Lab 1 - Reliable Data Transport Protocol

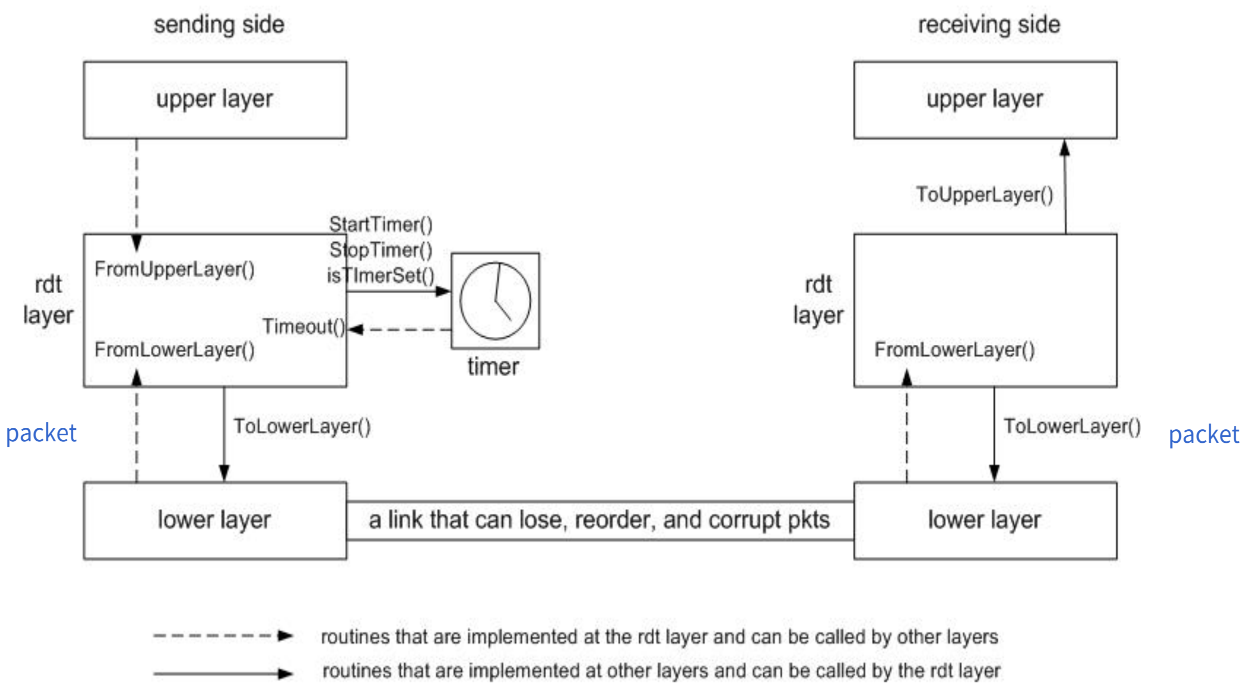
Handout: March 8, 2018
Deadline: March 18 23:00, 2018 (No extension)

Assignment overview:

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 don't have machines with OS that we can modify, your implementation will run in a simulated environment. This simulated environment is carefully constructed to follow the behavior of real systems. This also means that your program should be written in the same programming language as the simulator -- **C/C++**.

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 below:



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

```
struct message {
    int size;
    char *data;
};
```

message.size indicates the size of message.data measured in bytes. The reliable data transport protocol requires the data in the receiving node being 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:

packet between RDT layer and lower layer

```
#define RDT_PKTSIZE 128
struct packet {
    char data[RDT_PKTSIZE];
};
```

maybe need fragmentation

This means that the lower layer always deliver packets in 128-byte chunks. Therefore, fragmentation may be necessary if the message passed from the upper layer is too big to fit into a packet.

The routines you need to implement are detailed below. Such routines in real-life would be part of the operating system.

routines I need to implement

- void Sender_Init(); - sender initialization, called once at the very beginning. This routine is here to help you. Leave it blank if you don't need it.
- Sender_Final(); - sender finalization, called once at the very end. This routine is here to help you. Leave it blank if you don't need it.
- void Sender_FromUpperLayer(struct message*); - called when a message is passed from the upper layer at the sender.
- void Sender_FromLowerLayer(struct packet*); - called when a packet is passed from the lower layer at the sender.
- void Sender_Timeout(); - called when the timer expires at the sender.
- void Receiver_Init(); - receiver initialization, called once at the very beginning. This routine is here to help you. Leave it blank if you don't need it.
- void Receiver_Final(); - receiver finalization, called once at the very end. This routine is here to help you. Leave it blank if you don't need it.
- void Receiver_FromLowerLayer(struct packet *pkt); - called when a packet is passed from the lower layer at the receiver.

