UNIVERSITY OF DELAWARE
DEPARTMENT OF COMPUTER & INFORMATION SCIENCES

CISC 650 / CPEG 651 / ELEG 651: Computer Networks II

Fall Semester, 2015 Professor: Adarsh Sethi

PROGRAMMING PROJECT 2

This project is an individual assignment and must be done independently by each student. No late submissions will be accepted for this project.

In this project, you will implement a Transport Layer protocol to transmit data with Reliable Data Transfer from a client (Sender) to a server (Receiver) in the presence of channel errors and loss. The protocol to be implemented by you is the Go-Back-N protocol. The protocol will be unidirectional in which data is sent in one direction only (client to server) with acknowledgements being sent in the reverse direction (server to client). Only positive ACKs are used. The transmission of packets will be done over UDP (that represents an unreliable network layer channel) using fixed UDP ports.

To implement this protocol, you will write two separate programs called *client* and *server* which represent the actions to be executed by the sending and receiving nodes respectively. Both the client and the server will run on *orioles*.

Packet Formats

The format of a data packet is shown in the figure below. Each data packet contains a 4-byte long header followed by a number of data characters. The header contains 2 fields, each of length 16 bits (2 bytes) as shown in the figure. You must convert the values in these fields into the network byte order when they are transmitted, and convert them back to host byte order when they are received.

The first field of the header is a count of the number of data characters in the packet. This value must be in the range 0 through 80. If the count is 0, then there are no data characters in the packet.

```
<----------------- 16 bits (2 bytes) ------------------>
--------------------------------------------------------
|              Count (no. of data bytes)               |
--------------------------------------------------------
|                Packet Sequence Number                |
--------------------------------------------------------
|            Data Bytes      |          ...            |
--------------------------------------------------------
```

The second field of the header is the packet sequence number. Each packet transmitted by the client is assigned a sequence number in the range 0 through 15.

The format of an acknowledgement packet that is returned from the server to the client is shown in the figure below:

```
<---------------- 16 bits (2 bytes) ----------------->
------------------------------------------------------
|                   ACK Sequence Number              |
------------------------------------------------------
```

Acknowledgements are transmitted (from the server to the client) as separate packets of fixed length, 2 bytes. The first and only field of the ACK packet is the ACK sequence number, which may be in the range 0 through 15. This is the sequence number of the data packet being acknowledged by this ACK.

Once again, you must perform the conversion between network and host byte orders on both the transmitting and receiving ends.

General Actions

The server starts by printing out its port number. It then prompts the user to enter some configuration parameters (see later section for details). It then waits for data packets to arrive from the client.

The client starts by prompting the user to enter the hostname and port number for the server. It then prompts the user to enter the name of the file to be transferred. Next the client prompts the user to enter some configuration parameters (see later section for details).

The client then reads the input file and sends it to the server in a series of packets as described below. The server receives the file and stores it with the name *out.txt*. When the file transfer is complete, both the client and the server terminate execution.

The client constructs packets by reading lines one at a time from the input file. Each line in the input file contains a sequence of printable characters (no control characters, etc.), with no more than 80 characters on a line. The “newline” character read from the file at the end of each line is also transmitted in the packet and is included within the limit of 80 characters per line. You should note that the “newline” character is not the same as the “null” character used by C as a string terminator. You should not include any “null” characters in the line to be transmitted in a packet.

The client transmits each line to the server in a separate packet. The server receives the packets and puts their data into distinct lines in the output file. You must make sure that duplicate packets are discarded and do not get their data stored in the output file.

When the client is finished transmitting all the lines in the data file, *and has received ACKs for all of them*, it will send a special last packet called EOT signifying “End of Transmission”. This packet will have a Count of 0 and no data characters. It will have a Sequence Number that is the next sequence number that would have been used if this were a valid data packet. It is important that this packet be transmitted only *after* the client has received ACKs for all transmitted data packets. The server will not transmit an ACK for the EOT packet, and the client will not expect any ACK to be returned for it. The client program can terminate once this packet has been transmitted. When the server receives the EOT packet, it closes the output file and also terminates.

Go-Back-N Protocol

Follow the general description of the Go-Back-N protocol studied in class (also in Figures 3.20 and 3.21 of your textbook) to design the actions executed by your programs. In addition, use the loop structure for the actions of the client and server described in Lecture Notes Topic 3, Part 3, page 5 of the handout (Go-Back-N: General Implementation).

There are two significant departures from the above protocol in your implementation:

- How you handle the "Call from above" represented by the *rdt_send(data)* function: If the client's window is not full, instead of waiting for the user above to call your protocol machine, you should call a function that reads the next line from the input file and hands it to you as the data to be transmitted in the next packet.
- Checksum: Your packet format does not have a checksum, and your data packets and ACKs are not going to carry a checksum. Thus you will not be doing any checksum computation, nor checking the received packets to see if there are any errors. Thus, the function *make_pkt* will not have a checksum parameter, and you will omit the calls to the functions *corrupt* and *notcorrupt* in your code.

