HALMSTAD UNIVERSITY

Lab 2

DT4015 - Data Communications

Assignment

Part 1: Simple Packet Forwarding (20%)

In this part of the lab, you will setup and run a network with more than two nodes. A source node will send a packet from one extreme of the network, the packet will traverse the network by hopping across nodes, and then reach the other end. You will perform two main activities: 1) configure an OMNET++ network with six nodes and see how packets traverse the network, and 2) collect statistics that will allow you to calculate latency (i.e., end-to-end delay).

Milestones

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

1. Running a 6-node, serial network (10%)
2. Statistics for end-to-end delay (10%)

Activity 1: Running a 6-node, serial network.

1. Create a directory for your new project. Let us call it *multihop*.
2. Once again, we need to define a network (.ned), the behavior of the nodes (.cc) and the parameters for the simulation (.ini). Let us start by defining the network. Each node will have inputs and outputs, but for this example, we will define a "gate" which is both an input and an output. Refer to the 12th step in the official TicToc tutorial if you require further information.

```
simple Txc1
{
    parameters:
        volatile double delayTime @unit(s);
    gates:
        inout gate[];
}

network Multihop
{
    submodules:
        node[6]: Txc1;
    connections:
        node[0].gate++ <--> { delay = 100ms; } <--> node[1].gate++;
        node[1].gate++ <--> { delay = 100ms; } <--> node[2].gate++;
        node[2].gate++ <--> { delay = 100ms; } <--> node[3].gate++;
        node[3].gate++ <--> { delay = 100ms; } <--> node[4].gate++;
        node[4].gate++ <--> { delay = 100ms; } <--> node[5].gate++;
}
```

The 6 in the submodules part indicates that we will have 6 nodes, which will be indexed from 0 to 5. In the connections, we specify the bidirectional gate that connects each node serially (i.e., 0 to 1, 1 to 2, and so on). In omnetpp.ini, we have to declare the network and, just thinking one step ahead, the "processing delay" which in this case will be the same for every node.

```
[General]
network = Multihop
Multihop.node[*].delayTime = exponential(0.1s)
```

3. We then adapt our Txc1 class for this exercise. You will notice that there are some elements missing from Lab 1, which may have to come back (e.g., signals, statistics). For simplicity, we do not have them in this example.

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

using namespace omnetpp;

class Txc1 : public cSimpleModule
{
    private:
        long msgCounter;
    protected:
        virtual void initialize();
        virtual void handleMessage(cMessage *msg);
        virtual void forwardMessage(cMessage *msg);
};

Define_Module(Txc1);
```

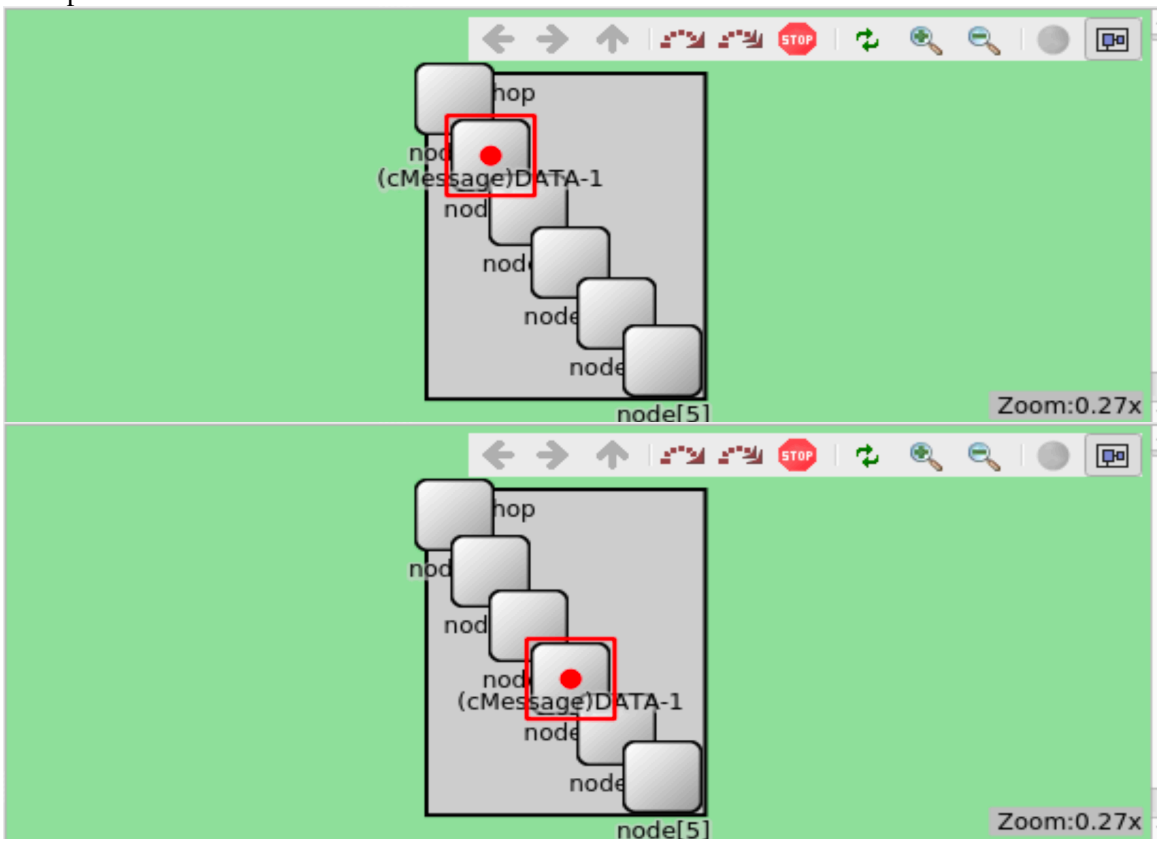
We get the first node (node[0]) to send the original message. In this example, we change the message's name to be able to differentiate them. We schedule the sending after the "processing delay".

```
void Txc1::initialize()
{
    msgCounter = 0;
    //Start messaging if I am the first node
    if (getIndex() == 0)
    {
        msgCounter++;
        EV << "Scheduling first send to a random time\n";
        char msgname[20];
        sprintf(msgname,"DATA-%d",msgCounter);
        cMessage *msg = new cMessage(msgname);
        scheduleAt(par("delayTime"),msg);
    }
}
```

4. The message handling method shall be different for the destination and for the forwarders. Forwarders implement an extra procedure to send the message out on its way. Here's a simple example of routing where something arrives on gate[0] and it is sent on gate[1] immediately (i.e., no processing delay).

