Assignment 2: Implementing a reliable transport protocol

Due Date: Wednesday October 30th, 11:59 p.m.

Objectives:

- Implementing a reliable transport protocol
- Getting to see in practice how a reliable transport protocol such as TCP is working

Instructions and submission instruction(10 points):

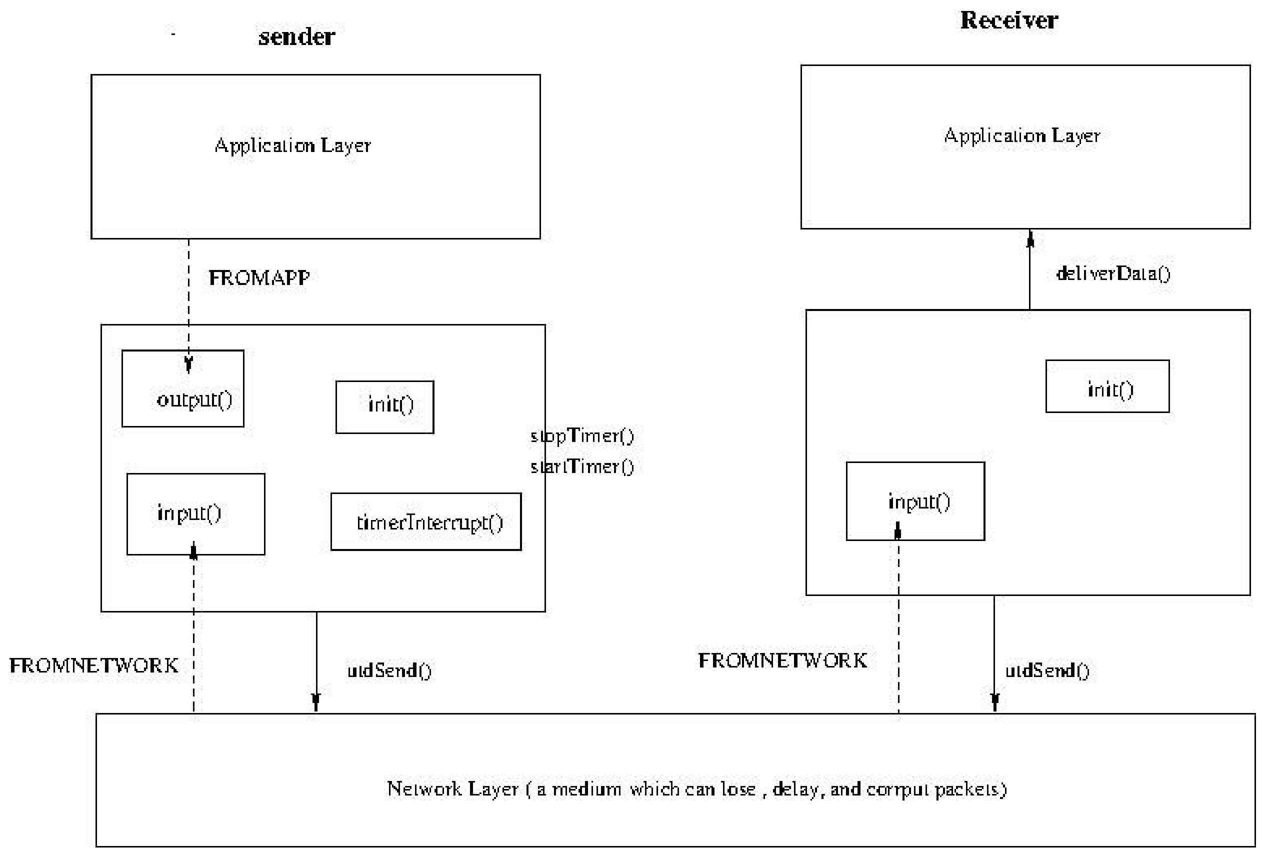
- Download startSTP.zip. It contains the files you need for this assignment
- Submit only **sender.py** and **receiver.py** files. **USE THE SAME NAMES.** Do not submit any other files.
- You should not change any other file. All the changes must be done within sender.py or receiver.py.
- Your output should match exactly the given output. They are printed by the emulator that I have developed based on this one and this one. Make sure you do not have extra printing in your sender and receiver code when submitting. When developing and debugging your code, you need to put lots of print statements. Remember to comment them out or remove them before submitting.

In this programming assignment, you will be writing the sending and receiving transport-level code for implementing a simple reliable data transfer protocol: It is named: the Alternating-bit protocol / stop-and-wait protocol/ rdt3.0 in the textbook

In a real system, the reliable transport protocol functionality is implemented inside the operating system. Since we don't have standalone machines (with an OS that you can modify), your code will have to execute in a simulated hardware/software environment. The simulator provides the programming interface to your routines, i.e., the code that would call your entities from above (application layer) and from below (network layer). Stopping/starting of timers are also simulated, and timer interrupts will cause your timer handling routine to be activated.

