SIMULATION EXPERIMENT 1

Gavin Austin (3279166) & Ayden Khairis (3282229)

28th August 2020

Experiment: Introduction to OMNeT Modelling

Section I

Objectives of the Laboratory Session:

The objective of Lab 1, undertaken on Friday 21st August, was to gain insight and knowledge of the inner workings of the Omnet++ simulation package software suite, specifically its usage modelling networks and discrete events in a network.

One key objective was to examine the file structure, and code excision within a running simulation, and make modifications that help teach the specific details that will be needed when creating further network simulations.

Model and File Structure:

The simulation is made up of 4 main files & operations:

1. Network Description file (.ned). This file described the shape and connections within a network but does not define how individual nodes act internally.
2. Configuration files (.ini) which handle inputting parameters into simulations.
3. Source files (.cc) which handle the inner workings of the objects in the network, specifically, how they initialize, handle messages and any message creation.
4. Running simulation. When these files are correctly created and modified, a simulation of the network is able to be run, and studied.

When looking over the solution to step 13, we can see how this structure works well. The .ned file houses a simple overview of the makeup of the network (as seen below), and the .cc file handles the complicated internal workings.

```
network Tictoc14
{
    @display("bgb=193,140");
    types:
        channel Channel extends ned.DelayChannel
        {
            delay = 100ms;
        }
    submodules:
        tic[6]: Txc14 {
            @display("p=48,62;is=l");
        }
    connections:
        tic[0].gate++ <--> Channel <--> tic[1].gate++;
        tic[1].gate++ <--> Channel <--> tic[2].gate++;
        tic[1].gate++ <--> Channel <--> tic[4].gate++;
        tic[3].gate++ <--> Channel <--> tic[4].gate++;
        tic[4].gate++ <--> Channel <--> tic[5].gate++;
}
```

```
void Txc14::initialize()
{
    // Initialize variables
    numSent = 0;
    numReceived = 0;
    WATCH(numSent);
    WATCH(numReceived);

    // Module 0 sends the first message
    if (getIndex() == 0) {
        // Boot the process scheduling the initial message as a self-message.
        TicTocMsg14 *msg = generateMessage();
        numSent++;
        scheduleAt(0.0, msg);
    }
}

void Txc14::handleMessage(cMessage *msg)
{
    TicTocMsg14 *ttmsg = check_and_cast<TicTocMsg14 *>(msg);

    if (ttmsg->getDestination() == getIndex()) {
        // Message arrived
        int hopcount = ttmsg->getHopCount();
        EV << "Message " << ttmsg << " arrived after " << hopcount << " hops.\n";
        numReceived++;
        delete ttmsg;
        bubble("ARRIVED, starting new one!");

        // Generate another one.
        EV << "Generating another message: ";
        TicTocMsg14 *newmsg = generateMessage();
        EV << newmsg << endl;
        forwardMessage(newmsg);
        numSent++;
    }
```

Model Modifications:

During step 10, we had to add an additional node, and modify the inout gates to be arrays, so that they can handle multiple connections. A snippet of our modified code can be found below.

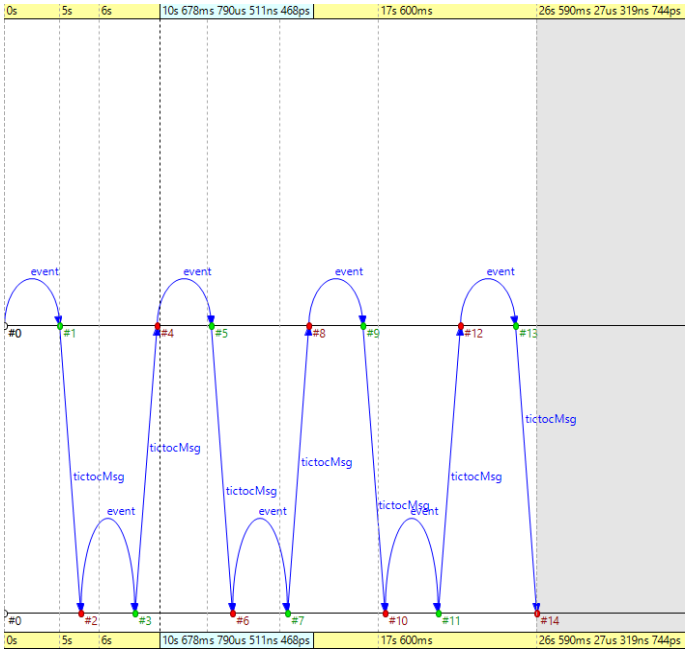
```
simple Txc10
{
    parameters:
        @display("i=block/routing");
    gates:
        input in[];
        output out[];
}
```

```
connections:
    tic[0].out++ --> {  delay = 100ms; } --> tic[1].in++;
    tic[0].in++ <-- {  delay = 100ms; } <-- tic[1].out++;

    tic[1].out++ --> {  delay = 100ms; } --> tic[2].in++;
    tic[1].in++ <-- {  delay = 100ms; } <-- tic[2].out++;

    tic[1].out++ --> {  delay = 100ms; } --> tic[4].in++;
    tic[1].in++ <-- {  delay = 100ms; } <-- tic[4].out++;

    tic[3].out++ --> {  delay = 100ms; } --> tic[4].in++;
    tic[3].in++ <-- {  delay = 100ms; } <-- tic[4].out++;

    tic[4].out++ --> {  delay = 100ms; } --> tic[5].in++;
    tic[4].in++ <-- {  delay = 100ms; } <-- tic[5].out++;
```

