POLITECNICO DI TORINO

Dipartimento di Elettronica e Telecomunicazioni

Corso di Laurea Magistrale
in Communications and Computer Networks Engineering

Tesi di Laurea Magistrale

Simulation of a Base Station with Integrated Services

Relatore

firma del relatore (dei relatori)
prof. Meo Michela
.....
.....

Candidato
firma del candidato
Alberto Benvenga

Marzo 2021

Index

1. Introduction

During last years, a new technology of mobile telephony has been discovered, called 5G, due to the changing key parameters that have to be satisfied in the next years, such as:

- Virtualization of most of web devices (switch, routers, …);
- Increase of the number of mobile devices to be satisfied;
- Reducing latency;
- Reducing energy consumption;
- Increase of transmission speed.

Each of these goals has been achieved employing some specific technology that I'm going to describe here:

SDN FOR VIRTUALIZATION:

Virtualization of web devices has been achieved thanks to the introduction of the SDN (Software Defined Networking), which basically divide network in 2 types of nodes: edge nodes, and central nodes.

Edge nodes are the user ones, and usually they are "dumb", since they just have to do what central nodes tell them to do. This is the main goal achieved by virtualization: moving the complexity from many nodes, to few central intelligent nodes, which have to take decisions and communicate them to the user nodes.

Any time that a dumb node does not know what to do with data, then it asks to controller nodes; then it gets answer, and from now on it knows what to do with that type of data.

Another advantage achieved by this approach, is of course the reduction of costs of web devices; basically the user nodes do no take decisions, they do what controllers tell them to do, so do not need complex hardware architecture.

For what concern controller nodes, usually are few, and completely software; sometimes they exchange information with neighbour controller nodes in order to better manage data of a specific geographic area for example.

EDGE COMPUTING FOR LATENCY:

Reducing latency is a crucial point for some type of applications, and in order to achieve it, edge computing has been implemented in 5G.

Edge computing is a network composed by many data centers able to elaborate and memorize critical data locally, and then send data to a central data center.

In this way, time-sensitive data can be elaborated locally, if we have a smart device, or by an intermediate server located near to us; non-time sensitive data, instead, can be sent to the cloud.

In small words, critical data, the one asking very low latency, are elaborated and memorized by servers near to us, and not by the central cloud, which can be very far from us, reducing significantly the latency.

In addition, edge computing is good also for privacy, because in this way data can be stored locally in intermediate servers, and not all in the central cloud, like happens in cloud computing.

NETWORK DENSIFICATION AND SMALL CELLS:

One of the main reason to switch from 4G to 5G is the continuous increase of mobile devices asking connection to the Internet. This big problem, and the need of increase the transmission speed, lead us to network densification and small cells.

This because having a major number of adjacent cells covering the same area means having higher total capacity, which is necessary as the demanding bit rates per user are increasing as time pass by.

In addition, increasing the base station sites means that we can handle higher traffic per square meters, because there will be many 5G base stations covering geographical area, and this means that the traffic is not directed all to the same cell, like it happens in 4G, where the coverage of the area is bigger, but it is directed in different cells; this helps in reducing the load of each base station, and as a consequence it allows to deal more traffic in the same geographical area.

Also, the fact that these cells are smaller means that, inside a 5G cell, the distance between the base station and the user equipment is smaller than the one of 4G, and it leads to higher per-user bit rate.

Each cell works on a specific range of frequencies: the higher the frequencies are, the higher the transmission speeds, but lower coverage. This turns into 3 possible type of cells:

- Small band cells: in these cells, working frequencies are around 700/800 MHz, and the achievable data rate is pretty low, 30/250 Mbps. On the contrary, their coverage area is very high, then can be implemented in places where we do not expect so much traffic.
- Medium band cells: here the working frequencies are of a few GHz (2/3), and the achievable data rate is 100/900 Mbps. These cells will be very common in metropolitan scenarios, and has been already implemented in some cities.
- Large band cells: these cells are the ones which really makes possible the network densification. Working at frequencies around 25/39 GHz, they can achieve very high bit rate, of the order of 1Gbps. They use mmWaves, which can be easily blocked by walls, meaning that the coverage area is very small, and that's why they are also called SMALL CELLS. The idea is to put these type of cells in places where we expect a lot of traffic, and cover areas by have many adjacent SMALL CELLS.

Unfortunately, it's not so easy to switch all the 4G technologies to the new 5G ones, it is much easier to mix both technologies before changing definitely to 5G.

The best way to do so, is to put a 5G small cell in an area covered already by a 4G macro cell, producing a heterogeneous network (HetNet). In particular, it is convenient to place the small cell where it is expected to have high traffic of users, called Hotspot.

The customers entering the macro cell, then, will face four possible scenarios:

1. They will finish their service in the macro cell;
2. They will move to the small cell, performing then a handover before finishing the service;
3. They will move to another macro cell, performing handover again.

Unfortunately, planning a HetNet is very complex, especially because we have to answer to 2 questions:

1) Is the presence of a small cell effective in the same way on different kinds of traffic (elastic and inelastic traffic)?

2) How effective is the presence of a small cell in a macro cell?

Once these questions have been answered, then it will be possible to dimension and position small cells in such a way that densification brings the desired benefits.

This thesis will be focused on the first question, but in order to achieve reliable results, it is necessary to create an accurate simulation model.

The thesis, then will be structured in the following chapters:

- Queueing model for cells with mixed traffic: in this chapter the analytic queueing model which describes the situation to study has to be created;
- Simulation model: once the analytic model is ready, then it has to be implemented in a simulation environment, which will be Omnet++. Omnet++ is an extensible, modular, component-based *C++* simulation library and framework, primarily for building network simulators. Its functionalities will be described in a clear way, such that the generated code will be easily understandable.
- Results for one cell: here it is reported how the small cell acts in presence of both inelastic and elastic traffic. Here all the statistics computed by the simulation will be plotted, and the results will be studied and analysed in order to get proper conclusions;
- Appendix: in here is present the code generated to simulate the different scenarios, and it is explained step by step.
- Conclusions.

2. Queueing model for cells with mixed traffic

In this scenario we are interested in the behaviour of a base station loaded with two classes of services: inelastic and elastic.

Our base station has a finite capacity that has to be shared between these 2 types of services. Inelastic services require a fix bit rate, a continuous data flow (like video conversation or video streaming); the service data rate is fixed by the rate at which data are produced at the source, and cannot be increased or decreased by the network.

Instead, elastic ones are transmitted at the maximum bit rate possible, but only if this one is greater than the minimum rate they ask. In this case, the data is available at the source, and the data rate is constrained by the network capabilities, as well as the source and destination capabilities of transmitting/receiving data, and the characteristics of the network protocols.

In conclusion, an elastic service requires a minimum data rate, and, on top of that, equally share all the capacity not used by inelastic services with the other elastic ones.

From now on, the base station will be simulated by using a queue, where packets will be stored until they end up their service or perform handover.

Each time a service enters our queue, it has 2 possibilities; either it completes its task, or it performs handover; in this model, handovers represents the mobility of users, because after a period of time we assume that the user moves into another cell, so outside the coverage of our base station. If a service performs handover, it has to start again from the beginning the service in the new cell.

Since the arrivals can be either elastics or inelastics, it is convenient to describe a queue model with 2 possible streams of customer arrivals: inelastic customers with rate λ_i, and elastic ones with rate λ_e. We initially will assume that the generation processes are Poisson ones, so this implies we got λ_i inelastic arrives per second, and λ_e elastic arrives per second on average.

It is important to notice that not each service that arrives at the queue enters it; it depends if the amount of available capacity is enough to serve it.

In particular, if we assume that the queue has a total capacity C, and R_i is the bit rate required by each inelastic customer, and R_e is the minimum bit rate required by each elastic service, then a service will be accepted if and only if:

- If customer is inelastic, it will be only accepted if:

  $$C - N_i R_i - N_e R_e > R_i \qquad (2.1)$$

  Where N_i and N_e represent the number of inelastic and elastic customers in service respectively.

  This equation can be summarized as follows: if we subtract from the total capacity C, the bit rate occupied by already in service inelastic customers ($N_i R_i$), and the bit rate that would be occupied by the elastic services if each one was working at its minimum

accepted bit rate R_e ($N_e R_e$), we obtain the residual available capacity. If this one is greater than R_i, then this customer can be served, otherwise it is discarded.

Notice that it is not mandatory that elastic services work at R_e, this rate just need to understand if in the worst situation for the elastic, the inelastic one can be served or not; if yes, then elastic customers equally share the capacity that is not used by inelastic ones, and this value can be larger than R_e.

- If customer is elastic, it will be only accepted if:

  $C - N_i R_i - N_e R_e > R_e$ (2.2)

  The explanation is exactly the same of the case above.

We will further assume that an inelastic service size is exponentially distributed with mean $\frac{1}{\phi_i}$ bits, and a service time exponentially distributed (at least at the beginning) with mean $\frac{1}{\mu_i}$ seconds; in this way, the relation between these two quantities is the following:

$$\frac{1}{\phi_i} = \frac{R_i}{\mu_i} \qquad (2.3)$$

For what concern elastic customers, things are a little bit different. Their packet sizes are exponentially distributed with mean $\frac{1}{\phi_e}$ bits, but this time, we cannot say in advance how its service time is distributed; this one depends on the number of inelastic and elastic customers actually in service, and so it's a value continuously changing.

This value, anyway, will be included among 2 values:

- $\frac{1}{C\phi_e}$, this value occurs in the best case, when the elastic service uses all the capacity C, meaning that it's the only service in the queue, leading to the minimum service time;
- $\frac{1}{R_e\phi_e}$, on the contrary this value represents the case in which the elastic customer has to work at its minimum accepted rate R_e, leading to the maximum service time for elastic customers.

Up to now, I described how are distributed packet sizes, service and generation times; but, as I said before, a customer has 2 options when enters the queue: either it completes its service with rates just computed, or it performs handover.

Handover is just represented with a random variable exponentially distributed with mean $\frac{1}{\mu_H}$ seconds; it easily means that each service remains on average $\frac{1}{\mu_H}$ seconds in our cell; if the service time is less than this value, then the customer completes its task, otherwise it leaves the cell (and the queue) without completing.

At each point, the state of the queue can be described by:

$S = (N_i, N_e)$ (2.4)

Since both handover and service time are exponentially distributed, for now, the minimum between these two random variables has a rate which is the sum of the 2 rates, and so the individual service rate of an inelastic customer is:

$$\mu_i(N_i, N_e) = \mu_H + \mu_i \qquad (2.5)$$

While for an elastic customer it becomes:

$$\mu_e(N_i, N_e) = \mu_H + \frac{(C - N_i R_i)}{N_e}\phi_e \qquad (2.6)$$

This because, as explained by the formula 2.3, the rate of service time should be the product between the bit rate associated to each elastic service, and the inverse of the mean of the elastic packet size (ϕ_e). Since the bit rate of elastic customer changes as the number of customers in service, then it has to be computed on the base of how many inelastic and elastic services are present in the queue in that moment.

In order to compute the elastic data rate, it is necessary to equally share the capacity not used by inelastic customers ($C - N_i R_i$) among all elastic customers in service (N_e).

So, if we want to compute the total service rate, in a specific queue state, of inelastic customers is:

$$\mu_{Ti}(N_i, N_e) = (\mu_H + \mu_i)N_i \qquad (2.7)$$

While, the total service rate for elastic ones is:

$$\mu_{Te}(N_i, N_e) = \mu_H N_e + (C - N_i R_i)\phi_e \qquad (2.8)$$

These two formulas are obtained, respectively, simply multiplying the equations 2.5 and 2.6 by N_i and N_e.

In addition, there cannot be more than ν_i inelastic customers in the queue, and more than ν_e elastics.

The analysis of the queueing model so can be performed either by a simulation model, which is what I did and will be shown after, or by a CTMC in the case of exponential distributions (which is our initial case).

In this way, the results given by the simulator can be checked studying the Markov-Chain associated.

3. Simulation model

The simulation model of the scenario described above has been designed using the simulator OMNET++. This type of simulator creates different modules, and each module can work standalone like a usual c++ code, or it can communicate and exchange information to the modules to which it is connected to.

The structure of modules of the simulation is represented as:

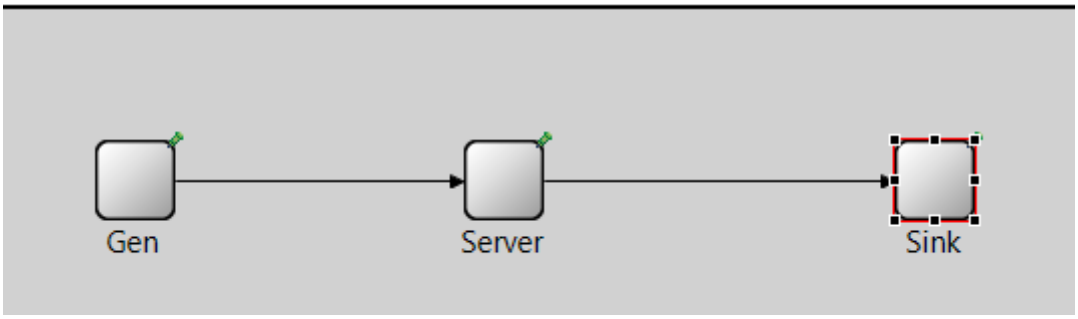

Figure 3.1: Topology of the modules composing the network

So, we got 3 types of modules, each simulating a specific part of our scenario: generator, server and sink.

Each one of these modules is described by 3 types of files created by the simulator:

- **NED FILE:** this file is used to create the physical topology connecting different modules; here we specify the connections among modules, the speed at which they exchange information, delay, the gates that are involved in the connections, and also parameters associated to them.
- **INI FILE:** this file is where all the initial parameters with their values are collected; each parameter is associated to a module.

  For example, if in the NED file we declared that the generator has a parameter called inel_prob, then in the INI file, in order to assign this parameter a value, we have to do as following:

  Generator.inel_prob=VALUE WE DECIDE

  This allows us to run different scenarios simply changing the values in the INI file, without need to change from scratch our code.
- **C++ FILE:** this file is the hearth of the modules; here is where we decide exactly what each module has to do when receives information.

  Differently from the INI and NED files, which are common to all the modules, the C++ files are specific for them; so each one has its own C++ file.

  In addition, each of these C++ files is divided in 3 main parts of code:

  - **Initialize part:** this part is the one in which we assign values to the variables of our code; it runs just once, as soon as the simulation starts.

- **HandleMessage part:** this one is the core of the module, because here we have all the procedures that we want our module to run. This part runs each time the module receives a message in input, which can be a self-message (if the module sent a message to itself, explained after), or a message coming from another module.
- **Finish part:** this part, obviously, it is entered only at the end of our simulation. In particular, OMNET++ allows us to decide for how many simulated seconds we want our simulation to run (this value is specified in the INI file with the name sim-time-limit=VALUES IN SECONDS).

  We will change often this value, in order to understand how the scenario works for different time intervals, or to have more accurate statistics.

  Well, once the simulation reaches the end of the seconds, the code of each module enters the finish part, and usually here we want display at screen the statistics computed on the run.

Now let's describe exactly what each module does:

- **GENERATOR:** it simulates the arrival of new customers in our cell. It uses a specific technique of this simulator called self-message, which basically is a message that the module sends to itself after a certain amount of time.

  Once it receives this message, then it triggers an action, which in our case is the generation of a packet to send to our output module, which is the server.

  The time between 2 self-messages is the interarrival time between 2 customer arrivals; so, if we decided to distribute customer arrivals through exponential random variables, then the self-message must be received, on average, after $\frac{1}{\lambda}$ seconds where λ is the rate of customer arrivals.

  So in the initialize part of the generator, we send a self-message, and as soon as it is received, we enter its handleMessage part.

  Here, in our specific case, we have to decide if the arrival is an elastic or inelastic one; once we decided, we sent this packet to the server module, and we send another self-message, so the cycle continues.

- **SERVER:** it basically simulates the behaviour of the base station. So as soon as it receives a message from the generator, it will take decisions on how and if this customer has to be served. In this module we also collect useful statistics to analyse the base station behaviour.

  If the customer finishes its service without performing handover, then we send this one to the sink; otherwise, if it performs handover, then it will be deleted after its dwell time expires.

- **SINK:** this one simply confirms us that the customer finished its service and we don't need any more to serve it.

Now, that we know which are the modules, what they simulate, and how are they connected among themselves, it can be explained how the simulation works.

Later on, it will be shown in the code the presence of 2 matrices in the C++ code of the server (simulating the base station); in particular, one matrix for elastic services and the other one for the inelastic ones.

The number of rows of the matrices is the maximum number of inelastic/elastic customers simultaneously present in the queue allowed, so respectively ν_i and ν_e.

In this way, each row is linked to a specific customer, and the cells contain parameters referring to it, in particular:

- First columns contain how many bits are remained to transmit for that customer;
- Second columns contain the remaining dwell time of that customer, that is how much time remains before that customer performs handover;
- Third columns contain the time needed to transmit the residual bit of that customer;
- Fourth columns contain, just for elastic ones, the bit rate associated to that customer, which will be the same for all elastic ones, since they equally share the residual capacity. This value is necessary for them since it's varying each time each customer enters/leaves the queue; instead, since the bit rate associated to an inelastic service is always the same, this value doesn't need to be specified for them.

How these values are exactly computed is explained below when the code is studied; the importance of these values is now explained.

Since each customer can both perform handover or complete the service, we cannot know in advance which of these 2 situations will be chosen for each customer; so at each change of state, we collect in the matrices these times for each service with the updated values.

The values of times keep decreasing, until one arrives to 0; when this happens it means that:

- If the 0 is in the second column: a handover has to be performed for that customer;
- If the 0 is in the third column: that customer has completed its services.

So, at each time, we have to compute the minimum among all remaining dwell and completion times, in order to know which will be the first time to arrive to 0; once we know which is the minimum time, a self-message will be scheduled after exactly that time. In this way as soon as we receive the self-message, we know that a handover or a completion has to be performed.

In both cases, the row linked to that customer is reset to -1, so from now on that row can be associated to a new customer.

Notice that if a new arrival happens before the scheduled departure message expires, then the scheduled message has to be deleted, and sent again once the new arrival has been assigned the reserved capacity; this is necessary because, as soon as a customer arrives, the residual capacity changes, and so also the bit rate associated to each elastic customer, and as a consequence also their completion times. So, as soon as a new arrival happens, we check if there is a scheduled message, if yes is deleted, and then we update whole residual times and bit to transmit and associated bit rate for each customer.

This procedure will go on until the simulation time ends.

4. Results for one cell

At the begin, I started in simulating just one cell, with mixed traffic, in different scenarios, in order to understand how things change if parameters change.

Different cases have been studied, and they are reported below.

4.1 First studied case: low arrival rate

The first studied case has the following parameters:

Table 4.1. Parameters of the first tested case

Parameters	Value	Unit of measure
Max # inelastics	1	Pure number
Max # elastics	10	Pure number
Interarrival rate (λ)	20	1/seconds
Probability a service is inelastic	0.5	Pure number
Mean of dwell time	1	Seconds
Capacity of the queue	100	Kbit/s
Rate asked by inelastics	100	Kbit/s
Minimum rate asked by elastics	0	Kbit/s
Mean of packet size of inelastic services	100	Bits
Mean of packet size of elastic services	100	Bits

The first important aspect to notice is that in this particular case, the minimum rate needed by elastic services is 0Kb/s; this means that when an inelastic customer arrives, and there are no other inelastic in service, then it will take the whole capacity (100Kb/s). Instead, all the elastic customers in service, and the ones that will enter the queue (accepted up to a maximum of 10 elastics), will temporarily get 0Kb/s, increasing their completion time to infinity.

Obviously, as soon as the inelastic service ends, then all the elastic ones will change their rate in such a way they equally share the capacity; this implies that, in this specific tested case, the elastic customers are discarded if and only if there are already 10 of them in service.

Another important observation that can be done, is that, given this specific choice of parameters, the behaviour of inelastic services is perfectly independent from the one of the elastics; this because their behaviour is not affected by how many elastic customers we have in service.

This is because, again, if we have no inelastic customers in service, as soon as one of them arrives, it doesn't matter how many elastics are present in the queue, because they can always adjust their rate to 0, given then priority to the inelastic customer.

Thanks to this observation, the behaviour of inelastics can be predicted by taking into account the M/M/m/0 case, where m in this case is 1 since there can be at maximum 1 inelastic in service. In particular:

$$\sum_{N_e=0}^{N_e=10} P(0, N_e) = \pi_0 \qquad (4.1)$$

Where π_0 is the probability of being in the 0 state of the M/M/1/0 queue. The formula to compute a state probability, for that queue is:

$$\pi_k = \frac{\frac{\rho^k}{k!}}{\sum_{i=0}^{i=m} \frac{\rho^i}{i!}} \qquad (4.2)$$

Where ρ simply is the ratio between the arrival and service rate of inelastic customers.

So, the formula 4.2, if we are interested in the 0 state probability, change as follows:

$$\pi_0 = \frac{1}{\sum_{i=0}^{i=1} \frac{\rho^i}{i!}} = \frac{1}{1+\rho} = \frac{1}{1+\frac{\lambda}{\mu}} = \frac{\mu}{\mu+\lambda} \qquad (4.3)$$

In the formula 4.3 particular attention should be focused on the parameters μ and λ, because as already mentioned before, this queue is simply modelling the inelastic behaviour, and so it means that we are referring just to them; that's why both μ and λ are not the general ones, but are the inelastic ones.

In particular:

$$\lambda_i = \lambda \times prob_{inel} = 20[1/s] \times 0.5 = 10[1/s] \qquad (4.4)$$

As can be seen by formula 4.4, the inelastic arrival rate is computed by multiplying the overall arrival rate and the probability that the customer is inelastic. The same could be done to compute the elastic arrival rate, and that means, basically, that our arrival process can be seen as two arrival processes (one for inelastics and one for elastics) working in such a way that their arrival rates, summed up, give the overall one.

For what concerns, instead the inelastic service rate, then:

$$\mu_{is} = \mu_H + \mu_i \qquad (4.5)$$

By reading formula 4.5, we understand that the rate of inelastic service is the sum between the dwell rate and the service one, and this is because the choice performed by the inelastic packet is the one with minimum time, and so its rate is the sum of the two because they are both exponentially distributed.

Dwell rate μ_H, since the dwell time is exponentially distributed, is simply the inverse of the mean of the dwell time, and reading table 4.1 we get that it is 1 second; so it turns that

$$\mu_H = \frac{1}{1[s]} = 1[1/s] \qquad (4.6)$$

Instead, the service rate μ_i is not specified by parameters, but thanks to the formula 2.3 we get :

$$\mu_i = R_i \times \phi_i = \frac{10^5[bit/s]}{100[bit]} = 1000[1/s] \qquad (4.7)$$

In formula 4.7 both R_i and ϕ_i value can be read from table 4.1, but R_i has been converted from Kbit/s to bit/s.

So now, putting together the results of 4.6 and 4.7, the formula 4.5 gives the following result:

$$\mu_{is} = (1+1000)[1/s] = 1001[1/s] \qquad (4.8)$$

So, finally, we can compute π_0 by means of the formula 4.3, using the values given by 4.4 and 4.8:

$$\pi_0 = \frac{\mu_{is}}{\mu_{is}+\lambda_i} = \frac{1001[1/s]}{(1001+10)[1/s]} = \frac{1001[1/s]}{1011[1/s]} = 0.99 \qquad (4.9)$$

As a consequence, by formula 4.1, we get that the sum of all the state probabilities with 0 inelastics is 0.99, meaning that for 99% of the time there will be no inelastic customer in the queue. This result may seem wrong, but it is due to the fact that the state (0,0) has a probability equal to 0.98, while all the other state probabilities are extremely low.

This is because the service rate is much higher than the arrival one, and so as soon as a service enters the queue, it will be immediately served, and a long time pass before a new arrival happens. Since very small probabilities, of the order of 10^{-8}, cannot be seen by the simulator, it will turns out as if that probabilities are 0, turning in low accuracy results; this is the main reason for which we proceeded in the next case.

Anyway, some key statistics have been computed, and it is shown in this table:

Table 4.2. Results of the first tested case

Statistic	Value	Unit of measure
Inelastic losses	24766	Pure number
Elastic losses	0	Pure number
Inelastic handovers	2462	Pure number
Elastic handovers	2652	Pure number
Inelastic completed services	2472893	Pure number
Elastic completed services	2497635	Pure number
Average inelastic customers in service	0.00987818	Pure number
Average elastic customers in service	0.010294	Pure number
Average inelastic fraction of capacity (%)	0.987818	Pure number

Average elastic fraction of capacity (%)	0.999165	Pure number
Average queue load	1986.98	Bit/s
Average elastic bit rate	994.08	Bit/s

The following results make sense, and let's analyse them one by one; first of all, we can notice we do not have any elastic customer lost. This is perfectly reasonable because the rate of arrival is 20, meaning 20 arrivals per second on average, in which 10 of them are elastic; but the completion time (service time), is much lower, because is computed as the ratio between the packet size of an elastic customer, and its associated rate, which will be almost always equal to whole the capacity, due to high service rate with respect to arrival one.

As for inelastic, under the assumption that the elastic customer is the only one in service, it will get whole the capacity, and so it means, as explained in formula 4.7, that its service rate will be 1000; this value, compared to the value of λ, turns out to be much bigger.

