CS422 Lab2: Reliable Transport Protocol

Due Date: 23:59:59 PM Oct 16, 2025
Total Points: 70 points

In this programming assignment, you will be writing the sending and receiving transport-level code in **C** to implement one simple reliable data transfer protocols, **RDT 3.0**. Note that it represent the versions of reliable data transfer protocols, the alternating-bit-protocol.

1 Background

Please read chapter 3.4 and lecture slides.

1.1 Programming Environment in C

You must complete this assignment individually. For this task, you will write **C code**. We have provided two sample files, `prog.c` and `prog.h`, and you are required to complete the **first seven functions**. In our scenario, we only have one **sender A** and one **receiver B**.

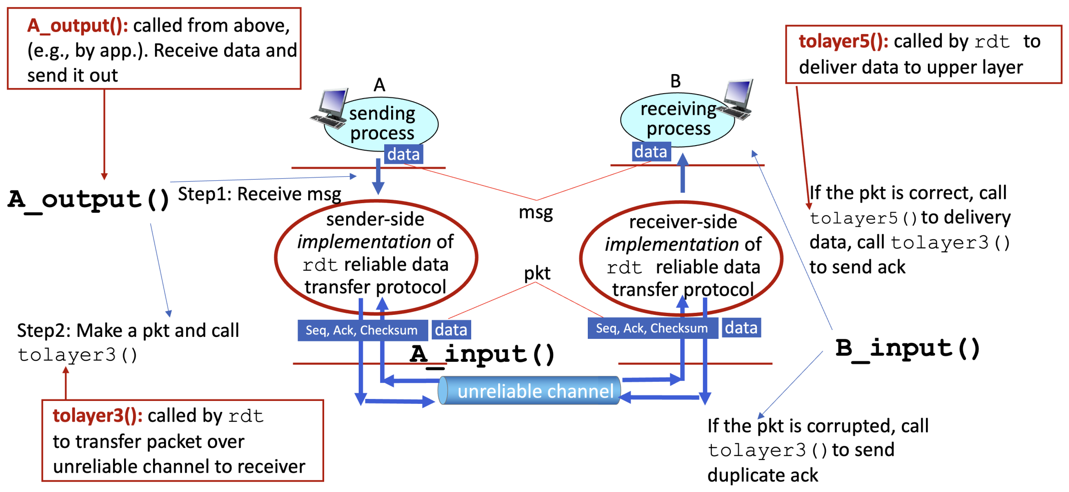
CS422@Purdue

1.2 Software Interfaces

Within the `prog.c` file, we have already implemented several utility functions for you. You may call these functions as needed when implementing your solutions.

1. **starttimer(calling_entity, increment)**
   - `calling_entity`: Either `0` (for starting the A-side timer) or `1` (for starting the B-side timer).
   - `increment`: One `float` value indicating the time units before the timer interrupts.
   - **Notes:**
     - The A-side timer **should only be started (or stopped) by A-side functions**, and similarly for the B-side timer.
     - Assume that a packet takes an average of 15 time units to reach the other side in the absence of congestion. Therefore, please set the timeout value to **15**.
2. **stoptimer(calling_entity)**
   - `calling_entity`: Either `0` (to stop the A-side timer) or `1` (to stop the B-side timer).
3. **tolayer3(calling_entity, packet)**
   - `calling_entity`: Either `0` (for A-side send) or `1` (for B-side send).
   - `packet`: A structure of type `pkt`.
   - **Description:** Calling this function sends the packet into the simulated network, destined for the other entity.
4. **tolayer5(calling_entity, message)**
   - `calling_entity`: Either `0` (for A-side delivery to layer 5) or `1` (for B-side delivery to layer 5).
   - `message`: A structure of type `msg`.
   - **Description:** Calling this function delivers data to application layer.
   - **Notes:**
     - For **unidirectional** data transfer (B is the receiver), this will only be called with `calling_entity = 1` (delivery to the B-side).
     - This effectively passes application-layer data upward.

