Assignment 2: Implementing a simple transport protocol

Due Date: Wednesday March 2nd, 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

General Instructions:

- Make sure to use python 3.5 or higher.
- Download **startSTP.zip**. It contains the files you need for this assignment
- You should not change any other files. All the changes must be done within **`sender.py`**, **`receiver.py`**, or **`common.py`**.
- When you first run the code, you get the output that is shown below:

```
> python main.py
>Network Simulator
>Enter number of messages to simulate (> 0): 5
>Enter the packet loss probability (0.0 for no loss): 0
>Enter the packet corruption probability (0.0 for no
corruption): 0
>Enter the average time between messages from the sender's
application layer (> 0.0): 1000
Initializing Network Simulator
Initializing sender: A: 12345
Initializing receiver: B: 67890
average = 0.5033190011370157
----------------------------------------
A: Sending the data aaaaaaaaaaaaaaaaaaa
----------------------------------------
A: Sending the data bbbbbbbbbbbbbbbbbbb
----------------------------------------
A: Sending the data ccccccccccccccccccc
----------------------------------------
A: Sending the data ddddddddddddddddddd
----------------------------------------
A: Sending the data eeeeeeeeeeeeeeeeeee
do not schedule: the maximum number of messages is
scheduled
```

you will see that the simulator is generating a set of events (FROMAPP events as explained below). The number of events will be equal to the number of messages you passed to the simulator (5 in this case). 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. Look at the sample outputs provided (output1, output2, output3) for how your output should look like after you develop the required. The function will be discussed later in this document.

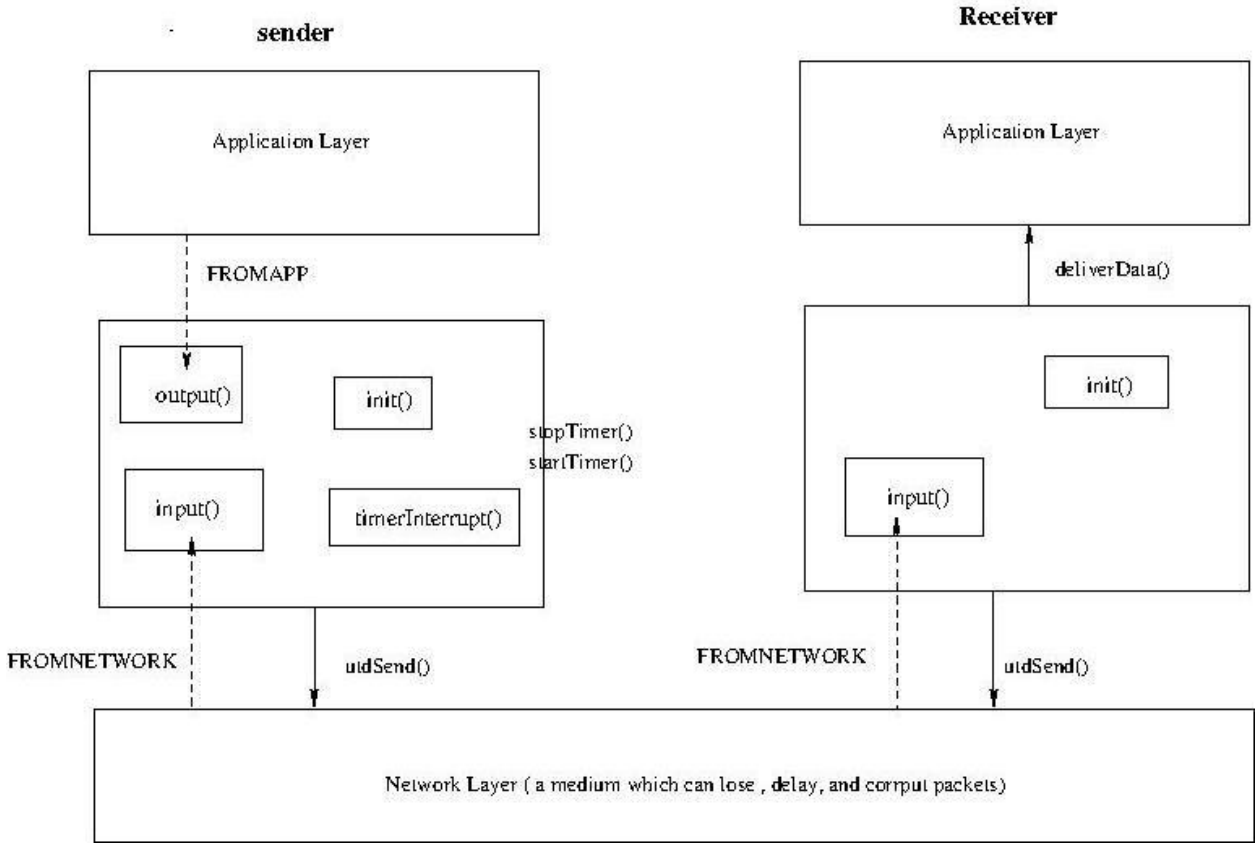
- Your output should match exactly the given outputs. They are printed by the simulator. Make sure you do not have extra printing in your sender and receiver code when submitting. When developing and debugging your code, you need to put lots of print statements. Remember to comment them out or remove them before submitting.

