate = 0.3, or 0.7 in case of batch arrivals). For this reason, in the exercises we will simply run a dummy batch in order to be sure to have removed the transient; in "*Appendix A – Warm-up removal*" are reported resulting charts

Obtained results – Queue 1

The first queue is an **M/M/1** one: several values for the arrival and service rates ("λ" and "μ") are tested, to see how the system performances vary according to them (but the system will be always ergodic, in order to satisfy the hypothesis to compute the confidence interval). The random seed will be always the same, in order to have better comparisons and to have repeatable experiments.

Detailed results (log files) and images are available in the files attached to this report: an example of obtained graphs for a given simulation can be found in "*Appendix B – Laboratory 01: Graphs of the exercise 01*".

Please note: E{Ts} and E{T} are evaluated only for served requests. SIM_TIME = 100, BATCHES = 24. Confidence intervals are reported as evaluated by the software. Actually, their range can be reduced eliminating negative values (which are impossible, for the quantities we are analyzing).

Test	#1	#2	#3	#4	#5
Arrival rate - λ	2	2	3.95	2	2
Service rate - μ	4	10	4	4	10
Arrived requests	5914	5880	17527	5914	5880
Served requests	5912	5880	17414	5912	5880
E{Nw}	0.86	0.02	82.94	0.86	0.02
E{Ns}	0.32	0.01	0.94	0.32	0.01
E{N}	1.18	0.03	83.88	1.18	0.03
E{Tw}	0.36	0.01	11.85	0.36	0.01
E{Ts}	0.14	0.01	0.14	0.14	0.01
E{T}	0.50	0.01	11.99	0.50	0.01
Response time CI: 90%	(0.14, 0.86)	(-0.02, 0.04)	(-3.45, 26.75)	(0.14, 0.86)	(-0.02, 0.04)
Response time CI: 99%	(-0.09, 1.09)	(-0.03, 0.05)	(-13.08, 36.38)	(-0.09, 1.09)	(-0.03, 0.05)
Buffer occupancy CI: 90%	(-0.05, 0.25)	(-0.01, 0.01)	(-2.73, 19.25)	(-0.05, 0.25)	(-0.01, 0.01)
Buffer occupancy CI: 99%	(-0.15, 0.35)	(-0.01, 0.01)	(-9.74, 26.26)	(-0.15, 0.35)	(-0.01, 0.01)

Commenting results – Queue 1

Actually, the 4th test and the 5th one are identical to the 1st and the 2nd ones: the reason is that the changed parameters involve only the second queue. So, from the point of view of the first queue, simulations appear to be identical each other.

The 2nd simulation is a light-traffic situation, while the 3rd represents a high-traffic: anyway, the system is always ergodic (as requested by the analysis methods I have adopted).

Every incoming request is accepted by the system. A subset of them (the majority) has been correctly served at the end of the simulation: others are still pending.

As we could expect, confidence intervals with an higher confidence level become wider.

Obtained results – Queue 2

The second queue is an **M*/M/1/B** one: several values for the capacity ("B"), for the arrival and service rates ("λ2" and "μ2") are tested, to see how the system performances vary according to them. The random seed will be always the same, in order to have better comparisons and to have repeatable experiments.

Detailed results (log files) and images are available in the files attached to this report: an example of obtained graphs for a given simulation can be found in "*Appendix B – Laboratory 01: Graphs of the exercise 01*".

Please note: E{Ts} and E{T} are evaluated only for served requests. SIM_TIME = 100, BATCHES = 24. Confidence intervals are reported as evaluated by the software. Actually, their range can be reduced eliminating negative values (which are impossible, for the quantities we are analyzing).

Test	#1	#2	#3	#4	#5
Arrival rate - λ2	2	2	1.95	2	2
Service rate - μ2	4	10	2	4	10
Buffer size - B	10	10	10	1	10
Minimum batch size - a	1	1	1	1	3
Maximum batch size - b	3	3	3	3	8
Accepted requests	10159	11842	5817	6150	23828
Served requests	10157	11842	5814	6150	23828
Rejected requests	1625	112	5744	5937	8268
E{Nw}	2.65	0.14	7.44	0.25	0.31
E{Ns}	0.55	0.03	0.96	0.34	0.06
E{N}	3.20	0.17	8.40	0.59	0.37
E{Tw}	0.65	0.03	3.20	0.10	0.03
E{Ts}	0.14	0.01	0.41	0.14	0.01
E{T}	0.79	0.04	3.61	0.24	0.04
Response time CI: 90%	(0.50, 1.06)	(-0.01, 0.07)	(2.81, 4.37)	(0.19, 0.29)	(0.00, 0.08)
Response time CI: 99%	(0.32, 1.24)	(-0.03, 0.09)	(2.32, 4.86)	(0.15, 0.33)	(-0.02, 0.10)
Buffer occupancy CI: 90%	(-0.35, 1.05)	(-0.03, 0.07)	(-0.48, 2.20)	(-0.02, 0.08)	(-0.02, 0.08)
Buffer occupancy CI: 99%	(-0.79, 1.49)	(-0.06, 0.10)	(-1.33, 3.05)	(-0.06, 0.12)	(-0.05, 0.11)
Rejected requests CI: 90%	(10.93, 120.15)	(-4.73, 13.31)	(138.77, 325.39)	(186.27, 288.89)	(200.24, 464.60)
Rejected requests CI: 99%	(-23.92, 155.00)	(-10.49, 19.07)	(79.24, 384.92)	(153.53, 321.63)	(115.91, 548.93)

Commenting results – Queue 2

According to the traffic conditions, the buffer size is quite limited: batches are not very big, so that the system is usually able to operate well.

In the 2nd and 5th simulation the system is faster: but in the last one the traffic is heavier, because of the average batch size (which is higher than in the 2nd test).

The 3rd and the 4th simulation, instead, represents the conditions under which the system is not able to work properly. This happens when the arrival rate is similar to the service one or when the buffer size is very little (according to traffic conditions).

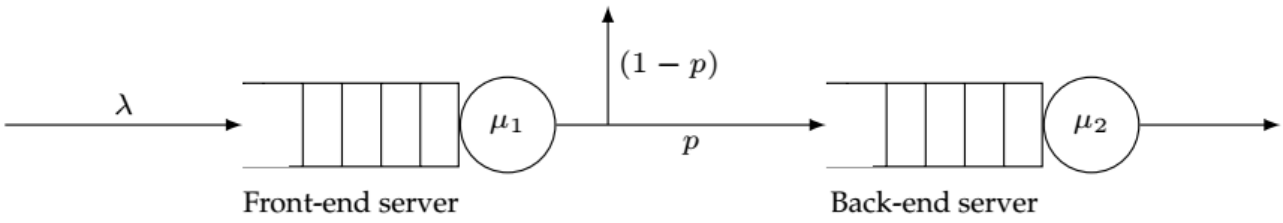
Every incoming request can be accepted or refused by the system. A subset of the accepted ones (the majority) has been correctly served at the end of the simulation: others are still pending.

The system is always ergodic, because of its limited buffer size. As we could expect, confidence intervals with an higher confidence level become wider.

Exercise 02

This exercise requires to model and analyze a system composed by **two concatenated queues** ($M^x/M/1/B$): the front-end server is much faster than the back-end and is able to satisfy user requests with probability 1 – p.

Requests arrive to the first queue in batches and, only if not served, they reach the back-end server.



What we can notice, is that installing a front-end server **could be dangerous**: this happens if the majority of the requests cannot be immediately satisfied (high values for p). This imply that the front-end introduces an unnecessary delay, that could be avoided uninstalling this server.

In general, installing a front-end server can be convenient if the average traversing time of the double-queued system is lower than the one-queue system traversing time:

$$E\{T_1\} + pE\{T_2\} = \frac{1}{\mu_1} + p\frac{1}{\mu_2} < \frac{1}{\mu_2} = E\{T_2\}$$

Obtained results

The code prints five **graphs** per each queue, which are referred to the current batch and are identical to the statistics graphs seen in the previous exercise. They can be found in the "*Appendix C – Laboratory 01: Graphs of the exercise 02*".

The program gives also to the user some useful **statistics**, which are requested by this exercise and are reported in the table below. Five tests are run, in order to verify how different values for the parameters can alter final results.

Please note: E{Ts} and E{T} are evaluated only for served requests. SIM_TIME = 100.

Front-end statistics

Test	#1	#2	#3	#4	#5
Probability not to be served by FE – p	0.5	0.5	0.5	0.5	0.1
FE Arrival rate – λ1	2	2	3.95	2	2
FE Service rate – μ1	4	4	4	4	4
FE Buffer size – B1	10	40	10	10	10
Minimum batch size – a	1	4	1	1	1
Maximum batch size – b	3	7	3	3	3
Received requests	501	1282	1630	529	421
Accepted requests	421	658	685	414	349
Elaborated requests	419	624	674	414	348
E{Nw}	2.86	29.33	6.43	3.76	3.12

E{Ns}	0.59	0.98	0.85	0.64	0.53
E{N}	3.45	30.31	7.28	4.40	3.65
E{Tw}	0.68	4.5	0.93	0.91	0.89
E{Ts}	0.14	0.15	0.12	0.15	0.15
E{T}	0.82	4.65	1.06	1.06	1.05

Back-end statistics

Test	**#1**	**#2**	**#3**	**#4**	**#5**
BE Service rate – μ2	0.4	0.4	0.4	0.4	0.4
BE Buffer size – B2	12	12	12	60	12
Requests served by FE	210	311	321	209	305
Received requests	209	332	353	205	43
Accepted requests	58	59	51	101	38
Elaborated requests	45	46	38	40	37
E{Nw}	10.94	11.05	11.35	50.42	6.47
E{Ns}	0.99	0.96	0.97	0.98	0.94
E{N}	11.93	12.01	12.32	51.40	7.41
E{Tw}	20.54	18.30	23.92	40.80	17.03
E{Ts}	2.18	2.02	2.47	2.38	2.51
E{T}	22.58	20.00	26.26	42.23	19.89

Whole system statistics

Test	**#1**	**#2**	**#3**	**#4**	**#5**
E{N}	15.38	42.32	19.60	55.80	11.06
E{Nw}	13.80	40.38	17.78	54.18	9.59
E{Ns}	1.58	1.94	1.82	1.62	1.47
E{T}	4.30	6.86	3.46	7.46	2.73
E{Tw}	0.52	0.41	0.38	0.53	0.42
E{Ts}	4.38	7.27	3.84	8.00	3.15

Commenting results

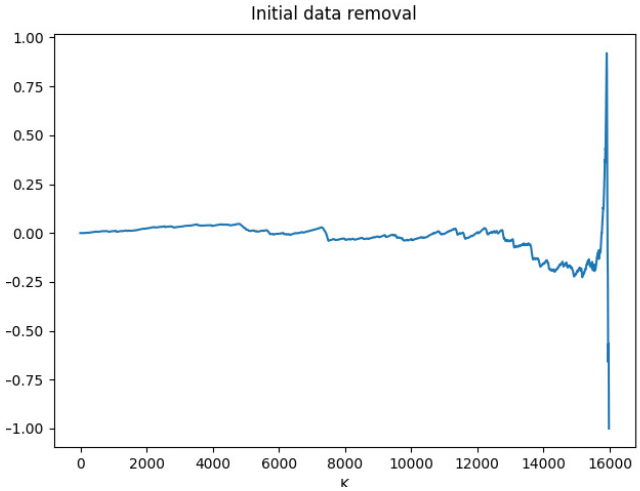
Every incoming request can be accepted or refused by every queue of the system. A subset of the accepted ones has been correctly served at the end of the simulation: others are still pending.