This leads to a scenario in which often as soon as a customer enters the queue, it will immediately be served even before some other one can enter the queue, explaining the high probability of state (0,0).

In addition, the minimum rate of elastics here is 0Kb/s, so even if there is an inelastic customer in service when an elastic one arrives, this last one will not be lost, but it will be accepted in the queue and assigned its minimum rate 0 Kb/s. So, the only way to have elastic losses is to reach Max #el customers in service, but this scenario will never happen because service rate is much higher than interarrivale one, as explained before.

On the contrary, inelastic losses can happen because it is exceeded the max # inelastic customers allowed, which is 2; so, in order to have a loss, it is sufficient that an inelastic arrives when there is already another inelastic in service, which is not so frequent, but it happens. That explains why the losses are so small with respect to the completed services.

In addition, also the handovers turns out to be not so frequent with respect to the completed services; this is because the dwell time is exponentially distributed with a mean of 1 sec, while, as said before, the service time, when we are the only one in the queue (most of the time), is 10^{-3} seconds. So the service time, on average, its 1000 times smaller than the average dwell time, and so the completed events will be of the order of 1000 times bigger than the handovers, as demonstrated. It is also possible to sum up handovers, losses and completions of both inelastics and elastics, in order to check that these 2 values are almost equal, respecting the 0.5 probability of having inelastic services.

For average customers in service, both inelastic and elastic, we get values around 0.01, meaning that, on average, there is 0.02 customers per second, or better, there is 1 customer every 50 seconds. This is perfectly reasonable, because λ is 20 and μ is 1000 (if the customer is the only one in the queue), so the arrival rate is exactly 50 times smaller than the service rate. As a consequence, it can be notice also that the average queue load is respecting this criterion; in fact, its value (1986) is 50 times smaller than the overall capacity (100Kb/s), that's because 1 time over 50 there is a customer taking all the capacity, so there is a capacity usage of 100Kb/s over an interval of 50 seconds, with available capacity which is 50 times larger.

In addition, when there is just a customer in the queue, it has 50% possibility of being elastic, and so it makes perfectly sense that the average elastic rate is half the average queue load; it

simply follows the same criterion of queue load, but it has to be halved in order to consider just the elastic ones.

In the end, for what concerns the average fraction of capacity, if we assume to be most of the time with just 1 customer in the queue, again we can assume it will get all the capacity; unfortunately, as already mentioned before, it happens 1 time every 100 for both inelastics and elastics, so it follows that they use just $\frac{1}{100}$ of the total capacity, so around 1%.

So now, in order to have more accurate results, we move to the second case.

4.2 Second tested case: increasing arrival rate

This case has exactly the same parameters of table 4.1, except for λ, which now is 1000 instead of 20.

In this way, the probability of the state (0,0) is expected to be significantly reduced, while the other states are expected to be more probable, in such a way the simulator is able to give reliable state probabilities.

Formulas 4.1, 4.2 and 4.3 are still valid for this case, but now λ has changed, and as a consequence also the inelastic arrival rate changed, in particular it becomes:

$$\lambda_i = \lambda \times prob_{inel} = 1000[1/s] \times 0.5 = 500[1/s] \qquad (4.10)$$

The service rate, instead, does not change, so it does not need to be computed again.

By using the value given by the formula 4.10, and putting it in the formula 4.9, we obtain a different π_0, which is:

$$\pi_0 = \frac{\mu_{is}}{\mu_{is} + \lambda_i} = \frac{1001[1/s]}{(1001+500)[1/s]} = \frac{1001[1/s]}{1501[1/s]} = 0.666 \qquad (4.11)$$

By looking at formula 4.11, we can conclude that, in this case, there will be no inelastic customers in service for 66% of the time.

This is the theoretical result; let's see if the simulator gives the same values:

Table 4.3. State probabilities of the second simulated case

State probabilities for $\lambda_i = \lambda_e = 500$	**Analytic results**	**Simulation results**	**Error (%)**
P(0,0)	1,811533E-01	1,807240E-01	-0,24
P(0,1)	1,205879E-01	1,204550E-01	-0,11
P(0,2)	9,016861E-02	9,019070E-02	0,02
P(0,3)	6,982462E-02	6,993070E-02	0,15
P(0,4)	5,455086E-02	5,467800E-02	0,23
P(0,5)	4,267993E-02	4,277860E-02	0,23

P(0,6)	3,337343E-02	3,336530E-02	-0,02
P(0,7)	2,606739E-02	2,612890E-02	0,24
P(0,8)	2,033531E-02	2,035790E-02	0,11
P(0,9)	1,584310E-02	1,581840E-02	-0,16
P(0,10)	1,230431E-02	1,230990E-02	0,05
P(1,0)	6,038437E-02	6,022180E-02	-0,27
P(1,1)	6,031053E-02	6,023220E-02	-0,13
P(1,2)	5,013937E-02	5,017010E-02	0,06
P(1,3)	3,996537E-02	4,002950E-02	0,16
P(1,4)	3,148280E-02	3,159540E-02	0,36
P(1,5)	2,469947E-02	2,478350E-02	0,34
P(1,6)	1,933794E-02	1,932920E-02	-0,05
P(1,7)	1,511743E-02	1,514880E-02	0,21
P(1,8)	1,180211E-02	1,185050E-02	0,41
P(1,9)	9,224557E-03	9,212990E-03	-0,13
P(1,10)	1,064731E-02	1,068880E-02	0,39

First of all, an interesting check can be performed summing up the probabilities from P(0,0) to P(0,10); it comes out 0.666, confirming the theoretical prediction.

In addition, I computed the error, which is the difference between the analytic results and the simulation ones; it is nice to see that errors are very small, this is thanks to the choice of the parameter λ.

The choice of this λ equal to 1000 is such that the arrival rate is bigger than the service one; that's why, as already mentioned before, μ is 1000 if and only if we are the only one in service, which is not definitely the case anymore. This means that queue fills up in a frequent way, giving us state probabilities greater than the previous simulated case.

Other statistics computed on this scenario are:

Table 4.4. Results of the second tested case

Statistic	Value	Unit of measure
Inelastic losses	4168896	Pure number
Elastic losses	287458	Pure number
Inelastic handovers	8414	Pure number
Elastic handovers	69680	Pure number
Inelastic completed services	8333827	Pure number
Elastic completed services	12146356	Pure number

Average inelastic customers in service	0.333263	Pure number
Average elastic customers in service	2.79224	Pure number
Average inelastic fraction of capacity (%)	33.3263	Pure number
Average elastic fraction of capacity (%)	48.601	Pure number
Average queue load	81927.3	Bit/s
Average elastic bit rate	22591.2	Bit/s

First of all, as can be noticed, the number of losses is increased with respect to the first case, and now we also have elastic losses; this is obvious, because we increased lambda to 1000, so it means on average 1000 arrivals per second instead of 20, and in addition, now as highlighted in the table 4.3, it can happen we have up to 10 elastic customers in service, leading to an elastic loss if a new one arrives. The reason because inelastic losses are much more than the elastic ones is because there can be just 1 inelastic in the queue; it means that, if there is already an inelastic customer in the queue, as soon as a new one arrives, then it is an inelastic loss. Instead, there can be up to 10 elastic customers in service, and given the fact that elastic and inelastic are equiprobable, it means that is much more frequent to have an inelastic loss instead of an elastic one, and for the same reason the inelastic completed services are much less than elastic ones.

For what concerns handovers, an exponential increase for elastic ones can be noticed; this happens because in this scenario usually we have more than one elastic customer in service, and so their assigned rate will not be anymore 100 Kb/s, but it will decrease as the number of elastic customers in the queue increases. As a consequence, their completion time increases, and the probability that it's bigger than the dwell time increases as well. That's why the handover increase is much more evident for elastic customers than inelastic ones.

Anyway, in order to check the correct functioning of the simulation code, it is possible to sum up handover, losses and completed services both inelastic and elastic, in order to confirm the probability of having one or the other is 0.5.

As expected, rates and fraction and number of customers in service increases in this scenario, and in particular, it can be verified that in this special case, the average elastic fraction can be also theoretically computed doing:

$$(P(0) - P(0,0)) \times 100 = 48.53 \qquad (4.12)$$

This is because we have elastic rate different from 0 if and only if there are no inelastics in service $(P(0) - P(0,0))$, that's because the overall capacity is exactly equal to the rate asked by inelastic customers, so if there is just one of them in queue, it will occupy all the capacity. Given that, the elastic fraction, in these cases, will be always equal to the whole capacity since there are no inelastics in queue, so they can adjust their rate to equally share the whole capacity. In a similar way can be computed theoretically the inelastic fraction:

$$P(1) \times 100 = (1 - P(0)) \times 100 = 33.33 \qquad (4.13)$$

The result given by formula 4.13 is obvious, since we can have just 1 inelastic in service, and will take all the capacity.

An interesting observation can be done summing up elastic and inelastic fraction; the result is not 100%, but:

$$Total\_fraction\_of\_capacity\_used = (48.601 + 33.3263)\% = 81.9273\% \qquad (4.14)$$

By formula 4.14 turns out that just 82% of the capacity has been used, and since it is sufficient that just one inelastic or elastic customer is in the queue in order to maximize the queue load, it means that 18% of the capacity has not been used because there were no customers in queue. This is confirmed checking up the value of P(0,0) in the table 4.3; it is exactly the residual amount of capacity not used by customers (around 18%).

As a consequence, can be easily checked the result of the average queue load; this because the queue load will always be the total capacity, except in the case (0,0).

This can be explained in a simple way: since the rate needed by an inelastic customer is equal to whole the capacity, then it doesn't matter how many inelastics or elastics we have in the queue, the average load will always be the maximum one. Because if we have an inelastic one, it will use all the capacity, instead, if we don't have it, the elastics will adjust their rate in order to use all the bit rate; so anyway there will be no free capacity left.

For this reason, we can claim we have 100 Kb/s average load always, except the case (0,0), so:

$$(1 - P(0,0)) \times 100[Kb/s] = 81.9276[Kb/s] = 81927.6[b/s] \qquad (4.15)$$

Notice that the value obtained by formula 4.15, could also be obtained by the formula 4.14.

The elastic average bit rate, instead, can be computed knowing that in all the states $P(1, N_e)$ it will be 0; so we have to consider only the states $P(0, N_e)$, except $P(0,0)$.

It is sufficient to multiply the probability of one of this state (reported in the table 4.3), by the rate associated to each elastic; so if 1 then 100 Kb/s, if 2 then 50 Kb/s, and so on so forth, and then summed up this values, so basically, written in formula, can be theoretically computed by:

$$\sum_{N_e=1}^{N_e=10} P(0,N_e) \times \frac{100}{N_e}[Kb/s] = 22.591[Kb/s] = 22591[b/s] \qquad (4.16)$$

In the end, the average number of inelastic services can be easily checked by taking into account the fraction of capacity used by them; since their rate will be fixed to 100Kb/s, then it is sufficient to divide the inelastic capacity used, by their rate; it will turn out the value in the table 4.4. Notice that it is exactly the same value of the inelastic fraction of capacity, which make perfectly sense there can be at maximum one inelastic customer in the queue.

On the contrary, the average elastic services in the queue are much more difficult to be checked, because their rate is not fixed, and so cannot be computed by taking into account their fraction of capacity. Anyway, since all the other results are consistent, and make all sense, it is sure that also the average number of elastic customers in service is correct.

4.3 Third tested case: a realistic scenario

in this scenario we started to simulate a more realistic scenario, in which more elastic and inelastic customers can enter the queue, in particular the parameters are:

Table 4.5. Parameters of the third tested case

Parameters	Value	Unit of measure
Max # inelastics	30	Pure number
Max # elastics	30	Pure number
Interarrival rate (λ)	20	1/seconds
Probability a service is inelastic	0.5	Pure number
Mean of dwell time	50	Seconds
Capacity of the queue	10	Mbit/s
Rate asked by inelastics	300	Kbit/s
Minimum rate asked by elastics	50	Kbit/s
Mean of packet size of inelastic services	500	KBits
Mean of packet size of elastic services	500	KBits

The biggest difference with respect to the other 2 scenarios is the rate needed by inelastic/elastic customers; now, an inelastic customer does not require all the capacity, and on the contrary, the minimum rate needed by an elastic one is not anymore 0, so the situation is pretty different.

The state probabilities are not reported in the following, due to the large number, instead, the statistics computed on this case are the following:

Table 4.6. Statistics of the third simulated case

Statistic	Value	Unit of measure
Inelastic losses	2965	Pure number
Elastic losses	60881	Pure number
Inelastic handovers	80554	Pure number
Elastic handovers	58869	Pure number
Inelastic completed services	2417708	Pure number
Elastic completed services	2381104	Pure number
Average inelastic customers	16.1165	Pure number

in service		
Average elastic customers in service	11.8752	Pure number
Average inelastic fraction of capacity (%)	48.3496	Pure number
Average elastic fraction of capacity (%)	47.6093	Pure number
Average queue load	9.59589e+006	Bit/s
Average elastic bit rate	955797	Bit/s

First of all, running the simulation, can be noticed that the probabilities from $P(29,27)$ to $P(29,30)$, and from $P(30,21)$ to $P(30,30)$ are exactly 0, and that is because they will exceed the total capacity of 10 Mb/s.

It can be noticed also, that some other probability will seem to be 0, but in reality it's not; they are showed to be 0 on the simulator due to the very small value they have, and so the program is not able to give a good accuracy to them, as explained before. As the number of inelastic customers increases, the state probabilities are bigger and bigger, before starting to decrease again when the inelastic customers are more or less above 20; that is verified by the results of the simulation, in fact it is showed that the average number of inelastics in service is 16, so the probabilities with inelastic customers around this value will be bigger.

Again, from the results, can be easily checked that the average queue load will be the sum of the inelastic and elastic fractions, which turns to be a value very near to the maximum capacity.

In addition, since the λ tested in this case is the same of the first simulated case, then we expect they have the same number (more or less) of elastic/inelastic generated services, and this is confirmed by checking the values in the table 4.2 and 4.6; if we sum up, in both tables, the handovers, completed services and lost ones, both inelastic and elastic, we obtain, more or less, always 2500000.

The first big difference between these 2 cases is given by the number of elastic losses; now there are many of them, with respect to the first case when they were exactly 0. This is due to the fact that the minimum rate needed by elastics is no more 0Kb/s, but 50Kb/s; this means that now elastic packets can also be discarded because there is no enough capacity to serve them, and this has a big impact on this result. On the contrary, instead, turns out that inelastic losses reduce in this case, and this is obvious given the fact that in this case the queue can guest up to 30 inelastic services, instead of just 1 of the first case.

For what concern, instead, the handovers in this case, are increasing both for inelastic and elastic with respect to the first tested case. It can seem wrong, because now the mean of the dwell time is 50 seconds instead of 1 of the first case, but it has to be considered that the packet size has a strong impact on this statistic, and now is 3 order of magnitude greater than the one of the first case, as can be noticed by tables 4.2 and 4.6. This increase in packet size has a much stronger impact on the handovers with respect to their mean, leading to the increase noticed.

4.4 Fourth tested case: changing packet distributions

in this case, we run a simulation with the same parameter of the previous scenario, except for the dwell time, which now increases to 300 seconds (5 minutes). In addition, we test for 4 different combinations of distributions, in particular changing the duration of the service time, both inelastic and elastic, that means to vary the file distribution of them.

In these situations, different curves have to be drawn, for different values of lambda, for:

- probability of inelastic losses
- probability of elastic losses
- average number of inelastic customers in the queue
- average number of elastic customers in the queue
- average elastic rate

The resulting curves are:

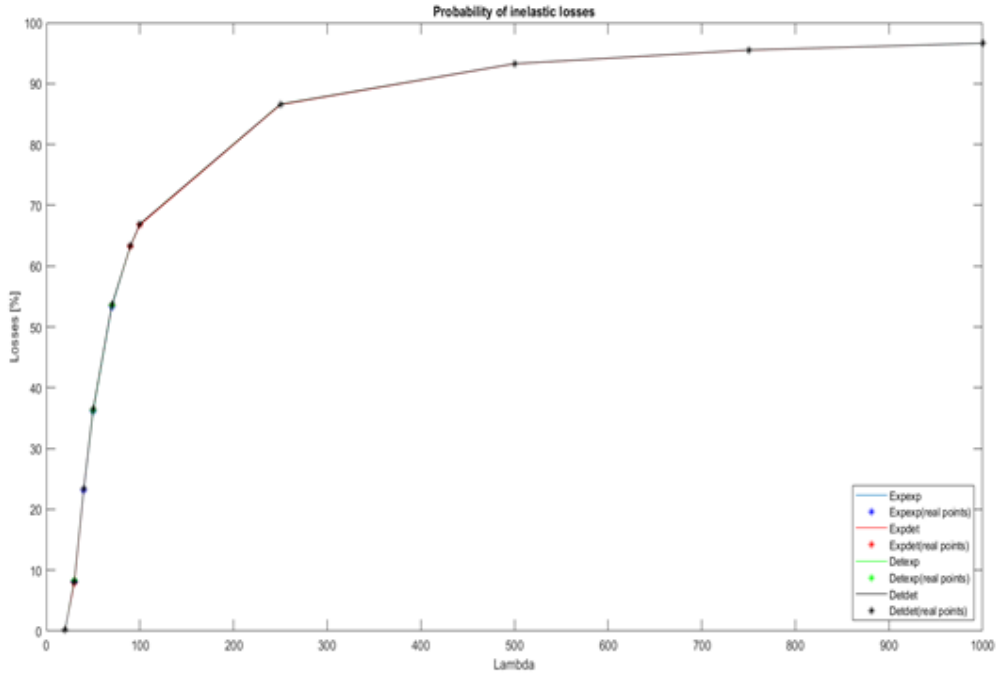

Figure 4.1: Probability of inelastic losses with different types of service distributions

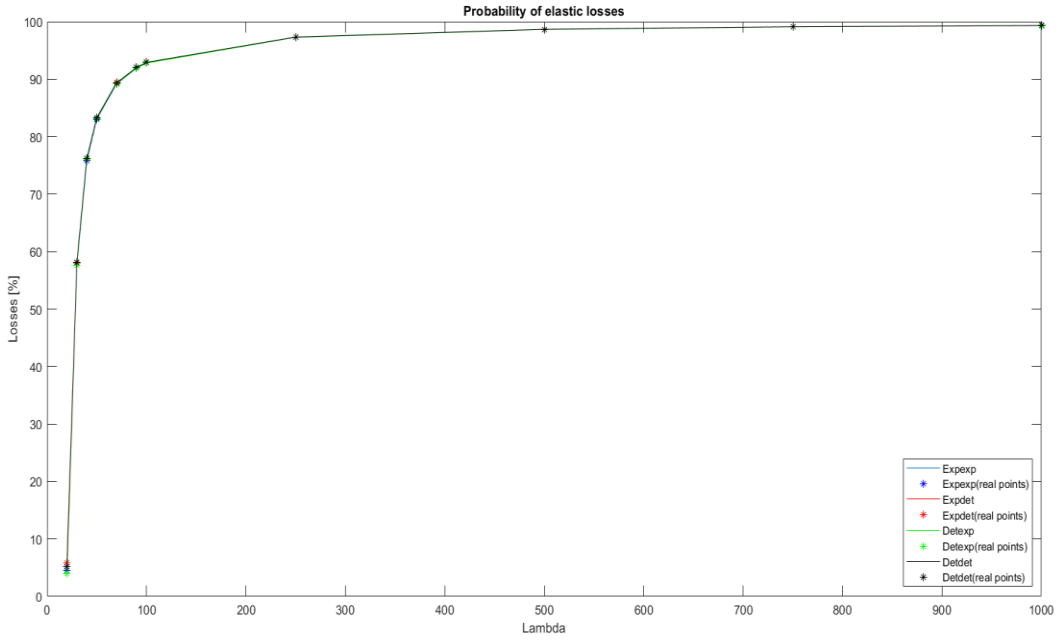

Figure 4.2: Probability of elastic losses with different types of service distributions

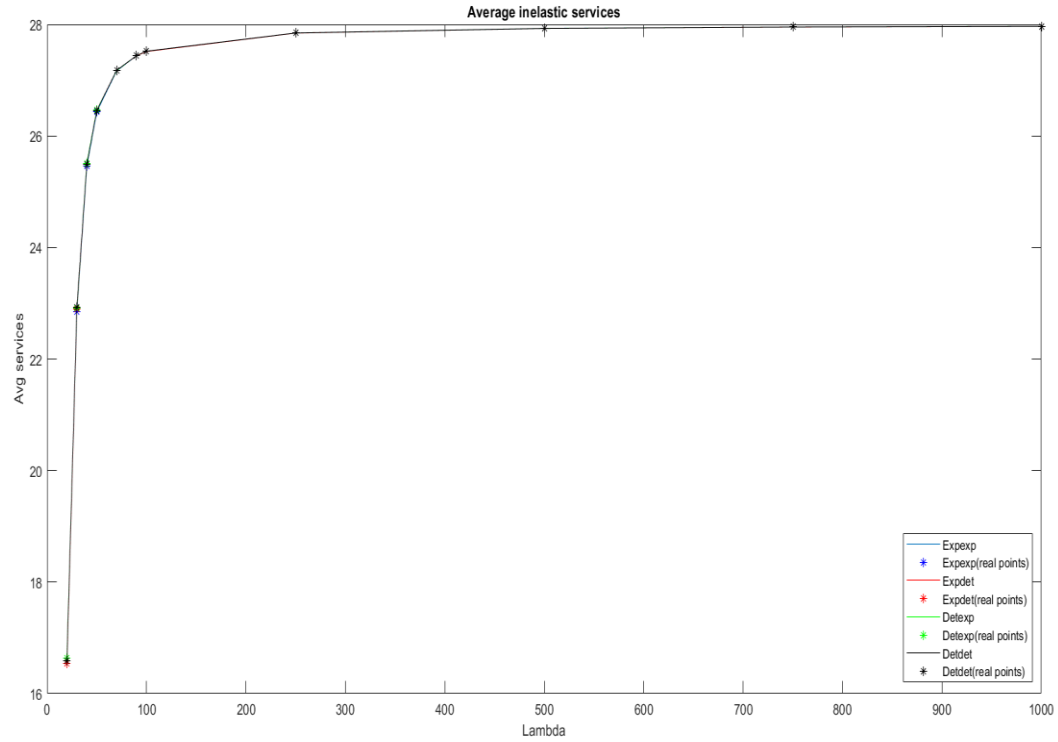

Figure 4.3: Average inelastic services with different types of service distributions

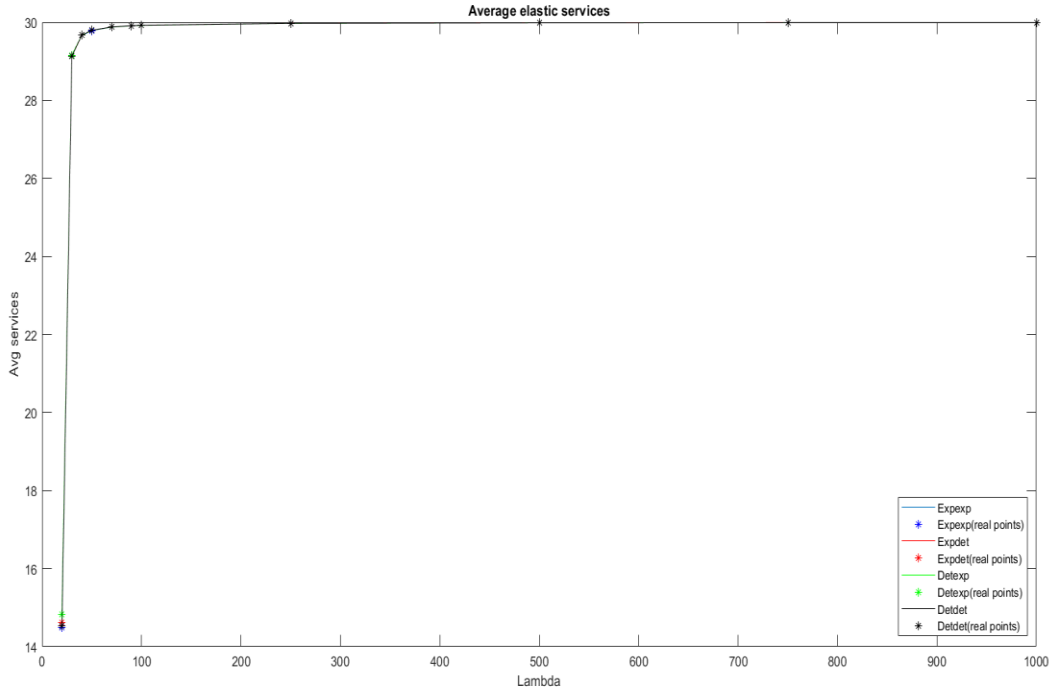

Figure 4.4: Average elastic services with different types of service distributions

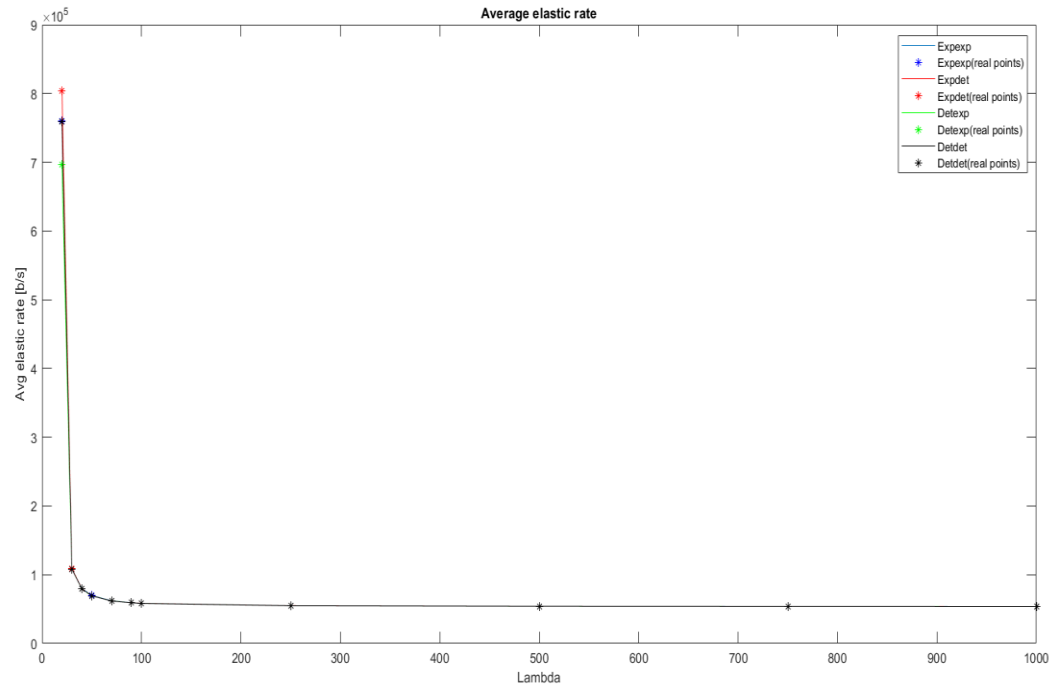

Figure 4.5: Average elastic rate with different types of service distributions

For all 5 statistics the following values of lambda have been used: 20, 30, 40, 50, 70, 90, 100, 250, 500, 750, 1000.