```
void Txc1::handleMessage(cMessage *msg)
{
    if (getIndex() == 5){
        // Message arrived
        EV << "Message " << msg << " arrived.\n";
    } else {
        // Message has to be forwarded
        forwardMessage(msg);
    }
}

void Txc1::forwardMessage(cMessage *msg)
{
    // For this example, we always receive in the gate with
    // a lower number out of the two we have. So we forward
    // using our higher-numbered gate
    int n = gateSize("gate");
    int k = n-1;
    EV << "Forwarding message " << msg << " on gate[" << k << "]\n";
    send(msg, "gate$o", k);
}
```

5. Compile the simulation and run it. It should look like this.



The packet hops through the network until it reaches node[5].

6. TASK. Extend your simulation so that node[0] sends messages every second and keep a record of the number of transmissions and receptions on each node. Run the simulation for at least 200 seconds. Show your results in the Checkpoint section.

Activity 2: Statistics for End-to-End delay.

1. To be able to observe the effect of transmission and processing delay in multi-hop networks, we have to randomize the time before a forwarder sends the packet to the next hop. Remember the code from Lab 1, when we used the "event" messages to schedule a transmission after some random time.
2. Also, using what you did during Lab 1, add statistics collection to your simulation. In this example, we declare a vector and record only certain events (e.g., only transmissions in the source, and only receptions in the destination)

```
class Txc1 : public cSimpleModule
{
    private:
        cMessage *event = nullptr;
        cMessage *multihopMsg = nullptr;
        long numSent;
        long numReceived;
        long msgCounter;
        cOutVector txVector;
        cOutVector rxVector;
    public:
        virtual ~Txc1();
    protected:
        virtual void initialize();
        virtual void finish();
        virtual void handleMessage(cMessage *msg);
        virtual void forwardMessage(cMessage *msg);
};
```

We name the vectors on initialization.

```
void Txc1::initialize()
{
    msgCounter = 0;
    numReceived = 0;
    numSent = 0;
    event = new cMessage("event");
    txVector.setName("txVector");
    rxVector.setName("rxVector");
    //Start messaging if I am the first node
    if (getIndex() == 0)
    {
        msgCounter++;
        EV << "Scheduling first send to a random time\n";
        char msgname[20];
        sprintf(msgname,"DATA-%d",msgCounter);
        multihopMsg = new cMessage(msgname);
        scheduleAt(par("delayTime"),event);
    }
}
```

And append them on each relevant event:

```
if (getIndex() == 5){
    // Message arrived
    EV << "Message " << msg << " arrived.\n";
    numReceived++;
    rxVector.record(numReceived);
    delete msg;
```

Here we only show the recording for receptions, but the same logic applies to the other vector.

3. Compare the vectors (transmissions for 0 and receptions for 5). And there you will be able to obtain the end-to-end delay of each packet. In this example, we can see the first ten transmissions and the first ten receptions. By subtracting the timestamps, we can obtain the end-to-end delay for each packet.

```
version 2
run General-0-20230510-17:17:31-1003
attr configname General
attr datetime 20230510-17:17:31
attr experiment General
attr inifile omnetpp.ini
attr iterationvars ""
attr iterationvarsf ""
attr measurement ""
attr network Reliability
attr processid 1003
attr repetition 0
attr replication #0
attr resultdir results
attr runnumber 0
attr seedset 0
param Reliability.node[*].transmissionTime "uniform(1s , 2s)"
param Reliability.node[*].delayTime "uniform(0.01s, 0.2s)"
param Reliability.node[*].lossProbability 0.8
vector 0 Multihop.node[0] txVector ETV
vector 1 Multihop.node[5] rxVector ETV
```

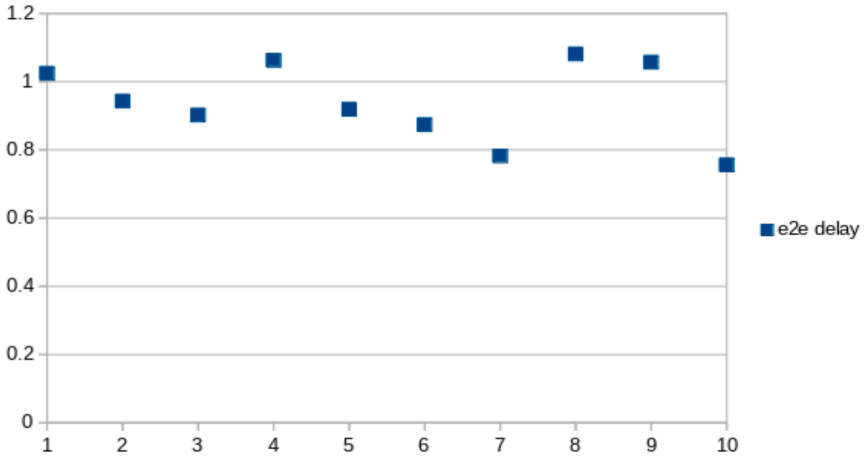
On the top of the .vec file, you will find details about the simulation

Then, you will find the information about the vectors, such as number, module, and name

```
Vec.  Event  Time              Value
0   1   0.193735669288  1
0   11  2.094357123985  2
0   21  3.889173118477  3
0   31  4.950637291339  4
0   41  6.203619652348  5
0   51  7.288778802226  6
0   61  9.192179475599  7
0   71  11.004127225759 8
0   80  12.048287279359 9
0   90  13.067312020496 10
1   10  1.217899915904  1
1   20  3.037235048444  2
1   30  4.711204985261  3
1   40  6.013197346793  4
1   50  7.122580568985  5
1   60  8.082622922847  6
1   70  9.974345211155  7
1   81  12.085204662516 8
1   91  13.105100103987 9
1   100 13.823116251133 10
```

Finally, the vector itself. Columns are:
Vector Number
Event Number
Time
Stored Value

We can take the vector (i.e., the columns) and bring them to a spreadsheet (e.g., Excel). After some post-processing, we can obtain a plot for each value:



Or some statistics, such as **averages, percentiles, confidence intervals,** among other metrics.

4. TASK: extend the simulation so that:
    a. The source node (0) sends packets at random times (e.g., an exponential of average *5s*), and records the time it transmitted a message.
    b. Forwarders send the message after a "processing delay" (e.g., an exponential of average 0.01s).
    c. The destination node (5) receives and records the message (i.e., number of messages received and when they were received)

Run the simulation for at least 200 seconds. Calculate the End-to-End delay (i.e., reception timestamp – transmission timestamp) for at least 3 different processing delays. Report your results **(averages, standard deviations)** in the Checkpoint section.

Checkpoints 1

1. Transmissions and receptions for periodic message generation.

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

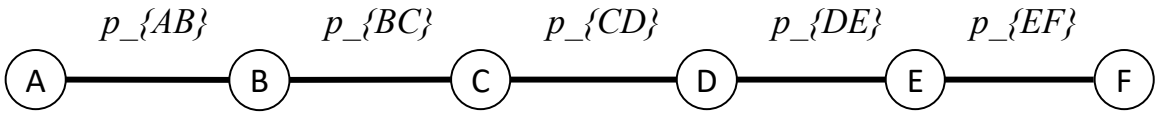
2. Statistics for end-to-end delay with random times between transmissions and random processing delays.

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

Part 2: Network Reliability with a Single Path (20%)

In this part of the lab, you will use your 6-node, serial network to transmit packets from a source, across multiple hops, to a destination. Each hop will lose a packet with a probability *p*, which you know how to set from Lab 1. You will perform two tasks: 1) analyzing network reliability in a single path with equal probability of losses, and 2) analyzing network reliability with varying probabilities of losses.

The model for our network is as follows:



Where the probability of a message from A arriving to F depends on the probability of link failures between each hop. Let us imagine that all links are equally reliable and have a probability of failure of 0.2. What is the probability of a message getting from A to F?

Milestones

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

1. Network reliability with equal probability of losses (10%)
2. Performance metrics in a network with multiple probabilities of losses (10%)

Activity 1: Network reliability with equal probability of losses.

1. Extend your code (using the methods from Lab 1) to make nodes lose a packet with a given probability. You will need to create a parameter for this probability in the .ini file, declare it as a parameter in the .ned file, and declare it as an attribute, read it, and use it in the .cc file. Here, we show you one way to assign the same probability value to all nodes in the .ini file.