The functions you will develop

In this programming assignment, you will be writing the sending and receiving side of 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 environment. The simulator provides the interface to the application layer (**output** and **deliverData**) and the interface provided by the network layer to the transport layer (**udt_send, input**).

Your job is to write the code for the sending side (A) and the receiving side (B) of the transport protocol. 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.The overall structure of the simulated environment is shown in the following figure:



The functions you will write are detailed below. Such functions in reality would be part of the operating system, and would be called by other functions in the operating system. The function you develop will be called by functions simulating the application layer and will call functions which simulate a network environment.

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 function will be called whenever the application layer at the sending side (A) has a message to send. It is the job of your protocol to ensure that the data in such a message is delivered in-order, and correctly, and reliably to the receiving side application layer.
- **`input(packet):`** The `packet` is the (possibly corrupted) packet sent from the B-side. The `packet` is an object of type Packet. This function will be called whenever a packet sent from the B-side arrives at the A-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 function will be called when A's timer expires (thus generating a timer interrupt). It is needed to control the retransmission of packets. See **`startTimer()`** and **`stopTimer()`** below for how the timer is started and stopped.
- **`init()`** This function will be called once, before any of the A-side functions are called. It can be used to do any required initialization.

You don't call the above function. You just implement them. They will be called by the simulator.

The receiver side

- **`input(packet)`**, where packet is an object of type packet. This function will be called whenever a packet sent from the A-side arrives at the B-side. packet is the (possibly corrupted) packet sent from the A-side.
- **`init()`** This function will be called once, before any of the other B-side functions are called. It can be used to do any required initialization.

You are to write the function, 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). Your protocol should use ACK messages as explained in the protocol description. You will implement other helper functions as listed in the supplied code.

Software Interfaces

The functions described above are the ones that you will write. The following functions can be called by the above functions.

- **`starttimer(calling_entity,increment)`**: `calling_entity` is either A or B (constants defined in common.py). A is used by the functions on the A side. The `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 functions. 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 function 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 B (delivery to the B-side). Calling this function will cause data to be passed up to the application layer.

A call to **`udtSend()`** function sends packets into the network layer. On both the sender and receiver side, **`input()`** is called when a packet is to be delivered from the network layer to your application layer.

The network layer is capable of corrupting and losing packets. It will **not** reorder packets.

The simulated network environment

When you run the simulator, you will be asked to specify values regarding the simulated network environment:

- **Number of messages to simulate**. The simulator will stop as soon as this number of messages have been passed down from the application layer, regardless of whether or not all of the messages have been correctly delivered. Thus, you need not to worry about undelivered or unACK'ed messages still in your sender when the simulator 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 corruption 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 the ack fields.

- **Average time** between messages from the 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.

  You should choose a very large value for the average time between messages from the 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() function*.

- **Trace** Setting a tracing value of 1 or 2 will print out useful information about what is going on inside the simulation (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 implementers of real transport protocols do not have underlying networks that provide such nice information about what is going to happen to their packets!

Sample outputs:

In addition to the simulator code, two sample outputs are provided. The number of messages in each run that generated each output is 5.

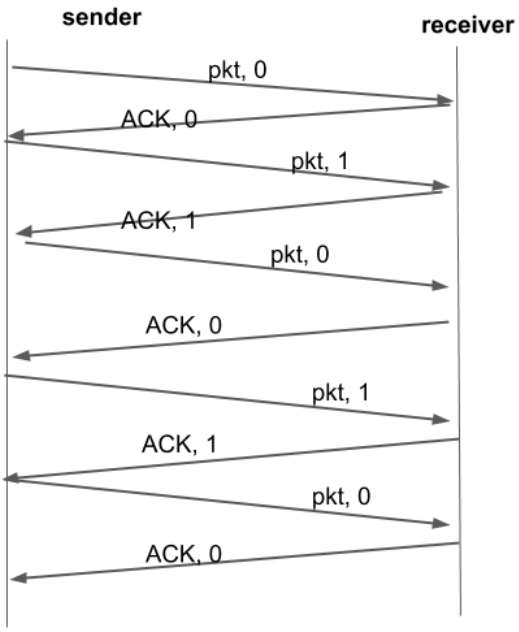
- output1: loss probability=0.0, corruption probability=0.0
  - shows a trace of protocol when the underlying layer is reliable (no loss or corruption) and tracing level is one.
- output2: loss probability=0.3, corruption probability=0.0
  - shows a trace of protocol when the loss probability is 0.1, corruption probability is zero, and tracing level is one.
- output3: loss probability=0.1, corruption probability=0.3
  - shows a trace of protocol when the loss probability is 0.1, corruption probability is zero, and tracing level is one.

