Problem Set 9

Your answers will be graded by actual human beings (at least that's what we believe!), so don t limit your answers to machine-gradable responses. Some of the questions specifically ask for explanations; regardless, it s **always a good idea to provide a short explanation for your answer**.

Most of this PSet covers the material in Chapter 19. Some questions may require you to apply Little s law from Chapter 16 (§16.4).

Problem 0.

Give the date, time, and location of Quiz 3 in the space below. If you have been given a conflict date and location by the registrar, give that information below.

(points: 1)

Problem 1. Alyssa P. Hacker sets up a wireless network in her home to enable her computer ("client") to communicate with an Access Point (AP). The client and AP communicate with each other using a stop-and-wait protocol.

The data packet size is 10000 bits. The total round-trip time (RTT) between the AP and client is equal to 0.2 milliseconds (that includes the time to process the packet, transmit an ACK, and process the ACK at the sender) **plus** the transmission time of the 10000 bit packet over the link.

Alyssa can configure two possible transmission bit rates for her link, with the following properties:

Bit rate	Bi-directional packet loss probability	RTT
10 Megabits/s	1/11	___
20 Megabits/s	1/4	___

Alyssa s goal is to select the bit rate that provides the higher throughput for a stream of packets that need to be delivered reliably between the AP and client using stop-and-wait. For both bit rates, the retransmission timeout is 3 milliseconds.

A. Calculate the round-trip time (RTT) for each bit rate?

(points: 1)

B. For each bit rate, calculate the **expected time**, in milliseconds, to successfully deliver a packet and get an ACK for it. Show your work.

(points: 1)

C. Using the above calculations, which bit rate would you choose to achieve Alyssa s goal?

(points: .5)

Problem 2. Fritter

Energized by Twitter s success, Alyssa P. Hacker starts Fritter, a service that thinks that 140-byte Twitter messages are 100 too many for the next generation: Fritter messages are only 40 bytes long. Fritter has a simple request-response interface. The client sends a request (called a frequest, which Fritter s Marketing person promptly trademarks), to which the server sends a response (a fresponse, also trademarked). The frequest fits in one 40-byte packet, as does the fresponse. When the client gets a fresponse, it immediately sends the next frequest.

Alyssa s server is in Cambridge. Clients come from all over the world. Alyssa s measurements show that one can model the typical client as having a 100 millisecond round-trip time (RTT) to the server.

Each client connects to Fritter, sends some number of unique frequests, and after some

period of frittering (their time) away, leaves. If a client does not get a fresponse from the server to a particular frequest in a timeout duration T, it assumes (correctly) that either the frequest or fresponse were lost, and resends the frequest. It keeps doing that until it gets a fresponse.

A. Alyssa needs to provision the link's transmission rate for Fritter. She hopes to be wildly successful, and anticipates that at any given time, the largest number of clients making frequests is 60000. What minimum outgoing link rate from Fritter will ensure that all the fresponses for one round of frequests will be delivered before the next round of frequests starts to arrive? Give your answer in *megabits* per second.

(points: 0.5)

B. If the data rate from the server to the client is C = 10000 bytes/s and the RTT (as before) = 100 milliseconds, what is the smallest number of outstanding (i.e., awaiting a fresponse) frequests that the client should make to achieve the highest possible throughput? Assume that no packets are lost.

(points: 1.0)

Problem 3.

Ben Bitdiddle gets rid of the timestamps from the packet header in the 6.02 stop-and-wait transport protocol running over a best-effort network. The network may lose or reorder packets, but it never duplicates a packet. In the protocol, the receiver sends an ACK for each data packet it receives, echoing the sequence number of the packet that was just received. The sender uses the following method to estimate the round-trip time (RTT) of the connection:

1. When the sender transmits a data packet with sequence number k, it stores the time on its machine at which the packet was sent, t_k. If the transmission is a retransmission of sequence number k, then t_k is updated.
2. When the sender gets an ACK for packet k, if it has not already gotten an ACK for k so far, it observes the current time on its machine, a_k, and measures the RTT sample as $a_k - t_k$.

