CSE4/589: PA2 Description

# CSE 489/589
# Programming Assignment 2
# Reliable Transport Protocols

## 1. Objectives

*In a given simulator, implement three reliable data transport protocols: Alternating-Bit (ABT), Go-Back-N (GBN), and Selective-Repeat (SR).*

## 2. Getting Started

### 2.1 Alternating-Bit Protocol (rdt3.0)
Text book: Page 214 – Page 217

### 2.2 Go-Back-N Protocol
Text book: Page 221 – Page 226

### 2.3 Selective-Repeat Protocol
Text book: Page 226 – Page 232

### 2.4 Install the PA2 template
Read the document at https://goo.gl/G4cPfH in full and install the template.
***It is mandatory to use this template.***

## 3. Implementation

### 3.1 Programming environment
You will write C (or C++) code that compiles under the GCC (GNU Compiler Collection) environment. Furthermore, you should ensure that your code compiles and operates correctly on the **ONE** CSE server assigned to you (see section 3.2).
You should NOT use any CSE server other than the one assigned to you.

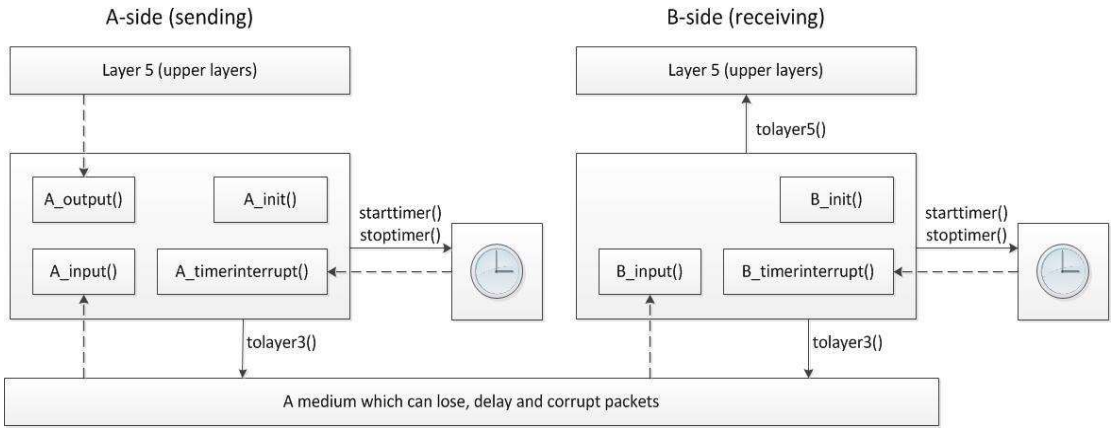
Your code should successfully compile using the version of gcc (for C code) or g++ (for C++ code) found on the CSE servers and should function correctly when executed.

### 3.2 CSE Servers
**You will receive your server assignment for PA2 in an e-mail from the course staff.** If you do not receive this e-mail within the first week of PA2 release, contact us.
For the purpose of this assignment, you should only use (for development and/or testing) the directory created for you on the assigned server. Change the access permission to this directory so that only you are allowed to access the contents of the directory. This is to prevent others from getting access to your code.

### 3.3 Overview
In this programming assignment, you will be writing the sending and receiving transport-layer code for implementing a simple reliable data transfer protocol. There are 3 versions of this assignment, the Alternating-Bit Protocol version, the Go-Back-N version, and the Selective-Repeat version.

Since 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. However, the programming interface provided to your routines, i.e., the code 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 is also simulated, and timer interrupts will cause your timer handling routine to be activated.

### 3.4 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 which simulate a network environment. The overall structure of the environment is shown below:




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 structures and simulator routines, as well as stub routines (i.e., those you are to complete) are inside the template files, described later. Your sending entity will thus receive data in 20-byte chunks from Layer 5; your receiving entity should deliver 20-byte chunks of correctly received data to Layer 5 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 Layer 5. 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 (as a result of a tolayer3() (see section 3.5) being called 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 (as a result of a tolayer3() (see section 3.5) being called by a 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.

