

Telecommunications II - Flow Control Assignment Report

Section 1: Stop & Wait

As provided by the sample code, the client and server operate by running two concurrent threads: the listener in the parent Node class which runs as infinite loop waiting to receive packets from the instance's socket, and the main method of the client or server subclass.

When the client calls the start method from the main thread, a connection is made to the server and sending can begin. The start method handles all aspects of sending from that point, and automatically packaging and sending of data to the server. In this implementation, the start method uses a Stop-and-Wait protocol, which I describe below.

The client has the capacity to send any data type, so long as the user provides a *toByteArray* method to convert the data type into an array of bytes. This is ensured by the protocol because it does not assume that bytes sent or received represent any particular type: the client simply sends the array of bytes it is given, and the server receives and stores these bytes. Users are free to interpret the data, and for simplicity in this report, the data being sent are interpreted as strings.

```

*/
public synchronized void start() throws Exception
{
    //GET DATA TO SEND
    DatagramPacket packet= null;

    byte[] datum = stringToByteArray(DEFAULT_MESSAGE); //datum
    terminal.println("Message to send: ");
    printByteArray(datum);

    //SPLIT DATUM INTO FRAMES (OF SIZE DEFAULT_FRAME_SIZE)
    //FOR EACH FRAME, SEND PACKET TO SERVER
    //IF TIMEOUT OR UNEXPECTED ACK, RESEND PACKET
    terminal.println("Sending packet...");
    startedSending = true;
    //sendMonoByteFrames(datum, packet);
    sendMultiByteFrames(datum, packet, DEFAULT_FRAME_SIZE);
    finishedSending = true;
}

```

Client sending mechanism (ID + DATA)

The data sent in a DatagramPacket is an array of bytes. The client method *sendMultiByteFrames(data, packet, frameSize)* sends the data in frames of size *frameSize*. So if frameSize is set to 1 kilobyte, the data is split into frames of 1000 bytes, and each datagram packet sends one frame.

The input array of bytes is divided into a 2-D array of frames (byte arrays) using the *splitData* method. This method returns a 2-D array holding all the frames as arrays.

```

public byte[][] splitData(byte[] datum, int frameSize)
{
    byte[][] frames = new byte[datum.length/frameSize+1][frameSize];
    for(int index = 0; index < (int)(datum.length/frameSize) + 1; index++)
    {
        for(int j = 0; j < frameSize & (index*frameSize + j) < datum.length ; j++)
        {
            frames[index][j] = datum[index*frameSize + j];
        }
    }
    return frames;
}

```

Note that each frame is created with a spare final element. When creating the datagram packet before sending, each frame is also coupled with a single extra byte called the ID byte. The values of IDs sent alternate between 1 and 0 for each packet sent, allowing the receiving application to verify that the packet it received is the one which comes next in the sequence, and not a repeat.

```
376 //PREPARE BYTE ARRAY TO SEND AS PACKET,  
377 //COMBINE frames[i] ELEMENTS WITH ID BYTE  
378 byte[] frame = new byte[frames[i].length + 1]; //frame = {elems of frames[i] + currentID};  
379 //COPY ELEMENTS OF FRAMES[i] TO FRAME  
380 for(int index = 0; index < frame.length - 1 ; index++){ frame[index] = frames[i][index]; }  
381 frame[frame.length - 1] = currentID;  
382  
383 packet = new DatagramPacket(frame, frame.length, dstAddress);
```

Server receive mechanism

The server runs an infinite loop in the listener thread, waiting to receive a packet from the socket. Once a packet is received, the listener thread calls the onReceipt method, which processes the packet.

The program checks the ID byte to verify that the packet is the next one it needs in the sequence. It does this by comparing the received ID byte to another byte variable in the server object, nextID, which holds the value of the expected ID of the next packet.

If the ID bytes are equal, then the packet is as required and the program enqueues the message to its queue (implemented by an ArrayList). The server's nextID byte is updated.

If the ID bytes are different, then the packet is not the required one, and the program discards it, leaving the nextID unchanged until the correct packet is received.

Server Acknowledge mechanism

When the server receives a packet, after processing the ID, it replies with an acknowledgement packet. This is a fundamental property of the Stop-and-Wait protocol: it is this acknowledgement which allows the client to send its next packet. The client cannot move on to the next packet until it has received an acknowledgement that the server has received the current packet.

The acknowledgement is sent as a datagram packet containing just 4 bytes: the characters "ACK" followed by the server's nextID byte (the ID it expects next).

Timeout mechanism

Both client and server implement the `onTimeout` function, which is called when a timeout occurs (ie: a `SocketTimeoutException` is thrown and caught within Node Listener Thread).

For the client, this means resending the last Datagram packet, instead of moving on to the next.

For the server, this means resending the acknowledgement for the most recently received packet.