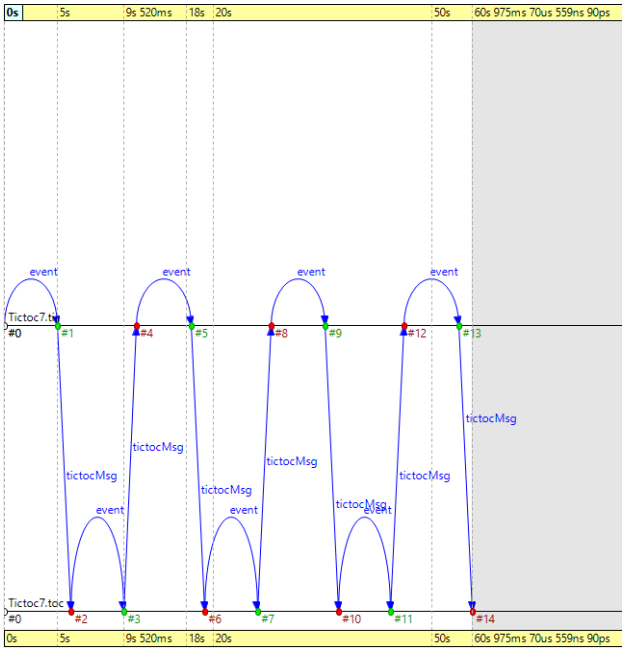
Results:

The key results for this Lab was to compare the total time for sending & receiving data with 3 different Random delay models.

Standard Delay:

