Performance Evaluation of Computer Systems and Networks

Project 11 Report

Jacopo Carlon

Contents

1 Introduction

The presented system consists of one queue and two servers, one double as fast as the other.
When both servers are idle, an incoming job is assigned to one server with a probability p, or to the other server with a probability *1-p*.
If only one server is idle, an incoming job is assigned to the idle server.
Otherwise, jobs accumulate in the queue, from where they will be served with a **FCFS** (First-Come First-Served) policy (which is not to be confused with FIFO "First-In First-Out", since not all our servers have the same service times).

Considering that the number of servers is fixed, the only parameters at hand are:

- **probability** p of choosing the first server, in range [0,1];
- **job-interarrival-times** t_n, which are IID RVs
- **service-times** t_{so}, which are IID RVs.

The variables in this system other than the probability p are of temporal measure, therefore they will be used interchangeably according to this conversion :

- $\lambda = \frac{1}{t_n}$
- $\mu_0 = \frac{1}{t_0}$
- $\mu_1 = \frac{1}{t_1}$

From the project specifics, we know that : $t_1 = 2 * t_0$; hence $\mu_0 = \frac{1}{2}\mu_1$.

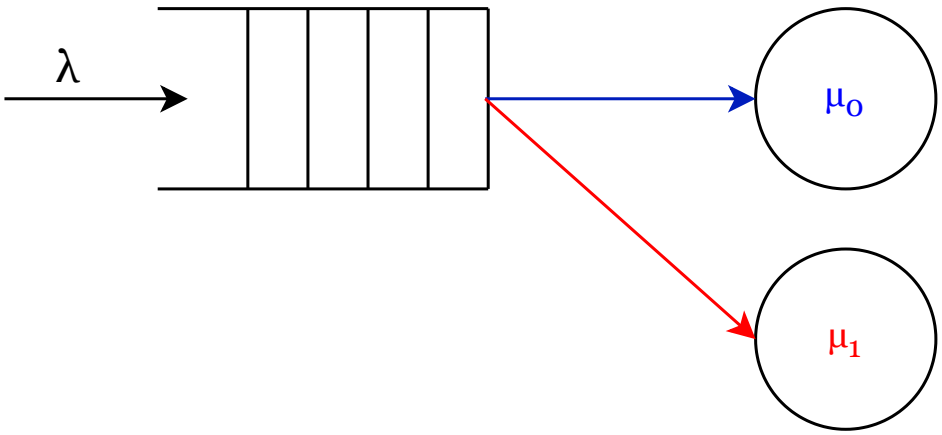


Figure 1: J_Network : queueing system

Observations regarding the stability conditions of this system can be found at section 4.2.

2 Implementation

They system "J_Network" has been implemented in OMNeT++.
The chosen implementation consists of :

- **Spawner** "spawner" :
  This module generates jobs with following a customizable time distribution, and sends them to the queuer.
  This module has a single output-gate.
- **Queuer** "queuer" :
  This module is an *infinite-capacity FCFS queue*. It handles the queueing logic towards the servers.
  This module has both a single input-gate (for incoming jobs from Spawner), and a vector of input-gates for incoming *control-messages* from the servers, as well as a vector of output-gates used to send the jobs to the servers.
- **Server** "server_0" and "server_1" :
  This module receives a job from the queuer, "elaborates" it following a customizable time distribution, then sends the job to the Sink (representing a possible outlet to the external world), as well as a *control-message* to the queuer (declaring that the server is now ready to accept another job).
  A server has a parameter "slowerness" which can be set when initializing the simulation, used to implement the specification "one server double as fast as the other"; this parameter can also be used for degeneracy testing forcing the slow server to be thousands of time slower than the fast server, rendering it effectively non-existant for specific combinations of TN and TSO (see sections 3.2 and 4.1 where such uses were implemented).
  This module has a single input-gate (for incoming jobs from Queuer), an output-gate used for jobs (towards the Sink), and an output-gate used for the control-messages (towards the Queuer).
- **Sink** "sink_0" and "sink_1" :
  This module, in this implementation, simply destroys all the jobs it receives.

It should be noted that all modules have been connected using omnetpp standard "IdealChannel", meaning that in our analysis we will consider the delays of communication between modules to be of zero seconds.

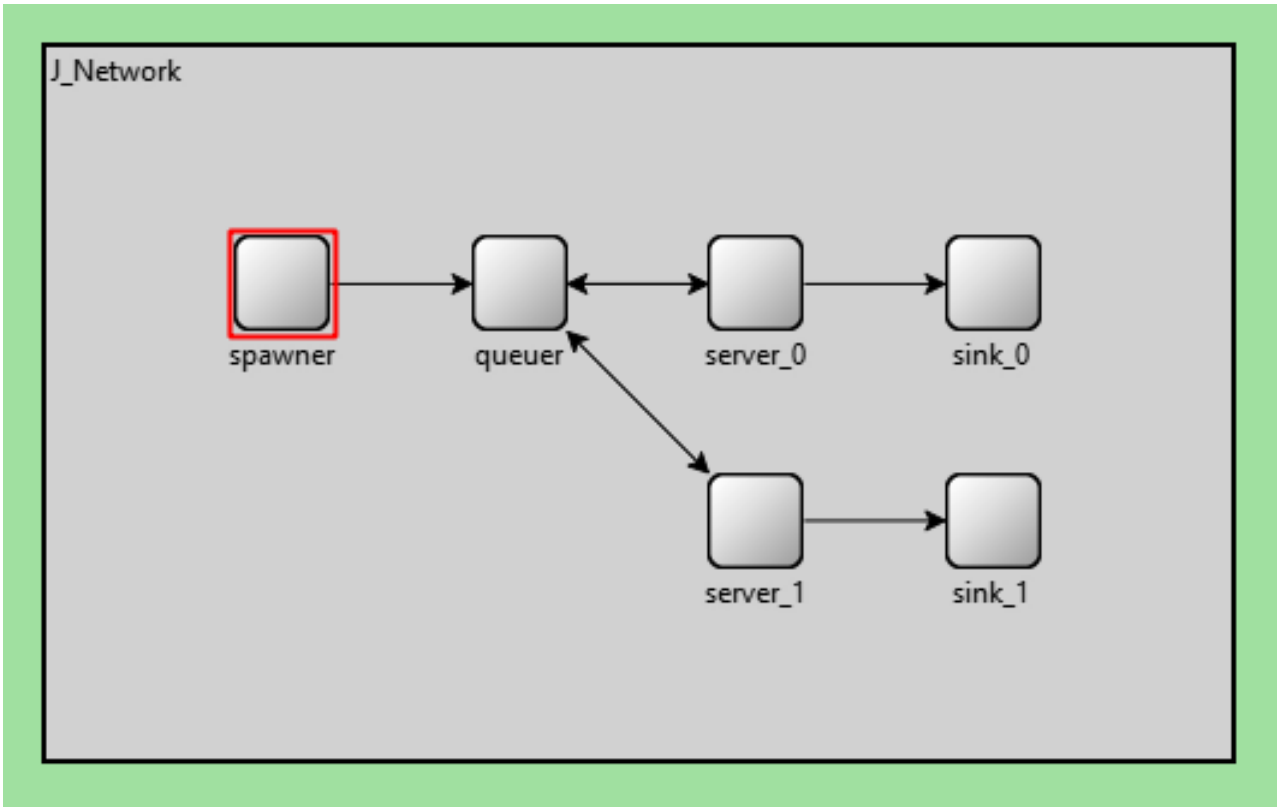


Figure 2: View of the system implementation

The J_Network works is as follows:

1. The spawner generates a job following a customizable time distribution, and sends it to the queuer.

   Since OMNeT++ gives the possibility to define custom **messages**, I have defined the **Job** message (in Job.msg) that holds internally the parameters : t_{qi}, t_{qosi}, t_{so}, which represent respectively *job-interarrival-times*, *queue-service-time* (which for this project is always zero seconds), and *service-times*.

2. As the job is received from the queuer, the `omnetpp::cMessage::setTimestamp()` member function is used in order to save the time of "arrival" in the effective system (since we are using IdealChannels, it's the same as its generation time). The job is immediately inserted in a `omnetpp::cQueue` *jobsQueue_*.

   Upon initialization, the queuer knows the number of its servers, and considers them to be initially free. At all times, the queuer also keeps a bit-variable indicating whether a given server is free (either the simulation just started, or the server has sent a *free* message) or is occupied (this bit is **set** by the queuer upon sending a job to a previously-free server).

3. If neither server is free, then the queuer just does nothing, and is ready to receive other jobs to add in its queue, or any "free" control-message from any server.

   If both servers are free, the queuer generates a random number x (in range $[0, pRange\_]$) using the `omnetpp::cComponent::uniform` function with a specific rnGenerator. If x is above a given $pVal\_$ then the job is sent to server_1 (which is the slow one), otherwise the job is sent to server_0.

   If only one server is free, the job is sent to that server.

   Whenever a job is sent to a server, the queuer sets a bit-flag and considers the server to be "working" and no more "free".

4. When a server receives a job from the queuer, it begins "elaborating" it following a customizable time distribution.

   As the "elaboration" ends, the server measures the **response_time** for that job by calculating the difference between the time it entered the queuer, obtained via the `omnetpp::cMessage::setTimestamp()` function, and the current simulation time obtained via the `omnetpp::setTime()` function. This difference is measured in seconds, and a signal *completedJobSignal_* is emitted.

   The server forwards then the job to the corresponding sink (as far as the specifics of the project go, the job is now uninteresting).

   Moreover, the server sends a control-message to the queuer, indicating that it is now free and can receive new jobs.

5. Upon receiving the control-message, the queuer updates its bit-flags and, if there is any job in queue, then it is sent following the previous algorithm of choice (the job is sent to the now-free-server, due to the fact that there would not be "that" job in queue should the other server also be free).

6. When the sink receives the job, it deletes the job (this part was done simply to simulate an external user of the elaborated job).

To maintain independence between the random-generated values (spawner, Server0, Server1, choice at Querer) *a different pseudo-random generator has been used for each rng purpose* (one for each module).

This implementation could have been made slightly simpler by moving the generating-function of the spawner to the queuer, and the deleting function of the sink to the server(s).

Considering however that the simulation results would not differ (as per experimented), I have chosen to this more "complete" network, as it feels far more realistic from a general networking point of view.

It is also possible to combine the queuer and servers of this network into a single OMNeT++ model, which I have called "Operators". This leads to no difference as far as simulation results go, and could properly represent the real situation of having some data/packets/messages/jobs which are generated by some spawner, sending them into a "box" (my "Operators" module) and receiving, after some time, the elaborated results of said data/... .

It is also worth mentioning that most of the code has been written with scalability in mind, therefore it would be quite easy to increase the number of Servers that the Queuer should manage (the only non-scalable element is the current implementation of choice in case of more-than-one free server), as well as adding an elaboration delay in the queuer (like reading packets in a router), which would basically only require an additional cQueue to be used.

All the code for this project can be found on github here :
`https://github.com/JacopoCarlon/Performance_Evaluation_of_Computer_Systems_and_Networks_PECSN`.

3 Verification

3.1 Code verification

I checked that the code was working as expected with the debug GUI that comes with OMNeT++, the use of OMNeT++ specific containers (such as `cQueue`) and the use of the `EV` prints during simulation with *Qtenv* UI, allowing a step-by-step analysis of proper code execution and logic flow.

It shall be noted that the preliminary analysis of the parameters to use, and specific considerations on estimate before, during and after the testing and data-acquisition is done quite in-lenght at far at later sections, and can be found specifically at Sect. 5.1.1, Sect.5.1.2 and Sect.5.1.3.

The specific choices described in the aforementioned sections will be used all throughout this report.

3.2 Degeneracy test

Degeneracy tests aims to check the system behaviour at extreme values of its parameters.

For this project, the effective parameters are 3 : t_{so}, t_n and p.