In general, a request is **served** by the system if and only if it is served correctly by one of the internal queues. This means that the number of total served request is given by the sum of "Requests served by FE" and "Elaborated requests" (for the second queue's table).

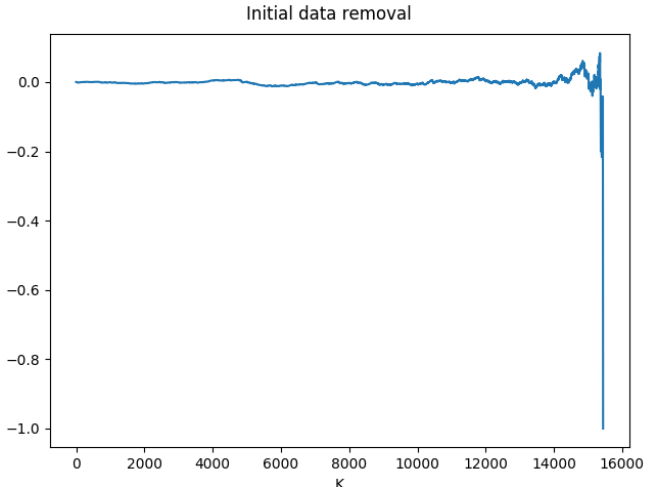
The second queue is much **slower** than the first one: except for the 5th test, in which the value of p is little, the majority of received requests (the ones not served by the front-end) is rejected by the system. The reason is that the second buffer, which is not sufficiently large, becomes full in small time.

Although the average batch size is quite limited, the traffic results to be heavy for the system when p is high. Clearly, augmenting the buffer size the percentage of rejected requests is reduced. The system is always **ergodic**, because of its limited buffers size.

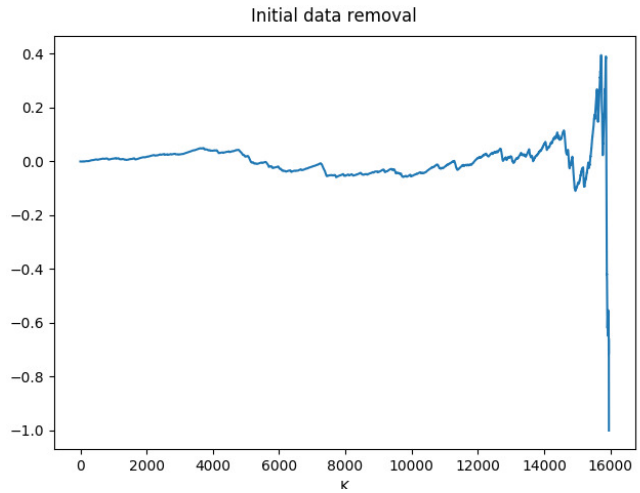
Appendix A – Warm-up removal



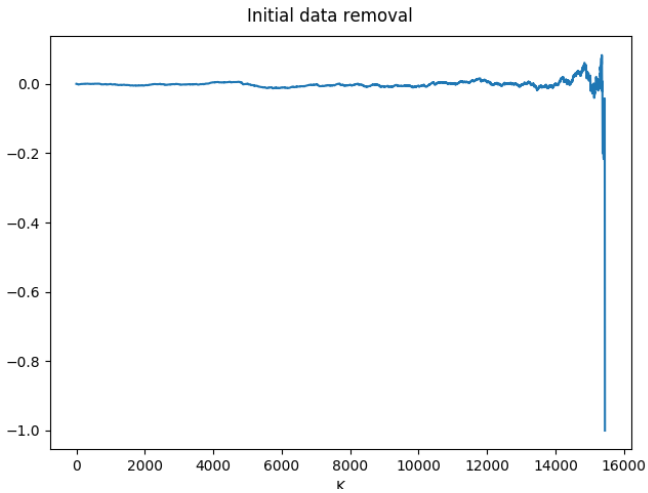
Q1: M/M/1; λ = 0.2, μ = 0.3



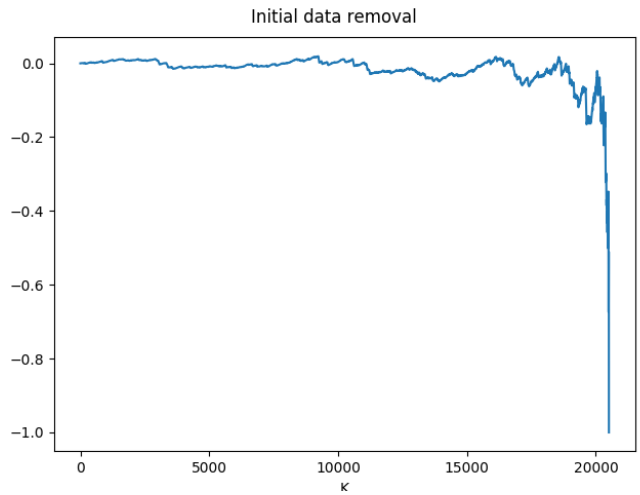
Q2: M/M/10; λ = 0.2, μ = 0.3



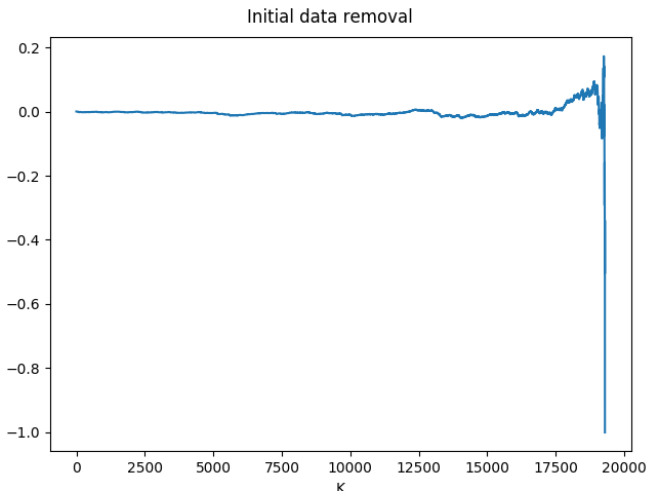
Q3: M/M/1/10; λ = 0.2, μ = 0.3



Q4: M/M/5/10; λ = 0.2, μ = 0.3



Q5: M^X/M/1; λ = 0.2, μ = 0.7, **batch** = U(1,3)



Q6: M^X/M/1; λ = 0.2, μ = 0.7, **batch** = U(1,3)

Appendix B – Laboratory 01: Graphs of the exercise 01

Meaning of the parameters

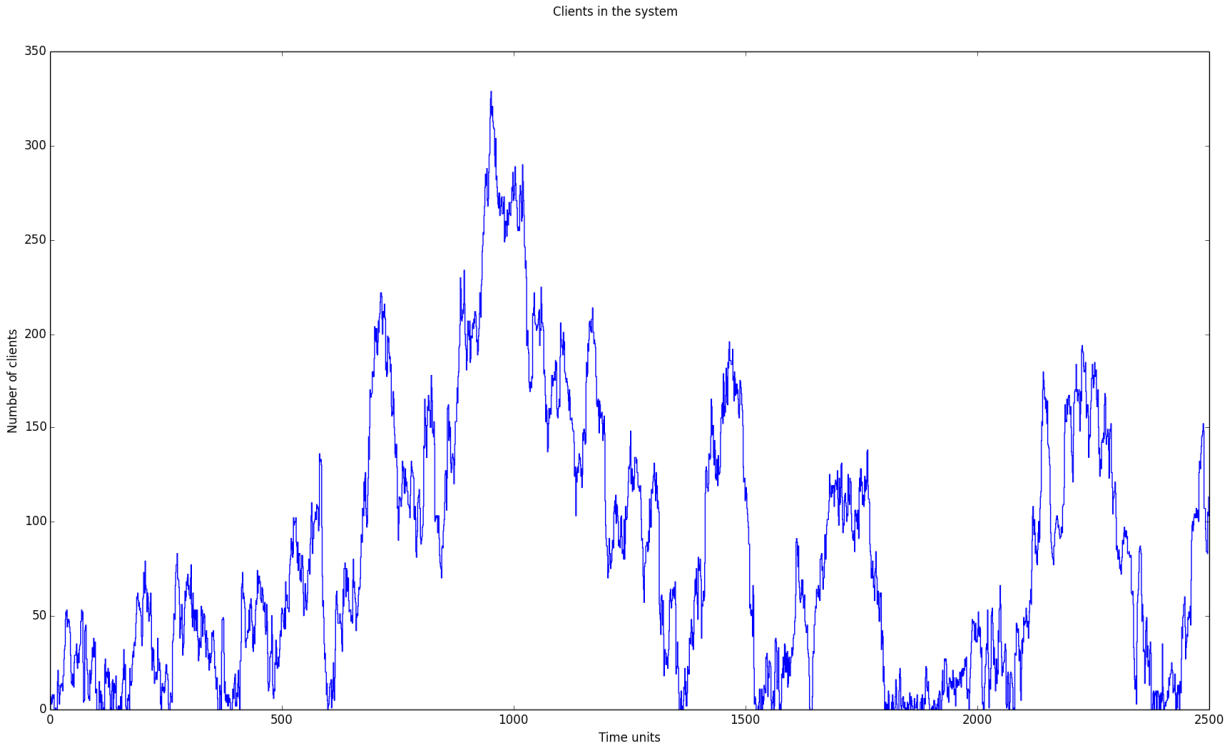
Remember that the first queue is M/M/1, while the second one is a M/M/1/B, with batch arrivals

- **λ**: arrival rate of the first queue
- **μ**: service rate of the first queue
- **λ2**: arrival rate of the first queue
- **μ2**: service rate of the first queue
- **B**: buffer size of the second queue
- **a**: minimum size for a batch, arriving at the second queue
- **b**: maximum size for a batch, arriving at the second queue

