UNIVERSITÀ DI PISA

COMPUTER ENGINEERING

Performance Evaluation of Computer Systems and Networks

WORKGROUP PROJECT DOCUMENTATION

Aerocom System:

An Omnet++ Simulation

Students

Giovanni Menghini

Andrea Gerratana

Tommaso Giorgi

Academic Year 2020/2021

INDEX

1 DESIGN

1.1 BRIEF INTRODUCTION

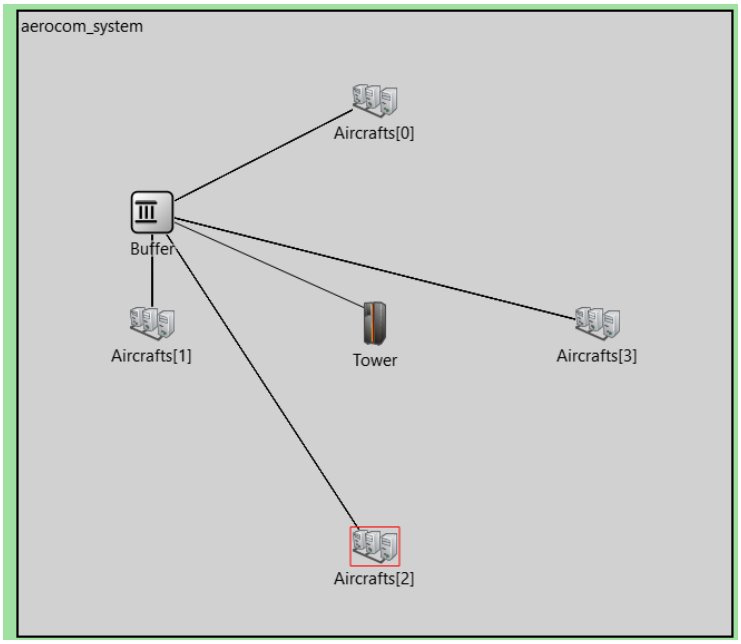


Figure 1. Aerocom System

Our Aerocom System is composed by a variable number of Aircrafts, the Buffer and the Control Tower (Figure 1). Each Aircraft communicates with the control tower generating one package of a fixed size every *k* seconds, the communication goes through a Buffer that is in charge of handling all the packets sent to the control tower. Aircrafts have five datalinks available for communication with the buffer (and so with the control tower), only one datalink can be used at a time as *serving datalink.* Each datalink has a time-varying capacity, in fact every *t* seconds a new target capacity is selected and the datalink's capacity will linearly change from the current one to the target one. The target capacity is reached after *t* seconds. Before performing each transmission, the aircraft can select a new datalink as serving DL. A new datalink is chosen if the capacity between transmission i-2 and i-1 decreased by more than *X*%, this action is called **DL Selection** and takes *S* seconds. The new datalink is selected as the one with the highest capacity at the current time. We calculated different simulations results depending on different values of *X* and *S*, also using a variable number of aircrafts (four and then eight aircrafts) (Check Paragraph **Results** for additional information)

1.2 PRELIMINARY CHOICES

- We chose to represent the Aircraft as a compound module composed by five datalinks and one controller. We decided to use the simple module Controller to manage the DL Selection, it is in charge of generating packets every k seconds and choosing the right datalink after the DL Selection.
- We also decided to use a Buffer in order to store multiple messages directed to the control tower. Only one message at a time can be served by the control tower, so it is important to store the other messages in a queue, these are ready to be sent to the control tower when it will become available to serve a new message.

- We used a channel with variable capacity for transmitting packets to the control tower. The channel's capacity is strictly related to datalinks that are in charge of sending packets. The higher is channel's capacity, the faster packets will be sent (a packet sent through a channel with lower capacity would be slower than a packet sent in a channel with higher capacity). This happen because packets have fixed size.
- The datalink's capacity will increase or decrease linearly until the target capacity is reached, we decided to implement this dividing the time *t* (time needed to reach the target capacity) by a coefficient. The coefficient is a constant value that can be configured through the ini file. The coefficient is just a counter of increment/decrement steps to make in order to reach the target capacity.
- We introduced a serving time for the control tower, it is configurable from the omnetpp.ini file too. For simplicity we set a constant serving time of 50ms.
- We introduced a serving time (1 ms) for the controller too, because it could happen to have two messages that want to be served at same time (during the message queued realising) and both of them cannot enter the channel, there would be an error. In order to avoid this, the controller will serve one message and lock the resource for 1 ms so that the second message in queue can find the channel busy.

1.3 SET UP OF THE RANDOM VARIABLES

1.3.1 Time *t*

The time *t* is the time needed to generate a new target capacity of a specific Datalink. The datalink's capacity will increase or decrease linearly until the target capacity will be reached. It will happen in *t* seconds. During our simulations, we select two scenarios:

- *t* is a random variable generated from an exponential distribution.
- *t* is a random variable generated from a lognormal distribution.

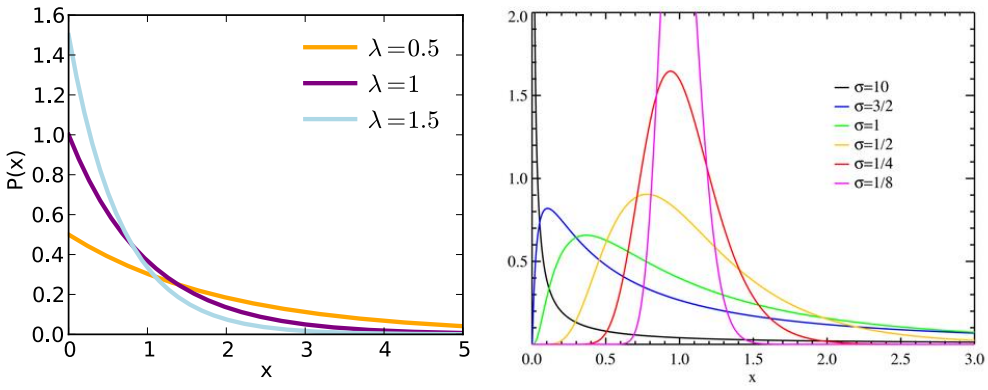


Figure 2. Exponential & Lognormal Distribution

In particular, we decided to use the omnetpp::exponential function for the exponential distribution:

```
simtime_t DataLink::tGenerator(){
    return exponential(0.5,1);

}
```

Figure 3. Random t generation, exponential distribution

This function takes two parameters as arguments, the first one is the *mean(m)* of the exponential distribution and the second one is the random number generator. The *lambda(λ)* can be easily obtained by the mean, in fact it is just $\frac{1}{m}$. We decided to use the random number generator "1" for the generation of *t* in the Datalink, in order to do so we had to include the total number of random generators (in our case, three) and the random generators itself in the omnetpp.ini file.