```
[General]
network = Multihop
Multihop.node[*].transmissionTime = uniform(1s , 2s)
Multihop.node[*].delayTime = uniform(0.01s, 0.2s)
Multihop.node[*].lossProbability = 0.2
record-eventlog = true
sim-time-limit = 240s
```

   You could declare a different probability for each node by replacing the wildcard (*) with each node number, or with a range of nodes. For example, to give a different probability to each node individually, you would need 6 lines, one for each node. That could come in handy in the next activity, but it suffices with the wildcard for now.
2. Modify your code to implement this probability of losses. Also, make sure you are collecting statistics for receptions/transmissions on every node.
3. Compile and run your extended model. The output in the .sca file shall look like this:

```
param Multihop.node[*].transmissionTime "uniform(1s , 2s)"
param Multihop.node[*].delayTime "uniform(0.01s, 0.2s)"
param Multihop.node[*].lossProbability 0.2

scalar Multihop.node[0] "#Sent " 156
scalar Multihop.node[0] "#Received " 0
scalar Multihop.node[1] "#Sent " 126
scalar Multihop.node[1] "#Received " 126
scalar Multihop.node[2] "#Sent " 96
scalar Multihop.node[2] "#Received " 96
scalar Multihop.node[3] "#Sent " 79
scalar Multihop.node[3] "#Received " 79
scalar Multihop.node[4] "#Sent " 65
scalar Multihop.node[4] "#Received " 65
scalar Multihop.node[5] "#Sent " 0
scalar Multihop.node[5] "#Received " 55
```

The destination node received 55 out of 156 packets for a PDR of 0.35. If the loss probability between each link is 0.2, does a PDR of 0.35 approximate the calculated probability of delivery?

4. TASK: using the repetition feature of OMNET++ (i.e., adding repeat=5 in the .ini file), run your setup for 5 rounds for a probability of 0.25. Analyze all the .sca files and obtain the **average PDR.** Obtain the standard deviation and the confidence intervals and see if the results are close to the analytical results.
5. TASK: do the same for another 4 probabilities of failure (0.1, 0.25, 0.50, 0.75). Calculate the analytical result, run the 5 rounds for each probability (for at least 200s) and obtain **average PDRs** and confidence intervals and compare them with the results. Show what you obtain in a plot or a set of plots.
6. Report the results from the last two points in the Checkpoints section.

Activity 2: Performance metrics in a network with multiple probabilities of losses.

Extend your current setup to take different probabilities of losses on each link. Calculate network reliability analytically for 5 different combinations and then simulate each combination and run it 5 times.

Obtain results for **PDR, Inter-generation Gaps and Inter-packet Gaps**. Present your results in the Checkpoint section in two plots: one comparing the analytical results with PDR and one showing the difference between IGG (at the source) and IPG (at the receiver) for each combination you simulate.

Remember, Inter-generation Gaps are the times between two message generations (i.e., the average difference between two events in the transmission vector for node[0]), and Inter-packet Gaps are the times between two successful receptions (i.e., the average difference between two events in the reception vector for node[5]). You can work with the data in the .vec files in. the results folder.