The first thing that can be immediately noticed is that, in all the statistics analysed, the curves drawn for different combination of distributions are overlapping. It is necessary to zoom a lot inside the pictures in order to see very small differences among them.

This leads to a very simple conclusion: changing the distribution of the file size (inelastic or elastic) do not affect neither the loss probability, average services, nor the average elastic rate. In particular, I tested the exponential and deterministic distributions.

Anyway, an interesting observation can be done comparing the first 2 curves: both have a positive-exponential shape, but the first one (probability of inelastic losses) has a slower increase with respect to the second one. In particular, the probability of elastic losses increases very fast when lambda changes from 20 to 30; it has a jump from around 5% to around 60%.

This situation can be matched with the curve related to the average elastic services; again, a huge jump from 20 to 30 happens; it changes from 14 elastic services in the queue to around 29. This makes sense with the big increase of elastic losses.

As a consequence, the average elastic rate drastically decreases when lambda changes from 20 to 30; it varies from around 700 Kb/s, to around 100 Kb/s, as shown in the last curve.

In conclusion, by choosing these parameters, big changes happen when the arrival rate is between 20 and 30.

So, since all the combination of distributions give almost the same results, the statistics computed for this scenario are:

Table 4.7. Statistics of the fourth simulated case for $\lambda = 20$

Statistic	Value	Unit of measure
Inelastic losses	454	Pure number
Elastic losses	12039	Pure number
Inelastic handovers	1410	Pure number
Elastic handovers	1226	Pure number
Inelastic completed services	248855	Pure number
Elastic completed services	237092	Pure number
Average inelastic customers in service	16.4901	Pure number
Average elastic customers in service	14.3669	Pure number
Average inelastic fraction of capacity (%)	49.7618	Pure number
Average elastic fraction of capacity (%)	47.4869	Pure number
Average queue load	9.72487e+006	Bit/s
Average elastic bit rate	781290	Bit/s

Table 4.7 refers to the case with λ equal to 20, instead when it becomes 30:

Table 4.8. Statistics of the fourth simulated case for $\lambda = 30$

Statistic	Value	Unit of measure
Inelastic losses	30246	Pure number
Elastic losses	216993	Pure number
Inelastic handovers	1967	Pure number
Elastic handovers	2398	Pure number
Inelastic completed services	343684	Pure number
Elastic completed services	156891	Pure number
Average inelastic customers in service	22.8772	Pure number
Average elastic customers in service	29.1558	Pure number
Average inelastic fraction of capacity (%)	68.5831	Pure number
Average elastic fraction of capacity (%)	31.4156	Pure number
Average queue load	9.99987e+006	Bit/s
Average elastic bit rate	108626	Bit/s

An important observation arises: completed elastic services decreases even though lambda increases; in fact, the increase of the elastic losses is exponential, as shown by means of the graphs above. These turns into an unbalanced system, because 68% of the capacity will be reserved for inelastic customers, on average; on the contrary, for lambda equal to 20, the situation was more balanced, where both inelastic and elastic services were using around 48% of the capacity.

The explanation is given by the fact that, as soon as inelastic services increase, the rate associated to the elastic drastically reduces, and so they will stay in the queue for a much longer time; so, when new elastic customers arrive, they will probably find the capacity empty, or the maximum number of elastic customers in service, and then they will get lost. Instead, the increase of completed inelastic services is much more linear, because their rate is fixed.

As a conclusion, then, the more we increase the arrival rate, the more elastic losses we will have, resulting in an unbalanced system which tends to favour inelastic customers.

Also the increase of completed inelastic services gets slower and slower as lambda increases; so, it is inefficient to increase lambda too much; it will just result in many customers to get lost, the only advantage is the usage of most of the capacity, which increases more and more.

On the contrary, if we are interested to get a balanced system, or to minimize the percentage of losses, a lower value of arrival rate would be reasonable.

In the end, also the 95% confidence intervals have been computed; in order to do so, it's been necessary to compute, first of all, the mean among all the 4 combination of distributions, and then computing the standard deviation. Then, the next step was to take the right value of the t_student function from the relative tables, in this case it was the one associated to the third row, and column under 0.025; respectively, the number of samples minus one, and $\frac{1-0.95}{2}$.

Once all these parameters have been computed, the confidence intervals can be computed by performing:

$$[mean - t_{stud}\left(\frac{stdDev}{\sqrt{N}}\right), mean + t_{stud}\left(\frac{stdDev}{\sqrt{N}}\right)] \qquad (4.17)$$

Where N, in our case, is 4, since we got 4 combination of distributions.

The confidence intervals, obtained for the different statistics, are hard to be shown on a plot, because, again will be values very near to the computed one, so it is more useful to show them in tables:

Table 4.9. Inelastic losses (%) confidence intervals

0.1342	7.9230	23.094	35.983	53.317	63.279	66.765	86.517	93.250	95.495	96.622
0.2243	8.3962	23.548	36.520	53.712	63.420	66.982	86.632	93.294	95.538	96.641

Table 4.10. Elastic losses (%) confidence intervals

3.603	57.439	75.717	82.985	89.135	91.843	92.777	97.270	98.649	99.106	99.326
6.140	58.265	76.426	83.421	89.454	92.106	92.969	97.346	98.693	99.135	99.351

Table 4.11. Inelastic services confidence intervals

16.535	22.849	25.449	26.431	27.166	27.438	27.513	27.845	27.927	27.952	27.965
16.643	22.941	25.517	26.473	27.179	27.440	27.520	27.846	27.928	27.953	27.965

Table 4.12. Elastic services confidence intervals

14.394	29.121	29.664	29.788	29.875	29.910	29.921	29.971	29.986	29.990	29.993
14.843	29.172	29.672	29.795	29.880	29.913	29.923	29.972	29.986	29.991	29.993

Table 4.13. Elastic rate confidence intervals

6.84e5	1.07e5	7.92e4	6.91e4	6.17e4	5.91e4	5.83e4	5.49e4	5.40e4	5.38e4	5.36e4

8.24e5	1.09e5	8.00e4	6.96e4	6.20e4	5.92e4	5.83e4	5.49e4	5.40e4	5.38e4	5.36e4

Basically, these intervals mean that: given that mean, and that standard deviation, for each statistic, if we run many simulation, then 95% of the times the value given by the simulator will fall in the specific confidential interval associated to that statistic.

In addition, it is possible to compute the accuracy (or error) with respect to the mean, associated to each confidential interval; it's computed as:

$$Accuracy = \frac{upper\_bound - lower\_bound}{mean} \qquad (4.18)$$

The lower this value is, the more reliable the results are, because it basically computes the distance between the mean and the bounds.

In order to improve the accuracy, a possible solution could be to run many simulations; in this way, the confidential intervals will automatically shrink, and so the distance from the mean will be lower.

The error associated to our confidential intervals are:

Table 4.14. Error on inelastic losses (%) confidence intervals

0.5030	0.0580	0.0195	0.0148	0.0074	0.0022	0.0032	0.0013	4.715 e-4	4.4201 e-4	2.0303 e-4

Table 4.15. Error on elastic losses (%) confidence intervals

0.5206	0.0143	0.0093	0.0052	0.0036	0.0029	0.0021	7.7817 e-4	4.4087 e-4	2.8596 e-4	2.5430 e-4

Table 4.16. Error on inelastic services confidence intervals

0.0065	0.0040	0.0027	0.0016	4.986 e-4	8.337 e-5	2.406 e-4	4.572 e-5	2.849 e-5	1.972 e-5	9.292 e-6

Table 4.17. Error on elastic services confidence intervals

0.0307	0.0018	2.673 e-4	2.412 e-4	1.387 e-4	1.107 e-4	7.439 e-5	3.126 e-5	1.733 e-5	1.812 e-5	8.664 e-6

Table 4.18. Error on elastic rate confidence intervals

0.1848	0.0105	0.0099	0.0082	0.0045	0.0016	0.0014	2.904 e-4	1.005 e-4	1.940 e-4	1.011 e-4

As can be noticed, the error decreases as λ increases, that's because as λ increases, the bounds of the confidential intervals tend to shrink, and, as a consequence, it results in higher accuracy.

4.5 Fifth tested case: testing lognormal distribution

As we have already tested deterministic and exponential distribution for packet size, it is interest to see what happens in the case of a high-variance distribution, such as the lognormal one.

This type of distribution is described such as its logarithm is normal distributed with parameter μ, which is the mean, and σ, which is the standard deviation. In particular, we want to test this distribution with the same mean of the exponential packet size, but with variance bigger 10 times with respect to that case.

In order to do so is necessary to apply some formula; first of all, it is necessary to know which are the mean and the standard deviation of an exponential distribution, they are described as follows:

$$\mu = \frac{1}{\lambda} \qquad (4.19)$$

$$\sigma^2 = \frac{1}{\lambda^2} \qquad (4.20)$$

Formula 4.19 says that the mean is the inverse of the parameter of the distribution, while the 4.20 means that the variance is the inverse of the square of the parameter.

By knowing that, since we had a packet size that is, on average, 500000 bits, as can be read in the case 4.3, then applying 4.19, the parameter λ is:

$$\lambda = \frac{1}{\mu} = \frac{1}{500000[bit]} = 2 \times 10^{-6}[1/bit] \qquad (4.21)$$

So now, using formula 4.20, it is possible to compute its variance:

$$\sigma^2 = \frac{1}{\lambda^2} = \frac{1}{2 \times 10^{-12}[1/bit^2]} = 5 \times 10^{11}[bit^2] \qquad (4.22)$$

As we sad previously, we want to test a lognormal distribution which has a variance 10 times bigger that the exponential case, so reading 4.22, it means we want a lognormal distributions with variance equal to $5 \times 10^{12}[bit^2]$.

Once we know that, it is necessary now to know how the mean and the variance of the lognormal distribution are computed:

$$Mean = \exp(\mu + \frac{\sigma^2}{2}) \qquad (4.23)$$

$$Variance = (\exp(\sigma^2) - 1)(\exp(2\mu + \sigma^2)) \qquad (4.24)$$

In formula 4.23 and 4.24, μ and σ are, respectively, the mean and the standard deviation of the underlying normal distributed function, and they are our unknowns. In order to compute them, it is necessary to create a system of 2 equations and 2 unknowns, where the equations are formula 4.23 and 4.24. This is possible because we know exactly which values of mean and standard deviation of the lognormal we expect.

Since this is a system of non-linear equations, I used Matlab to solve it, by the following code:

```
syms x y %%x=mu and y=sigma
eqn1 = exp(x+(y^2)/2) == 5e+5;
eqn2 = (exp(y^2)-1)*exp(2*x+y^2) == 5e+12;

[solx,soly] = solve(eqn1, eqn2)
```

In this way, we got solx which contains μ, and soly which contains σ.

Now, given these 2 parameters, the resulting lognormal distribution is the following:

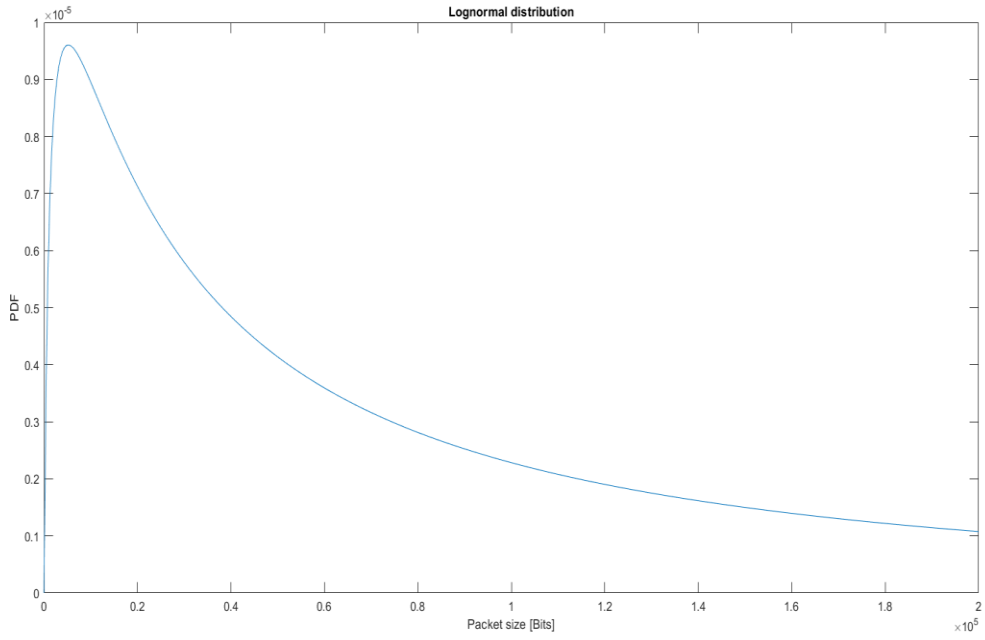

Figure 4.6: Lognormal distribution with mean 500000 bits, and 5×10^{12}

Once we know the parameters μ and σ, we can apply the lognormal distribution to the simulation code by simply writing:

```
Act_1.Server.packetSize_inel = lognormal(11.6,1.744856)
```

The values in the parenthesis are respectively μ and σ.

After the run ends, as usual, the values are collected and copied in Matlab, so that we can plot the figures of the statistics in which we are interested to.

It turned out that:

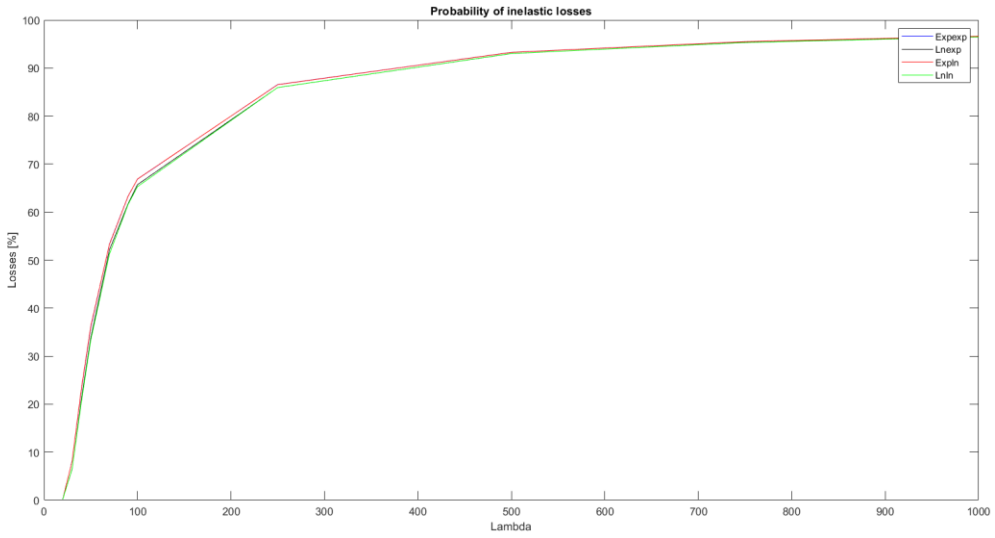

Figure 4.7: Probability of inelastic losses implementing the lognormal distribution

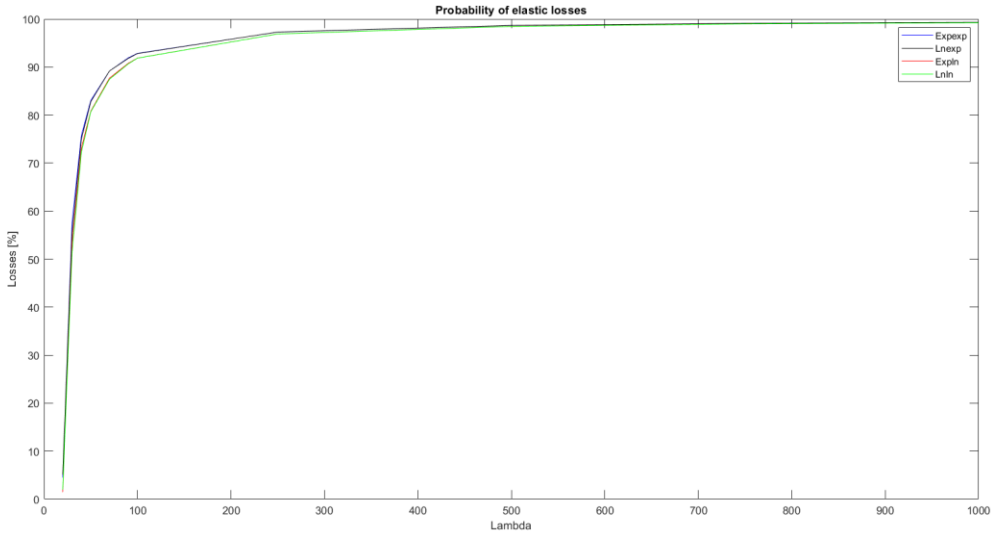

Figure 4.8: Probability of elastic losses implementing the lognormal distribution

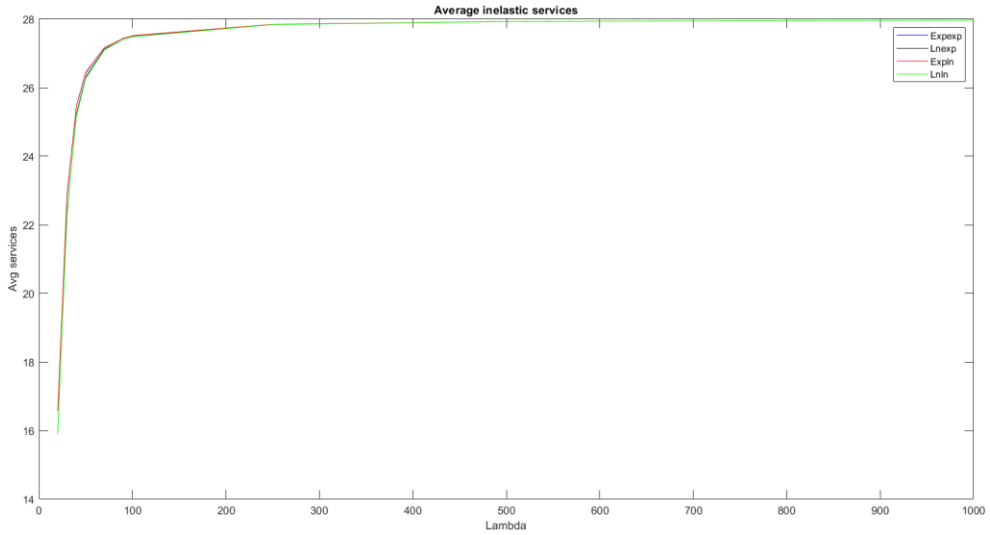

Figure 4.9: Average inelastic services implementing the lognormal distribution

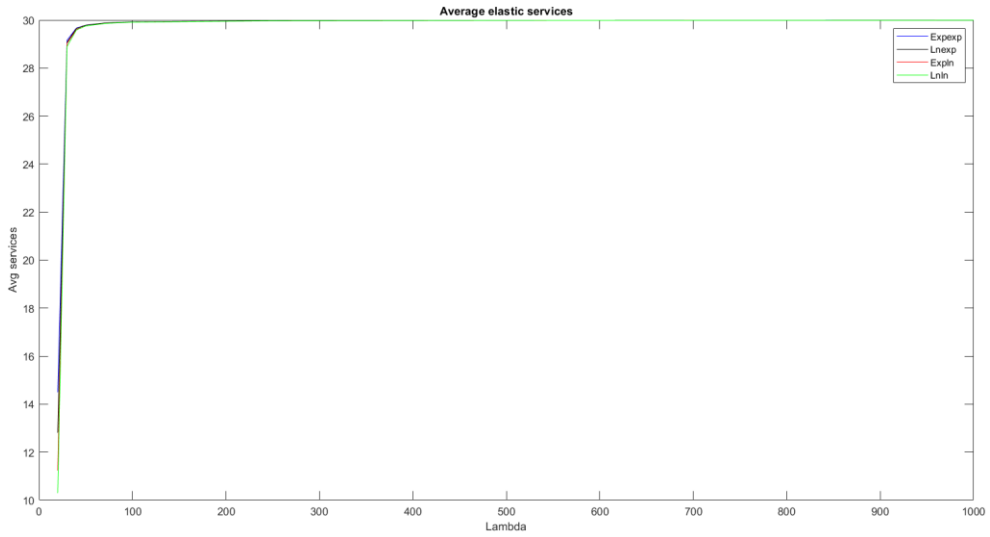

Figure 4.10: Average elastic services implementing the lognormal distribution

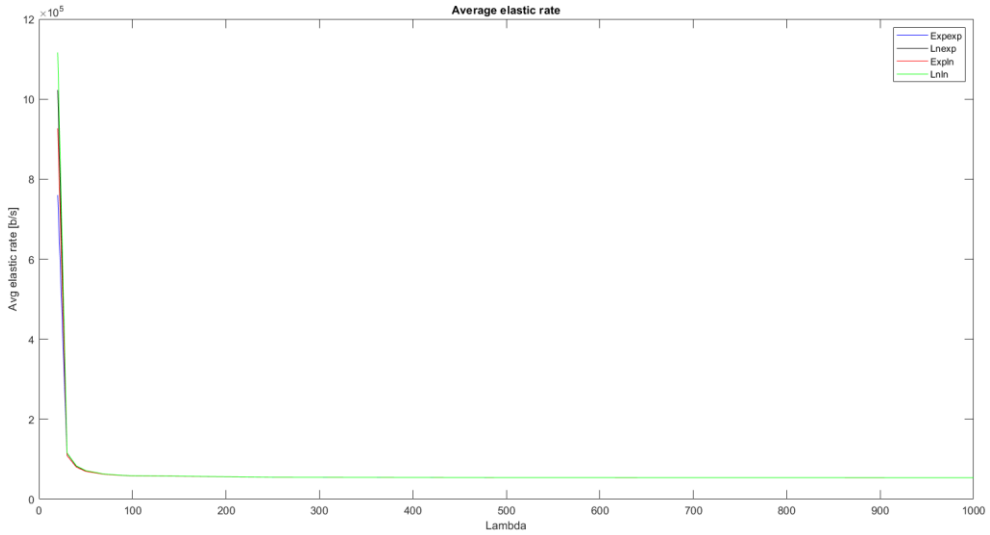

Figure 4.11: Average elastic rate implementing the lognormal distribution

As can be observed by the figures from 4.7 to 4.11, the implementation of the lognormal distribution as packet size, and so as service time, does not change significantly the statistics.

The only difference, which is negligible, is represented by figures 4.7 and 4.8; the loss of the packets is slightly bigger when their sizes are exponentially distributed. In fact, curves red and blue are slightly above the others in figure 4.7; the same happens in figure 4.8 for the blue and black ones.

Anyway, the Matlab code which generates all these figures will be written in the Appendix, so that the figures could be zoomed.

