The goal of the project is to help you to get familiar with the reliable transfer mechanism. You will implement ones that are similar to rdt 3.0 (Alternating-Bit-Protocol or Stop-and-wait) and Go-Back-N on the textbook.

In this programming assignment, you will be writing the sending and receiving transport-level code for implementing a simple reliable data transfer protocol. This lab should be fun since your implementation will differ very little from what would be required in a real-world situation.

Since you probably don't have standalone machines (with an OS that you can modify), your code will have to execute in a simulated hardware/software environment. However, the programming interface provided to your routines, i.e., the code that would call your entities from above and from below is very close to what is done in an actual UNIX environment. (Indeed, the software interfaces described in this programming assignment are much more realistic that the infinite loop senders and receivers that many texts describe). Stopping/starting of timers are also simulated, and timer interrupts will cause your timer handling routine to be activated.

Your Assignment

- Implement the following routines in the source code and any other helper functions that you design to support the reliable transfer from A to B (only one direction) with a simulated lossy and noisy channel:

The routines you will write are detailed below.

- **A_output(message)**, where message is an instance of type msg, containing data to be sent to the B-side. This routine will be called whenever the upper 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, to the receiving side upper layer.
- **A_input(packet)**, where packet is an instance of type pkt. This routine will be called whenever a packet sent from the B-side (i.e., as a result of a tolayer3() being done by a B-side procedure) arrives at the A-side. packet is the (possibly corrupted) packet sent from the B-side.
- **A_handle_timer()** 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 start_timer() and remove_timer() below for how the timer is started and stopped.
- **The initialization of A**. You need to initialize its initial sequence, states,etc.
- **B_input(packet)**, where packet is an instance of type pkt. This routine will be called whenever a packet sent from the A-side (i.e., as a result of a tolayer3() being done by a A-side procedure) arrives at the B-side. packet is the (possibly corrupted) packet sent from the A-side.
- **The initialization of B.** You need to initialize its initial sequence, states, etc.

- Providing justification for the reliable transfer behaviors of your system by providing at least three test cases for each version (rdt 3.0 and Go-Back-N) of the lab.

What to Submit

A zip file (.tar.xz or .zip formats), named as <firstAndLastName>-project2.zip (or .tar.xz) (example: AliceBoberson-project2.tar.xz), to the class canvas site containing:

1. Your source code (comprised of two folders):
    a. Stop and Wait (as stop_and_wait folder) inside of which:
        i. Folder container primary code (as pj2 folder) inside which:
            1. Your A modified code - using Stop and Wait (as A.py)
            2. Your B modified code - using Stop and Wait (as B.py)
            3. event.py
            4. event_list.py
            5. __init__.py
            6. msg.py
            7. packet.py
            8. simulator.py (with the all the sim values set to that which they started, eg: when you downloaded it)
        ii. main.py
    b. Go Back N Folder (as go_back_n as folder) inside of which:
        i. Folder container primary code (as pj2 folder) inside which:
            1. Your A modified code - using Go Back N (as A.py)
            2. Your B modified code - using Go Back N (as B.py)
            3. event.py
            4. event_list.py
            5. __init__.py
            6. msg.py
            7. packet.py
            8. simulator.py (with the all the sim values set to that which they started, eg: when you downloaded it)
        ii. main.py
2. A readme file (as README.md using Markdown syntax) with sections::
    - A header (comprised of two lines)
        - <Your name>,Zhang CSCI4211,<current date (day/month/year)>
        - Ex: Marky Markdowns,Zhang CSCI4211,31/12/1969
        - <coding language>,<server source code file name>,<compile script (if any)>,<executable file name>
        - Ex(s):
            - Java,server.java,compile.sh,server.class
            - Python3,server.py,,server.py
                - Note: ‘,,’ is not a typo, here it means “No compile script needed”

        - Python2,server.py,,server.py
        - C,server.c,makefile,main
        - C++,server.cpp,makefile,main
        - Go,server.go,build.sh,main
    - Compilation section
        - Step by step instructions on how to compile and setup your code
        - Ex:
            1. Run gcc -o test test.c in command line
            2. Copy input.txt to dir/
            3. Etc...
    - Execution/Running section
        - Step by step instruction on how to run/execute your program
        - Ex:
            1. Run myprogram.exe
            2. Etc...
    - Description section
        - Describe overall what your program does and how logical it operates, be detailed
        - If you added any data structures or fields, explain what they are and why you added them
        - Functions or methods. Explain how you implement each function or method. Functions or methods include but not limited to A_output(), A_input(), A_timerinterrupt() B_input(), initialization functions.
        - Ex: This program does x with y and creates z …
        - Ex: Go-Back-N does … Stop and Wait does ...
    - Evaluation section
        - Test cases. List your test cases and expected output with a detailed analysis. You might want to hand in output for a run that was long enough so that at least 20 messages were successfully transferred from sender to receiver. You might want to describe how your protocol correctly recovered from packet loss and corruption.

