IIT Goa CS 378 (Computer Networks LAB)
Instructor: Dr. Neha Karanjkar

LAB 4: Principles of Reliable Data Transfer

Instructions:

- This is an **individual** assignment. You may discuss concepts with other students, but sharing of code or reports with others is not permissible. Any form of copying or collusion will be treated as a plagiarism case (reported to DAC, with grade penalties) for **all** students involved.

- This is a **take-home** assignment. The deadline for submission (on Google classroom) is **Monday 16th March**. You may be asked to show a demo of the working solution to the TAs during the next lab

- **Format for final submission:**
  Your final submission should consist of the following files, uploaded on Google clasroom:
  1. A typeset report in pdf format. The file should be named **<ROLL NUM>.pdf**, for example “18000123.pdf”. The quality of the report carries significant weightage. Your report should be complete, correct, clear and concise.
  2. A Python file named **Protocol_rdt22.py** containing your implementation of the rdt2.2 protocol as asked in question 2. The code should contain comments where appropriate to explain your logic.
  3. A Python file named **Protocol_rdt3.py** containing your implementation of the rdt3.0 protocol as asked in question 3.

- Do **not** include any other files in your submission (such as the template files provided by the instructor). Your implementation of the protocols must adhere to the interface provided in the template. It should be possible for anyone to run the simulation with the existing template, by only adding your protocol files to the existing template folder and changing the name in the “import <protocol-file>” line in the testbench.

1. Go through the Python code provided in the Template, and try to understand the behavior of each block. The file `Channel.py` implements a model for an unreliable channel over which packets can be corrupted or lost. This model has the following parameters:

   Pc: The probability of a packet being corrupted
   Pl: The probability of a packet being lost
   Delay: The time it takes for a packet to travel over the channel and reach the receiver.

   The file `Protocol_rdt1.py` implements the trivial protocol rdt1.0 which works only if the channel is assumed to be ideal. Run the simulation first with Pc=0, and then Pc=0.5. Check that the protocol fails in the second case, and list the failure symptoms.

2. The file `Protocol_rdt2.py` implements the simple ACK/NAK based protocol rdt2.0 that can work when data packets can get corrupted. Check that this protocol indeed works by setting Pc>0 for the data-channel.

3. For the testbench using `Protocol_rdt2.py`, modify the code such that:
   a) The sending application generates a fixed total number of messages (say 1000), with a fixed time interval between each message (say 3 units of time)
   b) As soon as the protocol at the sending-side (rdt_Sender) receives positive acknowledgements for all of the 1000 messages, the simulation ends, and a quantity "T_avg" is printed as output, where T_avg is the time between sending a packet and receiving a positive acknowledgement for it, averaged across all packets. This is, in essence the Average Round-Trip Time (RTT avg).

      (Note that you need to run each simulation as long as necessary for all 1000 messages to be acknowledged.)

   You would expect that T_avg should increase with Pc, but in what manner? (linearly? exponentially? geometrically?). Obtain a plot of T_avg versus Pc for (0<=Pc<=0.9) and explain what trend you observe and why.

4. Protocol_rdt2.py will not work if ACK/NAK can also get corrupted. Check this by setting Pc>0 for the ack-channel and state the symptoms you observe.

5. Develop an alternating-bit protocol as described in K&R (rdt_2.2) which can work even when both the data and ack packets can get corrupted. Test that your protocol works by simulating for a large amount of time or a large number of packets sent, with Pc>0 for both the data and ack channels. You need to submit only the protocol as a file named `Protocol_rdt22.py`.

6. The protocol rdt2.2 will not work if packets can be lost.

a) Check this by setting Pl>0 in the testbench with your implementation of Protocol_rdt22. What failure symptoms do you observe?

b) Implement an alternating-bit protocol with Timeouts (rdt3.0) that can work when data or ack packets can be corrupted or lost. Set the timeout value to 3*Delay. Test that your protocol indeed works by simulating for a large amount of time or a large number of packets sent, with Pc>0 and Pl>0 for both the data and ack channels. You need to submit the protocol as a file named "Protocol_rdt3.py".

[**Hint:** To implement timeouts, you may need to model a Timer in SimPy. A "skeleton" for implementing such a timer is provided in Appendix A .]

c) You would expect that T_avg (defined the same way as in question 1) would increase with Pl. For your implementation of the rdt3.0 protocol, with channel Delay=2, Pc=0.2, timeout=3*Delay and total packets generated =1000, plot T_avg versus Pl.

d) **[BONUS QUESTION]** For the scenario where packet loss is possible (assume Pc=0), derive an analytical expression for how T_avg should vary with Pl. Check if the trend observed from simulations matches this.

e) Protocol rdt3.0 will not work if packets can be re-ordered over the channel. This can be easily modeled by setting the channel delay for each packet to be a randomly chosen number instead of having the same value for all packets. Thus, packet-2 sent after packet-1, might experience a lower delay and arrive ahead of a packet-1 at the receiver. Implement this feature in the model, and check if the rdt3.0 protocol indeed fails. In your report, show the code snipette that models this packet reordering.

APPENDIX A : SimPy Template for implementing Timeouts

```
# Here is a “skeleton” code for modeling timers and timeout events for
the rdt_Sender

class rdt_Sender(object):

        def __init__(self,env):
                .....

                # additional timer-related variables
                self.timeout_value=10
                self.timer_is_running=False
                self.timer=None

        # This function models a Timer's behavior.
        def timer_behavior(self):
                try:
                        # Start
                        self.timer_is_running=True
                        yield self.env.timeout(self.timeout_value)
                        # Stop
                        self.timer_is_running=False
                        # take some actions
                        self.timeout_action()
                except simpy.Interrupt:
                        # upon interrupt, stop the timer
                        self.timer_is_running=False

        # This function can be called to start the timer
        def start_timer(self):
                assert(self.timer_is_running==False)
                self.timer=self.env.process(self.timer_behavior())

        # This function can be called to stop the timer
        def stop_timer(self):
                assert(self.timer_is_running==True)
                self.timer.interrupt()

        def timeout_action(self):
                # add here the actions to be performed
                # upon a timeout
                ...

        def rdt_send(self,msg):
                # whatever actions should go here
                ....

        def rdt_rcv(self,packt):
                # whatever actions should go here
                ....
```