HALMSTAD UNIVERSITY

Lab 1

DT4015 - Data Communications

Assignment

Part 1: Getting Started with OMNET++ (20%)

In this part of the lab, you will setup and run a simple network simulation using OMNET++. For this purpose, you can use the **virtual machine** that you will find in this **link** which requires you to have VirtualBox in your computer (https://halmstaduniversity.box.com/s/my3fo1sypazwhm1eq2nbqabe0hgrie3m), or you can follow the instructions in the official website for OMNET++ (https://omnetpp.org/). For simplification, this document follows the virtual machine method.

After the installation, you will perform two main tasks: 1) setting up a simple network to exchange packets, and 2) collect and summarize statistics from the simulation. During the lab session, you will follow a simplified, step-by-step tutorial on how to perform these tasks. For extra information, or for an in-depth tutorial on this exercise, feel free to visit the official OMNET++ TicToc Tutorial in https://doc.omnetpp.org/omnetpp4/tictoc-tutorial/index.html.

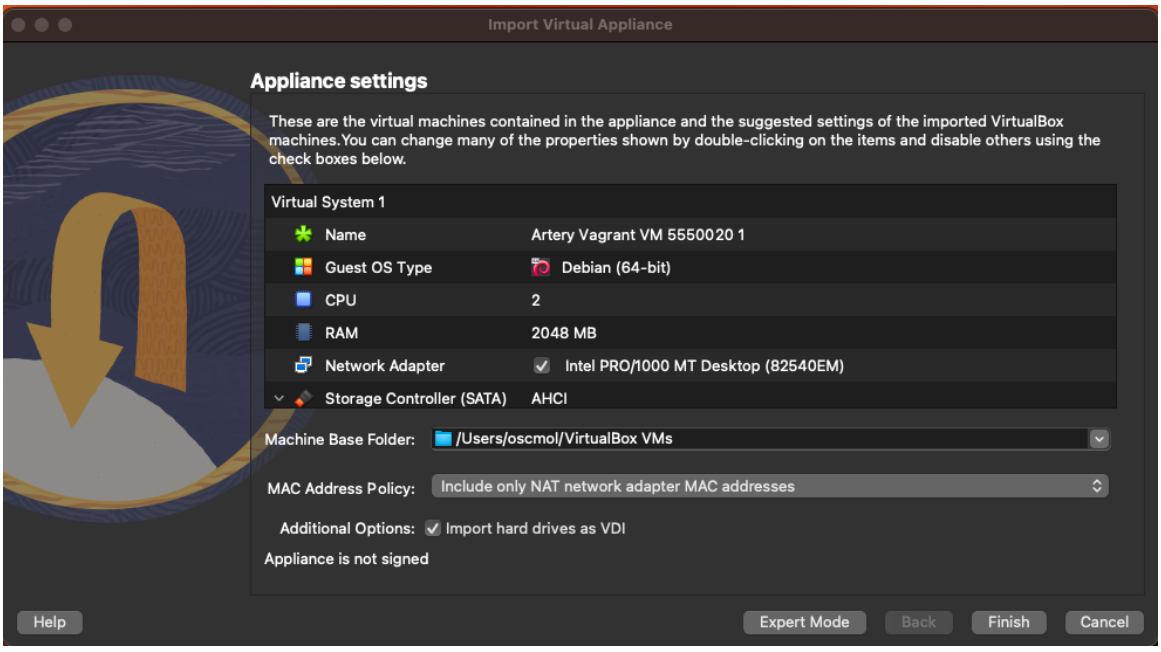
Milestones

To be awarded the points for this part, you will show that you were able to perform the following tasks:

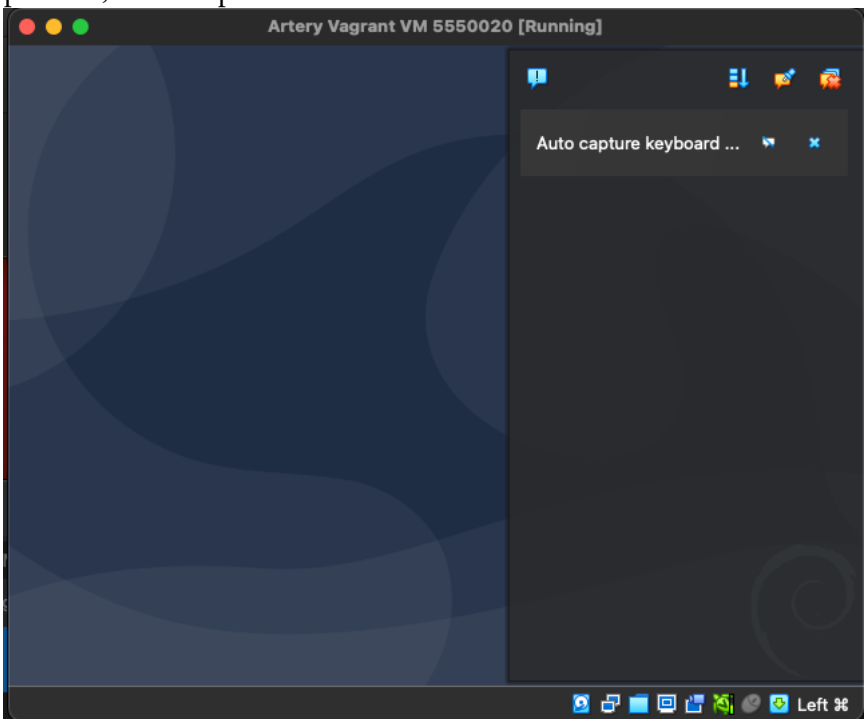
1. Running a 2-node TicToc (10%)
2. Statistics for Transmitted and Received Packets (10%)

Activity 1: Executing OMNET++

1. Make sure you have all the pre-requisite software installed. You will need:
    a. VirtualBox up and running (it might require you to activate virtualization capabilities in your computer)
    b. The Artery Vagrant VM appliance that you can find on this **link**.
    c. If you are planning on running OMNET++ outside of your virtual box, follow the guide in the official web page.
2. Open the .ova file containing the virtual appliance to be imported. Click on "Finish".



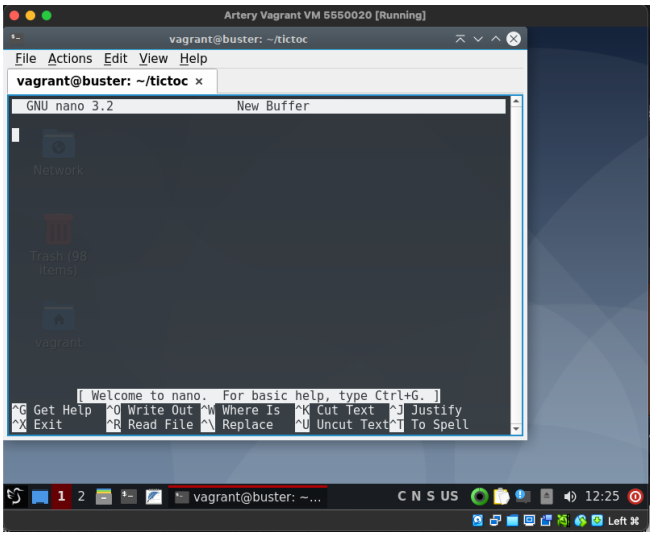
3. After importing. Start the virtual machine by double clicking on it. Depending on your computer's configuration, you might need to accept or enable mouse/keyboard capture (which will impede you from pointing/typing outside the machine). If possible, avoid capture.



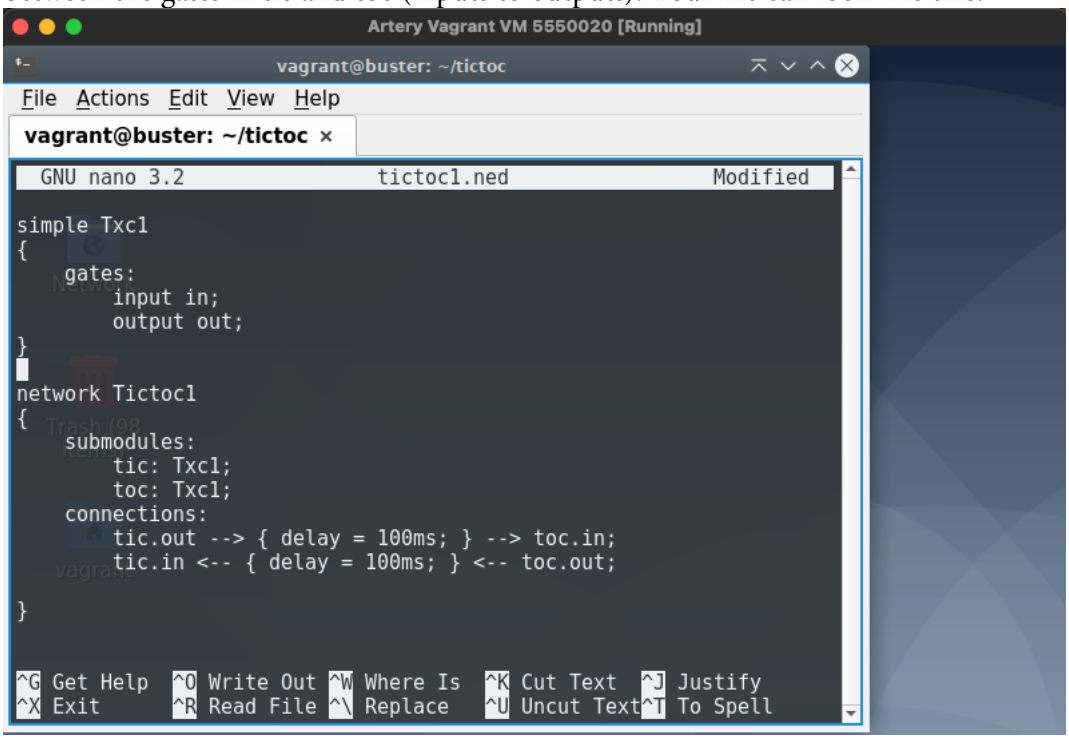
4. From this point on, we will follow a simplified version of the TicToc tutorial.
5. After loading, open the VMs terminal and create a working directory called tictoc and move into the directory.

```
vagrant@buster:~$ mkdir tictoc
vagrant@buster:~$ cd tictoc
vagrant@buster:~/tictoc$
```

