Instructions

Sliding Window Protocols

Goal: To implement two sliding window protocols, namely: Stop-and-Wait and Go-Back-N.

Description:

Sliding window protocols are used to ensure a reliable transfer of data. The client (sender) sends packets to the server (receiver) and waits for acknowledgments from the server to ensure that each packet has been received correctly. In case a packet is lost or received with errors, the client has to retransmit that packet again to the server. A sliding window is used by client that specifies which packets can be sent. Upon receiving acknowledgments from the server, the window slides, allowing next packets to be transmitted to the server.

In this lab, you will implement two protocols.

1- Stop-and-Wait: which is a one-bit sliding window protocol. The client sends one packet each time and waits for an acknowledgment from the server. Once the acknowledgment is received, the window slides one position and the client sends the next packet. The process continues until all packets received correctly by the server. Clearly, this protocol has a very low efficiency. You are required to implement the server-client pair for this protocol.

2- Go-Back-N: in this protocol, the size of the window is larger than one, allowing the client to send multiple packets before receiving any acknowledgment from the server. As the client receives acknowledgments from the server, the window slides and next packets can be transmitted. This protocol has a higher efficiency than Stop-and-Wait protocol. You are required to implement the client side of this protocol. The server is provided to you.

Section 1: Stop-and-Wait

In this section, you will write a server-client pair (two applications; one running as server and one as client) that runs the Stop-and-Wait protocol.

1. Similar to the chat client-server pair (Lab 1 instructions), you will need a Server Socket in the server that waits until a client connects.

2. After having a socket connected, You can read inputs from the socket's input stream using read() method of the bufferedReader. You can also write inputs to the socket using write() method of DataOutputStream. For simplicity, in this lab, we will exchange Integers between client and server rather than strings.

3**. Client:**

a- Use Scanner to read the number of packets from the user. This can be done as
Scanner scr = new Scanner(System.in);
Use nextInt() method of scanner to read the number of packets from the user, and store it in a variable, *noPackets*.

b- Send that number to the server

c- Define another variable, *sent*, that keeps track of the packet in the sliding window. Initially, *sent* is set to 1

d- Send the packet number (*sent*) to the server and wait for an acknowledgment.

e- Read the acknowledgement from the server, if the received number is equal to sent, slide the window by one position, i.e. *sent* = *sent* + 1.

f- Continue the process until all packets have been sent.

4. **Server:**

a- Read the number of packets from the client.

b- Define variable *lastAck* that keeps track of the last packet that has been acknowledged. Initially, *lastAck* = 0

c- Read the input from the client, if received number equals *lastAck* + 1, send back the received number to the client. Increment *lastAck* by 1

d- Continue the process until all packets have been acknowledged.

Section 2: Go-Back-N, Client

In this section, you will do the first step to implement the client side of the Go-Back-N protocol. You are asked to implement the client similar to Stop-and-Wait client. However, the client at this time is responsible for sending packets only, DON'T WAIT FOR ACKNOWLEDGMENTS FROM THE SERVER. This will be done in section 3 of this lab. The client should be able to send all packets, one after the other. In other words: window size equals number of packets in this case.

You need to read two pieces of information from the user and **write them to the server**:

a- The number of packets to be sent to the server (*noPackets*)

b- The probability of dropping packets by the server (*probError*), which is a number between 0-100. For this section, let *probError* = 0

Section 3: Acknowledgment Receiver Thread

In section 2, we implemented a client that sends packets without reading acknowledgments from the server. In Go-Back-N, the client needs to send packets to the server. Simultaneously, it should be listening to the server, to slide the window upon receiving acknowledgments from the server. In order to have both tasks, sending packets and receiving acknowledgments, running at the same time, we will implement the acknowledgment receiver in a separate thread, using java.lang.Thread.

