CIS*3210 Computer Networks

Assignment 2: Reliable Data Transfer

Due Friday, November 15, 2019 @ 11:59pm

Overview

This assignment can be done **in teams of two**. In this assignment, you will be writing the sending and receiving transport-level code for implementing one reliable data transfer protocol, Stop-And-Wait (SAW). This programming assignment should be fun since your implementation will differ very little from what would be required in a real-world situation.

Since we don't have standalone machines with an OS that you can modify, your code will execute in a simulated hardware/software environment. However, the programming interface provided to your routines, i.e., the code that would call your routines 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 (timeouts) will cause your timer handling routine to be activated.

Getting Started

First, upload the starter code **pa2starter.c** to your development computer. You will use this starter code in this assignment for implementing the Stop-And-Wait protocol. Read the starter code to understand what you'll need to do. **When developing your program, do not change the parts that are marked as DO NOT MODIFY in the starter code.**

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 the 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 are provided in the starter code (which emulate a network environment). The overall structure of the environment is shown in the following picture.

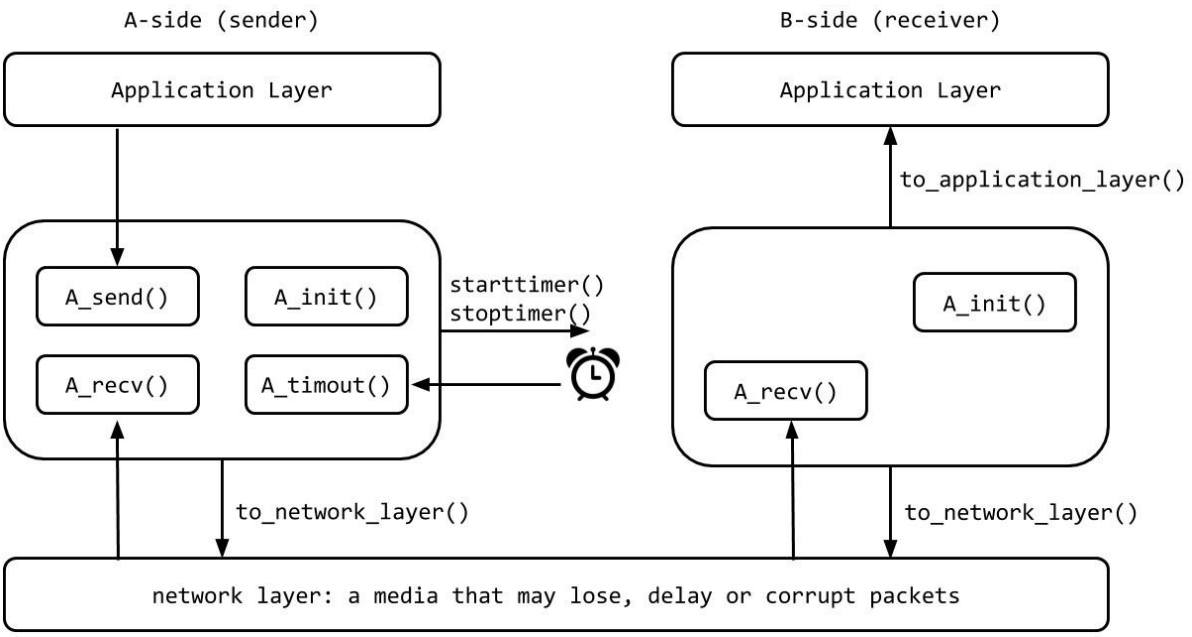

Figure 1: Structure of the emulated environment

The unit of data passed between the upper layers and your protocols is a message, which is declared as:

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

Your sending entity A will receive data in chunks of size MSG_SIZE bytes from the application layer; your receiving entity B should deliver same-sized chunks of correctly received data to application layer 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[MSG_SIZE];
};
```

Your routines will fill in the payload field from the message data passed down from the application layer. The other packet fields will be used by your protocols to insure reliable transfer, as we've seen in class.

The routines you will write

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_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.
- A_send(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 application 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's application layer.
- A_recv(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 to_network_layer() being done by a B-side procedure) arrives at the A-side. The parameter packet is the (possibly corrupted) packet sent from the B-side.
- A_timeout(): 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.
- 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.
- B_recv(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 to_network_layer() being done by an A-side procedure) arrives at the B-side. The parameter packet is the (possibly corrupted) packet sent from the A-side.

Software Interfaces

The following routines (provided in the starter code) 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 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.
- stoptimer(calling_entity), where calling_entity is either 0 (for stopping the A-side timer) or 1 (for stopping the B side timer).
- to_network_layer(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.

- to_application_layer(calling_entity,message), where calling_entity is either 0 (for A-side delivery to the application layer) or 1 (for B-side delivery to the application layer), 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 the application layer.

The simulated network environment

A call to procedure to_network_layer() sends packets into the medium (i.e., into the network layer). Your procedures A_recv() and B_recv() 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 run your program, you will specify values regarding the simulated network environment:

- **Number of messages to simulate**: The simulator will stop as soon as this number of messages have been passed down from the application layer, 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 probability**: a value of 0.1 would mean that one in ten packets (on average) are lost.
- **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.
- **Average interval between messages from sender's application layer**: you can set this value to any non-zero, positive value. Note that the smaller the value you choose, the faster packets will be be arriving to your sender.

The above the parameters will be passed to your program as command-line arguments in the following format (assuming the name of the executable is saw:

```
./saw num_msgs loss_prob corrupt_prob interval
```

For example, you could run