These six routines are where you can implement your protocols.
You are not allowed to modify any other routines.

**3.5 Software Interfaces**
The procedures described above are the ones that you will write. We have written 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 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).

- *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 (calling_entity, data)*
  where *calling_entity* is either 0 (for A-side delivery to layer 5) or 1 (for B-side delivery to layer 5), and *data* is a char array of size 20. 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.

- *getwinsize()*
  returns the window size value passed as parameter to **-w** (see section 3.6).

- *get_sim_time()*
  returns the current simulation time.

**3.6 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 transport protocol layer.

The medium is capable of corrupting and losing packets. However, 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 certain values regarding the simulated network environment as command-line arguments. We describe them below:

- *Seed* **(-s)**
  The simulator uses some random numbers to reproduce random behavior that a real network usually exhibits. The seed value (a non-zero positive integer) initializes the random number generator. Different seed values will make the simulator behave slightly differently and result in different output values.

- *Window size* **(-w)**
  This only applies to Go-back-N and Selective-Repeat binaries. Both these protocols use a finite-sized window to function. You need to tell the simulator before hand, what window size you want to use. Infact, your code will internally use this value for implementing the protocols.

- *Number of messages to simulate* **(-m)**
  The simulator (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 simulator stops. **This value should always be greater than 1.** If you set this value to 1, your program will terminate immediately, before the message is delivered to the other side.

- *Loss* **(-l)**

Specify a packet loss probability [0.0,1.0]. A value of 0.1, for example, would mean that one in ten packets (on average) are lost.

• *Corruption* **(-c)**
You are asked to specify a packet corruption probability [0.0,1.0]. A value of 0.2, for example, 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 time between messages from sender's layer5* **(-t)**
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.

• *Tracing* **(-v)**
Setting a tracing value of 1 or 2 will print out useful information about what is going on inside the simulation (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 simulator-debugging purposes. A tracing value of 2 may be helpful to you in debugging your code. You should keep in mind that, in reality, you would not have underlying networks that provide such nice information about what is going to happen to your packets!

## 4. Protocols

### 4.1 Alternating-Bit-Protocol (ABT)
You are to write the procedures which together will implement a stop-and-wait (i.e., the alternating bit protocol, which is referred to as rdt3.0) unidirectional transfer of data from the A-side to the B-side. **Your protocol should use only ACK messages.**
You should 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 should buffer the data being passed to the A_output() routine.

### 4.2 Go-Back-N (GBN)
You are to write the procedures which together will implement a Go-Back-N unidirectional transfer of data from the A-side to the B-side, with a certain window size.
It is recommended that you first implement the easier protocol (the Alternating-Bit version) and then extend your code to implement the more difficult protocol (the Go-Back-N version). Some new considerations for your GBN code (which do not apply to ABT) are:

• **A_output()**
will now sometimes be called when there are outstanding, unacknowledged messages in the medium, implying that you will have again to buffer multiple messages in your sender. Also, you'll need buffering in your sender because of the nature of Go-Back-N: 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 1000 messages) and have your sender simply abort (give up and exit) should all 1000 buffers be in use at one point (Note: If the buffer size is not enough in your experiments, set it to a larger value). In the “real-world”, of course, one would have to come up with a more elegant solution to the finite buffer problem!

### 4.3 Selective-Repeat (SR)
You are to write the procedures which together will implement a Selective-Repeat unidirectional transfer of data from the A-side to the B-side, with a certain window size.
It is recommended that you implement the GBN protocol before you extend your code to implement SR. Some new considerations for your SR code are:

• **B_input(packet)**
will have to buffer multiple messages in your receiver because of the nature of Selective-Repeat. The receiver should reply with ACKs to all packets falling inside the receiving window.