Otherwise for the lognormal distribution we decided to use the omnetpp::lognormal function:

```
simtime_t DataLink::tGenerator(){
    return lognormal(0.5,0.5,1);

}
```

Figure 4. Random t generation, lognormal distribution

This function takes three parameters as arguments:

- The mean *(m)*
- The standard deviation *(σ)*
- The Random Number Generator *(RnG)*

We used the same mean as the exponential distribution, then we selected a standard deviation of 0.5. We decided to use 0.5 to not get exaggerated peaks in the distribution. We also used the same random number generator as before, "1".

1.3.2 Time S

S is the time needed to complete the DL Selection. We used different values of *S* in different simulations, then we compared all the results to check how *S* influenced them.

The variable *S* can be configured through the omnetpp.ini configuration file. If we select a high value for S, the dl selection will last longer. This time must be balanced with the average time to send a packet through the channel, in fact if we select a very high value of *S* we completely nullify the advantage of a higher channel's capacity.

1.3.3 Time k

The time *k* is the time needed to generate a new packet. We used the same function that we explained before for the *t* variable. The random variable generated from the exponential distribution is got from the omnetpp::exponential function and the lognormal one is obtained from the omnetpp::lognormal function. We used "0" as random number generator for the value *k*.

1.3.4 Value X

The value *X,* like the previous value *S,* is fixed and configured for each simulation in the configuration file. It is used as a threshold for the DL Selection. If the capacity between transmission i-2 and i-1 decreased by more than *X*%, a DL Selection will occur. Using very high values of *X* will cause extremely rare DL Selections, so we preferred to use values lower than 40 for the *X* value.

2 IMPLEMENTATION

2.1 AIRCRAFT

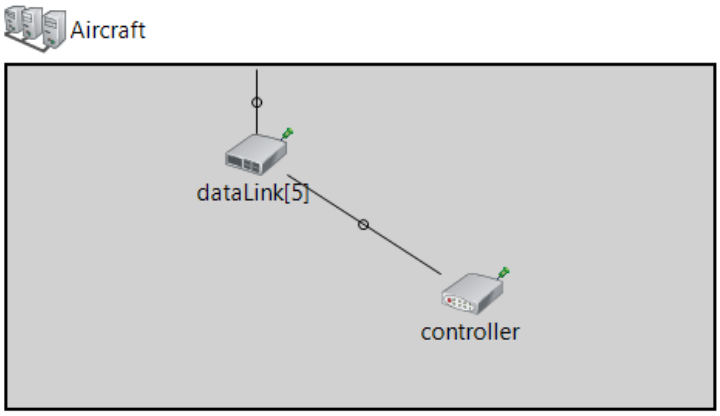


Figure 5. Aircraft Ned Structure

The aircraft is a compound module (Figure 5) composed by:

- Five datalinks
- One controller

The Aircraft gates are connected to each Datalinks gate. They are used to communicate with the Buffer and then with the Control Tower.

2.1.1 Datalink

Figure 6. DataLink Ned Icon

The datalink is a simple module connected to the Controller and to the extern of the Aircraft, this module is mainly characterized by the capacity value. It is a double varying between 1 and 100 that keeps changing through the simulation. In particular, every *t* seconds *(t is the random variable described before)* a new target capacity is selected, and the actual capacity will linearly change until reaching the new one, it will take *t* seconds.

At the extern of the Aircraft there is a channel that links the Aircraft to the Buffer, channel's datarate changes with the capacity: it is equal to the capacity multiplied by a coefficient called `datarateMultiplier` used to make transmission faster. The multiplier can be set in the ini file.

The new target capacity is generated (Figure 7) using the function *uniform(1,100,2).* So, the new capacity is a double selected from the interval [1,100) and it must be different from the actual capacity. The third argument is the RnG's value.

```
double DataLink::generateDifferentCapacity(double capacity){
    double differentCapacity;
    do
    {
        differentCapacity=uniform(1, 100, 2);
    }
    while(differentCapacity == capacity);
    return differentCapacity;
}
```

Figure 7. Capacity Generation Function

The new capacity is reached linearly, to do so we decided to use self-messages called "*update capacity*". These messages are sent every $\frac{t}{coefficient}$ seconds and when the datalink receives one of them the capacity is increased or decreased by $\frac{targetCapacity - currentCapacity}{coefficient}$. At the same time also the capacity of the channel is updated. The coefficient is configurable from the initialization file.

```
void DataLink::handleMessage(cMessage *msg)
{
    //if it is a self message change parameter capacity until target (if greater add, if lower subtract)
    if(strcmp(msg->getName(),"update capacity")==0)
    {
        cDatarateChannel *channel = check_and_cast<cDatarateChannel*>(gate("portToO$o")->getTransmissionChannel());
        cancelAndDelete(msg);
        double actualCapacity = par("capacity");
        //changing data rate using capacity
        actualCapacity = actualCapacity + (this->targetCapacity - this->oldCapacity)/this->coefficient;
        par("capacity").setDoubleValue(actualCapacity);
        channel->setDatarate(actualCapacity*this->datarateMultiplier);
        if( abs(this->targetCapacity - actualCapacity) <= 0.000000001)
        {
            this->targetCapacity = generateDifferentCapacity(actualCapacity);
            this->oldCapacity = actualCapacity;
            this->t = tGenerator();
        }
        sendUpdate(this->t/this->coefficient);
        return;
    }

    if(strcmp(msg->getName(),"to control tower")==0)
    {
        send(msg, "portToO$o");
    }
}
```

Figure 8. DataLink's handleMessage function

2.1.2 Controller

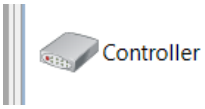


Figure 9. Controller Ned Icon

The Controller is the Aircraft's core entity, it is directly connected with five Datalinks and it generates messages every *k* seconds (*k* is the random variable described before). The new message, called "*to control tower*", embeds an attribute containing the aircraft index, which is a unique identifier for the aircraft. It has a queue for handling messages that cannot be sent.