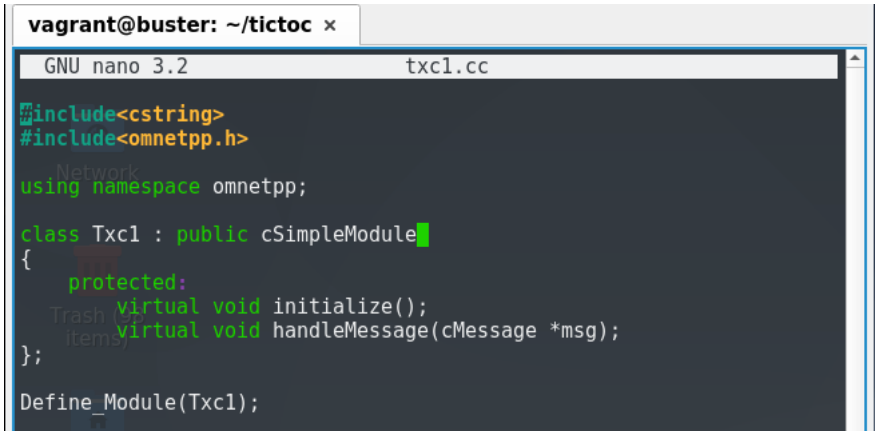
6. We will now create a "topology file" that defines the network's nodes and the links between them. For this, we type "nano" in the terminal (to open the command line text editor)



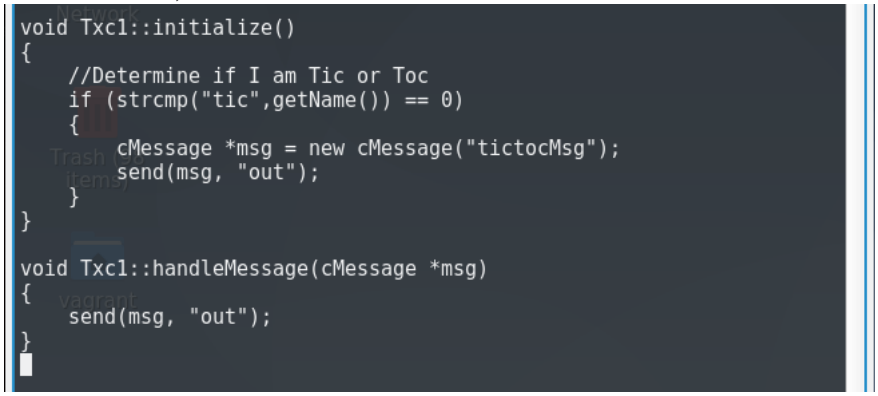
7. We define two main parts: the network and a module type. We start by defining the module type, which is an abstraction of a Transmitter/Receiver (i.e., transceiver). For this, we define two gates, one for outgoing data (output) and one for incoming data (input). Then, we define a network with two instances of our transceiver, one which we will call tic and another one called toc. Finally, we define the connections between the gates in tic and toc (inputs to outputs). Your file can look like this:



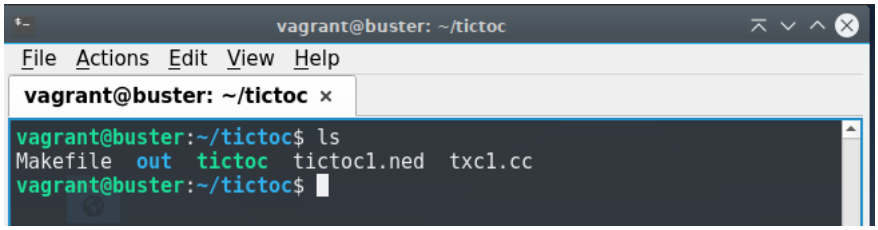
8. Save your file as tictoc1.ned
9. Now, we have to program our transceiver. OMNET++ uses C++, and the behavior of our modules is programmed in .cc files. Open nano and let us write our txc1 class.
10. First, we import the necessary headers, define the Txc1 class with its methods (initialize and handleMessage), and register the class to OMNET++



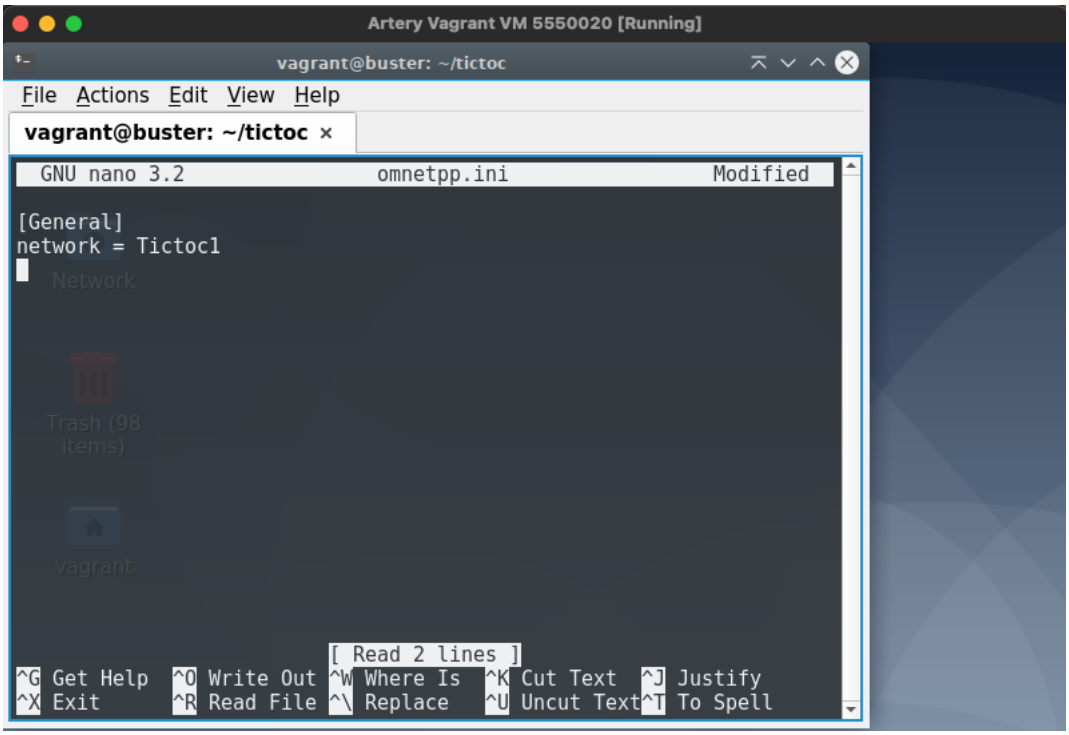
11. Then, we program the methods. In *initialize*, we determine if we are the first node in the network and start sending data through our *out* gate. This will send the first packet out, which is then processed in the other node by *handleMessage*, which only sends the same message through its out gate, thus, creating some kind of ping-pong (or better said, tic toc).



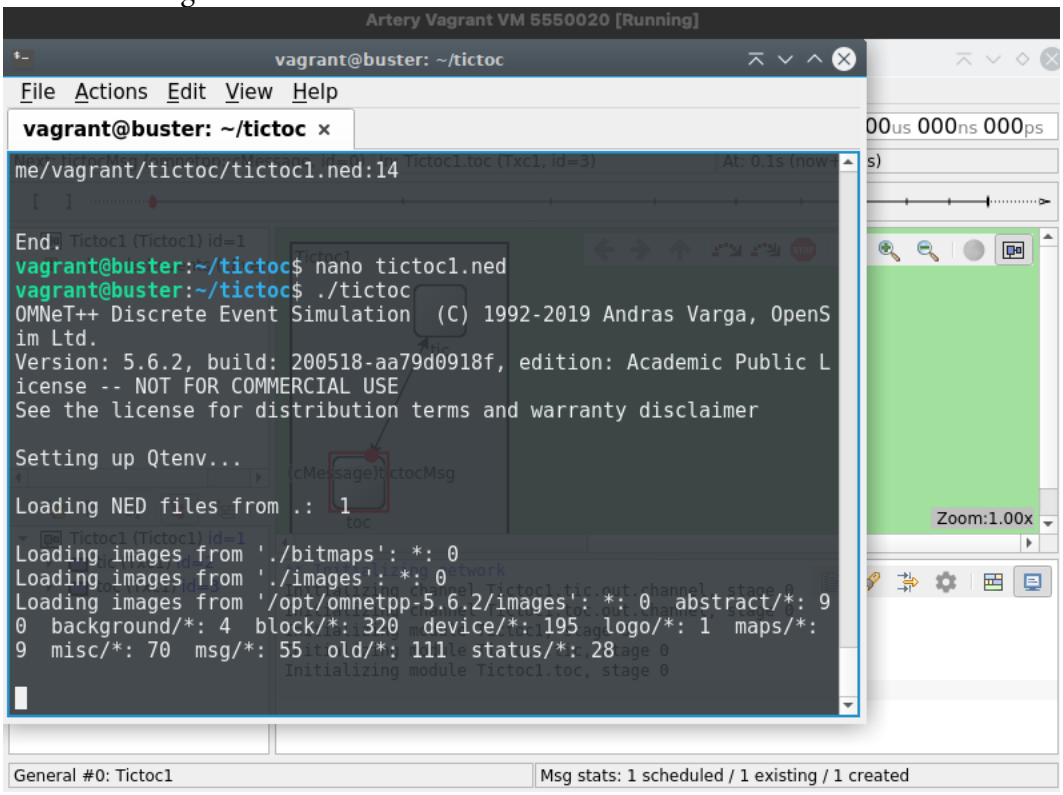
12. Save your file as txc1.cc
13. Now, we make an executable file for our simulation. First, type *opp_makemake* and then *make*. It is in this step where the .cc file gets compiled, so you may be prompted to correct errors.