• **A_timerinterrupt()**
will be called when A's timer expires (thus generating a timer interrupt).
Even though the protocol uses multiple logical timers, **remember that you've only got one hardware timer**, and may have many outstanding, unacknowledged packets in the medium. **You will have to think about how to use this single timer to implement multiple logical timers. Note that an implementation that simply sets a timer every T time units and retransmits all the packets that should have expired within those time units is NOT acceptable. Your implementation has to ensure that each packet is retransmitted at the exact time at which it would be retransmitted if you had multiple timers.**

## 5. Testing

We will test your implementation of the three protocols with the settings/parameters described below. We will begin with the most basic tests and then move on to more involved tests (in the order mentioned below).

### 5.1 SANITY Tests

Here we perform two types of checks.
(i) Check for duplicate and/or out-of-order packets at B.
For each of the three protocols:

Environment Settings
- ➔ Number of messages to simulate (-m): 1000
- ➔ Average time between messages from sender's layer5 (-t): 50
- ➔ Window size (-w): 10

Test Cases
- ➔ Loss (-l): 0.1, 0.2, 0.4, 0.6, 0.8, Corruption (-c): 0.0
- ➔ Loss (-l): 0.0, Corruption (-c): 0.1, 0.2, 0.4, 0.6, 0.8

(ii) Confirm that the behavior of your protocols is the expected one under some very simple tests (e.g. no packets is delivered under 100% loss).
For each of the three protocols:

Environment Settings
- ➔ Number of messages to simulate (-m): 20
- ➔ Average time between messages from sender's layer5 (-t): 1000 (ABT); 50 (GBN,SR)
- ➔ Window size (-w): 50

Test Cases
- ➔ Loss (-l): 0.0, Corruption (-c): 0.0
- ➔ Loss (-l): 1.0, Corruption (-c): 0.0
- ➔ Loss (-l): 0.0, Corruption (-c): 1.0

### 5.2 BASIC Tests

For each of the three protocols:

Environment Settings
- ➔ Number of messages to simulate (-m): 20
- ➔ Average time between messages from sender's layer5 (-t): 1000 (ABT); 50 (GBN,SR)
- ➔ Window size (-w): 50

Test Cases
- ➔ Loss (-l): {0.1, 0.4, 0.8}, Corruption (-c): 0.0
- ➔ Loss (-l): 0.0, Corruption (-c): {0.1, 0.4, 0.8}

### 5.3 ADVANCED Tests

For each of the three protocols:

Environment Settings
- ➔ Number of messages to simulate (-m): 1000
- ➔ Average time between messages from sender's layer5 (-t): 50
- ➔ Window size (-w): 10

Test Cases
- ➔ Loss (-l): {0.1, 0.2, 0.4, 0.6, 0.8}, Corruption (-c): 0.0
- ➔ Loss (-l): 0.0, Corruption (-c): {0.1, 0.2, 0.4, 0.6, 0.8}

## 6. Analysis and Report

For the analysis part, we provide you with a set of experiments (see section 6.1) to compare your implementation of the three different protocols' performance, consisting of various loss probabilities, corruption probabilities, and window sizes.

You need to present your results in the form of a report in a file named as **Analysis_Assignment2.pdf**. The report should have the following **statement right at the top:**

**I have read and understood the course academic integrity policy.**

Your submission will NOT be graded without this statement.

Even if you decide not to attempt the analysis part of the assignment, **you need to have a report file** (with the name given above) with the following three items:

- Academic integrity declaration mentioned above.
- Brief description of the timeout scheme you used in your protocol implementation and why you chose that scheme.
- Brief description (including references to the corresponding variables and data structures in your code) of how you implemented multiple software timers in SR using a single hardware timer.

We expect you to use graphs to show your results for each of the experiments in 6.1 and then write down your observations. Further, your report, at the very least, should answer questions like: What variations did you expect for throughput by changing those parameters and why? Do you agree with your measurements; if not then why?

### 6.1 Performance Comparison

In each of the following 2 experiments, run each of your protocols with a total number of 1000 messages to be sent by entity A, a mean time of 50 between message arrivals (from A's layer5) and a corruption probability of 0.2.