When a "*to control tower*" message is scheduled (Figure 10), in order to be transmitted, there are **four conditions** to be checked:

1. The channel needs to be free (the channel is considered busy if a packet is already using the channel).
2. The resources need to be unlocked (because two messages cannot be transmitted at same time).

3. A DL selection, started before this message arrival, needs to be finished.
4. The queue of previous messages needs to be empty.

If one of these conditions is not satisfied, the message enters the queue and, if the reason why the message entered the queue was because the channel was busy in a transmission, a new message named *"serve next message*" will be scheduled at the end of the transmission, so that the controller can start to free the queue.

```
if(strcmp(msg->getName(),"to control tower")==0)
{
    createNewMessage(kGenerator(),"to control tower");
    cChannel *channel = getParentModule()->gate("portToO$o",this->servingDL)->getTransmissionChannel();
    // control if the system is processing a dl selection or queue is not empty or channel busy
    if(this->dlSelectionProcessing || !this->queue->isEmpty() || channel->isBusy() || this->lock)
    {
        this->queue->insert(msg);
        if(channel->isBusy())
        {
            cMessage *msg2 = new cMessage("serve next message");
            scheduleAt(channel->getTransmissionFinishTime(), msg2);
        }
    }
    else
    {
        //updating capacities last transmissions
        this->queue->insert(msg);
        this->messageToServe =check_and_cast<AerocomMessage*>(this->queue->pop());
        controllerSend(this->capacityI, this->capacityI1, this->capacityI2, this->servingDL, this->X,
                this->S, this->dlSelectionProcessing, this->messageToServe, this->timeToServe, this->lock);
    }
}
```

Figure 10. Handling "to control tower" message

Instead, if all conditions are satisfied, the controllerSend function will be called (Figure 10/11). This function checks the capacity of the two previous transmissions and starts the DL selection scheduling a message called *"dl selection end*" after *S* seconds if the capacity decreased more than *X*% , otherwise it will send the message through the serving DL, it will lock the resource, it will schedule the "*unlock*" message *timeToServe* seconds later (1 ms) and it will set the priority of this message to 1 (because the "*unlock*" message must be the last handled for messages arriving at same time), these last three operations are made in order to be sure about not serving two messages at same time.

```
void Controller::controllerSend(double &capacityI, double &capacityI1, double &capacityI2, int &servingDL, double X,
        simtime_t S, bool &dlSelectionProcessing, AerocomMessage* messageToServe, simtime_t timeToServe, bool &lock){
    capacityI2 = capacityI1;
    capacityI1 = capacityI;
    //less than 0 first transmission
    if((((capacityI2 - capacityI1)*100 / capacityI2) > X) || (capacityI1 < 0))
    {
       dlSelectionProcessing = true;
       cMessage *msg = new cMessage("dl selection end");
       scheduleAt(simTime()+S, msg);
    }
    else
    {
       updateCapacityServingDL(capacityI, servingDL);
       send(messageToServe, "portToDL$o", servingDL);
       // it may happen that it tries to send 2 message simultaneously because of the serve next message
       // so we need to lock
       lock = true;
       cMessage *msg2 = new cMessage("unlock");
       msg2->setSchedulingPriority(1);
       scheduleAt(simTime()+timeToServe,msg2);
    }
}
```

Figure 21 controllerSend function

When a message named *"serve next message"* arrives (Figure 12) the first three conditions listed before are checked again, the fourth one has changed: now there is the need to have a message queued. If these are satisfied, the controller will call the controllerSend function in order to serve the earlier arrived packet that

is still in the queue. Otherwise it will do nothing or scheduling a new “serve next message” (if the channel was busy) at the end of transmission.

```
if(strcmp(msg->getName(),"serve next message")==0)
{
    if(this->queue->isEmpty())
    {
        cancelAndDelete(msg);
        return;
    }
    // control if last transmission ended
    cChannel *channel = getParentModule()->gate("portToO$o",this->servingDL)->getTransmissionChannel();
    if(channel->isBusy() || this->lock || this->dlSelectionProcessing)
    {
        if(channel->isBusy())
        {
            scheduleAt(channel->getTransmissionFinishTime(), msg);
        }
        else
        {
            cancelAndDelete(msg);
        }
        return;
    }
    cancelAndDelete(msg);
    this->messageToServe = check_and_cast<AerocomMessage*>(this->queue->pop());
    controllerSend(this->capacityI, this->capacityI1, this->capacityI2, this->servingDL, this->X,
                   this->S, this->dlSelectionProcessing, this->messageToServe, this->timeToServe, this->lock);
    return;
}
```

Figure 32 handling “serve next message” packet

There are the last two types of message that the controller can handle, the first one is the “*dl selection end*” message (Figure 13). When it arrives the “*to control tower*” packet which called for the DL selection will be sent and the controller will make all the operations to lock and then unlock the resource for sending (they were explained in the controllerSend part).

```
//if received a message of end dl selection
if(strcmp(msg->getName(),"dl selection end")==0){
    dlSelection(this->capacityI, this->servingDL);
    this->dlSelectionProcessing = false;
    send(this->messageToServe, "portToDL$o", this->servingDL);
    this->lock = true;
    cMessage *msg2 = new cMessage("unlock");
    msg2->setSchedulingPriority(1);
    scheduleAt(simTime()+this->timeToServe,msg2);
    cancelAndDelete(msg);
    return;
}
```