Analysis of p has been conducted and is completely described in 4.1, resulting to be (with exception of trivial values of t_{so}, such as $t_{so} < t_n$) almost non-influential, especially as the system gets closer to saturation.

While analyzing the response time for both the constant and exponential cases, the value of generation-time (GT : job-interarrival-times) has been kept fixed to 30 minutes, chosen as reference since it is a value big enough to allow precise measurement for results orders of magnitude above and below, while the service time (ST : service-times) has been varied between strictly less than the GT and greater than the "saturation point" (which was expected and experimentally confirmed to be around 3/2 * GT, i.e. 45 minutes), thus ST has been studied in `range(24..51 step 3` , and also more loosely `range(25--50 step 5`.

Analysis of extreme values has been conducted but will not be graphically shown due to the extreme results, and produced the following results :

- < GT == 0s AND ST != 0s > :
  - both constantTimes and exponentialTimes behave like the cases of ST >> GT , therefore QueueLen goes to +infinity and responseTime goes to +infinity(seconds).

  This behaviour is well within expectation, since both constant and exponential results definitely explode with $ST > GT * \frac{3}{2}$, and in this case $ST >> GT * \frac{3}{2}$.
- < GT!= 0s AND ST == 0s > :
  - if constantTimes : QueueLen == 0 and responseTime == 0s ;
  - if exponentialTimes : QueueLen == 0 and responseTime == 0s ;

  This result is well within expectation, since having < ST == 0s AND QT != 0 > is a specific case of ST << 2*QT, meaning that whichever probability (even when choosing the slow server) it will be far faster than the generator, therefore the response time will depend only on the ST, which is 0s .

3.3 Consistency Test

The consistency test aims at verifying that the system and output react consistently to different configurations that generate the same load as input.

This was done by keeping the same ratio, but changing the reference TN. *In all tests this performed as expected* without any notable change to other numerical results that appear later in this report (this is honestly not quite the surprise, considering that the behaviour over dilation of time metrics should not be -and is not- influential).

3.4 Continuity Test

This test aims to verify that a slight change in the input does not cause *wild* changes on the system behaviour. This is amply measured by all detailed analysis later shown, and was expecially tested with TN=30m, TSO= [(24,25,26), (34,35,36) and (44,45,46)].

In all tests, the system performed as expected : when the system was below saturation it (e.g. the mean ResponseTime) stayed steady for all values, when it was close to saturation or saturated results, RT resulted to be increasing as expected.

It should be noted that these measures apply quite a fine test, being in range of 4% to 2% of the main parameter, the ratio $\frac{t_{so}}{t_n}$.

3.5 Exponential-P50 case

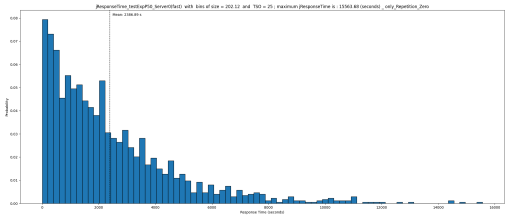
Simulating the network having both *job-interarrival-times* and *service-times* being exponentially distributed RVs, with respective means t_n and t_{so} (obviously, server_0 has mean t_{so}, while server_1 has double that mean), I have tested keeping t_n at 30minutes and varying t_{so}, all the test have been done with p= 0.5.

The following pictures 3 and 4 (for the response times RT of respectively Server0 and Server1) will show that the behaviour of RT with exponentially-distributed-input-times will be "limited" when the ratio of $\frac{t_{so}}{t_n}$ is below or around 1, will increase as it approaches 1.5 and will explode as it is equal or above 1.5 .

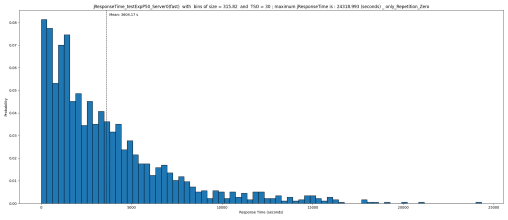
Such results are confirmed by the behaviour of queueLength (QL) measured for the same cases (as can be seen in Figure 5, and are in line with what expected, considering that the above mentioned ratio-steps are most important also in the constant-time analysis that will be expanded later.

The same results (extremely similar) were observed in all 30 iterations done fore each case of this analysis.

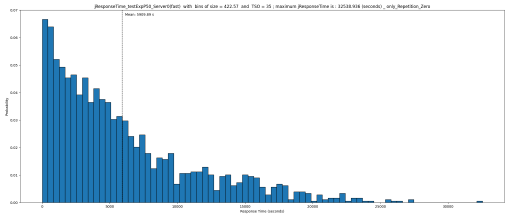
3.5.1 Server_0 (FAST)



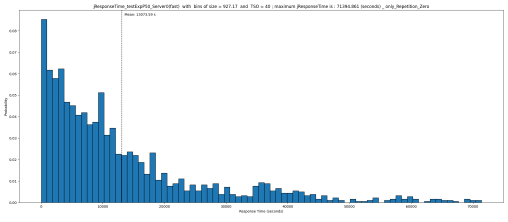
(a) jResTime25ExpP50Server0



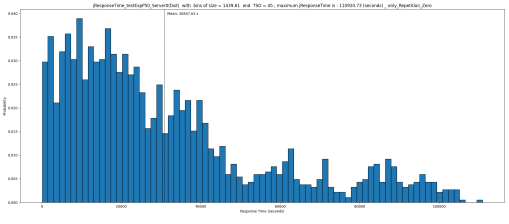
(b) jResTime30ExpP50Server0



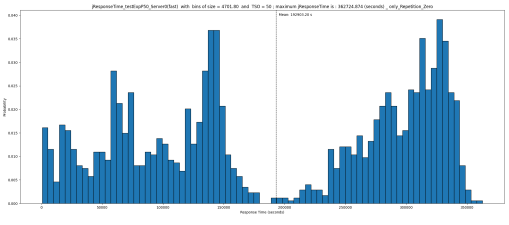
(c) jResTime35ExpP50Server0



(d) jResTime40ExpP50Server0



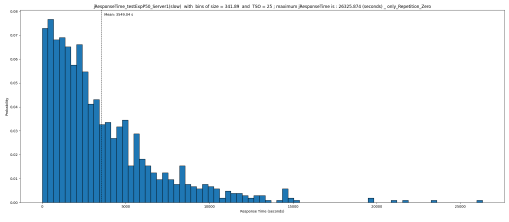
(e) jResTime45ExpP50Server0



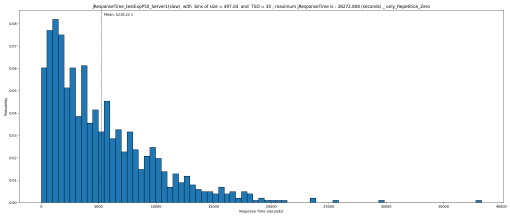
(f) jResTime50ExpP50Server0

Figure 3

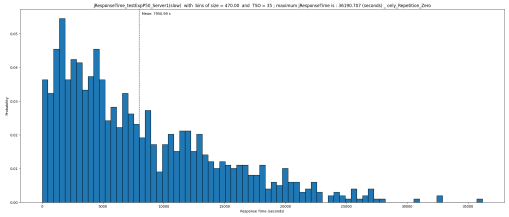
3.5.2 Server_1 (SLOW)



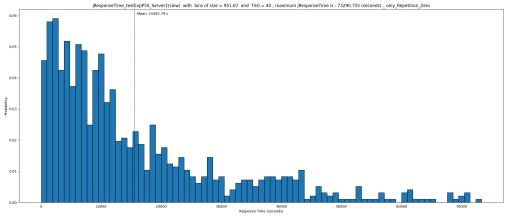
(a) jResTime25ExpP50Server1



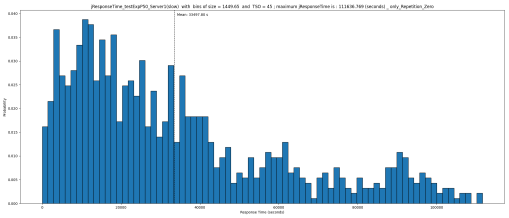
(b) jResTime30ExpP50Server1



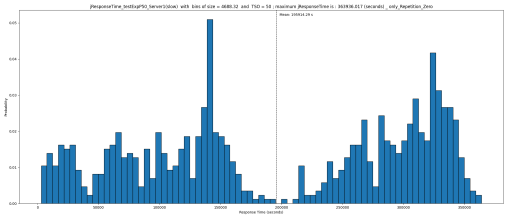
(c) jResTime35ExpP50Server1



(d) jResTime40ExpP50Server1



(e) jResTime45ExpP50Server1

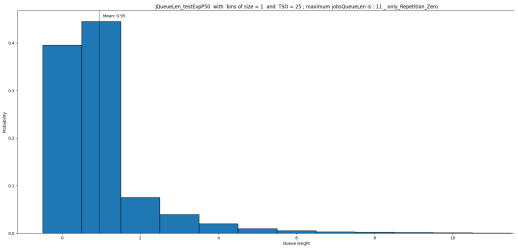


(f) jResTime50ExpP50Server1

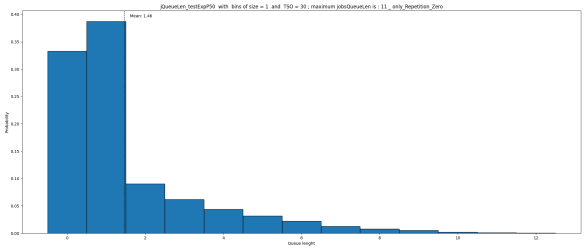
Figure 4

3.5.3 Exponential-P50 : QueueLen

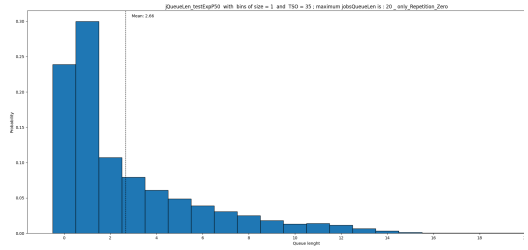
On the x-axis we have the queue-lenght, while on the y-axis we have its probability (measured as occurence of said lenght over all measures, considering that the lenght was measured once every new packet arrival and once for its respective send).



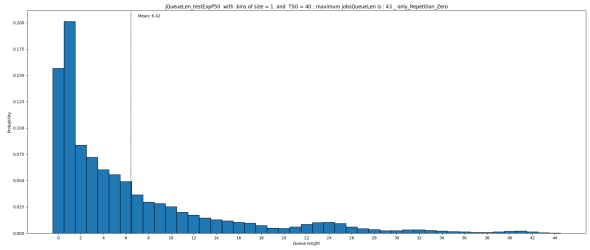
(a) jQLen25ExpP50



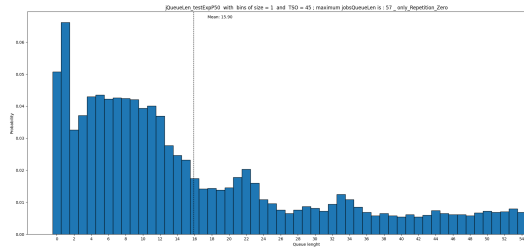
(b) jQLen30ExpP50



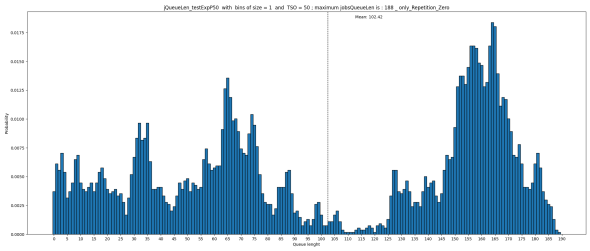
(c) jQLen35ExpP50



(d) jQLen40ExpP50



(e) jQLen45ExpP50



(f) jQLen50ExpP50

Figure 5

As visible from 5, while ST/GT is less than (lt) 1, the QueueLen averages below 1, meaning that jobs tend to not accumulate in the Queue. As the ratio increases lt 1.5, the QL remains limited but its average grows. As we reach and surpass the 1.5 ratio mark (with ST of 45min or 50min) the QL tends to grow indefinitely, and appears bounded only because the simulation run was stopped after 60 days.

