Implementing TCP Features in UDP

Start Assignment

Due May 9 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** zip and tgz

1. Purpose

In this assignment you will implement the **stop-and-wait** and **sliding window** algorithms and evaluate their performance in transferring 20,000 packets over the 1Gbps network in the Linux Remote Lab.

2. UDP

UDP (**User Datagram Protocol (http://en.wikipedia.org/wiki/User_Datagram_Protocol)**) is a connectionless, unreliable transport layer protocol which does not prevent loss or or re-ordering of messages.

3. HW2 Test Program

Your homework will allocate a 1460-byte message[], and evaluate the performance of UDP point-to-point communication using four different test cases:

1. **Case 1: Unreliable test** simply sends 20,000 UDP packets from a client to a server. The actual implementation can be found in:
   - void clientUnreliable(UdpSocket &sock, const int max, int message[]): sends message[] to a given server using the sock object max (=20,000) times .
   - void serverUnreliable(UdpSocket &sock, const int max, int message[]): receives message[] from a client using the sock object max (=20,000) times. However it does not send back any acknowledgment to the client.

   This test may hang the server due to some UDP packets being sent by the client but never received by the server.

2. **Case 2: Stop-and-wait test** implements the stop-and-wait algorithm (this can be done as a sliding window with a size = 1), i.e., a client writes a message sequence number in message[0], sends message[] and waits until it receives an integer acknowledgment of that sequence number from a server, while the server receives message[], copies the sequence number from message[0] to an acknowledgment message, and returns it to the client. You are to implement this algorithm in the following two functions:

- int **clientStopWait**(UdpSocket &sock, const int max, int message[]): sends message[] and receives an acknowledgment from the server max (=20,000) times using the sock object. If the client cannot receive an acknowledgment immediately, it should start a Timer. If a timeout occurs (i.e., no response after 1500 usec), the client must resend the same message and repeat the process of waiting for the ACK. The function must count the number of messages retransmitted and return it to the main function as its return value. Note that in order to check for a timeout, you will need to find a way to decide if the acknowledgement has arrived without blocking on recvfrom. There are many ways to do this. setsockopt can be use to set a socket to non-blocking, recvfrom has a MSG_NOBLOCK flag that can be set, you can also use poll() or select() to see if there is data available before calling recvfrom.
- void **serverReliable**(UdpSocket &sock, const int max, int message[]): repeats receiving message[] and sending an acknowledgment at a server side max (=20,000) times using the sock object.

This test may not reach the peak performance supported by the underlying network. This is because the client must wait for an acknowledgment every time it sends out a new message.

3. **Case 3: Sliding window** implements a sliding window algorithm, using the first element in the message[] (a 1460 byte message, its contents don't matter other than the integer value in first element) for storing sequence numbers and acknowledgement numbers.
   - A *client* keeps writing a message sequence number in message[0] and sending message[] as long as the number of in-transit messages is less than a given windowSize
   - The *server* receives message[], saves the message's sequence number (stored in message[0]) in a local buffer array and returns *an acknowledgement. You may implement selective ack or cumulative ack*.

   You are to implement this algorithm in the following two functions:
   - int **clientSlidingWindow**(UdpSocket &sock, const int max, int message[], int windowSize): sends message[] and receiving an acknowledgment from a server max (=20,000) times. As described above, the client can continuously send a new message[] and incrementing the sequence number as long as the number of in-transit messages (i.e., # of unacknowledged messages) is less than windowSize. That number should be decremented every time the client receives an acknowledgment. If the number of unacknowledged messages reaches windowSize, the client should start a Timer. If a timeout occurs (i.e., no response after 1500 usec), it must resend the message. You may decide if you will resend all messages, just the lowest unacked, or all unacked (in the case of selective acknowledgement). The function must count the number of messages (not bytes) retransmitted and return it to the main function as its return value.
   - void **serverEarlyRetrans**(UdpSocket &sock, const int max, int message[], int windowSize) receives message[] and sends an acknowledgment to the client max (=20,000) times. Every time the server receives a new message[], it must save the message's sequence number in its array and return an acknowledgement (as described above).

   [**NOTE: the windowSize in this assignment refers to the number of *messages*, not the number of *bytes***].
   This test may get close to the peak performance supported by the underlying network. This is because the client can send in a pipelined fashion as many messages as the server can receive.
4. **Case 4: Sliding Window with Errors**
   Modify the code so that test case #3 runs for a sliding window of size 1 and size 30. This will simulate errors since you're unlikely to run into many on the Linux Lab. Add code in the serverEarlyRetrans function so that packets are randomly dropped every N% of the time where N is every integer percent from 0 to 10. You can simulate a drop by just not returning an ACK when you receive a packet.

Fun Fact: Modern C++ has a much better random function than rand() now --

```
#include <random>

std::random_device rd;  //Will be used to obtain a seed for the random number engine
 std::mt19937 gen(rd()); //Standard mersenne_twister_engine seeded with rd()
 std::uniform_int_distribution<> randomInt(0,100);    // set up the range

 int dropVal = randomInt(gen);  // pick a random value between 0 and 100
```

4. Statement of Work

```
Client side example:
   $ ./hw3 csslab1

Server side example:
   $ ./hw3
   Choose a testcase
      1: unreliable test
      2: stop-and-wait test
      3: sliding window
      4: sliding window with errors
   -->
```

Type 1, 2, 3, and 4 to evaluate the performance of one of the three different test cases respectively.

5. What to Turn in

The homework is due at 11:59 PM on the due date.

Criteria	Percentage
Documentation of (1) a stop-and-wait and (2) a sliding window implementation, including explanations and illustrations in details. Overall professionalism of the document will be included in this.	10
Source code that adheres good modularization, coding style, and an appropriate amount of comments. The source code is graded in terms of (1) clientStopAndWait( )'s message send and ack receive, (2) clientStopAndWait( )'s timeout and message re-transfer, (3) serverReliable( )'s message receive and ack send, (4) clientSlidingWindow( )'s message transfer and ack receive, (5) clientSlidingWindow( )'s timeout and duplicated message re-transfer, (6) serverEarlyRetrans( )'s cumulative acknowledgment, (7) serverEarlyRetrans random packet drop and (8) comments.	40
Execution output such as a snapshot of your display/windows. Type **import -window root X.jpeg; lpr -Puw1-320-p1 X.jpeg** on a uw1-320 linux machine. Or, submit partial contents of standard output redirected to a file. You don't have to print out all data. Just one page evidence is enough.	5
Performance evaluation The correctness is evaluated in terms of 1. sliding window performance over 1Gbps (graph of time and retransmits from 1-30 window size) 2. performance with random drops (graph of time and retransmits from 1-30 window size with 11 lines, one for each of 0%-10% drops)	20

Discussion of (1) the difference in performance between stop and wait and sliding window, (2) influence of window size on sliding window performance, (3) differences in the number of messages retransmitted between stop-and-wait and sliding window algorithms. (4) Include discussion of the effect of drop rates on both the window size of 1 and 30. You should also discuss about the difference in the number of messages retransmitted between stop-and-wait and sliding window algorithms. Make sure to discuss how your design choices (eg, cumulative vs selective ack and how you chose to retransmit) did/would affect performance.	25
Total	100