If the ACK received by the sender at time a_k was sent by the receiver in response to a data packet sent at time t_k, then the RTT sample $a_k - t_k$ is said to be correct. Otherwise, it is incorrect. State whether the following three statements are true or false, **explaining your answer**.

A. If the sender never retransmits a data packet during a data transfer, then all the RTT samples produced by Ben s method are correct.

(points: 1)

B. If data and ACK packets are never reordered in the network, then all the RTT samples produced by Ben s method are correct.

(points: 1)

C. If the sender makes no spurious retransmissions during a data transfer (i.e., it only retransmits a data packet if all previous transmissions of data packets with the same sequence number did in fact get dropped before reaching the receiver), then all the RTT samples produced by Ben s method are correct.

(points: 1)

Problem 4.

Carrie Coder has set up an email server for a large email provider. The email server has two modules that process messages: the *spam filter* and the *virus scanner*. As soon as a message arrives, the spam filter processes the message. After this processing, if the message is spam, the filter throws out the message. The system sends all non-spam messages immediately to the virus scanner. If the scanner determines that the email has a virus, it throws out the message. The system then stores all non-spam, non-virus messages in the inboxes of users. Carrie runs her system for a few days and makes the following observations:

- On average, M messages arrive per second.
- On average, the spam filter has a queue size of N_s messages.
- A fraction s of all email is found to be spam; spam is discarded.
- On average, the virus scanner has a queue size of N_v messages.
- A fraction v of all non-spam email is found to have a virus; these messages are discarded.

A. On average, in T seconds, how many messages are placed in the inboxes?

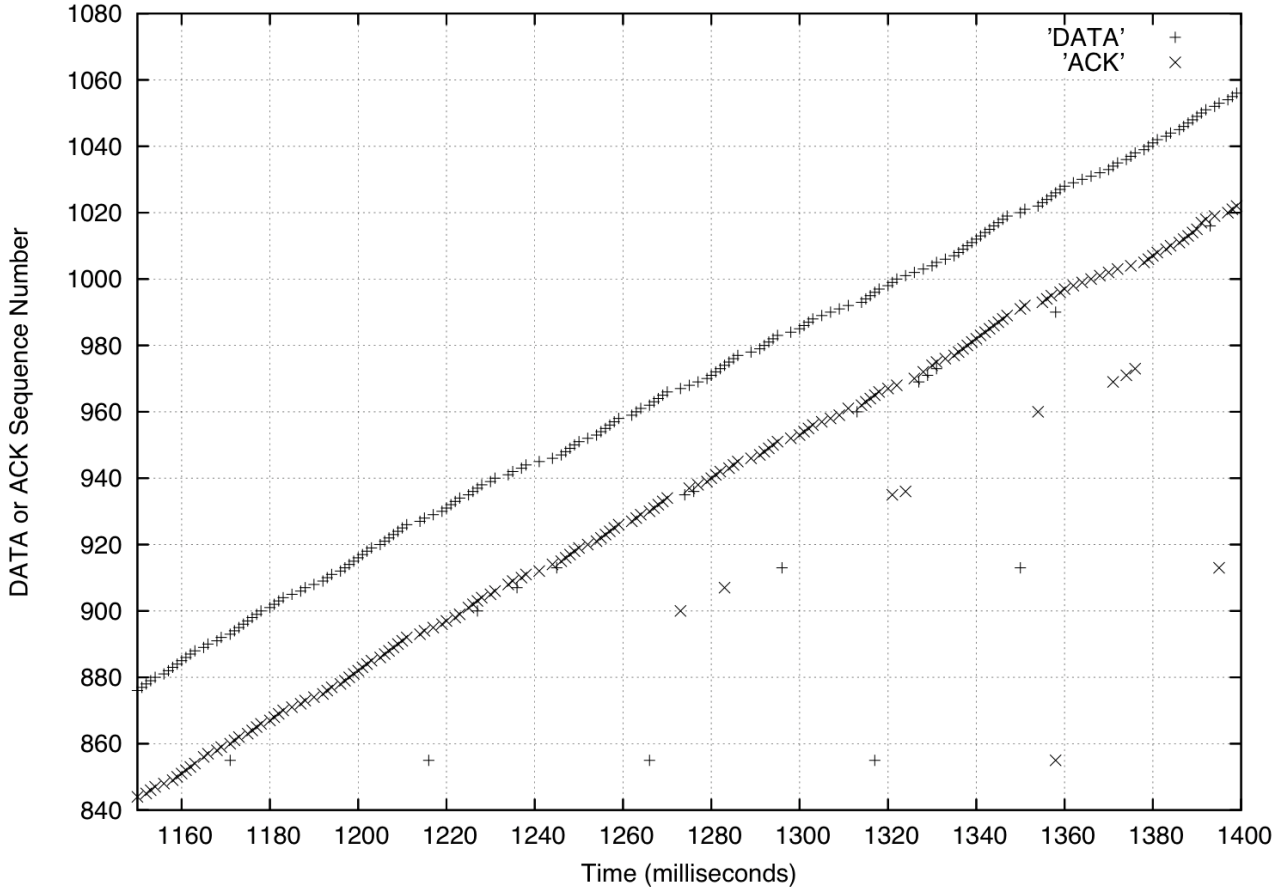
(points: 0.5)