Figure 13. handling “dl selection end” message

The second one is the “*unlock*” message (Figure 14). When it arrives, the controller will free the resource for sending (note that the lock is different from checking if channel is busy) and if any message is present in queue, a new “serve next message” packet is scheduled after timeToServe seconds (1 ms).

```
if(strcmp(msg->getName(),"unlock")==0)
{
    cancelAndDelete(msg);
    this->lock = false;
    if(!this->queue->isEmpty())
    {
        cMessage *msg2 = new cMessage("serve next message");
        scheduleAt(simTime()+this->timeToServe, msg2);
    }
}
```

Figure 14. handling “unlock” message

2.2 BUFFER

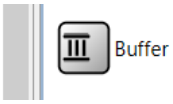


Figure 15. Buffer Ned Icon

The buffer is a simple module, connected to every aircraft and to the control tower, all packets sent from the aircrafts are first delivered to it. They are inserted in a queue waiting for being delivered to the control tower, but if the tower is not busy a packet can be sent immediately. We use the FCFS algorithm to schedule the packets in the queue: the first packet that arrives to the buffer is the first one to be sent to the control tower.

Each message received from the control tower has the label *"served"*. When the buffer receives that message means that the control tower is not busy anymore and a new packet can be sent and processed. The buffer has a private field called `busy` that is used to check if the control tower is free or not when a new message arrives.

```
void Buffer::handleMessage(cMessage *msg)
{
    // if to control tower insert in queue
    if(strcmp(msg->getName(),"to control tower")==0)
    {
        //insert in queue until served
        this->queue->insert(msg);
        if(!this->busy){
            msg = check_and_cast<AerocomMessage*>(this->queue->pop());
            send(msg, "portToO2$o");
            this->busy = true;
        }
    }
    if(strcmp(msg->getName(),"served")==0)
    {
        cancelAndDelete(msg);
        if(!(this->queue->isEmpty()))
        {
            AerocomMessage *msg2 = check_and_cast<AerocomMessage*>(this->queue->pop());
            send(msg2,"portToO2$o");
        }
        else
        {
            this->busy = false;
        }

    }
```

Figure 16. Buffer's handleMessage function

2.3 CONTROL TOWER

Figure 17. Control Tower Ned Icon

The control tower is a simple module; it is the destination of our packets. It has an input-output connection with the buffer. The control tower will take a packet from the buffer and it will process it, this operation will last 50ms (time to serve of the control tower that can be configured from the omnetpp.ini file). During the processing of the packet, the control tower is considered busy and cannot accept packets. When this

operation ends, the control tower will send a message to the buffer with the label *"served"* to let it know that it is ready again to serve a new packet.

```
void ControlTower::handleMessage(cMessage *msg)
{
    if(strcmp(msg->getName(),"to control tower")==0)
    {
        //serve a message
        msg->setName("served");
        scheduleAt(simTime()+this->timeToServe,msg);
    }
    else
    {
        send(msg,"portToO$o");
    }
}
```

Figure 18. ControlTower's handleMessage function

When the control tower receives a message, it checks if the message is a scheduled message ("served" label) or not ("*to control tower*" label). If the message is a packet from one of the aircrafts, it will change the message label into "served" and it will schedule this packet at simTime() + timeToServe (50ms) in order to tell the buffer that the processing of the packet is complete and that it is now ready to process a new packet.

3 RESULTS

We decided to do multiple simulations, in order to check how *X* and *S* influenced our simulation results. We evaluate these two different scenarios:

- Four aircrafts
- Eight aircrafts

For each scenario we tested different values of *X* and *S*, in particular:

- X = 10 and S = 1ms
- X = 10 and S = 100ms
- X = 40 and S = 1ms
- X = 40 and S = 100ms

We used these fixed values for each simulation:

- Coefficient = 10
- datarateMultiplier = 5000
- Time to serve of the Control Tower = 50ms
- Time to serve of the Aircraft = 1ms
- Simulation Time Limit = 1500s

3.1 COLLECTING THE DATA

We used the Omnet++ `cOutVector` to gather all the data of our simulations. The cOutVector records the simulation time in a particular part in the code of our simulation, this simulation time will be converted into a double in order to compute all the results required.

We need the response time of every packet sent by the aircrafts. We used these vectors in our simulation:

```
void Buffer::initialize()
{
    this->queue=new cQueue("myQ");
    this->busy = false;
    this->recordsIN_Aircraft0.setName("Buffer In[0]");
    this->recordsENDQ_Aircraft0.setName("Buffer EndQ[0]");
    this->recordsSERV_Aircraft0.setName("Buffer Serv[0]");
    this->recordsIN_Aircraft1.setName("Buffer In[1]");
    this->recordsENDQ_Aircraft1.setName("Buffer EndQ[1]");
    this->recordsSERV_Aircraft1.setName("Buffer Serv[1]");
    this->recordsIN_Aircraft2.setName("Buffer In[2]");
    this->recordsENDQ_Aircraft2.setName("Buffer EndQ[2]");
    this->recordsSERV_Aircraft2.setName("Buffer Serv[2]");
    this->recordsIN_Aircraft3.setName("Buffer In[3]");
    this->recordsENDQ_Aircraft3.setName("Buffer EndQ[3]");
    this->recordsSERV_Aircraft3.setName("Buffer Serv[3]");
}
```

```
void Controller::initialize()
{
    //retrieving constants
    this->S = getParentModule()->par("S");
    this->X = getParentModule()->par("X");
    this->timeToServe = getParentModule()->par("timeToServe");
    this->lock = false;
    this->capacityI = -1;
    this->capacityI1 = -1;
    this->capacityI2 = -1;
    this->dlSelectionProcessing = false;
    this->servingDL = 0;
    this->queue = new cQueue("msgQ");
    this->recordsIN.setName("Aircraft In");
    this->recordsENDQ.setName("Aircraft EndQ");
    this->recordsSERV.setName("Aircraft Serv");
    createNewMessage( kGenerator(), "to control tower");

}
```