The routines you will write

The procedures you will write are for the sending entity (A) and the receiving entity (B). You will implement them in `sender.py` and `receiver.py`. Only unidirectional transfer of data (from A to B) is required. Of course, the B side will have to send packets to A to acknowledge receipt of data. Your routines are to be implemented in the form of the procedures described below. These procedures will be called by (and will call) procedures that I have written which simulate a network environment. The overall structure of the simulated environment is shown in the following figure:



The routines you will write are detailed below. Such procedures in real-life would be part of the operating system, and would be called by other procedures in the operating system.

Sender side (A side)

- **output(Message)**, where message is an object of type Message, containing data to be sent to the B-side(receiver side). This routine will be called whenever the upper layer (application layer) at the sending side (A) has a message to send. It is the job of your protocol to insure that the data in such a message is delivered in-order, and correctly, to the receiving side upper layer.
- **input(packet)**, where packet is an object of type Packet. This routine will be called whenever a packet sent from the B-side (i.e., as a result of a udtSend() being done by a B-side procedure) arrives at the A-side. packet is the (possibly corrupted) packet sent from the B-side. Note that since you are implementing the unidirectional transfer of data from A to B, the B-side only sends acknowledgement to the other side.
- **timerInterrupt()** This routine will be called when A's timer expires (thus generating a timer interrupt). You'll probably want to use this routine to control the retransmission of packets. See **startTimer()** and **stopTimer()** below for how the timer is started and stopped.
- **init()** This routine will be called once, before any of your other A-side routines are called. It can be used to do any required initialization.

The receiver side

- **input(packet)**, where packet is an object of type packet. This routine will be called whenever a packet sent from the A-side (i.e., as a result of a udtSend being done by an A-side procedure) arrives at the B-side. packet is the (possibly corrupted) packet sent from the A-side.
- **init()** This routine will be called once, before any of your other B-side routines are called. It can be used to do any required initialization.

Software Interfaces

The procedures described above are the ones that you will write. The following routines can be called by the sender or receiver:

- **starttimer(calling_entity,increment)**, where calling_entity is either 12345 (for starting the A-side timer, as defined in `common.py`) or 67890 (for starting the B side timer), and increment is a ***float*** value indicating the amount of time that will pass before the timer interrupts. A's timer should only be started (or stopped) by A-side routines. You don't need to have a timer on the B-side since you are implementing unidirectional transfer of data from A to B timer. To give you an idea of the appropriate increment value to use: a packet sent into the network takes an average of 5 time units to arrive at the other side when there are no other messages in the medium.
- **stoptimer(calling_entity)**, where calling_entity is either A (for stopping the A-side timer) or B (for stopping the B side timer).
- **udtSend(calling_entity,packet)**, where calling_entity is either A (for the A-side send) or B (for the B side send), and packet is an object of type Packet. Calling this routine will cause the packet to be sent into the network, destined for the other entity.
- **deliverData(calling_entity,message)**, where calling_entity is either A (for A-side delivery to application layer) or B (for B-side delivery to application layer), and message is an object of type Message. With unidirectional data transfer, you would only be calling this with calling_entity equal to A (delivery to the B-side). Calling this routine will cause data to be passed up to application layer.

The simulated network environment

A call to procedure **udtSend()** sends packets into the medium (i.e., into the network layer). On both the sender and receiver side, **input()** is called when a packet is to be delivered from the medium to your protocol layer.

The medium is capable of corrupting and losing packets. It will **not** reorder packets. When you compile your procedures and my procedures together and run the resulting program, you will be asked to specify values regarding the simulated network environment:

- **Number of messages to simulate**. My simulator (and your routines) will stop as soon as this number of messages have been passed down from application layer, regardless of whether or not all of the messages have been correctly delivered. Thus, you need no to worry about undelivered or unACK'ed messages still in your sender when the emulator stops. Note that if you set this value to 1, your program will terminate immediately, before the message is delivered to the other side. Thus, this value should always be greater than 1.
- **Loss**. You are asked to specify a packet loss probability. A value of 0.1 would mean that one in ten packets (on average) are lost.
- **Corruption**. You are asked to specify a packet loss probability. A value of 0.2 would mean that one in five packets (on average) are corrupted. Note that the contents of the payload, sequence, ack, or checksum fields can be corrupted. Your checksum should thus include the data, sequence, and ack fields.
- **Average time** between messages from sender's application layer. You can set this value to any non-zero, positive value. Note that the smaller the value you choose, the faster packets will be arriving to your sender.
- **Trace** Setting a tracing value of 1 or 2 will print out useful information about what is going on inside the emulation (e.g., what's happening to packets and timers). A tracing value of 0 will turn this off. A tracing value greater than 2 will display all sorts of odd messages that are for the simulator. A tracing value of 2 may be helpful to you in debugging your code. You should keep in mind that real implementors do not have underlying networks that provide such nice information about what is going to happen to their packets!