The same results were observed in all 30 iterations done fore each case of the above analysis.

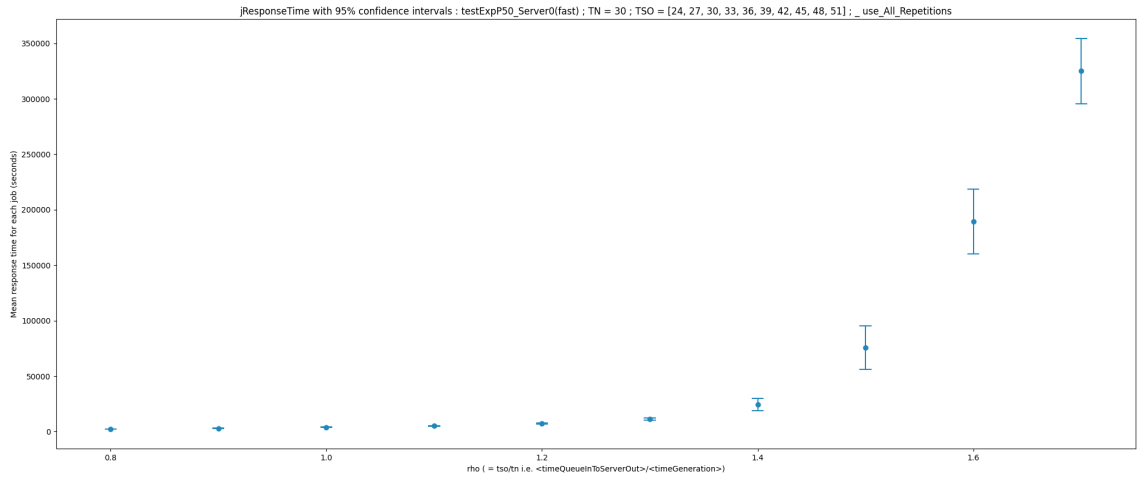
3.5.4 Exponential-P50 Confidence Intervals

For each of the above configurations, I have done *30* repetitions, each with `warmup-period` time of 20 days and `sim-time-limit` of 60 days.

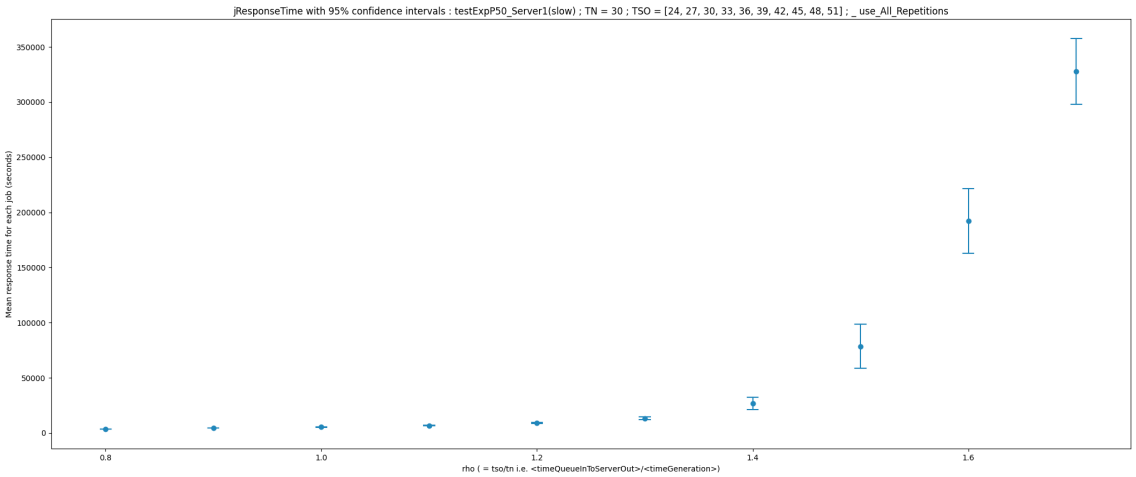
In this analysis, the values of t_{so} were even more finely tested, being in `range(24..51 step3`.

The **mean response time** in the exponential analysis with probability p==50/100 has been thus calculated, with confidence intervals of 95%.

For the following images, on the x-axis we have the ratio $\frac{t_{so}}{t_n}$, while on the y-axis the time measured in seconds.



(a) jConfIntExpP50Server0



(b) jConfIntExpP50Server1

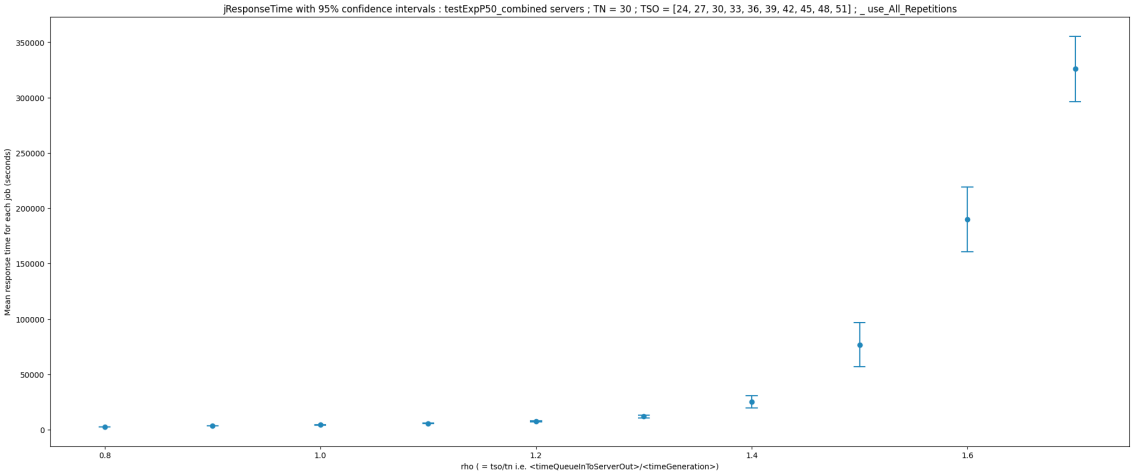
Figure 6: Mean ResponseTime of ExponentialDistributed times with p==50/100 with 95% Confidence Intervals

These images again confirm the above analysis : with ratio << 1.5, the response time is on average limited, and as it approaches 1.5, it begins increasing as the queue just keeps accumulating unserved jobs.

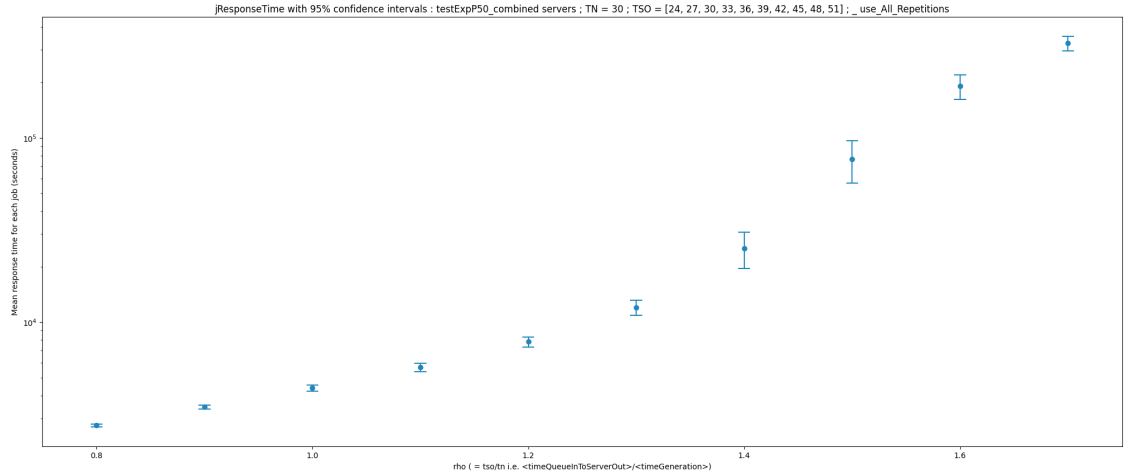
Considering that the graph spans quite some magnitude, I will report the numerical values of the means below :

- server0 :
  - ratio : 0.8 i.e. ST==24min==1440s; mean : 2158.6705381035 s
  - ratio : 0.9 i.e. ST==27min==1620s; mean : 2766.1025662651 s
  - ratio : 1.0 i.e. ST==30min==1800s; mean : 3604.1681520633 s
  - ratio : 1.1 i.e. ST==33min==1980s; mean : 4728.0887925473 s
  - ratio : 1.2 i.e. ST==36min==2160s; mean : 6945.1232022222 s
  - ratio : 1.3 i.e. ST==39min==2340s; mean : 11114.735138158 s
  - ratio : 1.4 i.e. ST==42min==2520s; mean : 18189.243390403 s
  - ratio : 1.5 i.e. ST==45min==2700s; mean : 30847.633052489 s
  - ratio : 1.6 i.e. ST==48min==2880s; mean : 103177.94427011 s
  - ratio : 1.7 i.e. ST==51min==3060s; mean : 237707.92000000 s
- server1 :
  - ratio : 0.8 i.e. ST==24min==1440s; mean : 3456.4955541942 s
  - ratio : 0.9 i.e. ST==27min==1620s; mean : 4007.1814561234 s
  - ratio : 1.0 i.e. ST==30min==1800s; mean : 5218.2237934783 s
  - ratio : 1.1 i.e. ST==33min==1980s; mean : 6654.5392644964 s
  - ratio : 1.2 i.e. ST==36min==2160s; mean : 9159.3797224490 s
  - ratio : 1.3 i.e. ST==39min==2340s; mean : 13115.965964472 s
  - ratio : 1.4 i.e. ST==42min==2520s; mean : 20712.073972428 s
  - ratio : 1.5 i.e. ST==45min==2700s; mean : 33497.802233836 s
  - ratio : 1.6 i.e. ST==48min==2880s; mean : 106074.09369069 s
  - ratio : 1.7 i.e. ST==51min==3060s; mean : 241677.80349355 s

The resulting ***average response time*** is as follows (y axis scale is linear in the first image, logarithmic in the second one):



(a) WeightedConfidenceIntervalCombinedP50Exponential



(b) LogWeightedConfidenceIntervalCombinedP50Exponential

Which has specific timings :

- ratio : 0.8 i.e. ST==24min==1440s; mean : 2753.0287088525783 s
- ratio : 0.9 i.e. ST==27min==1620s; mean : 3459.5841854031173 s
- ratio : 1.0 i.e. ST==30min==1800s; mean : 4387.675307136619 s
- ratio : 1.1 i.e. ST==33min==1980s; mean : 5684.105940762075 s
- ratio : 1.2 i.e. ST==36min==2160s; mean : 7820.137725400642 s
- ratio : 1.3 i.e. ST==39min==2340s; mean : 11981.371627221526 s
- ratio : 1.4 i.e. ST==42min==2520s; mean : 25103.90972204597 s
- ratio : 1.5 i.e. ST==45min==2700s; mean : 76694.1487542752 s
- ratio : 1.6 i.e. ST==48min==2880s; mean : 190190.7765981627 s
- ratio : 1.7 i.e. ST==51min==3060s; mean : 325936.9857610515 s

3.6 Constant-P50 case

Simulating the network having both *job-interarrival-times* and *service-times* being constant values, respectively t_n and t_{so} (obviously, server_0 has mean t_{so}, while server_1 has double that mean), I have tested keeping t_n at 30minutes and varying t_{so}.

The following pictures 8 and 9 (for the response times RT of respectively Server0 and Server1) will show that the following behaviours :

- ratio less or equal to 1 :

  For both servers, the responseTime is exactly the ServiceTime of that server.