Figure 19. cOutVector declarations

1) **In the Aircraft** (so in the Controller module)
    - We want to record when the packet is generated. We used the cOutVector called *"Aircraft In"* (Figure 19).
    - We want to know when a packet is withdrawn from the queue of the aircraft, the cOutVector called "*Aircraft EndQ*" will record this time (Figure 19).
    - Finally, we are interested in knowing when a message is served. A message is served in the aircraft when it will be sent through the channel. The cOutVector that will record this time is called "*Aircraft Serv*" (Figure 19)
2) **In the Buffer** we must use a different triplet of vectors for every aircraft. In fact, the first packet that will reach the buffer is not necessarily the first packet generated in our simulation. But knowing who that packet belongs to (the aircraft that generated it), we are sure that it is the first packet generated by this aircraft. So we include the following switch-case statement before every recording (Figure 20)
    - We decided to record when the packet reached the buffer (*"Buffer In"* vector)
    - We recorded when the packet leaved the buffer's queue in order to be sent to the control tower (*"Buffer EndQ"* vector)
    - Finally, we want to know when the packet has been served by the control tower ("*Buffer Serv"* vector)

```
switch(aerocom_msg2->getAircraftIndex()){
case 0:
    this->recordsSERV_Aircraft0.record(SimTime().dbl());
    break;
case 1:
    this->recordsSERV_Aircraft1.record(SimTime().dbl());
    break;
case 2:
    this->recordsSERV_Aircraft2.record(SimTime().dbl());
    break;
case 3:
    this->recordsSERV_Aircraft3.record(SimTime().dbl());
    break;
}
```

Figure 20. Switch-Case Statement

3.2 PROCESSING RAW DATA

All the data recorded using the vectors can be exported directly from Omnet++ to a csv Spreadsheet. This file is "Raw" because we must extract from it only the useful information and compute some formulas in order to get the results that we want to compute. We created an *Excel Macro* to get all the results automatically, first of all it calculates all the response times using the vectors described before, the response time is the difference between the simulation time when a packet is served by the control tower and the time when the packet was generated. **We know that every statistic taken from a sample is itself a random variable** so we needed to quantify its accuracy using a confidence interval in order to compare results. We decided to choose a 95% confidence interval and for doing this **our sample must be IID** (independent and identically distributed), so with the Excel Macro we computed the sample autocorrelation function of the mean response time:

$$\overline{R}(j) = \frac{1}{n}\sum_{i=1}^{n-j}\frac{(X_i - \overline{X})\cdot(X_{i+j} - \overline{X})}{S^2}$$

Figure 21. Sample Autocorrelation Function

We also plotted the correlogram to check if our sample is IID, it is IID if the dots are inside the threshold lines with 95% confidence, if it is not we needed to subsample our results and check the IID-ness again. The confidence threshold is equal to $\frac{z_{0.025}}{\sqrt{n}}$ $(z_{0.025} = 1.96)$. In the following pictures we can see all the correlograms used to prove the IID-ness:

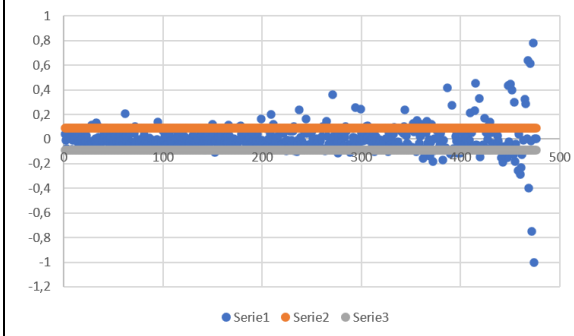
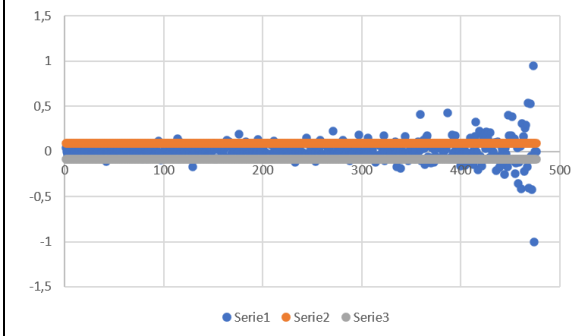
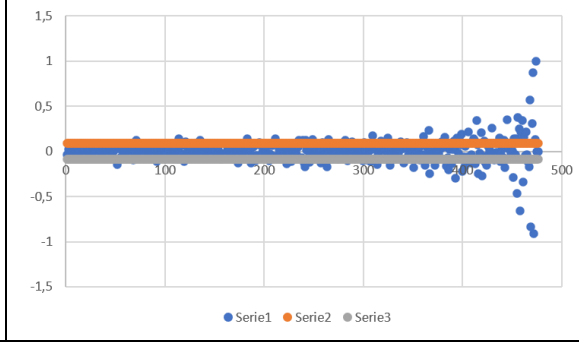
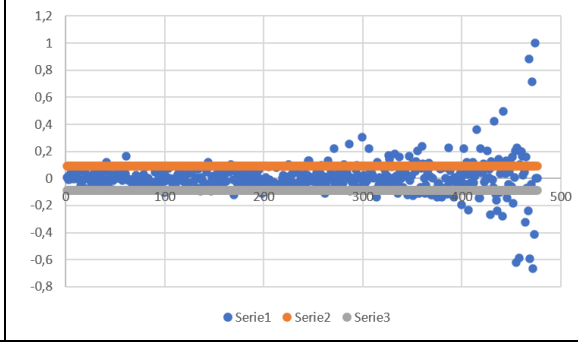
	X=10	*X=40*
S=1		
S=100		

