

# Practical 2: (Program) (80% of prac 2 marks) - Reliable Transport (PG students)

---

**Due** May 1 by 17:00      **Points** 100

---

## Computer Networks & Applications

### Programming Assignment 2: Building a Reliable Transport Protocol<sup>1</sup>

<https://cs.adelaide.edu.au/users/third/cn/Practicals/Debug-prac-2015/#footnotes>

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

Note - 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 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 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. *I expect this will happen after each submission to the oracle.*

*Please make use of the emulator to help you learn!*

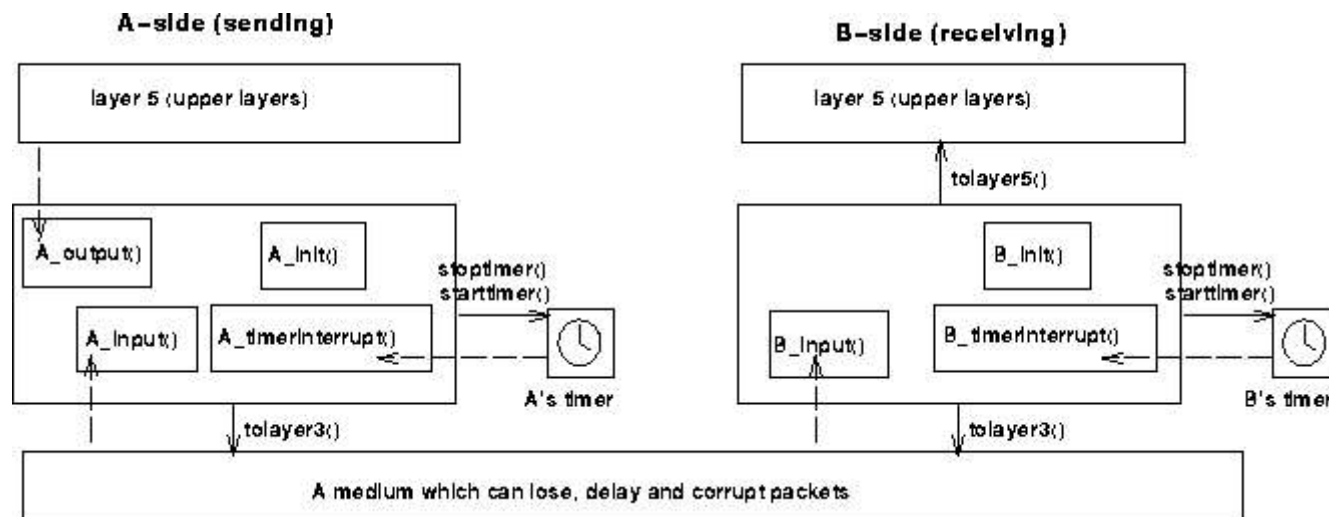
## Part 1 - Alternating Bit Protocol

You've been hired by "Net Source", a company specialising in networking software. Their programmer, Mue, who was working on an implementation of the Alternating Bit protocol has caught the flu and the implementation must absolutely be finished in the next week. Most of the implementation is complete and Mue left comments for parts still to be completed. 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. Your boss realises you're not a C expert, but at least you've programmed in some language with C like syntax (e.g. Java, C++) before so you're the closest they have as an expert. Mue has written a [C hints for programmers of C like languages programmers](#) sheet which explains what you need to know about C that is different than Java. 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.

## The testing system

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 implements Alternating Bit 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 (Alternating Bit receiver) and if not corrupt or out of order, the message is extracted and delivered to the receiving application (layer 5) on host B.

Mue has supplied the code for the Alternating Bit sender procedures `A_output()`, `A_init()`, `A_input()`, and `A_timerinterrupt()`. She has also written the code for the Alternating Bit 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` or `B_output`. Of course, B will have to send ACK packets to A to acknowledge receipt of data.

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.

- **A\_output(message)**, where *message* is a structure of type *struct msg*, 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 Alternating Bit 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 *struct pkt* . 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.
- **A\_timerinterrupt()** 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.
- **A\_init()** This routine will be called once, before any other A-side routines are called. It is used to do any required initialization.
- **B\_input(packet)**, where *packet* is a structure of type *struct pkt* . 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.
- **B\_init()** 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 Alternating Bit (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 Alternating Bit (transport layer) and the network layer is the *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 are used by the Alternating Bit protocol to insure reliable delivery, as we've seen in class.