1. In the client program that you created in Section 2, define a static variable called *lastAck* that keeps track of last acknowledged packet. Initially, *lastAck* = 0

2. In addition, create a static method *setAckNum* that updates the *lastAck* variable

public static void setAckNum(int ackNum)

This method should update *lastAck* to the new value *ackNum*

3. Create a new class (in separate file) and call it Listener. The new class that implements the Runnable interface is as follows:

public class Listener implements Runnable

4. Create a constructor for the class Listener that accepts a socket as an argument

public Listener (Socket socket)

5. The class Listener should have a method with the name: *void run()*.

6. Inside run() method, you should listen continuously to the server to read input, which are the acknowledgments. Once an acknowledgment is received, you should update *lastAck* variable in the client program. This can be done using *setAckNum* from step 2.

7. Go back to the client program. Create a thread and Pass the Listener class to the new thread.

Thread thread=new Thread(new Listener(socket));

8. To run the thread, simply use the command: *thread.start().*

Once the command *thread.start()* is called, the *run()* method from the Listener class is called to run on the thread. Now Listener should be listening continuously to the server to make the window slide upon receiving acknowledgments from the server.

Section 4: Sliding Window

In this section, you will implement the sliding window on the client program.

1. In the client program, you need to define the following variables:

- Window size (*wSize*): the size of the sliding window, which represents N in Go-Back-N

- Timeout (*timeOut*): the timeout interval, after which the client will go back and retransmits packets.

Remember: you already have other variables defined in previous two sections. *noPackets*, *lastAck* and *sent*.

You can get the values for *wSize* and *timeOut* from the user. Use Scanner to get them from the user.

2. You need to define an array *timer*[] of type long, that keeps the times when each packet was sent. The size of this array must be equal to *wSize*. You can set the times using System.currentTimeMillis() after sending each packet.

3. At the beginning, the client should send at least *wSize* packets to the server, one after the other. Store the time when sending each packet in the array *timer*[] as follows

timer[(sent-1) % wSize] = System.currentTimeMillis();

Q: why at least wSize packets can be sent in step 3? Give an example that it could be more than wSize.

4. Next, while there are more packets to be sent, you should monitor two things:

a- If *lastAck* is updated, meaning that an acknowledgment has been received**.** You can check this by checking if *(sent - lastAck) <= wSize*. In this case, you need to send the next packet to the server, and increment the variable *sent*. Don't forget to update *timer*[] in this case.

b- The time passed since sending the first packet in the window (current time - *timer*[*lastAck % wSize*]) is greater than *timeOut*. In this case, we have a timeout.

Q: what should you do in this case?

Section 5: Performance

You can easily measure the time since you send the first packet, until you receive the acknowledgment for the last packet.

Q: How much time it took to send 10 packets with *wSize*= 4 and *probError* = 0, for different timeouts. Record your answer for different values of timeout.

Q: Is it preferable to have a small or large timeout interval? Explain.

Don't forget to close the socket at the end of the application.

Sample output:

The channel connecting client and server has a random delay between 1-3000 ms for each acknowledgment. Therefore, your output could be slightly different from the one below.

Client:

```
Connected to : localhost:9876
Enter number of packets to be sent to the server, 0 to Quit:
7
Enter the probability of dropping packets (0-100):
0
Enter the window size:
3
Enter the timeout interval:
12000
```

sending packet no:1
sending packet no:2
sending packet no:3
Received packet no:1
sending packet no:4
Received packet no:2
sending packet no:5
Received packet no:3
sending packet no:6
Received packet no:4
sending packet no:7
Received packet no:5
Received packet no:6
Received packet no:7
Total time to send all packets is 10 seconds
All packets have been sent successfully
Quit

Server:

Server online.
Host name: localhost
Host Address: xxx.xxx.x.xx HYPERLINK "http://192.168.0.12:9876/":9876
waiting for requests.
request received. request number: 1 client: /127.0.0.1:58184
connection established:
service type:ARQ_SERVER

mode:VERBOSE
client id:1
socket: Socket[addr=/127.0.0.1,port=58184,localport=9876]
Received packet 1
Acknowledging packet 1
Received packet 2
Acknowledging packet 2
Received packet 3
Acknowledging packet 3
Received packet 4
Acknowledging packet 4
Received packet 5
Acknowledging packet 5

Received packet 6
Acknowledging packet 6
Received packet 7
Acknowledging packet 7
All packets have been received successfully
connection to 1 closed.