1.3 Introducing Seven Functions Need To Be Implemented By You



A_output(): called from above, (e.g., by app.). Receive data and send it out
A_output()
Step1: Receive msg
Step2: Make a pkt and call tolayer3()
tolayer3(): called by rdt to transfer packet over unreliable channel to receiver
A
sending process
data
sender-side implementation of rdt reliable data transfer protocol
Seq, Ack, Checksum
data
A_input()
msg
pkt
unreliable channel
B
receiving process
data
receiver-side implementation of rdt reliable data transfer protocol
Seq, Ack, Checksum
data
tolayer5(): called by rdt to deliver data to upper layer
If the pkt is correct, call tolayer5() to delivery data, call tolayer3() to send ack
B_input()
If the pkt is corrupted, call tolayer3() to send duplicate ack

Figure 1: Here is an overview plot to give you a clear view.

CS422@Purdue

The procedures you will write are for the sending entity (A) and the receiving entity (B). Only **unidirectional** transfer of data (from A to B) is required. Of course, the B side will have to send acknowledge packets to A. Your are required to implement the following functions and their requirements are described below. These procedures will be called by (and will call) functions that have been provided to you which emulate a network environment. The unit of data passed between the application layer and your transport protocol is a message, which is declared as:

struct msg {   char data[20]; };

Your sending entity (A) will receive data in 20-byte chunks from application layer; your receiving entity (B) should deliver 20-byte chunks of correctly received data to its application layer.

The unit of data passed between the transport layer (layer 3) and the network layer is the packet, which is declared as:

```
struct pkt {
        int seqnum;
        int acknum;
        int checksum;
        char payload;
};
```

The payload of the **pkt** is the **msg** received from the upper application layer. The header infomation will be used by your protocols to ensure reliable delivery.

The seven functions you will implement are detailed below. Such procedures in real-life would be part of the operating system, and would be called by other procedures in the operating system.

1. **A_output(message)**, where message is a structure of type msg, containing data to be sent to the B-side. This function 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's application layer.

2. **A_input(packet)**, where packet is a structure of type **pkt**. This routine will be called whenever an acknowledgement packet sent by the B side (i.e. as a result of a tolayer3() being called by a B-side procedure) arrives at the A side. packet is the (possibly corrupted) packet sent from the B-side.

3. **A_timerinterrupt()** This function will be called when A's timer expires (thus generating a timer interrupt). You'll use this function to control the retransmission of packets. See **starttimer()** and **stoptimer()** for how the timer is started and stopped.

4. **A_init()** This function will be called once, before any of your other A-side routines are called. It can be used to do any required initialization.

5. **B_input(packet)** This function will be invoked whenever a packet sent by the A-side (as a result of **tolayer3()** being called from an A-side routine) arrives at the B-side. The variable packet is a structure of type **pkt**, which may have been corrupted by the unreliable network.

6. **B_init()** This function will be called once, before any of your other B-side functions are called. It can be used to do any required initialization.

7. **B_timerinterrupt()** No need to modify this function. In rdt3.0, only the sender (A) uses a timer for retransmissions. This function is included in the simulator for consistency—because in reality, either side (A or B) could act as a sender and require a timer.

2 Programming Assignments

2.1 Part A: Simulated Network Environment (15 points)

The code offered for this lab has almost implemented this simulated network environment but you need to get it run as required below.

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 our procedures together and run the resulting program, you will be asked to specify values regarding the simulated network environment:

CS422@Purdue

1. **Number of messages to simulate:** Our emulator 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.

2. **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.

3. **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 payload, sequence, ack, or checksum fields can be corrupted. Your checksum should thus include the data, sequence, and ack fields. 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 8 bit integer and just add them together).

4. **Tracing:** Setting a tracing value of 1 or 2 will print out useful information about what is going on inside the emulation (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 our own emulator debugging purposes. A tracing value of 2 may be helpful to you in debugging your code. You should keep in mind that real implementors do not have underlying networks that provide such nice information about what is going to happen to their packets!

5. **Average time between messages from sender's layer5:** 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.

2.2 Part B: RDT3.0 (45 points)

(30 points) RDT3.0 is one Alternating Bit Protocol. You are to implement the following procedures, **A_output(), A_input(), A_timerinterrupt(), A_init(), B_input(), and B_init()** which together will implement a stop-and-wait (i.e., the alternating bit protocol) unidirectional transfer of data from the A-side to the B-side. Your protocol can use both ACK and NACK messages (you can only use ACK messages as well).