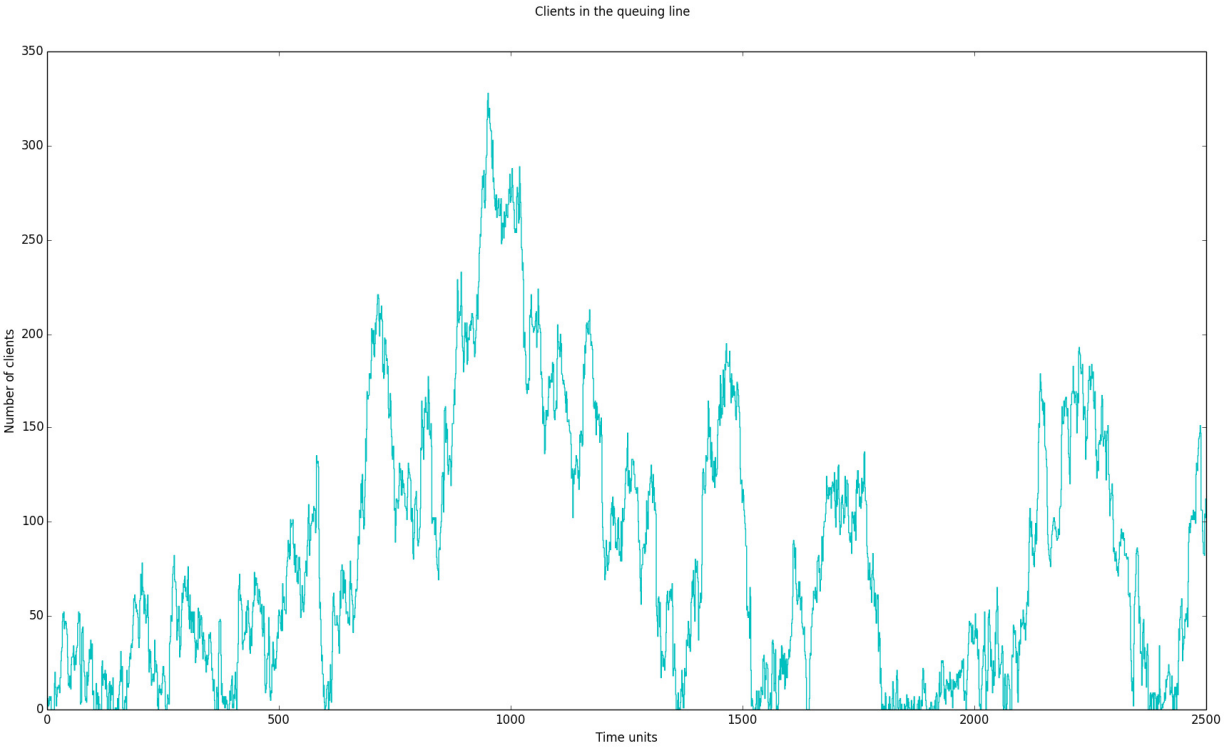
Simulation 03: λ=3.95, μ=4, λ2=1.95, μ2=2, B = 10, a=1, b=3

Higher resolution images and textual output for all the simulation I have run can be found in the files attached to this report. As an example of what is the output of the program, here you can find images of the third test.