- ratio in range (1, 1.5] (i.e strictly above 1, and below or equal to 1.5) :

  - Server 0 has half the times a responseTime equal to its speed, and half the times a slightly higher responseTime, due to the fact over the periodic behaviour of the system, for some ranges of time depending on the ratio there will be 1 job in queue for more than an instant. This will be analyzed more in depth later at Section 4.3;

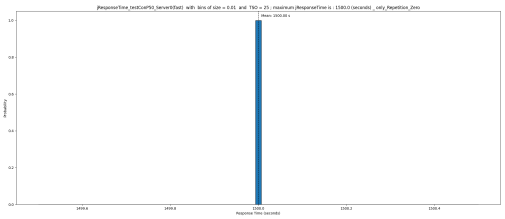
  - Server 1 has response time equal to its ServiceTime, due to the fact that, in conjunction with the work of the fast server, it manages to keep the mean bounded by instantly starting and completing any job it receives, and having some leftover time due to how p/q is composed.

- ratio strictly above 1.5 :

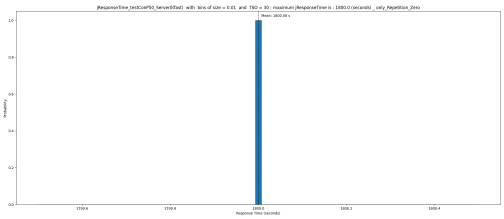
  both servers' response times increase due to them not being able to properly serve the input generation : int these case we have that : $\frac{1}{t_n} > \frac{1}{t_{so}} + \frac{1}{2*t_{so}}$ and this explosion of response time is perfectly expected.

The exact same results were observed in all 30 iterations done fore each case of this analysis.

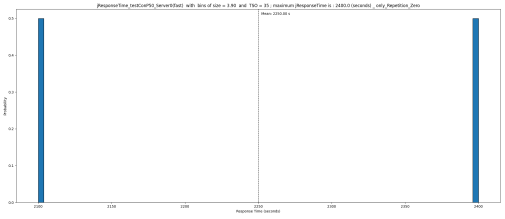
3.6.1 Server_0 (FAST)



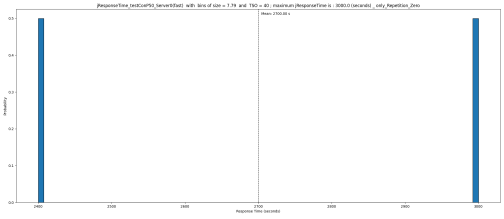
(a) jResTime25ConP50Server0



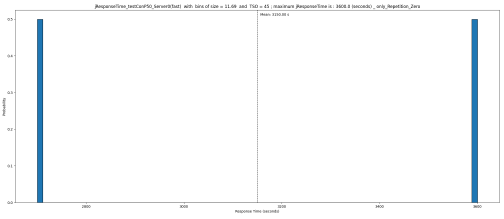
(b) jResTime30ConP50Server0



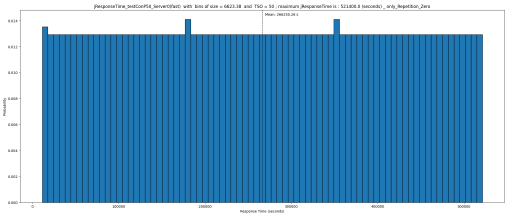
(c) jResTime35ConP50Server0



(d) jResTime40ConP50Server0



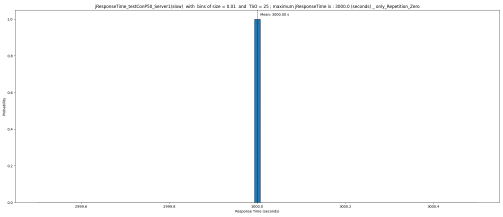
(e) jResTime45ConP50Server0



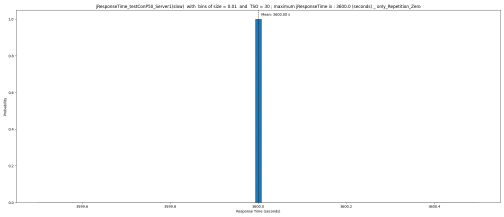
(f) jResTime50ConP50Server0

Figure 8

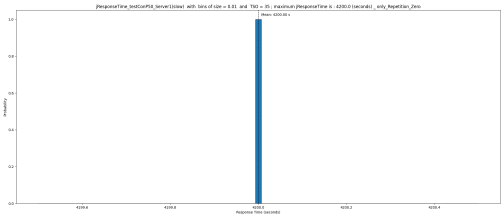
3.6.2 Server_1 (SLOW)



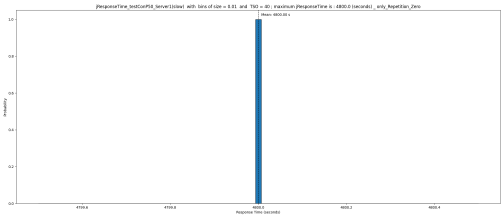
(a) jResTime25ConP50Server1



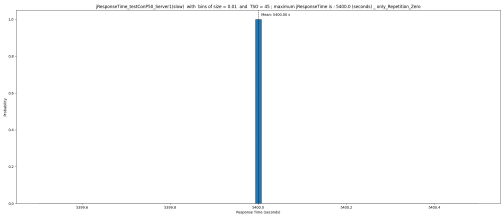
(b) jResTime30ConP50Server1



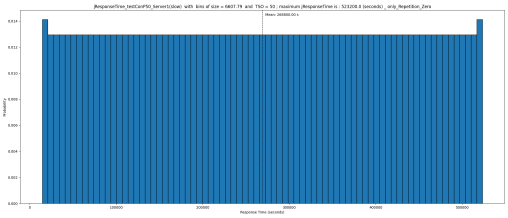
(c) jResTime35ConP50Server1



(d) jResTime40ConP50Server1



(e) jResTime45ConP50Server1

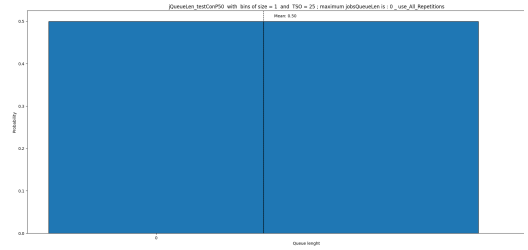


(f) jResTime50ConP50Server1

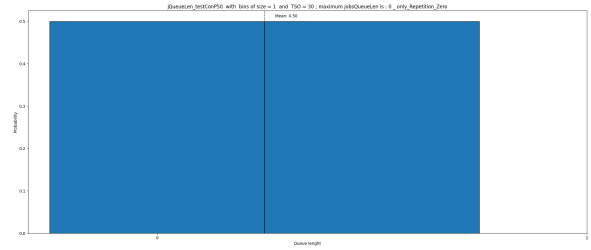
Figure 9

3.6.3 Constant-P50 : QueueLen

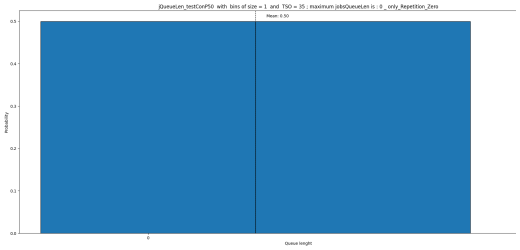
On the x-axis we have the queue-lenght, while on the y-axis we have its probability (measured as occurence of said lenght over all measures, considering that the lenght was measured once every new packet arrival and once for its respective send).



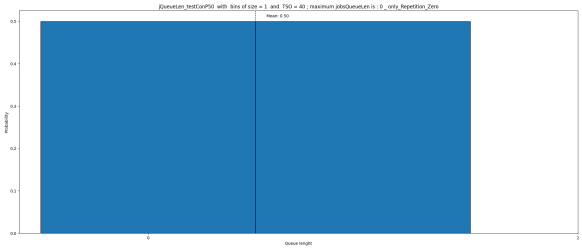
(a) jQLen25ConP50



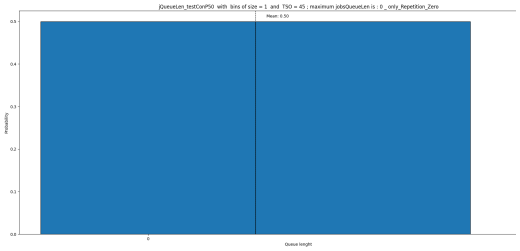
(b) jQLen30ConP50



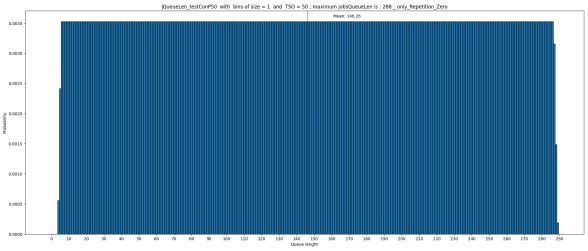
(c) jQLen35ConP50



(d) jQLen40ConP50



(e) jQLen45ConP50



(f) jQLen50ConP50

Figure 10

As visible from 10, while ST/GT is less than (lt) 1.5, the QueueLen is always either 0 or 1. This makes sense since in suck cases $\frac{1}{t_n} <= \frac{1}{t_{so}} + \frac{1}{2*t_{so}}$, meaning that the system is expected to (and does) behave within equilibrium, since all events take up only constant times.

As we surpass the 1.5 ratio mark (with ST of 50min) the QL starts to grow indefinitely, and appears bounded only because the simulation run was stopped after 60 days. This is in line with the previous analysis regarding responseTimes in ConstantTimes.

The exact same results were observed in all 30 iterations done fore each case of the above analysis.

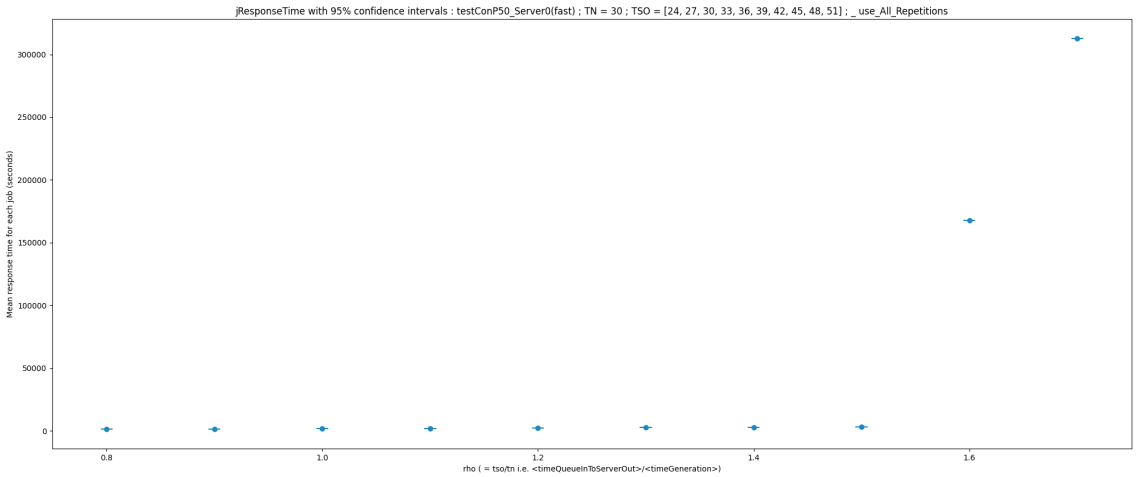
3.6.4 Constant-P50 Confidence Intervals

For each of the above configurations, I have done *30* repetitions, each with `warmup-period` time of 20 days (which in this case was way abundant) and `sim-time-limit` of 60 days.

