Sensor Networks and Mobile Data Communication, Assignment 4

UID: 1690550

March 12, 2017

1 Introduction

The simulated problem involves a delay-tolerant network (DTN), with two stationary nodes 0 and 2, positioned 10000 m apart, and a mobile node 1, moving between them, starting from (0,3) in the x-y plane. The initial position of the nodes are shown in Fig. 1. We introduce the movement later; for now Node 1 is stationary.

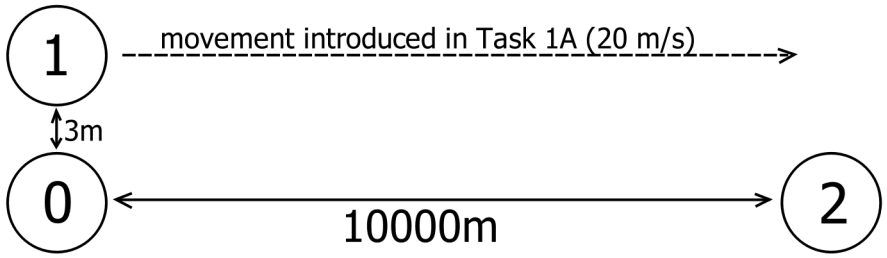

Figure 1: Initial topology of the network, with the movement as introduced in the first task

1.1 Node 0

The code responsible for Node 0 (first village) behaviour starts at line 234 of the original code.

The function `Node0DataGen()` maintains the buffer with the messages. It calls itself every second. With each call, it increases the `global_counter` variable which acts as a stamp for the mails, and adds the message to the buffer, by updating the head. If adding the message to the buffer would exceed the buffer size, it also moves the tail forward. On the other hand, if the buffer is empty, it also marks the `isbufferempty` flag as non-empty after adding a message to the buffer. In this case, the tail and head are set to 1.

Sending the messages to Node1 (the bus) is done by `Node0SendPacket(...)` function. It tries to send the buffer from Node0 every 0.25 seconds. The encapsulation of the buffer is done by creating a `MyHeader` object. It contains the following pieces of information:

- packet type, which here is always 1 for data
- `isbufferempty` - the flag indicating that the buffer is empty. Given that this is performed within an if statement, only if the buffer contains messages, this value should always be 0.
- `head` - head of the buffer
- `tail` - tail of the buffer

The `MyHeader` is then put into a packet and sent. If the buffer is empty, instead of creating a `MyHead` we just log that there is nothing to send.

Finally, whenever Node0 receives an acknowledgement from Node1 that it got the packet with messages, it clears the buffer by setting `isbufferempty` to 0.

1.2 Node 1

The code describing Node1's behaviour starts at line 298. It has three functions, `Node1ReceivePacket(...)`, `Node1AckLoop(...)`,and `Node1SendPacket(...)`.

The `Node1ReceivePacket(...)` picks up packets sent by Node0. If it's a data packet, it copies the packet's content into its own buffer, and sets `Node1SendAck` flag to 1. If it's not a data packet, then it is an acknowledgement from Node 2, which gets logged.

When `Node1SendAck` flag is set to 1, the `Node1AckLoop(...)` function, running every 0.01 s, sends the acknowledgement, unsets the `Node1SendAck` and sets the `Node1Pending` flag to 1. Finally it schedules the packet to be sent to Node2.

Once the acknowledgement is sent, `Node1SendPacket(...)` begins its attempts to send the packet to Node2, in 0.25 s intervals. It copies information stored in its buffer to a packet and tries sending it.

1.3 Node 2

Node2's role is to receive packets from Node1 and acknowledge it. Starting at line 373, the receiving of packets is covered by function `Node2ReceivePacket(...)`. It extracts the information from the header and stores them in local variables. Of a particular note is the `Pkt_no_last_seen_by_node2`. It keeps track of the last received packet by storing the stamp of the last message in the previous packet. If the arriving packet contains a message with a newer stamp, Node2 updates its records, and logs the values from the header. Finally it marks `Node2SendAck`, which will then be used to prompt an acknowledgement.

`Node2AckLoop(...)` runs at 0.01 s intervals. Every time it creates a packet, however it is sent only if `Node2SendAck` is marked as 1 by the previous function, after being set to type 0 for ack (cf. 1 for data). This function also logs that the acknowledgement is sent, and marks `Node2SendAck` back to 0.