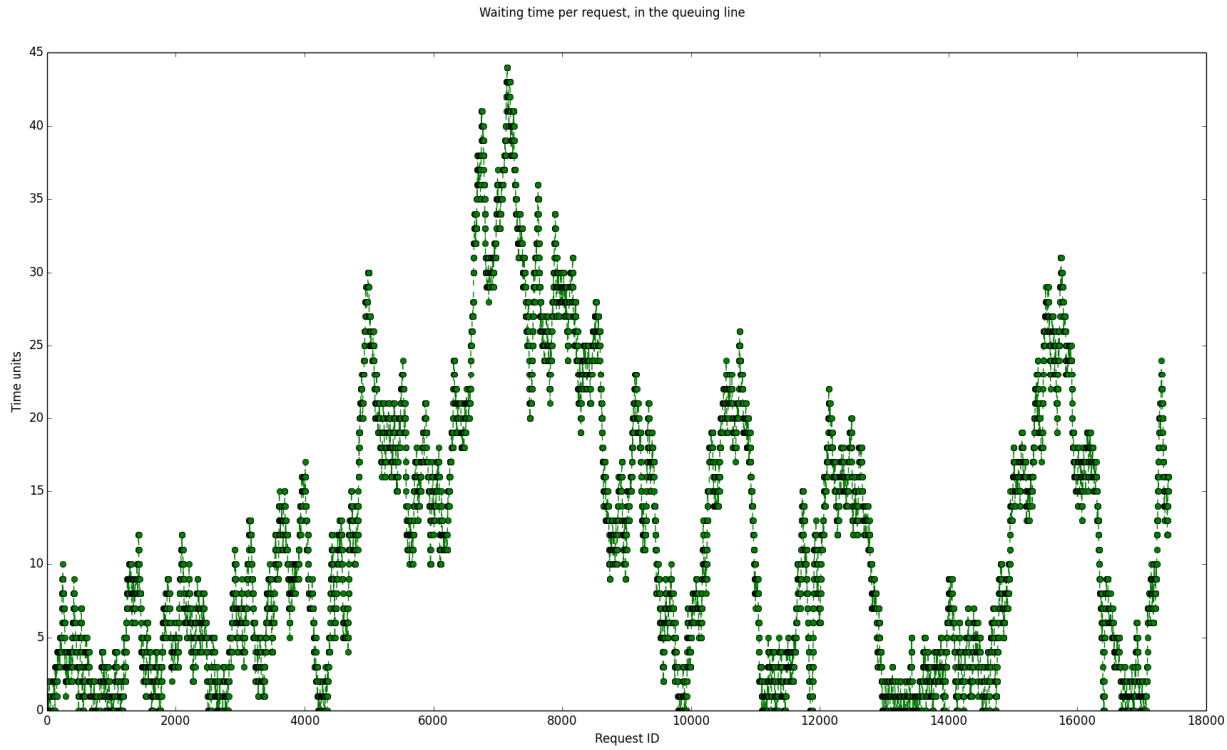
Queue 1



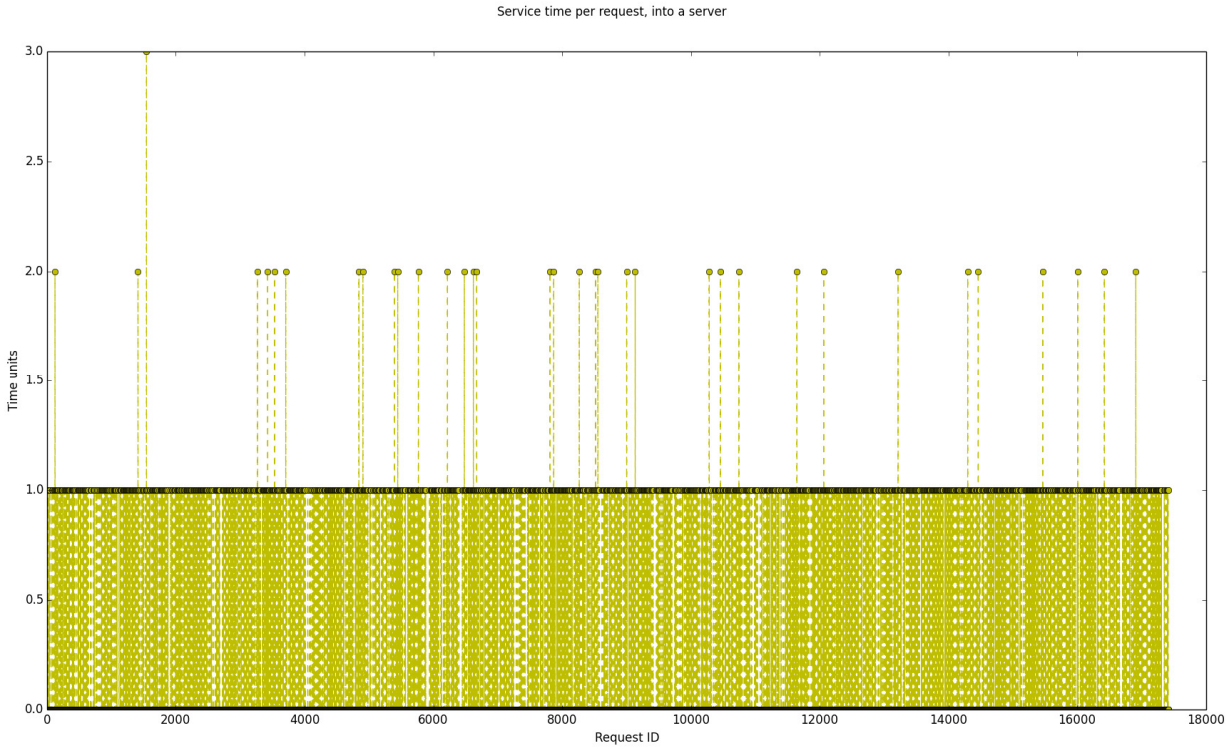
Clients in the whole system



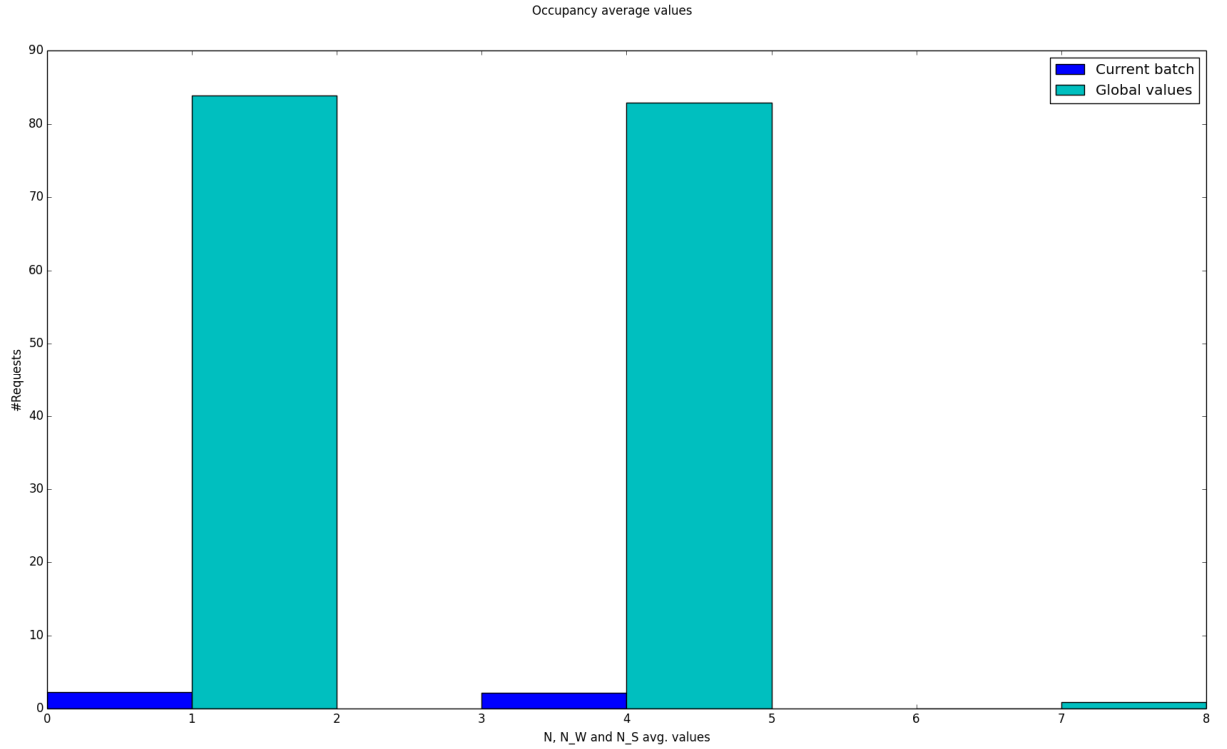
Clients in the queuing line only



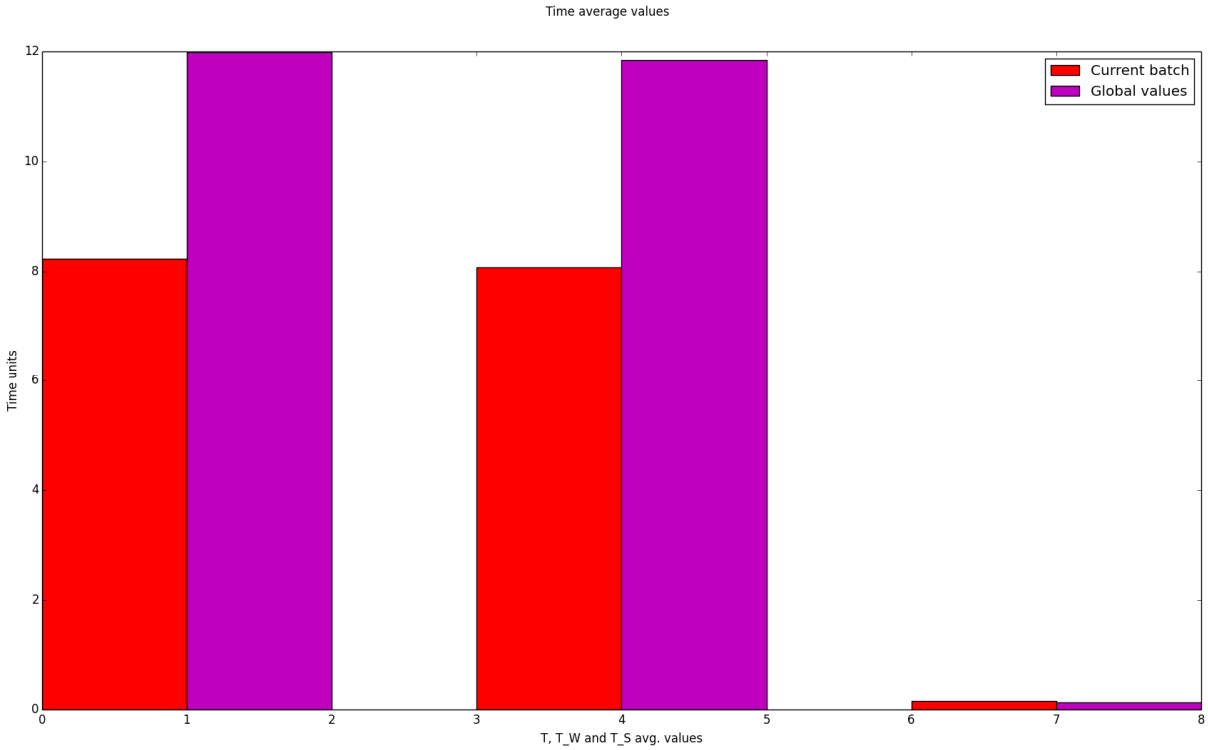
Waiting time per request, in the queuing line



Service time per request – First batch

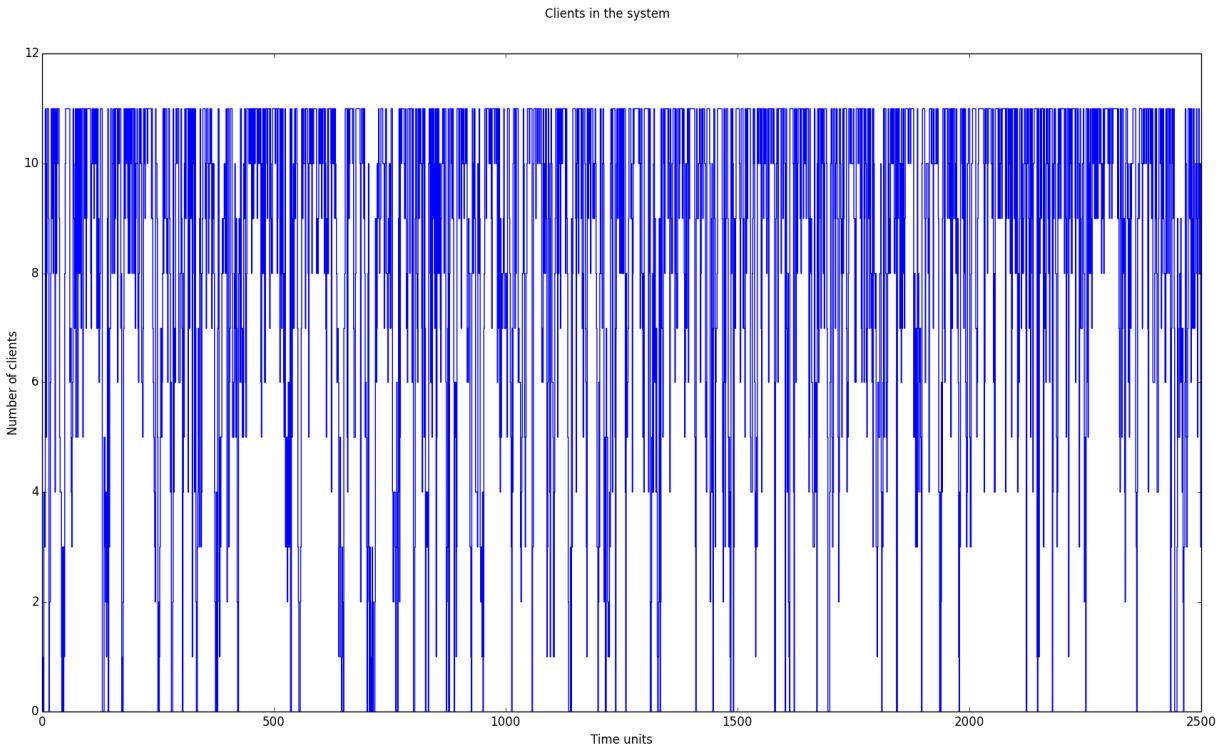


Average system (left), waiting line (center) and server (right) occupancies – Last batch and global results

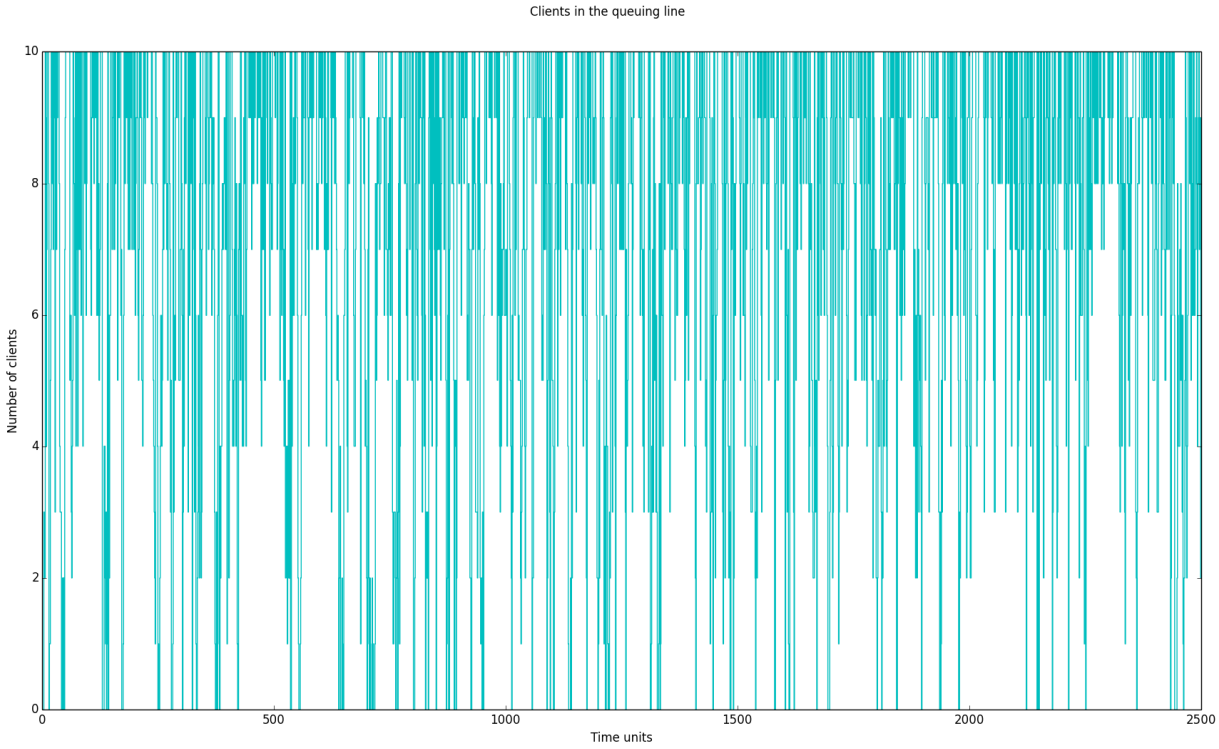


Average traversing (left), waiting (center) and service (right) times – Last batch and global results