```
./saw 100 0.1 0.2 1000
```

Task #1: Draw the FSM for the receiver side of protocol rdt 3.0

Review the rdt3.0 protocol in Section 3.4. Draw the missing receiver FSM for rdt 3.0 protocol.

Task #2: Implementing the Stop-And-Wait protocol

Make a copy of the starter code and name it **saw.c**. You will implement the Stop-And-Wait protocol in this file.

You will implement the **rdt3.0** protocol that we learned in class. You are to complete the six routines mentioned above to support **reliable unidirectional transfer of data from the A-side to the B-side**.

You should choose a very large value for the average interval between messages from sender's application layer, so that your sender is never called while it still has an outstanding, unacknowledged message it is trying to send to the receiver. I'd suggest you choose a value of 1000. You should also perform a check in your sender to make sure that when A_send() is called, there is no message currently in transit. If there is, you can simply ignore (drop) the data being passed to the A_send() routine.

Once you have tested your implementation and are convinced that it is correct. You will write up some experimental results in a report named **report.pdf**. You will present and argue that your implememented protocol is correct using the printout of a test case with 10 messages successfully transferred, a loss probability of 0.2, a corrupt probability of 0.3, and an average interval of 1000. More details about **report.pdf** in the "Submissions" section below.

Useful tips

Below are some tips that you might find helpful.

1. **Checksumming**. You can use whatever approach for checksumming you want. Remember that the sequence number and ack field can also be corrupted, so your checksum should take them into consideration.
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 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.
5. The **TRACE** variable defines the verbosity level of the printout. You may adjust it

according to your need. Read the code to understand what each trace level means.

6. **Random Numbers**. The emulator generates packet loss and errors using a random number generator. Our past experience is that random number generators can vary widely from one machine to another. You may need to modify the random number generation code in the emulator we have supplied you. Our emulation routines have a test to see if the random number generator on your machine will work with our code. If you get an error message:

   > "*It is likely that random number generation on your machine is different from what this emulator expects. Please take a look at the routine jimsrand() in the emulator code*"

   then you'll know you'll need to look at how random numbers are generated in the routine jimsrand(); see the comments in that routine.

   Note that we have tested our school Linux server *linux.socs.uoguelph.ca* as well as alternative programming environment which is a pre-configured Debian Virtual Machine that has been recommended for the convenience of your code development. These environment is compatible with random number generation. Nevertheless, this assignment can be completed on any machine supporting C as long as it is compatible with random number generation needed for the emulation. It makes no use of UNIX features. (You can simply copy the "pa2starter.c" file to whatever machine and OS you choose).

Requirements

1. Below are some specific requirements your code needs to satisfy just so that it can be properly marked by the TA.
2. The code must be tested on the school Linux server before submission.
3. Your code must be compiled successfully using the following command.

   ```
   gcc saw.c
   ```

   Note that Code that does not compile will receive zero mark.
4. Your code must compile without warning or error with gcc 8.3.0 (the version on linux.socs.uoguelph.ca).
5. The TA will use commands like the following to test your program. Make sure your program follows this format of the arguments.

   ```
   ./saw 100 0.2 0.3 10
   ```
6. You may NOT add any include other than what's already included in the starter code.
7. You may NOT add printouts to stderr in the code your write, i.e., do NOT write any line like

   fprintf(stderr, ...) . However, feel free to use printf which prints to stdout.
8. Read the comments in the starter code carefully, and do not change the parts that are marked as DO NOT MODIFY.

9. Keep your implementation simple, i.e., only implement what's necessary for the corresponding protocol. Unnecessary implementations (e.g., using a larger range of sequence numbers than necessary, using an unnecessary type of packet) may cause deductdion in marks.

Submission

You will submit the following two files using the “DropBox” on the CourseLink.

1. saw.c: the code that implements the Stop-And-Wait protocol.
2. A Makefile that compiles the program
   - make saw: creates executable saw
   - make clean: deletes the executable and intermediate files
   - make zip: creates a zip archive with all your deliverables
3. A README file that provides all the instructions necessary to run your assignment
4. report.pdf: the report that includes the following,
   - The names and student IDs of all group members.
   - The sender FSM for rdt 3.0 protocol
   - An overall description of what's done and what's not done in your submission, and any other information that you think is relevant for the TA to know while marking.
   - An example showing that your Stop-And-Wait is working correctly. Use the printout of a test case with 10 messages successfully transferred, a loss probability of 0.2, a corrupt probability of 0.3, and an average interval of 1000. Explain concisely why it is correct.

Presentation matters: your report must be presented in a clean and concise manner that is easy-to-understand for others.

To hand in your PA2, create a zip archive with all your deliverables and submit it on CourseLink. The filename must be cis3210_ass2_XXX.zip, where XXX is your University of Guelph's email ID (Central Login ID). This naming convention facilitates the tasks of marking for the instructor and course TAs. It also helps you in organizing your course work. Failure to follow the requirements will result in mark reduction. Do not include any binary files in your submission.

You can submit your PA2 multiple times and only the latest version will be kept and marked, so it is a good practice to submit your first version well before the deadline and then submit a newer version to overwrite when you make some more progress. Again, make sure your code runs as expected on linux.socs.uoguelph.ca.

Marking

Below is the tentative overall marking scheme of this assignment:

- Stop-And-Wait code including coding style and documentation: 80%
- Report: 20%

Coding style matters. Your code must be written in a proper style and must be well commented so that anyone else can read your code and easily understand how everything works in your code.

Q&A

This assignment is adopted from our textbook by Jim Kurose and who have taught this lab in their introductory networking course for many years. Their students have posed various questions related to this assignment.

If you are interested in looking at the questions they've received (and answers), check out http://gaia.cs.umass.edu/kurose/transport/programming_assignment_QA.htm