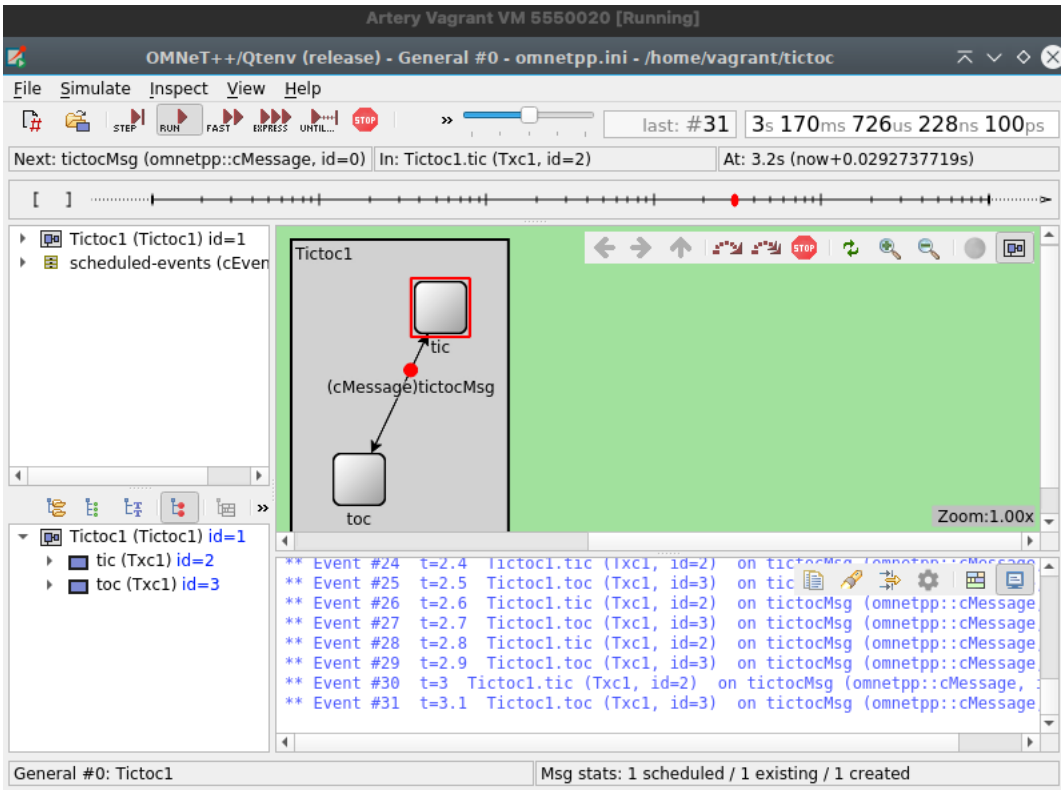
14. Now, there is an executable file in your directory. However, it cannot be run until we create an initialization file called *omnetpp.ini*. For this, we once again open nano and create the file which looks as follows.



15. Execute your simulation by typing ./tictoc. The IDE should launch and allow you to start simulating



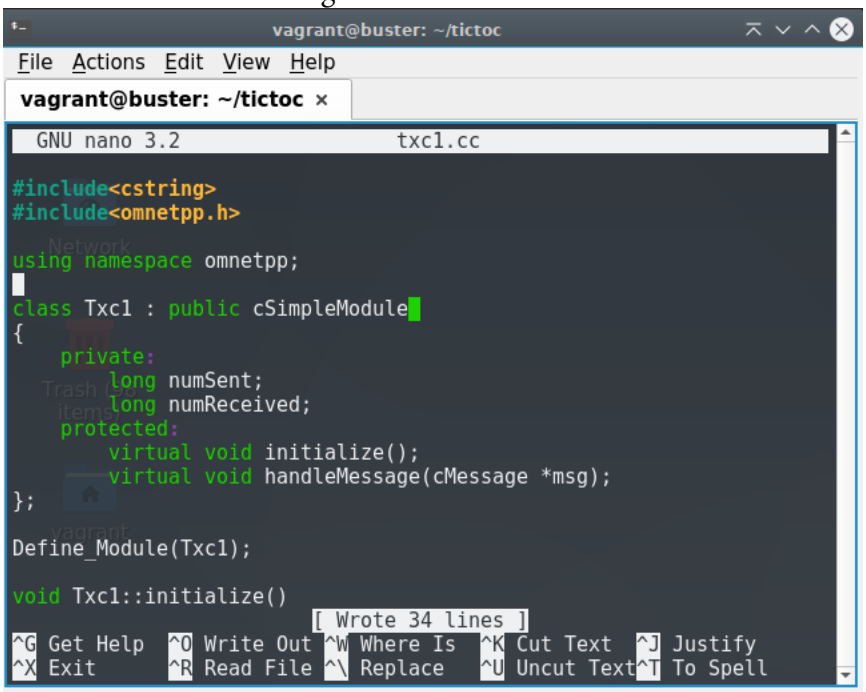
16. Click on *Run* and see what happens.



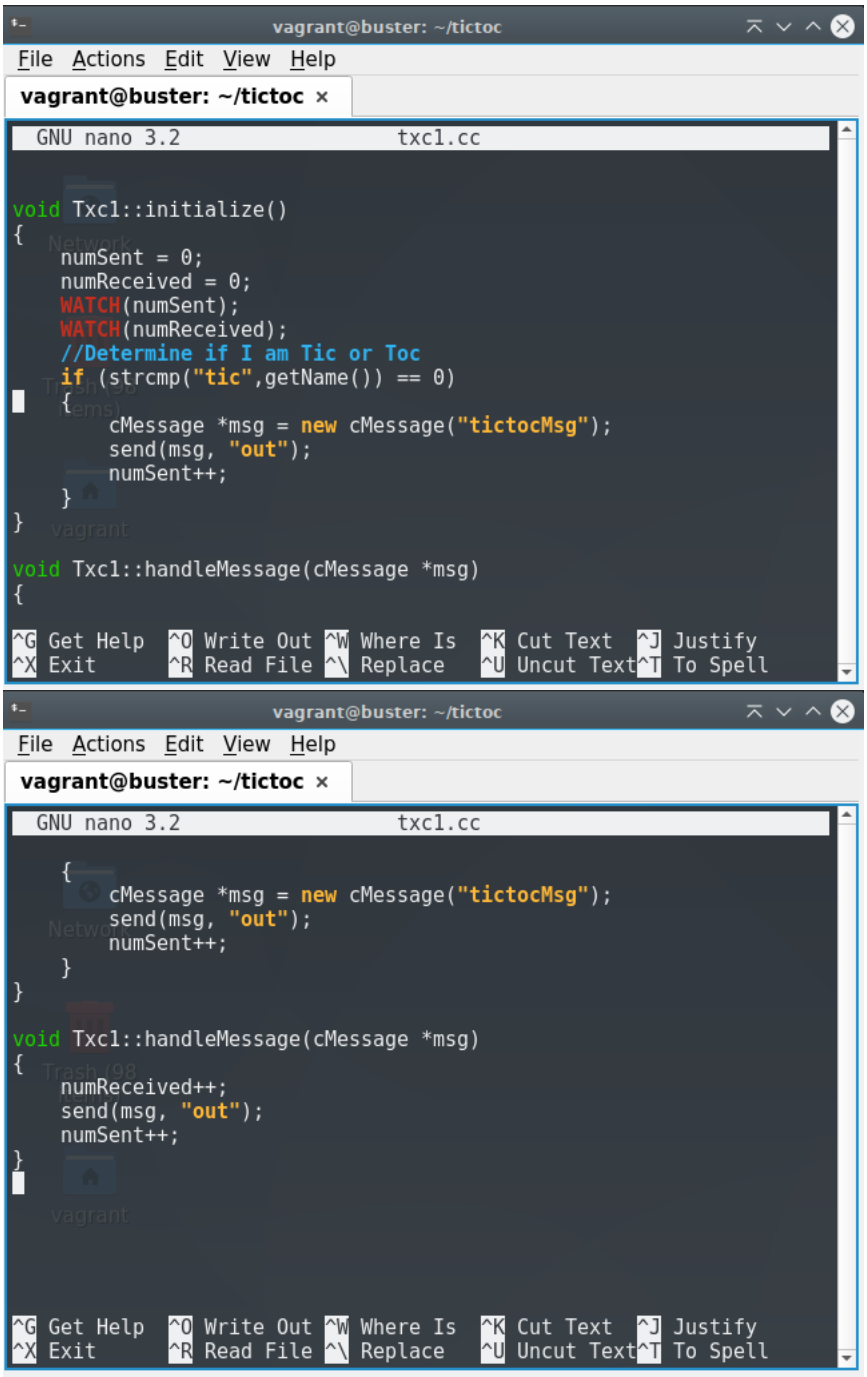
17. Document what you see in the Checkpoint section. Analyze what occurs if you change the delays. Is it possible to have asymmetric delays?

Activity 2: Building a simple network simulation with statistics collection.

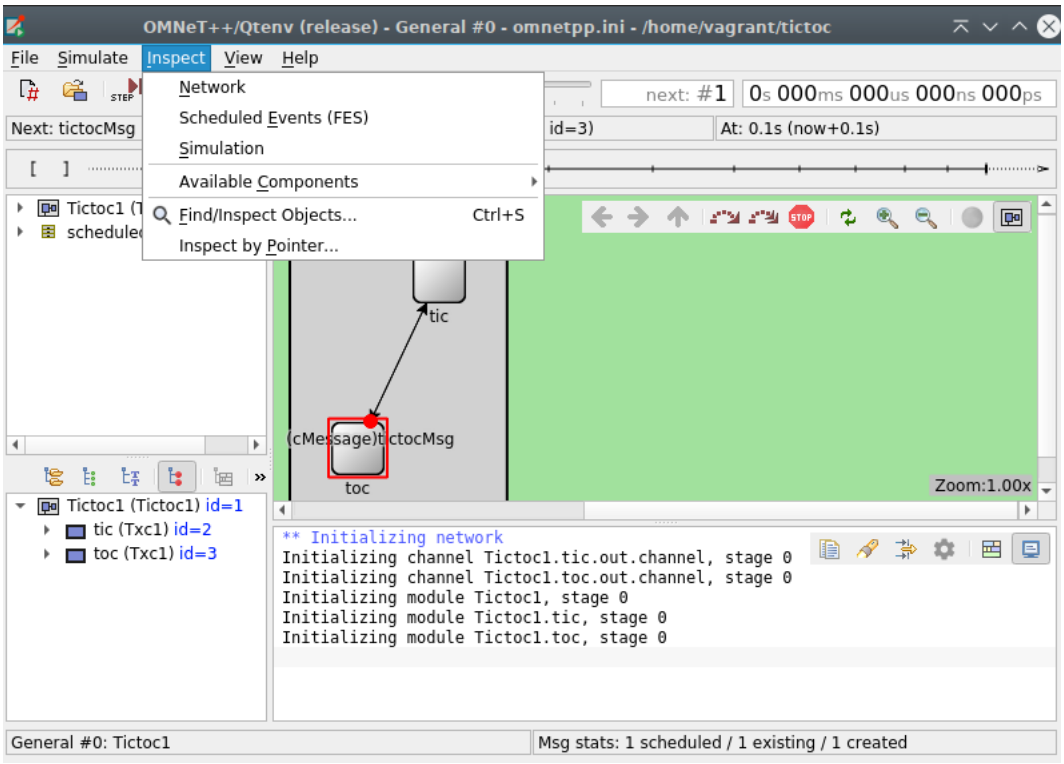
1. First, we add counters as properties to the Txc1 class. We want to count the number of transmitted and received messages.