Table 1. Four Aircraft - Exponential Distribution

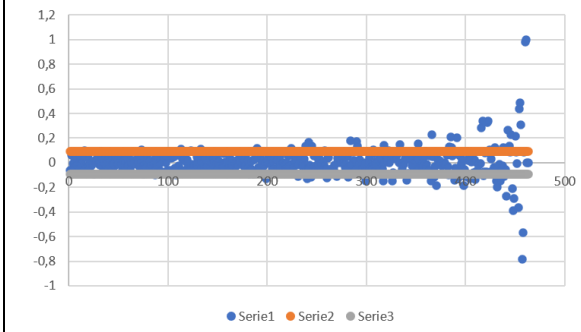
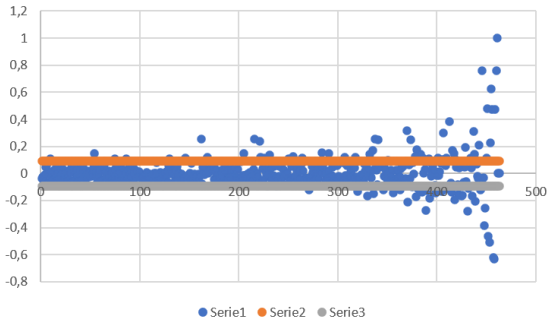
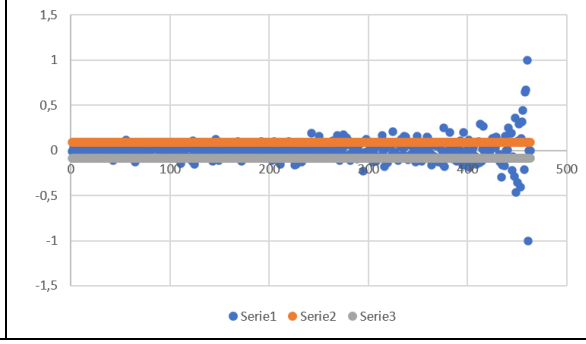
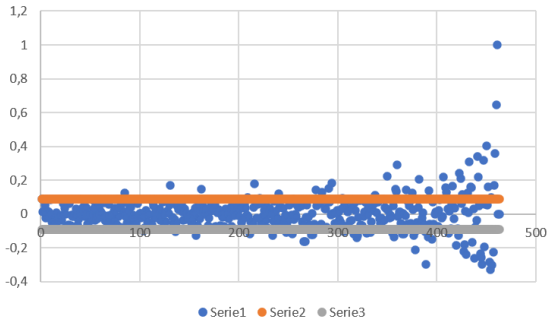
	X=10	X=40
S=1		
S=100		

Table 2. Four Aircraft - Lognormal Distribution

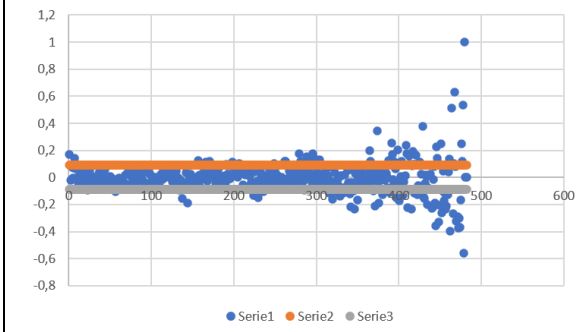
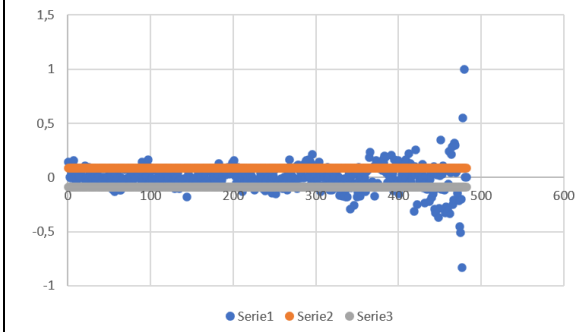
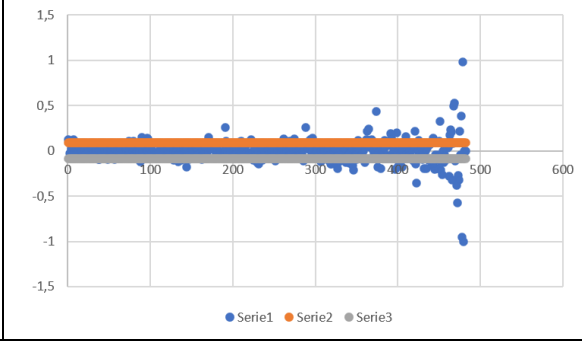
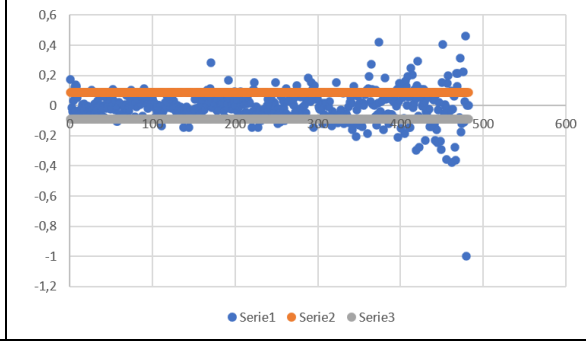
	X=10	X=40
S=1		
S=100		

