Introduction

Goal: Using a network simulator written in Python, implement reliable packet transport for a best-effort network that drops packets.

Pre-lab (read before 2pm, Mon., May 7, 2007)

As we discussed in lecture, packets can be lost by the network for a number of reasons. A “best-effort” network layer is simple and has a number of attractive properties, but from the point of view of an application trying to reliably transmit a stream of data from one node to another, the network could “do more”. A reliable, in-order transport mechanism is invaluable.

The goal of this lab is to implement a reliable data stream on top of the sometimes flaky best-effort packet delivery by adding the appropriate code in the `transmit` method of the sending node and the `receive` method of the receiving node. We’ll implement a transport protocol where the receiver sends an ACK packet to the sender for each DATA packet received. If an excessive amount of time passes without receiving an ACK packet, the sender assumes that the corresponding DATA packet has been lost (in truth, either the sender’s DATA packet or the receiver’s ACK packet was lost, but the sender can’t tell which exactly), and retransmits the DATA packet. At some point both the DATA packet and corresponding ACK packet are successfully received and that portion of the data stream has been reliably delivered.

We describe below how a simple reliable transport protocol can be implemented. The description separates the sender and receiver functions. We also note that many optimizations to this reliable transport protocol are possible, but we’ll just take the straightforward approach for this lab.

Lab setup

First, copy our Lab 11 files to your local Athena home directory:

```
  athena% setup 6.082
  athena% cp -R /mit/6.082/Labs/Lab11 ~/6.082/
```

You should now have three files: `lab11_stop_and_wait.py`, `lab11_sliding_window.py`, and `network11.py` in your Lab11 directory. The following discussion will refer to the code contained in these files.

We will be using the same simulator you have become familiar with in Lab 10, and we will use Python inheritance to extend class `Node` with added functionality for path-vector routing, end-to-end reliable sending, and end-to-end reliable receiving.
Reading the code

First, we extend the Node class to PathVectorNode as shown in Figure 1. The route method of PathVectorNode uses a routing table built using the path-vector protocol described in lecture. The code for building the routing tables has been added to the process and transmit methods of PathVectorNode, in network11.py. You should not need to make further changes to PathVectorNode.

Next, we will extend the PathVectorNode class to implement end-to-end reliable sending and receiving functionality (ReliableSenderNode and ReliableReceiverNode, respectively). We have started the implementation of ReliableSenderNode and ReliableReceiverNode; you will be responsible for finishing the code for a stop-and-wait protocol lab11_stop_and_wait.py (in lecture we called this protocol “Take 1”) and a sliding window protocol lab11_sliding_window.py (“Take 3” in lecture).

Network

In network11.py we have implemented a test framework for building your implementation of a reliable transport protocol. The network has the same topology as was used for Lab 10. However, node D has been given a “specified recipient” of E using the ReliableSenderNode instance variable stream_destination, so that after time 100 it will start to send a packet stream to node E. You should not need to change this part of the code.

The links in the network are instances of LossyLink, implemented in lab11_stop_and_wait.py and lab11_sliding_window.py. This type of link drops packets with probability PLOSS. Setting PLOSS to 0 causes the link to deliver all packets without loss. LossyLink prints a message each time a packet is dropped.

Packets

To create a new packet for transmission at the sender, you may call the make_packet method on the network field of Node objects (and their subclasses). The API for make_packet is:

```
make_packet(source_address, dest_address, time, properties...)
```
All data packets and acknowledgements you send should have three properties: seqnum, ptype, and color. The value of the ptype property should be DATA for DATA packets and ACK for ACKs. The value of the color property should be ’black’ for DATA packets and ’red’ for ACK packets. The seqnum property should be the correct sequence number of the packet. You will use the sequence number to uniquely identify the packet to the receiver, and to the sender when the receiver sends an ACK. The time at which each DATA packet was transmitted should also be noted.

We have added a simple “application” to ReliableReceiverNode in the form of a app.receive method. You should call this method with incoming DATA packets in sequence order (i.e., no duplicates or out-of-order packets should be delivered to the application). app.receive checks the sequence numbers and complains if they are not exactly in-order. The initial behavior of the receive method is to call app.receive for each arriving DATA packet without any pre-processing at all.