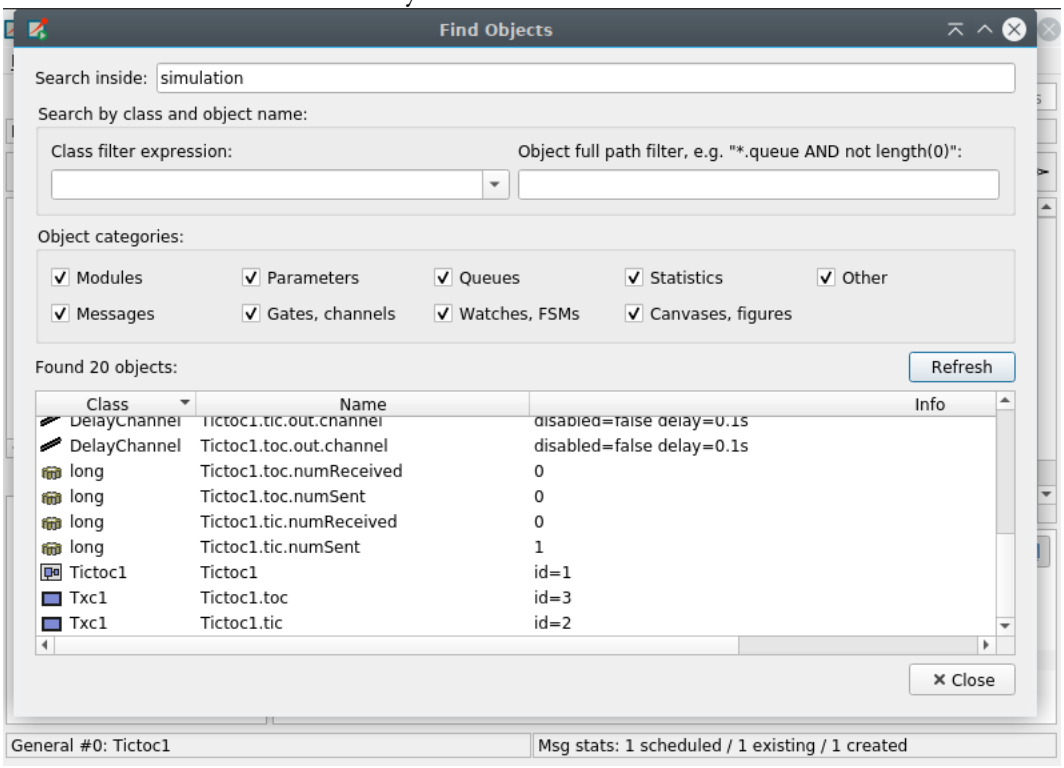
2. We then set the counters to zero and "watch" them in the initialize method. Finally, we increment the counters every time we send or receive a message.



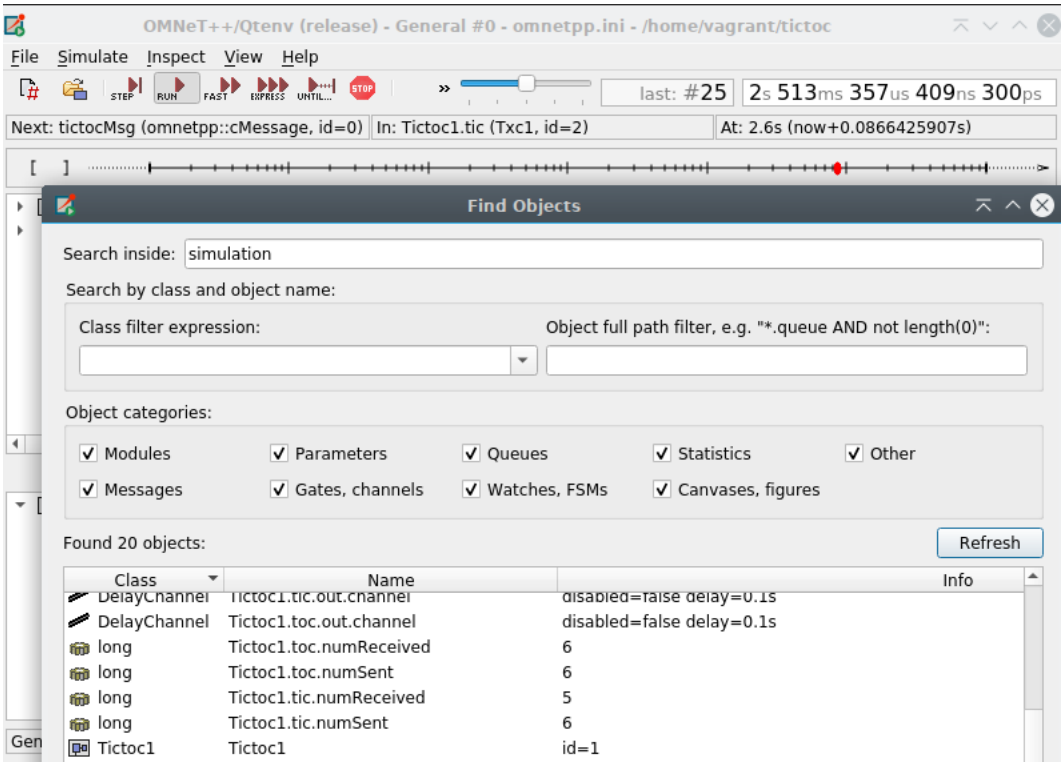
3. Recompile and run the simulation.
4. Go to Inspect / Find Objects



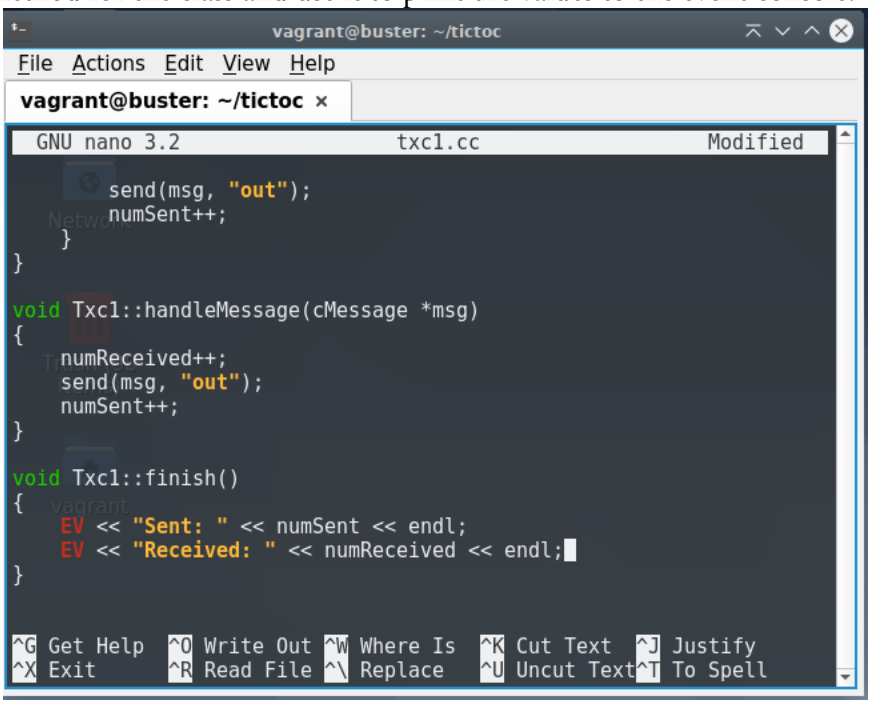
5. Click on “Refresh” and look for your watched variables.



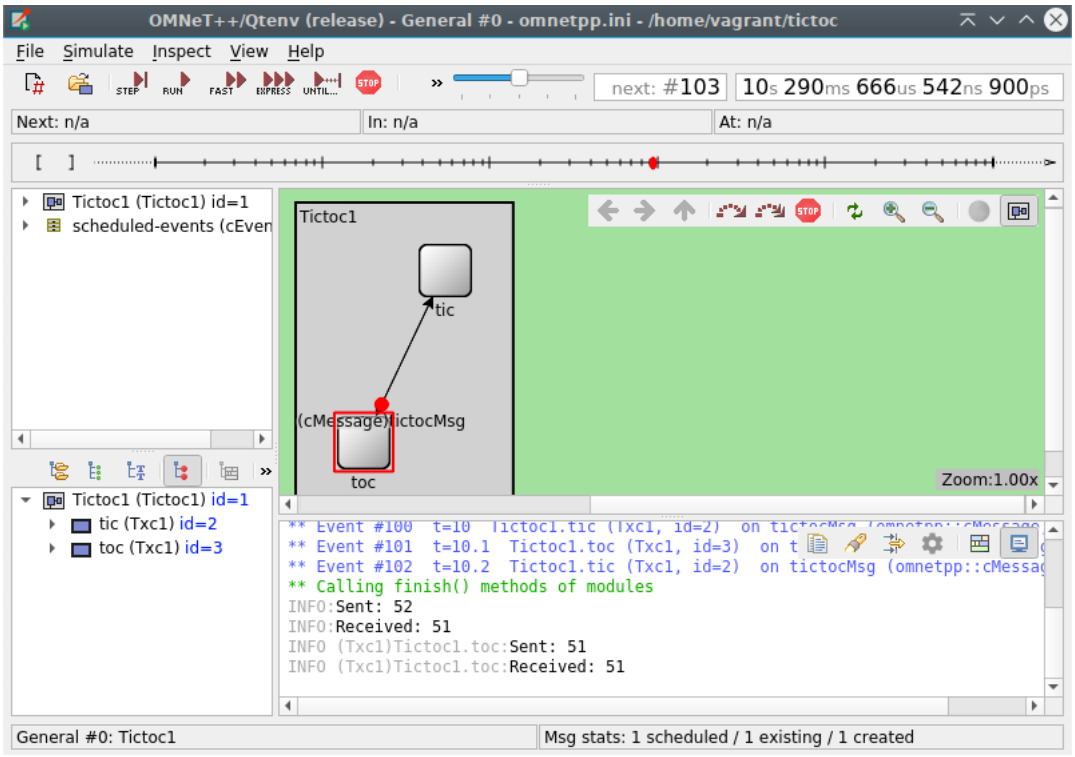
6. Refresh again and see how the values increase.