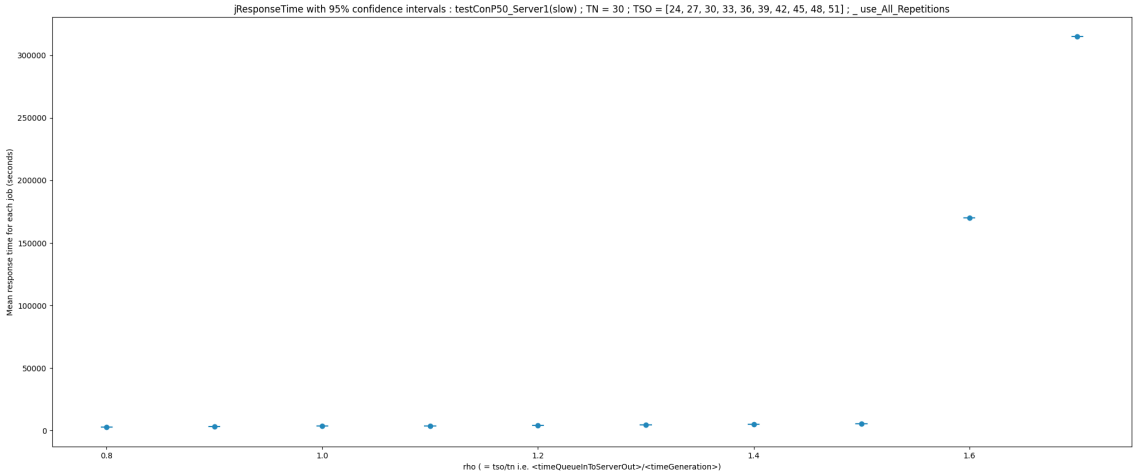
In this analysis, the values of t_{so} were even more finely tested, being in `range(24..51 step3)`.

The **mean response time** in the constant analysis with probability p==50/100 has been thus calculated, with confidence intervals of 95%.

For the following images, on the x-axis we have the ratio $\frac{t_{so}}{t_n}$, while on the y-axis the time measured in seconds.



(a) jConfIntConP50Server0



(b) jConfIntConP50Server1

Figure 11: Mean ResponseTime of ExponentialDistributed times with p==50/100 with 95% Confidence Intervals

These images again confirm the above analysis : with ratio lower or equal 1.5, the response time is on increasing but limited, and explodes for ratios strictly above 1.5, perfectly in line with the previous considerations.

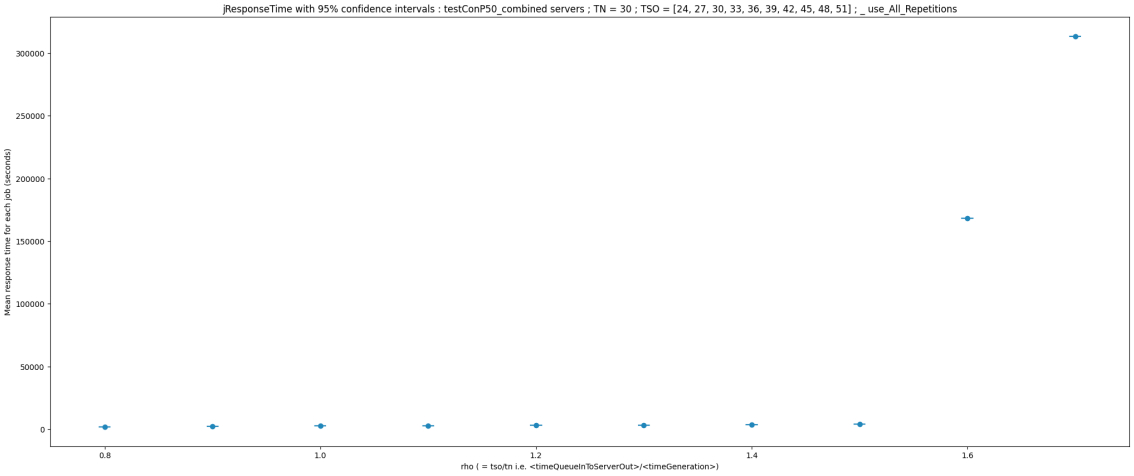
Considering that the graph spans quite some magnitude, I will report the numerical values of the response times (which are identical to their means) below :

- server0 :
  - ratio : 0.8 i.e. ST==24min==1440s; mean : 1440.0 s
  - ratio : 0.9 i.e. ST==27min==1620s; mean : 1620.0 s
  - ratio : 1.0 i.e. ST==30min==1800s; mean : 1800.0 s
  - ratio : 1.1 i.e. ST==33min==1980s; mean : 2070.0 s
  - ratio : 1.2 i.e. ST==36min==2160s; mean : 2340.0 s
  - ratio : 1.3 i.e. ST==39min==2340s; mean : 2610.0 s
  - ratio : 1.4 i.e. ST==42min==2520s; mean : 2880.0 s
  - ratio : 1.5 i.e. ST==45min==2700s; mean : 3150.0 s
  - ratio : 1.6 i.e. ST==48min==2880s; mean : 167580.0 s

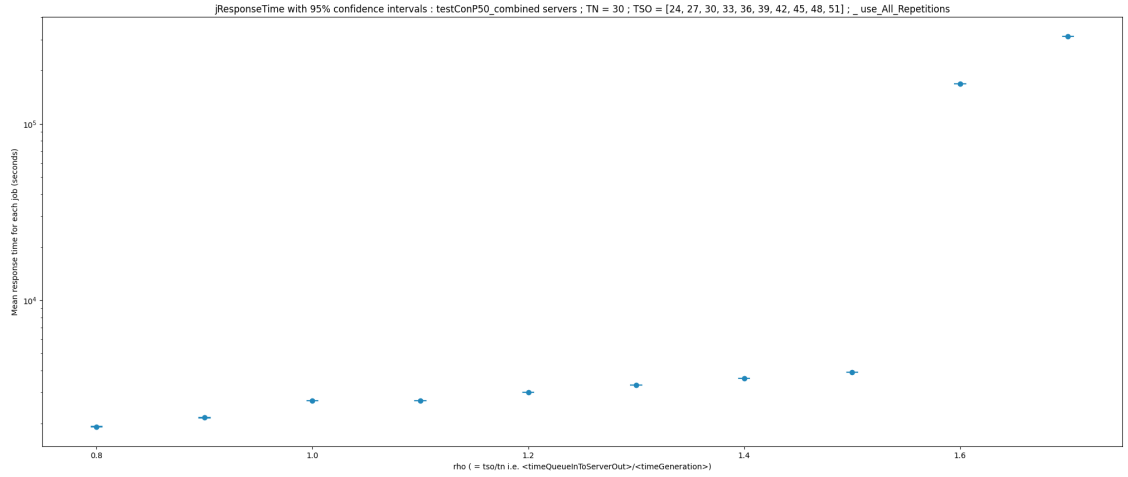
  - ratio : 1.7 i.e. ST==51min==3060s; mean : 312654.0 s

- server1 :
  - ratio : 0.8 i.e. ST==24min==1440s; mean : 2880.0 s
  - ratio : 0.9 i.e. ST==27min==1620s; mean : 3240.0 s
  - ratio : 1.0 i.e. ST==30min==1800s; mean : 3600.0 s
  - ratio : 1.1 i.e. ST==33min==1980s; mean : 3960.0 s
  - ratio : 1.2 i.e. ST==36min==2160s; mean : 4320.0 s
  - ratio : 1.3 i.e. ST==39min==2340s; mean : 4680.0 s
  - ratio : 1.4 i.e. ST==42min==2520s; mean : 5040.0 s
  - ratio : 1.5 i.e. ST==45min==2700s; mean : 5400.0 s
  - ratio : 1.6 i.e. ST==48min==2880s; mean : 169920.0 s
  - ratio : 1.7 i.e. ST==51min==3060s; mean : 315000.0 s

The resulting ***average response time*** is as follows (y axis scale is linear in the first image, logarithmic in the second one):



(a) WeightedConfidenceIntervalCombinedP50Constant



(b) LogWeightedConfidenceIntervalCombinedP50Constant

Which has specific timings :

- ratio : 0.8 i.e. ST==24min==1440s; mean : 1921.6435272541357 s
- ratio : 0.9 i.e. ST==27min==1620s; mean : 2161.848968160903 s

- ratio : 1.0 i.e. ST==30min==1800s; mean : 2700.0 s
- ratio : 1.1 i.e. ST==33min==1980s; mean : 2700.0 s
- ratio : 1.2 i.e. ST==36min==2160s; mean : 3000.0 s
- ratio : 1.3 i.e. ST==39min==2340s; mean : 3300.0 s
- ratio : 1.4 i.e. ST==42min==2520s; mean : 3600.0 s
- ratio : 1.5 i.e. ST==45min==2700s; mean : 3899.943522767383 s
- ratio : 1.6 i.e. ST==48min==2880s; mean : 168360.0 s
- ratio : 1.7 i.e. ST==51min==3060s; mean : 313436.2241793434 s

4 Deterministic case

In this section I will present the results obtained from the analytical modelling of the system in a deterministic scenario where all the parameter of the system are in fact constants.

4.1 Considerations Regarding Probability *p*

It must first be noted the following :

- The probability p which determines the preference of Server_0 to Server_1 only matters when both servers are free, in which case the queue must also be empty!

  This is because should the queue not be empty the jobs would be served to a free server, thus making `<both server are free>==False`.
- Since the queue begins at t_0 empty with both servers free, should it become empty again with both servers free at a time $t_1 > t_0$, then the following behaviour will be within the same definition of the expectable behaviour (which depend on p) at t_0.

It should also be noted the following behaviour in relation with the probability p (which is obviously bounded in [0,1]):

- $p == 0$:

  A) **(&) If** $t_n > 2 * t_{so}$ (the right-side of the equation being the service time of Server_1), then the ***ResponseTime*** **will be exactly** $2 * t_{so}$, since only that server will be chosen, and whenever a job enters the queue will see both servers free, thus choosing with the probability==1 the slow Server_1.

  In this case **A**, therefore, ***QueueLength*** **will be == 0** (as soon as a job enters the Queuer, it immediately exits to Server_1).

- $p == 1$:

  B) **(&) If** $t_n > t_{so}$ (the right-side of the equation being the service time of Server_0), then the ***ResponseTime*** **time will be exactly** t_{so}. This is obtained following the same reasoning as above.

  In this case **B**, therefore, ***QueueLength*** **will be == 0** (as soon as a job enters the Queuer, it immediately exits to Server_0).

- should the case be neither **A** nor **B**, or in general have ***p*** **not null nor 1**, then NOT ALWAYS will an incoming job find both servers free.

  Whenever the t_{so} is such that not both servers are free (i.e. $t_{so} > t_n$), the importance of p will lessen: having *QueueLenght* not identically zero means that the following job will be sent to the other server.

  In this configurations, the ResponceTime will be *mostly* defined by the ratio $\frac{t_{so}}{t_n}$, rather than by p.

The above observations for $t_{so} <= t_n$ clearly match the numerical results obtained at 3.6.4 .

4.2 Stability Condition

In light of the considerations above (at section 4.1), I will now study the behaviour of the system with a generic probability p..

It should be remembered that this study will focus $t_{so} > t_n$, since the otherwise the solution is quite trivial and has been discussed above at section 4.1 .

The system will be stable (with *constant interarrival-times (t_n) AND constant service-times (t_{so} and $2 * t_{so}$ for respectively Server_0 and Server_1* if and only if the input rate is less than the output rate, i.e. :

$$\lambda <= \mu_0 + \mu_1 \tag{1}$$

$$\frac{1}{t_n} <= \frac{1}{t_{so}} + \frac{1}{2 * t_{so}} \tag{2}$$

$$\frac{1}{t_n} <= \frac{3}{2 * t_{so}} \tag{3}$$

$$t_{so} <= \frac{3}{2} t_n \tag{4}$$

Proof - (The system is stable $\Rightarrow t_{so} <= \frac{3}{2}t_n$) $\Leftrightarrow$ ($t_{so} > \frac{3}{2}t_n \Rightarrow$the system is unstable).
The relation above comes from the *input-output rate analysis of the system Queuer-Servers* : should the generation(==input) rate be any faster than the service(==output) rate, the service would clearly overload.