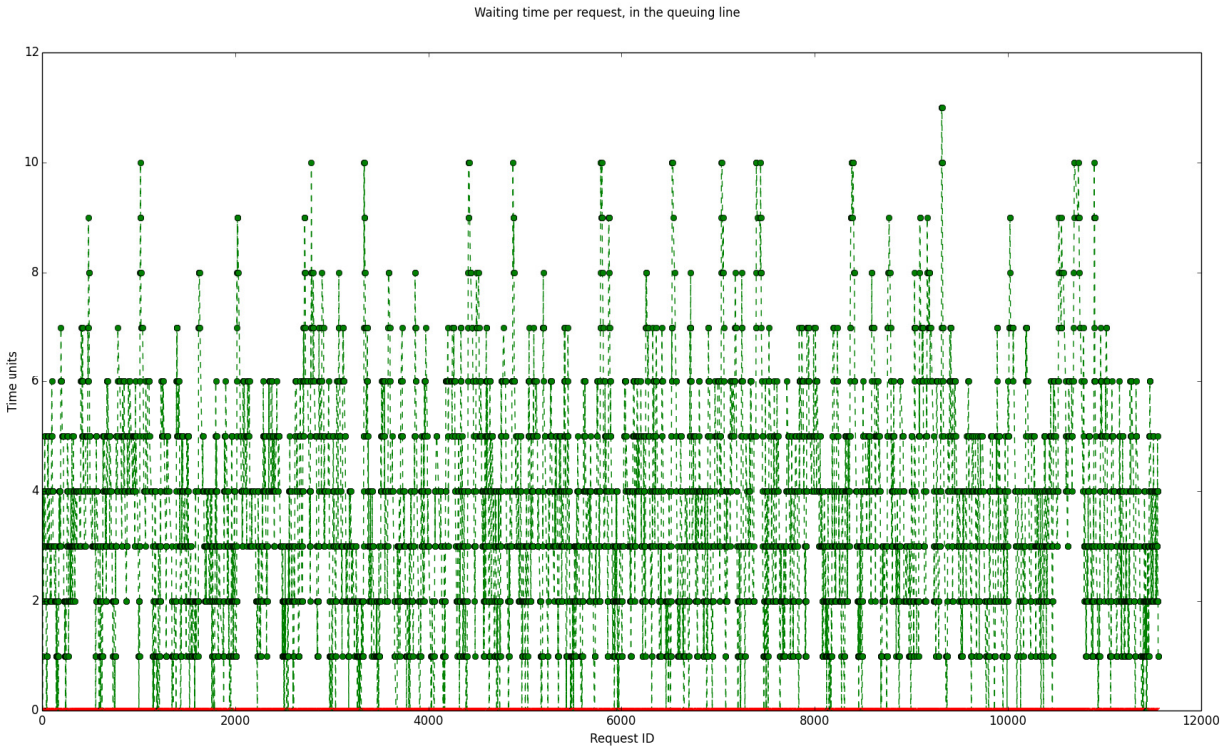
Queue 2



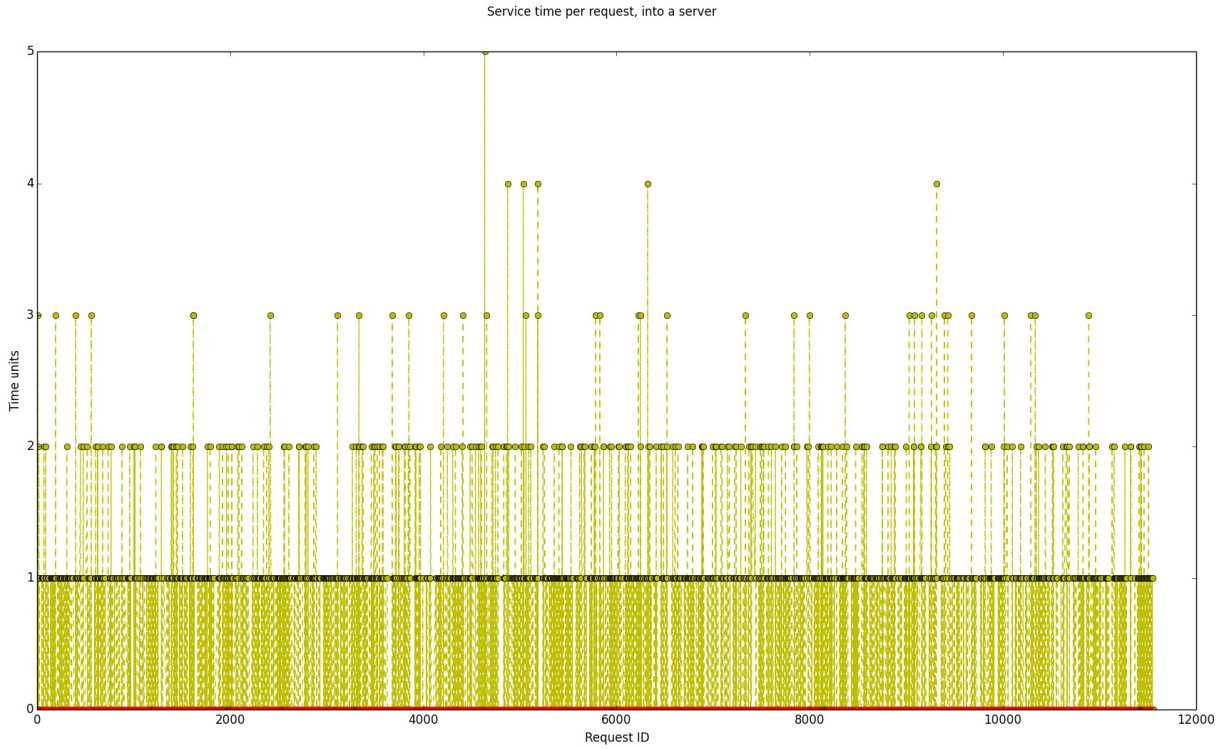
Clients in the whole system



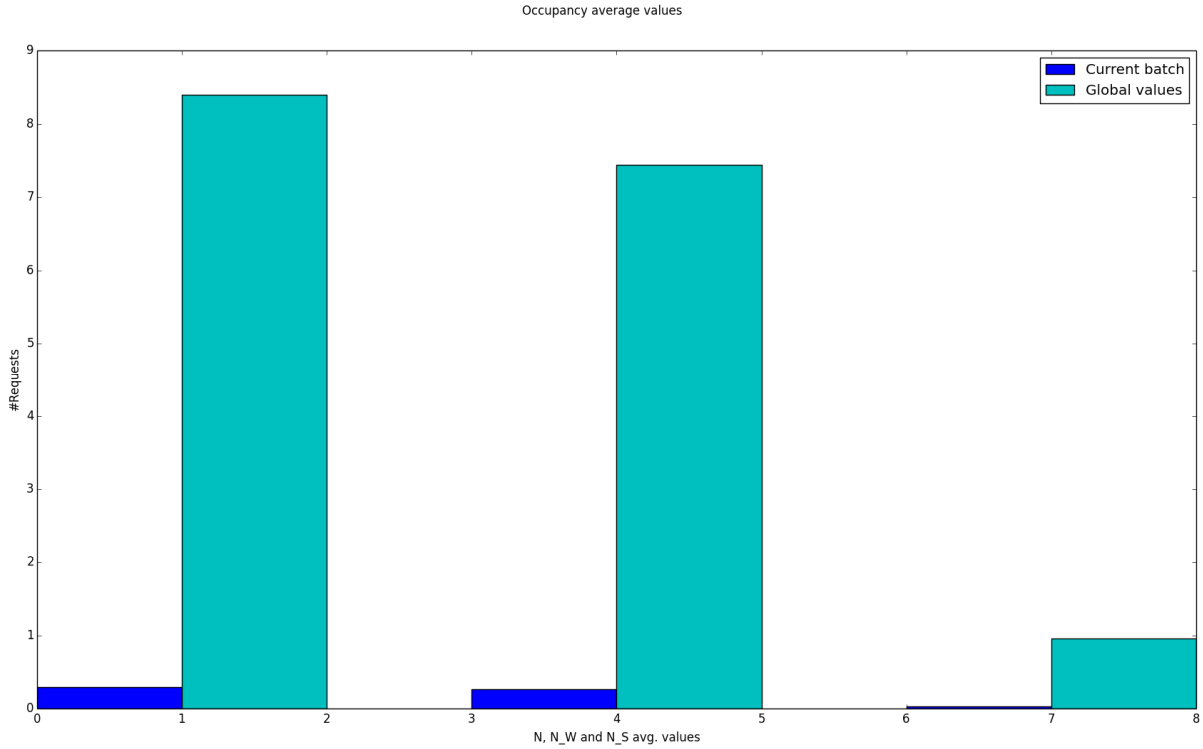
Clients in the queuing line only



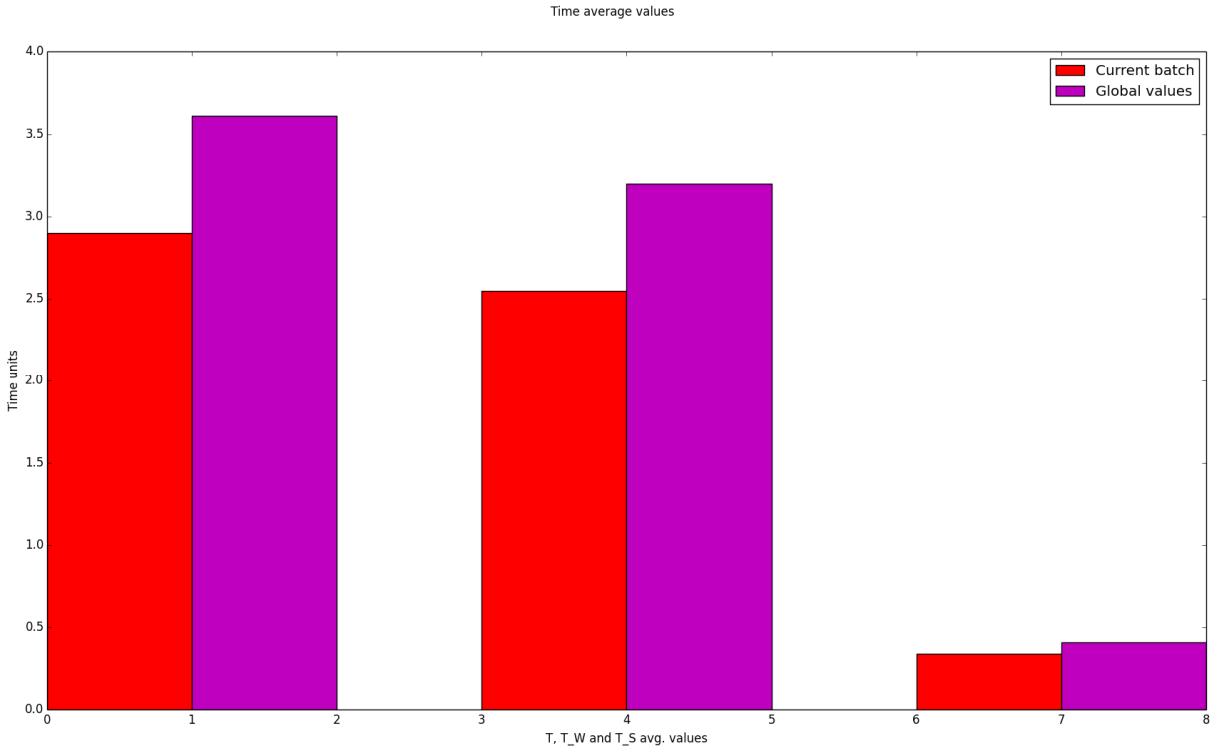
Waiting time per request, in the queuing line (lost ones are marked in red)



Service time per request – First batch (lost ones are marked in red)



Average system (left), waiting line (center) and server (right) occupancies – Last batch and global results



Average traversing (left), waiting (center) and service (right) times – Last batch and global results