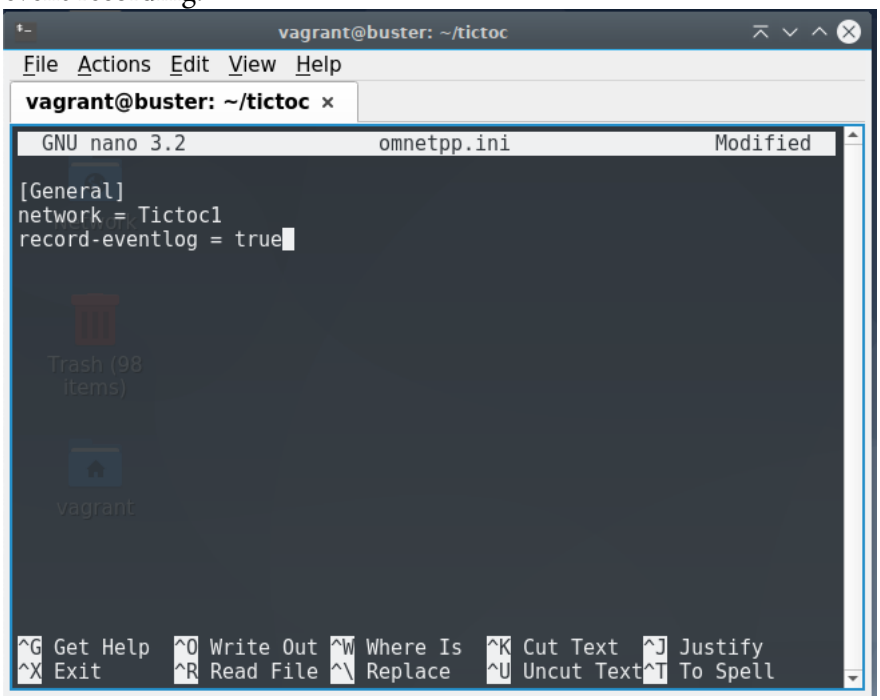
7. Now, we "automatize" the checking process by recording the results into a file. For that, we have to modify txc1.cc again.
8. We create a *finish* method to print the results when we finish the simulation. Declare it as a method for the class and use it to print the values to the event console.



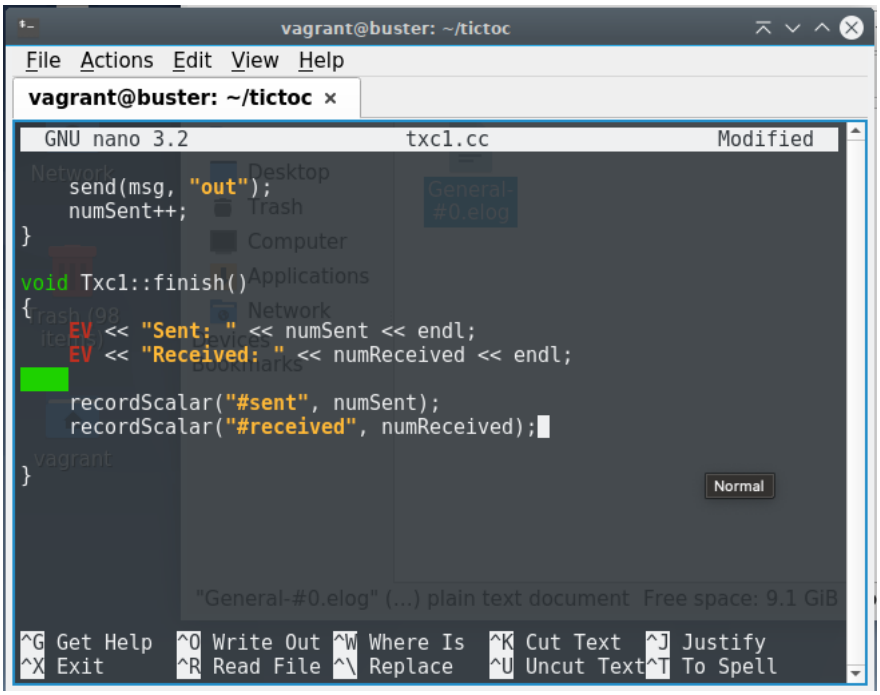
9. Recompile the simulation, execute it, and finish it after a few moments. To finish the simulation, go to "Simulate / Conclude Simulation", or look for the finish flag on the bar.
10. Look at the values that are printed and figure out their meaning.



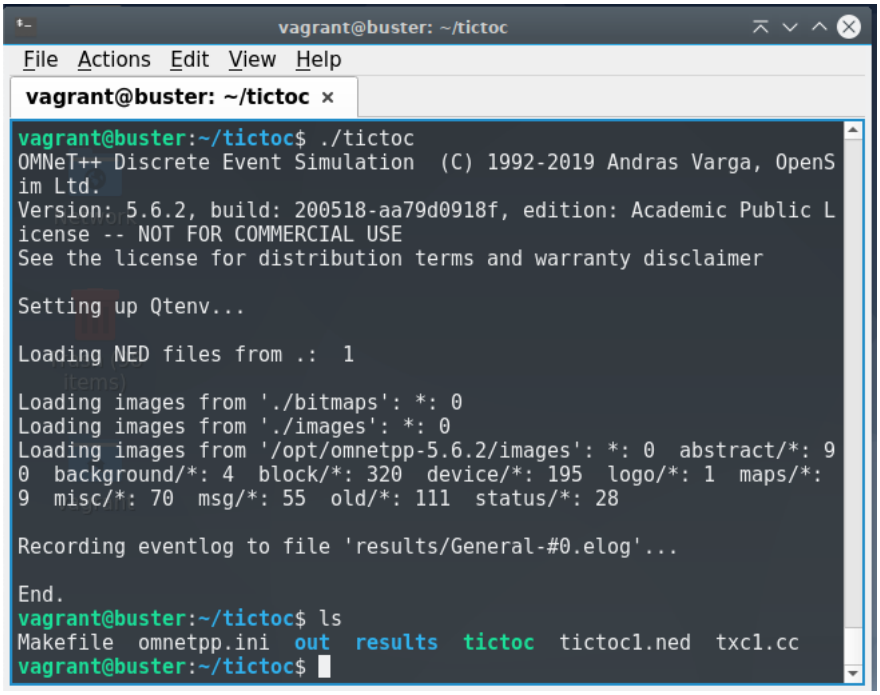
11. Now, let us automatize statistics collection even more. Edit *omnetpp.ini* to activate event recording.



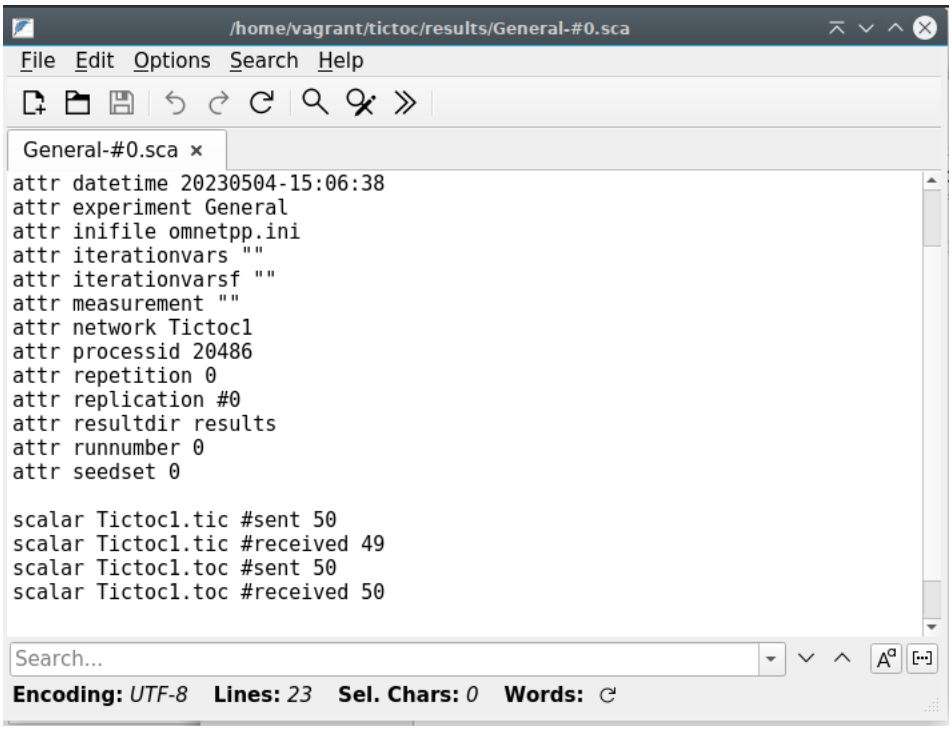
12. We now have to adapt our finish method in txc1.cc to record these numbers into a file.



13. Recompile and rerun the simulation. Start it and finish it after some time
14. If you inspect your directory and find a new folder called *results*. In there, you will find the file with the simulation results.



15. Enter the folder, open the .sca file, and see the results.



16. Configure, in *omnetpp.ini* your simulation to finish exactly after 30 seconds (in simulation time). There is a parameter called *sim-time-limit* where you establish the upper time limit for your simulation.
17. Run your simulation without the graphical interface (*./tictoc -u Cmdenv*) and it should finish almost immediately. Check the results for your simulation. Do they make sense?
18. TASK: Run the simulation for 30 seconds with at least 5 different combinations of delays (symmetrical and asymmetrical).
19. Report and explain what you observed in the last points in the Checkpoints section.

Checkpoints 1

1. Running a 2-node TicToc

Add your evidence here, writing a concise explanation of what you did.

2. Statistics for transmitted and received packets

Add your evidence here, writing a concise explanation of what you did.

Part 2: Simulating Data, Acknowledgements, and Losses (20%)

In this part of the lab, you will model a network where "data" and "acknowledgements" are exchanged between nodes. Then, you will model a lossy channel where packets are lost with a certain probability. We will start from the network you set up in the first part, so make sure yours is up and running.

During the lab, you will perform two main tasks: 1) simulate processing times and schedule transmissions after a delay, and 2) setting up a network where packets are lost with a probability,

Milestones

To be awarded the points for this part, you will show that you were able to perform the following tasks:

1. Adding Randomness (10%)
2. Setting up a network with a "lossy" channel and show transmission/reception statistics (10%)

Activity 1: Adding randomness.

A real network does not "bounce back" a packet like a ping-pong game. In a real network, upon packet reception, the receiver processes it (e.g., to use it or discard it due to errors) and can send an acknowledgement to the sender (i.e., a confirmation that the message was received correctly). This process occurs during a session as many times as needed, for example, when sending a large file consisting of several chunks.

Your original network had two transceivers bouncing back a packet. Now, we have to model a more realistic behavior. Let us thing of Tic as a sender and Toc as the receiver of a large file. Tic will now send "DATA" packets, and Toc will confirm receptions using "ACK" packets.

