Programming Assignment 2: (PG only) (80% of Prog. Assignment marks) Reliable Transport with Selective Repeat Programming Assignment

Due 30 Apr by 17:00 **Points** 100 **Available** until 30 Jun at 23:59

This assignment was locked 30 Jun at 23:59.

Adapted from Kurose & Ross - Computer Networking: a top-down approach featuring the Internet

CBOK categories: Abstraction, Design, Data & Information, Networking, Programming

Foreword

This practical requires thought and planning. You need to start early to allow yourself time to think of what and how to test before modifying any code. Failing to do this is likely to make the practical take far more time than it should. Starting the practical in the final week is likely to be an unsuccessful strategy with this practical.

Very Important

Although this practical is marked automatically, all marks will be moderated by a marker using the following information.

You must add a comment explaining the changes you've made to ***each*** of your svn repository commits. In other words, each commit should explain what was changed in the code and why briefly (e.g. it may be a bug fix or a protocol behaviour fix).

We should be able to review your development process and see the changes made to code with each version you have committed. **Please keep in mind that we can check the number of lines of code as well as actual code changes made with each commit.**

During manual marking and plagiarisms checks, we will look look at your development process. If we do not see a clear path to a solution (i.e. code changes and regular commits and comments reflecting your learning to develop the protocol you **will forfeit 50% of the marks allocated for this prac and in the extreme case you will be allocated a mark of 0%. An example case of a 0% would be the sudden appearance of working code passing multiple tests**.

It is up to you to provide evidence of your development process.

Please make use of the emulator to help you learn!

Task

Implement reliable transport protocol using Selective Repeat in C by studying and extending an implmentation of Go Back N running on a network simulator.

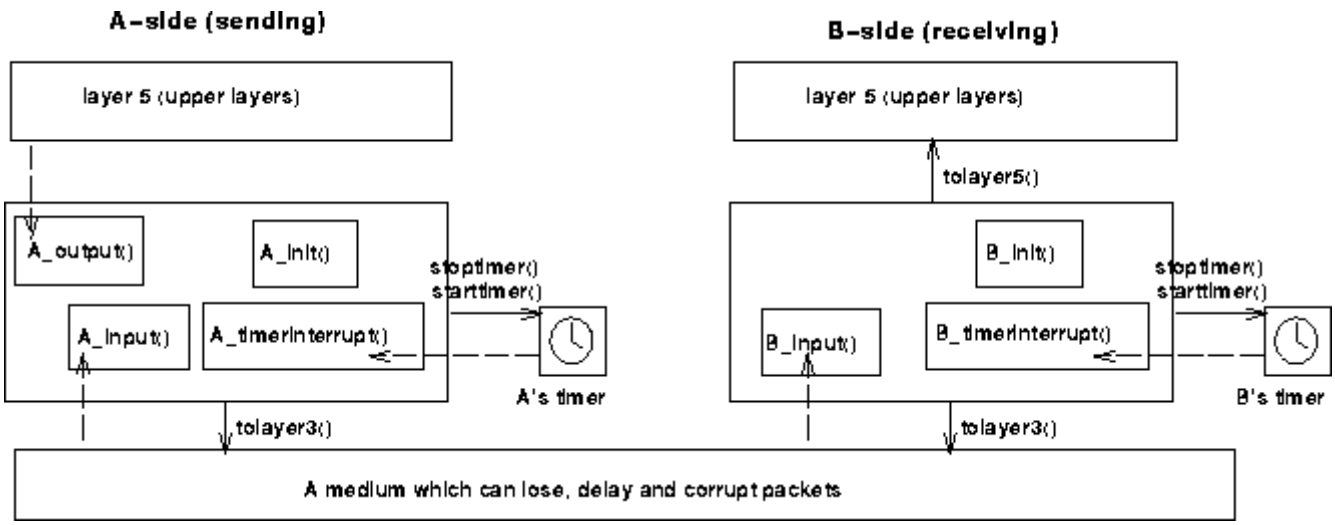
Background

You've been hired by "Net Source", a company specialising in networking software to build Selective Repeat code for their network emulation tool. They have an existing implementation of Go-Back-N that interfaces with their emulator. Their programmer, Mue, who was working on an implementation of the Selective Repeat protocol has caught the flu and the implementation must absolutely be finished in the next week. As the code isn't finished, Mue hasn't completed testing yet. Mue is an experienced C programmer and there is nothing wrong with the C syntax or structure of the code that Mue has written. So you don't have to correct any C syntax errors, your job is to finish the code, correct any ***protocol*** errors and test it.