Checkpoints 2

1. Analytical calculation and simulation results for Packet-delivery Ratio.
Add your evidence here, writing a concise explanation of what you did.

2. Network reliability metrics (PDR, IPG) for different probability combinations.
Add your evidence here, writing a concise explanation of what you did.

Part 3: Network Reliability with Multiple Paths (20%)

In this part of the lab, you will modify your 6-node network with a single path into different models for 6-node networks with multiple paths and different propagation schemes. The tasks you will complete are summarized in three points: 1) measure network reliability and performance when multiple paths are available and one is chosen randomly, 2) measure performance metrics when multiple paths are available and messages are disseminated through multiple paths simultaneously, and 3) model more complex networks (i.e., with more nodes and connections) and measure performance in terms of reliability.

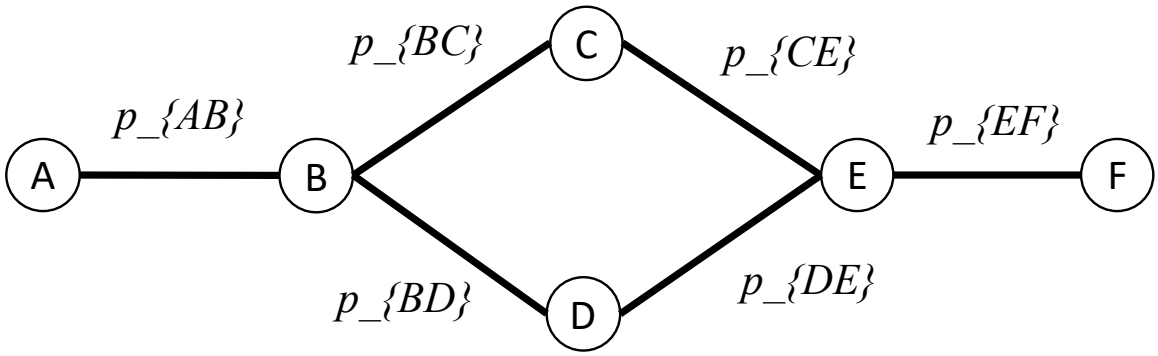
Milestones

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

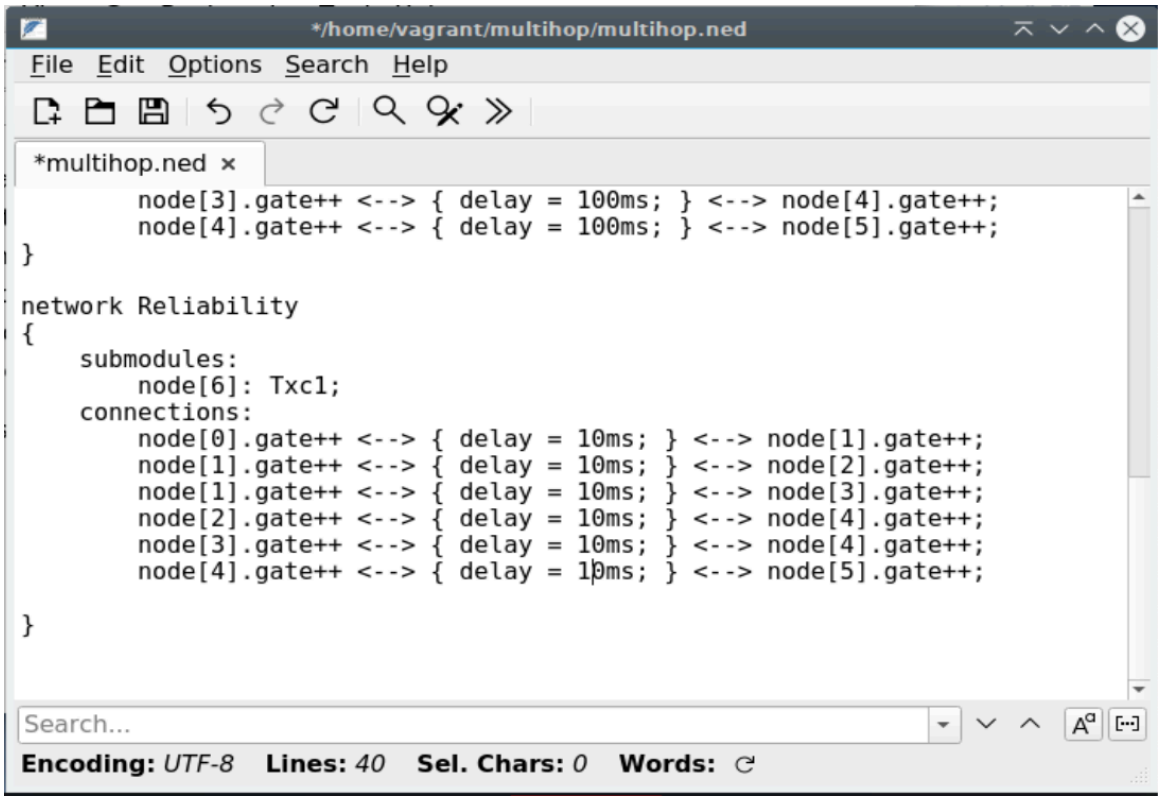
1. Network reliability and performance measurements with a single, random path in a complex network (10%)
2. Network reliability and performance measurements with multiple paths (10%)

Activity 1: Network reliability with a single-random path in a complex network.

We have to model the following network. We can start from our multihop.ned file and modify it so that the nodes are connected in a shape like this.



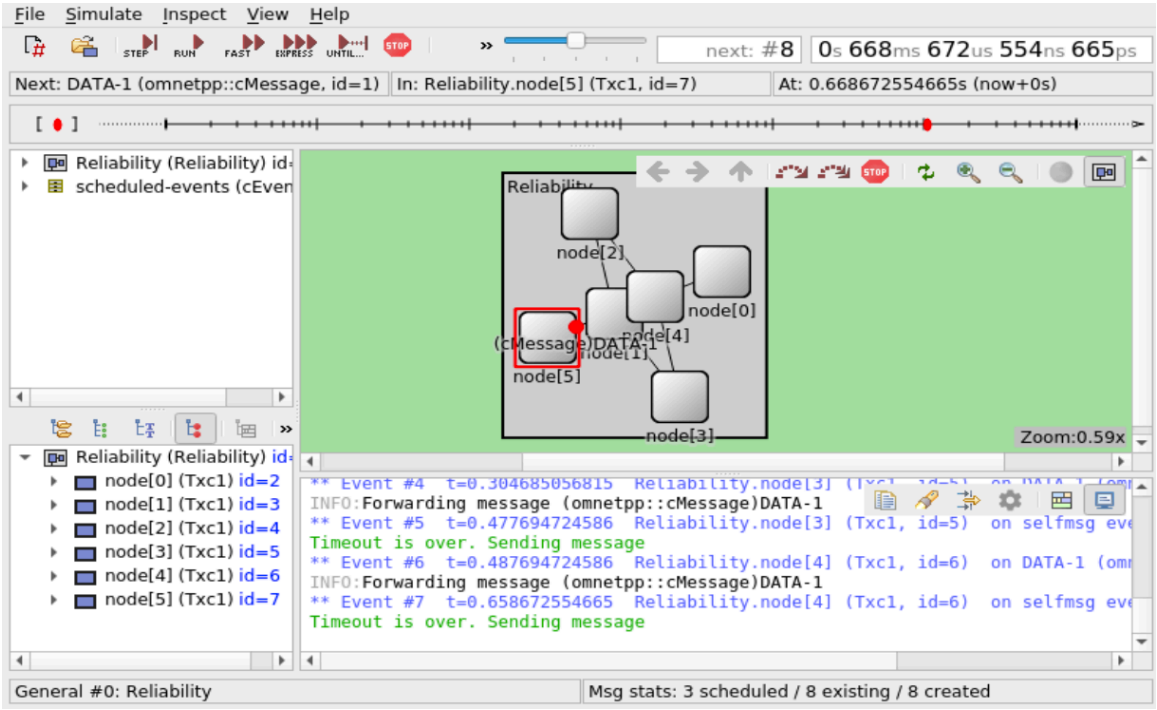
To do so, just define a new network in the multihop.ned file. The following example creates a gate for each new connection. For example, the model above makes it look like there is only one gate between B-C and B-D, but the example below makes them two different gates (e.g., like two computers connected to different ports in a switch). Also, just for the sake of it, we change the network delay to 10ms. It will influence your end-to-end measurements when compared to the last activities.



We also have to modify our omnetpp.ini file. We invoke the new network and modify the parameters so that they apply to the new network. It can look something like this.

```
[General]
network = Reliability
Reliability.node[*].transmissionTime = uniform(1s , 2s)
Reliability.node[*].delayTime = uniform(0.01s, 0.2s)
Reliability.node[*].lossProbability = 0.2
record-eventlog = true
sim-time-limit = 240s
repeat = 5
```

If you run your simulation in the graphical environment, you will see the new topology. If you try and run the actual simulation, you will notice that the packet will travel across the network but usually using the same path, e.g., ignoring one of the nodes altogether.



This is because, in this example, we are using the same code as in the serial network, where each hop selected the highest-numbered gate available to forward a message. This is why, if your followed the suggested configuration, node[1] always forwards to node[3], and never node[2]. This is due to the fact that our configuration created the gate to node[3] in the last place. To give node[2] a chance, we have to change our .cc file. For this example, since we do not want to lose the serial network and its behavior, we create a Txc2 class and a txc2.cc file.