1. For simplicity, we will show you this using the same file as in part 1. We edit *txc1.cc* and make Tic send the first "DATA" packet. To make things more realistic, Tic will send this initial packet not during initialization, but after waiting for a random time between 0 and 1 seconds. We will need two new attributes (the message and the event scheduler), and a destructor.

vagrant@buster: ~/tictoc

File Actions Edit View Help

vagrant@buster: ~/tictoc ×

GNU nano 3.2 txc1.cc

```
#include<cstring>
#include<omnetpp.h>

using namespace omnetpp;

class Txc1 : public cSimpleModule
{
    private:
        cMessage *event = nullptr;
        cMessage *tictocMsg = nullptr;
        long numSent;
        long numReceived;

    public:
        virtual ~Txc1();

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

[Read 94 lines]

```
Define_Module(Txc1);

Txc1::~Txc1()
{
    cancelAndDelete(event);
    delete tictocMsg;
}
```

2. Initialize the new attributes and setup Tic to send the first "DATA" message after a random time. In this case, we use a uniform distribution, but OMNET++ can handle different random distributions (e.g., an exponential with average 1s could have worked)

```
    event = new cMessage("event");
    tictocMsg = nullptr;
    WATCH(numSent);
    WATCH(numReceived);
    //Determine if I am Tic or Toc
    if (strcmp("tic",getName()) == 0)
    {
        EV << "Scheduling first send to a random time\n";
        tictocMsg = new cMessage("DATA");
        scheduleAt(uniform(0,1),event);
    }
}
```

3. The *event* object will now wait for a random time and then send a "self message" to Tic. Upon reception of this "event" message, Tic sends the "DATA" packet it created and then "forgets" it and records the sending for statistics.

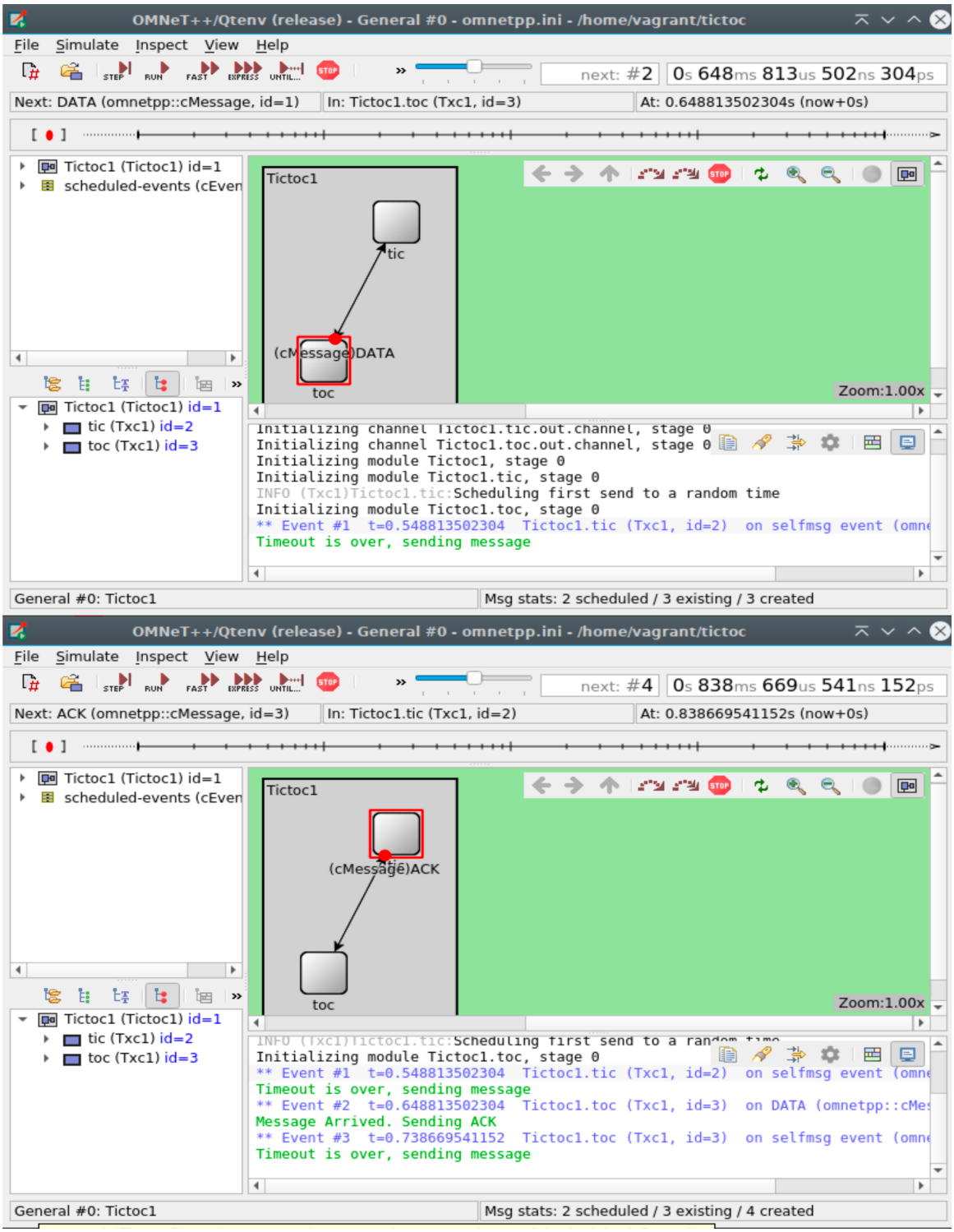
```
void Txc1::handleMessage(cMessage *msg)
{
    if (msg == event){
        EV << "Timeout is over, sending message";
        send(tictocMsg,"out");
        tictocMsg = nullptr;
        numSent++;
    }
    else {
        if (strcmp("tic",getName()) == 0){
            EV << "Acknowledgement arrived";
            numReceived++;
            delete msg;
            tictocMsg = nullptr;
            cancelEvent(event);
            tictocMsg = new cMessage("DATA");
            scheduleAt(simTime()+1.0,event);
        } else {
                EV << "Message Arrived. Sending ACK";
                numReceived++;
                delete msg;
                tictocMsg = new cMessage("ACK");
                scheduleAt(simTime()+exponential(0.1),event);
            }

        }
    }
}
```

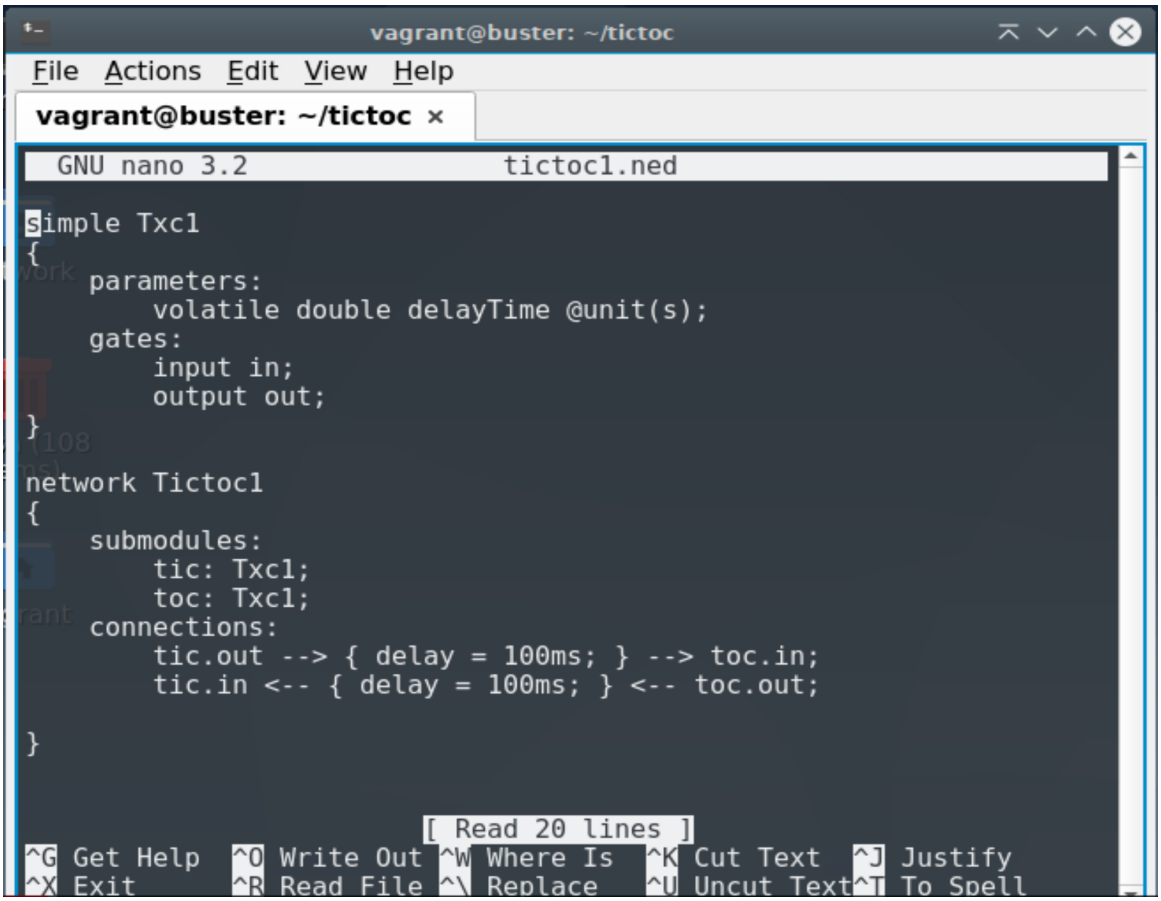
4. The packet now goes to Toc. Which has to consume it. Then, it should send an acknowledgement. For that, we delete the "DATA" message (because we have consumed it) and create an "ACK" packet. We then schedule the sending of "ACK" after some random time (to simulate the processing time in the receiver). We now can use the exponential time of average 0.1s.
5. Since we are still working on the reception procedure, we can also program Tic's behavior, so that upon receiving an "ACK" it programs the sending of a "DATA" packet one second later.

```
        if (strcmp("tic",getName()) == 0){
            EV << "Acknowledgement arrived";
            numReceived++;
            delete msg;
            tictocMsg = new cMessage("DATA");
            scheduleAt(simTime()+1.0,event);
        } else {
                EV << "Message Arrived. Sending ACK";
                numReceived++;
                delete msg;
                tictocMsg = new cMessage("ACK");
                scheduleAt(simTime()+exponential(0.1),event);
        }
```

6. Compile and run the simulation. See how Tic sends "DATA" packets and Toc sends "ACK" messages.



7. Since hard-coding delay times might not be a good idea (since it requires compiling over and over), let us change the program so that these variables can come from parameters in the .ned or .ini files. In tictoc1.ned, we add a new parameter for these delays.



8. In omnet.ini, we can specify whether we want the same delay time for Tic and Toc or if we would like them to be different. Here is an example for different values. Using the wildcard symbol (*) instead of tic/toc, would have given them the same value.