This can also easily be verified by showing that if $t_{so} == \frac{3}{2}t_n$ then the servers are working all at the same time, and the queueLenght(which is the `backlog` at a given time) is bounded *in a periodic manner*.

This inevitably means that, should the relative inputrate increase any more, which is the same as the resulting serverrate be any lesser, i.e should t_{so} increase by any measure, then the jobs would begin stacking in queue, meaning that the network would be unstable and both QL and RS would indefinitely increase.

4.3 Discrete Analysis with $t_{so} = t_n * 3/2$

Let t_n = 2s, and t_{so} = 3s, meaning Server_0 (S0) serviceTime is 3s, and Server_1 (S1) serviceTime == 6s. The value OCC will be 0b00=0 if both servers are free, 0b01=1 if S0 is occupied, 0b10=2 if S1 is occupied, 0b11=3 if both servers are occupied.

At t=0^- seconds : OCC=0 and QL=0.
In this first *Case Analysis*, the first server chosen will be S0.

0. t=0s, Job0 arrives, is assigned randomly to S0
   - OCC becomes 1 : OCC == 1
   - QL has become 1 and then 0 instantly
1. t=1s,
   - OCC == 1
   - QL == 0
2. t=2s, Job1 arrives, must be assigned to S1
   - OCC becomes 3
   - QL has become 1 and then 0 instantly : QL == 0
3. t=3s, S0 finishes Job0
   - OCC becomes 2 : OCC == 2
   - QL == 0
4. t=4s, Job2 arrives, must be assigned to S0
   - OCC effectively is 3 : OCC == 3
   - QL has become 1 and then 0 instantly : QL == 0
   - !!! S0 has just started a job, S1 has done $\frac{1}{3}$ of its current job, QL==0
5. t=5s,
   - OCC == 3
   - QL == 0
6. t=6s, Job3 arrives, but OCC is 3 so it stays in queue !!!
   - OCC == 3
   - QL = 1
7. t=7s, S0 finishes Job2, and it takes Job3
   - OCC effectively is 3 : OCC == 3
   - QL= 0
8. t=8s, S1 finishes Job1 and Job3 arrives, it must be assigned to S1
   - OCC effectively is 3 : OCC == 3
   - QL has become 1 and then 0 instantly : QL == 0
9. t=9s,
   - OCC == 3
   - QL == 0
10. t=10s, S0 finishes Job3 and Job5 arrives, it must be assigned to S0
    - OCC effectively is 3 : OCC == 3
    - QL has become 1 and then 0 instantly : OCC == 0
    - !!! S0 has just started a job, S1 has done $\frac{1}{3}$ of its current job, QL==0

11. t=11s,
    - OCC == 3
    - QL == 0

12. t=12s, Job6 arrives, but OCC is 3 so it stays in queue !!!
    - OCC == 3
    - QL = 1

13. t=13s, S0 finishes Job5, and it takes Job6
    - OCC effectively is 3 : OCC == 3
    - QL= 0

14. t=14s, S1 finishes Job1 and Job3 arrives, it must be assigned to S1
    - OCC effectively is 3 : OCC == 3
    - QL has become 1 and then 0 instantly : QL == 0

It can easily be noted that ever since t=10s the pattern is repeating identically to that from t=4s.

It can also be noted, and was the **thesis** of the above proof, that **ever since t=4s both the servers are at all times working**, and this allows to overall keep the queue from increasing (it is periodically the same!).

Should the serviceTimes be any longer, and this equilibrium will be clearly broken.

It must also be noted that in this condition the probability only influenced the choice at t=0s.

I will now begin again the same Case analysis choosing the other server, and proving that effectively for these values of $\frac{t_{so}}{t_n}$, p is overall non-influential :

0. t=0s, Job0 arrives, is assigned randomly to S1
    - OCC becomes 2 : OCC == 2
    - QL has become 1 and then 0 instantly : QL == 0

1. t=1s,
    - OCC == 2
    - QL == 0

2. t=2s, Job1 arrives, must be assigned to S0
    - OCC becomes 3
    - QL has become 1 and then 0 instantly : QL == 0
    !!! S0 has just started a job, S1 has done $\frac{1}{3}$ of its current job, QL==0

3. t=3s,
    - OCC == 3
    - QL == 0

4. t=4s, Job2 arrives, but OCC is 3 so it stays in queue !!!
    - OCC == 3
    - QL == 1

5. t=5s, S0 finishes Job1, and it takes Job2
    - OCC effectively is 3 : OCC == 3 - QL= 0

6. t=6s, S1 finishes Job0 and Job3 arrives, it must be assigned to S1
    - OCC effectively is 3 : OCC == 3
    - QL has become 1 and then 0 instantly : QL == 0

7. t=7s,
    - OCC == 3
    - QL == 0

8. t=8s, S0 finishes Job2 and Job4 arrives, it must be assigned to S0
    - OCC effectively is 3 : OCC == 3
    - QL has become 1 and then 0 instantly : QL == 0
    !!! S0 has just started a job, S1 has done $\frac{1}{3}$ of its current job, QL==0

9. t=9s,
   - OCC == 3
   - QL == 0

10. t=10s, Job2 arrives, but OCC is 3 so it stays in queue !!!
    - OCC == 3
    - QL == 1

Again, there is a clear pattern (completely identical status of the system) between t=2s and t=8s.

There are a number of notable facts from these analysis regardin $t_{so} == \frac{3}{2}t_n$

- The system behaves periodically with period 2*t_{so} (after a short beginning)
- During all periods the servers are at all times at their maximum throughput and this allows to keep the behaviour of response time and queue lenght periodical, thus bounded;
- The queue lenght stays at 1 (for longer than an instant) for exactly $\frac{t_n}{2}$ times every period, and only one job is delayed every period i.e. only one job every 3 jobs. Furthermore, the delayed job is always served by Server_0.

The last considerations allows us to observe that every period (which has lenght 2*t_{so}), **Server_0** serves 2 jobs, one in exactly t_{so} time, and the other in $t_{so} + \frac{t_n}{2}$, meaning that the average response time is :

$$\frac{t_{so} + (t_{so} + \frac{t_n}{2})}{periodLenght} \tag{5}$$

$$\frac{\frac{7}{3}t_{so}}{2 * t_{so}} \tag{6}$$

$$\frac{7}{6}t_{so} \tag{7}$$

This value **perfectly matches** the numerical result visible in Figure 8e and numerically at Section 3.6.4 where Server_0 with ST==45min==2700s has a mean response time of 3150s, which is exactly $\frac{7}{6} * 2700s$.

Regarding **Server_1**, we have observed that in every period it is fully working, but without delays, meaning that the expected response time is to be its own ServiceTime == $2 * t_{so}$ == 3*t_n, which also perfectly matches the numerical results in Figure 9e and in Section 3.6.4.

In light of these results, and knowing that every period (i.e. every 3 jobs) S0 serves two jobs, while S1 only one, we can estimate the **average responseTime** of jobs to be :

$$\frac{7}{6}t_{so} * \frac{2}{3} + 2 * t_{so} * \frac{1}{3} \tag{8}$$

$$\frac{13}{9}t_{so} \tag{9}$$

$$\frac{13}{6}t_n \tag{10}$$

Again, this result perfectly matches the simulation results of meanResponseTime for $t_{so} == \frac{3}{2}t_n$, as per Figure 12a in section 3.6.4 : $\frac{13}{9}t_{so} = \frac{13}{9}45min = \frac{13}{9}45 * 60s = 3900$ seconds.

4.3.1 Mean Number of Jobs

Since *no jobs are created nor destroyed within the Queuer-Servers system* , and since we have a Constant InterArrival rate $\lambda = \frac{1}{t_n}$, then we can use Little's Law to estimate the mean number of jobs in the Queuer-Servers system as :

$$E[responseTime] = \frac{E[N]}{E[arrivalRate]} \tag{11}$$

$$E[N] = \frac{13}{6} t_n * \frac{1}{t_n} \tag{12}$$

$$E[N] = \frac{13}{6} \tag{13}$$

Knowing that at ratio 1.5 both servers are at all times occupied, and the queue has 1 job waiting for exactly one sixth of each period, this result is perfectly aligned with the theoretical analysis and the numerical values.

5 Exponential case

5.1 Preliminary Analysis

5.1.1 Warm up Time

Considering that a system where both the generator and servers operating times follow exponential distributions, it was necessary to decide a warm up period before analyzing any simulation result.

Waiting a warm-up time allows a correct measurement, and later estimate, of the asyntotic behaviour of the network with regards to time (or number of jobs, since they are correlated by the generator).

Considering that there are exponential distribution, oscillation is expected to appear, *but*, if the parameters of the system allow stability, then it should be bounded.

Since the lenght of the queue in the Queuer is related to the responceTime experienced by each job (if the queue is empty, RT depends only on ST, otherwise jobs need to wait the queue), a good estimate of *cruise-rate-reached* of the system was achieved by observing the mean number of jobs in queue.

It should be mentioned that these measures were taken also at varying probability p, however the considerations of 4.1 regarding the behaviour of the system are perfectly valid, and numerical results have proven that also with exponential distribution (as far as the interesting/ed ratios are concerned) the value of probability is essentially *non influential*.

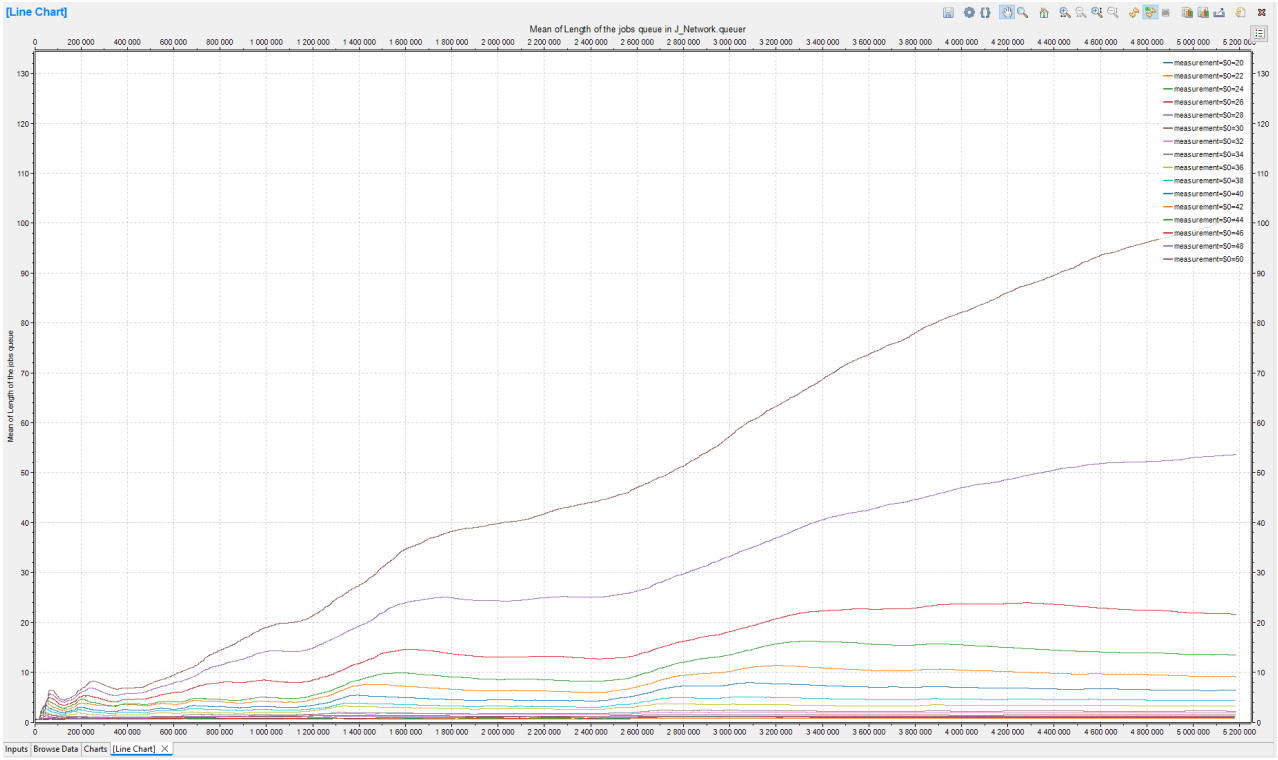


Figure 13: WarmupStudyJQueueLenghtProb50Mean:DetailedRange(20..50step2)

