Stop & Wait Protocol

Introduction

The Stop & Wait protocol is used to help ensure that information being transmitted is lost due to dropped packets or packets received in the incorrect order. It involves sending one frame at a time from the sender to the receiver. Once the receiver receives a good frame, is sends an acknowledgement to the sender, indicating that it is ready to receive the next frame. However, if the sender does not receive this acknowledgement before time, or timeout, then the sender resends the packet and waits for the correct acknowledgement. Essentially, Stop and Wait is the same as the Go-Back-N protocol, where N is equal to 1.

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.

High Level Description

Stop and Wait involves:

1. The client sending information in frames.
2. The server processes the frame and returns an acknowledgement.
3. The client either sends the next frame or resends the frame.

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 splits the information into chunk packets and sends each chunk to the server with a frame number (either 1 or 0). It is supervised by the integer `packetNum`.

Pseudo code:

```
void start(){
      inputStringToSend();
      splitInputString();
      for (each chunk){
            sendChunkAndFrame();
            awaitResponse();
      }
}
```

The above pseudo-code gives an overview of how the start function works. It is called once and loops until all the chunk packets are sent.

Server.java – `onReceipt():`

This method is called every time the server receives a packet. The server processes the packet and extracts the frame number. It uses this to determine which acknowledgement number it should send back.
Pseudo code:

```
void onReceipt(){
      processPacket();
      getAcknowledgement();
      sendResponse();
}
```

Client.java – `onReceipt():`

This method is called every time the client receives a response. The client decides whether to send the next chunk or resend the chunk. This is governed by the integer `packetNum`.
Pseudo Code:

```
void onReceipt(){
      processIncomingPacket();
      if (correctAck){
            advance();
      } else {
            resend();
      }
}
```

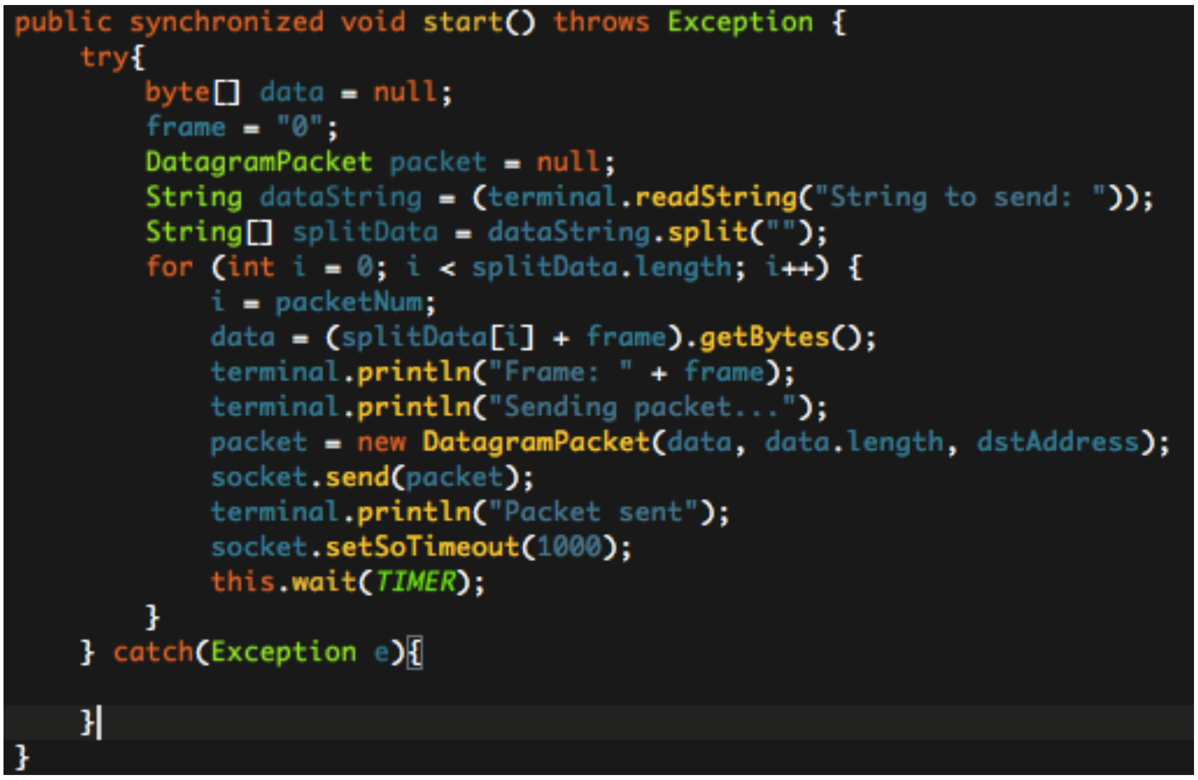
The process then repeats, starting from inside the loop in the start method of the client class.