```
[General]
network = Tictoc1
Tictoc1.tic.delayTime = exponential(1s)
Tictoc1.toc.delayTime = exponential(0.1s)
record-eventlog = true
```

9. Now, we can call the parameter in the code and only change it in the .ini file. This way, we can change the values without having to re-compile.

```
if (strcmp("tic",getName()) == 0)
{
    EV << "Scheduling first send to a random time\n";
    tictocMsg = new cMessage("DATA");
    scheduleAt(par("delayTime"),event);
}
```

To better observe the effect of randomness, we can use a different method to collect statistics. OMNET++ offers a way to record the time when an event occurred using signals and vectors. For this, we will edit our files so that every transmission and reception is recorded.

10. We declare two signals as attributes for our transceivers.

vagrant@buster: ~/tictoc

File Actions Edit View Help

vagrant@buster: ~/tictoc ×

GNU nano 3.2 txc1.cc Modified

```
#include<cstring>
#include<omnetpp.h>

using namespace omnetpp;

class Txc1 : public cSimpleModule
{
    private:
        cMessage *event = nullptr;
        cMessage *tictocMsg = nullptr;
        long numSent;
        long numReceived;
        simsignal_t transmissionSignal;
        simsignal_t receptionSignal;
```

11. We register the signals on initialization.

```
};

Define_Module(Txc1);

Txc1::~Txc1()
{
    cancelAndDelete(event);
    delete tictocMsg;
}

void Txc1::initialize()
{
    numSent = 0;
    numReceived = 0;
    event = new cMessage("event");
    tictocMsg = nullptr;
    WATCH(numSent);
    WATCH(numReceived);
    transmissionSignal = registerSignal("transmissionSignal");
    receptionSignal = registerSignal("receptionSignal");
    //Determine if I am Tic or Toc
```

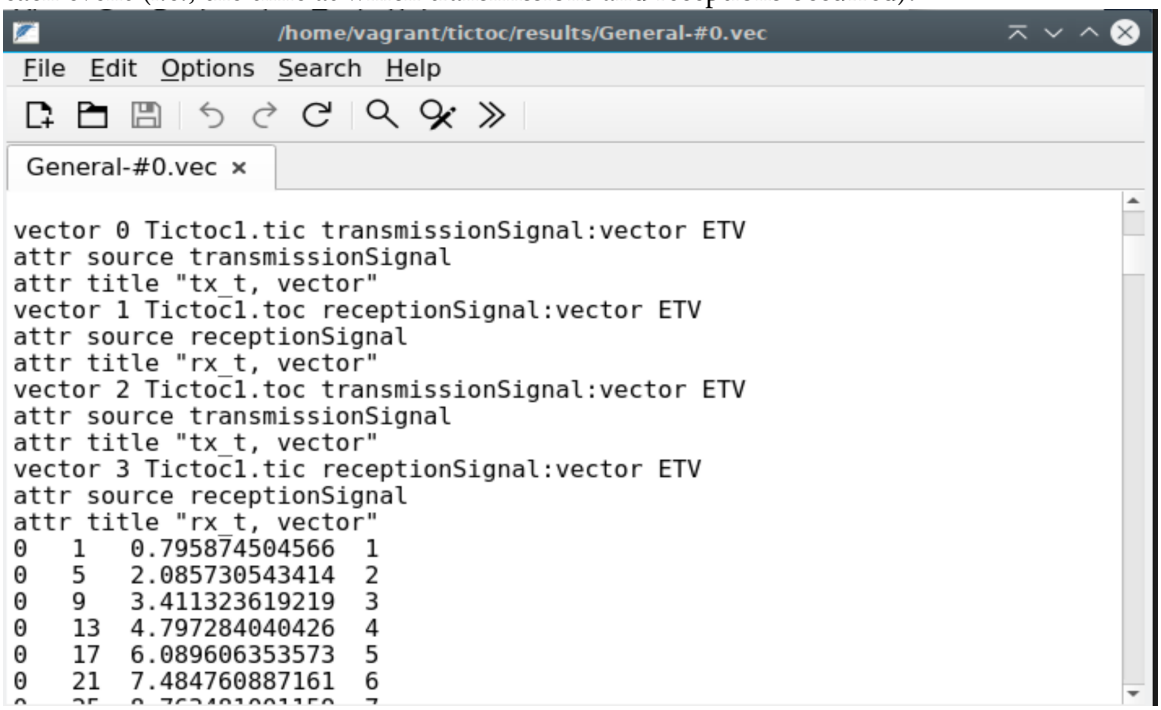
12. We emit the signals upon each reception. This way, we will have more granularity to analyze our results, since not only we will record the total number of transmissions or receptions. See the examples below (showing only for Tic due to display constraints).

```
void Txc1::handleMessage(cMessage *msg)
{
    if (msg == event){
        EV << "Timeout is over, sending message";
        send(tictocMsg,"out");
        tictocMsg = nullptr;
        numSent++;
        emit(transmissionSignal,numSent);
    }
    else {
        if (strcmp("tic",getName()) == 0){
            EV << "Acknowledgement arrived";
            numReceived++;
            emit(receptionSignal,numReceived);
```

13. Declare the signal and the statistic in the .ned file [the cut part reads “record=vector);”]

```
simple Txc1
{
    parameters:
        volatile double delayTime @unit(s);
        @signal[transmissionSignal](type="long");
        @statistic[transmissionSignal](title="tx_t";source="transmissionSignal";recor$
        @signal[receptionSignal](type="long");
        @statistic[receptionSignal](title="rx_t";source="receptionSignal; record=vect$
    gates:
        input in;
        output out;
}
```

14. Run your simulation for a few seconds, finish it, and look in the results folder for a “.vec” file. It will show the statistics that were collected. The top of the file describes the simulation information, and you can scroll down to the definition of each vector. In this case, vector 0 is the transmission vector for Tic. Then we can see the values for each event (i.e., the time at which transmissions and receptions occurred).



15. TASK: Run the simulation with different values for delays and calculate:
    a. Inter-generation Gaps (IGGs): the average times between transmissions for each node.
    b. Inter-packet Gaps (IPGs): how often (in average) nodes hear from each other (e.g., how often Toc receives messages from Tic)

    Obtain the average and the standard deviation for IGGs and IPGs and report them in the Checkpoints section.

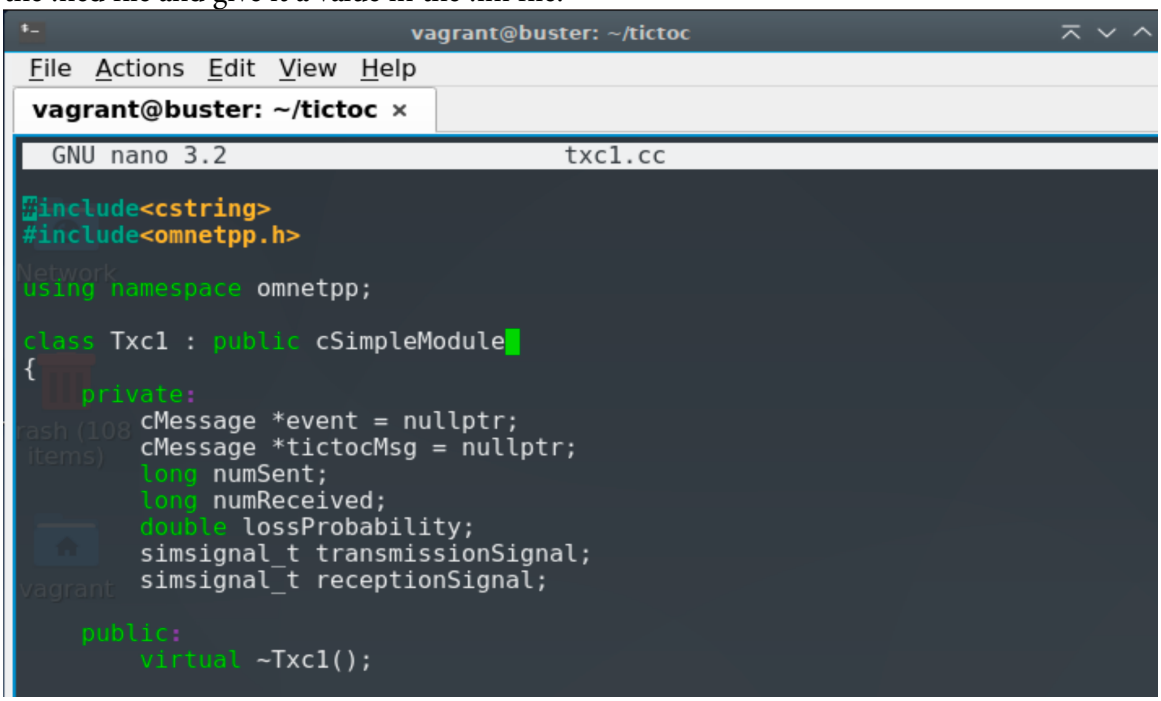
Activity 2: Setting up probability of losses.

Your original network had two transceivers. We will change that so that Tic becomes a transmitter and Toc is only a receiver. For that, we have to make Tic send periodic messages (e.g., every second) and Toc will receive and record them. We can depart from our last code and just comment out the sending of ACKs.

1. Let us edit *txc1.cc*. We now want Tic to transmit messages periodically, e.g., every 1 second. For that, we modify the "event" listener. This way, after the first message, Tic schedules new messages every second. It is hardcoded here but remember we can parametrize this value.