Table 3. Eight Aircraft - Exponential Distribution

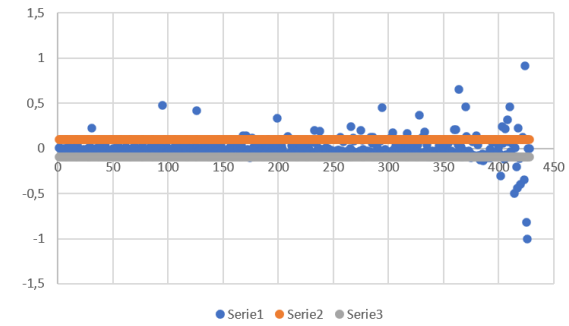
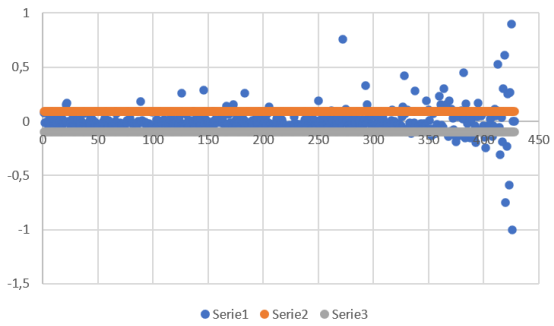
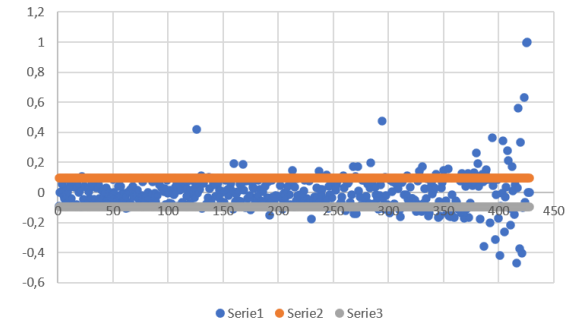
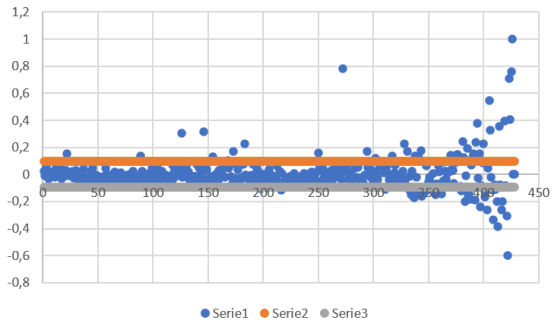
	X=10	X=40
S=1		
S=100		

Table 4. Eight Aircraft - Lognormal Distribution

As we can see only few points are above the thresholds so we can say that **they are IID with 95% confidence**. Once we know that we can compute the confidence interval in this way:

$$[\bar{X} - \frac{S}{\sqrt{n}} * z_{0.025}, \bar{X} + \frac{S}{\sqrt{n}} * z_{0.025}]$$

Figure 21. Confidence Interval Formula (95%)

3.3 COMPARING RESULTS

After all these computations we can compare the results plotting the mean response time with their confidence interval.

3.3.1 Four Aircrafts

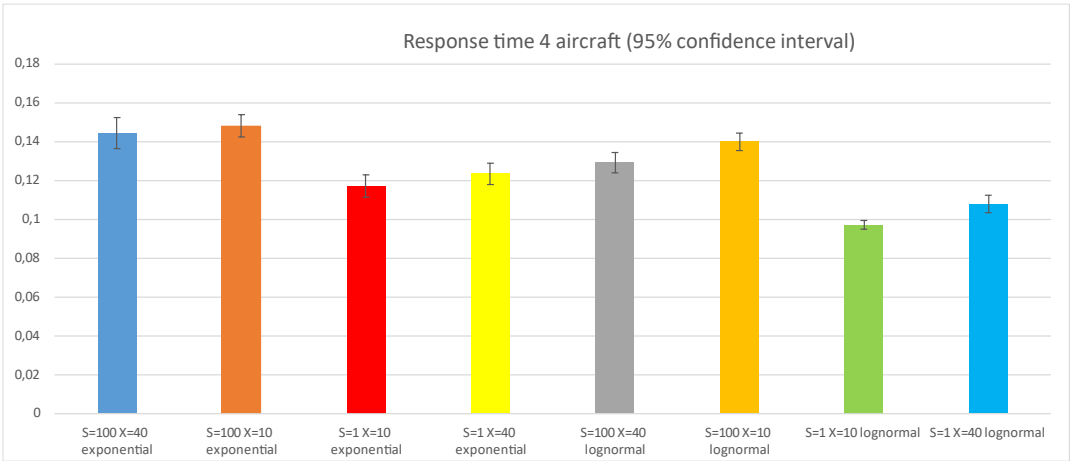


Figure 22. Response Time Comparison - Four Aircrafts

The four bins to the left of the graph are referred to the exponential distribution, instead the other bins are related to the lognormal one. Checking only the exponential distribution we can see that we have a major reduction of the response time with the *S=1* experiment, this is pretty obvious because the time required for the DL Selection (*S*) is very small compared to the one used in the other experiment. But it is important also to see that fixing the S value we probably have better performance with frequent DL Selections and so with small values of *X*. This happen because the advantage given by a potential better channel (with a bigger capacity than the previous one) surpasses the time needed for the DL selection and overall, we get better results. This consideration is verified in the lognormal distribution with *S=1*, instead for the exponential we are not sure if this will be true for every simulation because the two confidence intervals are overlapped.

3.3.2 Eight Aircrafts

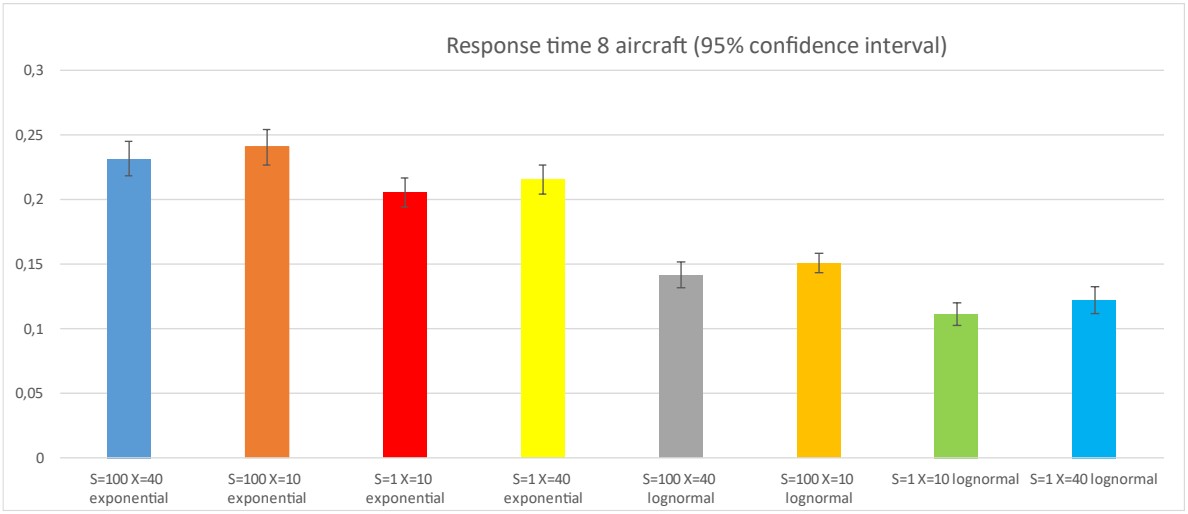


Figure 23. Response Time Comparison - Eight Aircrafts

Similar results are obtained with the eight aircraft experiment, in addition we can see that all the lognormal mean response times are very low compared to the exponential ones. As we will see in the 2^k factorial analysis in the next paragraph, the type of distribution is the main influence in our eight aircraft experiment. This occurs because with eight aircrafts we have lots of packets and the queue time is a real issue, so with the lognormal distribution packets are generated slower than in the exponential distribution and the risk of very long queue is reduced.

