Go-Back-N Protocol

Introduction

The Go-Back-N Protocol using a sliding window protocol to send multiple frames of data to a receiver. The window size (or N) determines how many frames are sent at a time. If an error occurs, then an incorrect acknowledgement is sent back, and the sender resends all the packets in the window again. If no error occurs, then the windows slides to the next N packets. The Go-Back-N protocol helps to ensure that packets are not lost but is a more efficient connection than the Stop & Wait protocol.

Existing Material

The main classes which served as foundation for this implementation are the `Node` class with `Server` and `Client` subclasses, `PacketContent` and `StringContent`. These classes implement the simplest protocol of sending packets. The sender sends packets and the receiver receives all the data sent. There is nothing to account for lost packets are packets being received in the wrong order. The `PacketContent` and `StringContent` classes allow for easy extraction of data from packets. This material was previously adapted to implement the Stop & Wait Protocol, and so will act as the base for implementing the Go-Back-N Protocol.

High Level Description

Go-Back-N involves:

1. The client sending N packets from the sliding window.
2. The server sending acknowledgements for the packets received.
3. The client processing acknowledgements to determine if any packets were lost.
4. The 'sliding' of the window.

The program runs in a sequential and logical manner, calling various classes at certain times. The main classes edited and implemented throughout the program, are as follows:

Client.java – `start()`:

This method is called once and runs until all of the data has been sent. This method splits the information into segments by converting it into an array of `Strings`, where each character of data is an element in the array. The client sends the packets to the server in groups of N items. The client attaches the frame number to the end of the packet, after the data. This makes processing the packet's metadata easier to process. It then awaits a response.

Pseudo code:

```
void start(){
      inputString();
      splitString();
      while (packetsLeft){
            for (slidingWindowSize){
                  sendPacketAndFrame();
            }
            awaitResponse();
      }
}
```

Server.java – `onReceipt()`:

This method is called every time the server receives a packet. The server processes the packet. It prints out the data in the packet and extracts the frame number. The server sends back an acknowledgement that it received the packet. If an error has occurred in sending the packets, the client will become aware of this by the acknowledgements it receives.

Pseudo code:

```
void onReceipt(){
      processPacket();
      outputData();
      extractAck();
      sendResponse();
}
```

Client.java – `onReceipt()`:

This method is called every time the client receives a packet. The purpose of this method is to ensure that no packets were lost during the transmission. The client extracts the acknowledgement number sent by the server and keeps track of the number of the packet it previously processed. It compares these numbers to check if an error has occurred. It can then determine if the sliding window should move or not.

Pseudo code:

```
void onReceipt(){
      processPacket();
      compareAck();
      if (correctAck){
            moveWindow();
      } else {
            resend();
      }
}
```

The process then repeats itself, starting from inside the loop in the client's `start()` method.

Implementation Details

Client.java

To begin, the input is split into smaller segments. This is done using the `split()` method to turn it into an array of `Strings`. The program then enters two loops, the top one is governed by a `boolean` that will switch when the client receives the correct acknowledgement for the final packet. The second loop is governed by the *`WINDOW_SIZE`* or N. Packets are formed with an element of the `String` array representing the data to be sent, followed by a frame number, in the range of 0 to N. Placing the frame number at the end of all packets, including those sent by the server, make the data easier to process. The client sends packets to the server in groups of size N, then awaits a response for all of these packets. Figure 1 below show the code for `start()`.

When the client receives a packet from the server, `onReceipt()` is called. It processes the data by using the `StringContent` class, removes any unnecessary

white-space using the `trim()` function and finally uses `split()` to separate the data, making it easier to extract. The acknowledgement is easily found in the last element of the array. Now the client can check whether or not the acknowledgement is correct. This is done by comparing the current acknowledgement with the acknowledgement of the packet it previously received. If the current acknowledgement is correct then the client knows to move the sliding window (represented by the integer `window`). If not, then the client moves the start of the window to the last packet that was sent correctly. Finally, if the value of `window` surpasses the length of the `String` array (i.e. the sliding window moves past the end of the data), then the client knows that all data has been sent and the program finishes. Figure 2 below shows the code for the client's `onReceipt()`.

```
public synchronized void start() throws Exception {
    try{
        byte[] data = null;
        DatagramPacket packet = null;
        String dataString = (terminal.readString("String to send: "));
        String[] splitData = dataString.split("");
        length = splitData.length;
        while (!finished) {
            for (int i = 0; i < WINDOW_SIZE; i++){
                data = (splitData[window + i] + i).getBytes();
                terminal.println("Frame: " + i);
                terminal.println("Sending packet...");
                packet = new DatagramPacket(data, data.length, dstAddress);
                socket.send(packet);
            }
            terminal.println("Packet sent");
            socket.setSoTimeout(1000);
            this.wait(TIMER);
        }
    } catch(Exception e){

    }
}
```

Figure 1. `start()` from the Client class.

```
public synchronized void onReceipt(DatagramPacket packet){
    StringContent content= new StringContent(packet);
    String clientOutput = content.toString().trim();
    String[] array = clientOutput.split("");
    ackReceived = Integer.parseInt(array[array.length - 1]);
    if (ackReceived == prev+1 || (ackReceived == 0 && prev == WINDOW_SIZE-1)){
        window++;
        prev = ackReceived;
    } else {
        window = prev;
    }
    if (window >= length) finished = true;
}
```

Figure 2. `onReceipt()` from the Client class.

Server.java

Once the server begins receiving packets, it begins processing them in the `onReceipt()` method. It separates the data using an identical method in the client's

`onReceipt()`. It responds to the client by sending an acknowledgement number that is equal to the frame number of the packet it has just received. The server checks the acknowledgement for two specific cases: if the packets have been received in the incorrect order, or if packets have been sent after the sliding window has changed position. Both these scenarios are dealt with by returning the number of the previous number it received to 0, indicating a new sliding window position or an error. The server outputs the data onto the terminal and sends a packet with an acknowledgement back to the client. Figure 3 below shows the code for the server's `onReceipt()`.

```
public synchronized void onReceipt(DatagramPacket packet) {
    try {
        wait(100);
        StringContent data = new StringContent(packet);
        String dataString = data.toString().trim();
        String[] array = dataString.split("");
        ackNumber = Integer.parseInt(array[array.length - 1]);
        if (ackNumber != prev + 1) {
            prev = 0;
        } else {
            prev = ackNumber;
        }
        terminal.println(array[0] + ", Ack Number: " + ackNumber);
        DatagramPacket response;
        response = (new StringContent("" + ackNumber))
                .toDatagramPacket();
        response.setSocketAddress(packet.getSocketAddress());
        socket.send(response);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Figure 3. `onReceipt()` from the Server class.

Conclusion

This implementation of the Go-Back-N protocol works successfully, however, I would have liked to develop the error-handling more if I had more time. For example, implement an `onTimeOut()` function to better deal with timeouts. I have identified the following advantages and disadvantages of my implementation:

Advantages

- Good user interface.
- Fast transmission of data.
- Works for multiple values of N.
- Little chance of data being lost or mixed up.

Disadvantages

- Does not deal with timeouts properly.
- Implementation with only text files in mind.
- Some high maintenance code. Many if and else statements are used.