Project 2 (Reliable Data Transport Protocol)

Due Date

- **Wed, Oct 30th @ 11:59pm**

A Note on Collaboration

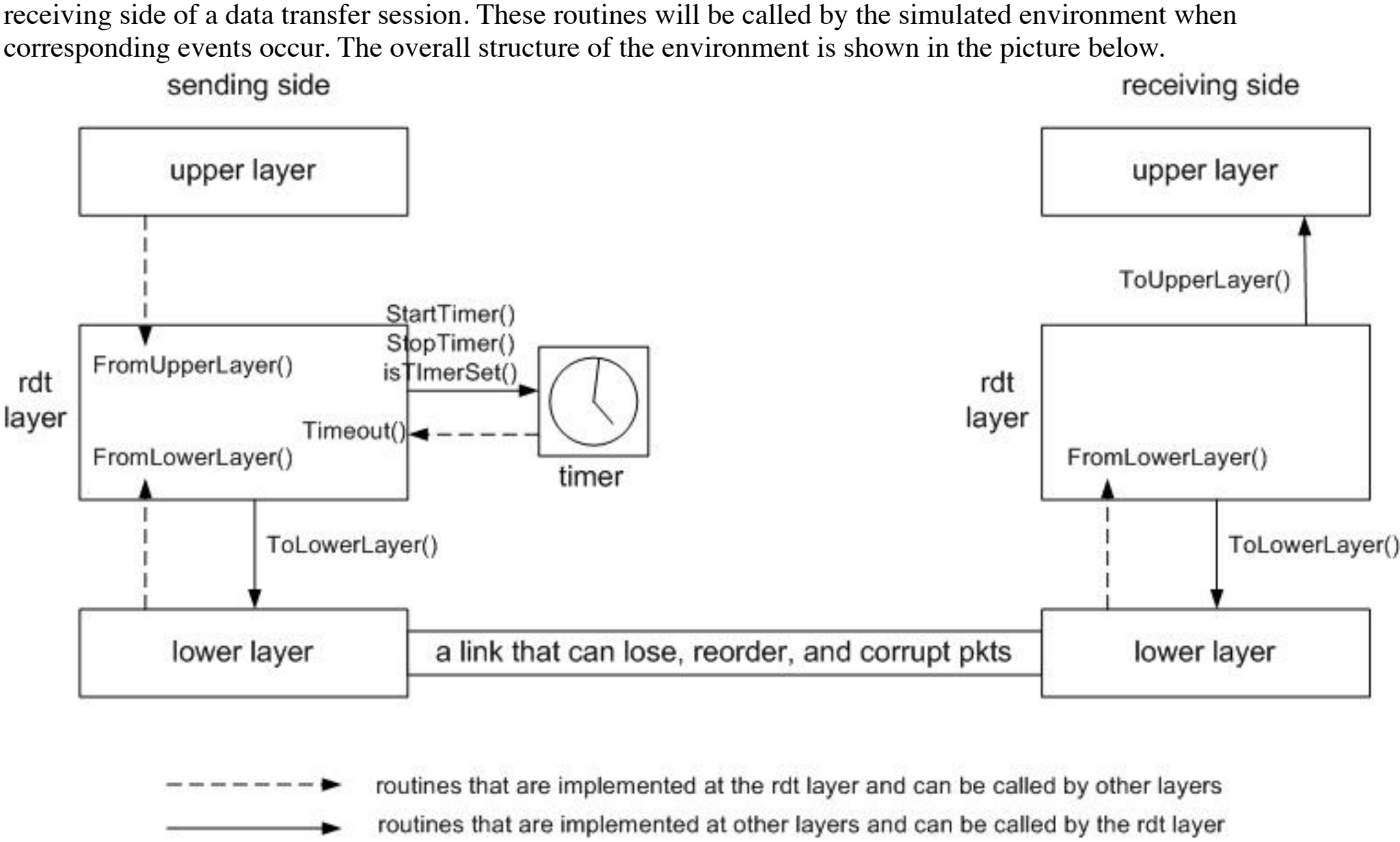
You are permitted to work together in pairs on this programming assignment.

Objective

In this assignment, you will be implementing the sending and receiving side of a reliable data transport (RDT) protocol. Your protocol should achieve error-free, loss-free, and in-order data delivery on top of a link medium that can lose, reorder, and corrupt packets. Your implementation can follow any variation of the sliding window protocol, e.g., Go-Back-N, Selective Repeat, or the TCP rdt protocol. Since we do not have machines with an operating system that we can modify, your implementation will run in a simulated environment. This also means that your program should be written in the same programming language as the simulator -- Java. The simulator is available at: https://github.com/jiup/rdt_skeleton.