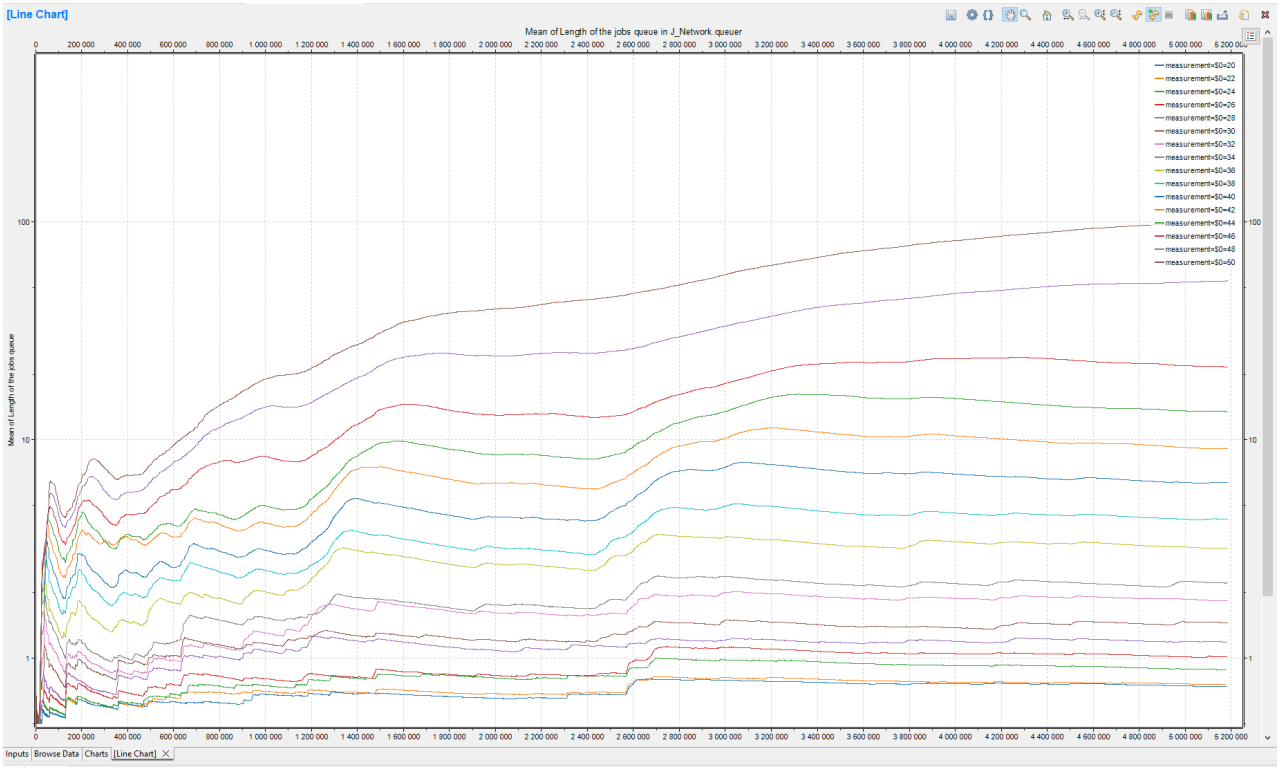


Figure 13: WarmupStudyJQueueLenghtProb50MeanLog:DetailedRange(20..50step2)

The above plots are obtained from testing the exponential problem with different TSOs (in *range(20..50 step2)min*) with 0s of warm up for a period of 60d.

The following plots are obtained again over a period of 60 days with 0s of warm up, over 30 iterations of only TSO==30min, a condition in which the servers seem to "on average" keep up with the spawner.

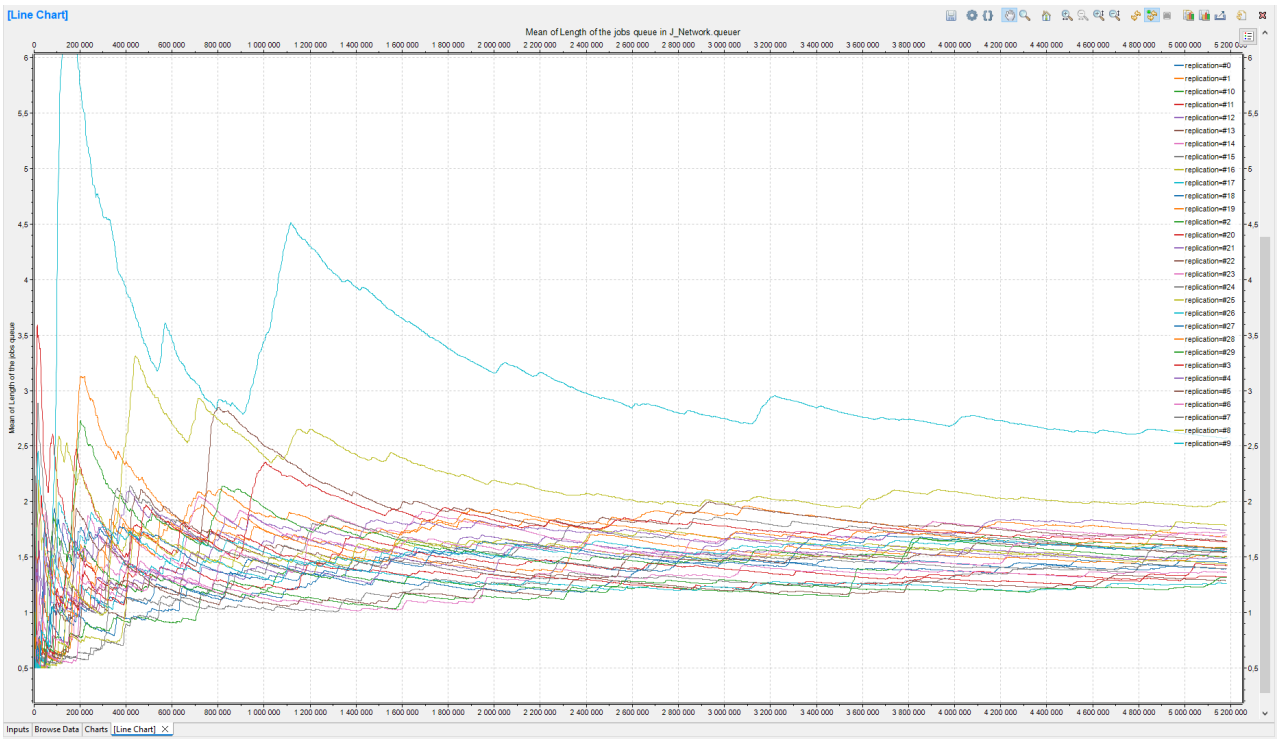


Figure 14: WarmupStudyJQueueLenghtProb50MeanTSO30

Figure 14: WarmupStudyJQueueLenghtProb50MeanTSO30Log

Keeping in mind the fact that most of the analysis done were focused on TSOs in range [25,50]min, which appear to start converging after 100.000s and seem quite stable after 1.400.000 seconds, value at which also the non-saturated system is stabilized, I decided to use a most conservative *warm up time of 20days==1.728.000seconds.*

This value has been used for all plots and numerical results in this analysis throughout all this project report, except for those values or plots who specifically indicate otherwise (and for those whose x-axis is a time axis).

5.1.2 Time dimensions

Considering that the problem at hand evolves depending only on $\frac{t_{so}}{t_n}$ and p, there is no particular need to have the "time unit" big or small other than convenience.

Since using small numbers could incur numerical problems due to approximation, I have decided to use t_n of 30 minutes and t_{so} usually ranging in [20,51]minutes.

5.1.3 Probability

As far as Probability is concerned, (p is used by the Querer module to choose which server to forward to a job), it is saved as a combination of `int probVal = default(50)` and `iint probMax = default(100)`, both parameters modifiable at the omnetpp.ini file for the different tests.

5.2 Response Times

Most of the consideration on response time are in line with Section 4. Looking at the plots in Section 3.5 it can easily be observed, as previously mentioned, that both Server0's and Server1's responseTime and queueLenght are quite stable for TSOs lower or equal to 40 (in particular see figures 3d and 4d for RT, and figure 5d for QL).

Observing the overall mean response time for Server0 in figure 6a, for Server1 in 6b, and more in particular the *Mean Response Time* with y-axis in logarithmic scale at Figure 7b, it can easily be noted that for ratios ($\frac{t_{so}}{t_n}$) lower than 42 the response time follows quite closely a straight line, suggesting an exponential, and later it seems to increase, having definitely reached saturation already at ratio 1.5.

This behaviour is within expectation, considering that we have calculated that the saturation point for the constant-time is 1.5 .

The exponential distribution inevitably will tend to surpass times, due to the simple fact that :

- if for an interval of time the ST is lower than the GT, the queue will overall decrease or stay empty

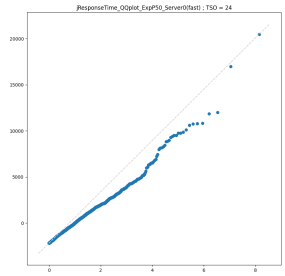
- if for an interval of time the ST is in range [1, 1.5), the queue will stay about emtpy (similar to the deterministic case, but for a limited time interval !)
- if for an interval ot time the ST is larger than 1.5 GT, then the queue will accumulate jobs and the RT will increase

Now, since we have both the generator and servers have exponential distribution, and since the queue in the middle affects packets (that have to wait if a server is slow once), the system overall will tend to fill more than to empty. This is mostly due to the fact that sequential nature of the problem means that a one-time very long ServiceTime definitely increases the responseTime for many other jobs !

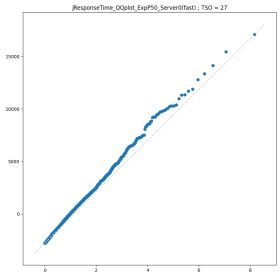
It could be said that this behaviour is somewhat similar to the *Drunkard's Walk probability problem.*

5.2.1 ResponseTime QQ plots

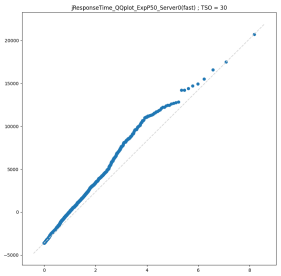
In the following QQplots the response times will be *tested against the exponential* distribution :



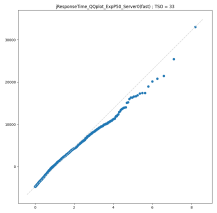
(a) P 0.5, TSO 24



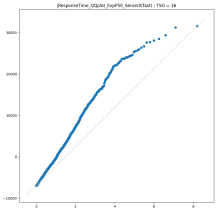
(b) P 0.5, TSO 27



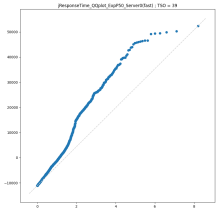
(c) P 0.5, TSO 30



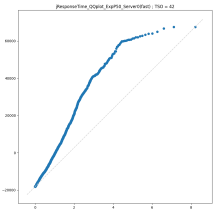
(d) P 0.5, TSO 33



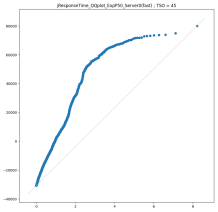
(e) P 0.5, TSO 36



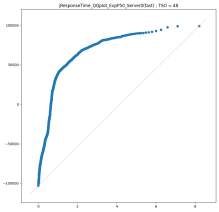
(f) P 0.5, TSO 39



(g) P 0.5, TSO 42



(h) P 0.5, TSO 45



(i) P 0.5, TSO 48

Figure 15: QQplots against Exponential Distribution of ResponseTimes of Server_0 with Probability 0.5

The QQplots above (figure 15) are relative to Server_0; the ones for Server_1 are extremely similar, so i will not report them, in order to avoid extreme redondance.