```
void Txc1::handleMessage(cMessage *msg)
{
    if (msg == event){
        EV << "Timeout is over, sending message";
        send(tictocMsg,"out");
        tictocMsg = nullptr;
        numSent++;
        emit(transmissionSignal,numSent);
        if (strcmp("tic",getName()) == 0){
            tictocMsg = new cMessage("DATA");
            scheduleAt(simTime()+1.0,event);
        }
```

2. We declare an attribute for the probability of a loss. Since we will use several probabilities for losses in the exercise, remember to also declare it as a parameter in the .ned file and give it a value in the .ini file.



vagrant@buster: ~/tictoc

GNU nano 3.2 txc1.cc

```
        virtual void finish();
};

Define_Module(Txc1);

Txc1::~Txc1()
{
    cancelAndDelete(event);
    delete tictocMsg;
}

void Txc1::initialize()
{
    numSent = 0;
    numReceived = 0;
    event = new cMessage("event");
    tictocMsg = nullptr;
    lossProbability = par("lossProbability");
    WATCH(numSent);
    WATCH(numReceived);
    transmissionSignal = registerSignal("transmissionSignal");
```

vagrant@buster: ~/tictoc

GNU nano 3.2 tictoc1.ned

```
simple Txc1
{
    parameters:
        volatile double delayTime @unit(s);
        double lossProbability;
        @signal[transmissionSignal](type="long");
        @statistic[transmissionSignal](title="tx_t";source="transmissionSignas
        @signal[receptionSignal](type="long");
        @statistic[receptionSignal](title="rx_t";source="receptionSignal"; res
    gates:
        input in;
        output out;
}
```

vagrant@buster: ~/tictoc

GNU nano 3.2 omnetpp.ini

```
[General]
network = Tictoc1
Tictoc1.tic.delayTime = exponential(1s)
Tictoc1.toc.delayTime = exponential(0.1s)
Tictoc1.*.lossProbability = 0.2
record-eventlog = true
```

3. Now, in the Toc reception procedure, we add a control loop that decides whether the packet should be accounted as successful or not – i.e., a “decider”. We compare our probability parameter against a uniform distribution between 0 and 1. If we decide it is lost, we just delete the message. Otherwise, we count it as received and add it to our statistics. In this example, we commented out the ACK procedure.

```
        if(uniform(0,1) < lossProbability){
            EV << "Message is lost";
            delete msg;
        } else {
            EV << "Message Arrived. Sending ACK";
            numReceived++;
            emit(receptionSignal,numReceived);
            delete msg;
            //tictocMsg = new cMessage("ACK");
            //scheduleAt(simTime()+par("delayTime"),event);
        }
```

4. Now, compile and run the simulation for some seconds and see the result (e.g., in the console or in the .sca files)

```
Running simulation...
** Event #0   t=0   Elapsed: 7e-06s (0m 00s)  0% completed  (0% total)
     Speed:     ev/sec=0   simsec/sec=0   ev/simsec=0
     Messages:  created: 3   present: 3   in FES: 2
** Event #201   t=100   Elapsed: 0.000739s (0m 00s)  100% completed  (20% tota
l)
     Speed:     ev/sec=0   simsec/sec=0   ev/simsec=0
     Messages:  created: 103   present: 3   in FES: 1

<!> Simulation time limit reached -- at t=100s, event #201

Calling finish() at end of Run #0...
[INFO]  Sent: 100
[INFO]  Received: 0
[INFO]  Sent: 0
[INFO]  Received: 84
```

In this example, we ran the simulation for 100 seconds, and 100 packets were sent, but only 84 arrived. That means that our simulation has a **Packet-delivery Ratio (PDR) of 0.84**, or 84%.

5. TASK: Run your simulation for 100 seconds for 5 different probabilities of loss (e.g., 0.1, 0.4, 0.6, 0.8, 0.99). Describe the following results in the Checkpoint section.
    a. Average IPG for each probability.
    b. PDR for each probability.

Checkpoints 2

1. Randomness, Inter-generation Gaps, Inter-packet Gaps.

Add your evidence here, writing a concise explanation of what you did.

2. Adding probability of losses and effects on IPG and Packet-delivery Ratio

Add your evidence here, writing a concise explanation of what you did.

Part 3: Automated Repeat reQuest and Packet Transmission Delay (60%)

In this part, you will 1) implement Stop-and-Wait Automated Repeat reQuest (ARQ) between two nodes in a lossy data channel and a perfect feedback channel, and 2) measure in a simulation if the delay analysis from the lectures holds in an experimental setup.

Milestones

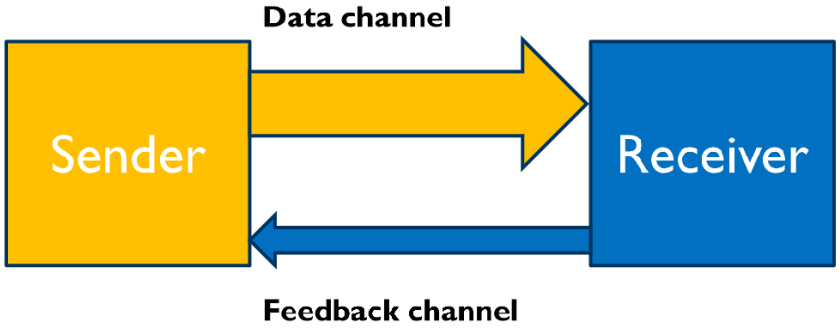
To be awarded the points for this part, you will show that you were able to perform the following tasks:

1. Implement Stop-and-Wait ARQ (30%)
2. Simulation of Transmission Delay with Retransmissions (30%)

Activity 1: Simple Stop-and-Wait Automated Repeat reQuest (ARQ).

The goal of the activity is to model a simple re-transmission protocol like the one in the lectures and lab slides.

Model: communication channels



- ❑ Data channel is used to transmit **data** packets
- ❑ Feedback channel is used to transmit **acknowledgements**

As a reminder, our network model shall have the following characteristics:

- Communication channel
  - Data channel: independent errors with probability *p*.
  - Feedback channel: no errors
- Timing:
  - Data + Acknowledgement transmission takes one unit of time

This means that Tic (sender) should not implement the probability of losses upon reception, which will only apply for Toc (receiver). Also, the timeout that the sender sets to give time for the receiver to send an acknowledgement should consider the communication delay (i.e., the value of 100ms that we configured in our TicToc network in the first activity). The file we used in Part 2 is a good starting point, but now the ACK procedure in Toc has to come back and a reaction from Tic has to be implemented.

Also, since now packets get lost, keeping track of "which" data chunk we are sending (i.e., the sequence number of the packet, in case packet 2 gets lost, we retry packet 2 and not packet 3). There is an elegant way in OMNET++ to add information into a packet, but for simplicity, here we will keep a counter that only increases when an ACK is received. That is the value we will emit and keep track of in the vector.

```
using namespace omnetpp;

class Txc1 : public cSimpleModule
{
  private:
    cMessage *event = nullptr;
    cMessage *tictocMsg = nullptr;
    long numSent;
    long numReceived;
    long msgCounter;
    double lossProbability;
    simsignal_t transmissionSignal;
    simsignal_t receptionSignal;

  public:
    virtual ~Txc1();
```

Every time Tic sends a message, it starts a timeout to re-transmit the same message if no ACK is received from Toc. In this case, that timeout is 1 second.

```
  if (msg == event){
      EV << "Timeout is over, sending message";
      send(tictocMsg,"out");
      tictocMsg = nullptr;
      numSent++;
      emit(transmissionSignal,msgCounter);
      if (strcmp("tic",getName()) == 0){
          tictocMsg = new cMessage("DATA");
          scheduleAt(simTime()+1.0,event);
      }
  }
```

If an ACK is received, the "copy" of the original message is erased, the ACK is "consumed", and a new packet is scheduled, with an increased counter, at a random time.

```
      if (strcmp("tic",getName()) == 0){
          EV << "Acknowledgement arrived";
          numReceived++;
          msgCounter++;
          emit(receptionSignal,numReceived);
          delete msg;
          delete tictocMsg;
          cancelEvent(event);
          tictocMsg = new cMessage("DATA");
          scheduleAt(simTime()+par("delayTime"),event);
```

Run your simulation with a probability of losses of 20 and observe the transmission vector for Tic. You will notice that, sometimes, a message is "repeated".

General-#0.vec ×

```
attr source receptionSignal
attr title "rx_t, vector"
0   1   0.795874504566  1
0   5   2.981071792446  2
0   9   4.163427466016  3
0   13  5.395538741076  4
0   17  6.219570754331  5
0   21  6.700307865628  6
0   25  7.41574558466   7
0   29  9.444846396747  8
0   33  10.549667506369 9
0   37  12.817862179229 10
0   39  13.817862179229 10
0   43  15.071590790149 11
0   45  16.071590790149 11
0   49  19.600509485169 12
0   53  21.855946828477 13
0   57  23.082178188069 14
0   61  24.062507855185 15
```

Observe that, while messages with different sequential numbers occur at random intervals, those with the same sequential number occur exactly one second apart from each other. This is the Automated Repeat reQuest algorithm in effect.

TASK: Configure a network that implements Stop-and-Wait ARQ with the following scenarios:

1. Fixed transmission interval at Tic of 0.5Hz. Obtain average IGGs (at the sender), IPGs (at the receiver), and PDR with at least 5 different probabilities of losses (e.g., 0.1, 0.25, 0.50, 0.75, 0.9).
2. Random transmission interval at Tic (exponential with 2s average). Calculate the same metrics for the same probabilities.

Activity 2: Analysis of transmission delay.

Using your simulation setup, prove that mean number of transmission attempts is a value inverse to the probability of a successful packet delivery in one attempt. As modeled by:

$$E[n] = \frac{1}{1-p}$$

Set up a simulation with the following characteristics:

- Sender transmits at a fixed rate of 1Hz
- Receiver will fail to get the packet successfully with a probability p. Upon transmission, it sends an ACK after a short random "processing interval".
- Sender retransmits the "same" packet until success.

Obtain the **average** number of attempts it took for each packet to be received successfully at probabilities 0.10, 0.25, 0.50, 0.75, and 0.90. Also, obtain **standard deviation** for the number of attempts for each probability. Show the results in **plots**.

Checkpoints 3

1. IGGs, IPGs, and PDR for ARQ.

Add your evidence here, writing a concise explanation of what you did.

2. Statistics (average, standard deviation) for transmission delay (i.e., number of attempts)

Add your evidence here, writing a concise explanation of what you did.

Bonus Exercise: Full Scale Simulation (up to 4 bonus points)

If you are reading this, congratulations. It means you were able to complete successfully the 3 parts of this lab and can now perform a full-scale simulation. The intention of this final part is for you to exhibit your command of statistics and simulations.

The task at hand is quite simple. You will rely on your simulation setup to perform a very similar experiment and obtain reliable results. For that, your simulation will use the repetition feature in OMNET++, where random values change between runs. If you run your simulation twice, you will notice that all the values are the exact same, so the repetition feature in OMNET++ helps you by changing the seed in the random-number generator. This way, two experiments will get two different values.

The setup is the next one:

- Sender transmits at a variable rate (exponential with average 1s)
- Receiver will fail to get the packet successfully with a probability *p*. Upon transmission, it sends an ACK after a short random "processing interval".
- Sender retransmits the "same" packet until success.
- Five probabilities (0.10, 0.25, 0.50, 0.75, 0.90)

Perform five repetitions and obtain:

- Average IPG for the receiver. *1 bonus point.*
- Average PDR and IGG for messages from the sender. *2 bonus points.*
- Show the values in plots with error bars (i.e., confidence intervals of 95%). *1 bonus point.*