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

## Software Interfaces





The procedures described above implement the Alternating Bit transport layer protocol. The following emulator procedures are called by the Alternating Bit procedures. They are explained here so you know how they fit in. They are *not* part of the Alternating Bit implementation and these routines work correctly. Do not modify them:

- **starttimer(calling\_entity,increment)**, 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 for mark submission linking, the timeout value must be set to 15.0*

Note that `starttimer()` is not `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 starttimer() call should occur immediately after the tolayer3() call to send the packet being timed.***

- **stoptimer(calling\_entity)**, where calling\_entity is either A (for stopping the A-side timer) or B (for stopping the B side timer).
- **tolayer3(calling\_entity,packet)**, where calling\_entity is either A (for the A-side send) or B (for the B side send), and packet is a structure of type struct pkt. Calling this routine will cause the packet to be sent into the network, destined for the other entity.
- **tolayer5(calling\_entity,message)**, where calling\_entity is either A (for A-side delivery to layer 5) or B (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 B (delivery to the B-side). Calling this routine will cause data to be passed up to layer 5.

The emulator code is in the file [emulator.c](#) . The *incorrect* implementation of Alternating Bit is in the file [altbit.c](#) . There are also two header files [emulator.h](#)  and [altbit.h](#) , which define the procedures and shared variables used in the program.

You should download the 4 files and examine the code in `altbit.c` with the description above and your knowledge of Alternating Bit and make sure you understand how the program fits together. You do not need to understand `emulator.c`; but if you want to know how the emulator works, you are welcome to look at the code.

To build the program use the command:

```
gcc -ansi -Wall -pedantic emulator.c altbit.c
```

The executable program will be called `a.out`. To run the program, type:

```
./a.out
```

## The simulated network environment

The medium 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:

- **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.
- **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!
- **Average time between messages from sender's layer 5.** 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.

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. Some of the errors with the Alternating Bit implementation should be apparent.

## What to do

The implementation of Alternating Bit *should* be identical to that described in your book (page 244/245, Extended FSM description of Alternating Bit sender and receiver are shown).

There may be errors in the protocol. You will need to test it. The errors are only in the `altbit.c` file. The other three files are correct and do not need to be modified. None of the mistakes are with the programming language or the data structures, they are all errors in the protocol **behaviour**. The program will compile and run as written; but the sender and receiver do not follow the Alternating Bit protocol.

This part has 4 tests which test that you have correctly finished the code. Since mark submission link will show you what should have happened if you submit an incorrect solution, **there is a 1 mark penalty for each submission you make after the first**. You should test and correct any errors before submission. If you rely on mark submission link to do your testing, then your mark will reflect this. However, if you are really stuck, realise that this is a small percentage of the marks for each test, and the output should help clarify any misconception. We have provided an Oracle (see details below in the Handing In section) which will tell you if you pass the tests or not; but will give you no other information. **There is no mark penalty to use the Oracle.**

I suggest the following strategy for this practical. In each step I'd recommend focussing on getting the receiver behaving correctly first and then turn your attention to the sender (the receiver behaviour is simpler and therefore easier to get right and test). :

1. Compile the code and do a simple test sending just one packet with no corruption or loss. Does both the sender and receiver behave properly in this scenario? If not fix any problems. At this stage you should be able to pass test 1 (you can check with Oracle). If not, look at what your sender and receiver do in your test. (10 marks)
2. Fill in the code items numbered 1. Now run some small tests with loss. Does your sender follow the protocol properly when a packet is lost? If not, fix the behaviour. You should now be able to pass test 2. (10 marks)
3. Fill in the code items number 2 (check if ACK is a duplicate ACK) and 3 (B receives wrong sequence number or corrupt packet). When you have this correct you should be able to pass test 3. (20 marks)
4. Fill in the code item numbered 4. Now run a small test with corruption. Do your sender and receiver detect corruption when it occurs and do they react correctly when a corrupted packet or ACK is received? You should now be able to pass test 4. (10 marks)

#### Some Tips

- 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.
- For every behaviour you correct, devise a test case to check that your changes actually work. For example, if you wanted to check that B was responding correctly to a lost 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 alternating bit protocol working, or if you are really stuck on what is wrong, submit to marking submission link. Mark Submission Link will run a series of tests specifically designed to identify whether errors have been dealt with. If your corrected program does not pass one of the tests, mark submission link 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.


Mark Submission Link is not a program with high level reasoning abilities, it's possible that you might do something unexpected that will trick it. If you think this is the case, then please post to the discussion board or e-mail me and explain why you think mark submission link is incorrect and I'll check. Be sure to include your reasoning, so that if mark submission link is correct, I can help you correct any misconceptions. If you are able to provide a valid reason why the mark submission link test incorrectly marked you as wrong, I will remove the penalty for that one submission. Note that not following the instructions in this prac are not grounds for removing a penalty (e.g. not committing to svn before submitting, relying on changes to files other than altbit.c, etc.).