Implementation Details

Client.java

To begin, the information to be sent has to be divided into smaller, easier-to-deal-with parts. To do this, the `split()` function is called on the `String` of data, to turn it into an array of `Strings`, where each character in the `String` is an element in the array. Then the client enters the loop, sending each character with the frame number (in this case either 0 or 1). To make the packet easier to process, the frame number is placed after all the data. Figure 1 below shows the code for `start()`.

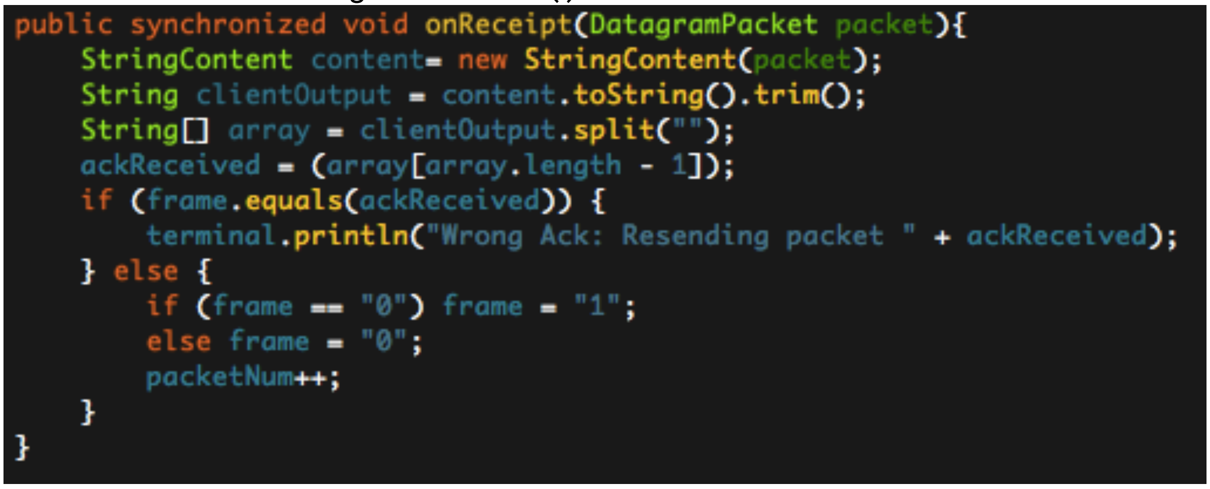
When the client receives a packet, `onReceipt()` is called. Unlike `start()`, this function is called for every packet it receives and so does not have a loop. The packet that is received is processed. The acknowledgement number is extracted from the end of the packet. This is done by converting the content of the packet into a `String`, removing all whitespace and extracting the final character. Once this is done, the acknowledgement is compared to the frame number of the next packet to be sent to the server. If the numbers match, then an error has occurred and the packet is resent. Otherwise, the client sends the next packet. The integer `packetNum` governs which packet to send. If no error occurs, `packetNum` is increased by 1, indicating that the next packet should be sent. Figure 2 below shows the code for the client's `onReceipt()`.



```java
public synchronized void start() throws Exception {
    try{
        byte[] data = null;
        frame = "0";
        DatagramPacket packet = null;
        String dataString = (terminal.readString("String to send: "));
        String[] splitData = dataString.split("");
        for (int i = 0; i < splitData.length; i++) {
            i = packetNum;
            data = (splitData[i] + frame).getBytes();
            terminal.println("Frame: " + frame);
            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.

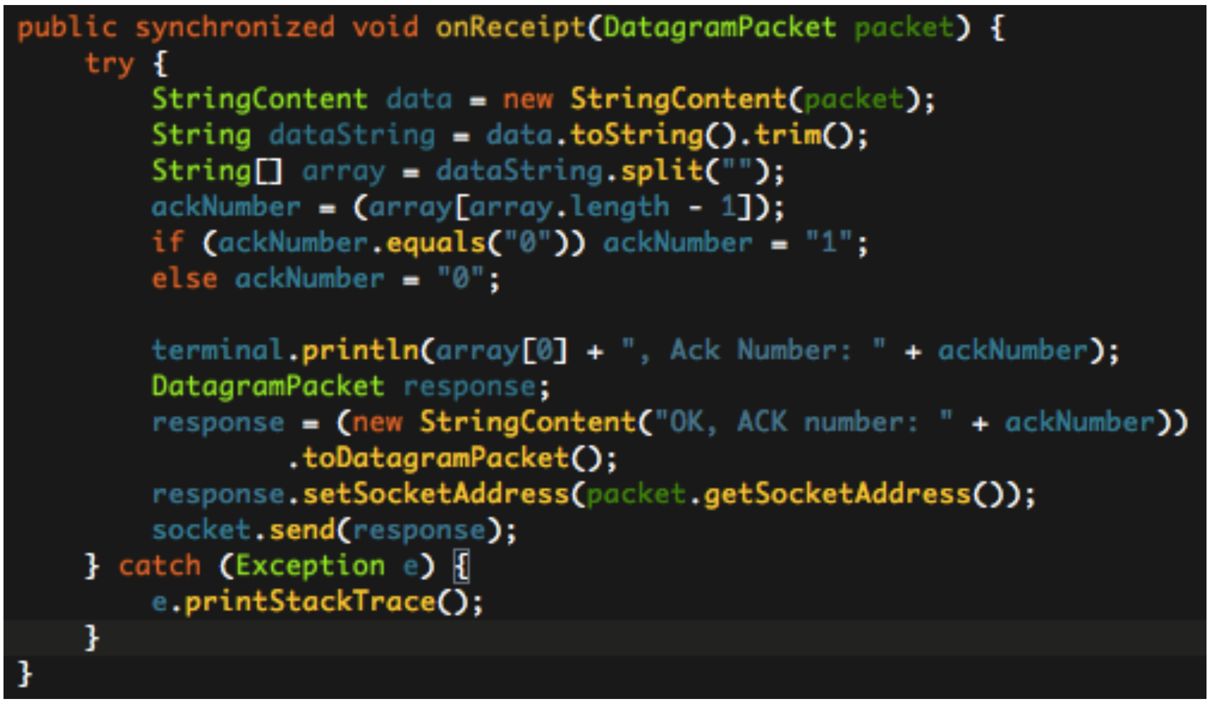