The Alternating-Bit-Protocol

You are to write the procedures, output(), input(), timerInterrupt(), init() on the sender sides and input(), init() which together will implement a **stop-and-wait** (i.e., the **alternating bit protocol**, which is referred to as **rdt3.0** in the text). You will implement the unidirectional transfer of data from the A-side to the B-side. Your protocol should use ACK messages as explained in the protocol description

You should choose a very large value for the average time between messages from sender's application layer, so that your sender is never called while it still has an outstanding, unacknowledged message it is trying to send to the receiver. I'd suggest you choose a value of **1000**. You should also perform a check in your sender to make sure that when `output()` is called, there is no message currently in transit. If there is, you can simply ignore (drop) the data being passed to the output() routine.

You should put your procedures in two files named `sender.py` and `receiver.py`. The starter code is provided which include the code for simulator. This lab can be completed on any machine supporting Python.

Sample outputs:

In addition to the simulator code, two sample outputs is provided. The number of messages in each run that generated each file is 40.

- File1: output1-lost=0.1-corrupt=0-trace=1
  - shows a trace of simulator when the loss probability is 0.1, corruption probability is zero, and tracing level is one.
- File2: output1-lost=0.1-corrupt=0-trace=2
  - The same as above but with a tracing level of 2

A summary of what is going on in File1

In the File1, first A sends a message to B, B receives the message and deliver it to the application layer and sends an ACK, A receives the acknowledgement and it is done. A stops the timer (The first three events in the file, events are separated by dashed lines).

Then A side sends another message to B. This time message is lost (you see the following in line#20: *SIMULATING PACKET LOSS*

So, after a while A receives a timer interrupt and sends the message again:

Line 23: *A: sending the last packet again*

Then B receives the message (the retransmit) and sends an ACK. The ACK gets lost, so A sends the message again because it receives a timer interrupt, and so on.

What is going on in File2

The same as File1 but tracing level is 2, so it provides more information.

Helpful Hints

Checksumming. You can use whatever approach for checksumming you want. Remember that the sequence number and ack field can also be corrupted (if you are curious you can find it in the simulator code). We would suggest a TCP-like checksum, which consists of the sum of the (integer) sequence and ack field values, added to a character-by-character sum of the payload field of the packet (i.e., treat each character as if it were an integer and just add them together).

START SIMPLE. Set the probabilities of loss and corruption to zero and test out your routines. Better yet, design and implement your procedures for the case of no loss and no corruption, and get them working first. Then handle the case of one of these probabilities being non-zero, and then finally both being non-zero.

Debugging. We'd recommend that you put LOTS of print statements in your code while your debugging your procedures. Don't forget to disable them before you submit your code. **I do not want any print statement by your code**

Random Numbers. The simulator generates packet loss and errors using a random number generator. Our past experience is that random number generators can vary widely from one machine to another. Our simulation routines have a test to see if the random number generator on your machine will work with our code.You may need to modify the random number generation code in the simulator we have supplied you. It is likely that random number generation on your machine is different from what this simulator expects. When you run the code it generates an average (line #5 in sample output). If this average is between 0.25 and 0.75, that is good.

Initial sequence number: You can assume that the three way handshake has already taken place. You can hard-code an initial sequence number into your sender and receiver.

Timer when working with timer (calling startTimer and stopTimer) pass them a float value not an integer value

Events in simulator. There are three events happening on the sender side:

- FROMAPP

- TIMERINTERRUPT
- FROMNETWORK

On the receiver side only one event happens

- FROMNETWORK

A description of different files in the provided code

Common.py: include the definitions of classes: `Packet`, `Message`, `Event`, `EventType` and `EventList`. You need to know about `Packet` and `Message` classes. The rest are used by the simulator. In addition, the constant variables that are needed within different files are defined here. **`A`**, to represent the sender entity and **`B`** to represent the receiver entity. When the sender/receiver objects are initialized (inside `iniSimulator` function of NetworkSimulator.py), they are assigned the correct entity name.

main.py: It asks the user to enter the parameters needed to initialize the simulator. You can hardcode some values when testing your code. It initialize an object of type NetworkSimulator.

NetworkSimulator.py: The main code for simulator. It includes the implementations for all the interfaces needed to communicate with application layer and network layer. You should not change this code. When initialized, it also creates the sender and receiver objects.The sender and receiver objects are initialized from within the simulator. When the sender and receiver are initialized, they are passed a reference to the current network simulator object.

sender.py: The functions on the sending side should be implemented here. It includes the declarations for some other auxiliary functions such as isDuplicate, etc. that is needed in the functions you are going to implement. Each sender/receiver class has a member named `networkSimulator`. So each sender/receiver object has access to the simulator methods through this member variable.

For example, if you want to call `udtSend` function from one of the methods in sender/receiver, you should call it like the following:

```
    self.networkSimulator.udtSend
```

To access a member variable from within a method of a class, you need to precede that with the `self` keyword (python way of doing it)

In addition, each sender/receiver has a member named entity which holds the values A/B. A is for the sender and B is for receiver. More details are provided in the supplied codes.

NOTE: The return statement at the end of methods of sender.py and receiver.py are not needed. They are just there so when you start the code, it runs and you do not face with errors.

When you first run the code (by running `main.py)`, you will see that the simulator is generating a set of FROMAPP events. The number of events will be equal to the number of messages you passed to the simulator. It means that the application layer on the sender side is sending the message to its transport layer, but nothing more is there unless you implement the rest.