3.4 2^k FACTORIAL ANALYSIS

Last but not least, we wanted to see how the factors (X, S and the type of distribution) influence the mean response time. In order to do so we made a 2^k factorial analysis, we have k=3 factors described below:

$$x_A = \begin{cases} -1, & S = 1 \\ +1, & S = 100 \end{cases} \qquad x_B = \begin{cases} -1, & X = 10 \\ +1, & X = 40 \end{cases} \qquad x_C = \begin{cases} -1, & Lognormal\ Distribution \\ +1, & Exponential\ Distribution \end{cases}$$

3.4.1 Four Aircrafts Experiment

In the following table we calculated the mean response time for each experiment done with four aircrafts. These values will be used in the 2^k factorial table to compute the remaining results.

	S=1		**S=100**	
	Lognormal	*Exponential*	*Lognormal*	*Exponential*
X=10	96,835514	117,246567	139,900115	148,015133
X=40	107,725134	123,443036	129,176612	144,224047

Table 5. Mean Response Time comparison - 4 Aircrafts

	I	**A**	**B**	**C**	**AB**	**AC**	**BC**	**ABC**	**Y**
	1	-1	-1	-1	1	1	1	-1	96,835514
	1	1	-1	-1	-1	-1	1	1	139,900115
	1	-1	1	-1	-1	1	-1	1	107,725134
	1	1	1	-1	1	-1	-1	-1	129,176612
	1	-1	-1	1	1	-1	-1	1	117,246567
	1	1	-1	1	-1	1	-1	-1	148,015133
	1	-1	1	1	-1	-1	1	-1	123,443036
	1	1	1	1	1	1	1	1	144,224047
Sum	1006,566	116,0657	2,5715	59,29144	-31,60	-12,96	2,2392	11,62	
Mean(qi)	125,8207	14,50820	0,3214	7,411426	-3,950	-1,620	0,2799	1,453	
$\mathbf{8 * qi^2}$		1683,905	0,8265	439,4339	124,8	21,01	0,6267	16,89	2287,528
Variation		74%	0%	19%	5%	1%	0%	1%	

Table 6. 2^k factorial analysis - four Aircrafts

In this table the meaningful results are given by the **Variation** row. This row tells us how each factor influenced our simulation. In particular we have a very high percentage (*74%)* for the A column (*S* value), this means that if we want better results and a lower mean response time our interest is keeping the *S* value as low as possible. Instead the *X* value (column B) has a very low impact on the final results, for this reason if we want to give it a major importance at the end of the simulation, we probably need to increase the range of the datalink's capacities so that we can actually see an advantage from choosing another datalink with a better capacity. Then the type of distribution influenced our simulation with a percentage of *19%.* In fact, with a exponential distribution, packets are generated faster than in the lognormal one and so more of them will flow from the aircrafts to the control tower in our simulation. This will cause an increase of the queue time in the buffer or controller and so a bigger mean response time. This is a very small value compared to the one that we will obtain for the eight aircrafts experiment. Finally the last four percentages obtained from the combination of the previous factors are negligible compared to the percentages discussed before, probably the AB combination can increase if we successed in making the X factor more influent in the simulation.

3.4.2 Eight Aircrafts Experiment

The mean response time table for every experiment done on eight aircrafts is the following:

	S=1		S=100	
	Lognormal	*Exponential*	*Lognormal*	*Exponential*
X=10	111,497909	205,31366	150,85894	240,57987
X=40	122,197889	215,636383	141,493295	231,436159

Table 7. Mean Response Time Comparison - Eight Aircrafts

With these values we can easily compute the 2^k factorial table with the factors decribed before.

	I	**A**	**B**	**C**	**AB**	**AC**	**BC**	**ABC**	**Y**
	1	-1	-1	-1	1	1	1	-1	111,4979
	1	1	-1	-1	-1	-1	1	1	150,8589
	1	-1	1	-1	-1	1	-1	1	122,1978
	1	1	1	-1	1	-1	-1	-1	141,4932
	1	-1	-1	1	1	-1	-1	1	205,3136
	1	1	-1	1	-1	1	-1	-1	240,5799
	1	-1	1	1	-1	-1	1	-1	215,6363
	1	1	1	1	1	1	1	1	231,4361
Sum	1419,014	109,7224	2,513347	366,918	-39,53	-7,590	-0,155	0,5991	
Mean(qi)	177,3768	13,7153	0,314168	45,86475	-4,941	-0,948	-0,019	0,0748	
$\mathbf{8 * qi^2}$		1504,876	0,789614	16828,61	195,34	7,201	0,003	0,0448	18536,87
Variation		8%	0%	91%	1%	0%	0%	0%	

Table 8. 2^k factorial analysis - Eight Aircrafts

The final results are slightly different from the ones obtained with four aircraft. In this experiment the influence of *S* is lower than before because with eight aircraft the number of packets is doubled and so the type of distribution (exponential or lognormal) prevailed. In fact, with a very high number of packets, the queue time during which the packet is waiting to be served is the most significant part in the computation of the mean response time. A solution to this problem may be for example decreasing the service time of the control tower from 50ms to a lower value, or in addition to serve more packets simultaneously. This will significantly reduce the queue time of every packets giving us faster response times and it will make the other factors more influent during the simulation. The percentage obtained from the *X* factor is very low also in this experiment, the same solution decribed in the previous paragraph can be used also in this experiment.