```
if (msg == event){
    EV << "Timeout is over. Sending message";
    // For this example, we always receive in the gate with
    // the lowest number. So we choose a random gate to forward
    // a message always above the first value
    int n = gateSize("gate");
    int k = n-1;
    if (getIndex() == 0){
        send(multihopMsg, "gate$o", k);
        numSent++;
        multihopMsg = nullptr;
        txVector.record(msgCounter);
        msgCounter++;
        char msgname[20];
        sprintf(msgname,"DATA-%d",msgCounter);
        multihopMsg = new cMessage(msgname);
        scheduleAt(simTime()+par("transmissionTime"),event);
    } else {
        k = intuniform(1,n-1);
        send(multihopMsg, "gate$o", k);
        numSent++;
        multihopMsg = nullptr;
    }
```

In this example, if we do not limit the values for k in the forwarder, there is a chance that the message goes backwards. Hence the need to limit the lowest value for k (since sometimes we have to connections backwards), so a hardcoded 1 might not be the best solution. Additionally,

if you have, e.g., a disconnected node or a node with a single gate, there is a chance for k to get invalid values. Remember this if you find yourself having trouble with the values for *k*.

1. TASK: Measure the **Packet-delivery Ratio, Inter-packet Gaps** for this network when all probabilities of losses are equal (e.g., 0.2). Run the simulation 5 times and obtain averages and 95% confidence intervals. Compare it with the analytical result you calculate,
2. TASK: Measure the same metrics but having **different probabilities of losses** for each link. Compare the averages and confidence intervals to the analytical results that you obtain and expect.
3. Report these two results in the Checkpoints section.

Checkpoints 3.1

1. Analytical calculation and simulation results for Packet-delivery Ratio with equal probability of losses.

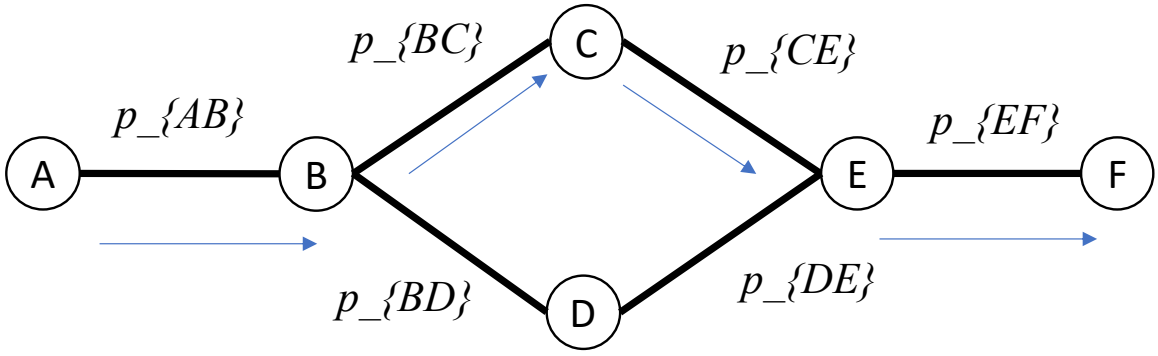
Add your evidence here, writing a concise explanation of what you did.

2. Analytical calculation and simulation results for Packet-delivery Ratio with different probability of losses.

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

Activity 2: Network reliability and performance measurements with multiple paths.

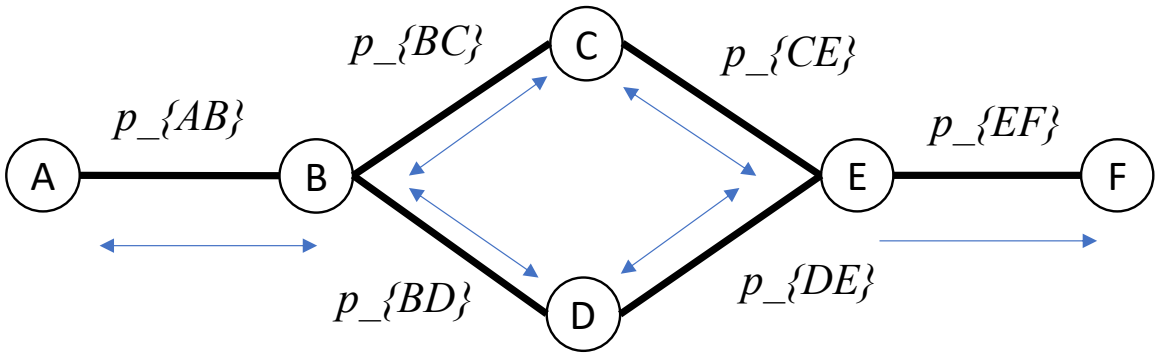
In the first activity, you tested network reliability in a network where a source has multiple possible paths to a destination and, at each hop, a random function determines the next hop for the packet and, in the end, some nodes do not receive the packet and network reliability reduces to a function of the probability of a path being chosen and the probability of failure of the chosen path.



Take the figure as an example. B has to decide (randomly) whether to forward a packet to C or to D. Each time it receives a packet from A, it "throws a coin" and one path (and only that path) is followed. Can you define the probability of a packet arriving from A to F as a function of the probability of link failures between the connected nodes and the probability of B choosing either C or D?

Simple Forwarding Mechanism

A Simple Forwarding Mechanism is one that, upon the reception of a new packet, the receiver either consumes it (if it is the destination) or **broadcasts** it to every neighbor. This way, even the last hop (i.e., the node that sent the packet to that receiver) gets another copy. If a receiver gets a copy of the same message (a duplicate), it ignores it. This is done through a mechanism called Duplicate Packet Detection (DPD), which relies on keeping and maintaining a Duplicate Packet List (DPL) that stores information about a packet which will make it easy to identify as a duplicate (even if it has gone through multiple hops).



If we extend our existing model to implement Simple Forwarding, as expressed in the figure, A (the source) will generate a packet that is sent to the next hop and registered in its DPL. In

this case, its only neighbor is B. Once B (forwarder) receives this new packet, it identifies it as new, records it in its DPL and forwards it to all its neighbors (A, C and D). A will use DPD and discard the copy, C and D will re-broadcast the packet to all their neighbors and record them in their DPLs. Let us follow the path from C, which will broadcast to B and E. DPD is activated in B, which discards the packet and E receives it as a new message and adds it to its DPL and schedules its forwarding to C, D and F. If we follow the path from D, which broadcasts the message to B, which receives another duplicate and discards it, and E. For the sake of the example, and since we have a random processing delay, let us imagine that E had received the copy from C before the one from D, so it discards the copy from D. Finally, E broadcasts its copy after the "processing delay", DPD is executed in C and D, and F (destination) receives and consumes the packet.

1. TASK: Model the probability of a successful delivery from A to F as a function of the probability of link failures between connected nodes. Note that, in this case, **all possible paths are chosen**.

Simple Forwarding in OMNET++

We will model a multi-hop network using Simple Forwarding in our 6-node, complex network. We will start from the same model (i.e., you can modify or copy it in another folder) and modify the necessary files. The .ned and .ini files that we have should be useful in their current form, so we just need to tweak our .cc file and add two features: **broadcasting** and **duplicate-packet detection**. You may want to set the probability of losses to 0 for the initial verification.

2. Let us start by changing the way our nodes select the next hop. In the past activity, the outgoing gate was chosen from a random number between the lowest and highest outgoing ports. Now, Simple Broadcast does not mind if the message goes backwards, so we send a copy of the message on every gate. One way to do so is like in this example:

```
for(int i = 0 ; i < n ; i++){
    send(multihopMsg->dup(), "gate$o", i);
    numSent++;
}
```

   This example sends a copy of the receiving message on each available gate. These messages share one characteristic in common which will enable us to keep track of duplicates.
3. Now, an easy call to enable Duplicate Packet Detection is to have a vector store certain information from a packet, i.e., a Duplicate Packet List. In this example, the information we store is the packets *Tree ID*, which identifies a "family" of packets. The original packet and every "children" (i.e., every copy generated by the *dup* method) have the same Tree ID. This value is of type *long int*. So, our DPL vector shall store long integers.

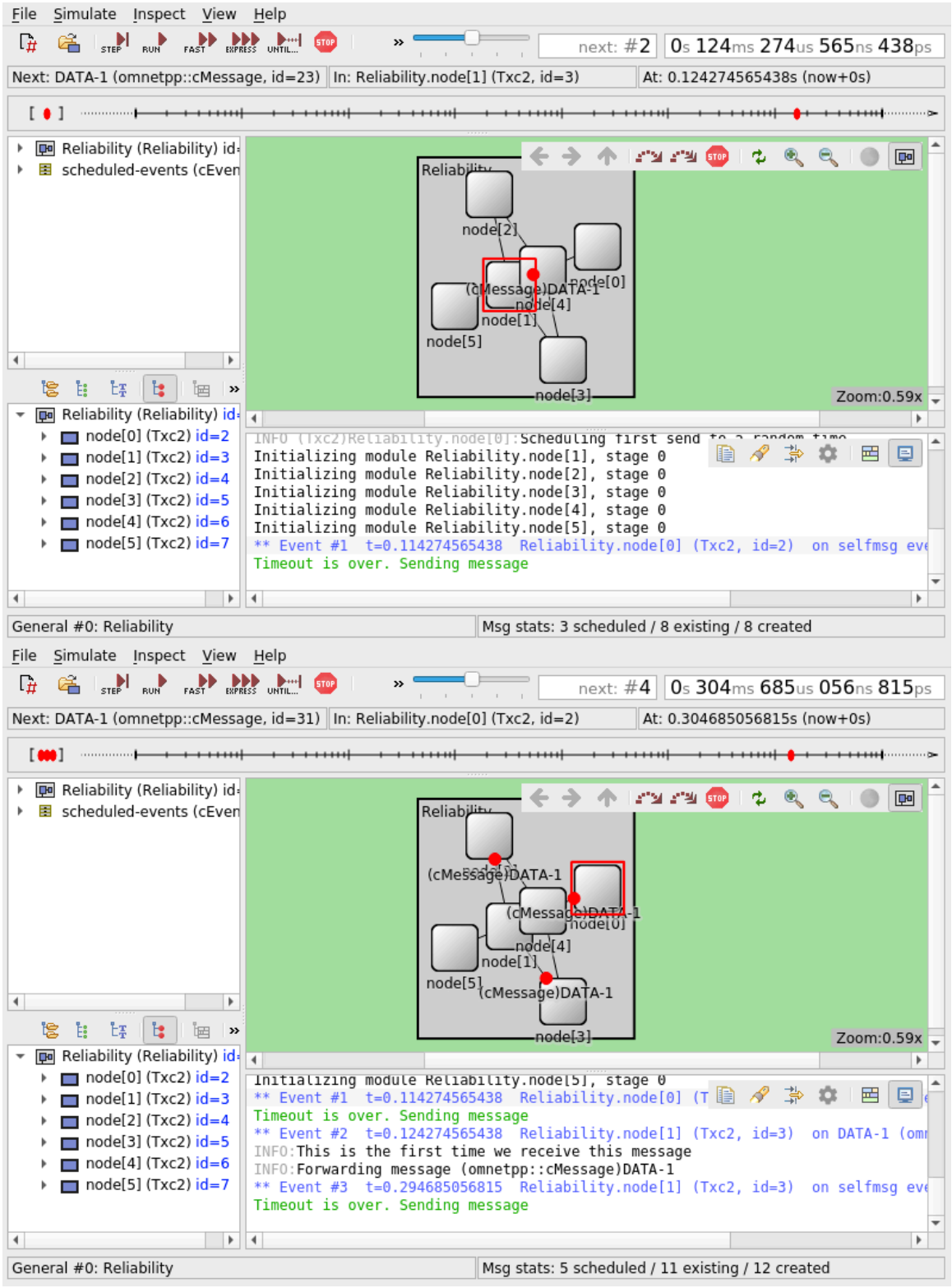
```
#include<cstring>
#include<omnetpp.h>
#include<vector>
#include<algorithm>

using namespace omnetpp;