B. What is the average delay between the arrival of an email message to the email server and when it is ready to be placed in the inboxes? All transfer and processing delays are negligible compared to the queueing delays.

(points: 1.5)

Problem 5.

Annette Werker correctly implements the fixed-size sliding window protocol described in Chapter 19. She instruments the sender to store the time at which each DATA packet is sent and the time at which each ACK is received. A snippet of the DATA and ACK traces from an experiment is shown in the picture below. Each + is a DATA packet transmission, with the x-axis showing the transmission time and the y-axis showing the sequence number. Each x mark is an ACK reception, with the x-axis showing the ACK reception time and the y-axis showing the ACK sequence number. All DATA packets have the same size.



A. Estimate the sender's window size, in packets. Explain your answer.

(points: 0.5)

B. Estimate the throughput of the connection in packets per second. Explain your answer.

(points: 1)

C. Considering only sequence numbers larger than 880, estimate the packet loss rate experienced by DATA packets? Explain your answer.

(points: .5)

Problem 6.

The bottleneck (i.e., slowest) link on a network path between two computers, A and B, has a bit rate of 10 Megabytes/s, and the round-trip time (RTT) between the computers is 50 milliseconds. The queueing and processing delays are negligible. You run a sliding window protocol between the two computers to reliably send data, with a data packet size of 1000 bytes and a window size of 80 packets.

A. What is the maximum utilization of the bottleneck link along this path assuming no lost data packets or ACKs?

(points: 0.5)

B. For each of the following techniques, explain if the technique will double the utilization of the bottleneck link. Assume that no packets are lost. Explain your answer for each of the four proposed techniques.

A. Double the window size.
B. Double the speed of the bottleneck link.
C. Halve the speed of the bottleneck link.
D. Halve the RTT of the network path.

(points: 2)

Problem 7

Annette Werker conducts tests between a server and a client using the 6.02 sliding window protocol. There is no other traffic on the path and no packet loss. She finds that:

1. With a window size W1 = 50 packets, the throughput is 200 packets per second.
2. With a window size W2 = 100 packets, the throughput is 250 packets per second.

Annette finds that even this small amount of information allows her to calculate several things, assuming there is only one bottleneck link. Calculate the following, explaining your answers.

A. The minimum round-trip time between the client and server.

(points: .5)

B. The average **queueing** delay at the bottleneck when the window size is 100 packets.

(points: .5)

C. The average queue size when the window size is 100 packets.

(points: 1)

Problem 8.