1.4 Overall behaviour

All nodes are set to unicast mode, with Node0 connected to Node1, and Node1 connected to Node2. They all follow the 802.11 standard, and adhere to AODV protocol with route timeout of 10 min. Note that this is longer than the simulation, which runs for 500 s. The transmission power remains constant at 1.5 dBm throughout the simulation.

1.5 Propagation Loss Model

The propagation loss model is constant range, which means that up to the given maximum distance (200 m initially) the packets are transmitted with the given transmission power (1.5 dBm in this case). Beyond the maximum range, the transmission power drops to -1000 dBm, which is effectively 0 [1]. Note that with Node1 travelling along the $y = 3$ line, it means that it needs to be at $x \leq 199.98 \approx 200$ to reach Node0, and at $x > 9800$ to reach Node2.

2 Methods

2.1 Task 1A

To introduce movement of Node1, we first add `"ConstantVelocityMobilityModel"` using the Mobility Helper that already exists in the code. It is done by adding the following line at line 476:

```
mobilityMobileNode.SetMobilityModel("ns3::ConstantVelocityMobilityModel");
```

After installing the Mobility Helper, we explicitly set the velocity of Node1, by:

```
(mobileNode.Get(0) -> GetObject<ConstantVelocityMobilityModel>()) -> SetVelocity
(Vector(20.0, 0.0, 0.0));
```

This is a much more certain way of setting the velocity of the node, than setting it as an attribute while setting the Constant Velocity Mobility Model, which sometimes may not be parsed correctly.

Note that with the transmission range of 200 m, even with Node1 moving, its buffer gets overwritten multiple times, before it is out of range of Node0.

To work out the maximum distance between Node0 and Node1 for a given speed of Node1, we iterate over Node1's speeds, ranging from 20 to 200 m/s, at 10 m/s intervals, and over distances, ranging from 3 to 300 m. We do not expect to see anything at a range exceeding 200 m because of the propagation loss model.

In each iteration, we calculate the duration of the simulation by the following formula:

$$\text{duration} = \frac{\text{Distance between Node0 and Node1}}{\text{speed of Node1}} + 10$$

We add 10 at the end to make sure that Node1 goes past the x coordinate of Node2.

Our findings are summarised in the graph in Fig 2.

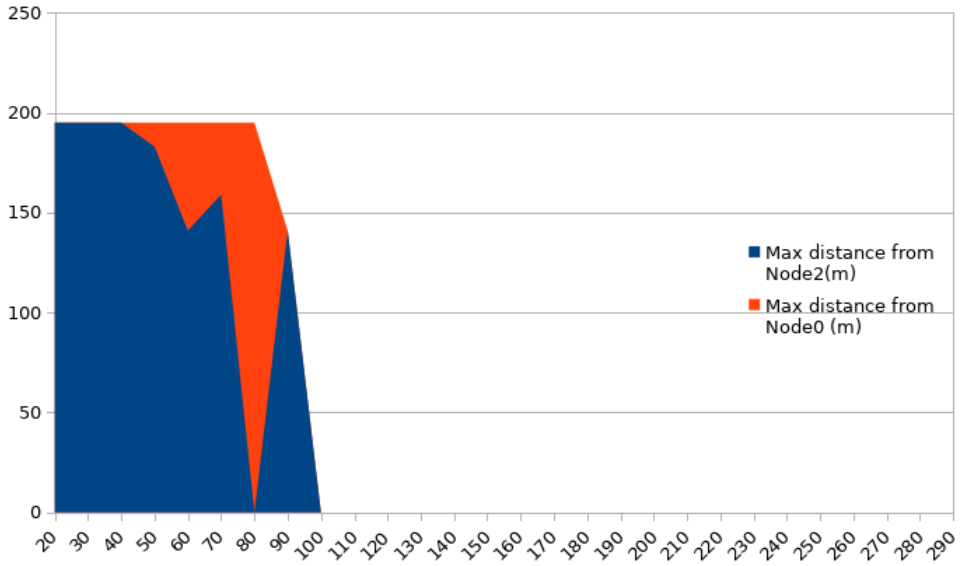

Figure 2: Relationship between the speed of Node1 and the maximum distance along the y axis between Node0 and Node1, for which transmission occurs.

2.2 Tast 1B

Now we implement the Friis propagation loss model. It takes 3 parameters: frequency, which we are told to fix at 2.4 GHz, and System Loss and Minimum Loss, which we are leaving at 1 and 0 dB

respectively. Recall that Friis model involves a relationship between the loss and the distance between nodes which is quadratic. I.e. the loss is proportional to the square of the distance.