The routines you need to implement

Only unidirectional transfer data is required in this assignment. Of course, the receiver may need to send back packets to acknowledge receipt of data. The routines you need to implement are for the RDT layer at both the sending and receiving side of a data transfer session. These routines will be called by the simulated environment when corresponding events occur. The overall structure of the environment is shown in the picture below.



The unit of data passed between the upper layer and the RDT layer is the *message*, which is declared as:

```
byte[] message;
```

`message.length` indicates the size of `message` measured in bytes. The reliable data transport protocol requires that the data in the receiving node be delivered in order. It does not, however, require the preservation of message boundaries.

The unit of data passed between the RDT layer and the lower layer is the *packet*, which is declared as:

```
public class Packet {
    public static final int RDT_PKTSIZE = 64;
    private static Random random = new Random();

    public byte[] data;

    public Packet() {
        this.data = new byte[RDT_PKTSIZE];
        random.nextBytes(this.data); // random fill
    }
}
```

This means that the lower layer always delivers packets in 64-byte chunks. Therefore, fragmentation may be necessary if the message passed from the upper layer is too big to fit into a packet.

You will need to modify two files (MyRdtSender.java and MyRdtReceiver.java) by enhancing three methods for handling requests to send data from the upper layer, and receipt of data from the lower layer for both sender and receiver. The routines you need to implement are detailed below. Such routines in real life would be part of the operating system.

- `(Sender) public void receiveFromUpperLayer(byte[] message);`
- `(Sender) public void receiveFromLowerLayer(Packet packet);`
- `(Sender) public void onTimeout();`
- `(Receiver) public void receiveFromLowerLayer(Packet packet);`

Note -- you may also choose to add static variables or constructors to either `MyRdtSender` or `MyRdtReceiver`. You should not modify any code for the underlying simulation.

The routines you can call (implemented by the simulated environment):

The routines you can call (implemented by the simulated environment) are detailed below. Such routines in real life would also be part of the operating system.

- `(Sender) public void startTimer(double timeout);` - start the sender timer with a specified `timeout` (in seconds). This timer is canceled when `(Sender) public void stopTimer()` is called or a new `(Sender) public void startTimer()` is called before the current timer expires. `(Sender) onTimeout()` will be called when the timer expires.
- `(Sender) public void stopTimer();` - stop the sender timer.
- `(Sender) public boolean isTimerSet();` - check whether the sender timer is being set, return true if the timer is set, return false otherwise.
- `(Sender) public double getSimulationTime();` - return the local simulation time, which may be useful in timer management. You should not assume that the time is synchronized between the sender and receiver side.
- `(Sender) public void sendToLowerLayer(Packet packet);` - pass a packet to the lower layer at the sender for delivery.
- `(Receiver) public void sendToLowerLayer(Packet packet);` - pass a packet to the lower layer at the receiver for delivery.
- `void (Receiver) sendToUpperLayer(byte[] message);` - deliver a message to the upper layer at the receiver.

The simulated network environment:

The overall structure of the simulation environment is shown in the picture above. There is one and only one timer available at the sender. The underlying link medium can lose, reorder, and corrupt packets. The default one-way latency for this link is 100ms when the link does not reorder the packets. After you compile your code along with the code given and run the resulting program, you will be asked to specify the following parameters for the simulation environment.

- `sim_time` - total simulation time, the simulation will end at this time (in seconds).
- `mean_msg_arrivalint` - average intervals between consecutive messages passed from the upper layer at the sender (in seconds). The actual interval varies between zero and twice the average.
- `mean_msg_size` - average size of messages (in bytes). The actual size varies between one and twice the average.
- `outoforder_rate` - the probability that a packet is not delivered with the normal latency - 100ms. A value of 0.1 means that one in ten packets are not delivered with the normal latency. When this occurs, the latency varies between zero and 200ms.
- `loss_rate` - packet loss probability: a value of 0.1 means that one in ten packets are lost on average.
- `corrupt_rate` - packet corruption probability: a value of 0.1 means that one in ten packets (excluding those lost) are corrupted on average. Note that any part of a packet can be corrupted.
- `tracing_level` - levels of trace printouts (higher level always prints out more information): a tracing level of 0 turns off all traces, a tracing level of 1 turns on regular traces, a tracing level of 2 prints out the delivered message. Most likely you want to use level 1 for debugging and use level 0 for final testing.

