HW3 - Sliding Window

Re-submit Assignment

Due May 15 by 11:59pm **Points** 25 **Submitting** a file upload
File Types zip and tgz **Available** after Apr 10 at 12am

1. Overall Requirement

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

2. Existing code wrapping up UDP and Timer behaviors

To finish this assignment, it is recommended that you use the UdpSocket class which cane be found in **hw3.zip**. You may also want to use the Timer class in the zip file for your implementation and performance evaluation.

You may find this page helpful -- **https://faculty.washington.edu/athirai/css432/prog/prog2_faq.html (https://faculty.washington.edu/athirai/css432/prog/prog2_faq.html)**

Explanation about the UdpSocket and Timer classes

UdpSocket Class

Methods	**Descriptions**
UdpSocket(int port)	A default constructor that opens a UDP datagram socket with a given port. Use the last five digits of your student ID as the port number.
~UdpSocket()	A default destructor that closes the UDP datagram socket it has maintained.
bool **setDestAddress**(char ipName[])	Set the destination IP address corresponding to a given server ipName (e.g., "uw1-320-16"). This method must be called before sending the first UDP message to the server.
int **pollRecvFrom**()	Check if this socket has data to read. The method returns a positive number if there is data to read; 0 or negative, otherwise. In general, a program is blocked to receive data (e.g., using recvFrom()) if there are no data to receive. Call this method if you need to prevent your program from being blocked when receiving data.

int **recvFrom**(char msg[], int length)	Receive data into msg[] of size length. This method can be used both to receive a message from a client as well as an acknowledgment from a server. As an example, if you wish to receive a single char variable, say c, your call should be recvFrom(&c, sizeof(c)).
int **sendTo**(char msg[], int length)	Send msg[] of size length size from a client to a server (whose address must have been previously set by setDestAddress()). As an example, if you wish to send a single char variable, say c, your call should be sendTo(&c, sizeof(c)).
ackTo(char msg[], int length)	Send an acknowledgment from a server to a client. Prior to calling this method, a server must receive at least one message from a client, (i.e, have called recvFrom() at least one time).

Timer Class

Methods	**Descriptions**
Timer()	A default constructor that zero-initializes its internal tv_sec and tv_usec variables.
start()	calls gettimeofday() to get the current date information and saves it in tv_sec and tv_usec.
long **lap**()	calls gettimeofday() to get the current date information and returns the time elapsed since its start() method was last called.
long **lap**(long oldTv_sec, long oldTv_usec)	calls gettimeofday() to get the current date information and returns the time elapsed since the time specified by oldTv_sec and oldTv_usec.
long **getSec**()	Retrieve the internal tv_sec variable value, which was set by the last start() method call.
long **getUsec**()	Retrieve the internal tv_usec variable value, which was set by the last start() method call.

3. HW3 Program

To start, please copy **hw3.cpp** to your working directory. This program, hw3.cpp, instantiates sock, (i.e., a UdpSocket object), allocates a 1460-byte message[], and evaluates the performance of UDP point-to-point communication using three 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. **This is an already implemented function, you don't need to implement it.**
- **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 is an already implemented function, you don't need to implement it.**

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, 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. **Please create a file named udphw3.cpp, and write code in updhw3.cpp to implement the described "stop-and-wait" 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. The function must count the number of messages retransmitted and return it to the main function as its return value.
   - **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.

   Again, create a new file named udphw3.cpp, and DO NOT modify hw3.cpp. 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 the "sliding window" algorithm, using the first element in the message[] for storing sequence numbers (in messages sent by client) and acknowledgement numbers (in messages sent by server).
   - 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 a *cumulative acknowledgement*, i.e., the last received message in order.

   Update udphw3.cpp to implement this "sliding window" 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 using the sock object. As described above, the client can continuously send a new message[] and increasing 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 with the minimum sequence number among those which have not yet been acknowledged. The function must count the number of messages (not bytes) re-transmitted 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 using the sock object. Every time the server receives a new message[], it must save the message's sequence number in an array and return a *cumulative acknowledgment*, i.e., the last received message in order.

[**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 pipeline fashion as many messages as the server can receive. **By now, you should have at least four new functions implemented in udphw3.cpp.**

4. **Case 4:** you will modify the code implemented in Case 3. **Please save the new version of your code as udphw3case4.cpp (modified from udphw3.cpp). Please also make a copy of hw3.cpp, and name this new file as hw3case4.cpp.**

   Modify hw3case4.cpp to include test case 4. Test case 4 runs similar to test case 3, but the "sliding window" algorithm/implementation **only runs for a sliding window of size 1 and size 30**. Add code to the **serverEarlyRetrans** function in **udphw3case4.cpp** so that **packets are randomly dropped packets N% of the time**, where N is every integer from 0 to 10. You will also need to modify the output of **hw3case4.cpp** so that it outputs the drop percentage (instead of the window size) into the file. You can simulate a drop by just not returning an ACK when you receive a packet.