4.6 Sixth tested case: changing dwell time

In this specific case we tested the same values of λ of the case 4.4, but we took into account just the combination of exponential file size distributions. In addition, 5 values of the mean of the dwell time have been tested, which are: 50, 100, 300, 500 and 1000 seconds. In this way, it is possible to understand which is the impact of the dwell times on the statistics. The figures are:

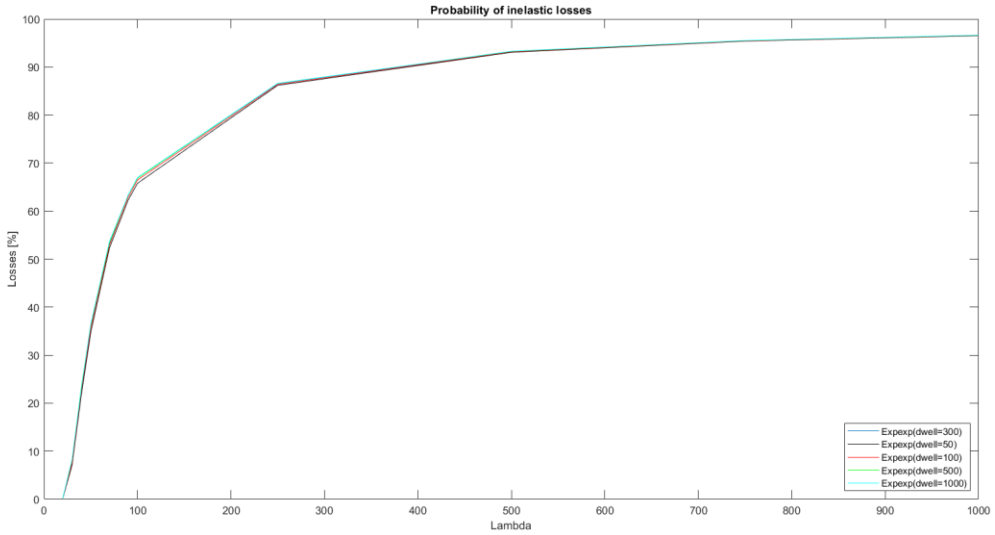

Figure 4.12: Probability of inelastic losses with different dwell time means

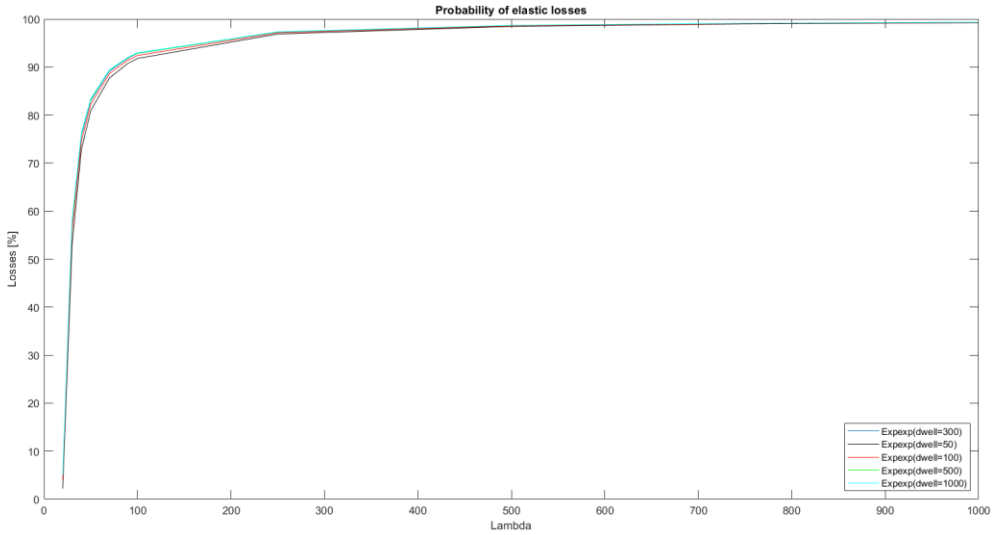

Figure 4.13: Probability of elastic losses with different dwell time means

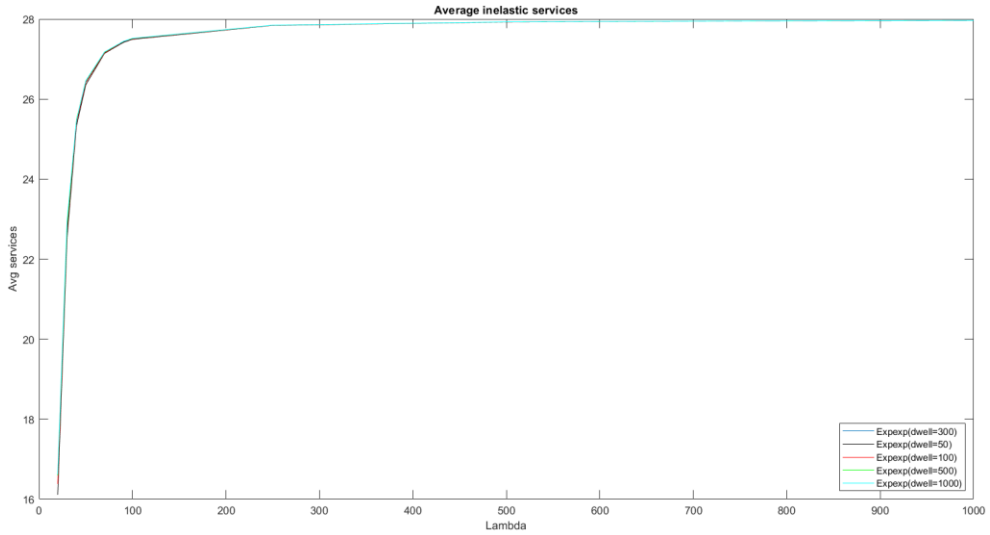

Figure 4.14: Average inelastic services with different dwell time means

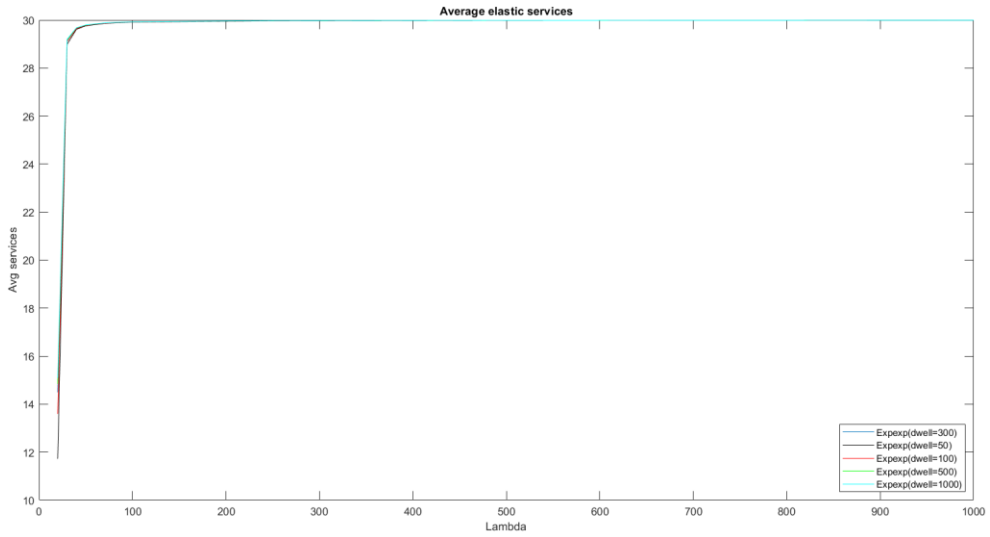

Figure 4.15: Average elastic services with different dwell time means

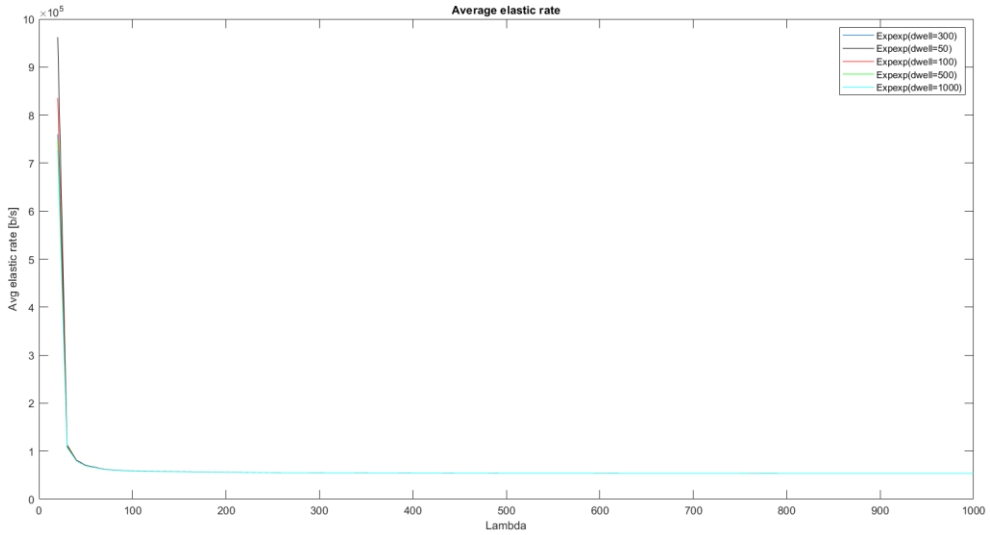

Figure 4.16: Average elastic rate with different dwell time means

Like in the case 4.4, changing this parameter does not affect so much these statistics, some little difference can be notices by looking at figures 4.12 and 4.13, in which we can see that probability of losses is slightly increasing as dwell means; for what concern, instead, figure 4.14, 4.15, and 4.16, no significant changes arise. It would be necessary to zoom a lot in order to notice some difference in the curves.

Since a packet can both perform handover or complete its service, if we increase the mean of the dwell time, it will probably complete its service in a reasonable time, such that it does not affect so much the losses.

4.7 Seventh tested case: testing extreme dwell time of 1 second

The previous case demonstrated that changing the dwell time does not affect the statistics; anyway, if we have smaller dwell times, we expect to have lower load in the cell, because the services will remain in the cell for a smaller time, and as a consequence there should be less customers in service, and so less losses.

So, why this conclusion is not matched by the previous case?

First of all, let's think to two different situations: low arrival rate, and high arrival rate. In the first case, there will be almost always few customers in the queue, meaning that each one will have a low service time, in particular, given the parameters of the case 4.4 and assuming that there is only one customer in the queue, it would be $\frac{500000[bit]}{300000[bit/s]} = 1.67[s]$ for inelastics, which is almost 30 times less than 50 seconds, which is the minimum dwell time tested in the case 4.6. This means, that varying the dwell time from 50 to 1000 seconds does not change the statistics because the service times are significantly smaller, and so the handovers do not affect the behaviour of the inelastic customers in service most of the time. This result is even more evident for elastic services, because assuming there is just one elastic customer in the queue, it will take the whole capacity, resulting in a service time which is much smaller than the dwell time.

For what concerns, instead, the high arrival rate case, the scenario does not change for inelastic customers, because their rate is not affected by the number of services in the queue, and as a consequence their service time remains 1.67 seconds, again much smaller than the minimum tested dwell time. Instead, the things change for elastic ones; now, it is true that low dwell time implies lower load in the cell, but, since we are working at high arrival rate, as soon as a client performs handover and exit the queue, then a new one will take its place, and so the queue will be overloaded again, so the advantage is negligible, and that's why there are no big differences in the figures from 4.6 to 4.10. The only small difference can be noticed in the figures 4.6 and 4.7; because, even if there will be more or less almost the same number of customers in the queue, this means that if we are working at low dwell times, then there will be both handover and losses, while if we have high dwell times, then the packets exiting the queue will be almost always lost.

In order to make the dwell time change the statistics, it has to be of the same order of magnitude of the service time; so, if we consider the service time of inelastic customer, which is 1.67 second, we could try to run a simulation with a dwell time equal to 1 second, so that handover and completion times should be almost equal on average.

The figures of the statistics are the following:

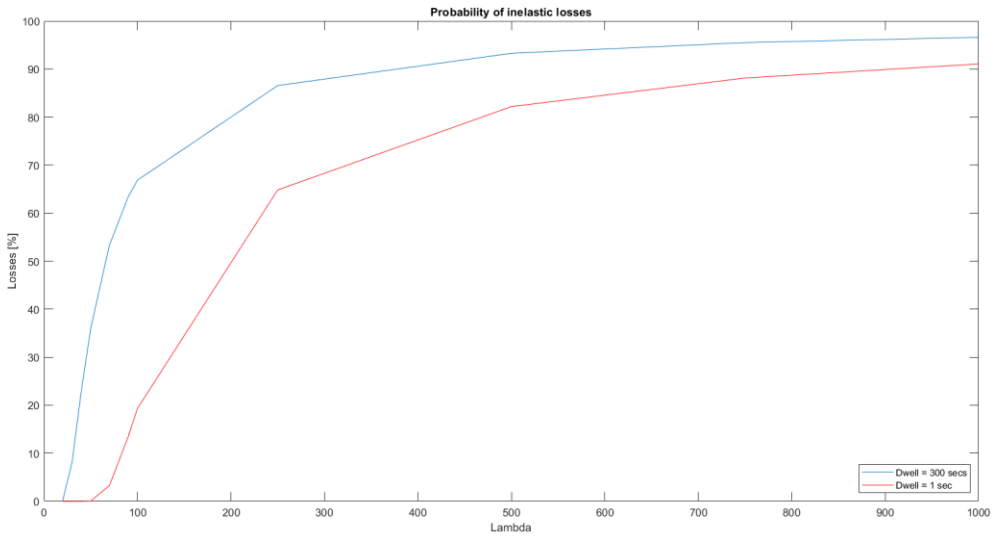

Figure 4.17: Probability of inelastic losses with dwell 1 second

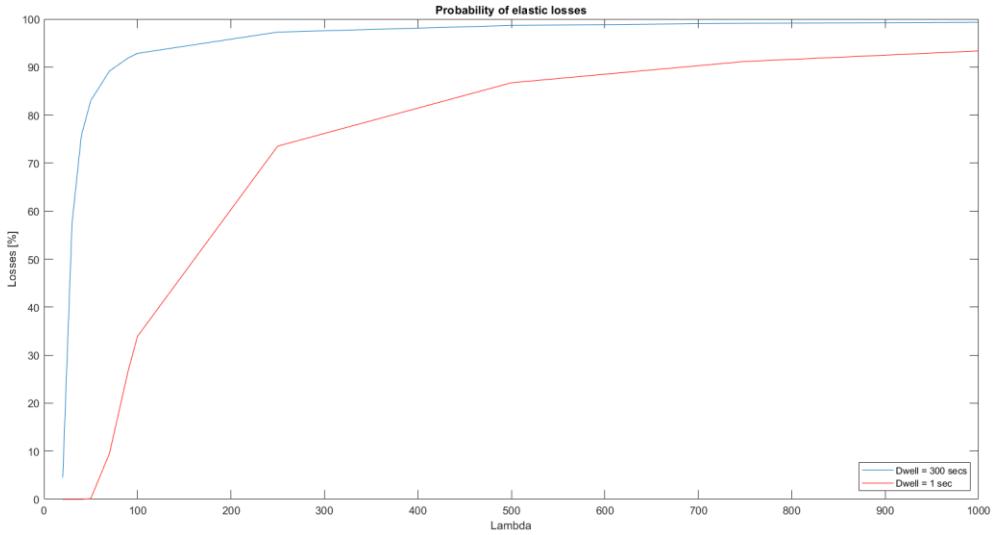

Figure 4.18: Probability of elastic losses with dwell 1 second

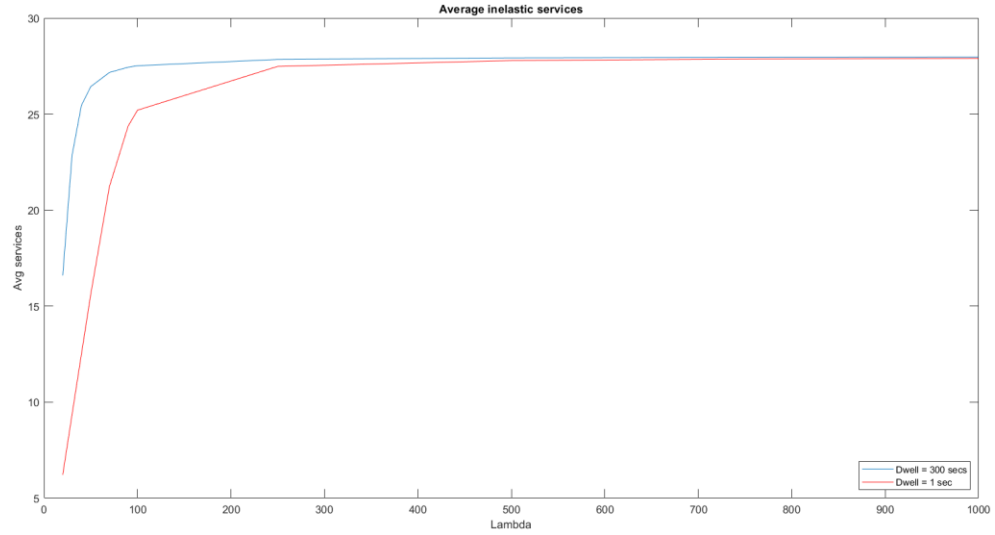

Figure 4.19: Average inelastic services with dwell 1 second

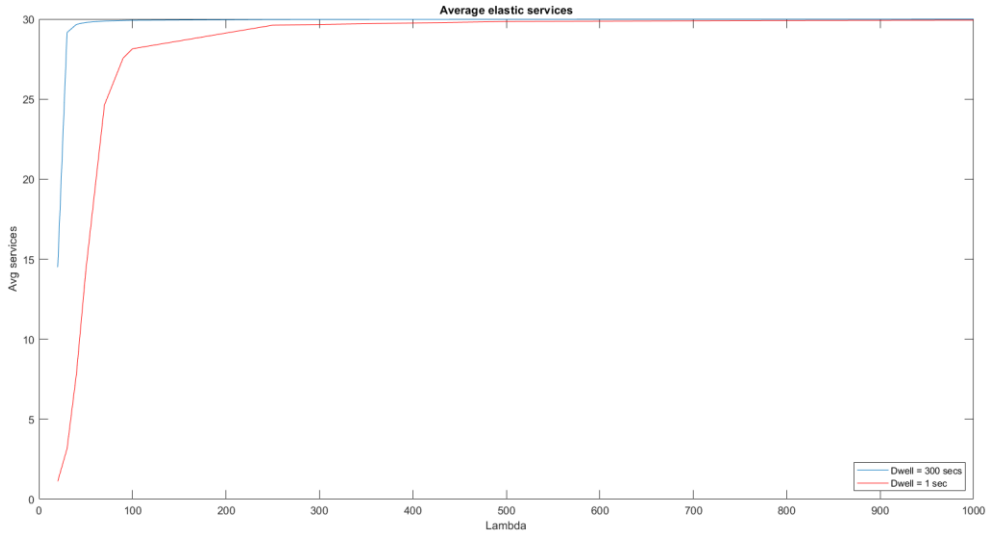

Figure 4.20: Average elastic services with dwell 1 second

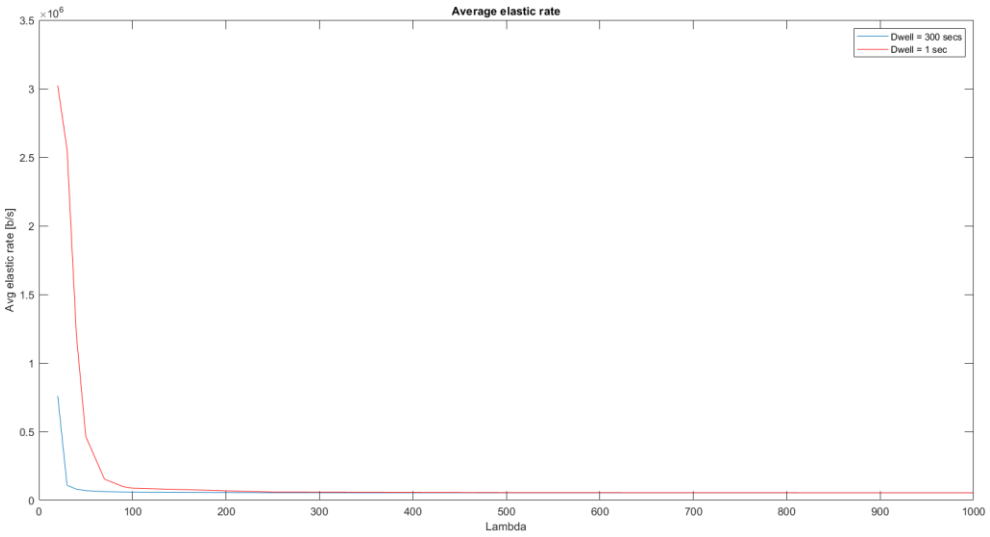

Figure 4.21: Average elastic rate with dwell 1 second

As expected, now the statistics change a lot; the average number of customer in service is smaller for small values of λ, while for higher values there are more or less the same number of customer in the queue for the reason I explained before. The big difference is that, now, even if the queue is full, it does not necessary mean that customers are lost, because dwell time is very low, and as soon as the service time becomes longer, the handover becomes much more probable because it's only 1 second.

As a consequence, the average elastic rate, for high arrival rate, is more or less the same, while for small rates, since there are less customers, is much bigger.

4.8 Eighth tested case: changing elastic size

We have tested different combination of distributions and dwell means; in this case, now, we see how the statistics computed change as we increase or decrease the mean of the elastic packet size, in particular its tested values are: 50, 100, 300, 500, and 1000 Kbits.

The different curves refer to a specific value of elastic size, and the figures are the following:

