Programming Project 2

DUE: May 7th 11:59PM

This project is a group assignment; each group can have no more than two members; you should work on it with your group partner; Collaboration with or help from anyone except your group partner is not permitted. Only one submission per group is required.

In Project 1, you wrote a client and server to transfer a file over TCP. Since TCP provides reliable data transfer, that was a relatively simple task. Now, in Project 2, you will transfer a file over UDP. Since UDP is unreliable and can lose packets, you must implement reliable data transfer functionality into your client and server. The protocol that you will implement to provide this functionality is the Stop-and-Wait Protocol.

We will try to keep the requirements as similar as possible to Project 1, so you can re-use a good part of the code that you wrote there. Once again, both the hostname and the server port number may be hardcoded into both the client and server programs, but this should be done in such a way that they are easy to change.

The server starts by waiting for an initial message to arrive from the client.

The client starts by prompting the user to enter the name of the file to be transferred. The client then sends the filename to the server in its initial message. The server reads the file and sends it to the client in a series of packets as described below. The client 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 server 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. The server transmits each line to the client in a separate packet. The client receives the packets and puts their data into distinct lines in the output file.

The format of a data packet is shown in the figure below:

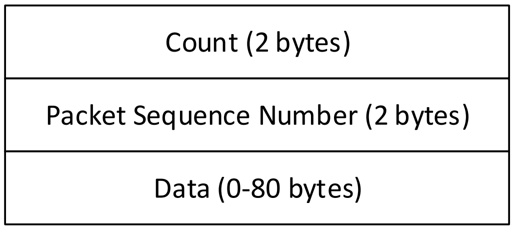

Figure 1: Format of a data packet.

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 Figure 1. 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. The second field of the header is called the packet sequence number. Each packet transmitted by the server is assigned a sequence number starting with 0 and *alternating between 0 and 1* for each packet. Note that this is different from how you used the sequence number in Project 1, because now we are using the Stop-and-Wait Protocol in which only 0 and 1 are used as sequence numbers.

The filename is sent by the client to the server using the same format as the data packet shown above. In this packet, the data characters will contain the filename, but there will not be any newline character at the end. The Count will be the number of data characters in the packet (i.e. length of the filename), and the sequence number will be 0.

The transmission of each packet will be done *as a single UDP segment* using a single UDP `sendto` operation, with this segment containing both the 4-byte header and the data characters as described above. Note that this is different from how you did the transmission in Project 1. Similarly, the receiver will use a single UDP `receivefrom` operation to receive the packet which will contain both the header and the data bytes.

After sending each data packet, the server waits for an ACK to be returned from the client. The format of an ACK packet is shown in Figure 2, where the ACK sequence number is either 0 or 1.

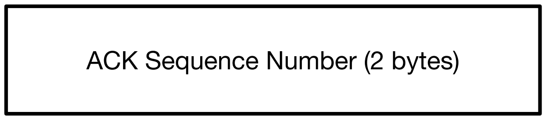

Figure 2: Format of an ACK packet.

When the server 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 signifying "End of Transmission" (EOT). 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 server has received ACKs for all transmitted data packets*. The client will not transmit an ACK for the EOT packet, and the server will not expect any ACK to be returned for it. The server program can terminate once the EOT packet has been transmitted. When the client receives the EOT packet, it closes the output file and also terminates.

Simulation of Loss

Although your packets and ACKs are transmitted over UDP which provides unreliable data transfer, it is not common to observe packet loss when transmitting to the same host (which is the case with us). For this reason, we will explicitly simulate packet and ACK loss by implementing the two functions described below. Packet loss is implemented in the server before the data packet is actually transmitted and ACK loss is implemented in the client before the ACK is actually transmitted.

The function `SimulateLoss` simulates loss in the server's transmission of data packets to the client. This function uses the configuration parameter *Packet Loss Ratio* which is a user-specified parameter (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 Ratio*, return the value 1, otherwise return the value 0.

The `SimulateLoss` function is called just before the server transmits a data packet. If this function returns 0, the packet is transmitted normally, but if it returns 1, the packet is not transmitted and the server simply continues to its next action just as if the packet had been transmitted.