class Txc2 : public cSimpleModule
{
    private:
        cMessage *event = nullptr;
        cMessage *multihopMsg = nullptr;
        long numSent;
        long numReceived;
        long msgCounter;
        std::vector<long> duplicatePacketList;
        double lossProbability;
        cOutVector txVector;
        cOutVector rxVector;
```

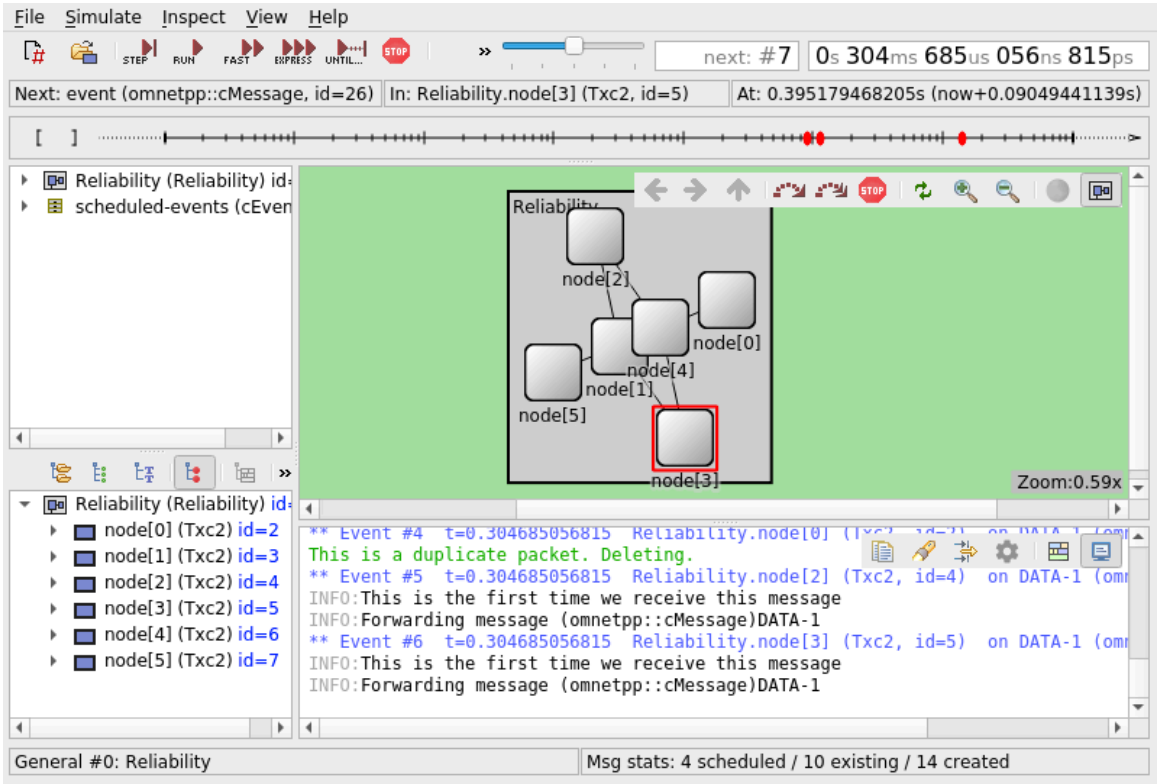
Notice the newly included libraries, which support the vector and the methods to find values in them.

4. This example shows a simple DPD mechanism. We use find to look for the message's TreeId attribute in our DPL vector. If the result of find is the end of the list, it means that the TreeId is not in the vector, thus, the packet is newly received, and we should forward it. Then, we insert the packet's TreeId into our list so that the next time we receive it, we can discard it.

```
if( std::find(duplicatePacketList.begin(),duplicatePacketList.end(),msg->getTreeId()) !=
     duplicatePacketList.end() ){
    EV << "This is a duplicate packet. Deleting.";
    delete msg;
} else {
    EV << "This is the first time we receive this message\n";
    duplicatePacketList.push_back(msg->getTreeId());
    forwardMessage(msg);
}
```

5. Make sure your code now implements Simple Broadcasting and Duplicate-packet Detection. Compile it, and check how it behaves. In this example, DATA is sent from node[0] to node [1]. It's a new packet, so 1 broadcast it to 0, 2 and 3. DPD is executed in 1, but 2 and 3 forward the message to 4 (at different times due to the "processing delay" parameter, one will be forwarded and the other one will be discarded. A similar process will occur at 4, which will re-broadcast the message, DPD will be executed in some nodes and 5 will consume the packet.


File Simulate Inspect View Help
next: #2
0s 124ms 274us 565ns 438ps
Next: DATA-1 (omnetpp::cMessage, id=23)
In: Reliability.node[1] (Txc2, id=3)
At: 0.124274565438s (now+0s)
Reliability (Reliability) id
scheduled-events (cEven
node[2]
node[0]
(cMessage)DATA-1
node[4]
node[1]
node[5]
node[3]
Zoom:0.59x
Reliability (Reliability) id
node[0] (Txc2) id=2
node[1] (Txc2) id=3
node[2] (Txc2) id=4
node[3] (Txc2) id=5
node[4] (Txc2) id=6
node[5] (Txc2) id=7
Initializing module Reliability.node[1], stage 0
Initializing module Reliability.node[2], stage 0
Initializing module Reliability.node[3], stage 0
Initializing module Reliability.node[4], stage 0
Initializing module Reliability.node[5], stage 0
** Event #1 t=0.114274565438 Reliability.node[0] (Txc2, id=2) on selfmsg eve
Timeout is over. Sending message
General #0: Reliability
Msg stats: 3 scheduled / 8 existing / 8 created
File Simulate Inspect View Help
next: #4
0s 304ms 685us 056ns 815ps
Next: DATA-1 (omnetpp::cMessage, id=31)
In: Reliability.node[0] (Txc2, id=2)
At: 0.304685056815s (now+0s)
Reliability (Reliability) id
scheduled-events (cEven
(cMessage)DATA-1
node[4]
node[1]
node[5]
node[3]
Zoom:0.59x
Reliability (Reliability) id
node[0] (Txc2) id=2
node[1] (Txc2) id=3
node[2] (Txc2) id=4
node[3] (Txc2) id=5
node[4] (Txc2) id=6
node[5] (Txc2) id=7
Initializing module Reliability.node[5], stage 0
** Event #1 t=0.114274565438 Reliability.node[0]
Timeout is over. Sending message
** Event #2 t=0.124274565438 Reliability.node[1] (Txc2, id=3) on DATA-1
INFO:This is the first time we receive this message
INFO:Forwarding message (omnetpp::cMessage)DATA-1
** Event #3 t=0.294685056815 Reliability.node[1] (Txc2, id=3) on selfmsg eve
Timeout is over. Sending message
General #0: Reliability
Msg stats: 5 scheduled / 11 existing / 12 created



Note that one node executed DPD (node[0]), and two nodes will forward the message (node[2] and node[3]).

6. Run the full simulation for at least 240 seconds and analyze the results.

```
param Reliability.node[*].transmissionTime "uniform(1s , 2s)"
param Reliability.node[*].delayTime "uniform(0.01s, 0.2s)"
param Reliability.node[*].lossProbability 0.0