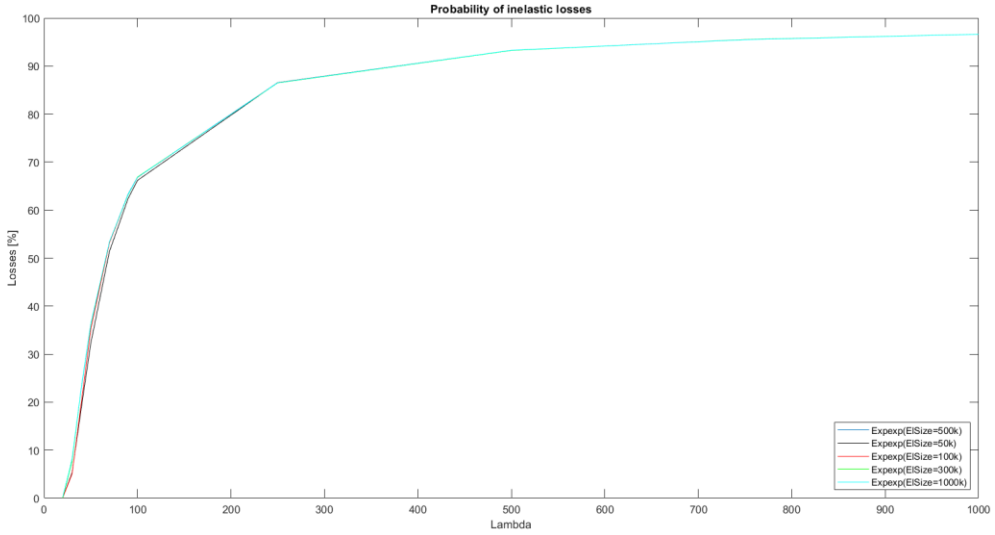

Figure 4.22: Probability of inelastic losses with different elastic sizes

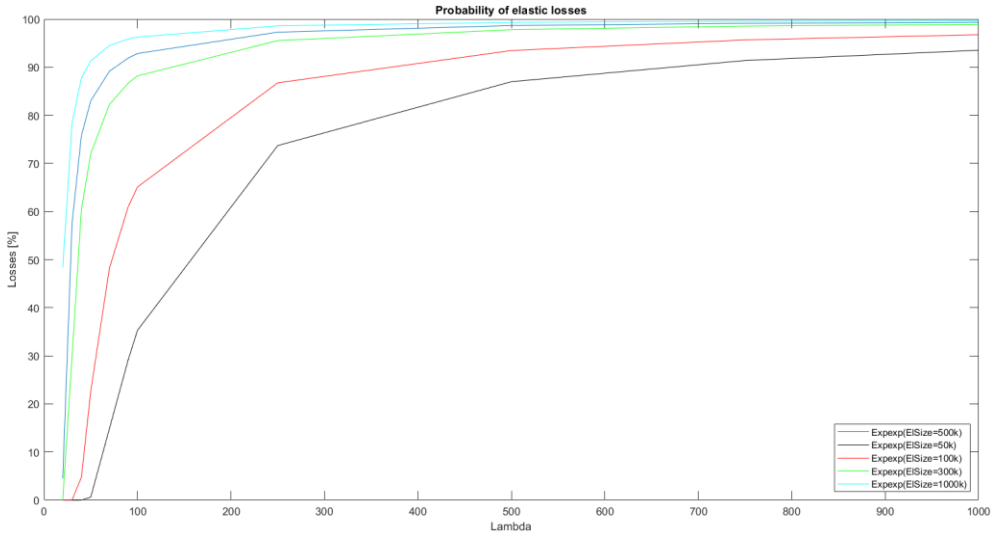

Figure 4.23: Probability of elastic losses with different elastic sizes

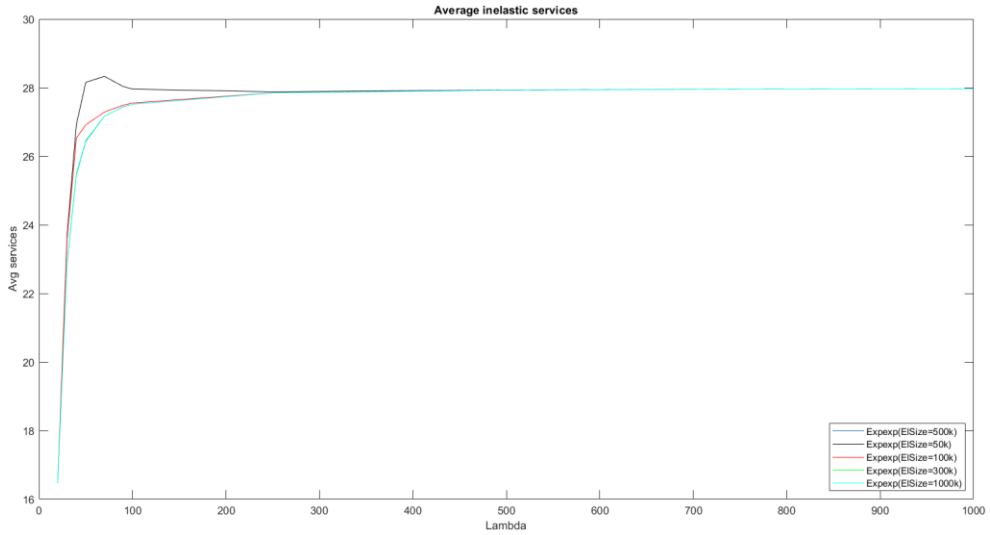

Figure 4.24: Average inelastic services with different elastic sizes

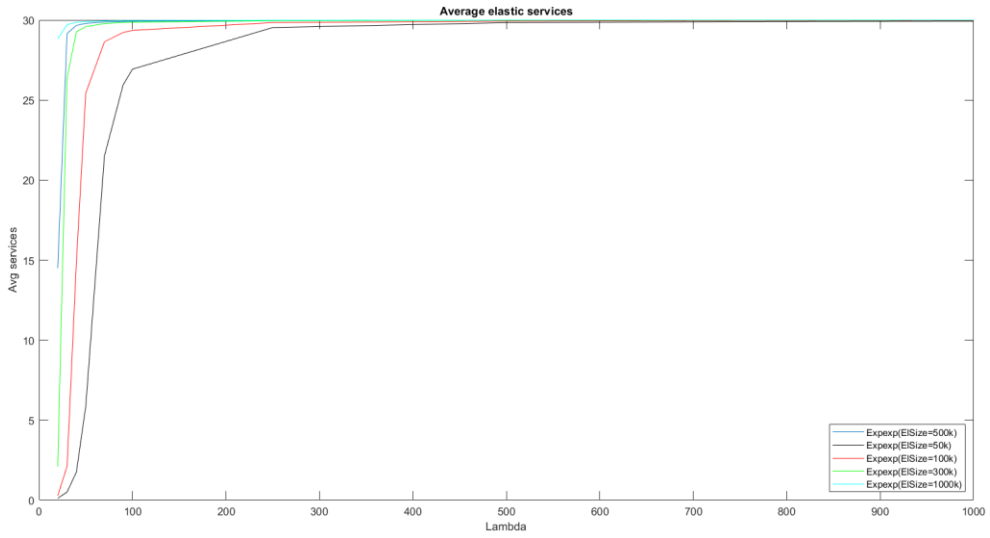

Figure 4.25: Average elastic services with different elastic sizes

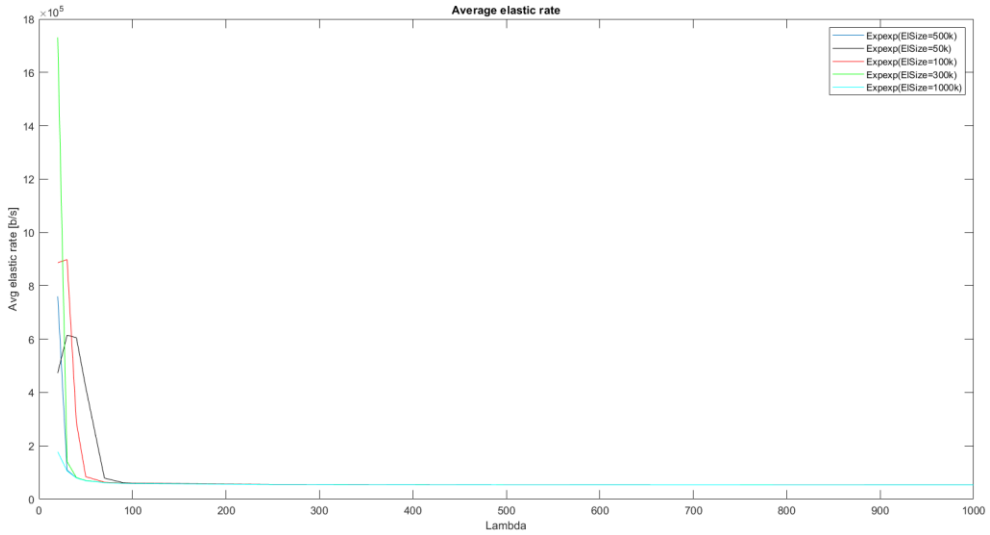

Figure 4.26: Average elastic rate with different elastic sizes

As expected, the statistics regarding inelastic customers do not change so much as the elastic size changes, as it can be noticed by figure 4.22 and 4.24.

Anyway, an interesting observation can be done by looking at the figure 4.24; for all the elastic sizes, the behaviour of the curve is similar to a positive exponential shape, except for the case 50 Kbit. In that case, it seems there is an initial phase in which the average number of inelastic customer in service exponentially increase, and then, around $\lambda = 70$, it starts to exponentially decrease, and, in the end, it stabilizes around the same value of the other curves, which is more or less 28.

A possible explanation of this behaviour could be the following; since the elastic packets, in the black curve, are, on average, 10 times smaller than inelastic ones, it means that usually they are very fast in being served, with respect to inelastics. So, at the beginning, inelastic services accumulate in the queue, but, around $\lambda = 70$, many elastic customers are stored in the queue, as can be noticed by figure 4.25. In particular they increase from 5 (for $\lambda = 50$) to 22 (for $\lambda = 70$). It means that it is not possible anymore to have so many inelastic customers in queue, otherwise it would exceed the total capacity, and so we can observe the exponentially decrease, above described, before the stabilization around 28.

For what concern, instead, elastic statistics, we can observe at first the figure 4.23. It represents the probability of elastic losses, and it clearly show that the they increase as the elastic size increases.

This is true, because as the size increases, it means that elastics need more and more time to complete their services; so, it means that each elastic, on average, will stay in the queue for a longer time, and these turns in a queue with no available capacity to room the new arrival ones.

This situation is confirmed by the figure 4.25, in which we see that the number of average elastic customers in the queue will always stabilize around 30, but this value is reached sooner as the elastic size increases.

For what regards the elastic rate, its behaviour is reported in figure 4.26, and something very strange happens.

All the 5 curves will tend, at the end, to a value near to 50Kb/s, which happens when we have all 30 elastic customers in the queue; but the interesting thing, is the value that each curve assumes when we have low values of λ, such as 20, 30, 40, 50 and 70. If we try to zoom figure 4.15, we see that from $\lambda = 70$ each curve assumes, more or less, the same values, and at the end they stabilize to a common value.

Instead, if we consider low values of λ, each curve has a very different behaviour from each other; considering these low arrival rates, we can notice that, up to an elastic size equal to 300 Kb/s, an increase in the elastic bit rate happens, and then from sizes greater than 500 Kb/s, instead, the elastic bit rate decreases.

The reason of the increase, up to the case 300 Kb/s, is because, for small values of λ, if the size is too small, then they will be immediately served, even before a new elastic customer arrives, leaving no elastic customers in queue for a reasonable amount of time, and so 0 elastic associated rate for that interval of time; that's why, in this specific scenario, the more is the elastic size, the more is the elastic rate.

But, if the packet size becomes too large, then it means that each customer will require too much time to get served, and so they will remain in the queue for a longer time; this will cause a huge increase in the average number of elastics in service (notice figure 4.25), leading the customers to be served to a rate near to the minimum asked one, which is 50 Kb/s.

5. Appendix

The simulations of all the scenarios and cases have been tested by means of a code written in Omnet++, which is a simulation library and framework, primarily for building network simulators.

In this framework, it is possible to create networks composed by modules; each module is descripted by 3 type of codes, as already mentioned before:

- .INI: this file contains all the values of the parameters that will be used by the module. The grammar to do that is:

<module_name>.<parameter_name>=VALUE

- .NED: this file designs the topology of the network and assigns parameters to the modules.
- .C++: this is the core of each module, because here is written what each module do by using its parameters, or local variables of the code.

This means that each module has its own C++ code, while the .INI and .NED files are common to all of them.

Obviously, in order to simulate a network, it is necessary to assign at each module a precise task, so that, at the end, the connection among all of them simulates a network model. So, this means that modules need to exchange messages among each other, and in Omnet++ they are of type cMessage.

This class of messages allows us to use different functions, called “methods”; each method is used to modify or to see the parameters associated to that message. For example:

<message_name> -> isSelfMessage()

This method returns 1 if the message is a self-message, which means if the message has been generated and received by the same module.

Once the cMessage has been received by the module, then it can also be converted in a cPacket, by means of the line of code:

check_and_cast<cPacket*>(message_name)

This performs a cast from the class cMessage to cPacket, which allows us to inspect additional parameters of the message, such as getting its dimension in bit by doing:

<packet_name> -> getBitLength()

Unfortunately, the parameters in the class cPacket were not sufficient to simulate well the network; that’s why I needed to create another type of file, which is of type .MSG, and the code is the following:

```
packet myPacket extends cPacket
{
```

```
    int flowId; //integer value: if it is assigned to 1 means pkt is inelastic, to 2 means elastic
    int position; //integer value: used to know the packet to which row of the matrix is linked to
    int overallPosition; //integer values: position of pkts in the queue. VALUE NOT USED
};
```

Reading the first row can be noticed that this type of file is creating a new class of packets called “myPacket”, which is an extension of the class cPacket. It basically has all the parameters associated to the cPacket class, but in addition, it also has the new parameters that I created, which in this code are:

- flowId: it is equal to 1 if the packet is an inelastic one, or 2 if it’s an elastic one;
- position: this parameter will be explained later

Coming back to the other files, the first one that has to be written is the .NED; this is because it creates the topology of the network and assign the parameters to the modules.

My .NED code is:

```
simple generator
{
  parameters:
    volatile double interArrivalTime @unit(s); //arrival time in seconds taken as parameter
    double inel_prob; //probability that the generated customer is inelastic, taken as parameter
  gates:
    output out; //output gate of generator
}


simple server
{
  parameters:
    int capacity; //maximum capacity of server, taken as parameter
    int needRate_inel; //rate needed by an inelastic service, taken as parameter
    int needRate_el; //minimum rate needed by an elastic service, taken as parameter
    int MAXinel; //maximum number of inelastic services in the queue, taken as parameter
    int MAXel; //maximum number of elastic services in the queue, taken as parameter
    volatile double dwell_time @unit(s); //dwell time in seconds taken as parameter
    volatile double packetSize_inel @unit(b); //pkt size of inelastics in bits, taken as parameter
    volatile double packetSize_el @unit(b); //pkt size of elastics in bits, taken as parameter
  gates:
```

```
    input in; //input gate of the server
    output out; //output gate of the server
}

simple sink
{
  parameters:
  gates:
    input in; //input gate of sink
}

network Act_1 //our network
{
  //here I create the modules
  @display("bgb=538,152");
  submodules:
    Gen: generator { //module of generator(Inelastic)
      @display("p=67,85");
    }
    Server: server { //module of server
      @display("p=251,85");
    }
    Sink: sink { //module of sink
      @display("p=466,85");
    }
  connections:
    Gen.out --> Server.in; //output gate of generator sends pkts to input gate of the server
    Server.out --> Sink.in; //output gate of server sends pkts to input gate of the sink
}
```

I decided to divide my network in 3 modules:

- GENERATOR: it creates the packets, and simulates the arrival of both inelastic/elastic customer in the cell;
- SERVER: it elaborates the packets; so it discard/completes the service/perform handover on the base of the current situation in the cell, so it simulates the base station.

- SINK: basically confirms that a packet finished its service, and does not need any more to be served.

So, in first place, it is necessary to create these classes of modules with their respective parameters, and input/output gates (modules are connected among each other through gates); this part is performed by writing:

simple <name_of_module_class>

{

parameters:

LIST OF PARAMETERS

gates:

LIST OF INPUT/OUTPUT GATES

}

Then we create the modules assigning them the class in which we are interested to; in my case one module of the class of generators, one of server and one of sink. Once all modules have been created, then the connections among them have to be specified (indicating exactly which input gate is connected to which output gate). The code of this procedure is:

network <name_of the_network>

{

submodules:

<module_name>: <name_of_module_class>

connections:

<mod_name>.<out_gate> -> <mod_name>.<in_gate>

}

Once the network is set, then the parameters of the modules have to be specified in the .INI file; in particular, my file is:

[General]

network = Act_1 #name of the network to simulate

seed-0-mt = 14 #seed from which values of random variables will be chosen

sim-time-limit =25000s #simulation time in seconds

Act_1.Server.capacity = 10000000 #capacity of the queue expressed in bps

Act_1.Server.needRate_el = 50000 #minimum rate needed by an elastic service in bps

Act_1.Server.needRate_inel = 300000 #rate needed by an inelastic service in bps

Act_1.Server.dwell_time = exponential(300s) #dwell time exponentially distributed (mean in seconds)

#Act_1.Server.dwell_time = 0 # for testing just services without dwells

Act_1.Server.packetSize_inel = exponential (500000b) #pkt size of inelastic services (mean in bits)

Act_1.Server.packetSize_el = exponential (500000b) #pkt size of elastic services (mean in bits)

Act_1.Server.MAXel = 30 #maximum number of elastic services in the queue

Act_1.Server.MAXinel = 30 #maximum number of inelastic services in the queue

#Act_1.Gen.interArrivalTime = exponential(5e-2s) #interarr. time exp distributed (mean 1/lambda)

Act_1.Gen.interArrivalTime= exponential(${lambda=5e-2s,3.33e-2s,2.5e-2s,2e-2s,1.42e-2s,1.11e-2s,1e-2s,4e-3s,2e-3s,1.33e-3s,1e-3s})

#Act_1.Gen.interArrivalTime = exponential(1e-3s) # for testing lambda=1000

Act_1.Gen.inel_prob = 0.5 #probability that the generated customer is inelastic

#Act_1.Gen.inel_prob = # for testing just elastic

The first thing to set is the name of the network; in this way, we can assign modules to a specific network simply declaring it in the .NED, as I did; my net name is Act_1.

Now, it is necessary to decide how long the simulation will run, in seconds; it is important to notice that these are not real seconds, but they are simulation seconds, which run much faster than real ones. In the end, a seed is necessary to take random numbers from that pool, when needed.

Now that general parameters have been decided, we can assign values to parameters of each module. In my specific case, I assigned, first of all, to server:

- Capacity of queue/base station in bps;
- Rates needed by inelastic/elastic customers in bps;
- Dwell time in seconds;
- Packet size of inelastic/elastic customers in bits;
- Max number of inelastic/elastic customer.

Concerning the generator, then:

- Interarrival time in seconds: different values could be run in the same simulation by putting them in {} brackets; this is useful when we want to test different values of lambda;
- Probability that a customer is inelastic.

Now that both topology and parameters are set, it is possible to write C++ code of each module.

I start explaining the generator code:

#include <omnetpp.h>

#include "myPacket_m.h"

#include <time.h>

```
using namespace omnetpp;

class generator : public cSimpleModule
{
  private:
    cMessage *MESSAGE; // self-message of the generator, which trigger the transmission of a message on the output gate
    myPacket *pkt; // message to be sent on the output gate, which is server
    double inel_prob; //this is the probability that a generated customer is inelastic

  public:
    // constructor
      generator(); // constructor
      virtual~ generator(); // destructor

  protected:
      virtual void initialize();
      virtual void handleMessage(cMessage *msg);
      virtual void finish();
};

Define_Module(generator);

generator::generator() {
  // init all msgs to NULL
}

generator::~generator() {
  // delete messages with cancelAndDelete()
}

void generator::initialize()
{
  srand ( time(NULL) ); //this is useful to generate different random numbers at each run
```

```
    inel_prob=par("inel_prob"); //assign prob that generated service is inelastic (written in .ini)
    MESSAGE = new cMessage("generate"); //initialize the self-message
    //schedule the first message after a time according to our Poisson process
    scheduleAt(simTime()+par("interArrivalTime"),MESSAGE);
}

void generator::handleMessage(cMessage *msg)
{
    pkt = new myPacket("PDU"); //generate the pkt of type myPacket to send out
    //I generate a random float number from 0 to 1; if this value is less than inel_prob
    //then the generated customer is inelastic, else it's elastic.
    if(((double)rand()/(double)RAND_MAX)<inel_prob) //means inelastic service
    {
        pkt->setFlowId(1);
    }
    else //means elastic service
    {
        pkt->setFlowId(2);
    }
    send(pkt,"out"); //send pkt to the output gate, which is the server
    //schedule the next message after a time according to our process
    scheduleAt(simTime()+par("interArrivalTime"),MESSAGE);
}

void generator::finish()
{


}
```

As I explained before, the C++ code of the modules are divided in 3 parts: initialize, handleMessage and finish.

In the initialize part, I just set up the cMessage and assigned the value of inelastic probability from the parameter. The scope of the generator is to schedule message to the future, so as soon as it arrives, it triggers the action in the handleMessage part; for that reason, the last row in the initialize part is the scheduling row.

From now on, as soon as a message is received by the module, it will enter the handleMessage part; since I scheduled a cMessage in the initialize, it will enter in the handleMessage. Once we are in, it means that a packet is ready to be sent to the server/base

station, simulating a customer arrival in the cell. So, I created a packet of type myPacket, so I can exploit the additional parameters of this class, and then I decide if it's an inelastic or elastic one.

In order to decide so, I generated a random number varying from 0 to 1, by the line of code:

((double)rand()/(double)RAND_MAX)

The function rand() basically generates a random number, taken from the seed declared in the .INI, ranging from 0 to RAND_MAX, which is a variable already present in the C++ library; in this way, if we divide this number, by RAND_MAX itself, I get a random number ranging from 0 to 1.

Now, if this number, which is a float, is less than the variable representing the inelastic probability, then I decided that this customer is inelastic, otherwise it's an elastic one. It is important to noticed that, in order to do that, I exploited the method "flowId" of my class myPacket; in this way, if I recognize a packet is inelastic, I simply assign 1 to the value of the parameter "flowId", by doing:

pkt->setFlowId(1);

Otherwise I assign 2 if it's an elastic one.

This is very useful, because next modules can simply understand which type of packet is by simply doing:

pkt->getFlowId();

Again if they get 1 it's inelastic, otherwise not.

So now that I generated the packet, and I decided if it's inelastic or elastic, then it is ready to be sent to the module of the server by:

send(pkt,<name_of_output_gate>);

In the end, a new customer arrival has to be scheduled, by:

scheduleAt(simTime()+par("interArrivalTime"),MESSAGE)

It basically schedules the next arrival to "interArrivalTime" seconds, starting from the current simulation time.

In this way, a loop has been created; as soon as a packet is sent to the server, a new one is scheduled, simulating exactly customer arrivals in a cell.