Appendix C – Laboratory 01: Graphs of the exercise 02

Meaning of the parameters

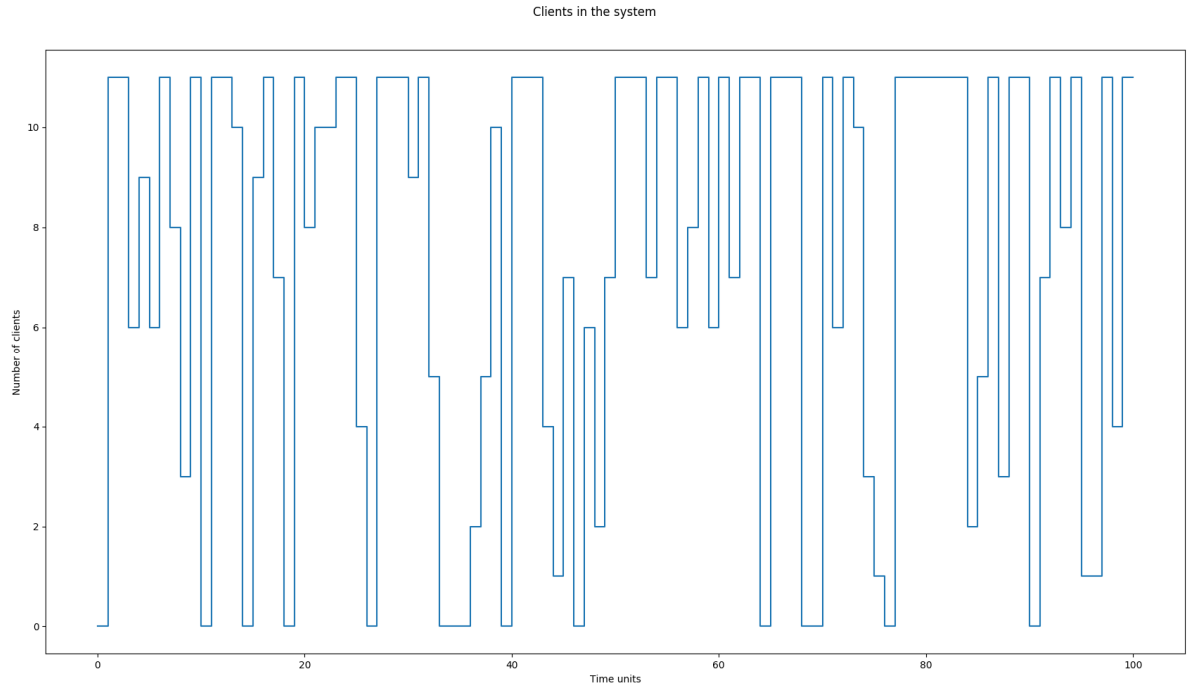
Remember that the system is composed by two concatenated queues M/M/1/B, with batch arrivals

- **λ_1**: arrival rate of the requests (first queue)
- **μ_1**: service rate of the first queue
- **μ_2**: service rate of the first queue
- **B_1**: buffer size of the first queue
- **B_2**: buffer size of the first queue
- **a**: minimum size for a batch, arriving at the second queue
- **b**: maximum size for a batch, arriving at the second queue
- **p**: probability that the front-end is not able to satisfy the request, passing it to the back-end server

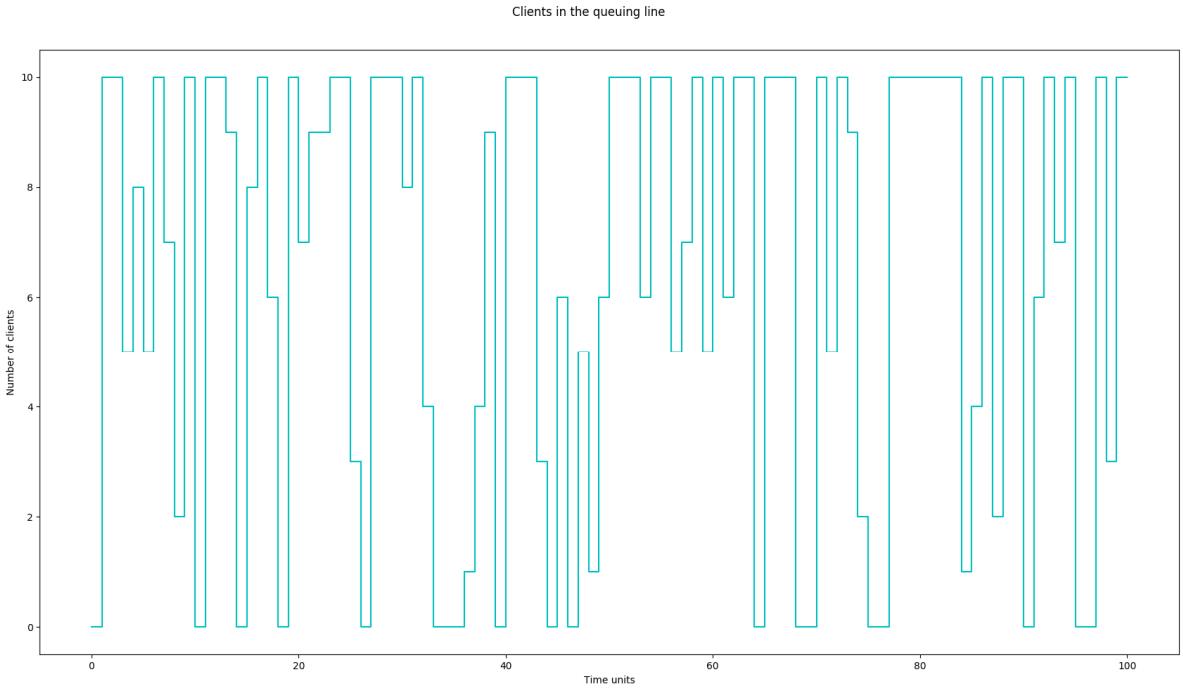
Simulation 03: λ_1=3.95, μ_2=4, μ_2=0.4, B_1 = 10, B_2 = 12, a=1, b=3, p=0.5

Higher resolution images and textual output for all the simulation I have run can be found in the files attached to this report. As an example of what is the output of the program, here you can find images of the third test.

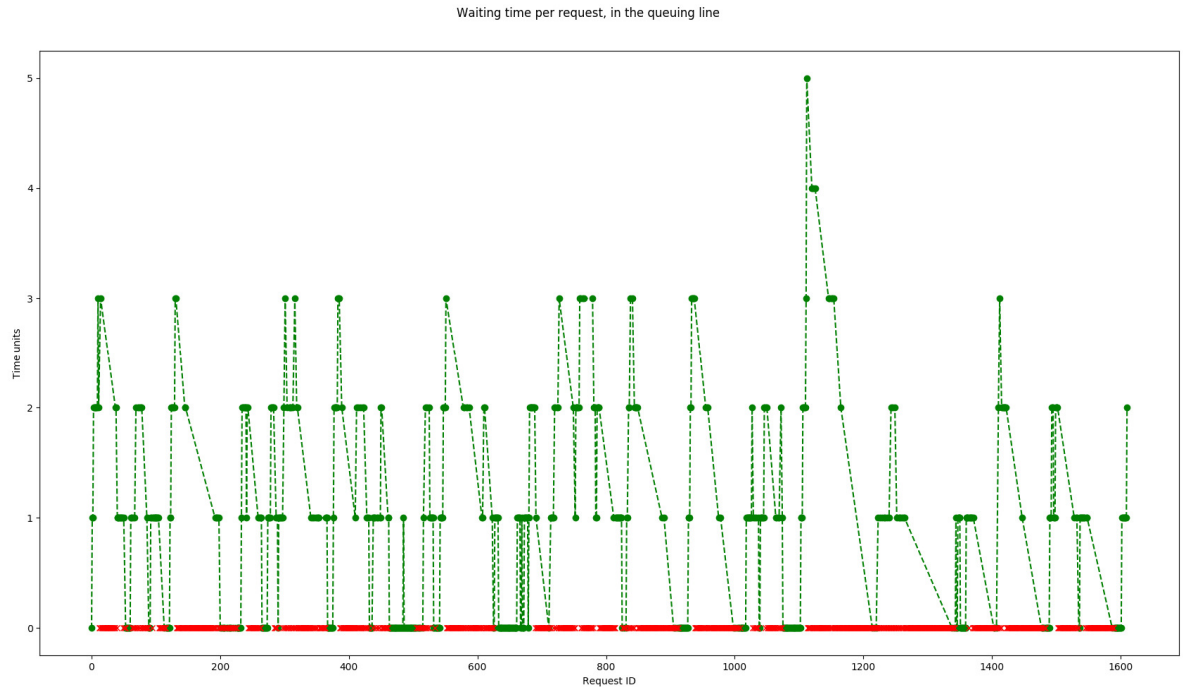
Queue 1



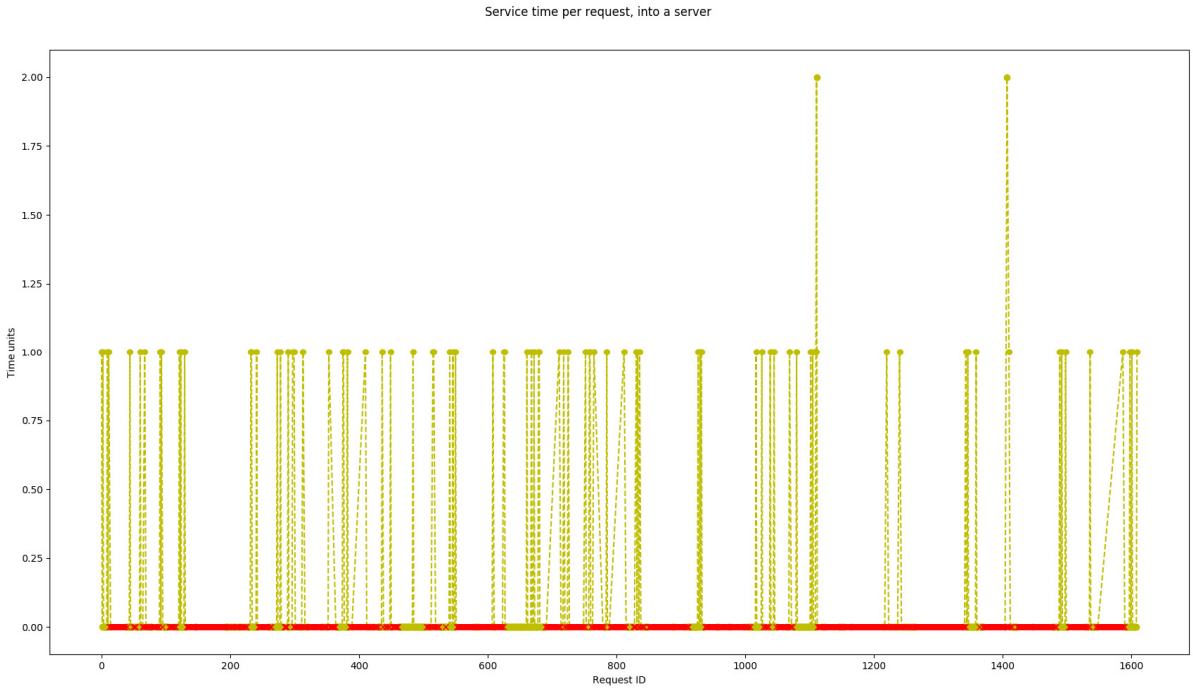
Clients in the whole system (queue)