To help with they Assignment

Software Interfaces

The procedures described above are the ones that you will write. The following routines which can be called by your routines:

- **start_timer(calling_entity,increment)**, where calling_entity is either A or B, 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, and similarly for the B-side 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. It is implemented in event_list.
- **remove_timer()**. It is implemented in event_list.
- **tolayer3(calling_entity,packet)**, where calling_entity is either A or B, and packet is an instance of packet. Calling this routine will cause the packet to be sent into the network, destined for the other entity. It is implemented in simulator.
- **tolayer5(calling_entity,msg)**, where calling_entity is either 0 (for A-side delivery to layer 5) or 1 (for B-side delivery to layer 5), and msg is an instance of msg. With unidirectional data transfer, you would only be calling this with calling_entity equal to 1 (delivery to the B-side). Calling this routine will cause data to be passed up to layer it is implemented in simulator.

The simulated network environment

A call to procedure tolayer3() sends packets into the medium (i.e., into the network layer). Your procedures A_input() and B_input() are 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 emulator (and your routines) will stop as soon as this number of messages have been passed down from layer 5, regardless of whether or not all of the messages have been correctly delivered. Thus, you need not 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) is corrupted. Note that the contents of payload, sequence, ack, or checksum fields can be corrupted. Your checksum should thus include the data, sequence, and ack fields.
- **The average time between messages from the sender's layer5 (lambda)**. 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 at your sender.

The Alternating-Bit-Protocol version of this lab

You are to write the procedures, A_output(),A_input(), A_timerinterrupt(), B_input(), the initialization function of A and B which together will implement a stop-and-wait (i.e., the alternating bit protocol, which we referred to as rdt3.0 in the text) unidirectional transfer of data from the A-side to the B-side. Your protocol should use both ACK and NACK messages. You should choose a very large value for the average time between messages from the sender's layer5, 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.

The Go-Back-N version of this lab

You are to write the procedures, A_output(), A_input(), A_handle_timer(), B_input(), and the initialization function of A and B which together will implement a Go-Back-N unidirectional transfer of data from the A-side to the B-side, with a window size of 8. Your protocol should use both ACK and NACK messages. Consult the alternating-bit-protocol version of this lab above for information about how to obtain the network emulator. We would STRONGLY recommend that you first implement the easier lab (Alternating Bit) and then extend your code to implement the harder lab (Go-Back-N). Believe me - it will not be time wasted! However, some new considerations for your Go-Back-N code (which do not apply to the Alternating Bit protocol) are:

- **A_output(msg)**, where msg is an instance of type msg, containing data to be sent to the B-side. Your A_output() routine will now sometimes be called when there are outstanding, unacknowledged messages in the medium - implying that you will have to buffer multiple messages in your sender. Also, you'll also need buffering in your sender because of the nature of Go-Back-N: sometimes your sender will be called but it won't be able to send the new message because the new message falls outside of the window. Rather than have you worry about buffering an arbitrary number of messages, it will be OK for you to have some finite, maximum number of buffers available at your sender (say for 50 messages) and have your sender simply abort (give up and exit) should all 50 buffers be in use at one point (Note: using the values given below, this should never happen!) In the “real-world,” of course, one would have to come up with a more elegant solution to the finite buffer problem!
- **A_handle_timer(),** This routine will be called when A's timer expires (thus generating a timer interrupt). Remember that you've only got one timer, and may have many outstanding, unacknowledged packets in the medium, so you'll have to think a bit about how to use this single timer.

Helpful Hints

- **Checksumming**. The checksum function is provided as a method of the class packet. You can use that to perform checksum verification.
- **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 function in your code while you are debugging your procedures.
- **Circular buffer**. Circular buffer has been provided to support the buffer for the sender.

1. **Expected output**: Let's say that you have 20 messages to send. The output will look like this:

```
data recieved : aaaaaaaaaaaaaaaaaaaa
data recieved : bbbbbbbbbbbbbbbbbbbb
data recieved : cccccccccccccccccccc
data recieved : dddddddddddddddddddd
data recieved : eeeeeeeeeeeeeeeeeeee
data recieved : ffffffffffffffffffff
data recieved : gggggggggggggggggggg
data recieved : hhhhhhhhhhhhhhhhhhhh
data recieved : iiiiiiiiiiiiiiiiiiii
data recieved : jjjjjjjjjjjjjjjjjjjj
data recieved : kkkkkkkkkkkkkkkkkkkk
data recieved : llllllllllllllllllll
data recieved : mmmmmmmmmmmmmmmmmmmm
data recieved : nnnnnnnnnnnnnnnnnnnn
data recieved : oooooooooooooooooooo
data recieved : pppppppppppppppppppp
data recieved : qqqqqqqqqqqqqqqqqqqq
data recieved : rrrrrrrrrrrrrrrrrrrr
data recieved : ssssssssssssssssssss
simulation end
```

It shows that all the messages are sent correctly. There is a possibility that some messages are dropped because the state is "WAIT_ACK" (stop and wait) or the buffer is full (go back n). If that happens, you can increase the value of lambda, then you will see the above result.