It is done by replacing the three lines starting at line 459 with the following:

```
std::string lossModel = "ns3::FriisPropagationLossModel");
std::string atr1 = "Frequency";
wifiChannel.AddPropagationLoss (lossModel, atr1, DoubleValue(2400000000));
2.4 GHz = 2400000000 Hzs
```

The maximum distance away from Node0 and Node2 as a function of speed of Node1 is shown in Fig. 2.s

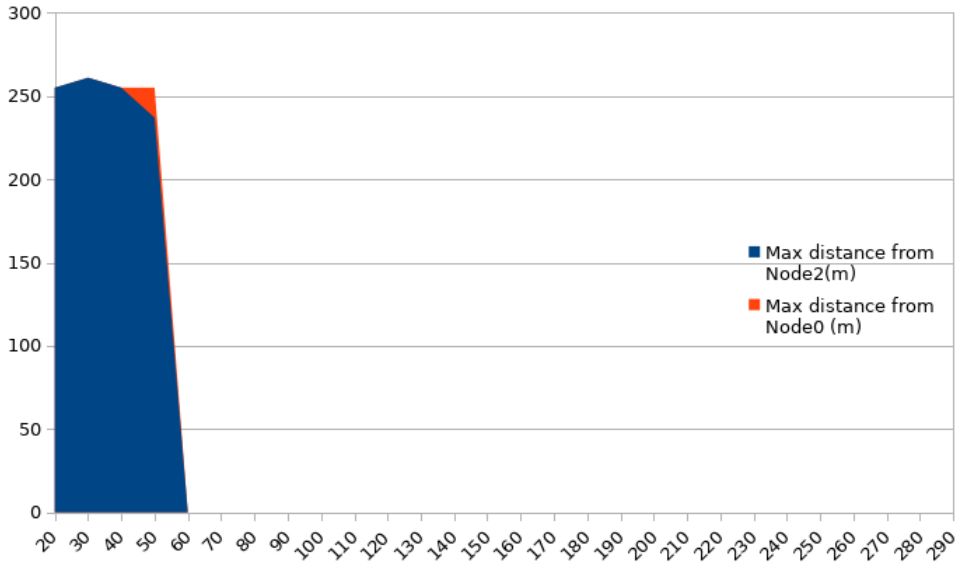

Figure 3: Relationship between the speed of Node1 and the maximum distance along the y axis between Node0 and Node1, for which transmission occurs.

2.3 Task 2A

To count the number of mails received at Node2, we introduce a global variable, `Number_mails_rcvd_at_node2` at the start of the program, and set it to 0. Then in the `Node2ReceivePacket(...)` function we update it after parsing the message:

```
Number_mails_rcvd_at_node2 += receivedpacket_head - (receivedpacket_tail -
1)
```

And add it to the logging:

```
std::cout << "The total number of mails received so far is " << Number_mails_rcvd_a
<< std::endl;
```

The results are as shown in Fig. 4. Throughout the simulation, we consistently receive 1 packet, regardless of the speed of Node1 and distance from Node0, up to the transmission limits shown before. While disappointing, it is understandable, once we inspect contents of the Node1's buffer at each point.

Node0 sends its data across every 0.25 s, while generating a new message every second. So it first sends its message 1, receives acknowledgement from Node1, purges its buffer, and then creates message number 2, which is now the only message in the buffer.

Meanwhile Node1 keeps replacing the content of its own buffer with the packet received from Node0. Each of these packets contain only a single message, and so Node1's buffer can contain at most 1 message at a time in this scenario. Thus there's only one message sent to Node2 in each iteration.