The corresponding time-sequence diagrams are shown below:

output1

Loss probability = 0.0

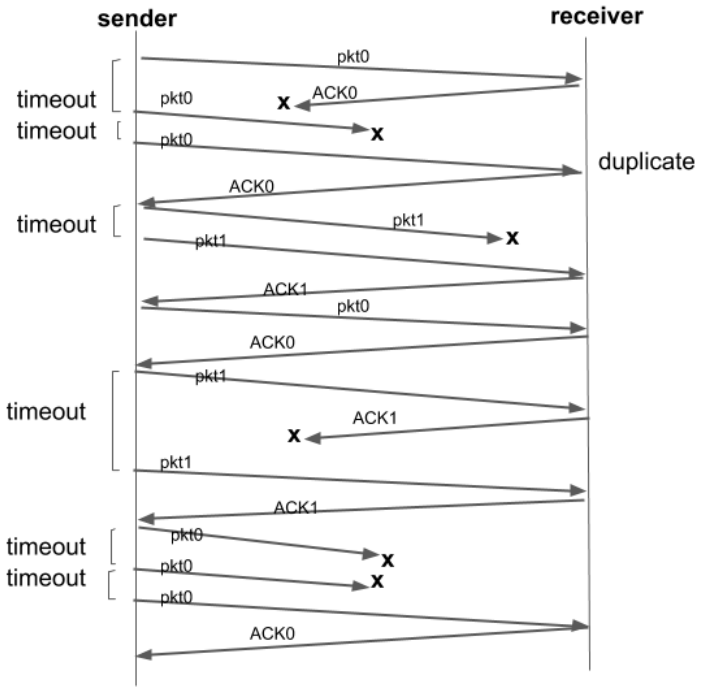
Corrupt probability = 0.0



output2

Loss probability = 0.3

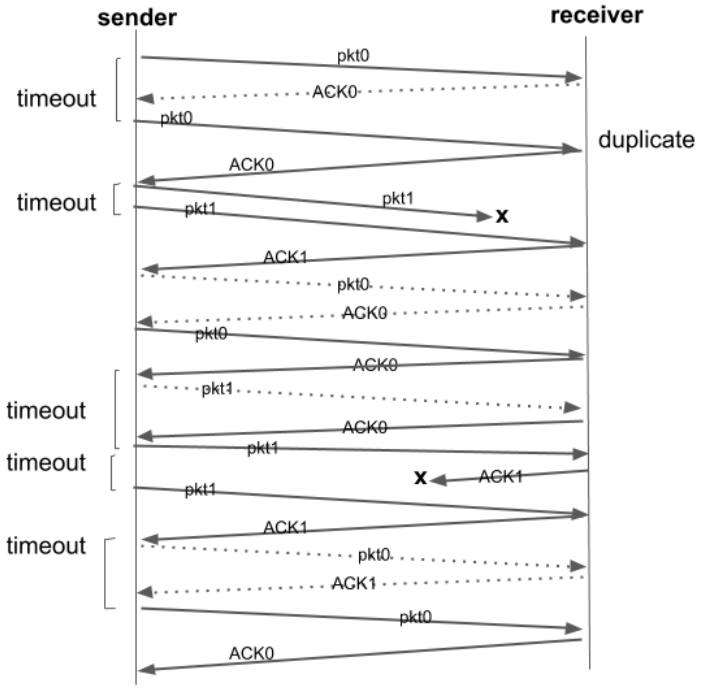
Corrupt probability = 0.0



output3

Loss probability = 0.1

Corrupt probability = 0.3



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). Checksum function should be implemented in `common.py` as it is needed by both sender and receiver.

Start simple and go step by step. Set the probabilities of loss and corruption to zero and test out your functions (You can hardcode them in the program to save time in testing the code). Design and implement your functions 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 you're debugging your functions. 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 functions 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 outputs). 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 and acknowledgement number into your sender and receiver. **Make the initial sequence number on sender to be zero and the initial acknowledgement number on the receiver to be zero.**

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

In response to FROMAPP event, `output` function is being called on the sender. In response to TIMERINTERRUPT event, `timerInterrupt` function is being called on the sender side. In response to FROMNETWORK event, `input` function is called on the server side. On the receiver side only one event happens

- FROMNETWORK

In response to FROMNETWORK event, `input` function is being called on the receiver side.

A description of files in the supplied 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. For example, **`A`**, to represent the sender entity and **`B`** to represent the receiver entity. When the sender/receiver objects are initialized (inside the `iniSimulator` function of `NetworkSimulator.py`, they are assigned the correct entity name. The implementation of checksum function is placed in this file as it is used by both the sender and receiver.

`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 initializes 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 of the transport protocol 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 an attribute of a class, you need to precede that with the `self` keyword in Python.

In addition, each sender/receiver has an attribute 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.

submission instruction:

- Submit only the following three files: `sender.py`, `receiver.py`, and `common.py` files.
- Do not submit any other files. **Do not zip the files.**

Grading

[35 points] Sender:
[20 points] receiver:
[5 points] common.py
[5] Following submission instruction
[-25] code does not run
A large portion of your mark will be deducted if your code does not run.

Make sure your code works in at least one scenario. Remember you need to consider the following scenarios: there are no packet loss or corruption, there is either packet loss or packet corruption, and there are both packet loss and corruption.
Try to develop and run your code step by step. Follow the hints and tips provided