A transport protocol for delivering interactive video does not need all the transmitted packets to arrive at the receiver application. Missing packets are acceptable, but the packets must be delivered to the application by the transport protocol in **increasing sequence order**. One way to achieve this goal is to **remove all retransmissions from the 6.02 sliding window protocol**, so now the sender just sends data packets with incrementing sequence numbers at some constant rate. Packets may show up in arbitrary order, be delayed arbitrarily, and be

duplicated in the network. We would like to apply the following rules at the receiver transport layer:

1. Deliver packets to the application only in strictly increasing sequence order (perhaps skipping some).
2. Every t milliseconds, deliver exactly one packet to the application. This packet should be the one with the smallest sequence number in the receiver s buffer that is strictly greater than the one previously delivered to the application. You may assume that such a packet always exists in the receiver s buffer.

Suppose the receiver transport protocol receives the following packets at the times shown:
Time (ms): 0, 8, 9, 13, 14, 19, 44, 48, 56
Packet: 5, 3, 1, 2, 10, 8, 5, 7, 4

A. Let t=10 ms. The receiver transport delivers the first packet to the video application at time 10 ms. Write the time and packet sequence at which various packets are delivered to the application.

(points: .5)

B. In the space provided below, write Python-like code to implement the receiver transport protocol according to the rules above.

Write two functions: **receive** and **deliver2app**. The specifications of these functions are given below. Name variables descriptively, or define them.

```
'''
receive() is called whenever a data packet arrives. Maintain
self.rcvbuf, a buffer of packets that haven't yet been delivered
to the app. Delete packets that will never be delivered to the app
from rcvbuf. You may use Python's library modules and functions,
e.g., foo.sort() if you wish to sort list foo in ascending order.
'''
def receive(self, pkt):
    # Notation: pkt.seq is the sequence number of pkt
    # Your code here

'''
Assume that deliver2app is called every t seconds by the reciver
transport protocol's main loop. Call app_receive(pkt) to deliver
a pkt from self.rcvbuf to the app, following rules R1 and R2.
'''
def deliver2app(self):
    # Your code here
```

(points: 2)

Reliable Transport: Introduction to Python tasks

Useful download link:

PS9.zip

In this lab, you will develop the core logic of the `ReliableSenderNode` and `ReliableReceiverNode` classes. These two classes implement the functions of a *reliable data transport protocol* between sender and receiver that delivers packets *reliably and in order* to the receiving application. The sender and receiver are connected over one or more network hops via nodes of the Router class. You don't have to worry about how routing and forwarding work in this lab.

This lab has two tasks. You will write the code for the main components of a *stop-and-wait protocol* and a *sliding window protocol* (with fixed window size). Your code will involve both the sending and receiving sides of the protocol.

You can run the Python programs for this lab using the python command line; this lab may not work in IDLE.

To understand the different parameters in NetSim for this lab, start a terminal (command line) application and enter:

```
# python PS9_1.py -h

# python PS9_2.py -h
```

The programs take the following options:

- `-l LOSS_PROB` causes both data and ACK packets to be lost on *each link* with the specified `LOSS_PROB`, which must be a number between 0 and 1. The default per-link loss probability is 0.01.
- `-b BOTTLENECK_RATE` specifies the rate of the bottleneck link between sender and receiver, in packets per time slot. It should be between 0 and 1 (default is 1).
- `-q QUEUE_SIZE` specifies the queue size at each node for each link (default is 10 packets).
- `-g` runs the program with the GUI, useful for debugging.
- `-t SIMTIME` sets the number of time slots in the simulation (default is 5000).
- `-v` runs the program in "verbose" mode; in this mode, the program prints out a line whenever the sender gets an ACK, the receiver application gets a packet, or a packet is dropped on a link.

In addition, for the sliding window protocol, the following option is important:

- `-w WINDOW_SIZE` sets the window size (in number of packets); the maximum number of *unacknowledged* packets sent by the sender must not exceed this window size setting.