Although you are not required to use this code, they expect much of this code is reusable and only some modifications are needed to change Go-Back-N to Selective Repeat. Your boss realises you're not a C expert, but at least you've programmed in Java or C++ before so you're the closest they have as an expert. Their last C programmer has written a **C hints for Java and C++ programmers** sheet which explains what you need to know about C that is different than Java and C++. She's confident that if you stick to the sheet, anything else you need to add or change would be the same if you wrote it in Java or C++.

System Overview

To help isolate and demonstrate the behaviour of the sender and receiver, you have a simulation system at your disposal. The overall structure of the environment is shown below:



There are two hosts (A and B). An application on host A is sending messages to an application on host B. The application messages (layer 5) on host A are sent to layer 4 (transport) on host A, which must implement the reliable transport protocol for reliable delivery. Layer 4 creates packets and sends them to the network (layer 3). The network transfers (unreliably) these packets to host B where they are handed to the transport layer (receiver) and if not corrupt or out of order, the message is extracted and delivered to the receiving application (layer 5) on host B.

The file `gbn.c` has the code for the Go Back N sender procedures `A_output()`, `A_init()`, `A_input()`, and `A_timerinterrupt()`. `gbn.c` also has the code for the Go Back N receiver procedures `B_input()` and `B_init()`. At this stage, only unidirectional transfer of data (from A to B) is required, so B does not need to implement `B_timerinterrupt()`. Of course, B will have to send packets to A to acknowledge receipt of data.

Go Back N Software Interface

The routines 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. In the simulator the simulator will call and be called by procedures that emulate the network environment and operating system.

```
1 | void A_output(struct msg message)
```

`message` is a structure containing data to be sent to B. This routine will be called whenever the upper layer application at the sending side (A) has a message to send. It is the job of the reliable transport protocol to insure that the data in such a message is delivered in-order, and correctly, to the receiving side upper layer.

```
1 | void A_input(struct pkt packet)
```

This routine will be called whenever a packet sent from B (i.e., as a result of a *tolayer3()* being called by a B procedure) arrives at A. *packet* is the (possibly corrupted) packet sent from B.

```
1 | void A_timerinterrupt(void)
```

This routine will be called when A's timer expires (thus generating a timer interrupt). This routine controls the retransmission of packets. See `starttimer()` and stoptimer`()` below for how the timer is started and stopped.

```
1 | void A_init(void)
```

This routine will be called once, before any other A-side routines are called. It is used to do any required initialization.

```
1 | void B_input(struct pkt packet)
```

This routine will be called whenever a packet sent from A (i.e., as a result of a *tolayer3()* being called by a A-side procedure) arrives at B. The *packet* is the (possibly corrupted) packet sent from A.

```
1 | void B_init(void)
```

This routine will be called once, before any other B-side routines are called. It is used to do any required initialization.

The unit of data passed between the application layer and the transport layer protocol is a message, which is declared as:

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

That is, `data` is stored in a `msg` structure which contains an array of 20 chars. A char is one byte. The sending entity will thus receive data in 20-byte chunks from the sending application, and the receiving entity should deliver 20-byte chunks to the receiving application.

The unit of data passed between the transport layer and the network layer is a `packet`, which is declared as:

```
struct pkt {
   int seqnum;
   int acknum;
   int checksum;
   char payload[20];
};
```

The `A_output()` routine fills in the payload field from the message data passed down from the Application layer. The other packet fields must be filled in the a_output routine so that they can be used by the reliable transport protocol to insure reliable delivery, as we've seen in class.

These functions implement what the sender and receiver should do when packets arrive.

The Emulator Software Interface

This section will describe the functions of the emulator code that you will be using in your implementation. **Do not edit the emulator code.**

The procedures described above implement the reliable transport layer protocol and are the procedures that you will be implementing.

The following emulator procedures can be called by the reliable transport procedures you write. They are explained here so you know how they fit in. They are *not* part of the reliable transport implementation and these routines work correctly.

```
void starttimer(int calling_entity, double);
```

Where `calling_entity` is either A (for starting the A-side timer) or B (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. You are free to experiment with different timeout values; but *when handing in, the timeout value must be set to 15.0*

Note that `starttimer()` is not the same as `restarttimer()`. If a timer is already running it must be stopped before it is started. Calling `starttimer()` when the timer is already running, or calling `stoptimer()` when the timer is not running, indicates an error in the protocol behaviour and will result in an error message.

