Programming Assignment 2: Implementing a Reliable Transport Protocol

Overview

[See posted grading criteria.] **You are encouraged to work in teams of two, but you are expected to clearly delineate the work done by each team member in your documentation. The work should be roughly equal and collaborative**. **Please review the academic conduct rules noted in the syllabus. This assignment is part of BU CS 655 material and is provided for educational purposes. Please do NOT share or post this assignment handout or your solution, on any public site, *e.g.* github. Of course, you are not allowed to share your solution with classmates, other than your teammate (if any).**

In this second programming assignment, in the first part, you will be writing the sending and receiving transport-level code for implementing the Selective-Repeat (with cumulative ACKs) reliable data transfer protocol. In the second part, you will be implementing Go-Back-N (with the SACK option) and asked to compare it with SR (with cumulative acks).

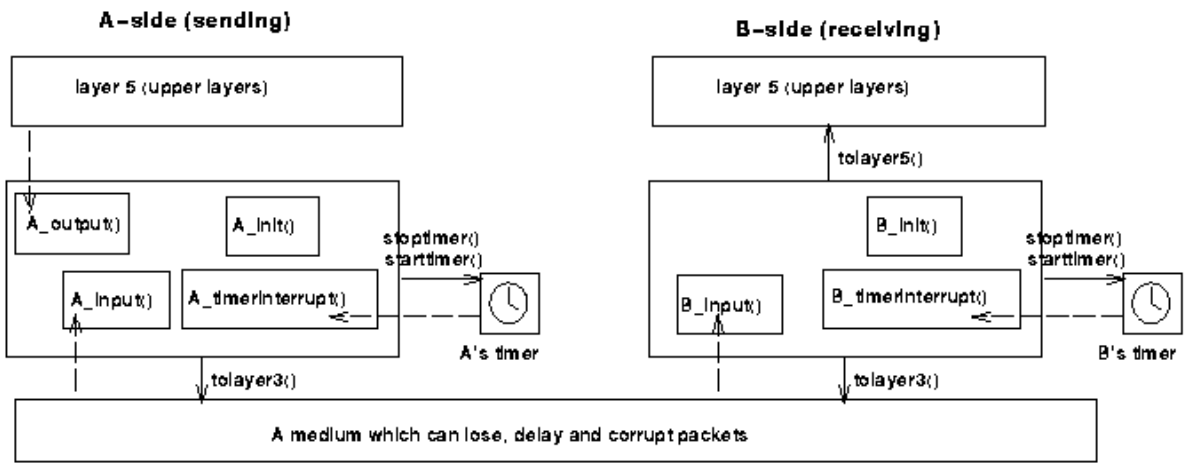
This should be **fun** since your implementation will differ very little from what would be required in a real-world situation.

Since we do not have standalone machines (with an OS that you can modify), your code will have to execute in a simulated hardware/software environment. However, the programming interface provided to your routines (i.e., the code that would call your entities from above (i.e., from layer 5, denoting the application layer) and from below (i.e., from layer 3, denoting the network layer) is very close to what is done in an actual UNIX environment. (Indeed, the software interfaces described in this programming assignment are much more realistic that the infinite loop senders and receivers that the K&R textbook describes). Stopping/starting of timers are also simulated, and timer interrupts will cause your timer handling routine to be activated.

The description of the routines below is given in C. See text at end regarding the C++ and Java versions of this assignment. Note that you do not need a network connection to run this assignment, so you can do it pretty much on any machine you would like. Recall however, that you need to make sure that your program ultimately compiles and runs correctly on our csa1/csa2/csa3 machines where it will be graded.

The routines you will write

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 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 you are given which emulate a network environment. The overall structure of the environment is shown in the following Figure:



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, **pa2.c,** 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 (and in sequence) 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[20];
    };
```

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 insure reliable delivery, as we've seen in class.

The routines you will write are detailed below. As noted above, such procedures in real-life would be part of the operating system, and would be called by other procedures in the operating system.

- **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 insure that the data in such a message is delivered in-order, and correctly, to the receiving side upper layer.
- **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 done by a B-side procedure) arrives at the A-side. `packet` is the (possibly corrupted) packet sent from the B-side.
- **A_timerinterrupt()** This routine will be called when A's timer expires (thus generating a timer interrupt). You'll probably want to use this routine to control the retransmission of packets. See `starttimer()` and `stoptimer()` below for how the timer is started and stopped.

- **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.
- **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 done by an A-side procedure) arrives at the B-side. `packet` is the (possibly corrupted) packet sent from the A-side.
- **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.

Software Interfaces

The procedures described above are the ones that you will write. You are given the following routines, which can be called by your routines:

- **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 *double* 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. Note that since we only care about unidirectional data transfer from A to B, only the A-side should maintain such 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.
- **stoptimer(calling_entity),** where `calling_entity` is either 0 (for stopping the A-side timer) or 1 (for stopping the B-side timer).
- **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.
- **tolayer5(message),** where `message` is a structure of type `msg` to be passed up to layer 5 of the B-side. Note that we simplified the interface here so you don't need to explicitly specify `calling_entity` (A-side or B-side) since we are only dealing with unidirectional data delivery to the B-side.

The simulated network environment

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

- **Number of messages to simulate.** The emulator (and your routines) will stop as soon as this number of messages has 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.
- **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.
- **Corruption.** You are asked to specify a packet corruption probability. A value of 0.2 would mean that FOR THOSE PACKETS THAT ARE NOT LOST, 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.

- **Average interarrival 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.

- **Window size.** This allows you to set a limit on the send window.

- **Retransmission timeout.** This allows you to set the value of the retransmission timeout.

- **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 were used for emulator-debugging purposes. A tracing value of 2 may be helpful to you in debugging your code. You should keep in mind that *real* implementers do not have underlying networks that provide such nice information about what is going to happen to their packets!

- **Random seed.** This allows you to set an initial seed for the generation of random numbers used to schedule packet arrivals, delays on the network, loss and corruption events.

Part I: Implementing Selective-Repeat (with cumulative ACKs)

You are to write the procedures, `A_output()`, `A_input()`, `A_timerinterrupt()`, `A_init()`, `B_input()`, and `B_init()` which together will implement a Selective-Repeat (SR) unidirectional transfer of data from the A-side to the B-side. Your protocol should use only ACK (i.e. no NACK) packets. The receiver should be able to buffer out-of-order (OOO) packets, and send **cumulative** ACKs. The sender should retransmit only the next missing (unACK'ed) packet either on a timeout or duplicate ACK. [See lecture's slide on "Selective Repeat, Explicit Retransmission".]

You should put your procedures in a file called pa2.c. You will need the initial version of this file, containing the emulation routines you are given, and the stubs for your procedures. You can obtain this program here together with a "compile" file - please use that compile file (which you may modify) and make sure your program runs on csa1, csa2 or csa3.

This lab can be completed on any machine supporting C. It makes no use of UNIX features. (You can simply copy the pa2.c file to whatever machine and OS you choose).

It is **STRONGLY** recommended that you first implement the Alternating-Bit (stop-and-wait) protocol and then extend your code to implement the harder version (SR). It will **not** be time wasted! However, some new considerations for your SR code (which do not apply to the Alternating-Bit protocol) 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. Also, you'll need buffering in your sender because of the nature of SR: 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_input()** This routine will be called when A receives an ACK packet from B. On receiving a packet from B, A needs to take into account the following:

    1. Check if the ACK packet is corrupted and take appropriate actions
    2. If you get a new ACK, slide the window and send any new data packets waiting
    3. If duplicate ACK, retransmit the first unACK'ed data packet

- **B_input()** This routine will be called when B receives a data packet from A. On receiving a packet from A, B needs to take into account the following:

    1. Check if the packet is corrupted and take appropriate actions
    2. If the data packet is new, and in-order, deliver the data to layer5 and send ACK to A. Note that you might have subsequent data packets waiting in the buffer at B that also need to be delivered to layer5
    3. If the data packet is new, and out of order, buffer the data packet and send an ACK
    4. If the data packet is duplicate, drop it and send an ACK

- **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. An easy way to use a single timer is to associate it with the whole window, e.g. restart the timer whenever you send a data packet.

What to Submit

You should use Gradescope to submit a design document and a testing document (with sample outputs).

For your sample output, your procedures might print out a message whenever an event occurs at your sender or receiver (a message/packet arrival, or a timer interrupt) as well as any action taken in response.

You should submit output 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). Example input values: number of messages = 1000, window size = 8, a retransmission timeout = 30, a random seed = 1234, a loss probability = 0.1, a corruption probability = 0.1, a trace level = 3, and a mean time between arrivals = 200.