The ReliableSenderNode.transmit method has code that sends a stream of DATA packets to the specified recipient (only one node in the network will have a recipient specified). Note that it waits until time 100 before starting to transmit the packet stream: this delay gives sufficient time for the routing tables to be constructed. The initial behavior is to send one DATA packet per unit time, each with an incrementing sequence number. No other bookkeeping is initially done, so this behavior will only be correct if no packets are lost.

**Pre-lab tasks**

1. Read the instructions below for what you will be doing in the lab.

2. Read the code and get familiar with it.

**In the lab (2pm–5pm, Wed., May 9, 2007)**

To start, read lab11_stop_and_wait.py into IDLE and set the value of PLOSS to 0 in the class definition for LossyLink. Run the simulation, clicking on “Step 100” until the time reaches 1000. You should see a stream of blue packets making their way from node D to node E. Now click on node E and it will report the number of DATA packets it has received, along with the average transfer rate in packets per unit time. Since the network is operating perfectly it will report a transfer rate of 1.00.

Quit the simulation and restore the value of PLOSS to its original value of 0.01 (causing each link to drop 1% of its packets). Now run the simulation again until time 1000. At some point you’ll see error messages as the application in node E reports receiving duplicate or out-of-order packets:

```
$ python lab11.py
Dropping packet Packet<D to E>: seqnum=14, type=DATA
*BUG* in app_receive: Expected DATA packet #14, got #15
Dropping packet Packet<D to E>: seqnum=21, type=DATA
*BUG* in app_receive: Expected DATA packet #21, got #22
Dropping packet Packet<D to E>: seqnum=61, type=DATA
*BUG* in app_receive: Expected DATA packet #61, got #62
```
These messages indicate that a packet has been lost and the simple initial behavior doesn’t implement the necessary reliable transport mechanisms.

**Step 1: a stop-and-wait protocol**

We will begin with the stop-and-wait protocol you saw in lecture (“Take 1”). In this protocol, the sender waits for the receiver to acknowledge each packet before sending the next. We will now add code to `lab11_stop_and_wait.py` to implement the stop-and-wait protocol.

**At the sender** *(ReliableSenderNode)*

The sender will maintain the following state: *xmit*-*packet*, the packet currently in-flight (None if no packet is in-flight), *xmit*-*seqnum*, the sequence number of the in-flight packet, and *last*-*xmit*-*time*, the time the in-flight packet was sent. We have added code to *init* to initialize state and *reset* to reset the state.

1. In *transmit*, send the next packet, if there is no packet in-flight towards the receiver. If an acknowledgement has not been received for a packet within some specified time interval (called the *timeout* interval), retransmit the packet using its original sequence number. Update the time of transmission. Your transmit code should be retransmit any packets that remain unacknowledged for self.TIMEOUT time units.

2. In *receive*, process incoming ACK packets by updating the state and preparing for the next transmission. Note that it’s possible to get an ACK for a packet that’s already been acknowledged; just ignore these redundant ACKs.

Please have your code print out a message whenever it transmits or retransmits a packet. If your code is operating correctly, these messages should appear interleaved with the dropped packet messages printed out by LossyLink.

**At the receiver** *(ReliableReceiverNode)*

The receiver maintains one piece of state: *rcv*-*seqnum*, the sequence number of the last packet that was delivered to the application via app_receiving. You should implement the following at the receiver.

In *receive*, send an ACK packet to the sender for each arriving DATA packet, **if and only if** the sequence number of the newly-arriving packet is the next in-order sequence number to be delivered to the application. The acknowledgement should include the sequence number of the DATA packet so that the sender will know which packet is being acknowledged. Remember to call *self*.*app*.*receive* whenever there’s an in-order DATA packet that should be delivered to the application.

**Question:** Can you think of a reason why we need to send the sequence number in the ACK even in a stop-and-wait protocol, where the sender sends packet K+1 only after the ACK for packet K is received?

If your code is operating correctly, lost DATA packets will be retransmitted after the timeout interval, eliciting ACKs. After running the simulation for a 1000 time units, click node E and see what it reports for the average

---

4
transfer rate. Since the packets have to travel over five links, the round trip time for DATA/ACK pair is ten time units, so the best possible transfer rate is 0.1 packets per unit time.