So now the packets will be received by server, which has the following code:

```
#include <omnetpp.h>
#include <time.h>
#include "myPacket_m.h"

using namespace omnetpp;

class server : public cSimpleModule
{
  private:
```

```
int MAXel; //maximum number of elastic customers allowed
int MAXinel; //maximum number of inelastic customers allowed
double** matrix_elastic; //create the matrix that will containt the elastic times
double** matrix_inelastic; //create the matrix that will contain the inelastic times
int k;
int j;
int needRate_inel; //bit rate needed for an inelastic service
int needRate_el; //minimum  bit rate needed for an elastic service
int inel_services; //number of inelastic customers currently in service
int el_services; //number of elastic customers currently in service
simtime_t dwell_time; //dwell time
int tot_capacity; //total capacity(in BITS) of our queue
double tmp;
simtime_t previous_time; //simTime of the previous change of state
double min; //minimum time among all dwell and completion ones
int min_pointer; //number of the row of the matrix containing the minimum time
simtime_t time_to_next_departure; //it's the same of min
simtime_t time_to_next_arrival;
int INEL; //INEL=1 means the minimum time is associated to an inelastic customer
int EL; //EL=1 means the minimum time is associated to an elastic customer
cMessage *DEPARTURE; //self-message that will trigger the departure of service out of queue
int HANDOVER; //HANDOVER=1 means the minimum time is a dwell, so handover happens
int SERVICE; //SERVICE=1 means the minimum time is a completion, so it ends the service
myPacket *pkt; //packet representing out packet, of type myPacket
int capacity; //capacity currently free in the queue
double comparator[4]={-1,-1,-1,-1};
int last_inserted; //will contain the number of the row in matrix of last entered service
cQueue buffer; //simulates our queue, collecting the packets
int el_handover; //number of elastic handovers performed
int inel_handover; //number of inelastic handovers performed
int el_completed; //number of elastic services completed
int inel_completed; //number of inelastic services completed
int discarded_inel; //number of discarded inelastic customers
```

```
    int discarded_el; //number of discarded elastic customers
    cStdDev inel_stats; //collect each time #inelastics in service (used to compute avg at end)
    cStdDev el_stats; //collect each time #elastics in service (used to compute avg at end)
    cStdDev fraction_inel; //collect each time fraction of capacity used by inelastics
    cStdDev fraction_el; //collect each time fraction of capacity used by elastics
    cStdDev queue_load; //collect each time the capacity used (used to compute avg at end)
    cStdDev el_bitrate; //collect each time the bit rate associated to elastic customers
    cStdDev inel_loss;
    cStdDev el_loss;
    cStdDev inelastic_services;
    cStdDev elastic_services;
    cStdDev average_elastic_rate;
    double size; //size of packets in float number
    int size_real; //used to round the packet sized to an integer value
    simtime_t** states; //matrix containing the times spent in each state
    //double proves[100][6]={-1.0};
    int overall;
    double inel_avg_serv;
    double el_avg_serv;
    double inel_avg_fract;
    double el_avg_fract;
    double avg_queue_load;
    double avg_el_br;

  public:
  // constructor
        server(); // constructor
        virtual~ server(); // destructor

  protected:
        virtual void initialize();
        virtual void handleMessage(cMessage *msg);
        virtual void finish();
};
```

```
Define_Module(server);

server::server() {
   // init all msgs to NULL
}

server::~server() {
   // delete messages with cancelAndDelete()
}

void server::initialize()
{
   for(k=0;k<4;k++)
   {
      EV << comparator[1,k] << " ";
   }
   EV<<endl;
   MAXel=par("MAXel"); //assign the max #elatics in queue (specified in .ini file)
   MAXinel=par("MAXinel"); //assign the max #inelatics in queue (specified in .ini file)
   matrix_elastic=new double *[MAXel]; //creates the pointer to each row of elastic matrix
   matrix_inelastic=new double *[MAXinel]; //creates the pointer to each row of inelastic
matrix
   states=new simtime_t *[MAXinel+1]; //creates the pointer to each row of states matrix
   //HERE EACH ROW OF ELASTIC MATRIX HAVE BEEN ASSIGNED 4 CELLS
INITIALIZED TO -1
   for(k=0;k<MAXel;k++)
   {
      matrix_elastic[k] = new double[4];
      for(j=0;j<4;j++)
      {
         matrix_elastic[k][j]=comparator[1,j];
      }
   }
   //HERE EACH ROW OF INELASTIC MATRIX HAVE BEEN ASSIGNED 4 CELLS
INITIALIZED TO -1
   for(k=0;k<MAXinel;k++)
```

```
    {
      matrix_inelastic[k] = new double[4];
      for(j=0;j<4;j++)
      {
        matrix_inelastic[k][j]=comparator[1,j];
      }
    }
    //HERE EACH ROW OF STATES MATRIX HAVE BEEN ASSIGNED MAXEL+1 CELLS INITIALIZED TO 0.0
    for(k=0;k<MAXinel+1;k++)
    {
      states[k] = new simtime_t[MAXel+1];
      for(j=0;j<MAXel+1;j++)
      {
        states[k][j]=0.0;
      }
    }
    needRate_inel=par("needRate_inel"); //assign the rate needed by inelastics (written in .ini)
    needRate_el=par("needRate_el"); //assign the minimum rate needed by elastics (written in .ini)
    inel_services=0; //initialize to 0 the number of current inelastic customers in service
    el_services=0; //initialize to 0 the number of current elastic customers in service
    tot_capacity=par("capacity"); //assign the value of the capacity of the queue (written in .ini)
    capacity=tot_capacity; //initialize the value of free capacity to total capacity
    INEL=0; //initially there's no minimum time, so INEL=0
    EL=0; //initially there's no minimum time, so EL=0
    HANDOVER=0; //initially there's no minimum time, so HANDOVER=0
    SERVICE=0; //initially there's no minimum time, so SERVICE=0
    DEPARTURE = new cMessage("Departure"); //initialize the self-message representing a departure
    previous_time=0.0; //initialize the simulation time of the previous change of state to 0
    el_handover=0; //initialize the number of elastic handovers to 0
    inel_handover=0; //initialize the number of inelastic handovers to 0
    el_completed=0; //initialize the number of completed elastic services to 0
    inel_completed=0; //initialize the number of completed inelastic services to 0
    discarded_inel=0; //initialize the number of discarded inelastic services to 0
```

```
    discarded_el=0; //initialize the number of discarded elastic services to 0
    overall=0;
    tmp=0;

    inel_avg_serv=0.0;
    el_avg_serv=0.0;
    inel_avg_fract=0.0;
    el_avg_fract=0.0;
    avg_queue_load=0.0;
    avg_el_br=0.0;

    inel_stats.setName("Inel_Serv_Stats"); //Give a name to all the statistics
    el_stats.setName("El_Serv_Stats");
    fraction_inel.setName("Inel_Fraction_Capacity");
    fraction_el.setName("El_Fraction_Capacity");
    queue_load.setName("Queue_Load");
    el_bitrate.setName("Elastic_Bitrate");
    inel_loss.setName("Inelastic_Losses");
    el_loss.setName("Elastic_Losses");
    /*inelastic_services.setName("Inelastic_Services");
    elastic_services.setName("Elastic_Services");
    average_elastic_rate.setName("Average_Elastic_Rate");*/
}

void server::handleMessage(cMessage *msg)
{
    if(!msg->isSelfMessage())//if pkt is not a self message means arriving service
    {
        EV <<"New packet arrived" <<endl;
        pkt=check_and_cast<myPacket*>(msg); //here I cast from type msg to type myPacket
        if(pkt->getFlowId()==1) //means inelastic service (FlowId=1 means inelastic, 2 elastic)
        {
            avg_el_br=avg_el_br+tmp*(simTime().dbl()-previous_time.dbl());

avg_queue_load=avg_queue_load+(inel_services*needRate_inel+el_services*tmp)*(simTime
().dbl()-previous_time.dbl());
```

```
inel_avg_serv=inel_avg_serv+inel_services*(simTime().dbl()-previous_time.dbl());

inel_avg_fract=inel_avg_fract+(inel_services*(double)needRate_inel/(double)tot_capacity)*(
simTime().dbl()-previous_time.dbl());
el_avg_serv=el_avg_serv+el_services*(simTime().dbl()-previous_time.dbl());
el_avg_fract=el_avg_fract+(el_services*tmp/tot_capacity)*(simTime().dbl()-
previous_time.dbl());
//if there is enough capacity and there are not already the max #inelastics in service
if(((tot_capacity-needRate_inel*inel_services-
needRate_el*el_services)>=needRate_inel) && inel_services<MAXinel)
{
overall++; //increase the counter of total services in queue
buffer.insert(pkt); //the packet is inserted in the queue
size=par("packetSize_inel"); //assign the value of size of packet according to .ini
size_real=((int)round(size)); //round from a double value to an integer
one(NEEDED)
EV <<"Inelastic service of " <<size<<" bits starts receiving "<<needRate_inel/1000
<<" kbps over a capacity of "<<capacity/1000<<" kbps"<<endl;
pkt->setBitLength(size_real); //assign to the packet the INTEGER size
//here I update the time spent in the previous state
states[inel_services][el_services]=states[inel_services][el_services]+(simTime()-
previous_time);
inel_services++; //increase by 1 the number of inelastic customers currently served
dwell_time=par("dwell_time"); //assign the value of dwell time
//HERE I'M LOOKING FOR FIRST FREE ROW (WHERE -1) IN MATRIX TO
ASSIGN THIS CUSTOMER
k=0;
while((matrix_inelastic[k][0]!=-1) && k<MAXinel)
{
k++;
}
pkt->setPosition(k); //as soon as I found, I assign this value to method Position
pkt->setOverallPosition(overall-1);
last_inserted=k; //last inserted service has been assigned the number of the row
matrix_inelastic[k][0]=pkt->getBitLength(); //assign pkt size to the first cell
matrix_inelastic[k][1]=dwell_time.dbl(); //assign dwell time to the second cell
matrix_inelastic[k][2]=(matrix_inelastic[k][0])/needRate_inel;//compl.time in 3 cell
matrix_inelastic[k][3]=simTime().dbl(); //current simulation time in fourth cell
```

```
            //if there are elastics in service,their rate must be adjusted once inelastic enters
            if(el_services>0)
            {
               tmp=(tot_capacity-needRate_inel*inel_services)/el_services;
               el_bitrate.collect(tmp); //collect value of bit rate assigned to each elastic
               average_elastic_rate.collect(avg_el_br);
            }
            capacity=tot_capacity-needRate_inel*inel_services-tmp*el_services; //free capacity
            queue_load.collect(tot_capacity-capacity); //collect value of the queue load
         }
         else //means that inelastic service cannot enter the queue
         {
            last_inserted=200; //last inserted has been assigned a random value very big
            //here I update the time spent in the state
            states[inel_services][el_services]=states[inel_services][el_services]+(simTime()-
previous_time);
            discarded_inel++; //increase by 1 the counter of discarded inelastic services
            EV << "Inelastic service lost because asking for " << needRate_inel/1000 << "
kbps, but available capacity is " << capacity/1000 << " kbps"<<endl;
            EV << "Inelastic services " << inel_services << " over a maximum of " <<
MAXinel << endl;
            delete pkt; //delete the packet
         }
      }
      else //means elastic service
      {
         avg_el_br=avg_el_br+tmp*(simTime().dbl()-previous_time.dbl());

avg_queue_load=avg_queue_load+(inel_services*needRate_inel+el_services*tmp)*(simTime
().dbl()-previous_time.dbl());
         inel_avg_serv=inel_avg_serv+inel_services*(simTime().dbl()-previous_time.dbl());

inel_avg_fract=inel_avg_fract+(inel_services*(double)needRate_inel/(double)tot_capacity)*(
simTime().dbl()-previous_time.dbl());
         el_avg_serv=el_avg_serv+el_services*(simTime().dbl()-previous_time.dbl());
         el_avg_fract=el_avg_fract+(el_services*tmp/tot_capacity)*(simTime().dbl()-
previous_time.dbl());
         el_services++; //increase by 1 the number of elastic customers currently served
```

```
//if there is enough capacity such all elastics can adjust their rate to a value bigger
//than the minimum, and there are less elastics in service than the maximum allowed,
//then the elastic customer can enter the queue
if((((tot_capacity-needRate_inel*inel_services)/el_services)>=needRate_el) && el_services<=MAXel)
{
   overall++;
   buffer.insert(pkt); //the packet is inserted in the queue
   size=par("packetSize_el"); //assign the value of size of packet according to .ini
   size_real=((int)round(size)); //round from a double value to an integer one(NEEDED)
   //it is then necessary to update the bit rate associated to each elastic customer
   tmp=(tot_capacity-needRate_inel*inel_services)/el_services;
   el_bitrate.collect(tmp); //collect the bit rate associated to each elastic customer
   average_elastic_rate.collect(avg_el_br);
   EV <<"Elastic service of "<<size<<" bits starts receiving "<<tmp/1000 <<" kbps which is greater than minimum rate "<<needRate_el/1000<<" kbps"<<endl;
   pkt->setBitLength(size_real); //assign to the packet the INTEGER size
   dwell_time=par("dwell_time");//assign the value of dwell time
   //HERE I'M LOOKING FOR FIRST FREE ROW (WHERE -1) IN MATRIX TO ASSIGN THIS CUSTOMER
   k=0;
   while((matrix_elastic[k][0]!=-1) && k<MAXel)
   {
      k++;
   }
   pkt->setPosition(k); //as soon as I found, I assign this value to method Position
   pkt->setOverallPosition(overall-1);
   last_inserted=k+100; //last inserted service has been assigned number of row+100
   matrix_elastic[k][0]=pkt->getBitLength(); //assign pkt size to the first cell
   matrix_elastic[k][1]=dwell_time.dbl(); //assign dwell time to the second cell
   if(tmp!=0) //if the bit rate associated to each elastic is different from 0
   {
      matrix_elastic[k][2]=(matrix_elastic[k][0])/tmp; //completion time in 3 cell
   }
   else
```

```
            {
               matrix_elastic[k][2]=10000.0;
               //compl.time has been assigned a random very big value,to avoid to get infinity
            }
            matrix_elastic[k][3]=tmp;//bit rate assigned to each elastic written in fourth cell
            capacity=0; //free capacity will be 0, since elastic use all the residual capacity
            queue_load.collect(tot_capacity); //collect value of the queue load (=all capacity)
            //here I update the time spent in the previous state
            states[inel_services][el_services-1]=states[inel_services][el_services-
1]+(simTime()-previous_time);
         }
         else //means that elastic service cannot enter the queue
         {
            last_inserted=200; //last inserted has been assigned a random value very big
            discarded_el++; //increase by 1 the counter of discarded elastic services
            EV << "Elastic service lost because " << (capacity/el_services)/1000 << " kbps is
less than the minimum elastic rate " << needRate_el/1000 << " kbps"<<endl;
            EV << "Elastic services " << el_services << " over a maximum of " << MAXel <<
endl;
            el_services--; //decrease el_services by 1, since we did +1 before the check
            //here I update the time spent in the state
            states[inel_services][el_services]=states[inel_services][el_services]+(simTime()-
previous_time);
            delete pkt; //delete the packet
         }
      }
      el_stats.collect(el_services); //collect number of elastic customers currently in service
      inel_stats.collect(inel_services); //collect number of inelastic customers in service
      inelastic_services.collect(inel_avg_serv);
      elastic_services.collect(el_avg_serv);
      //here I collect the fraction of capacity used time by time by inelastic/elastic customers

fraction_inel.collect((((double)(inel_services*needRate_inel))/((double)(tot_capacity)))*100);
      fraction_el.collect((((double)(el_services*tmp))/((double)(tot_capacity)))*100);
      //THE FOLLOWING PART IS NEEDED TO UPDATE TIMES(IN A DECRESCENT
WAY) AT EACH CHANGE OF STATE
      k=0;
```

```
while(k<MAXel)
{
  if(matrix_elastic[k][0]!=-1 && (k!=last_inserted-100)) //except for last inserted
  {
    //update both dwell and completion times by the quantity simTime-previous_time
    //where previous time is the time referred to the previous change of state.
    //So simTime-previos_time gives us exactly the quantity of time past between
    //these 2 change of state. Consequently update also the remaining bit to TX.
    matrix_elastic[k][0]=matrix_elastic[k][0]-(simTime().dbl()-
previous_time.dbl())*matrix_elastic[k][3];
    matrix_elastic[k][1]=matrix_elastic[k][1]-(simTime().dbl()-previous_time.dbl());
    if(tmp!=0)
    {
      matrix_elastic[k][2]=(((matrix_elastic[k][0])*el_services)/(tot_capacity-
inel_services*needRate_inel));
    }
    else
    {
      matrix_elastic[k][2]=10000.0;
    }
    matrix_elastic[k][3]=tmp;
  }
  k++;
}
k=0;
while(k<MAXinel)
{
  if(matrix_inelastic[k][0]!=-1 && (k!=last_inserted))
  {
    matrix_inelastic[k][0]=matrix_inelastic[k][0]-(simTime().dbl()-
previous_time.dbl())*needRate_inel;
    matrix_inelastic[k][1]=matrix_inelastic[k][1]-(simTime().dbl()-
previous_time.dbl());
    matrix_inelastic[k][2]=(matrix_inelastic[k][0])/needRate_inel;
    matrix_inelastic[k][3]=simTime().dbl();
  }
```

```
   k++;
}
previous_time=simTime(); //previous time has been assigned the current simulation time
min=100.0; //initialize the minimum time to a big value, so it will be easily changed
EV << "INELASTIC MATRIX:"<<endl;
for(k=0;k<MAXinel;k++)
{
   if(matrix_inelastic[k][0]!=-1)
   {
      for(j=0;j<4;j++)
      {
         EV <<matrix_inelastic[k][j]<<" ";
      }
      EV<<endl;
   }
}
//here I check for each row of the inelastic matrix different from -1
//the minimum time among all dwell and completion times
k=0;
while(k<MAXinel)
{
   if(matrix_inelastic[k][0]!=-1)
   {
      if(min>matrix_inelastic[k][1])// && dwell_time!=0)
      {
         min=matrix_inelastic[k][1];
         min_pointer=k; //min_pointer assigned the value of the row of the minimum time
         INEL=1; //minimum is inelastic
         EL=0;
         HANDOVER=1; //minimum is dwell, so an handover
         SERVICE=0;
      }
      if(min>matrix_inelastic[k][2])
      {
         min=matrix_inelastic[k][2];
```

```
            min_pointer=k;
            INEL=1;
            EL=0;
            HANDOVER=0;
            SERVICE=1; //minimum is completion, so service is finished
        }
    }
    k++;
}
EV << "ELASTIC MATRIX:"<<endl;
for(k=0;k<MAXel;k++)
{
    if(matrix_elastic[k][0]!=-1)
    {
        for(j=0;j<4;j++)
        {
            EV <<matrix_elastic[k][j]<<" ";
        }
        EV<<endl;
    }
}
//again I search for the minimum time among all times (now for elastics)
k=0;
while(k<MAXel)
{
    if(matrix_elastic[k][0]!=-1)
    {
        if(min>matrix_elastic[k][1])// && dwell_time!=0)
        {
            min=matrix_elastic[k][1];
            min_pointer=k;
            EL=1; //minimum is elastic
            INEL=0;
            HANDOVER=1;
            SERVICE=0;
```

```
            }
            if(min>matrix_elastic[k][2])
            {
               min=matrix_elastic[k][2];
               min_pointer=k;
               EL=1;
               INEL=0;
               HANDOVER=0;
               SERVICE=1;
            }
         }
         k++;
      }
      //at the end of this check, the value of min will be the minimum among all the times
      //so I schedule the departure for min seconds.
      time_to_next_departure=(simtime_t)min;
      if(DEPARTURE->isScheduled()) //if departure is already in schedule, then has to be deleted
      {
         cancelEvent(DEPARTURE);
      }
      scheduleAt(simTime()+time_to_next_departure,DEPARTURE);
   }
   else //means departure
   {
      avg_el_br=avg_el_br+tmp*(time_to_next_departure.dbl());

avg_queue_load=avg_queue_load+(inel_services*needRate_inel+el_services*tmp)*(time_to_next_departure.dbl());
      el_avg_serv=el_avg_serv+el_services*(time_to_next_departure.dbl());

el_avg_fract=el_avg_fract+(el_services*tmp/tot_capacity)*(time_to_next_departure.dbl());
      inel_avg_serv=inel_avg_serv+inel_services*(time_to_next_departure.dbl());

inel_avg_fract=inel_avg_fract+(inel_services*(double)needRate_inel/(double)tot_capacity)*(time_to_next_departure.dbl());
      /*inelastic_services.collect(inel_avg_serv);
```

```
elastic_services.collect(el_avg_serv);*/
previous_time=simTime(); //assign to previous_time the departure time (change of state)
//update again the value of the bit rate assigned to each elastic customer
if(el_services>0)
{
   tmp=(tot_capacity-needRate_inel*inel_services)/el_services;
   el_bitrate.collect(tmp);
   average_elastic_rate.collect(avg_el_br);
}
else
{
   tmp=0;
}
//again updates all times and bit residual in both matrices
k=0;
while(k<MAXel)
{
   if(matrix_elastic[k][0]!=-1)
   {
      matrix_elastic[k][0]=matrix_elastic[k][0]-
(time_to_next_departure.dbl())*matrix_elastic[k][3];
      matrix_elastic[k][1]=matrix_elastic[k][1]-(time_to_next_departure.dbl());
      if(tmp!=0)
      {
         matrix_elastic[k][2]=(((matrix_elastic[k][0])*el_services)/(tot_capacity-
inel_services*needRate_inel));
      }
      matrix_elastic[k][3]=tmp;
   }
   k++;
}
k=0;
while(k<MAXinel)
{
   if(matrix_inelastic[k][0]!=-1)
   {
```

```
            matrix_inelastic[k][0]=matrix_inelastic[k][0]-
(time_to_next_departure.dbl())*needRate_inel;
            matrix_inelastic[k][1]=matrix_inelastic[k][1]-(time_to_next_departure.dbl());
            matrix_inelastic[k][2]=(matrix_inelastic[k][0])/needRate_inel;
            matrix_inelastic[k][3]=simTime().dbl();
         }
         k++;
      }
      if(min<100.0)
      {
         time_to_next_departure=(simtime_t)min;
         //update the time spent in the previous state

states[inel_services][el_services]=states[inel_services][el_services]+time_to_next_departure;
         EV <<"MIN: " << time_to_next_departure << "EL: "<<EL<<"INEL:
"<<INEL<<endl;
         if(EL==1) //minimum is elastic beacuse EL=1
         {
            if(el_services==1) //if there are no others elastic, then free capacity increases
            {
               capacity=capacity+matrix_elastic[min_pointer][3];
            }
            else //free cap is 0 because elastics adapt rates using all the residual capacity
            {
               capacity=0;
            }
            //free the row assigned to the leaving service, so put all -1
            matrix_elastic[min_pointer][0]=-1;
            matrix_elastic[min_pointer][1]=-1;
            matrix_elastic[min_pointer][2]=-1;
            matrix_elastic[min_pointer][3]=-1;
            k=0;
            while(k<buffer.getLength()) //to define queue_length
            {
               pkt=check_and_cast<myPacket*>((cMessage*)(buffer.get(k))); //take the pkt
               //if the pkt we are considering has flowId=2 (means elastic) and
```

```
            //has Position=min_pointer, then it's the pkt we are interested in
            if(pkt->getFlowId()==2 && pkt->getPosition()==min_pointer)
            {
               pkt=check_and_cast<myPacket*>((cMessage*)buffer.remove(buffer.get(k)));
               //as soon as we detect, remove the pkt from the queue and stop while cycle
               break;
            }
            k++;
         }
         if(HANDOVER==1) //means handover
         {
            delete pkt; //delete the packet
            el_handover++; //increase the counter of elastic handovers
            EV <<"Elastic service in position "<<min_pointer<<" has done handover"<<
endl;
         }
         else
         {
            send(pkt,"out"); //send the packet to the sink
            el_completed++; //increase the counter of elastic completed services
            EV <<"Elastic service in position "<<min_pointer<<" has finished"<< endl;
         }
         el_services--; //reduce by 1 the number of elastic customers in service
      }
      else //minimum is inelastic
      {
         //do all the same steps exlained before for elastic ones
         if(el_services==0)
         {
            capacity=capacity+needRate_inel;
         }
         else
         {
            capacity=0;
         }
```

```
            //capacity=capacity+needRate_inel;
            matrix_inelastic[min_pointer][0]=-1;
            matrix_inelastic[min_pointer][1]=-1;
            matrix_inelastic[min_pointer][2]=-1;
            matrix_inelastic[min_pointer][3]=-1;
            k=0;
            while(k<buffer.getLength()) //to define queue_length
            {
               pkt=check_and_cast<myPacket*>((cMessage*)(buffer.get(k)));
               if(pkt->getFlowId()==1 && pkt->getPosition()==min_pointer)
               {
                  pkt=check_and_cast<myPacket*>((cMessage*)buffer.remove(buffer.get(k)));
                  break;
               }
               k++;
            }
            if(HANDOVER==1)
            {
               delete pkt;
               inel_handover++;
               EV <<"Inelastic service in position "<<min_pointer<<" has done handover"<<
endl;
            }
            else
            {
               send(pkt,"out");
               inel_completed++;
               EV <<"Inelastic service in position "<<min_pointer<<" of matrix has
finished"<< endl;
            }
            inel_services--;
         }
         EV << "New capacity is: " <<capacity<<endl;
         //IMPORTANT IS TO UPDATE RATE ASSIGNED TO EACH ELASTIC SERVICE
AFTER EACH DEPARTURE
         if(el_services>0)
```

```
{
   tmp=(tot_capacity-needRate_inel*inel_services)/el_services;
   el_bitrate.collect(tmp);
}
else
{
   tmp=0;
}
EV << "INELASTIC MATRIX:"<<endl;
for(k=0;k<MAXinel;k++)
{
   if(matrix_inelastic[k][0]!=-1)
   {
      for(j=0;j<4;j++)
      {
         EV <<matrix_inelastic[k][j]<<" ";
      }
      EV<<endl;
   }
}
EV << "ELASTIC MATRIX:"<<endl;
for(k=0;k<MAXel;k++)
{
   if(matrix_elastic[k][0]!=-1)
   {
      matrix_elastic[k][2]=(matrix_elastic[k][0])/tmp;
      matrix_elastic[k][3]=tmp;
      for(j=0;j<4;j++)
      {
         EV <<matrix_elastic[k][j]<<" ";
      }
      EV<<endl;
   }
}
//exactly as done after an arrival, we have to collect statistics after a departure
```

```
        el_stats.collect(el_services);
        fraction_el.collect((((double)(tot_capacity-capacity-
(inel_services*needRate_inel)))/((double)(tot_capacity)))*100);
        inel_stats.collect(inel_services);

fraction_inel.collect((((double)(inel_services*needRate_inel))/((double)(tot_capacity)))*100);
        queue_load.collect(tot_capacity-capacity);
      }
      //set all indices to 0, and start again the search of the minimum time
      INEL=0;
      EL=0;
      HANDOVER=0;
      SERVICE=0;
      min=100.0;
      if(buffer.getLength()>0)
      {
        k=0;
        while(k<MAXinel)
        {
          if(matrix_inelastic[k][0]!=-1)
          {
            if(min>matrix_inelastic[k][1])// && dwell_time!=0)
            {
              min=matrix_inelastic[k][1];
              min_pointer=k;
              INEL=1;
              EL=0;
              HANDOVER=1;
              SERVICE=0;
            }
            if(min>matrix_inelastic[k][2])
            {
              min=matrix_inelastic[k][2];
              min_pointer=k;
              INEL=1;
              EL=0;
```

```
            HANDOVER=0;
            SERVICE=1;
          }
        }
        k++;
      }
      k=0;
      while(k<MAXel)
      {
        if(matrix_elastic[k][0]!=-1)
        {
          if(min>matrix_elastic[k][1])// && dwell_time!=0)
          {
            min=matrix_elastic[k][1];
            min_pointer=k;
            EL=1;
            INEL=0;
            HANDOVER=1;
            SERVICE=0;
          }
          if(min>matrix_elastic[k][2])
          {
            min=matrix_elastic[k][2];
            min_pointer=k;
            EL=1;
            INEL=0;
            HANDOVER=0;
            SERVICE=1;
          }
        }
        k++;
      }
      time_to_next_departure=(simtime_t)min;
      scheduleAt(simTime()+time_to_next_departure,DEPARTURE); //again, schedule for
min time
```

```
        }
    }
}

void server::finish()
{
    //free all the memory dynamically allocated for the matrices
    for (k=0;k<MAXel;k++)
    {
        delete [] matrix_elastic[k];
    }
    for (k=0;k<MAXinel;k++)
    {
        delete [] matrix_inelastic[k];
    }
    delete [] matrix_elastic;
    delete [] matrix_inelastic;
    //display on the screen the computed statistics
    EV << "LOSS_INELASTIC: "<< discarded_inel <<" .LOSS ELASTIC: " << discarded_el
<< endl;
    EV << "HANDOVER INELASTIC: " << inel_handover <<" .HANDOVER ELASTIC: "
<< el_handover << endl;
    EV << "COMPLETED INELASTIC SERVICES: " << inel_completed <<" .COMPLETED
ELASTIC SERVICES: " << el_completed << endl;
    EV << "AVERAGE INELASTIC SERVICES: " << inel_stats.getMean() <<" .AVERAGE
ELASTIC SERVICES: " << el_stats.getMean() << endl;
    EV << "AVERAGE INELASTIC FRACTION: " << fraction_inel.getMean() <<"
.AVERAGE ELASTIC FRACTION: " << fraction_el.getMean() << endl;
    EV << "AVERAGE QUEUE LOAD: " << queue_load.getMean() << endl;
    EV << "AVERAGE ELASTIC BIT RATE: " << el_bitrate.getMean() << endl;
    EV << "AVERAGE INELASTIC SERVICES IN CONT. TIME: " <<
inel_avg_serv/(simTime().dbl()) << endl;
    EV << "AVERAGE ELASTIC SERVICES IN CONT. TIME: " <<
el_avg_serv/(simTime().dbl()) << endl;
    EV << "AVERAGE INELASTIC FRACTION IN CONT. TIME: " <<
(inel_avg_fract/(simTime().dbl()))*100 << endl;
    EV << "AVERAGE ELASTIC FRACTION IN CONT. TIME: " <<
(el_avg_fract/(simTime().dbl()))*100 << endl;
```

```
   EV << "AVERAGE QUEUE LOAD IN CONT. TIME: " << avg_queue_load/(simTime().dbl()) << endl;
   EV << "AVERAGE ELASTIC BIT RATE IN CONT. TIME: " << avg_el_br/(simTime().dbl()) <<endl;


   for(k=0;k<MAXinel+1;k++)
   {
      for(j=0;j<MAXel+1;j++)
      {
         EV << "Probability of state ("<<k<<","<<j<<") is: "<<states[k][j]/simTime()<<endl;
      }
   }
   k=0;
   for (k=0;k<MAXinel+1;k++)
   {
      delete [] states[k];
   }
   delete [] states;



inel_loss.collect((double)discarded_inel/(double)(discarded_inel+inel_handover+inel_completed));
   el_loss.collect((double)discarded_el/((double)discarded_el+el_handover+el_completed));
   recordScalar("Inel_loss_probability",(inel_loss.getMean())*100);
   recordScalar("El_loss_probability",(el_loss.getMean())*100);
   recordScalar("Average_inel_services",inel_avg_serv/(simTime().dbl()));
   recordScalar("Average_el_services",el_avg_serv/(simTime().dbl()));
   recordScalar("Average_elastic_rate",avg_el_br/(simTime().dbl()));
}
```

First of all, the code creates a cQueue object, which is basically a container where the arriving packets will be stored.