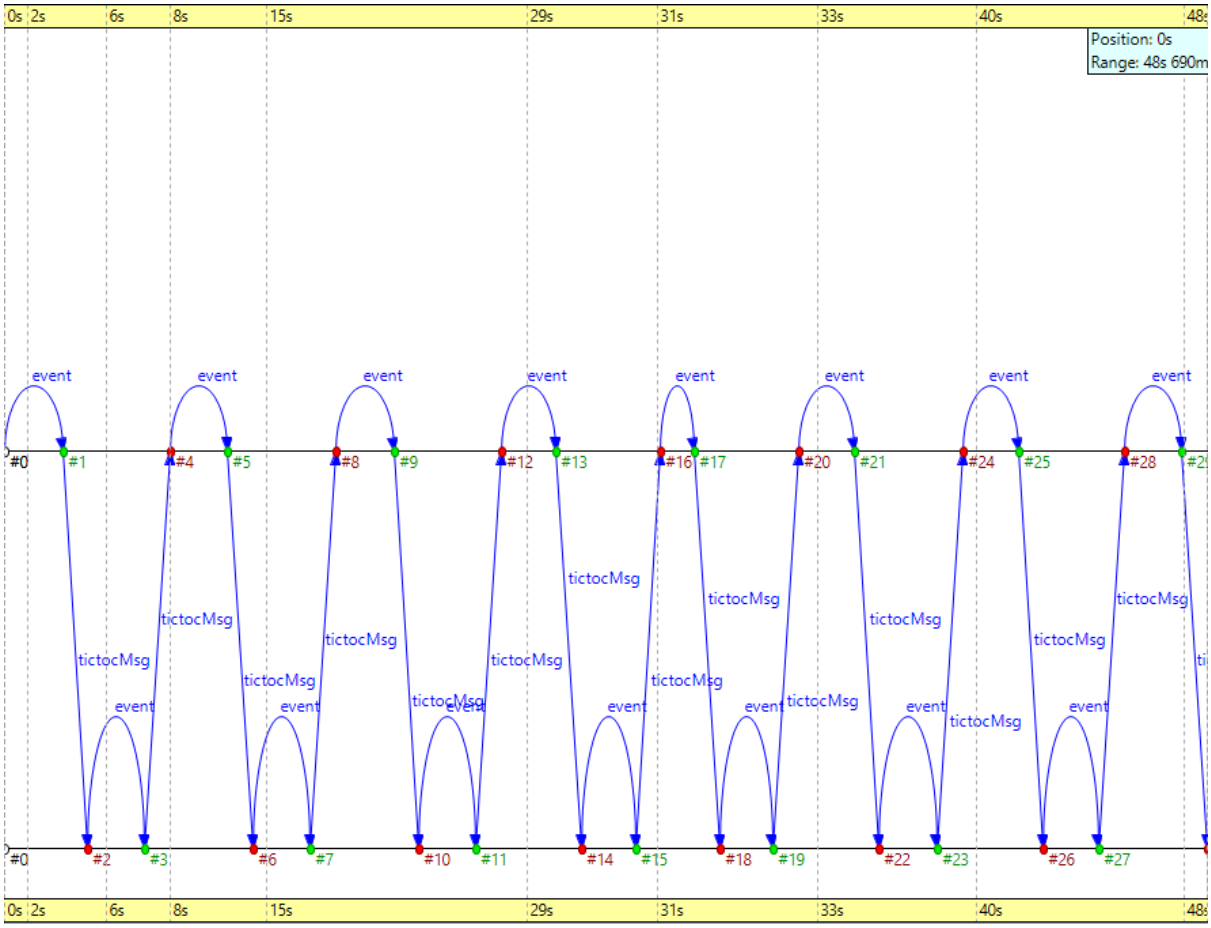


Modified Delay:



The standard Delay saw an average ‘loop’ of the circuit completed in ~10 seconds, with 3 loops being tested, whilst our modified delay (exponential(10s) & truncnormal(5s, 2s)), lasting an average of ~9.5 seconds, completing a full loop of the network in ~0.5 seconds faster than the standard dely.

Modifying the delays to be both exponential and using a standard 3second input, we saw the single distribution delay average ~14 seconds per loop of the network. This time is lost on the response of the ‘toc’ node, as a truncated normal distribution should be lower than the expected value of an ‘equal’ standard distribution. A graph displaying this can be found lower in the report. Below is the data generated by the equal distribution.



Gavin Austin, Ayden Khairis

Section II:

1. The .ned file stands for Network Description. It is used to declare modules and graphically show the network running in the simulation. The simplest ned file contains its class with variables input and output, network class with submodules containing the nodes which will be graphically shown, and the connections between the nodes.
   The .cc file is the c++ code which implements the simulation's functionality. It contains the initialize and handleMessage functions which start the simulation and handles the message which arrives at the module. Without the .cc file, the simulation would not exist.
   The .ini file is used to tell the simulation program which network we want to simulate. Within this file, you can enter the different types of ned files you want to simulate. You can also add in different paramaters such as random number generators to randomise the simulation.

2. The initialize() function is called at the beginning of the simulation. This function's role is to do all the necessary things required when a simulation is started up. This includes handling messages, setting a counter, etc.
   The handleMessage() function handles the message when it arrives at a different module. By default, it does nothing, but with some definition, the function will retrieve the event, timestamp said event, display the module containing this event, then handle the message or transfer to a different module.

3. Random number generators are used in the network model to show variance in networking. In the real world, there are a lot of factors which contribute to network variance and it's hard to properly show this variance in a network simulation. The best way to show this variance is using random numbers, as this creates unpredictable variance in the simulation.

4. The message wait time varies in step 7 because the distributions used add variance using random number generators. The message wait time is largest when using the exponential distribution compared to the truncated normal distribution because exponential growth can be theoretically infinite, whereas truncated normal distribution is (from what I have gathered) the distribution derived from that of a normally distributed random variable bounding from either below or above the normal distribution. Therefore, the exponential growth is (and always should be) larger. These random distributions also create a larger wait time compared to the same simulated event with no random distribution added.

5. The .vec file records data values as a function of time. The .sca files record aggregate values at the end of the simulation. These file types show the results of the simulation by displaying different types of statistics the simulation had (for example, mean, max, min, standard deviation, etc). You can select singular or multiple results and highlight/plot them on one chart.

6. I think that statistical distribution functions are useful for modelling communication networks because networks when simulated may seem like they are good/fast networks at first glance but might actually be really bad when looking at the statistical analysis of that network. I also think that statistical distribution functions can provide valuable analysis of network communication models because the functions make the results easy to read and understand, whilst also providing insight into the statistics of real-world network events.