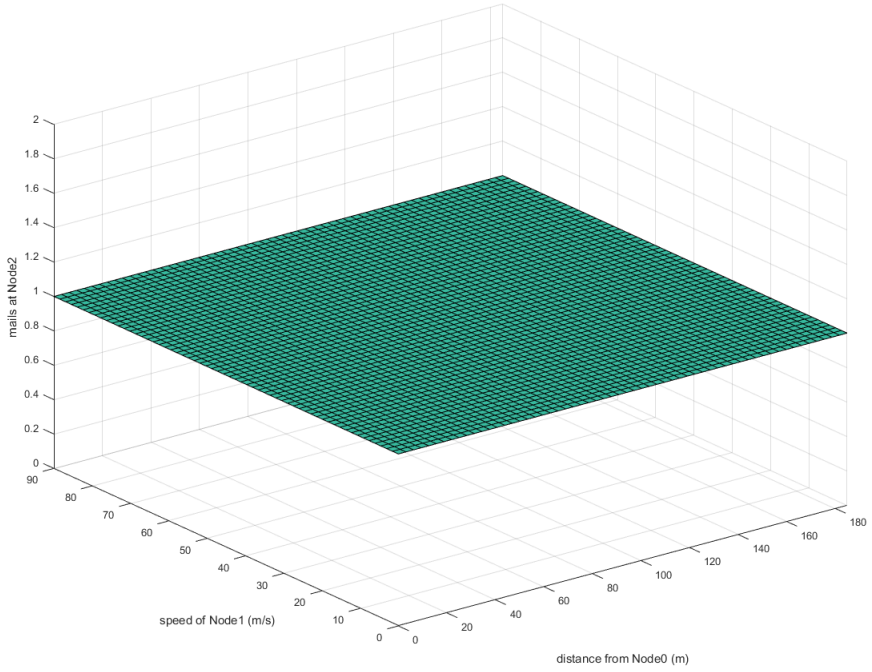

Figure 4: Number of packets received at Node2

2.4 Task 2B

First note that in order to keep all messages in the buffer, we should not simply store the value of the tail from the first data packet we receive, and keep updating the head. It is possible that we do not receive all messages between them. For example we may receive the first buffer load of Node0 with messages 1, 2, and 3, and the next packet may contain messages 5, 6, and 7. Thus we don't get message number 4, while keeping only one tail and head would imply otherwise. However in this simple simulation, this will not cause problems.

This was tested by logging the contents of each packet received by Node1.

Thus, we update Node1's tail only once, with the first packet received. To this end we survive 342 with an if statement:

```
if (Pkt_no_last_seen_by_node1 == 0) {
    node1_tail = header.Gettail();
}
```

We also need to make sure that we update the variable `Pkt_no_last_seen_by_node1` after that if statement, so we simply move line 334 further down.

This time we can clearly see the number of messages decreasing as both speed of Node1 and distance from Node0 increases, because Node1 leaves the transmission range of Node0 quicker.

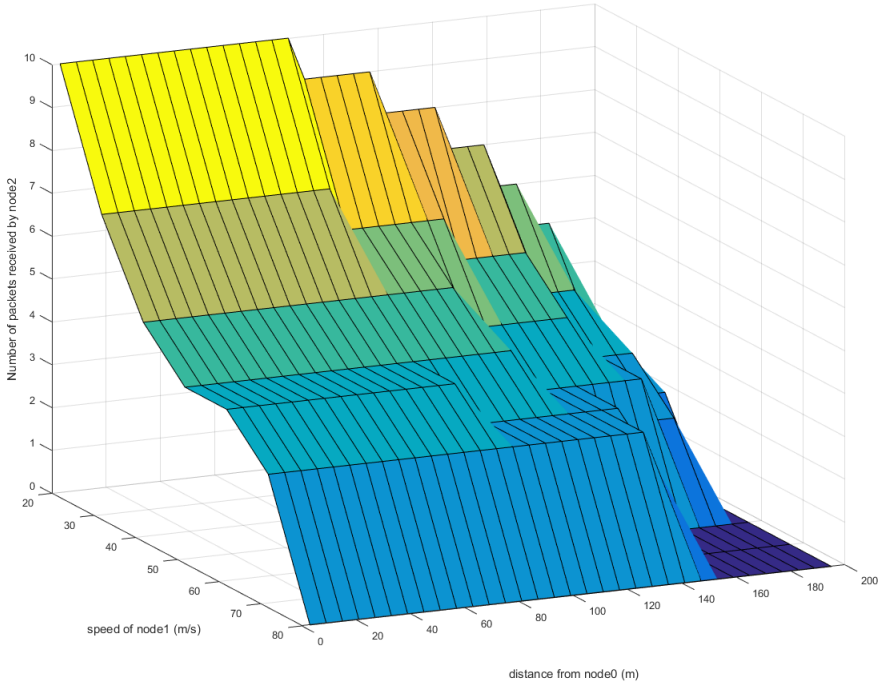

Figure 5: Number of packets received at Node2

2.5 Task 3A

References

[1] Range Propagation Loss Model, NS-3 documentation, available online: https://www.nsnam.org/doxygen/classns3_1_1_range_propagation_loss_model.html