```java
public synchronized void onReceipt(DatagramPacket packet){
    StringContent content= new StringContent(packet);
    String clientOutput = content.toString().trim();
    String[] array = clientOutput.split("");
    ackReceived = (array[array.length - 1]);
    if (frame.equals(ackReceived)) {
        terminal.println("Wrong Ack: Resending packet " + ackReceived);
    } else {
        if (frame == "0") frame = "1";
        else frame = "0";
        packetNum++;
    }
}
```

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

Server.java

Once the server receives the packet sent by the client, the server's `onReceipt()` method is called. First, the packet's data is processed. The data is converted into a `String` array, using a similar method to the client's `start()` method. The server outputs the data onto the terminal and sends back a response with an acknowledgement that is equal to the frame of the packet it expects next. Similar to the client, the acknowledgement number is placed after the data. Creating this standard makes it easier to process packets from both the client and server. Figure 3 below shows the code for the `onReceipt()` method in the Server class.



```java
public synchronized void onReceipt(DatagramPacket packet) {
    try {
        StringContent data = new StringContent(packet);
        String dataString = data.toString().trim();
        String[] array = dataString.split("");
        ackNumber = (array[array.length - 1]);
        if (ackNumber.equals("0")) ackNumber = "1";
        else ackNumber = "0";

        terminal.println(array[0] + ", Ack Number: " + ackNumber);
        DatagramPacket response;
        response = (new StringContent("OK, ACK number: " + 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 Stop and Wait protocol woks successfully, however, there is room to develop the program further. For example, more could be done should a timeout occur. I have identified the following advantages and disadvantages of my implementation:

Advantages:

- Good user interface.
- Fast transmission of small quantities of data.
- The flow of the implementation ensures the correct sequence of packets.
- There is little chance of a frame being lost.

Disadvantages:

- Does not deal with timeouts properly.
- Inefficient with large quantities of data.
- Implemented only for text files.