This lab has two main tasks that implement two different reliable transport protocols. Each task has a few sub-tasks. You will test these implementations on the test topology that will be generated when you run the corresponding task files.

Debugging and Testing Procedures

At any point in the simulation, when you run the programs with the `-g` option, you can click on the sender, node S, to see an estimate of the round trip time (RTT) to the receiver (R) and the current timeout being used by the sender. Clicking on R will show the current throughput

at the receiver measured at the number of useful (i.e., unique and in-order) packets passed up to the application per time slot, as well as the total number of spurious (duplicate) packets received. We will judge the quality of your transport protocol by the throughput printed at the end of the simulation, and also by the RTT measurements displayed at the end.

If your protocols work correctly, two things should happen:

1. *No deadlocks.* Neither the receiver nor the sender should "hang" -- i.e., the protocol should not deadlock with both sides waiting for something to happen.
2. *In-order delivery.* The receiver should not print an error message and terminate the program. That will happen if your protocol delivers an out-of-sequence packet to the receiving application in the `app_receive` call, as explained below.

If the protocol is correct (i.e., no deadlocks and only in-order delivery), then the only potential problems that might remain are performance problems: the protocol may be unacceptably slow or unexpectedly sluggish. Obviously, we want your protocol to come as close as possible to the theoretically expected performance. The performance metric of interest is the *receiver throughput, measured in packets per time slot*. By default, the receiver throughput is printed by the programs only at the end of the simulation, but if you use the `-v` (verbose) option, it will be printed roughly every 100 time slots (more precisely, it is printed the first time the receiver application gets a new in-order packet at least 100 time slots since the last such event). Note that if you want to extract only the throughput numbers in the verbose mode, you can use the `grep` utility, running a command such as:

```
python PS9_2py -w 12 -l 0.02 -v | grep throughput
```

In addition, you can, and probably should, initially run the program with the GUI on for debugging, and click on the sender (S) and receiver (R) in the GUI to view useful information about the performance of your implementation.

To help debug your code, you can print out your own debug statements whenever a significant event occurs at the sender or receiver. You can also see the total number of pending packets at various nodes on the bottom panel of the simulator; if this number is persistently in the hundreds, then very likely something is wrong, especially if the window size is much smaller than the number of pending packets.

You can use the `-l` option to test both the correctness and the performance of your protocols at different link loss rates. Note that both packets and ACKs will get lost, and the value set is the per-link packet loss probability.

Another test worth running (which we may do during check-off) is to introduce variable *cross-traffic* into the network. You can do that using the `-x` option, setting a value between 0 and 1 as the rate of the cross traffic on the bottleneck link. Note that cross-traffic will both take away from the link bandwidth available for your data transfer, and will make the round-trip times more variable.

In this lab, each data packet and ACK are the same size, 1 time slot long. Each link sends one packet per time slot. Hence the maximum possible throughput of the protocol is 1 packet per time slot; because of packet losses and cross traffic, you won t achieve that maximum, but your goal should be to maximize throughput while providing reliable, in-order delivery.

Please note: In both tasks below, please use the variable names `srtt` for the estimate of the smoothed RTT, `rttdev` for the estimate of the mean linear RTT deviation, and `timeout` for the sender s retansmission timeout value. These variables should be members of the `ReliableSenderNode` class. All these quantities will of course vary with time in your protocol, and you will write the code o maintain these values. By using the variable names as

mentioned here, the debugging and diagnostics information obtained when you click on the sender node will be correct (the diagnostics code assumes that these variables exist).

Note: This lab is best done either after reading Chapter 19.

Python Task #1: Stop-and-wait protocol

Useful download links:

PS9_netsim.py -- network simulator (modified for this lab)
PS9_1.py -- template file for this task

The stop-and-wait protocol works as follows:

- Each data packet includes a unique sequence number, derived from a counter that is incremented for each new data packet (see the lecture notes, § 19.2). Each data packet also includes a timestamp giving the time at which the data packet was transmitted by the sender. A data packet is saved by the sender until it receives an ACK from the receiver. You may use `p.start` to set and view the time at which a packet `p` was sent.