You will need to use the scripts bundled in the template to run the experiments. Refer to the template instructions for more details.
We will not accept results that you obtain by running your own tests.
Read the template instructions for more details.

- **Experiment 1**
With loss probabilities: {0.1, 0.2, 0.4, 0.6, 0.8}, compare the 3 protocols' throughputs at the application layer of receiver B. Use 2 window sizes: {10, 50} for the Go-Back-N version and the Selective-Repeat Version.

Expected Graphs
  - ➔ Window size: 10; X-axis: Loss probability; Y-axis: Throughput (ABT, GBN and SR) in one graph/plot.
  - ➔ Window size: 50; X-axis: Loss probability; Y-axis: Throughput (ABT, GBN and SR) in one graph/plot.

- **Experiment 2**
With window sizes: {10, 50, 100, 200, 500} for GBN and SR, compare the 3 protocols' throughputs at the application layer of receiver B. Use 3 loss probabilities: {0.2, 0.5, 0.8} for all 3 protocols.

Expected Graphs
  - ➔ Loss probability: 0.2; X-axis: Window size; Y-axis: Throughput (ABT, GBN and SR) in one graph/plot.
  - ➔ Loss probability: 0.5; X-axis: Window size; Y-axis: Throughput (ABT, GBN and SR) in one graph/plot.
  - ➔ Loss probability: 0.8; X-axis: Window size; Y-axis: Throughput (ABT, GBN and SR) in one graph/plot.

Note that the "Expected Graphs" for each experiment constitute the *minimum expectation*. You are encouraged to add any additional graphs that help you explain your observations or your design choices. Also, note that when grading your report, we will consider both content and presentation. Make sure your graphs are well-presented (good choice of curves and colors, legible fonts, proper description of axes, etc.) and your observations and explanations are clearly described.

## 7. Helpful Hints & FAQ (from Kurose-Ross)

- **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).

- 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 cannot share global variables.

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

**• 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**
We'd recommend that you set the tracing level to 2 and put LOTS of printf() statements in your code while you are debugging your procedures.

**• Random Numbers**
The simulator 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 simulator we have supplied you. Our simulation 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 simulator expects. Please take a look at the routine jimsrand() in the simulator code. Sorry.

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.

**• Q&A from Kurose-Ross**

*1. My timer doesn't work. Sometimes it times out immediately after I set it (without waiting), other times, it does not time out at the right time. What's up?*
The timer code is OK (hundreds of students have used it). The most common timer problem I've seen is that students call the timer routine and pass it an integer time value (wrong), instead of a float (as specified).

*2. You say that we can access you time variable for diagnostics, but it seems that accessing it in managing our timer interrupt list would also be useful. Can we use time for this purpose?*
Yes.

*3. How concerned does our code need to be with synchronizing the sequence numbers between A and B sides? Does our B side code assume that Connection Establishment (three-way handshake) has already taken place, establishing the first packet sequence number? In other words can we just assume that the first packet should always have a certain sequence number? Can we pick that number arbitrarily?*
You can assume that the three way handshake has already taken place. You can hard-code an initial sequence number into your sender and receiver.

*4. When I submitted my assignment I could not get a proper output because the program core dumped…. I could not figure out why I was getting a segmentation fault so ….*
Offhand I'm not sure whether this applies to your code, but it seems most of the problems with seg. faults in this assignment stemmed from programs that printed out char *'s without ensuring those pointed to null-terminated strings. (For example, the messages -- packet payloads -- supplied by the network simulator were not null-terminated). This is a classic difficulty that trips up many programmers who've recently moved to C from a safer language.

## 8. Grading and Submission

The grading will be done using a combination of automated tests (described in section 5) and manual evaluation of your analysis report. For a detailed breakup of points associated with each command/functions, see https://goo.gl/s74dAe

For packaging and submission, see the section **Packaging and Submission** at https://goo.gl/G4cPfH.