The queue got a maximum capacity, which value is specified in the .INI file; the packets will be inserted in the queue if and only if there is enough room to serve them. In particular, the following check has to be done:

- If the packet is inelastic, then if the available capacity is larger than the rate needed by the inelastic, the customer can be served;
- If the packet is elastic, the situation is a little bit different, because the elastic rates change. So, when a elastic customer wants to enter the queue, it is necessary to check

if the capacity not occupied by the inelastics is enough to assign to each elastic customer at least the minimum needed rate, which written in formula is:

((tot_capacity-needRate_inel*inel_services)/el_services)>=needRate_el

Where:

- tot_capacity is the total capacity of the queue;
- needRate_inel is the rate asked by each inelastic;
- inel_services is the number of inelastics;
- el_services is the number of elastics;
- needRate_el is the min rate asked by elastics;

If the answer is yes, then the elastic customer can join the queue, and the elastics equally share the capacity not occupied by inelastics; instead, if it's not, the customer is discarded.

Now, in order to decide, time by time, which is the customer that finishes service or performs handover, I created 2 matrices, one for inelastic services, and one for elastic ones.

The inelastic matrix will be composed by MAXinel rows, where MAXinel is the maximum number of inelastic customers simultaneously allowed (specified in the .INI file), and 4 columns; alternatively, the matrix associated to the elastics is made of MAXel rows and 4 columns.

The idea is to assign a row of the matrix to each customer that enters the queue, so that there we can write key parameters of each service, in order to take decisions.

In particular, the matrices will contain:

- First column: number of bits of that customer that has to be transmitted yet;
- Second column: the remaining dwell time associated to that service;
- Third column: the time necessary to transmit the residual bits;
- Fourth column: for elastics, it's the elastic rate, while for inelastics is the current simulation time.

How these matrices work has been already described when I was presenting the simulation model.

Anyway, an important notification is that, during the declaration part, I do not know yet how many rows they have, because I will know only after I read the value from .INI parameters in the initialize() phase; that's why it is necessary to dynamically allocate memory for them, by doing:

```
double** matrix_elastic;
matrix_elastic=new double *[MAXel];
for(k=0;k<MAXel;k++)
  {
    matrix_elastic[k] = new double[4];
    for(j=0;j<4;j++)
```

```
    {
      matrix_elastic[k][j]=comparator[1,j];
    }
  }
```

The first row is to create a pointer to a pointer to a double; then, after the MAXel has be read as parameter, it is possible to create MAXel pointers to double rows. In the end, all the rows have been assigned the same value of comparator vector, which is just -1, so all rows are initialized to -1.

Now that matrices are ready, and all .INI parameters have been read and assigned to variables, then it is possible also to declare some statistics variable that will be used at the end to analyse some results. The statistics computed are:

- Number of inelastic and elastic customers in service;
- Fraction of the queue capacity, in percentage, reserved for inelastics/elastics over time;
- Queue load in bps;
- Elastic rate in bps;
- Probability of inelastic/elastic losses;

These statistics will be updated each time an event occurs.

In addition, a third matrix is needed, which refers to the state probabilities; it will have MAXinel rows and MAXel columns, so that each cell refers to a specific state of the queue. Each cell will contain the time spent in that specific state, and at the end of the simulation, that value divided by the simTime(), will return the probability that queue will be in that state.

Now that we have initialized everything, as soon as a message arrive to the module, it will enter the handleMessage part.

The first thing to do is to cast the message received from the class cMessage to myPacket; in this way, it is possible to check if this packet is inelastic or not, by doing:

pkt->getFlowId()

If this method returns 1, then it means the packet is inelastic, otherwise is elastic.

Once I know which type of customer is, then I can perform the corresponding check to see if there is enough capacity to serve it.

If yes, then I assign to the packet a size, which respects the parameter in the .INI, and I generate a value for the dwell time; so now, as soon as I found the first row different from -1 in the respective matrix, then I write the parameters related to that customer.

It has to be noted that a specific scenario must be taken into account; if the rate assigned to each elastic is 0, then I cannot compute the time to transmit residual bits, otherwise I would obtain infinity, which does not make any sense, so I decided to put a very big value which will change only as soon as the elastic rate becomes different from 0. This happened in the first scenario tested.

Apart from this special case, all the statistics are updated each time a new arrival or departure happens. This is performed thanks to the variable **previous_time**; basically, each time a new arrival or departure from the queue happens, then I save the current simulation time in that variable. In this way, when the next event (arrival/departure) happens, I can exactly compute how much time past from 2 consecutive events by performing:

simTime()-previous_time

Once I know this value, I can update all statistics by this amount.

In addition, also the value in the table, such as dwell times, completion times and bit residual must be updated by this quantity, in order to be always updated on the base of what happens. The only thing to notice is that I do not have to update the parameters of the packet just entered the queue, and so I used the variable **last_inserted**.

Each row in the matrices different from -1 will be updated by the quantity **simTime()-previous_time**, except for the row referring to the last_inserted; so, if last_inserted is 2, then the second row will no be updated. A tricky thing is that, for elastic, the last_inserted is row+100; this has be done in order to differentiate if the last arrived is elastic or not.

Basically, if the last arrival is inelastic, then last_inserted-100 will never match an elastic row, because it will be a negative value, and so all the elastic rows will be updated, as I want; on the contrary, if the last arrival is elastic, then last_inserted will be greater than 100, and so an inelastic row will never match its value, updating then all inelastic rows. Of course this technique works just if the MAXinel and MAXel are less than 100; if they are greater, it is sufficient to modify the last_inserted to row+MAX(MAXinel,MAXel) for elastic customers, and put in the if check of elastics just:

K!=last_inserted-MAX(MAXinel,MAXel)

The last parameter to explain is **pkt->setPosition(k)**; this parameter is useful to know the packet to which row of the matrix is linked to, and it will be used a lot during the departure phase of the code.

Of course as soon as each packet (inelastic or elastic) enters the queue, the rate associated to each elastic has to be updated, and as a consequence its statistics.

Instead, if the packet cannot be accepted in the queue, then the packet is discarded, and **last_inserted** is put equal to -1; in this way, nor elastic or elastic rows can ever be equal to it, meaning that all rows will be updated, as I wish.

Once that the matrices are updated time by time, then the next step is to look for the minimum time among all; in particular, we have to search for the minimum among all the second and third columns of the matrices, which represents, respectively, the dwell and completion time.

If the minimum is in the second column, then it means than a handover is performed, instead if it's the third, it means that service has been completed. In particular, in the code, at the begin the variable **min** contains a very big number, so that could not be the minimum; at each loop, if the current value is lower than the minimum, then it becomes the minimum.

In addition, I keep track of which is the rwo related to the minimum, by putting **min_pointer** equal to its row number; then, if the minimum is inelastic the variable **INEL** will be 1, otherwise the variable **EL** will assume the value 1, and the same criteria work for variables **HANDOVER** and **SERVICE**. In this way, we know all about the minimum.

Once the minimum has been discovered, then we have to schedule the self-message triggering the departure with a time equal to the minimum found before. It is important to notice that, if the self-message **DEPARTURE** is already scheduled, then has to be deleted, and sent again,

because it was being sent in the past, but now the situation of the times has changed, and so could not be the minimum anymore.

So, we now know, that if we receive a self-message is because a departure has to be performed. The first thing to do if we receive a self-message is to update all the statistics and parameter in the matrices by the time **time_to_next_departure**, which is the time needed to perform the departure of the customer.

Then, reading the values of the variable of **EL/INEL, HANDOVER/SERVICE** and **min_pointer**, I exactly know which is the row of the matrix containing the minimum, so I can put it all equal to -1, because the customer is releasing it, and modify the free capacity accordingly.

But now comes the problem, how do I know the row of the matrix to which packet of the queue is related to? This is possible thanks to the parameter **pkt->getPosition()**; it will returns the row to which the packet is referred to, and together with **pkt->getFlowId()**, we know to which matrix the packet refers to.

So, after I found the row to delete, I immediately know which is the packet related to it, and so I proceed eliminating it from the queue, and I update the elastic rate.

Now, the search of the minimum starts again, in the same exact way of before, creating then a cycle.

This loop will go on until the simulation time ends, and in that moment the code enter in the finish() part.

Here I just print the statistics computed, and I free the memory dynamically allocated for the matrices.

The last module of the network is the sink, but its code is pretty empty, it contains just print messages to notify that customer has finished correctly its service.

Once all the codes of the simulation part are ready, then the results are stored in variables, and displayed to the output, which is the screen; in addition, the results are copied in a Matlab code, such that it can draw the pictures of the curves in which we are interested to. The code is the following:

```
clc

clear all

close all


lambda=20;


lambdas=[20,30,40,50,70,90,100,250,500,750,1000]; %%lambda tested values

dwells=[50,100,300,500,1000]; %%dwell tested means

elsizes=[50,100,300,500,1000]; %%Kbit, el sizes

%%ALL THE VALUES ARE TAKEN FROM THE OUTPUT OF THE SIMULATOR

%%THE FOLLOWING VALUES ARE THE ONES OBTAINED FOR BOTH EXPONENTIAL DISTRIBUTIONS
```

prob_inel_losses_expexp=[0.158751,8.11781,23.1179,36.0461,53.3362,63.3447,66.8639,86.5591,93.2788,95.5092,96.623];

prob_el_losses_expexp=[4.4995,57.6006,75.7725,83.0226,89.1748,91.8913,92.8524,97.2847,98.6623,99.1147,99.3313];

avg_inel_serv_expexp=[16.5931,22.8566,25.4538,26.4331,27.1672,27.4397,27.5166,27.8459,27.9282,27.9529,27.9651];

avg_el_serv_expexp=[14.4993,29.1609,29.6654,29.7888,29.8767,29.9108,29.922,29.9719,29.9864,29.991,29.9932];

avg_el_rate_expexp=[760058,108839,79958.9,69677.6,62047.5,59202.4,58363.8,54938,54081.4,53826.4,53698.3];

%%THE FOLLOWING VALUES ARE THE ONES OBATINED FOR INEL SIZES EXP. AND EL SIZES DETERMINITSIC

prob_inel_losses_expdet=[0.221171,7.96953,23.3297,36.2199,53.5373,63.292,66.7811,86.5322,93.2519,95.5023,96.6335];

prob_el_losses_expdet=[5.86049,58.0394,76.2513,83.3236,89.388,92.044,92.915,97.3276,98.6833,99.129,99.3465];

avg_inel_serv_expdet=[16.5463,22.8937,25.4878,26.4608,27.1759,27.4383,27.5144,27.8451,27.9278,27.9529,27.9652];

avg_el_serv_expdet=[14.6156,29.1324,29.6686,29.7927,29.8789,29.9126,29.923,29.9724,29.9866,29.9912,29.9934];

avg_el_rate_expdet=[803304,108891,79621.3,69306.9,61875.7,59154.6,58368.2,54942.4,54084.2,53822.8,53696.2];

%%THE FOLLOWING VALUES ARE THE ONES OBATINED FOR DETERMINITSIC INEL SIZES AND EL SIZES EXP.

prob_inel_losses_detexp=[0.170448,8.30333,23.4284,36.2893,53.5643,63.3637,66.9309,86.6101,93.2792,95.5285,96.6351];

prob_el_losses_detexp=[4.01622,57.6598,76.032,83.1723,89.2497,91.9158,92.7981,97.2918,98.6577,99.112,99.3334];

avg_inel_serv_detexp=[16.6296,22.9057,25.5055,26.4608,27.1721,27.4399,27.5187,27.8459,27.9279,27.9532,27.9653];

avg_el_serv_detexp=[14.8185,29.1608,29.6686,29.793,29.8772,29.9109,29.9215,29.9718,29.9864,29.9908,29.9933];

avg_el_rate_detexp=[696794,108266,79383.9,69325,61903.1,59141.5,58332.6,54931.9,54081.4,53821.4,53694.8];

%%THE FOLLOWING VALUES ARE THE ONES OBTAINED FOR BOTH DETERMINISTIC DISTRIBUTIONS

prob_inel_losses_detdet=[0.166836,8.24783,23.4111,36.4536,53.6216,63.3982,66.9186,86.5989,93.2805,95.5279,96.6366];

prob_el_losses_detdet=[5.1116,58.1104,76.2324,83.2951,89.3658,92.0473,92.9281,97.3306,98.6836,99.1283,99.3455];

avg_inel_serv_detdet=[16.5891,22.9254,25.4866,26.4553,27.1764,27.4395,27.5188,27.8459,27.9276,27.9532,27.9652];

avg_el_serv_detdet=[14.5437,29.1338,29.6715,29.7939,29.8794,29.9127,29.9228,29.9723,29.9862,29.9909,29.9933];

avg_el_rate_detdet=[758500,108231,79512.2,69333.2,61850.3,59134.9,58313.5,54932.3,54084.5,53818.5,53694.6];

%%THESE VALUES ARE GENERATED USING DIFFERENT DWELL MEANS(Exponentials)

%%DWELL=1

prob_inel_losses_dwell_1=[0,0,0.000998379,0.0528922,3.26538,13.3865,19.3253,64.8232,82.1755,88.1321,91.069];

prob_el_losses_dwell_1=[0,0,0.000600402,0.126884,9.50951,26.6975,33.9176,73.5367,86.7314,91.1671,93.3568];

avg_inel_serv_dwell_1=[6.22024,9.35932,12.4893,15.6264,21.2326,24.3738,25.2036,27.482,27.7861,27.8663,27.9025];

avg_el_serv_dwell_1=[1.1208,3.1634,7.80786,14.2488,24.6347,27.5603,28.1419,29.6168,29.8431,29.9017,29.9283];

avg_el_rate_dwell_1=[3.02259e+6,2.55696e+6,1.20317e+6,465203,153471,98512.6,87172,59310.4,55778.1,54858.7,54444.6];

%%DWELL=50

prob_inel_losses_dwell_50=[0.116908,7.07251,21.8709,34.7472,52.3216,62.2904,65.7926,86.1916,93.0743,95.4012,96.526];

prob_el_losses_dwell_50=[2.21062,52.9985,72.8692,80.9341,87.7331,90.7934,91.7931,96.8817,98.4682,98.9859,99.2341];

avg_inel_serv_dwell_50=[16.1103,22.5443,25.3254,26.3565,27.1413,27.4127,27.4941,27.8409,27.9256,27.9517,27.964];

avg_el_serv_dwell_50=[11.7236,28.9856,29.6123,29.7597,29.8578,29.8975,29.91,29.9676,29.9844,29.9897,29.9922];

avg_el_rate_dwell_50=[962372,112901,81408.9,70516.6,62326.3,59482.3,58620.8,54995.5,54113.2,53838.9,53710.4];

%%DWELL=100

prob_inel_losses_dwell_100=[0.174989,7.54891,22.7054,35.583,52.9549,62.8662,66.4862,86.3796,93.1779,95.4571,96.5826];

prob_el_losses_dwell_100=[3.88943,55.8795,74.6173,82.2823,88.6473,91.4865,92.4109,97.1318,98.5763,99.0581,99.2963];

avg_inel_serv_dwell_100=[16.3795,22.7291,25.4118,26.4113,27.1538,27.4292,27.5063,27.8431,27.9268,27.9525,27.9647];

avg_el_serv_dwell_100=[13.5989,29.0957,29.6434,29.7805,29.8703,29.9059,29.9172,29.9703,29.9855,29.9905,29.9929];

avg_el_rate_dwell_100=[835508,110846,80432.4,69897.8,62152.5,59297.1,58473.6,54974.8,54097.6,53829.3,53702.2];

%%DWELL=500

prob_inel_losses_dwell_500=[0.206859,8.15638,23.4152,36.3079,53.4278,63.2881,66.8623,86.6148,93.2723,95.5166,96.6297];

prob_el_losses_dwell_500=[5.04061,57.7803,76.202,83.2646,89.3597,92.0015,92.891,97.3171,98.6839,99.131,99.342];

avg_inel_serv_dwell_500=[16.5981,22.8419,25.4749,26.4475,27.172,27.4363,27.5156,27.8458,27.9281,27.953,27.9651];

avg_el_serv_dwell_500=[14.8458,29.1676,29.6744,29.7917,29.8794,29.9118,29.9228,29.972,29.9865,29.9912,29.9933];

avg_el_rate_dwell_500=[747945,108961,79656.2,69475.7,61910.9,59193,58357.9,54933.7,54078.7,53823.2,53696.5];

%%DWELL=1000

prob_inel_losses_dwell_1000=[0.18047,8.25655,23.1788,36.4251,53.6497,63.3141,66.9813,86.6019,93.2945,95.5265,96.6442];

prob_el_losses_dwell_1000=[4.96425,58.1947,76.2991,83.3867,89.4025,92.0482,92.9705,97.3453,98.6938,99.1309,99.3492];

avg_inel_serv_dwell_1000=[16.6231,22.9439,25.4588,26.4563,27.1767,27.4376,27.5182,27.8464,27.9284,27.9533,27.9652];

avg_el_serv_dwell_1000=[15.0914,29.1942,29.6785,29.7961,29.8793,29.9125,29.9231,29.9725,29.9868,29.9912,29.9934];

avg_el_rate_dwell_1000=[725248,107648,79790.5,69383.3,61872.7,59181.4,58340.6,54932.9,54077.8,53819.2,53695.2];

%%THESE VALUES ARE GENERATED USING DIFFERENT ELASTIC SIZES(Exponentials)

%%SIZE=50Kbit

prob_inel_losses_elsize_50k=[0.0778869,5.30227,18.9371,32.2512,51.4062,62.4209,66.1881,86.5596,93.2848,95.5119,96.6227];

prob_el_losses_elsize_50k=[0,0,0.0113963,0.612801,14.8769,29.1261,35.2988,73.7046,86.9855,91.3932,93.5319];

avg_inel_serv_elsize_50k=[16.5099,23.6867,26.9271,28.1539,28.3283,28.0334,27.958,27.8775,27.9324,27.9541,27.9656];

avg_el_serv_elsize_50k=[0.121459,0.505049,1.75844,5.86812,21.5116,25.9516,26.9327,29.5308,29.8346,29.9011,29.9287];

avg_el_rate_elsize_50k=[473107,614982,605488,418656,78804.6,61794.1,60128.8,55453.8,54332.7,53977.4,53812.5];

%%SIZE=100Kbit

prob_inel_losses_elsize_100k=[0.0961099,5.04935,20.0239,35.1443,53.1928,63.2114,66.7987,86.5408,93.2634,95.5228,96.6239];

prob_el_losses_elsize_100k=[0,0.0279143,4.74655,22.6709,48.2322,60.9498,65.1002,86.7494,93.466,95.6784,96.7601];

avg_inel_serv_elsize_100k=[16.5204,23.5922,26.5293,26.9165,27.2886,27.4874,27.5512,27.8475,27.9282,27.9533,27.9651];

avg_el_serv_elsize_100k=[0.282483,2.1172,14.8965,25.4367,28.6401,29.2208,29.3554,29.8397,29.9289,29.9543,29.9663];

avg_el_rate_elsize_100k=[886764,898270,287651,84393.8,63558.9,60121.5,59178.8,55179.4,54191.9,53889.9,53749];

%%SIZE=300Kbit

prob_inel_losses_elsize_300k=[0.086808,7.705,23.2114,36.1588,53.4003,63.1968,66.8773,86.4909,93.2558,95.5162,96.6283];

prob_el_losses_elsize_300k=[0.0139834,29.8651,60.4388,72.061,82.2542,86.6714,88.1985,95.5316,97.8075,98.5479,98.9039];

avg_inel_serv_elsize_300k=[16.5224,22.9085,25.5057,26.4438,27.1687,27.436,27.5158,27.8448,27.9279,27.953,27.9652];

avg_el_serv_elsize_300k=[2.09607,26.3715,29.2629,29.5873,29.7782,29.8431,29.864,29.9529,29.9774,29.9851,29.9889];

avg_el_rate_elsize_300k=[1.73094e+6,140702,80843.4,70092.9,62222.5,59356.3,58498.2,54987.5,54102.3,53833.9,53706.2];

%%SIZE=1000Kbit

prob_inel_losses_elsize_1000k=[0.269063,8.07981,23.2315,36.2869,53.4017,63.2608,66.8225,86.5231,93.2904,95.4958,96.6237];

prob_el_losses_elsize_1000k=[48.3347,78.4213,87.7515,91.2924,94.5058,95.8596,96.2732,98.5992,99.3154,99.5444,99.6597];

avg_inel_serv_elsize_1000k=[16.4589,22.8652,25.4596,26.4611,27.1728,27.4356,27.5155,27.8452,27.9283,27.9529,27.9651];

avg_el_serv_elsize_1000k=[28.8268,29.7139,29.8559,29.9031,29.9407,29.9564,29.9606,29.9855,29.993,29.9954,29.9966];

avg_el_rate_elsize_1000k=[177359,106076,79281.1,69056.3,61801,59102.9,58290.5,54921,54072.1,53815.7,53691.9];

%%HERE I COMPUTE THE CONFIDENCE INTERVALS

%%FIRST OF ALL COMPUTE THE MEAN OF THE STATISTICS

mean_inel_losses=(prob_inel_losses_expexp+prob_inel_losses_expdet+prob_inel_losses_detexp+prob_inel_losses_detdet)/4;

mean_el_losses=(prob_el_losses_expexp+prob_el_losses_expdet+prob_el_losses_detexp+prob_el_losses_detdet)/4;

mean_inel_services=(avg_inel_serv_expexp+avg_inel_serv_expdet+avg_inel_serv_detexp+avg_inel_serv_detdet)/4;

mean_el_services=(avg_el_serv_expexp+avg_el_serv_expdet+avg_el_serv_detexp+avg_el_serv_detdet)/4;

mean_el_rate=(avg_el_rate_expexp+avg_el_rate_expdet+avg_el_rate_detexp+avg_el_rate_detdet)/4;

%%THEN COMPUTE THE SQUARED STANDARD DEVIATIONS

stdDev_inel_losses=(((prob_inel_losses_expexp-mean_inel_losses).^2)+((prob_inel_losses_expdet-mean_inel_losses).^2)+((prob_inel_losses_detexp-mean_inel_losses).^2)+((prob_inel_losses_detdet-mean_inel_losses).^2))/3;

stdDev_el_losses=(((prob_el_losses_expexp-mean_el_losses).^2)+((prob_el_losses_expdet-mean_el_losses).^2)+((prob_el_losses_detexp-mean_el_losses).^2)+((prob_el_losses_detdet-mean_el_losses).^2))/3;

stdDev_inel_services=(((avg_inel_serv_expexp-mean_inel_services).^2)+((avg_inel_serv_expdet-mean_inel_services).^2)+((avg_inel_serv_detexp-mean_inel_services).^2)+((avg_inel_serv_detdet-mean_inel_services).^2))/3;

stdDev_el_services=(((avg_el_serv_expexp-mean_el_services).^2)+((avg_el_serv_expdet-mean_el_services).^2)+((avg_el_serv_detexp-mean_el_services).^2)+((avg_el_serv_detdet-mean_el_services).^2))/3;

stdDev_el_rate=(((avg_el_rate_expexp-mean_el_rate).^2)+((avg_el_rate_expdet-mean_el_rate).^2)+((avg_el_rate_detexp-mean_el_rate).^2)+((avg_el_rate_detdet-mean_el_rate).^2))/3;