- In each time slot the sender's `reliable_send` method is called. This method decides what packet to send, if any. It has two choices:
    - If there is a saved data packet, the sender should check to see if `self.timeout` slots have passed since the transmission by comparing the current time with the timestamp of the saved packet. If `self.timeout` slots have passed, the sender should retransmit the saved data packet, using the old sequence number *but with a new timestamp* that indicates the time of the retransmission. If `self.timeout` slots have not passed, the sender should not retransmit the packet.

    - If there is no saved packet, the sender should increment the sequence number counter and send a new data packet, remembering to also save the data packet pending the receipt of an ACK.

- When a data packet arrives at the receiver, the receiver's **`reliable_receive`** method is called in the code provided to you. This method, which you will write, should send an ACK to the sender (i.e., to the address specified in the source field of the received packet). The ACK should include the sequence number and timestamp of the received data packet. The method should then deliver the data packet to the application by calling the `app_receive` method. Note that that if an ACK is lost, the receiver may receive the same packet more than once and must take steps to ensure only one copy of the packet is delivered to the application (see §19.2.2) Note that `app_receive` maintains the `self.app_seqnum` instance variable, which indicates the sequence number of the last packet packet to be delivered to the application. `self.app_seqnum+1` should be the sequence number of the next packet to be delivered to the application. This instance variable may be useful in your code; please feel free to use it. The "time" value passed to the call to `app_receive` should be the time at which the function is called, not the time at which the packet being passed to `app_receive` originally arrived (for out-of-order packets, the two times won't be the same).

- When an ACK arrives at the sender, the sender's `process_ack` method is called by the code provided to you. `process_ack` should clear the saved packet, indicating that the next call to `reliable_send` is free to send the next data packet. It should also call the `calc_timeout` method to compute the ACKed packet's round trip time and update the value of `self.timeout`, using the formulas described in §19.3 of the lecture notes (or

something better, if you can improve on that).

In this task, you will implement the following functions in `ReliableSenderNode`:

`reliable_send(self,time)`
: This function, invoked every time slot at the sender, decides if the sender should (1) do nothing, (2) retransmit the previous data packet due to a timeout, or (3) send a new data packet. `time` is the current network time measured in time slots. The function `send_pkt()` can be used to build and send a data packet -- see the definition of this function in `PS9_1.py` for details about how to call this function. Note that `send_pkt` also returns the packet it transmitted, so it can be saved until an ACK for it arrives.

`process_ack(self, time, acknum, timestamp)`
: The code provided to you invokes this function whenever an ACK arrives. `time` is the network time when the ACK was received, `acknum` is the sequence number of the packet being acknowledged, and `timestamp` is the sender s timestamp that is echoed in the ACK. This function must call the `calc_timeout` function described below, among other things.

`calc_timeout(self, time, timestamp)`
: This function should be called by your `process_ack` method to compute the most recent data packet's round trip time (RTT) and then recompute the value of `self.timeout`.

In `ReliableReceiverNode` please implement:

`reliable_recv(self, sender, time, seqnum, timestamp)`
: The code provided to you invokes this function at the receiver upon receiving a data packet from the sender. You can use the `send_ack` method to build and transmit the ACK packet.

Note: If you introduce additional instance variables in the sender or receiver, you should add the appropriate initialization code to the `reset` method for the class.

The template code we have provided has additional comments that you may find helpful; please read them.

After writing the required functions, run the protocol for a few different link loss rates. Observe the throughput; click on the sender and observe the RTT and timeout estimates. These observations will help you answer the lab questions for this task.

When your code is working, pleease submit it on-line:

Upload code for Task 1: Browse...

