CSS 432
Program 2: Sliding Window

Professor Dimpsey

1. Purpose

This assignment implements the sliding window algorithm and evaluates its performance improvement over a 1Gbps network.

2. UDP

UDP (User Datagram Protocol) is a connectionless, unreliable datagram transfer protocol. In this lab a program will be implemented using the sliding window algorithm over packets sent using UDP. This will require the algorithm to account for typical network problems: packet loss and out-of-order delivery. The program will send packets whose size is fitted to the Ethernet MTU (Maximum Transmission Unti), 1500 bytes. To fit within the MTU (1500 bytes) the program utilizes a data size of 1460 bytes. This allows for 40 bytes of overhead. Fitting within an Ethernet frame avoids further fragmentation at the data link layer and allows the program to accurately implement the sliding window algorithm over UDP.

As a starting point for this program two classes have been provided: **UdpSocket** and **Timer**. These classes have been posted to Canvas. Please use these to implement this lab. The Timer class makes measuring an interval of time simple by calling the Start() member function at the beginning of the interval to measure and the End() at the conclusion. The details of **UdpSocket** are below.

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[], int port)	Set the destination IP address corresponding to a given server ipName and port. 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 the number of bytes to read or 0 if no messages have arrived. In general, a program is blocked to receive data if there are no data to receive. Call this method if you need to prevent your program from being blocked when receiving data.
int sendTo(char msg[], int length)	Send *msg[]* of the *length* size from a client to a server. The server address is set by *setDestAddress( )* before this call. If you send a single *char* variable, say **c**, your call should be *sendTo(**&c**, sizeof(**c**))*.
int recvFrom(char msg[], int length)	Receive data into *msg[]* of the *length* size. This method can be both used to receive a message from a client as well as an acknowledgment from a server. If you receive a single *char* variable, say **c**, your call should be

	*recvFrom(**&c**, int 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, call *recvFrom()* at least one time.)

3. HW2 Test Program

In addition to the **UDPSocket** and **Timer** classes, a driver program, named **hw2.cpp** has also been posted to Canvas. This is a test driver program which has been *partially* coded. This program requires you to complete the test program by implmenting a Stop-and-wait algorithm and the sliding window algorithm. The program instantiates **sock**, (UdpSocket object), allocates a 1460-byte **message[]**, and then evaluates the performance of UDP point-to-point communication using three different test cases:

1. Unreliable transer (already impmlemented)
2. Stop-and-Wait (required to implement)
3. Sliding Window (required to implement)

Details of each of these cases is provided below.

Case 1, Base UDP (unreliable): Sends 20,000 UDP packets from the client to a server. In this case if a packet is dropped along the way the server will not know and will hang continuing to wait for it. The actual implementation can be found in the hw2.cpp file in these functions:

- **void ClientUnreliable(UdpSocket &sock, int max, int message[]):** repeats **max** times of sending **message[]** to a given server using the sock object.
- **void ServerUnreliable(UdpSocket &sock, int max, int message[]):** repeats **max** times of receiving **message[]** from a client using the sock object. No acknowledgement is returned.

Case 2, Stop-and-wait test: Follows the stop-and-wait algorithm in that a client writes a message sequence number in message[0], sends message[] and waits until it receives an integer acknowledgment from a server, while the server receives message[], copies the sequence number from message[0] to an acknowledgment, and returns it to the client. You are to implement this algorithm in the following two functions:

- **int ClientStopWait(UdpSocket &sock, int max, int message[]):** repeats sending **message[]** and receiving an acknowledgment at the client side max times using the sock object. If the client cannot receive an acknowledgment immediately, it should start a timer and wait 1500 usec. If the wait timeouts, the client should 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, int max, int message[]):** repeats receiving **message[]** and sending an acknowledgment at a server side max times using the sock object.

Case 3, Sliding window: Implements the sliding window algorithm. The client keeps writing a message sequence number in message[0] and sending the message[] as long as the number of in-transit messages is less than a given window size. The server receives message[], records the sequence number in its array and returns as its acknowledgment the minimum sequence number of messages it has not yet received (called a cumulative acknowledgment). *NOTE! the window size in our assignments means the number of messages but not the number of bytes*. You are to implement this algorithm in the following two functions:

- **int ClientSlidingWindow(UdpSocket &sock, int max, int message[], int windowSize):** repeats sending **message[]** and receiving an acknowledgment at a client side **max** times using the sock object. As described above, the client can continuously send a new message[] and incrementing its sequence number as long as the number of in-transit messages, (i.e., # of unacknowledged messages) is less than windowSize. If the # of unacknowledged messages reaches windowSize, the client should start a timer for 1500usec. If the timer timeouts, it must follow the sliding window algorithm and resend the message with the minimum sequence number among those which have not yet been acknowledged. The function must count the number of messages retransmitted and return it to the main function as its return value.
- **void ServerEarlyRetrans(UdpSocket &sock, int max, int message[], int windowSize):** repeats receiving **message[]** and sending an acknowledgment at a server side **max** times using the sock object. Every time the server receives a new message[], it must record this message's sequence number in its array and returns a cumulative acknowledgment.

4. Statement of Work

Copy the following files from Canvas:

1. hw2.cpp
2. UdpSocket.h
3. UdpSocket.cpp
4. Timer.h
5. Timer.cpp

Code in **hw2.cpp** the following four functions that have been outlined above:

1. int ClientStopWait(UdpSocket &sock, int max, int message[]);
2. void ServerReliable(UdpSocket &sock, int max, int message[]);
3. int ClientSlidingWindow(UdpSocket &sock, int max, int message[], int windowSize);
4. void ServerEarlyRetrans(UdpSocket &sock, int max, int message[], int windowSize);

Change the **hw2.cpp**'s `PORT` definition to the last five digits of your student ID.

Compile the files with g++ to produce the executable **hw2**

```
$ g++ UdpSocket.cpp Timer.cpp udp.cpp hw2.cpp -o hw2
```

Run this program both on a client and a server hosts. For the client-side pass in as an argument the host name of the server. The server does not take any arguments. The program will display the following messages to the standard error:

```
Client side example:
   $ ./hw2 uw1-320-10 > data
   Choose a testcase
      1: unreliable test
      2: stop-and-wait test
      3: sliding window
   -->

Server side example:
   $ ./hw2
   Choose a testcase
```

```
      1: unreliable test
      2: stop-and-wait test
      3: sliding window
   -->
```

Type 1, 2, and 3 for evaluating the performance of the three different test cases respectively. The main function in **hw2.cpp** will print out the resulting data to cout. In the above example on the client side this will re-direct to the file called **data**.

Test case #1 has already been implemented for you. Run this on the client and server to make sure you have proper connectivity. You can recompile and run hw2 in verbose mode to get a printout to cerr on each message. Run this test case a number of times until the server hangs which shows a dropped message. Nothing needs to be turned in for this test case.

For the test case #2, run your program in the linux lab a number of time with the client and server on different hosts. Average the times to get good measure on the elpased time for this algorithm.

For test case #3, measure the window size from 1 - 30.

5. What to Turn in

Turn the following results into Canvas in a .zip file.

Criteria	Percentage
Documentation in detail of the the stop-and-wait and sliding-window algorithms including illustrations.	4pts (20%)
Source code that adheres to 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, and (7) coding guidelines.	10pts (50%)
Performance results in graph and discussion which illustrates the relative performance of stop-and-wait and sliding window with the differing window sizes from 1 - 30. Also, include discussions on the retransmission rate for the algorithms.	6pts (30%)
Total	20pts (100%)

6. FAQ

This FAQ page may answer your quetions. Click here