Client Actions

Because the client must check for timeouts as well as check for incoming ACKs, it must not block on the *read* function call if there is no ACK in the input buffer. For this reason, the reading of an incoming packet must be done in a non-blocking fashion. Thus, if there is no incoming ACK available, the read function should not block, but should instead return without reading so you can continue in the loop to check if a timeout has occurred. Use the nonblocking version of the UDP client for your implementation.

The sequence number of successive packets transmitted by the client must range from 0 through 15 and then wrap-around to 0. This is true no matter what window size is being used. It is acceptable if you use a buffer of size 16 to store the queue of unACK'ed packets.

Timeout:

The client needs to start a timer when a data packet is transmitted. A single timer should be used for this purpose (as in the FSM given in your textbook). The semantics of this timer can be tricky, so make sure you understand exactly when the timer needs to be started, cleared, or restarted.

It is sufficient for your purpose to have a simple synchronous implementation of the timer. In this implementation, you can find the current time with a microseconds resolution using the *gettimeofday()* function. Pay close attention to the structure returned by this function; *both* fields of this structure must be used. To start the timer, you can set a timer variable to the time at which the timer should expire (the current time plus the timeout interval). In each loop iteration, you can get the current time and compare with the time stored in the timer variable to decide whether or not the timer has expired. Be aware of the following pitfalls in using the time structures:

- You must use both the seconds and microseconds parts of the time structure; using only one of them is not acceptable.
- You should not combine the values of the seconds and microseconds parts into a single time value. Since each of the parts is a *long* integer, combining them together may result in an overflow and loss of information, which may cause strange behavior of the client.
- To add, subtract, or compare two time structures, always do the operation on each component (seconds or microseconds) separately, and then deal appropriately with any carry or borrow that results.

Server Actions

The server also runs in a loop; however, it can use the regular blocking version of the server program. In addition to the server actions for the Go-Back-N protocol, we will introduce one additional function: simulating loss, errors, and delay. We will need to do this because UDP clients and servers running on the same host will almost never see such behavior.

The actions of the server will be as follows:

- Start server loop.
- Wait for packet to arrive from client.
- If packet is received, check if Count field in packet is 0. If it is 0, this signifies an EOT packet, so quit loop.
- If packet is received, but Count field is not 0, this is a regular data packet. In this case, call the *simulate* function described below to simulate loss, errors, and delay.
- If *simulate* returns 0, then packet is lost or has errors; discard packet and return to start of loop.
- If *simulate* returns 1, then packet is correct. Process the packet according to the protocol actions of the Go-Back-N protocol. Generate an ACK if required.
- After ACK is generated, call the function *ACKsimulate* to simulate ACK loss.
- If the function *ACKsimulate* returns 0, the ACK is lost, and is not transmitted. If the function returns 1, then transmit the ACK.
- End of loop.

Functions *simulate* and *ACKsimulate*:

In the server program, the function *simulate* simulates loss, errors, and delays in the reception of data packets from the client. Errors and loss are not distinguished from each other, as they both result in the packet being dropped. This function uses the configuration parameters *Packet Loss Rate* and *Packet Delay* which are read when the server starts (see below).

Below are the actions of this function:

- Generate a uniformly distributed random number between 0 and 1.
- If the random number is less than *Packet Loss Rate*, then the packet is dropped. Return the value 0.
- Otherwise, the packet is not dropped. Continue to see if it should be delayed.
- If *Packet Delay* is 0, then packet is not delayed. Return the value 1.
- Otherwise, the packet should be delayed. In this case, generate another uniformly distributed random number between 0 and 1, and multiply it with 10^8. Execute a *for* loop that counts up from 1 to the value that results from the above multiplication. This loop will simulate the desired delay. After the loop completes, return 1.

The function *ACKsimulate* simulates loss for ACKs using the configuration parameter *ACK Loss Rate* as follows:

- Generate a uniformly distributed random number between 0 and 1.
- If the random number is less than *ACK Loss Rate*, then the ACK is dropped. Return the value 0.
- Otherwise, the ACK is not dropped. Return the value 1.

Client and Server Configuration Parameters

When the server starts, it prompts the user to enter values for the following configuration parameters:

- *Window Size:* An integer between 1 and 8.
- *Packet Loss Rate:* A real number between 0 and 1.
- *Packet Delay:* An integer which is either 0 or 1.
- *ACK Loss Rate:* A real number between 0 and 1.

When the client starts, it prompts the user to enter values for the following configuration parameters:

- *Window Size:* An integer between 1 and 8. You can assume that the user will always specify the same window size for both the client and the server.

- *Timeout:* The user enters an integer value n in the range 1-10, and the timeout value is then stored as 10^n microseconds. Note that the resultant timeout value should be stored with both seconds and microseconds components.

Output of your program

At specific places in both your client and server programs, you must print out specific messages. The symbol "n" below refers to the sequence number of the transmitted or received packet, and the symbol "c" below refers to the count (number of data bytes) in the transmitted or received packet.