The function `SimulateACKLoss` simulates loss for ACKs using the configuration parameter *ACK Loss Ratio* as described below. This function is implemented in the client and it causes the client to not transmit an ACK when a loss is indicated.

- Generate a uniformly distributed random number between 0 and 1.
- If the random number is less than *ACK Loss Ratio*, return the value 1, otherwise return the value 0.

The `SimulateACKLoss` function is called just before the client transmits an ACK to the server. If this function returns 0, the ACK is transmitted normally, but if it returns 1, the ACK is not transmitted and the client simply continues to its next action just as if the ACK had been transmitted.

Client and Server Configuration Parameters

When the client starts, it is given the following configuration parameters by the user:

- *Input File Name*: The name of the input file that is sent from the client to the server.
- *ACK Loss Ratio*: A real number between 0 and 1.

When the server starts, it is given the following configuration parameters by the user:

- *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.
- *Packet Loss Ratio*: A real number between 0 and 1.

These parameters should ideally be provided as command-line arguments when the client and server are started. Alternatively, the client and server may prompt the user to enter values for these parameters and then read in these values immediately after startup. It is not acceptable to hard code values for these parameters in your code.

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 (note that the sequence number must always be either 0 or 1), 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 **server** are:

- When a data packet numbered *n* is generated for transmission by the server for the first time:
  Packet n generated for transmission with c data bytes
- When a data packet numbered *n* is generated for re-transmission by the server:
  Packet n generated for re-transmission with c data bytes
- When a data packet numbered *n* is actually transmitted by the server:
  Packet n successfully transmitted with c data bytes
- When a data packet numbered *n* is generated, but is dropped because of loss:
  Packet n lost
- When an ACK is received with number *n*:
  ACK n received
- When a timeout expires:
  Timeout expired for packet numbered n
- When the "End of Transmission" packet is sent:
  End of Transmission Packet with sequence number n transmitted

The messages to be printed by the **client** are:

- When a data packet numbered *n* is received by the client for the first time:
  Packet n received with c data bytes
- When a data packet numbered *n* is received by the client, but is a duplicate packet:
  Duplicate packet n received with c data bytes
- 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 with number *n* is generated for transmission:
  ACK n generated for transmission
- When an ACK is actually transmitted with number *n*:
  ACK n successfully transmitted

- When an ACK is generated, but is dropped because of loss:
  ACK n lost
- When the "End of Transmission" packet is received:
  End of Transmission Packet with sequence number n received

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. Also do not include the first packet sent by the client with the filename.

- For server:
  1. *Number of data packets generated for transmission (initial transmission only)*
  2. *Total number of data bytes generated for transmission, initial transmission only (this should be the sum of the count fields of all packets generated for transmission for the first time only)*
  3. *Total number of data packets generated for retransmission (initial transmissions plus retransmissions)*
  4. *Number of data packets dropped due to loss*
  5. *Number of data packets transmitted successfully (initial transmissions plus retransmissions)*
  6. *Number of ACKs received*
  7. *Count of how many times timeout expired*
- For client:
  1. *Total number of data packets received successfully*
  2. *Number of duplicate data packets received)*
  3. *Number of data packets received successfully, not including duplicates*
  4. *Total number of data bytes received which are delivered to user (this should be the sum of the count fields of all received packets not including duplicates)*
  5. *Number of ACKs transmitted without loss*
  6. *Number of ACKs generated but dropped due to loss*
  7. *Total number of ACKs generated (with and without loss)*

Testing

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

- Packet and ACK loss rates 0, Timeout value $n = 5$
- Packet loss rate 0.2, ACK loss rate 0, Timeout value $n = 5$.
- Packet loss rate 0, ACK loss rate 0.2, Timeout value $n = 5$.
- Packet loss rate 0.2, ACK loss rate 0, Timeout value $n = 4$.

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

Grading

Your programs will be graded on correctness, proper output, readability, and documentation, as specified in the project rubric. Parts of the Correctness grade will depend on how your program performs during our testing. 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.