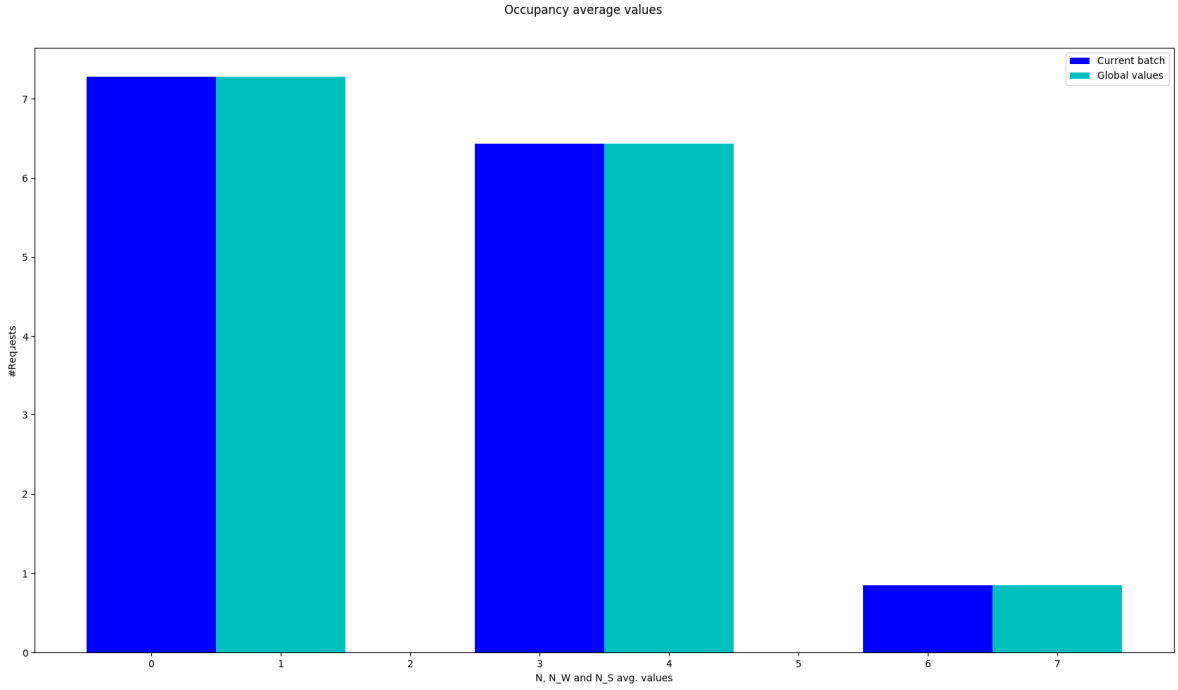
Clients in the queuing line only



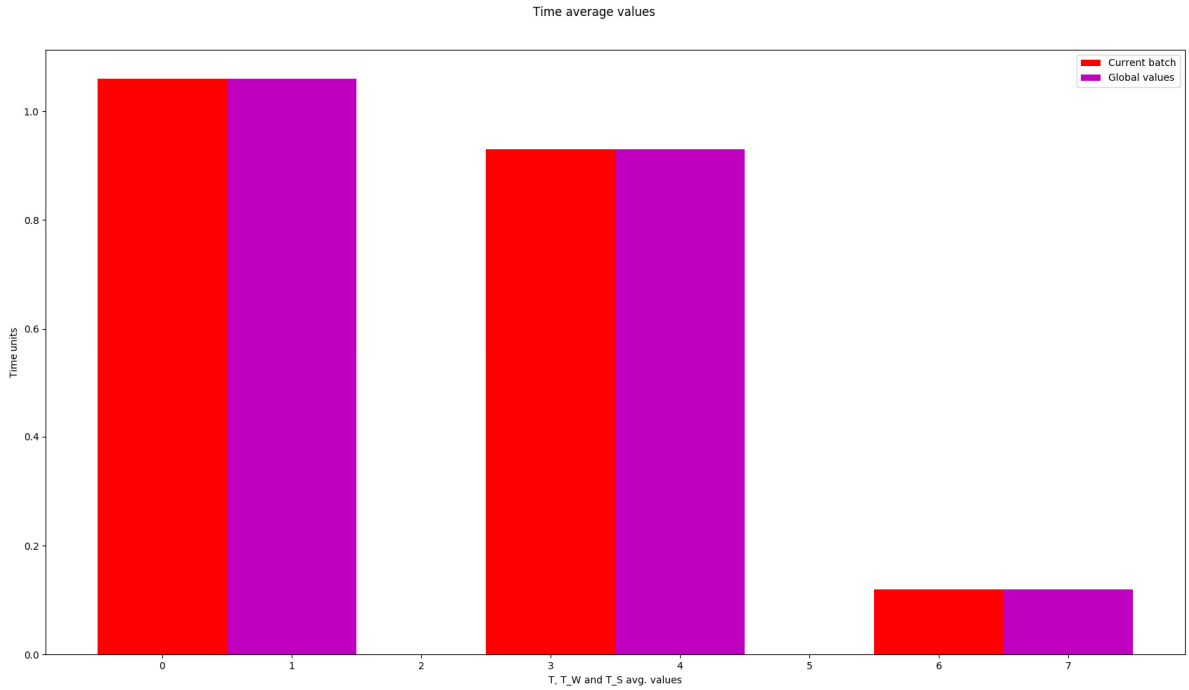
Waiting time per request, in the queuing line (lost ones are marked in red)



Service time per request – First batch (lost ones are marked in red)

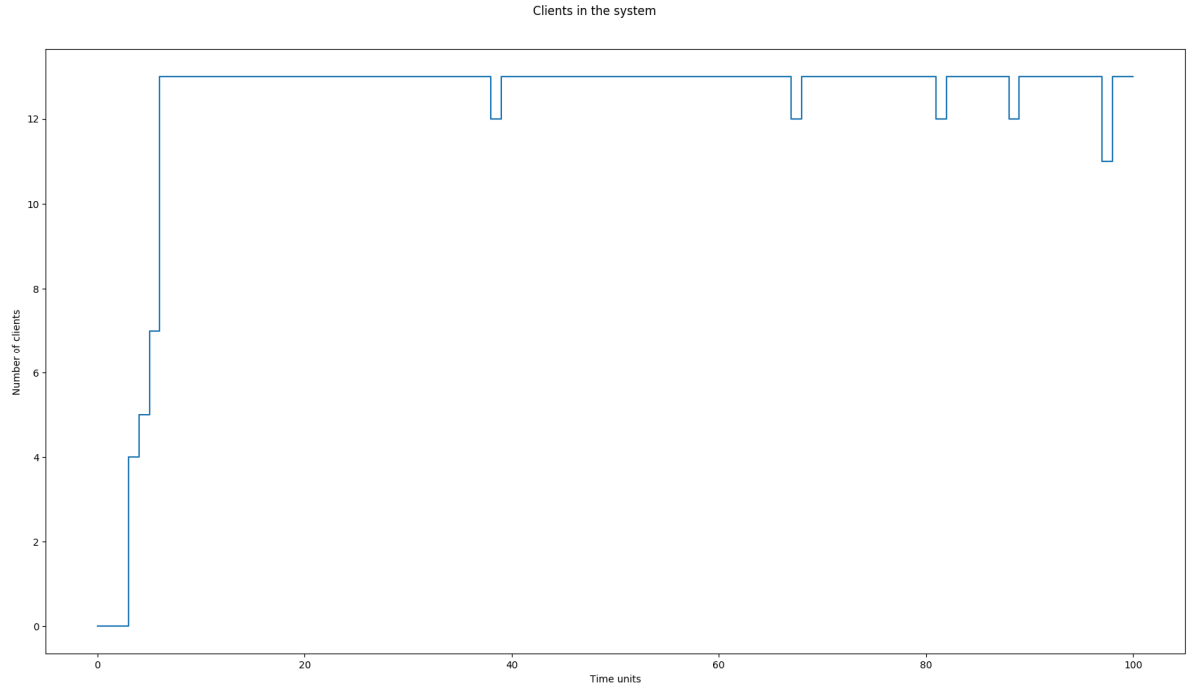


Average system (left), waiting line (center) and server (right) occupancies – Last batch and global results

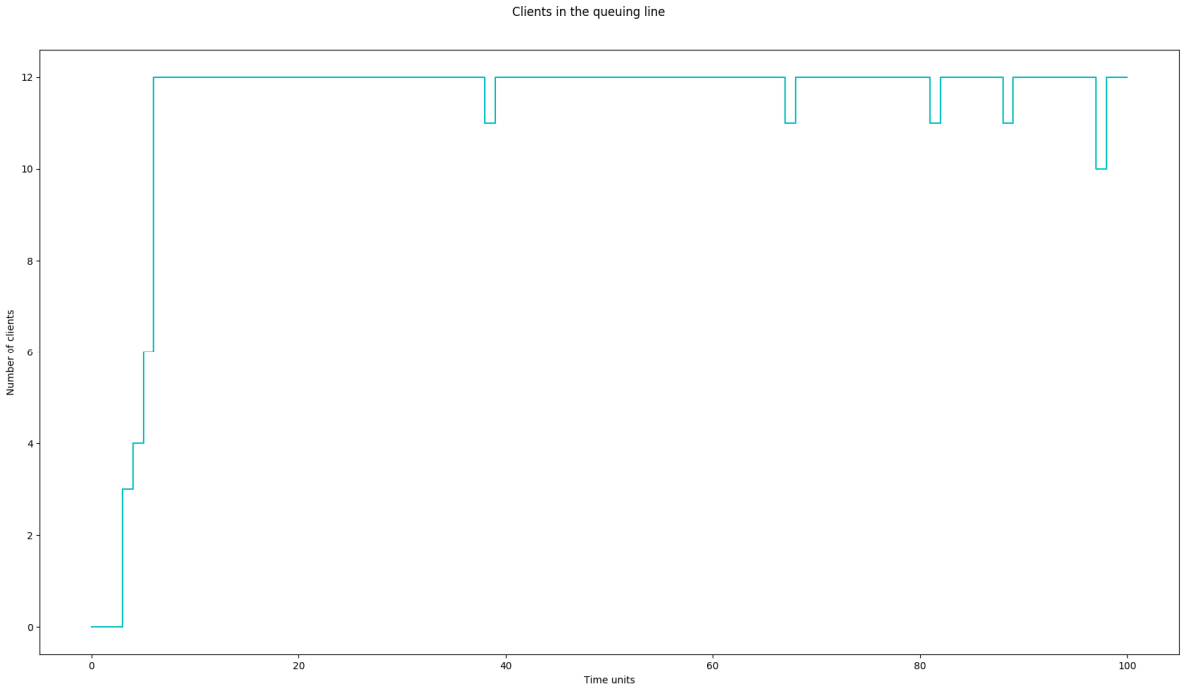


Average traversing (left), waiting (center) and service (right) times – Last batch and global results