scalar Reliability.node[0] "#Sent " 163
scalar Reliability.node[0] "#Received " 163
scalar Reliability.node[1] "#Sent " 489
scalar Reliability.node[1] "#Received " 489
scalar Reliability.node[2] "#Sent " 326
scalar Reliability.node[2] "#Received " 326
scalar Reliability.node[3] "#Sent " 326
scalar Reliability.node[3] "#Received " 326
scalar Reliability.node[4] "#Sent " 489
scalar Reliability.node[4] "#Received " 326
scalar Reliability.node[5] "#Sent " 0
scalar Reliability.node[5] "#Received " 163
```

In this example, with a probability of losses of 0 (i.e., no losses), we see that the source node generates 163 messages, and the destination node receives 163 packets. Why does the source node receive 163 packets, and why do the nodes in between have such numbers in receptions and transmissions?

7. TASK: Use your model for network reliability that you calculated in point 1 of this activity and calculate the probability of success for 5 probabilities of losses (the same for all links for simplicity, e.g., p=0.25). Run the simulation for the same probabilities (5 repetitions) and compare your analytical results with the results from the simulations (probability of success vs average PDRs). Report your results in the Checkpoints section.

Checkpoints 3.2

1. Analytical calculation and simulation results for Simple Forwarding with equal probability of loss.

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

Part 4: Network Reliability for different network models (40%)

In this part of the lab, you will calculate network reliability for complex networks using the techniques from Part 2. We will provide you three different models for networks, and you will 1) calculate network probability when the next hop is chosen randomly and applying Simple Forwarding (i.e., when all paths are explored), 2) simulate both schemes in each of the three network models with at least five runs and compare the PDR results with your analytical results, and 3) show the effect of at least 5 different probabilities on PDR and latency.

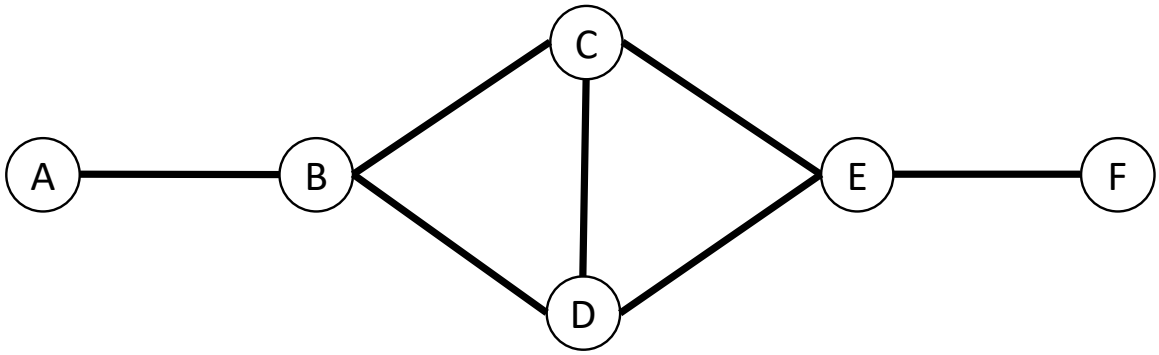
Milestones

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

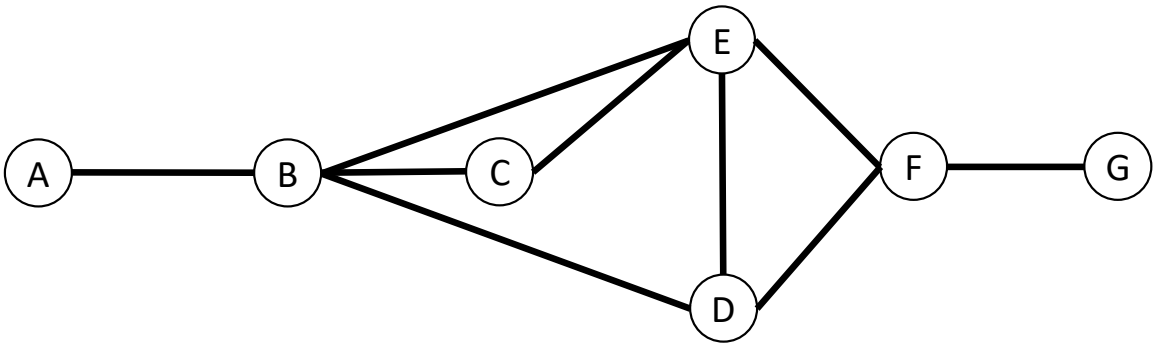
1. Compare network reliability metrics (probability against PDR and IPG) between two different forwarding schemes in 3 network models with **the same probability** (e.g., 0.25) (20%)
2. Compare network reliability metrics between two different forwarding schemes in 3 network models with **different levels of the same probability**, e.g., 0.0, 0.25, 0.50, 0.75 (20%)

Activity 1: Network reliability with a different forwarding schemes and different network models.

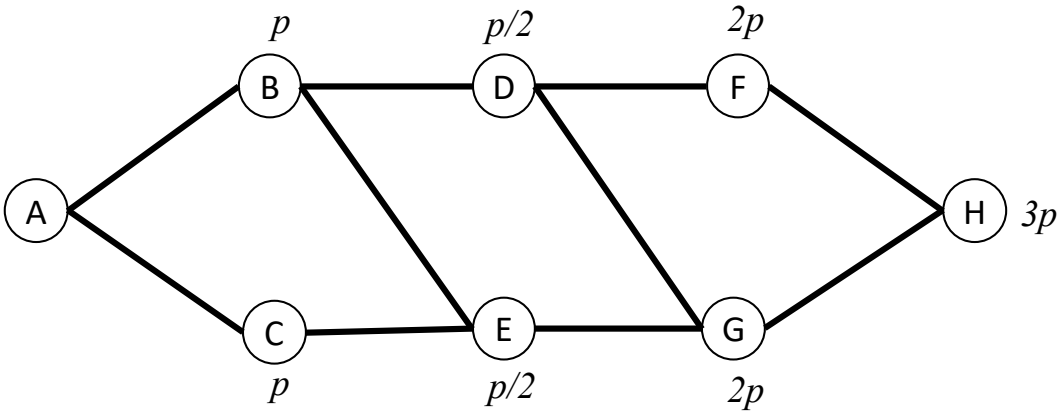
Network Model 1



Network Model 2



Network Model 3



For the last model, the probabilities of loss at certain nodes is higher or lower than at others. Let us imagine that we are analyzing for p=0.2. B and C will drop a packet with probability 0.2, D and E with probability 0.1, F and G with probability 0.4, and H will drop a packet with probability 0.6. This means that p_{AB} = p_{AC} = 0.2, or that p_{FH}= p_{GH}=0.6 and so on. Consider this when you are calculating the analytical result and when you configure your network.

Checkpoints 4

Due to the escalated complexity of the networks (even if just a few links are added), **we do not ask you to calculate the probability of losses analytically**. Only results from simulations are required in Checkpoints 3.

1. Simulation results for random next hop and Simple Forwarding in a single probability of losses. Metrics: PDR, IPG, latency. 3 networks.

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

2. Simulation results for random next hop and Simple Forwarding for different levels of probability. Metrics: PDR, IPG, latency. 3 networks.

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

Bonus exercise: Improving Network Reliability (up to 4 bonus points)

During the past two labs, you have explored the effect of packet loses in network performance. In Lab 1, you worked with a basic ARQ algorithm to ensure that a packet gets to its destination. Using what you know about data networks, solve the problem of network reliability that we analyzed in this Lab. For example, using ARQ to receive confirmation of receptions, but first, consider if the same ARQ algorithm you developed is the most adequate (e.g., how would you handle duplicate ACKs?).

The milestones of this exercise are:

1. Ensuring a Packet-delivery ratio of nearly 1 (100% success rate) of the random hop and the Simple Forwarding schemes.
2. Describe the effect of ensuring reliability on latency (i.e., end-to-end delay).
3. Compare and try to minimize the number of messages sent in each scheme.