Helpful hints

- **Timer** - There is only one physical timer available at the sender. In some cases you might want to keep multiple timers simultaneously. You can simulate multiple virtual timers using a single physical timer. The basic idea is that you keep a chain of virtual timers ordered in their expiration time and the physical timer will go off at the first virtual timer expiration.
- **Timeout** - The average oneway packet latency in the simulation is 0.1 second (the latency of a particular packet may be higher than the average). For the round-trip timeout value for retransmission, I recommend you to set it at 0.3 second. I make no guarantee that this is the best timeout value.
- **Window size** - Too big a window size may affect the efficiency of Go-Back-N (it needs to retransmit the entire window at a timeout). I recommend you to use a window size of 10. Again, I make no guarantee that this is the best setting.
- **Data buffer at the sender** - You may need extra data buffer at the sender in addition to the send window. The reason is that the upper layer may wish to send data at a faster rate than the link capacity for a sustained period of time. Without additional buffering, data may be lost when the send window is full and another message is passed from the upper layer at the sender. This buffer should be drained when slots in the send window become available. **Note:** An alternative here is to block the sending application if there is no free slot in the send window and the application wants to send more. This is not an option here due to the nature of the simulator --- the blocking of any routine will block the whole simulator.
- **Error detection and checksumming** - You will need some kind of checksumming to detect errors. The Internet checksum is a good candidate here. Remember that no checksumming can detect all errors but your checksum should have sufficient number of bits (e.g. 16 bit in Internet checksum) to make undetected errors very rare. We can not guarantee completely error-free delivery because of checksumming's limitation. But you should be convinced that this should happen very rarely with a good checksumming technique.
- **Packet format** - A key part of your design is the packet format. The packet will be split into a header part and a payload part. Some common header fields include payload size, sequence number, acknowledgment sequence number, and the checksum.
- **Random numbers** - Keep in mind that many parts of the simulated environment are based on random numbers, including the message arrival intervals and message sizes. Hence two different runs with the same input parameters may not generate the same results.

Simulation code provided and your turn-in:

You can obtain the simulator from your TA's github: https://github.com/jiup/rdt_skeleton. Instructions for download and setup are provided there.

Submission

You should turn in a PDF writeup (explaining your protocol and implementation results), the new `MyRdtSender.java`, `MyRdtReceiver.java`, and any new source files you added for your implementation. You should NOT turn in the simulator code because we will use the original versions of those files in grading. If the default makefile doesn't work for you, you should also turn in a new makefile. Add necessary explanations in your writeup.

Testing:

The provided simulation files should compile and run. However, they only work correctly when there is no packet loss, corruption, or reordering in the underlying link medium. Run `rdt_sim 1000 0.1 100 0 0 0 0` to see what happens. (Running `rdt_sim` without parameters will tell you the usage of this program.) In summary, the following are a few test cases you may want to use.

- `rdt_sim 1000 0.1 100 0 0 0 0` - there is no packet loss, corruption, or reordering in the underlying link medium.
- `rdt_sim 1000 0.1 100 0.02 0 0 0` - there is no packet loss or corruption, but there is reordering in the underlying link medium.
- `rdt_sim 1000 0.1 100 0 0.02 0 0` - there is no packet corruption or reordering, but there is packet loss in the underlying link medium.
- `rdt_sim 1000 0.1 100 0 0 0.02 0` - there is no packet loss or reordering, but there is packet corruption in the underlying link medium.
- `rdt_sim 1000 0.1 100 0.02 0.02 0.02 0` - there could be packet loss, corruption, or reordering in the underlying link medium.

Of course, your goal is to make the last test case work. Keep in mind that even if your program works, it may still occasionally report errors due to the limitation of checksumming. Your program, however, should never report errors when there is no packet corruption in the underlying link medium. If your program works for some cases with parameters differing from the above examples, please specify these cases in the README file to get partial credits.

Grading

- 20% each for error-free, loss-free, and in-order delivery of your RDT implementation.
- 40% for efficiency. There are two aspects of efficiency. The primary efficiency metric is measured by the number of packets your implementation needs to pass between the sender and receiver for each particular data transfer session. The secondary efficiency metric is the transmission throughput --- the maximum amount of data that can be successfully transmitted within a time frame. We are mostly interested in the primary efficiency metric. We bring up the secondary metric to guard against a Stop-and-Wait protocol (which does not pass too many packets to complete a data transfer session but has very low throughput). A reasonable implementation of Go-Back-N protocol should get you full credit on the efficiency. We will award 10% extra credit for the student with the best efficiency (primary metric) and a reasonable throughput (that is, DO NOT turn in a Stop-and-Wait protocol).

Acknowledgements

Thanks to Kai Shen for providing the original assignment. Special thanks to Juipeng Zhang for translating the simulator from C to Java.