Observing the QQplots, we can find confirmation that the responseTime is indeed exponential while the system is not overloaded, i.e. for TSO values below 36.

Already at 36 we can observe that the system is failing to keep up with the generator.

These results are in line with all previous considerations and not particoularly surprising.

6 Combinations

In continuation with the observations of the previous sections, I will present a few plots obtained using either Exponential Generation Times and Constant Service Times, or the opposite.

6.1 Exponential TN, Constant TSO

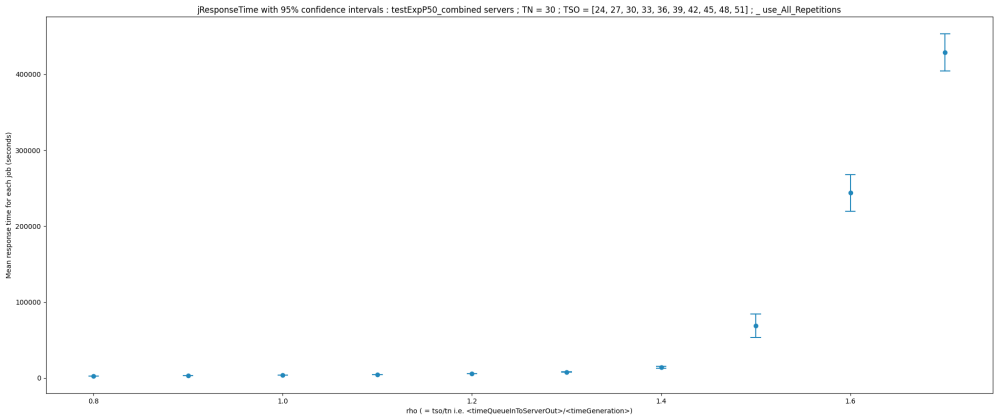


Figure 16: j50exTNcTSO51MergedCI

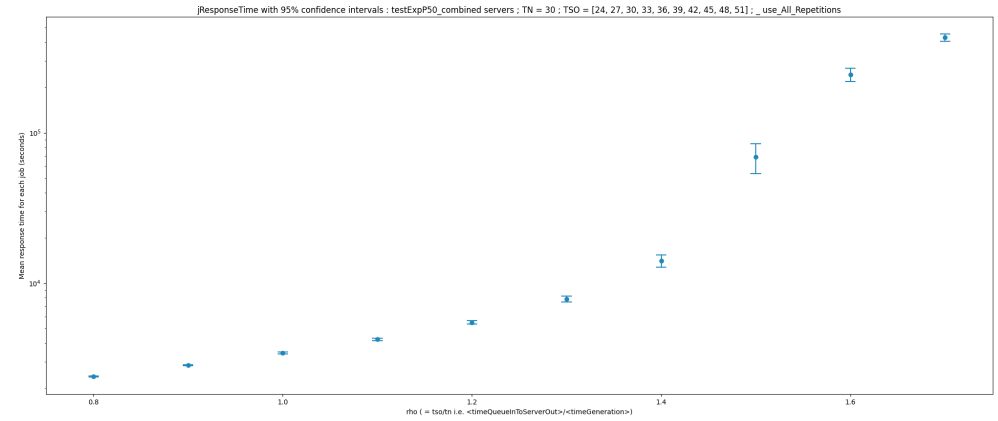


Figure 17: j50exTNcTSO51MergedCIlog

The CI has specific mean values of :

- ratio : 0.8 i.e. ST==24min==1440s; mean : 2395.42099799909 s
- ratio : 0.9 i.e. ST==27min==1620s; mean : 2854.1530597406836 s
- ratio : 1.0 i.e. ST==30min==1800s; mean : 3436.89947399731 s
- ratio : 1.1 i.e. ST==33min==1980s; mean : 4232.426919201562 s
- ratio : 1.2 i.e. ST==36min==2160s; mean : 5481.360835487705 s
- ratio : 1.3 i.e. ST==39min==2340s; mean : 7834.533118385726 s
- ratio : 1.4 i.e. ST==42min==2520s; mean : 14119.140287680932 s
- ratio : 1.5 i.e. ST==45min==2700s; mean : 68830.77935249131 s
- ratio : 1.6 i.e. ST==48min==2880s; mean : 243768.75445323435 s
- ratio : 1.7 i.e. ST==51min==3060s; mean : 428849.197702296 s

In this test, it can yet again be observed a continuous increase in the response time, which is accentuated as the ratio grows closer to 1.2.

Compared to the results observed in section 3.5.4, we can see that this combination has overall definitely lower ResponseTimes when the ratio is below 1.5 , and extremely greater delays as the ratio surpasses 1.5 .

A plausible expanation is the following :

- when the ratio is strictly below 1.5, the system is on average stabile, and the only accumulation in queue (which implies increase in RT) is due to fast generation (since the servers have constant elaboration time). This means that, on average, the impact of "single fast bursts" is alleviated by the presence of two servers! A single-unit-burst does not impact the queue, and the "other server" can go back to standard execution. This allows the queue to, in average, not increase sensibly (even after longer tests)
- when the ration is 1.5 or above, that situation is already of saturation for a fully constant system, so the exponential growth is expectable. One possible reason why Exp-Con here behaves worse than Exp-Exp is that it lacks events of quick processing which would allow to quickly reduce accumulated queue. Being constant and slower than the generator is a huge malus.

6.2 Constant TN, Exponential TSO

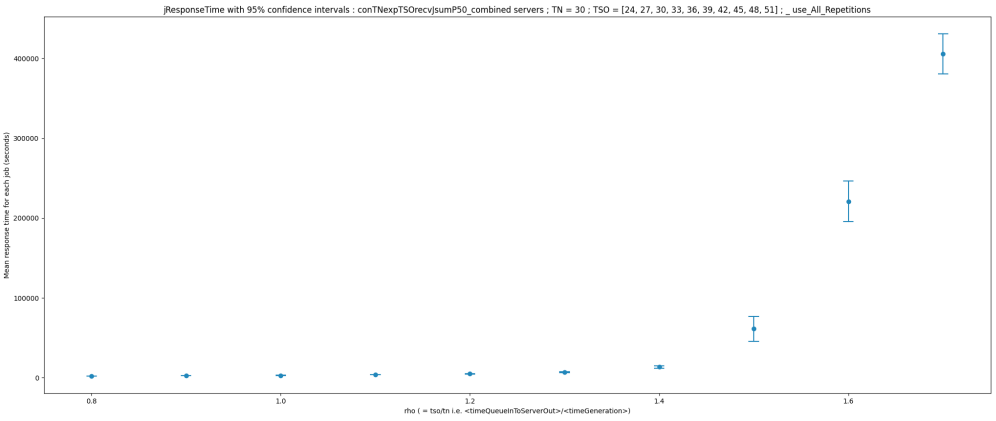


Figure 18: j50cTNezTSO51MergedCI

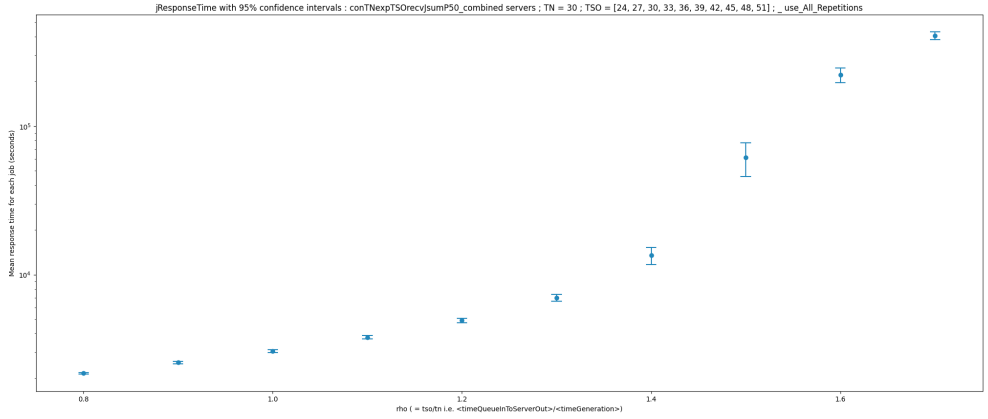


Figure 19: j50cTNezTSO51MergedCIlog

The CI has specific mean values of :

- ratio : 0.8 i.e. ST==24min==1440s; mean : 2168.140298755936 s
- ratio : 0.9 i.e. ST==27min==1620s; mean : 2560.1098315277845 s
- ratio : 1.0 i.e. ST==30min==1800s; mean : 3055.770958940562 s
- ratio : 1.1 i.e. ST==33min==1980s; mean : 3790.273757786835 s
- ratio : 1.2 i.e. ST==36min==2160s; mean : 4910.734929988562 s
- ratio : 1.3 i.e. ST==39min==2340s; mean : 6998.216991337422 s
- ratio : 1.4 i.e. ST==42min==2520s; mean : 13476.440434908645 s
- ratio : 1.5 i.e. ST==45min==2700s; mean : 61419.877588030955 s
- ratio : 1.6 i.e. ST==48min==2880s; mean : 220994.1893029235 s
- ratio : 1.7 i.e. ST==51min==3060s; mean : 405790.7470513826 s

This combinations appears to be better (lower average Response Times) than the previous (6.1) at all values of ratio.

Compared to the combination with both exponential distributions mentioned at 3.5.4, this combination performs better for all ratios below or equal to 1.5, and worse at higher values.

This result is possibly determined by the fact that having a constant input throughput implies that "the fault" for queue-lengthening is only based to specific server-throughput performance. Considering that we have two servers, both with exponential distributed serviceTimes, it is possible for one to "mend" the delay caused by the other somewhat.

This would suggest that a system with a single constant time generator and a number of exponential distributed ST servers is more stable the higher the number of said servers (this is not in relations only to the simply incremented service throughput), due to the ability of the servers to "help" each other better.

7 Conclusions

Summing up the performance observations of all previous sections, it can be observed that:

- the influence of the *probability p* is almost negligible, as discussed in section 4.1, since whenever the servers are not able to complete a job before the arrival of a new one (in general when $t_{so} > t_{tn}$ then both servers will contribute to elaborating the input-workload, and most of the queuer assignment choices will be forced (probability influences rng only when both servers are free, which is rare at higher values of ratio $\frac{t_{so}}{t_n}$; for values of $t_{so} < t_{tn}$, the probability should favour the fast server in order to reduce the mean response time, therefore there is overall no (practical) reason this system should use any probability other than 1, unless if both servers are faster than the generator, in which case using only one server would be sufficient.

- the most simple system to analyse is, as expected, the one with constant times on both generator and servers, and reaches saturation at ratio 1.5 . Before saturation, its average responseTime is the lowest amongst all other time distribution studied, reaching $\frac{13}{6}t_n$, and having the corresponding highest mean number of jobs in system of $\frac{13}{6}$.

- the system with exponential distributions for TN and TSO appears to reach saturation around ratio 1.4, which is reasonable considering that is has inevitably more opportunities to accumulate queue than a Const-Const system.

- amongst the combinations with exponential arrivals, the best performing (while below saturation limit) is the one with constant servers, while amongst the combinations with exponential servers, the best performing (while below saturation limit) is the one with constant interarrivals.

- in general, the system performs better (has lowest mean RN) when the distributions are constant and worsenes otherwise. This behaviour is perfectly normal and well within expectations.