(points: 4)

Run the following (you may use the `-g` option if you wish):

```
python PS9_1.py -l .04
```

What is the throughput of your protocol? Looking at the reported srtt and timeout, can you come up with a model and explanation for why your throughput is as observed? Note that the **per-link** loss rate in this experiment is 0.04, and that the number of hops in the topology is 12.

(points: 1)

If we reduced the number of hops between sender and receiver, and kept everything else unchanged, would the throughput increase or decrease? Explain your answer.

(points: 1)

If we changed the timeout to be a little smaller than the RTT, would the throughput increase or reduce? Explain your answer.

(points: 1)

Python Task #2: Sliding window protocol

Useful download links:

PS9_2.py -- template file for this task
PS9_runsliding.py -- script file for questions
PS9_prac-theory.py -- python plotter for questions

The sliding window protocol extends the stop-and-wait protocol by allowing the sender to have multiple packets outstanding (i.e., unacknowledged) at any given time. In this protocol, the maximum number of unacknowledged packets at the sender cannot exceed its window size, and is specified on the command line of the program with the `-w` option (default is 1). The window size is available as `self.window`.

Upon receiving a packet, the receiver sends an ACK for the packet's sequence number as before. The receiver then buffers the received packet and delivers each packet in sequence number order to the application. Check out §19.4.2 of the lecture notes to understand the semantics of the protocol and how it works.

In this task, you will implement the same functions that you did for the previous task, but with the necessary modifications to handle window sizes bigger than 1. A naive way of implementing the receiver will be to throw away all out-of-order packets, but this approach will have low throughput. A better strategy will be to create a buffer and deliver packets in order to the application.

You should copy over the **`calc_timeout`** function from the previous task; if you implemented it correctly, then it won t have to change. When you run your code with a window size of 1 (which is the default), you must get the same throughput as you did in the

previous task; this will serve as a necessary (*but not sufficient*) sanity check for the correctness of your sliding window protocol.

The template code we have provided has additional comments that you may find helpful; please read them.

After writing the required functions, run the protocol for various values of the window size ranging from 1 to 20, and also vary the link loss rate. Observe how throughput depends on window size and think about the reasons for the observed throughput. These observations will help you answer the lab questions for this task.

When your code is working, please submit it on-line:

Upload code for Task 2: Browse...

(points: 6)

Experiment with the sliding window protocol for a per-link loss rate of 0.02 (`-l 0.02`) and a topology with 5 hops between sender and receiver (`-n 5`), for various window sizes. What is the smallest window size at which the throughput reach its maximum possible value? What is the maximum value in your protocol? Note that you can set the window size using `-w`. You may want to come up with a reasonable initial guess for a good window size and go from there to the right answer.

(points: 1)

As you increase the window size from this smallest value, what happens to the RTT? Why?

(points: 1)

Now let's experiment with 6 hops between sender and receiver (the default setting) and a window size of 16 (-w 16). PS9_runsliding.py is a python script that runs PS9_2.py with various per-link loss rates. You can use the following commands to capture that output in a file and use PS9_prac-theory.py to produce two plots, one showing the throughput as a function of loss rate ("experiment") and the other showing (1 - loss rate)**12 ("theory" -- recall from lecture that this number is the estimated throughput when there's no variation in the RTT).

```
python PS9_runsliding.py > data
python PS9_prac-theory.py data
```

Please save the plot as a PNG file and then upload your PNG file using the following input field.

Additional helpful debugging notes

1. For both tasks, you may find it helpful to use the sequence number from the packet as suggested in the Python Task 2 template:

   ```
   p.properties['seqnum']
   ```

   On the other hand, `p.seqnum` will not work.
2. The semantics of send_ack : The 'sender' argument is the original sender from whom the receiver got the data packet, and not the receiver of the data packet. If you get a bunch of "No Routes to destination" it probably means you got the semantics of the send_ack function wrong.

Upload PNG file for Task 2: Browse...

(points: 1)

6.02 Introduction to EECS II: Digital Communication Systems

Fall 2012