You should annotate parts of your output with color markings showing how your protocol correctly recovered from packet (data and ack) loss and corruption. Your traces should show the following cases:

- Case1: your protocol works for no loss + no corruption
- Case2: identify (on output trace) case where ack is lost/corrupted and a later cumulative ack moves the sender window by more than 1
- Case3: identify (on output trace) case where data packet is lost/corrupted, and data is retransmitted after RTO
- Case4: identify (on output trace) case where data packet is lost/corrupted, and data is retransmitted after receiving duplicate ack
- Case5: identify (on output trace) case where data packet is lost/corrupted, and the retransmitted data is delivered and a cumulative ack moves the sender window by more than 1

For a sample-annotated output, see posted trace on Piazza.

Your program should also gather statistics and print them at the end of the program using function Simulation_done() in the provided code. Your program should print the following:

- Number of original packets transmitted by A
- Number of retransmissions by A
- Number of data packets delivered to layer 5 at B
- Number of ACK packets sent by B
- Number of corrupted packets
- Ratio of lost packets:
  Lost ratio = (retransmissions by A – corrupted packets) / ((original packets by A + retransmissions by A) + ACK packets by B)
- Ratio of corrupted packets:
  Corruption ratio = (corrupted packets) / ((original packets by A + retransmissions by A) + ACK packets by B - (retransmissions by A – corrupted packets))
- Average RTT: Average time to send a packet and receive its ACK for a packet that has **not** been retransmitted. Note that data packets that are ACKed by the ACK of a subsequent packet are not part of this metric.
- Average communication time: Average time between sending an original data packet and receiving its ACK, even if the data packet is retransmitted.

Explain how you decided on the initial value of your retransmission timer, and how that value changes during the simulation. (Note: a static retransmission timer value is acceptable for this assignment.)

Produce two meaningful figures that show the average time to communicate a packet, that is, the time from when the packet is first sent into the network until the time when the sender receives the first ack for that packet (dependent variable), as a function of loss (independent variable) **and** as a function of corruption (independent variable). For your results to be statistically meaningful, you should present the average performance over several independent runs (i.e. with different seeds for the random number generator) and compute confidence intervals (say, 90%) as a function of the sample mean (average) and sample standard deviation (Note: you may want to review these online notes on how to compute confidence intervals.)

For details on grading, see the grading criteria on Piazza.

Part II: GBN with SACK option

Also simulate unidirectional transfer of data using a Go Back N version that uses a SACK option (selective acks), in addition to cumulative acks (used in the initial part of this assignment.) Use a SACK option of size 5 by adding a `int sack[5]` field to the `pkt` structure (e.g., in the Java code, you need to add `int[] sack` to Packet.java and make appropriate changes to the engine code to accommodate it). Here the sender would behave like a GBN sender but retransmit *all* outstanding unACK'ed packets that have *not* been selectively ACK'ed, not just the next missing data packet. This is similar to TCP SACK.

Note that the SACK option has a limit of 5, so an ACK packet can selectively acknowledge at most 5 packets. You can use this SACK option by selectively acknowledging the latest five data packets that have been successfully received at the receiver.

Submit traces, statistics, etc., which clearly demonstrate that your code performs correctly. It is your responsibility to provide output that convincingly demonstrates that your code works correctly (similar to what was asked in the initial part of the assignment). Make sure you demonstrate the differences in

behavior between SR (with cumulative acks) and GBN (with SACK option) by appropriate annotations of your traces and by comparing various performance metrics (e.g., throughput, goodput, and average packet delay).

In your documentation, indicate your team member, if any, along with the division of labor.

Helpful Hints and the like

- **Checksumming.** You can use whatever approach for checksumming you want. Remember that the sequence number and ack fields 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).
- 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 your sender side uses one of your global variables, that variable should **NOT** be accessed by the receiving side entity, since in real life, communicating entities connected only by a communication channel cannot share global variables.
- There is a double global variable called *time* that you can access from within your code to help you out with your diagnostics messages.
- **START SIMPLE.** Set the probabilities of loss and corruption to zero and test out your routines. Better yet, design and implement 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.
- **Debugging.** It is recommended that you set the tracing level to 2 and put LOTS of printf's in your code while you're debugging your procedures.

- The routine **Simulation_done()** is called at the end of the simulation and could be used to print final statistics.

- Make sure that the limit on the sequence number (LIMIT_SEQNO), after which the sequence number wraps around, is set appropriately.

Q&A

http://www.cs.bu.edu/fac/matta/Teaching/cs655/F22/PA2/PA2_QA.htm

---

JAVA files

Here are the class files and source code for the JAVA files that you'll need if you want to do the assignment in Java rather than C.

Some notes:

- *NetworkSimulator* is an abstract class that is the bulk of the simulator. *StudentNetworkSimulator* is the only class that you will have to modify. **[For part II, you will also need to modify *Packet.java*]**

- *Packet, Message, Event,* and *EventListImpl* are support classes. EventList is an interface. *Project* is the "driver" for the whole thing.
- You only need the .class files, and *StudentNetworkSimulator.java*. The other sources are there in case you are interested.
- *StudentNetworkSimulator.java* contains inline comments documenting the interfaces of the other classes that you will need. These class files will need to be in the CLASSPATH (which will happen automatically if you edit and compile StudentNetworkSimulator.java in the same directory as the other class files).

C++ files

Here is the C++ version if you want to do the assignment in C++ rather than C.