**Question:** What would you expect the reported rate to be? Why?

Demonstrate your working code to a staff member.

**Step 2: a sliding window protocol**

We will now modify your code from Step 1 to allow the sender to transmit additional DATA packets without waiting for acknowledgements. The number of additional transmissions allowed is set by the `self.WINDOW` parameter. Setting `self.WINDOW` to 1 is equivalent to the protocol in Step 1. Once the first ACK has been received an additional DATA packet should be sent, and so on, with a total of `self.WINDOW` unacknowledged packets outstanding at any given time. Open `lab11_sliding_window.py` and follow the steps below.

Note that packets which remain unacknowledged for `self.TIMEOUT` time units should be retransmitted, just as before. Retransmission of old packets should take precedence over transmission of new DATA packets. **Question:** Why?

Also note that due to packet losses, packets may be acknowledged out-of-order, a circumstance your code should handle correctly.

**At the sender (ReliableSenderNode)**

At the sender, we will keep the following state. `xmit_seqnum` maintains the sequence number of the last transmitted packet. The sender also keeps a list of sent-but-unacknowledged packets in `unacked_packets`.

1. In the `transmit` method, if an acknowledgement has not been received for a packet within some specified timeout interval, add code to retransmit the packet using its original sequence number. Update the time at which the transmission occurred.

2. Also in the `transmit` method, if fewer than `self.WINDOW` packets are in-flight at any time, you should add code to send a data packet with the sequence number following the last data packet sent.

3. In the `receive` method, add code to process incoming ACKs by removing acknowledged packets (i.e., packets that have the matching sequence number) from the list created in step 2 above. Note that it's possible to get an ACK for a packet that's already been removed from the list (this happens when packets get delayed in the network and the sender has retransmitted unnecessarily); your code should just ignore these redundant ACKs.

**At the receiver (ReliableReceiverNode)**

Because packets may arrive at the receiver out-of-order, the receiver will buffer the packets it receives, and pass them to `app.receive` in order of sequence number. To accomplish this task, the receiver maintains a list `buffered_packets`.

Complete the following steps in the `receive` method of `ReliableReceiverNode`:
1. Write code to send an ACK to the sender for each arriving DATA packet. The acknowledgement should include the sequence number of the DATA packet so that the sender will know which packet is being acknowledged.

2. Write code to add the incoming packet to the list of buffered packets, in order of sequence number.

3. Use rcv_seqnum to remember the sequence number of the last DATA packet to be delivered to the application running on the receiver.

   (a) If the arriving DATA packet is the next packet in the sequence (i.e., has a sequence number equal to one more than the last delivered packet’s sequence number), deliver it to the application and update the instance variable. If there are any saved packets (see step (c) below), see if one or more of them can now be delivered in appropriate sequence order.

   (b) If the arriving DATA packet has a sequence number less than or equal to that of the last delivered packet, the arriving packet is a duplicate and should be discarded (but its receipt will be acknowledged in step 1 above). Duplicate packets may arise in the simulation when an ACK gets lost or delayed and the sender retransmits the packet because it didn’t receive an ACK in time.

   (c) If the arriving DATA packet has a sequence number greater than that of the last delivered packet plus one, an earlier DATA packet has been lost or delayed. The arriving packet should be saved for delivery at some later time after the missing packet is retransmitted, arrives at the receiver and is delivered to the application.

Once your code is working, run the simulation for 1000 time units and then click on node E to see your average transfer rate. Demonstrate your working code to a staff member.

Congratulations: you have now completed the lab! Don’t forget to comment your code for Step 2 and hand it in at the beginning of Friday’s lecture.

Post-lab (due at the start of lecture, 2pm, Fri., May 11, 2007)

For the sliding window protocol, if the value of self.WINDOW is larger than the round-trip time for a DATA/ACK packet pair then there will be a pipeline of packets following along the links between D and E, and the average transfer rate should once again approach 1.00 if there are no packet losses.

With lossy links that drop 1% of their packets, what average data transfer rate would you expect between nodes D and E?

Please hand in a listing for your updated MyNode class showing your solution for Step 2. Include enough comments that we’ll be able to tell what your code is doing.