You should put your procedures in a file called **prog_rdt.c**. The files, containing the emulation routines and the stubs for your procedures **prog.c** and **prog.h** are available in the zipped folder. Note that **prog.c** is an example for you. We will compile and test your **prog_rdt.c**, not **prog.c**.

Your program should be executed as follows:

./prog_rdt [num_sim] [prob_loss] [prob_corrupt time] [average time between messages] [debug_level], Such as **./prog_rdt 3 0 0 100 2**

(15 points) Try the following commands and report the output and mark the reactions of the one packet loss and one packet corruption. You can find the sample output in sample_output1.txt, sample_output2.txt and sample_output3.txt.

1. ./prog_rdt 5 0 0 100 2
2. ./prog_rdt 5 0.3 0 100 2
3. ./prog_rdt 5 0 0.3 100 2

Tips: While debugging and testing, you should choose a very large value for the average time between messages from the sender's layer 5. This ensures that the sender is never called while it still has an outstanding, unacknowledged message in transit. We suggest using a value of 1000. You should also perform a check in your sender to make sure that when A_output() is called, there is no message currently in transit. If there is, you can simply ignore (drop) the data being passed to the A_output() routine.

For your sample output, your procedures must print out a message whenever an event occurs at your sender or receiver i.e on a message/packet arrival or a timer interrupt as well as any action taken in response. You should also print the content of the messages and payload of packets.

Output requirements Note that all of your programs will be tested at the trace level of 2. So please printout all useful information in your report for each message at the sender/receiver side at the trace level of 2, such as **Seq Num**, **Ack Num**, **Checksum**, **Payload**. This will do a lot help for you to debug your codes and see how your messages are transmitted. Also, it will help the grading more efficient and fair because it helps the grader to observe how your program is implemented.

CS422@Purdue

2.3 Part C: Answer Questions (10 points)

Please answer the following questions in the report.

1. Run your program from Part B and set messages to simulate to be 10, packet corruption probability to be 0, and average time between messages from sender's layer5 to be 1000. Experiment with different packet loss probabilities: 0, 0.2, 0.4, and 0.6. For each packet loss probability, repeat the experiment 10 times, record the total number of packets sent from Layer 3 at the sender A, and compute the average. Present the results in a bar graph with the x-axis representing packet loss probability and the y-axis representing the average number of packets sent from Layer 3.

2. Run your program from Part B and set messages to simulate to be 10, packet loss probability to be 0, and average time between messages from sender's layer5 to be 1000. Experiment with different packet corruption probabilities: 0, 0.2, 0.4, and 0.6. For each packet corruption probability, repeat the experiment 10 times, record the total number of packets sent from Layer 3 at the sender A, and compute the average. Present the results in a bar graph with the x-axis representing packet corruption probability and the y-axis representing the average number of packets sent from Layer 3.

3. Which factor, packet corruption probability or packet loss probability, has a greater impact on retransmission at layer 3? Why?

3 Helpful Hints

1. Note that any shared "state" among your functions needs to be in the form of global variables. Note also that any information that your procedures need to save from one invocation to the next must also be a global (or static) variable. For example, your routines will need to keep a copy of a packet for possible retransmission. It would probably be a good idea for such a data structure to be a global variable in your code. Note, however, that if one of your global variables is used by your sender side, that variable should NOT be accessed by the

receiving side entity, since in real life, communicating entities connected only by a communication channel can not share global variables.

2. There is a float global variable called *time* that you can access from within your code to help you out with your diagnostics msgs.
3. **START SIMPLE.** Set the probabilities of loss and corruption to zero first and test your routines. Better yet, start out by designing and implementing 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.
4. Note that do not change the payload given to the student in the original code. You can try to run the origin code at the trace level of 3 to see what data is given to the sender side.

4 Submission

Your submission directory should contain:

1. All source files i.e. prog_rdt.c, as well as their header files.
2. Sample output for RDT as output_rdt.pdf.
3. A README file is required to explain how to compile their files, especially when these codes must be compiled with the extra libraries.

Note: Please document any reasonable assumptions you make or information in a README file, if any parts of your assignment are incomplete.

5 Support

Questions about the assignment should be posted on Campuswire or asked during PSOs.

CS422@Purdue