4. Statement of Work

Upzip the provided zip file and extract the following files to your working directory:

1. hw3.cpp
2. UdpSocket.h
3. UdpSocket.cpp
4. Timer.h
5. Timer.cpp
6. udp.plt

Complete implementations of the following four functions that have been outlined above ***in a separate file,* udphw3.cpp**:

1. int **clientStopWait**(UdpSocket &sock, const int max, int message[])
2. void **serverReliable**(UdpSocket &sock, const int max, int message[])
3. int **clientSlidingWindow**(UdpSocket &sock, const int max, int message[], int windowSize)
4. void **serverEarlyRetrans**(UdpSocket &sock, const int max, int message[], int windowSize)

Some functions might not use all arguments passed to them. Change the definition of PORT in hw3.cpp to the last five digits of your student ID, and thereafter compile them with g++ as follows:

```
$ g++ UdpSocket.cpp Timer.cpp udphw3.cpp hw3.cpp -o hw3
```

If a compilation successfully completes, you will get an executable named hw3. Run this executable on both a client and a server machine. The program will display the following messages to the standard error:

```
Client side example:
   $ ./hw3 uw1-320-10 > data

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

Type 1, 2, or 3 to evaluate the performance of one of the three different test cases respectively. The main function in hw3.cpp will redirect the output showing the performance results in the data file. Use cerr to print out any debugging information or message sequence numbers. Don't use cout, otherwise your debugging and message sequence information will be mixed up with performance results in the data file. If you wish to store cerr information (file descriptor for stderr is 2, stdout is 1) in a separate file, e.g., data1 or data2 (one for the client, one for the server), you should type:

```
Client side example:
   $ ./hw3 uw1-320-10 2> data1

Server side example:
   $ ./hw3 2> data2
```

For test case #1, all you have to do is simply repeat running the program several times and observe if it hangs due to the drop-off of some UDP messages, and the discrepancy between the iteration number and the sequence number of the message received by the server.

For test case #2, run your program over 1Gbps network and obtain an elapsed time for the stop and wait protocol. To view those results with gnuplot, you need to edit the **udp.plt** file.
1. Replace TTTTT (in udp.plt as shown below) with a elapsed time for 1Gbps network stop-and-wait performance test.

```
   plot "1gbps.dat" title "1gbps sliding window" with linespoints,
   TTTTT title "1gbps stopNwait" with line
```

2. Type the following commands

```
$ gnuplot udp.plt
$ ps2pdf udp.ps
```

For test case #3, conduct performance evaluation over the 1Gbps for the window size = 1 to 30 for the sliding window protocol. Store their performance results in **1gbps.dat**. **Note: 1gbps.dat already exists but contains made-up results. Do not Copy them!** HINT: the "stop-and-wait" algorithm which has a window size 1, and the "sliding widow" algorithm with window size 1 should have similar elapsed time. To view all results using gnuplot, type:

```
$ gnuplot udp.plt
$ ps2pdf udp.ps
```

For test case #4, please recompile your code as follows:

```
$ g++ UdpSocket.cpp Timer.cpp udphw3case4.cpp hw3case4.cpp -o hw3case4
```

Then store the performance results for "window size = 1" and "windows size = 30" in "1gbpsa-1.dat" and "1gbpsa-30.dat," respectively. You may use a similar plot file, e.g., udpa.plt, to plot the data.

```
$ gnuplot udpa.plt
$ ps2pdf udpa.ps
$ evince udpa.pdf
```

5. What to submit

Do not submit the files that are given to you for the project.

Only submit your write-up, and udphw3.cpp, udphw3case4.cpp, hw3case4.cpp.

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()'s random packet drop (for case 4) and (8) comments.	40

Execution output such as a snapshot of your display/windows. 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 in form of the udp.pdf The correctness is evaluated in terms of 1. stop-and-wait performance over 1Gbps 2. sliding window performance over 1Gbps 3. performance with random 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 re-transmitted between "stop-and-wait" and "sliding window" algorithms, and (4) 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 re-transmitted between stop-and-wait and sliding window algorithms.	25
Total	100

Your grade for this assignment will be "total points/4."

Some Rubric (1)

Criteria	Ratings		Pts
Source Code	10.0 pts Full Marks	0.0 pts No Marks	10.0 pts
Discussion	7.0 pts Full Marks	0.0 pts No Marks	7.0 pts
Performance Evaluation	5.0 pts Full Marks	0.0 pts No Marks	5.0 pts
Documentation	2.0 pts Full Marks	0.0 pts No Marks	2.0 pts
Output	1.0 pts Full Marks	0.0 pts No Marks	1.0 pts
			Total Points: 25.0