%%NOW TAKE THE VALUE OF THE T-STUDENT (t3,0.025) FROM TABLES

```
t_student=3.182446;

%%NOW COMPUTE THE CONFIDENTIAL INTERVALS, BOTH LOWER AND UPPER
BOUNDS

conf_inel_losses(1,:)=mean_inel_losses-t_student*((sqrt(stdDev_inel_losses))/(sqrt(4)));
conf_inel_losses(2,:)=mean_inel_losses+t_student*((sqrt(stdDev_inel_losses))/(sqrt(4)));
conf_el_losses(1,:)=mean_el_losses-t_student*((sqrt(stdDev_el_losses))/(sqrt(4)));
conf_el_losses(2,:)=mean_el_losses+t_student*((sqrt(stdDev_el_losses))/(sqrt(4)));
conf_inel_services(1,:)=mean_inel_services-
t_student*((sqrt(stdDev_inel_services))/(sqrt(4)));
conf_inel_services(2,:)=mean_inel_services+t_student*((sqrt(stdDev_inel_services))/(sqrt(4))
);
conf_el_services(1,:)=mean_el_services-t_student*((sqrt(stdDev_el_services))/(sqrt(4)));
conf_el_services(2,:)=mean_el_services+t_student*((sqrt(stdDev_el_services))/(sqrt(4)));
conf_el_rate(1,:)=mean_el_rate-t_student*((sqrt(stdDev_el_rate))/(sqrt(4)));
conf_el_rate(2,:)=mean_el_rate+t_student*((sqrt(stdDev_el_rate))/(sqrt(4)));

%%NOW LET'S TRY LOGNORMAL DISTRIBUTION

%%LOGNORMAL-EXPONENTIAL

prob_inel_losses_lnexp=[0.222478,6.44334,21.0564,33.81,52.137,61.7517,65.72,85.9467,93.
0195,95.3442,96.4982];
prob_el_losses_lnexp=[5.13969,55.581,75.0495,82.8036,89.1794,91.8043,92.8107,97.2985,9
8.6727,99.1098,99.3355];
avg_inel_serv_lnexp=[15.9543,22.3514,25.2217,26.3071,27.127,27.4009,27.4903,27.8378,27
.9254,27.9512,27.9637];
avg_el_serv_lnexp=[12.8114,29.0749,29.6512,29.7868,29.877,29.9098,29.9219,29.9718,29.9
865,29.9909,29.9932];
avg_el_rate_lnexp=[1.02253e+6,114750,82334.4,70920,62389.1,59542.6,58618.6,55020.5,54
110.6,53840.5,53711.1];

%%EXPONENTIAL-LOGNORMAL

prob_inel_losses_expln=[0.143514,8.08217,23.2202,36.2263,53.2827,63.315,66.8894,86.540
3,93.2667,95.5058,96.6292];
```

```
prob_el_losses_expln=[1.4878,53.4742,73.187,80.7999,87.688,90.7577,91.873,96.901,98.47
96,98.9915,99.237];

avg_inel_serv_expln=[16.573,22.9054,25.4655,26.4499,27.1701,27.4363,27.5183,27.8455,27
.9278,27.953,27.9652];

avg_el_serv_expln=[11.24,29.0263,29.6123,29.7546,29.8572,29.8965,29.9108,29.9677,29.98
46,29.9898,29.9922];

avg_el_rate_expln=[926861,109046,79987,69542.9,61983.2,59243.8,58378.3,54951.6,54086,
53823.8,53698];


%%LOGNORMAL-LOGNORMAL


prob_inel_losses_lnln=[0.159959,6.53374,20.4957,33.355,51.266,61.6179,65.3051,85.9553,9
3.0324,95.3079,96.4779];

prob_el_losses_lnln=[1.99827,51.502,72.297,80.6707,87.4605,90.6515,91.8564,96.891,98.46
18,98.9851,99.2366];

avg_inel_serv_lnln=[15.8996,22.4088,25.1692,26.2755,27.1048,27.3997,27.4831,27.8377,27
.9252,27.9509,27.9634];

avg_el_serv_lnln=[10.2968,28.9002,29.5895,29.7535,29.854,29.8954,29.91,29.9675,29.9843,
29.9897,29.9922];

avg_el_rate_lnln=[1.11646e+6,115509,83199.7,71324.8,62697.3,59613.6,58735.4,55025.9,5
4114.1,53846.8,53716];


%%FIGURE OF PROBABILITY OF INELASTIC LOSSES

figure(1)

plot(lambdas,prob_inel_losses_expexp)

hold on

plot(lambdas,prob_inel_losses_expexp,'*b')

hold on

plot(lambdas,prob_inel_losses_expdet,'r')

hold on

plot(lambdas,prob_inel_losses_expdet,'*r')

hold on

plot(lambdas,prob_inel_losses_detexp,'g')

hold on

plot(lambdas,prob_inel_losses_detexp,'*g')

hold on

plot(lambdas,prob_inel_losses_detdet,'k')

hold on
```

```
plot(lambdas,prob_inel_losses_detdet,'*k')
hold on
plot(lambdas,conf_inel_losses(1,:),'*y')
hold on
plot(lambdas,conf_inel_losses(2,:),'*y')
xlabel("Lambda")
ylabel("Losses [%]")
legend("Expexp","Expexp(real points)","Expdet","Expdet(real points)","Detexp","Detexp(real points)","Detdet","Detdet(real points)","Location","southeast")
title('Probability of inelastic losses')

%%FIGURE OF PROBABILITY OF ELASTIC LOSSES
figure(2)
plot(lambdas,prob_el_losses_expexp)
hold on
plot(lambdas,prob_el_losses_expexp,'*b')
hold on
plot(lambdas,prob_el_losses_expdet,'r')
hold on
plot(lambdas,prob_el_losses_expdet,'*r')
hold on
plot(lambdas,prob_el_losses_detexp,'g')
hold on
plot(lambdas,prob_el_losses_detexp,'*g')
hold on
plot(lambdas,prob_el_losses_detdet,'k')
hold on
plot(lambdas,prob_el_losses_detdet,'*k')
hold on
plot(lambdas,prob_el_losses_lnexp,'c')
xlabel("Lambda")
ylabel("Losses [%]")
legend("Expexp","Expexp(real points)","Expdet","Expdet(real points)","Detexp","Detexp(real points)","Detdet","Detdet(real points)","Location","southeast")
```

```
title('Probability of elastic losses')

%%FIGURE OF AVERAGE INELASTIC SERVICES
figure(3)
plot(lambdas,avg_inel_serv_expexp)
hold on
plot(lambdas,avg_inel_serv_expexp,'*b')
hold on
plot(lambdas,avg_inel_serv_expdet,'r')
hold on
plot(lambdas,avg_inel_serv_expdet,'*r')
hold on
plot(lambdas,avg_inel_serv_detexp,'g')
hold on
plot(lambdas,avg_inel_serv_detexp,'*g')
hold on
plot(lambdas,avg_inel_serv_detdet,'k')
hold on
plot(lambdas,avg_inel_serv_detdet,'*k')
hold on
plot(lambdas,avg_inel_serv_lnexp,'c')
xlabel("Lambda")
ylabel("Avg services")
legend("Expexp","Expexp(real points)","Expdet","Expdet(real points)","Detexp","Detexp(real points)","Detdet","Detdet(real points)","Location","southeast")
title('Average inelastic services')

%%FIGURE OF AVERAGE ELASTIC SERVICES
figure(4)
plot(lambdas,avg_el_serv_expexp)
hold on
plot(lambdas,avg_el_serv_expexp,'*b')
hold on
plot(lambdas,avg_el_serv_expdet,'r')
hold on
```

```
plot(lambdas,avg_el_serv_expdet,'*r')
hold on
plot(lambdas,avg_el_serv_detexp,'g')
hold on
plot(lambdas,avg_el_serv_detexp,'*g')
hold on
plot(lambdas,avg_el_serv_detdet,'k')
hold on
plot(lambdas,avg_el_serv_detdet,'*k')
hold on
plot(lambdas,avg_el_serv_lnexp,'c')
xlabel("Lambda")
ylabel("Avg services")
legend("Expexp","Expexp(real points)","Expdet","Expdet(real points)","Detexp","Detexp(real points)","Detdet","Detdet(real points)","Location","southeast")
title('Average elastic services')

%%FIGURE OF AVERAGE ELASTIC RATE
figure(5)
plot(lambdas,avg_el_rate_expexp)
hold on
plot(lambdas,avg_el_rate_expexp,'*b')
hold on
plot(lambdas,avg_el_rate_expdet,'r')
hold on
plot(lambdas,avg_el_rate_expdet,'*r')
hold on
plot(lambdas,avg_el_rate_detexp,'g')
hold on
plot(lambdas,avg_el_rate_detexp,'*g')
hold on
plot(lambdas,avg_el_rate_detdet,'k')
hold on
plot(lambdas,avg_el_rate_detdet,'*k')
hold on
```

```
plot(lambdas,avg_el_rate_lnexp,'c')
xlabel("Lambda")
ylabel("Avg elastic rate [b/s]")
legend("Expexp","Expexp(real points)","Expdet","Expdet(real points)","Detexp","Detexp(real points)","Detdet","Detdet(real points)","Location","northeast")
title('Average elastic rate')

%%FIGURE OF PROBABILITY OF INELASTIC LOSSES (CHANGING DWELL TIMES)
figure(6)
plot(lambdas,prob_inel_losses_expexp)
hold on
plot(lambdas,prob_inel_losses_dwell_50,'k')
hold on
plot(lambdas,prob_inel_losses_dwell_100,'r')
hold on
plot(lambdas,prob_inel_losses_dwell_500,'g')
hold on
plot(lambdas,prob_inel_losses_dwell_1000,'c')
xlabel("Lambda")
ylabel("Losses [%]")
legend("Expexp(dwell=300)","Expexp(dwell=50)","Expexp(dwell=100)","Expexp(dwell=500)","Expexp(dwell=1000)","Location","southeast")
title('Probability of inelastic losses')

%%FIGURE OF PROBABILITY OF ELASTIC LOSSES (CHANGING DWELL TIMES)
figure(7)
plot(lambdas,prob_el_losses_expexp)
hold on
plot(lambdas,prob_el_losses_dwell_50,'k')
hold on
plot(lambdas,prob_el_losses_dwell_100,'r')
hold on
plot(lambdas,prob_el_losses_dwell_500,'g')
hold on
plot(lambdas,prob_el_losses_dwell_1000,'c')
```

```
xlabel("Lambda")
ylabel("Losses [%]")
legend("Expexp(dwell=300)","Expexp(dwell=50)","Expexp(dwell=100)","Expexp(dwell=500)","Expexp(dwell=1000)","Location","southeast")
title('Probability of elastic losses')

%%FIGURE OF AVERAGE INELASTIC SERVICES (CHANGING DWELLS)
figure(8)
plot(lambdas,avg_inel_serv_expexp)
hold on
plot(lambdas,avg_inel_serv_dwell_50,'k')
hold on
plot(lambdas,avg_inel_serv_dwell_100,'r')
hold on
plot(lambdas,avg_inel_serv_dwell_500,'g')
hold on
plot(lambdas,avg_inel_serv_dwell_1000,'c')
xlabel("Lambda")
ylabel("Avg services")
legend("Expexp(dwell=300)","Expexp(dwell=50)","Expexp(dwell=100)","Expexp(dwell=500)","Expexp(dwell=1000)","Location","southeast")
title('Average inelastic services')

%%FIGURE OF AVERAGE ELASTIC SERVICES (CHANGING DWELLS)
figure(9)
plot(lambdas,avg_el_serv_expexp)
hold on
plot(lambdas,avg_el_serv_dwell_50,'k')
hold on
plot(lambdas,avg_el_serv_dwell_100,'r')
hold on
plot(lambdas,avg_el_serv_dwell_500,'g')
hold on
plot(lambdas,avg_el_serv_dwell_1000,'c')
xlabel("Lambda")
ylabel("Avg services")
```

```
legend("Expexp(dwell=300)","Expexp(dwell=50)","Expexp(dwell=100)","Expexp(dwell=500)","Expexp(dwell=1000)","Location","southeast")
title('Average elastic services')

%%FIGURE OF AVERAGE ELASTIC RATE (CHANGING DWELLS)
figure(10)
plot(lambdas,avg_el_rate_expexp)
hold on
plot(lambdas,avg_el_rate_dwell_50,'k')
hold on
plot(lambdas,avg_el_rate_dwell_100,'r')
hold on
plot(lambdas,avg_el_rate_dwell_500,'g')
hold on
plot(lambdas,avg_el_rate_dwell_1000,'c')
xlabel("Lambda")
ylabel("Avg elastic rate [b/s]")
legend("Expexp(dwell=300)","Expexp(dwell=50)","Expexp(dwell=100)","Expexp(dwell=500)","Expexp(dwell=1000)","Location","northeast")
title('Average elastic rate')

%%FIGURE OF PROBABILITY OF INELASTIC LOSSES (CHANGING EL SIZE)
figure(11)
plot(lambdas,prob_inel_losses_expexp)
hold on
plot(lambdas,prob_inel_losses_elsize_50k,'k')
hold on
plot(lambdas,prob_inel_losses_elsize_100k,'r')
hold on
plot(lambdas,prob_inel_losses_elsize_300k,'g')
hold on
plot(lambdas,prob_inel_losses_elsize_1000k,'c')
xlabel("Lambda")
ylabel("Losses [%]")
legend("Expexp(ElSize=500k)","Expexp(ElSize=50k)","Expexp(ElSize=100k)","Expexp(ElSize=300k)","Expexp(ElSize=1000k)","Location","southeast")
```

```
title('Probability of inelastic losses')

%%FIGURE OF PROBABILITY OF ELASTIC LOSSES (CHANGING EL SIZE)
figure(12)
plot(lambdas,prob_el_losses_expexp)
hold on
plot(lambdas,prob_el_losses_elsize_50k,'k')
hold on
plot(lambdas,prob_el_losses_elsize_100k,'r')
hold on
plot(lambdas,prob_el_losses_elsize_300k,'g')
hold on
plot(lambdas,prob_el_losses_elsize_1000k,'c')
xlabel("Lambda")
ylabel("Losses [%]")
legend("Expexp(ElSize=500k)","Expexp(ElSize=50k)","Expexp(ElSize=100k)","Expexp(ElSize=300k)","Expexp(ElSize=1000k)","Location","southeast")
title('Probability of elastic losses')

%%FIGURE OF AVERAGE INELASTIC SERVICES (CHANGING EL SIZE)
figure(13)
plot(lambdas,avg_inel_serv_expexp)
hold on
plot(lambdas,avg_inel_serv_elsize_50k,'k')
hold on
plot(lambdas,avg_inel_serv_elsize_100k,'r')
hold on
plot(lambdas,avg_inel_serv_elsize_300k,'g')
hold on
plot(lambdas,avg_inel_serv_elsize_1000k,'c')
xlabel("Lambda")
ylabel("Avg services")
legend("Expexp(ElSize=500k)","Expexp(ElSize=50k)","Expexp(ElSize=100k)","Expexp(ElSize=300k)","Expexp(ElSize=1000k)","Location","southeast")
title('Average inelastic services')
```

```
%%FIGURE OF AVERAGE ELASTIC SERVICES (CHANGING EL SIZE)
figure(14)
plot(lambdas,avg_el_serv_expexp)
hold on
plot(lambdas,avg_el_serv_elsize_50k,'k')
hold on
plot(lambdas,avg_el_serv_elsize_100k,'r')
hold on
plot(lambdas,avg_el_serv_elsize_300k,'g')
hold on
plot(lambdas,avg_el_serv_elsize_1000k,'c')
xlabel("Lambda")
ylabel("Avg services")
legend("Expexp(ElSize=500k)","Expexp(ElSize=50k)","Expexp(ElSize=100k)","Expexp(ElSize=300k)","Expexp(ElSize=1000k)","Location","southeast")
title('Average elastic services')

%%FIGURE OF AVERAGE ELASTIC RATE (CHANGING EL SIZE)
figure(15)
plot(lambdas,avg_el_rate_expexp)
hold on
plot(lambdas,avg_el_rate_elsize_50k,'k')
hold on
plot(lambdas,avg_el_rate_elsize_100k,'r')
hold on
plot(lambdas,avg_el_rate_elsize_300k,'g')
hold on
plot(lambdas,avg_el_rate_elsize_1000k,'c')
xlabel("Lambda")
ylabel("Avg elastic rate [b/s]")
legend("Expexp(ElSize=500k)","Expexp(ElSize=50k)","Expexp(ElSize=100k)","Expexp(ElSize=300k)","Expexp(ElSize=1000k)","Location","northeast")
title('Average elastic rate')

%%FIGURE OF PROBABILITY OF INELASTIC LOSSES (LOGNORMAL DISTRIBUTION)
```

```
figure(16)
plot(lambdas,prob_inel_losses_expexp,'b')
hold on
plot(lambdas,prob_inel_losses_lnexp,'k')
hold on
plot(lambdas,prob_inel_losses_expln,'r')
hold on
plot(lambdas,prob_inel_losses_lnln,'g')
xlabel("Lambda")
ylabel("Losses [%]")
legend("Expexp","Lnexp","Expln","Lnln")
title('Probability of inelastic losses')

%%FIGURE OF PROBABILITY OF ELASTIC LOSSES (LOGNORMAL DISTRIBUTION)
figure(17)
plot(lambdas,prob_el_losses_expexp,'b')
hold on
plot(lambdas,prob_el_losses_lnexp,'k')
hold on
plot(lambdas,prob_el_losses_expln,'r')
hold on
plot(lambdas,prob_el_losses_lnln,'g')
xlabel("Lambda")
ylabel("Losses [%]")
legend("Expexp","Lnexp","Expln","Lnln")
title('Probability of elastic losses')

%%FIGURE OF AVERGARE INELASTIC SERVICES (LOGNORMAL DISTRIBUTION)
figure(18)
plot(lambdas,avg_inel_serv_expexp,'b')
hold on
plot(lambdas,avg_inel_serv_lnexp,'k')
hold on
plot(lambdas,avg_inel_serv_expln,'r')
```

```
hold on
plot(lambdas,avg_inel_serv_lnln,'g')
xlabel("Lambda")
ylabel("Avg services")
legend("Expexp","Lnexp","Expln","Lnln")
title('Average inelastic services')

%%FIGURE OF AVERGARE ELASTIC SERVICES (LOGNORMAL DISTRIBUTION)
figure(19)
plot(lambdas,avg_el_serv_expexp,'b')
hold on
plot(lambdas,avg_el_serv_lnexp,'k')
hold on
plot(lambdas,avg_el_serv_expln,'r')
hold on
plot(lambdas,avg_el_serv_lnln,'g')
xlabel("Lambda")
ylabel("Avg services")
legend("Expexp","Lnexp","Expln","Lnln")
title('Average elastic services')

%%FIGURE OF AVERGARE ELASTIC RATE (LOGNORMAL DISTRIBUTION)
figure(20)
plot(lambdas,avg_el_rate_expexp,'b')
hold on
plot(lambdas,avg_el_rate_lnexp,'k')
hold on
plot(lambdas,avg_el_rate_expln,'r')
hold on
plot(lambdas,avg_el_rate_lnln,'g')
xlabel("Lambda")
ylabel("Avg elastic rate [b/s]")
legend("Expexp","Lnexp","Expln","Lnln")
title('Average elastic rate')
```

```
%%FIGURE OF PROBABILITY OF INELASTIC LOSSES (DWELL OF 1 SEC)
figure(21)
plot(lambdas,prob_inel_losses_expexp)
hold on
plot(lambdas,prob_inel_losses_dwell_1,'r')
xlabel("Lambda")
ylabel("Losses [%]")
legend("Dwell = 300 secs","Dwell = 1 sec","Location","southeast")
title('Probability of inelastic losses')

%%FIGURE OF PROBABILITY OF ELASTIC LOSSES (DWELL OF 1 SEC)
figure(22)
plot(lambdas,prob_el_losses_expexp)
hold on
plot(lambdas,prob_el_losses_dwell_1,'r')
xlabel("Lambda")
ylabel("Losses [%]")
legend("Dwell = 300 secs","Dwell = 1 sec","Location","southeast")
title('Probability of elastic losses')

%%FIGURE OF AVERAGE INELASTIC SERVICES (DWELL OF 1 SEC)
figure(23)
plot(lambdas,avg_inel_serv_expexp)
hold on
plot(lambdas,avg_inel_serv_dwell_1,'r')
xlabel("Lambda")
ylabel("Avg services")
legend("Dwell = 300 secs","Dwell = 1 sec","Location","southeast")
title('Average inelastic services')

%%FIGURE OF AVERAGE ELASTIC SERVICES (DWELL OF 1 SEC)
figure(24)
plot(lambdas,avg_el_serv_expexp)
hold on
plot(lambdas,avg_el_serv_dwell_1,'r')
```

```
xlabel("Lambda")
ylabel("Avg services")
legend("Dwell = 300 secs","Dwell = 1 sec","Location","southeast")
title('Average elastic services')

%%FIGURE OF AVERAGE ELASTIC RATE (DWELL OF 1 SEC)
figure(25)
plot(lambdas,avg_el_rate_expexp)
hold on
plot(lambdas,avg_el_rate_dwell_1,'r')
xlabel("Lambda")
ylabel("Avg elastic rate [b/s]")
legend("Dwell = 300 secs","Dwell = 1 sec")
title('Average elastic rate')

error_inel_losses=(conf_inel_losses(2,:)-conf_inel_losses(1,:))./mean_inel_losses;
error_el_losses=(conf_el_losses(2,:)-conf_el_losses(1,:))./mean_el_losses;
error_inel_services=(conf_inel_services(2,:)-conf_inel_services(1,:))./mean_inel_services;
error_el_services=(conf_el_services(2,:)-conf_el_services(1,:))./mean_el_services;
error_el_rate=(conf_el_rate(2,:)-conf_el_rate(1,:))./mean_el_rate;
```

6. Conclusions

As I explained in the introduction, the main scope of this thesis is to understand how a base station deals with different type of services, inelastic and elastic ones.

Thanks to the first scenario tested, it turned out that if the generation process is too slow with respect to the service time, the system will be almost always empty, meaning that our cell is underutilized. Then, it can never happen that we got elastic losses, since there will never be more than 10 elastic services in the queue; while, on the contrary, losses happen for inelastics because there cannot be 2 or more of them in the cell. In addition, since most of the time there will be just one customer in the queue, its service time will be very short, and so it is very difficult that it performs handover before finishing the service.

So, from the case 4.1. we got that if the customer arrival process is too slow, it is good in terms of completed services and losses, but not in terms of utilization of the network, because we are using just a little part of it, and if we assume that it is pretty expensive to install a 5G base station somewhere, it will be a waste of money to have such a great capacity that cannot be used.

That's why we moved into the second tested case increasing the arrival rate; there is, of course, an increase of the losses, but at the same time an increase of the completed services, especially for what concern the elastic ones. We can say, that elastics benefit of this increase in the speed of customer arrivals, because there can be up to 10 of them in the queue, so having more customers means more completed services, while, on the contrary, for inelastic ones the situation get worse. This because there can be just 1 of them in the cell, and so if we are increasing the rate of the arrivals, it means we are increasing the probability of having more than 1 inelastic in the queue, resulting in an inelastic loss, in fact there are much more inelastic losses than elastic ones.

In addition, since there are so many entering customers, the utilization of the queue increases a lot, meaning that we are better exploiting its resources.

But, in order to get into some more realistic scenario, we need to increase the maximum number of customers (both inelastic and elastic), in order to simulate a base station of a crowded place.

In the case 4.3 we tested a maximum number of inelastic and elastic customer equal to 30, with an arrival rate equal to the case 4.1. It can be noticed that inelastic losses reduce, due to the possibility of having up to 30 inelastic customers in the queue, instead of just one, while the elastic ones are increasing. Of course, handovers increase, since having more customers in the queue means more probability that one of them moves into another cell before finishing its service.

From now on, the following scenarios start from the same parameters of the case 4.3, changing each time a different parameter, in order to see how it affects the statistics computed for different values of the arrival rate.

In the case 4.4, we tested different distributions, in particular the exponential and the deterministic one, for the packet sizes, and, as a consequence changes also the service time distributions.

It can be noticed from the output figures that different distributions do not affect the statistics computed, and it is interesting to notice it. Our queue is a mix of two different queues: a

M/M/m/0 which describes the behaviour of inelastic customers, and a M/M/1-PS which describes the behaviour of the elastic ones.

Both of them are insensitive to the service distributions, and our queue seems to be insensitive to it too.

The same conclusion can be deducted from the case 4.5, in which we test a high variance distribution for the packet size, a lognormal distribution, and the statistics do not change again.

The next parameter to be studied has been the dwell time; in particular, in the cases 4.6 and 4.7. In the case 4.6, the dwell time does not seem to change anything, in fact the curves for different values of it are overlapping, meaning that no changes arise. In reality, this is due to the choice of the mean of the dwell time; in this specific scenario it was too big compared with the average service time. In this case, due to the particular choice of parameters, it turned out that there were almost always few customers in the queue, meaning high bit rate for each one of them, and so low service times.

Since, on average, the service was much smaller than the dwell time, it does not matter how small the latter is, because handover would be performed few times.

In order to see some changes, the dwell time has to be of the same order of magnitude of the average service time, and that's why the case 4.7 has been studied.

It is possible to see that, in this case, the choice of the dwell changes a lot the statistics, and in particular, lower dwell time means less queue load on average, because customers exit the queue much more times.

Most of the changes happen for smaller values of the arrival rate, that's because as soon as the arrival rate becomes bigger and bigger, the fact that one customer stays in the queue or performs handover does not change the overall situation, because there will be a new arriving customer taking its place.

As a consequence, especially for smaller arrival rates, we can notice less losses, because, on average, there will be less customers in the queue.

So, in order to notice some change, the dwell time has to be of the same order of magnitude of the service time.

The last tested case, instead, shows us how the statistics vary as the elastic size increases. The inelastic statistics do not change so much, while the elastic losses increase as the packet size increases, and the same happens for the average number of elastic customers in the queue.

In conclusion, we can claim that not all the changes affect the statistics, in particular our queue turns out to be insensitive to the service distribution, while the dwell time is relevant when is less, or at least of the same order of magnitude of the service time; it means that if a customer is moving slower than the service speed, then it will finish the service regardless of how slow it is.

Finally, changing the elastic packet sizes affects the elastic statistics, especially when we have low arrival rate; the changes are negligible when the customer arrival is high because the queue will fill anyway, so the behaviour is more or less the same.