The messages to be printed by the client are:

When a data packet numbered n is sent by the client for the first time:
Packet n transmitted with c data bytes
When a data packet numbered n is retransmitted by the client:
Packet n retransmitted with c data bytes
When a timeout expires:
Timeout expired for packet numbered n
When an ACK is received with number n:
ACK n received
When the "End of Transmission" packet is sent:
End of Transmission Packet with sequence number n transmitted with c data bytes

The messages to be printed by the server are:

When a data packet numbered n is received correctly by the server for the first time:
Packet n received with c data bytes
When a data packet numbered n is received correctly by the server, but is a duplicate or out-of-sequence packet:
Duplicate packet n received with c data bytes
When a data packet numbered n is received by the server, but the *simulate* function causes it to be dropped:
Packet n lost
When contents of data packet numbered n are delivered to the user, i.e., are stored in the output file:
Packet n delivered to user
When an ACK is sent with number n:
ACK n transmitted
When an ACK is generated, but is dropped by the function *ACKsimulate*:
ACK n lost
When the "End of Transmission" packet is received:
End of Transmission Packet with sequence number n received with c data bytes

At the end, before terminating execution, the following statistics should be printed. Do not include the last special "End of Transmission" packet in the count of data packets in these statistics.

For client:

Number of data packets transmitted (initial transmission only)
Total number of data bytes transmitted (this should be the sum of the count fields of all transmitted packets when transmitted for the first time only)
Total number of retransmissions
Total number of data packets transmitted (initial transmissions plus retransmissions)
Number of ACKs received
Count of how many times timeout expired

For server:

Number of data packets received successfully (without loss and not including duplicates)
Total number of data bytes received which are delivered to user (this should be the sum of the count fields of all packets received successfully without loss and without duplicates)
Total number of duplicate data packets received (without loss)
Number of data packets received but dropped due to loss
Total number of data packets received (including those that were successful, those lost, and duplicates)
Number of ACKs transmitted without loss
Number of ACKs generated but dropped due to loss
Total number of ACKs generated (with and without loss)

Testing

The files *test1.txt* and *test2.txt* in the directory *˜sethi/networks/proj2* on *orioles* are sample input files that may be used by you to test your programs. It is strongly suggested that you first use *test1.txt* for all your testing, and only if you have thoroughly debugged your programs, then proceed with using *test2.txt* for further testing.

It is also suggested that you test your programs in phases using the following configuration parameter values:

- Window size 1, Packet and ACK loss rates 0, Packet delay 0, Timeout value n = 5.
- Window size 4, Packet and ACK loss rates 0, Packet delay 0, Timeout value n = 5.
- Window size 1, Packet loss rate 0.2, ACK loss rate 0, Packet delay 0, Timeout value n = 5.
- Window size 4, Packet loss rate 0.2, ACK loss rate 0, Packet delay 0, Timeout value n = 5.
- Window size 4, Packet loss rate 0, ACK loss rate 0.2, Packet delay 0, Timeout value n = 5.
- Window size 4, Packet and ACK loss rates 0, Packet delay 1, Timeout value n = 4.

Once you have tested and debugged with the above parameter values, then you should try combinations of various parameters. Make sure it works well under all conditions, because we will test it out under a variety of different conditions.

Submission

Create two scripts, one each for the client and the server, and show the transfer of the file *test1.txt* with the first set of configuration parameters listed above (i.e., no loss, no delay). The scripts should contain a long listing of the directory that contains your files (using *ls -l*), should show them being compiled, then another long listing (using *ls -l*) of the directory after compilation is complete, a listing of the input file (for the client), show the execution of the programs including the program outputs, and then a listing of the output data file (for the server). Finally, do a *diff* of the input and output files. If you are unsure of the meaning of a script or how to generate one, please ask.

Submit a zipped copy of your project directory. This directory should include all your original source files, the executables, the input and output data files, and the scripts generated by you for your test run.

On the Assignments tab in Sakai, provide the following information as a cover page for your submission: The name of your project directory, the names of the source files, the names of the files that contain the executables, the names of the input and output data files, and the names of the files containing the scripts with the test runs.

Demo

You will be asked to schedule a time after your submission to come in and give a demo of your programs. During the demo, we will run your programs under a variety of different loss and delay conditions. **Your programs must not be modified in any way after you submit your project until the time the demo is completed.**

Grading

Your programs will be graded on correctness, proper output, readability, and documentation. The grade distribution is as follows:

Correctness: 60 %
Readability and Documentation: 20 %
Proper Output and Testing: 20 %

Part of the Correctness grade will depend on how your program performs during the demo. As with Project 1, points for documentation will not be awarded lightly; we will be looking for meaningful variable and function names, good use of comments, good modular structure with appropriate use of functions, good programming style, and proper indentation.

Deadline for submission: Wednesday December 2, 11 pm.

No late assignments will be accepted, because we will have to schedule times for demos. No extensions will be granted for *any* reason. It is to be understood that computers and computing facilities are subject to failure, overload, unavailability, etc., so it is your responsibility to plan and execute your project work sufficiently in advance so as to meet the due date.