Sensor Networks and Mobile Data Communication, Assignment 4

UID: 1690550

March 11, 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.

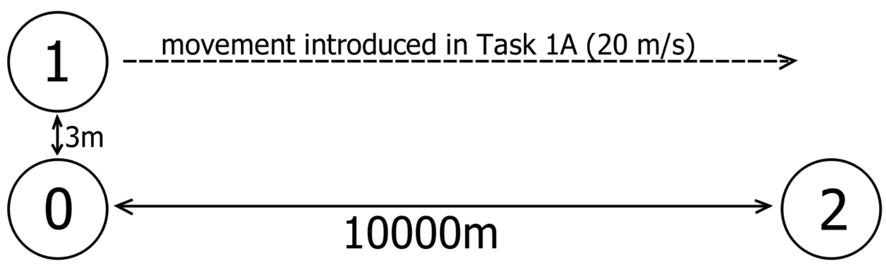

Figure 1: Initial topology of the network, with the movement as introduced in the first task

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.

References

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