Note 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, 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](#) 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 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](https://cs.adelaide.edu.au/users/third/cn/Practicals/SVN-instructions.php) [\\_ \(https://cs.adelaide.edu.au/users/third/cn/Practicals/SVN-instructions.php\)](https://cs.adelaide.edu.au/users/third/cn/Practicals/SVN-instructions.php) for setting up your repository for this practical). The handin key for this practical is

```
2018/s1/cna/AlternatingBit
```

You can submit and automatically mark your assignment by following [this link](https://cs.adelaide.edu.au/services/websubmission/)  [\(https://cs.adelaide.edu.au/services/websubmission/\)](https://cs.adelaide.edu.au/services/websubmission/) to the web-submission system. **Remember that each submission to the marking link after the first will result in a 1 mark penalty.**

**You can submit to the Alternating Bit Oracle with no penalty.** This is also done through the web-submission system.


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 `altbit.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.***

You should aim to have this part completed no later than 29 March. This date is a guide to help you manage your time

## Part 2 - adding pipelining

Network Source was so impressed with your ability to work with existing code and to get the protocol out on time, that they've asked you to take on their next task which is to add windows to allow pipelining. They want a "go back N" protocol with a window size of 6. Having studied networks, you know that go back N is just an extension of the alternating bit protocol, so you'll be able to reuse all the code and just adapt it to support a window of size 6.

You may find it useful to review the go back n protocol in the textbook and look at the finite state machines on page 251. You may also discuss the questions below on the discussion board. You may also find it helpful to experiment with the Go Back N protocol demo at [https://media.pearsoncmg.com/aw/ecs\\_kurose\\_compnetwork\\_7/cw/content/interactiveanimations/go-back-n-protocol/index.html](https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/go-back-n-protocol/index.html)  [\(https://media.pearsoncmg.com/aw/ecs\\_kurose\\_compnetwork\\_7/cw/content/interactiveanimations/go-back-n-protocol/index.html\)](https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/go-back-n-protocol/index.html)

- You should not attempt to change the code until you understand how the protocol works. Once you understand how and why the protocol behaves as it does, you can start into the code.

Start by copying altbit.c to gbn.c Change WINDOWSIZE to 6. Then work on changes to A\_output. What happens when a new packet arrives and there is already a packet waiting for an ACK? How many sequence numbers do you need? You only have one timer


available. To simplify managing with one timer, you can just time the **oldest** packet. When should the timer be started? What if there is already a packet in the window, should the timer be restarted? What should the timer do in A\_input when an ACK arrives (consider you might still have packets in the window)? What changes need to be made to the receiver? When you have A\_output, A\_input and B\_input correct, you should be able to pass test 1. (20 marks)

Now make changes to A\_input to handle cumulative acknowledgements. What changes need to be made to receiving acknowledgements to handle cumulative acknowledgements in go back n? Once you have A\_input handling cumulative acknowledgement you should be able to pass test 2. (20 marks)

Finally how does timerinterrupt need to be changed now that we have a window with multiple packets? Once you have corrected timerinterrupt, you should be able to pass test 3. (10 marks)

The handin key for this part is

2018/s1/cna/GBN

You can submit and automatically mark your assignment by following [this link](https://cs.adelaide.edu.au/services/websubmission/)  (<https://cs.adelaide.edu.au/services/websubmission/>) to the web-submission system. **Remember that each submission after the first will result in a 1 mark penalty.**

You can submit to the Go Back N Oracle with no penalty. This is also done through the web-submission system.

- 1 This assignment is adapted from Kurose and Ross Assignment. It is *not* the same though and has significantly modified support code. It is not compatible with the original Kurose and Ross assignment (nor the solutions), so make sure you are following the specifications and using the provided code from this practical sheet and not the textbook or website.


## Understanding the output from the testing

We use a utility called `diff` to compare the generated output from your protocol with ours. What you will see after a submission to the Oracle is a diff between the expected output and your output. The comparison results we provide is limited to the last seven lines in the generated output from testing your protocol.

Example of a differences found:

 followed by text indicates lines from the expected output

> followed by a text indicates lines from the output generated with your implementation

2,3c2,3 you may also see number like this at the top, you can find out more about these by reading about the diff utility (for example: <https://linuxhandbook.com/diff-command/>  [\(https://linuxhandbook.com/diff-command/\)](https://linuxhandbook.com/diff-command/)).