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.

- void Sender_StartTimer(double timeout); - start the sender timer with a specified timeout (in seconds). This timer is canceled when Sender_StopTimer() is called or a new Sender_StartTimer() is called before the current timer expires. Sender_Timeout() will be called when the timer expires.
- void Sender_StopTimer(); - stop the sender timer.

- bool Sender_isTimerSet(); - check whether the sender timer is being set, return true if the timer is set, return false otherwise.
- void Sender_ToLowerLayer(struct packet*); - pass a packet to the lower layer at the sender for delivery.
- void Receiver_ToLowerLayer(struct packet*); - pass a packet to the lower layer at the receiver for delivery.
- void Receiver_ToUpperLayer(struct message*); - deliver a message to the upper layer at the receiver.

The simulated network environment:

The overall structure of the environment is shown in the above figure. 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 and my code together 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.

How to simulate multiple virtual timers using a single physical timer

- *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, we recommend you to set it at 0.3 second. But we make no guarantee that this is the best timeout value.

recommended timeout 0.3 second

- *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). We recommend you to use a window size of 10. Again, we make no guarantee that this is the best setting.

recommended window size is 10

- *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 environments are based on random numbers, including the message arrival intervals and message sizes. So two different runs with the same input parameters may not generate the same results.

The simulation code and your turn-in:

The simulator is in C++ so your code should be in C/C++ to work with the simulator. Your will need to understand basic concepts about makefiles. Come and talk to TA if not. The RDT layer is implemented in rdt_sender.cc and rdt_receiver.cc. The current implementation assumes there is no packet loss, corruption, or reordering in the underlying link medium. You will need to enhance them to deal with all these situations. In general, you are not supposed to change rdt_receiver.h, rdt_sender.h, rdt_struct.h, and rdt_sim.cc. For debugging purpose, you may want to add more printouts in rdt_sim.cc. If you do so, definitely remember to test your program with the original rdt_sim.cc before turn-in.

As you can see, the main complexity of the simulator is enclosed in rdt_sim.cc. Our intention is that you don't have to read and understand this file to complete the assignment. You can ask for TA's assistance if you need to understand it for some reason.

Handin procedure:

You should turn in a README file, the new rdt_sender.cc, rdt_receiver.cc, and any new source files you added for your implementation. If the default makefile doesn't work for you, you should also turn in a new makefile. You should NOT turn in rdt_receiver.h, rdt_sender.h, rdt_struct.h, and rdt_sim.cc because we will use the original versions of those files in grading. Make necessary explanation in your README file for your turn-in. The README file should contain your student ID, name, e-mail address (within SJTU domain), and a description of your design and implementation strategies.

Upload your code as a gzipped tar file ,named as {Your student ID}.tar.gz ,

to ftp://exsz:public@public.sjtu.edu.cn/upload/lab1

Testing:

The original versions of all files here(http://sdic.sjtu.edu.cn/courses/ds/labs/rdt_files/rdt.tar.gz) 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. (Ruining `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.
Reference Value:

- rdt_sim 1000 0.1 100 0.15 0.15 0.15 0

```
#GBN
## Simulation completed at time 3206.00s with
   981689 characters sent
   981689 characters delivered
   86390 packets passed between the sender and the receiver
## Congratulations! This session is error-free, loss-free, and in order.

#SR
## Simulation completed at time 1656.02s with
   983959 characters sent
   983959 characters delivered
   38066 packets passed between the sender and the receiver
## Congratulations! This session is error-free, loss-free, and in order.
```

- rdt_sim 1000 0.1 100 0.3 0.3 0.3 0

```
##GBN
## Simulation completed at time 7019.09s with
   1010200 characters sent
   1010200 characters delivered
   153855 packets passed between the sender and the receiver
## Congratulations! This session is error-free, loss-free, and in order.

#SR
## Simulation completed at time 4321.40s with
   984465 characters sent
   984465 characters delivered
   59417 packets passed between the sender and the receiver
## Congratulations! This session is error-free, loss-free, and in order.
```

Grading:

- 10% for README.
- 40% each for error-free, loss-free, and in-order delivery of your RDT implementation.
- 50% 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, thus has very low throughput). A reasonable implementation of either Go-Back-N or Selective Repeat should get you full credits on the efficiency.