The emulator code is in the file `emulator.c`.Use the emulator header file to refere to the definitions of the emulator functions. You do not need to understand or look at `emulator.c`; but if you want to know how the emulator works, you are welcome to look at the code.

The `starttimer()` call should occur immediately after the `tolayer3()` call that sends the packet being timed.

```
1 | void stoptimer(int calling_entity)
```

Where `calling_entity` is either A (for stopping the A-side timer) or B (for stopping the B side timer).

```
1 | void tolayer3(int calling_entity, struct packet)
```

Where `calling_entity` is either A (for the A-side send) or B (for the B side send). Calling this routine will cause a the packet to be sent into the network, destined for the other entity.

```
1 | void tolayer5(int calling_entity, char[20] message)
```

Where `calling_entity` is either A (for A-side delivery to layer 5) or B (for B-side delivery to layer 5). With unidirectional data transfer, you would only be calling this when `calling_entity` is equal to B (delivery to the B-side). Calling this routine will cause data to be passed up to layer 5.

Download the following 4 files and examine the code in `gbn.c` with the description above and your knowledge of Go Back N to make sure you understand how the program fits together:

- **emulator.c (https://myuni.adelaide.edu.au/courses/64831/files/8364872/download?download_frd=1)**
- **emulator.h (https://myuni.adelaide.edu.au/courses/64831/files/8364871/download?download_frd=1)**
- **gbn.c (https://myuni.adelaide.edu.au/courses/64831/files/8364735/download?download_frd=1)**
- **gbn.h (https://myuni.adelaide.edu.au/courses/64831/files/8364736/download?download_frd=1)**

To build the program use the command:

```
1 | $ gcc -Wall -ansi -pedantic -o gbn emulator.c gbn.c
```

Running the simulated network

The emulator is capable of corrupting and losing packets. It will not reorder packets. When you compile and run the resulting program, you will be asked to specify values regarding the simulated network environment

For example

```
$ ./gbn
-----  Stop and Wait Network Simulator Version 1.1 --------

Enter the number of messages to simulate: 10
Enter packet loss probability [enter 0.0 for no loss]:0
Enter packet corruption probability [0.0 for no corruption]:0
Enter average time between messages from sender's layer5 [ > 0.0]:10
Enter TRACE:2
```

Number of messages to simulate

The emulator will stop generating messages as soon as this number of messages have been passed down from layer 5.

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 one in five packets (on average) are corrupted. Note that the contents of payload, sequence or ack field can be corrupted.

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 generated.

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 messages that detail what is happening inside the emulation code as well. 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!

Take a moment to experiment with the simulated environment and look at the events that occur. Try sending a few packets with no loss or corruption. Does the protocol work properly? Try with just loss. Try with just corruption.

Selective Repeat

Once you have inspected Go Back N and understand how the implementation works, you need to study carefully how Selective Repeat (SR) differs from Go Back N. Consider how GBN and SR respond to ACKs, timeouts, new messages, etc on both the sender and the receiver side. In what cases is their behavior the same and in what cases is it different.

When you have considered the similarities and differences, you should be able to identify what changes you will need to make to the existing GBN implementation in order to implement SR. Copy the gbn.c file to a file called sr.c and copy gbn.h to sr.h. Design your changes and then implement those changes in your sr.c file.

The implementation of Selective Repeat *should* be identical to that described in the textbook (Section: Principles of Reliable Data Transfer)

When you have completed your selective repeat implementation you can compile it with:

```
$ gcc -Wall -ansi -pedantic -o sr emulator.c sr.c
```

Some Tips

Compiler flags

When you recompile the program. Using the `-ansi -Wall -pedantic` flags will give you useful warnings if you are doing something that is likely to be wrong (like using = rather than ==, or misusing a data structure). If you get any warnings, fix them before submitting. Warnings indicate that there is something wrong, even if the program compiles. If you can't work out the warning, ask on the discussion board.

Code - Compile - Test - Repeat

Make one change at a time and devise a test case to check that your addition actually works. For example, if you wanted to check that B was responding correctly to an out of order packet, send a few (say 2) packets with a loss probability of .2 and see how B responds. If none of the packets are lost, then increase the loss probability until you get a case where one of the packets is lost and a packet arrives out of order so you can see B's response.