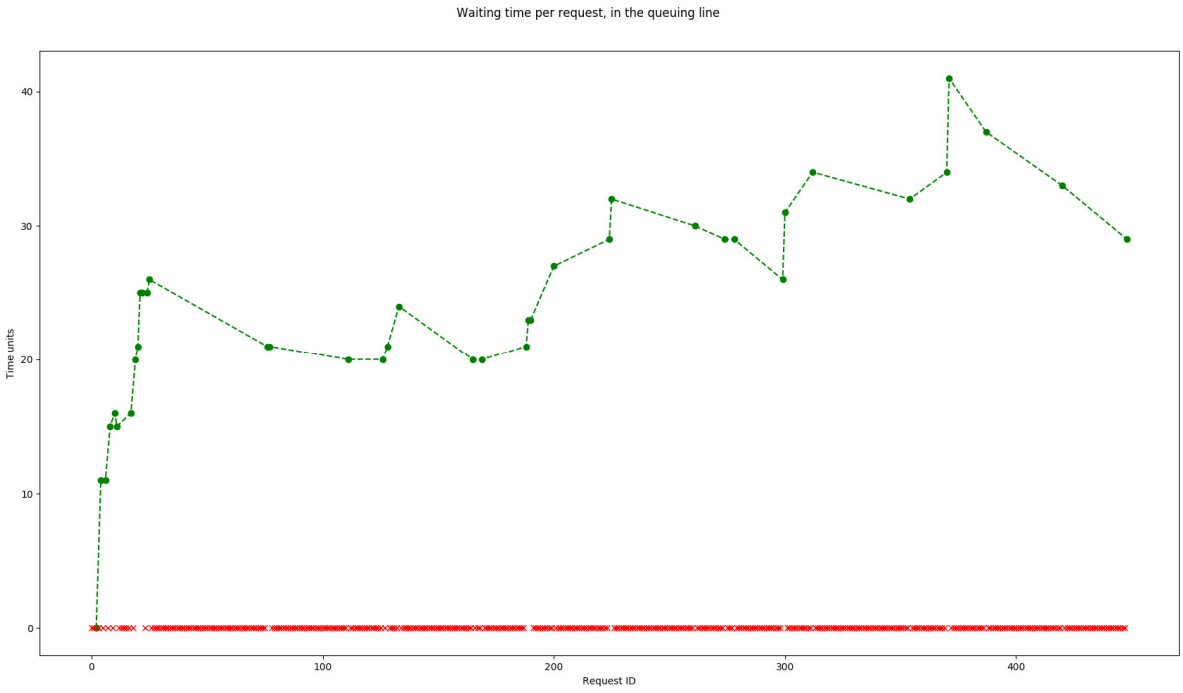
Queue 2



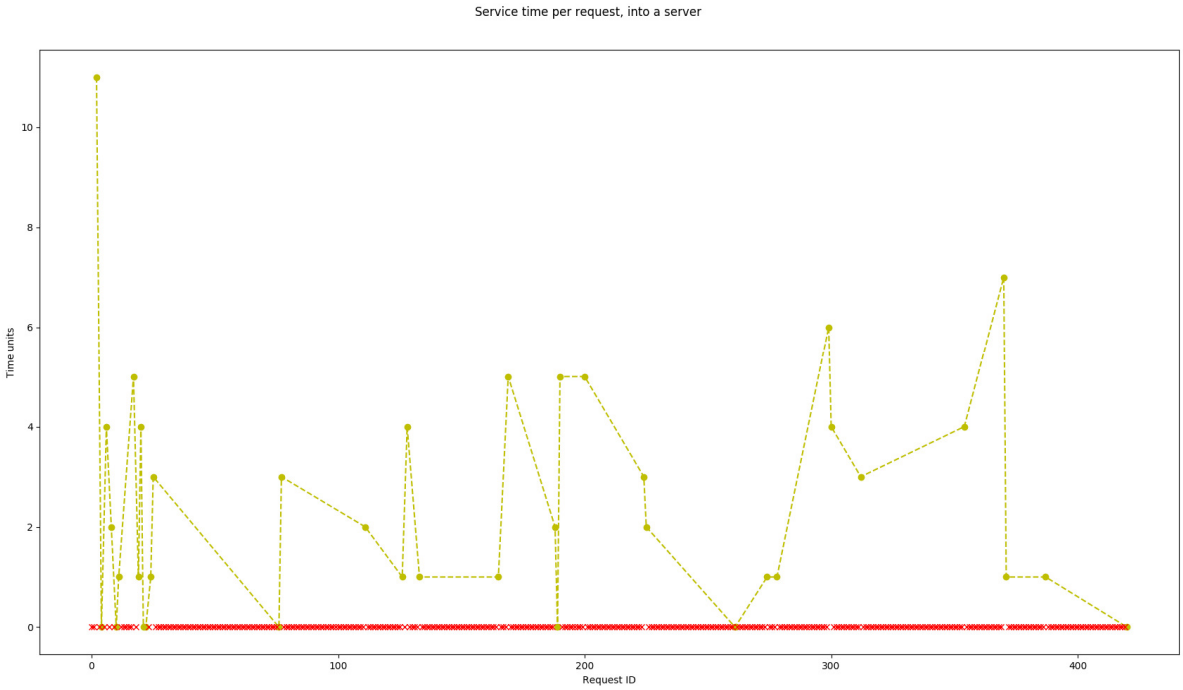
Clients in the whole system



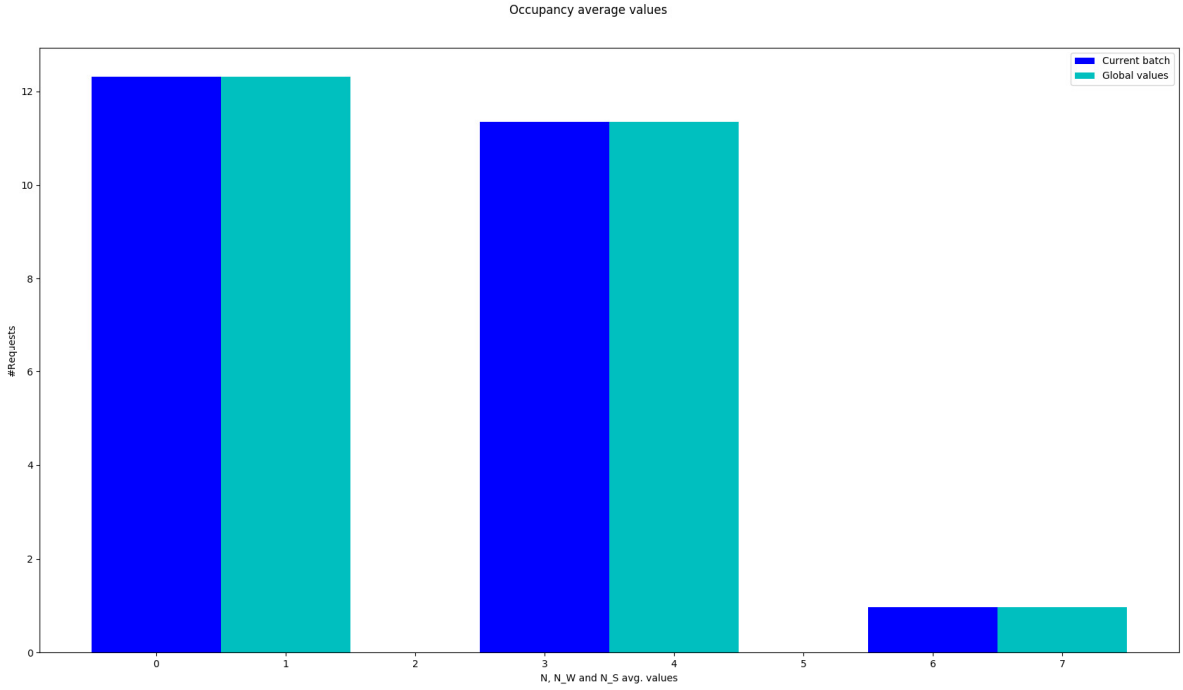
Clients in the queuing line only



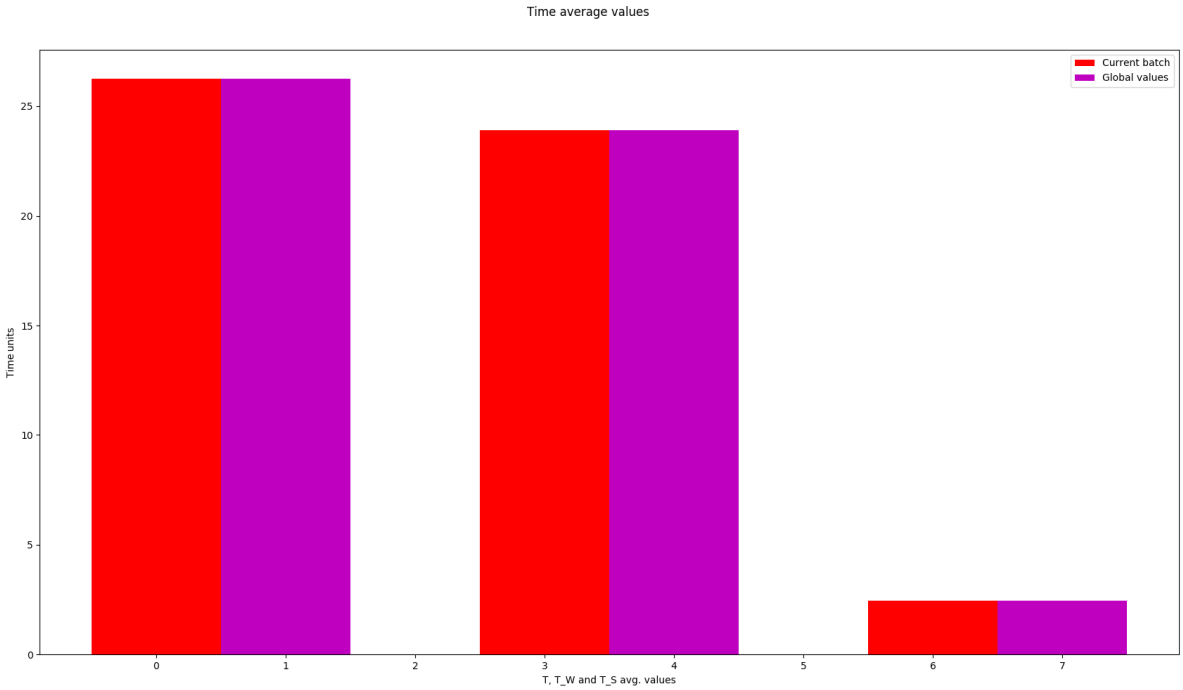
Waiting time per request, in the queuing line (lost ones are marked in red)



Service time per request – First batch (lost ones are marked in red)



Average system (left), waiting line (center) and server (right) occupancies – Last batch and global results



Average traversing (left), waiting (center) and service (right) times – Last batch and global results