CS536 Lab3: Reliable Transport Protocol

Due Date: Oct 18, 2020 (23:59:59 PM),, Total Points: 150 points

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

1 Instructions

Please read chapter 3.4 and lecture slides.

1.1 Programming Environment in C

You must work individually on this assignment. You will write **C code** that compiles and operates correctly on a Linux machine.

For this assignment, because we don't have standalone machines (with an OS that you can modify), your code will have to execute in a **simulated hardware/software environment**. Along with this lab assignment, we have provided you the programming interfaces for the routines you are going to implement. Despite this simulation environment, these codes that would call your entities from above and from below is very close to what is done in an actual UNIX environment. Stopping/starting of timers are also simulated, and timer interrupts will cause your timer handling routine to be activated.

1.2 Routines To Be Implemented

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 packets to A to acknowledge (positively or negatively) receipt of data. Your routines are to be implemented in the form of the procedures described below. These procedures will be called by (and will call) procedures that have been provided to you which emulate a network environment. The unit of data passed between the upper layers and your protocols is a message, which is declared as:

```
struct msg {        char data[20];
};
```

This declaration, and all other data structure and emulator routines, as well as stub routines (i.e., those you are to complete) are in the file, **prog.c** and its header, **prog.h**, described later. Your sending entity will thus receive data in 20-byte chunks from layer5; your receiving entity should deliver 20-byte chunks of correctly received data to layer5 at the receiving side.

The unit of data passed between your routines and the network layer is the packet, which is declared as:

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

Your routines will fill in the payload field from the message data passed down from layer5. The other packet fields will be used by your protocols to ensure reliable delivery.

The routines you will write 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 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.

2. **A_input(packet)**, where packet is a structure 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 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 routine will be called when A's timer expires (thus generating a timer interrupt). You'll use this routine to control the retransmission of packets. See starttimer() and stoptimer() below for how the timer is started and stopped.

4. **A_init()** This routine 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)**, where packet is a structure 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 called by a A-side procedure) arrives at the B-side. packet is the (possibly corrupted) packet sent from the A-side.

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

1.3 Software Interfaces

The procedures described above are the ones that you will implement. We have implemented the following routines which will be called by your routines:

1. **starttimer(calling entity, increment)**, where calling entity is either 0 (for starting the A-side timer) or 1 (for starting the B side timer), 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.

2. **stoptimer(calling entity)**, where calling entity is either 0 (for stopping the A-side timer) or 1 (for stopping the B side timer).

3. **tolayer3(calling entity, packet)**, where calling entity is either 0 (for the A side send) or 1 (for the B side send), and packet is a structure of type pkt. Calling this routine will cause the packet to be sent into the network, destined for the other entity.

4. **tolayer5(calling entity, message)**, where calling entity is either 0 (for Aside delivery to layer 5) or 1 (for B-side delivery to layer 5), and message is a structure of type 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 5.

2 Programming Assignments

2.1 Simulated Network Environment (30 points)

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:

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

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 loss 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.

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 RDT3.0 (60 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 choose a very large value for the average time between messages from 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. We suggest you choose 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.

You should put your procedures in a file called **prog2_rdt.c**. The files, containing the emulation routines and the stubs for your procedures **prog2.c** and **prog2.h** are available on blackboard.

In addition to your source code, you should submit a sample output along with your source code as **output_rdt.pdf**. 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. You have to hand in output for a run up to the point (approximately) when 10 messages have been ACK'ed correctly at the receiver, a loss probability of 0.1 (recommended, no smaller than 0.05), and a corruption probability of 0.3 (recommended, no smaller than 0.1) and a trace level of 2. In case all the messages are too long to print, please retain the key messages (and skip those unimportant ones) in your sample output. If you are not sure what messages are key to show the RDT working procedure, please contact TAs. You may find that it is very useful to annotate your key outputs with a colored pen showing how your protocol correctly recovered from packet loss and corruption.

2.3 Selective-Repeat (60 points)

You are to implement Selective-Repeat, which is a pipelined protocol. You are to implement the procedures, **A_output(), A_input(), A_timerinterrupt(), A_init(), B_input() and B_init()** which together will implement a pipelined, unidirectional transfer of data from the A-side to the B-side.

In this lab, please use a window size of 8. We recommend you not use the magic number but a pre-configured constant. This way, you will be able to see how it performs with regards to the window size if you have interests.

Your protocol use only ACK messages. Consult the alternating-bit-protocol version of this assignment above for information about how to obtain the network emulator. We would STRONGLY recommend that you first implement the easier version (RDT) and then extend your code to implement this harder pipelined version . *Believe us, it will save your time!.*

There are some **new** considerations for your pipelined code (which do not apply to RDT3.0) are: **A_output(message)**, where message is a structure 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 and/or receiver, due to the nature of Selective-Repeat.

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_timerinterrupt() 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.

You should put your SR procedures in a file called **prog2_sr.c**.

In addition to your source code, you have to submit output as output_sr.pdf for a run that was long enough so that at least 20 messages were successfully transferred from sender to receiver (i.e., the sender receives ACK for these messages) transfers, a loss probability of 0.2 (recommended, no smaller than 0.1), and a corruption probability of 0.2(recommended, no smaller than 0.1), and a trace level of 2, and a mean time between arrivals of 10. Please also print the message numbers in the sender's and receiver's window. In case all the messages are too long to print, please retain the key messages (and skip those unimportant ones) in your sample output. You may find that it is very useful to annotate your key outputs with a colored pen showing how your protocol correctly recovered from packet loss and corruption.

3 Helpful Hints

1. **Checksumming:** You can use whatever approach for checksumming you want. Remember that the sequence number and ack field can also be corrupted. 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).

2. Note that any shared "state" among your routines 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.

3. There is a float global variable called *time* that you can access from within your code to help you out with your diagnostics msgs.

4. **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 Submission

You will be submitting your assignment on blackboard. Your submission directory should contain:

1. All source files i.e. prog2_rdt.c, and prog2_sr.c, as well as their header files.

2. Sample output for both versions as output_rdt.pdf, and output_sr.pdf.

3. You also need to briefly modify the program to make sure that your output contains enough details to show how the two protocols work. We should be able to see **how** you handle lost and corrupt packages.

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