Be sure to commit your changes to svn or they won't be tested

Once you're confident you have the selective repeat protocol working, or if you are really stuck on what is wrong, submit to websubmission system. A test script will run a series of tests specifically designed to identify whether the protocol is behaving as expected. If your corrected program does not pass one of the tests, The test script will stop at that point and show you the expected output of the test and your output. By comparing the two you should be able to work out at least one thing that the protocol is still doing incorrectly.

The testing script we run is not a program with high-level reasoning abilities, it's possible that you might do something unexpected that will trick it. Inspecting the output should allow you to identify whether the problem is with your implementation or just an unexpected output.

What you "C" is what you get

This programming assignment is in the C language. Most networking system code and many networked applications are written in C, so we consider it important that you have at least seen C and can do some basic reading and debugging in the language. The code is given and the syntax of C and Java are very similar/same as are the basic structures (if/then, loops, etc), so we do expect that third year students can accomplish this. The main C concepts that you may not have come across in Java will be accessing structures and handling pointers. We have provided a **C hints (https://cs.adelaide.edu.au/users/third/cn/Practicals/Chints.html)** sheet which explains what you need to know that is different than Java. Unless you are familiar with C, please stick to our hints. It's easy to get memory manipulation wrong in C and it can be quite difficult to debug. The code has been carefully structured so that you do not need to make use of pointers. If you are experienced with C, you can use pointers, if you are not experienced with C, then stick to static arrays as the GBN implementation does. The hint sheet gives you the easy/clear way of doing the things. Remember that the given code is correct C, so you can use the given code as examples for how to manage the arrays and structures. If you do have any questions still after looking at the examples, please don't hesitate to post questions (even code fragments) on the discussion board.

Handing In

You will need to use SVN to store your practical files (refer to the **SVN notes** for setting up your repository for this practical). The handin key for this practical is

`YYYY/s1/cna/sr` (Note: replace YYYY with the year)

You can submit and automatically mark your assignment by following **this link (https://cs.adelaide.edu.au/services/websubmission/)** to the web-submission system.

The Websubmission script will expect a `sr.c` file and `sr.h` to be found in your repository.

Marking & Penalties

Each submission to the marking link after the third submission will result in a 1 mark reduction

from your final submission score (the *final submission* here implies the most recent submission before the deadline).

So you can submit **twice** to check the correct behaviour and re-submit your solution to get **full marks**.

After you have seen the results twice and the working solution behaviour, you have the option to improve your solution. After the third submission, since we have now provided the details of a working solution, we will deduct one mark for subsequent submissions.

So this means that if you submit 5 times, and you got 100 for the final submission, your marks will be (100-2)/100 = 98% => 8% (rounded up) of your grade.

So you can still get 8% of your grade as long as you don't go below 94/100.

If your submitted code hangs, crashes or does not compile you will get an automatic zero. Marks on the rubrics for each tests is allocated for a full and correct implementation only and there are no partial marks for a given test.

The marks assigned by mark submission link are provisional. You are responsible for writing a working implementation and for testing your implementation. Any errors discovered during review will reduce your mark for that functionality. We're not hiding anything, but it is possible that you might create an implementation that has an error not exercised by mark submission link.

PLEASE NOTE: the changes you make to `sr.c` must not add, remove or change any comments printed when the trace level is set to 2 or 3. You can, and should, add comments about what is happening in your code at other trace levels to assist in debugging.

Practical 2: Selective Repeat (If your submitted code hangs, crashes or does not compile you will get an automatic zero. Marks on the rubrics for each tests is allocated for a full and correct implementation only and there are no partial marks)

Criteria	Ratings		Pts
basic functionality	**5 Pts** **Full Marks**	**0 Pts** **No Marks**	5 pts
basic functionality with full window	**5 Pts** **Full Marks**	**0 Pts** **No Marks**	5 pts
lost ACK/timeouts	**10 Pts** **Full Marks**	**0 Pts** **No Marks**	10 pts
handles corrupted packets	**10 Pts** **Full Marks**	**0 Pts** **No Marks**	10 pts
handles corrupted ACK	**10 Pts** **Full Marks**	**0 Pts** **No Marks**	10 pts
delivering previously buffered packets	**10 Pts** **Full Marks**	**0 Pts** **No Marks**	10 pts
handles lost packet, duplicate packet, duplicate ACK	**30 Pts** **Full Marks**	**0 Pts** **No Marks**	30 pts
handles mixed events, window management and high message volumes	**20 Pts** **Full Marks**	**0 Pts** **No Marks**	20 pts
			Total points: 100