The timeout mechanism allows the client and server to resend a packet if they have not received a packet after a certain length of time. This prevents the applications from stalling indefinitely if some asynchronous behaviour occurs (ie: if server doesn't receive a packet, it won't send an ACK, then client could wait forever before sending the next packet. Timeout avoids this scenario).

I chose to implement this mechanism by taking advantage of the `SocketTimeoutException` thrown by `DatagramSocket` if it has not received a packet after a time `SO_TIMEOUT` defined by the user. If this exception is caught, the instance's `onTimeout` method is called. On Timeout is an abstract method declared in the parent `Node` class, left to be implemented in children. This allows `onTimeout` to be called by the listener thread, while performing the functions required by a given child (eg: `onTimeout` for client resends the packet, `onTimeout` for server resends the acknowledgement).

```
try{
    synchronized(this)
    {
        socket.receive(packet);
    }
    onReceipt(packet);
} catch (java.net.SocketTimeoutException timeout)
{
    System.out.println("Timeout caught in node: "+timeout.getMessage());
    onTimeout();
}
```

Modification of Node Listener

In my debugging, I simulated lost ACKs from the server by sending it into an infinite loop. When I tried this the Socket Exception would be thrown.

After some searching, I discovered that this was happening because the method was attempting to modify the socket parameters while the socket was already being accessed by another thread (ie: the listener thread was calling socket.receive).

When client calls setTimeoutOn, timeout must be off. If the timeout setting is off, that means that onReceipt blocks until a packet is received. And when no ACKs are being received, this state continues indefinitely. So in this case, the socket would throw an exception when setTimeoutOn tried to modify it.

To solve this, I surrounded both the socket.receive line and the setSoTimeout line with synchronized blocks. This prevents one thread from accessing a block while the other is being accessed by another thread.

Secondly, I created two conditions for the listener loop: one for if timeout is on, and one for when timeout is off. If timeout is off, the socket will still intermittently leave the receive method (every 0.5 seconds). This allows the setTimeoutOn method to act if it is waiting to.

```
52         while(true) {
53             DatagramPacket packet = new DatagramPacket(new byte[PACKETSIZE], PACKETSIZE);
54             //if timeout is on, exit socket.receive() intermittently to allow socket timeout parameter
55             if(!timeoutOn)
56             {
57                 try{
58                     synchronized(this)
59                     {
60                         socket.setSoTimeout(500);
61                         socket.receive(packet);
62                         socket.setSoTimeout(0);
63                     }
64                     onReceipt(packet);
65                 }catch(java.net.SocketTimeoutException t)
66                 {
67                     socket.setSoTimeout(0);
68                 }
69             }
70             else{
71                 try{
72                     synchronized(this)
73                     {
74                         socket.receive(packet);
75                     }
76                     onReceipt(packet);
77                 }catch (java.net.SocketTimeoutException timeout)
78                 {
79                     System.out.println("Timeout caught in node: "+timeout.getMessage());
80                     onTimeout();
81                 }
82             }
83         }
84     }
85 }
86 }
```

Client Wait mechanism

The client waits for an acknowledgement or timeout by calling this.wait() within a synchronized statement. This allows the execution of the main thread to block until the listener thread notifies it. This notification occurs either within the onReceipt method, or the onTimeout method. If onReceipt is called, the method checks the packet ID, and updates the loop-status Boolean variables.

These are a collection of global Boolean variables which are used for communication between threads. For example, if a packet is received, the variable received is set to true. If a timeout occurs, timedout is set to true. These variables allow the client to know if it needs to resend a packet or not.

```
//RESET RECEIPT STATUS VARIABLES
timedOut = false;
resend = false;
expectedACK = false;
unexpectedACK = false;
```

```
//WAIT FOR ACK
//WAIT STOPS WHEN
// -onReceipt() METHOD NOTIFIES THAT IT IS COMPLETED
// or
// -timeout Occurs (to be implemented)
waiting = true;
setTimeoutOn();
terminal.println("\nMain thread: waiting for acknowledgement");
synchronized(this)
{
    this.wait();
}
setTimeoutOff();
waiting = false;
terminal.println("\nMain thread: Wait Over");
```

Client Resend mechanism

When the client stops waiting, this means one of two things has happened: onTimeout was called, or onReceipt was called. The main client thread proceeds to enter a series of conditional statements which use the loop-status booleans to decide whether the packet is to be resent.

```
//WHILE LOOP PROVIDES A RESEND MECHANISM
terminal.println("\nMain thread: while loop ");
//positiveACK and negativeACK can only be modified by the onReceipt() method.
//if a positive ACK is received before timeout, while loop terminates and for loop moves on to
//if a negative ACK is received before timeout, if condition is true and packet is resent
//if timeout occurs, if condition is true and packet is resent
while(!expectedACK & !resend) //while there is no response and no timeout, wait for response
{
    //IF TIMEOUT, RESEND PACKET
    if(unexpectedACK | timedOut)
    {
        resend = true;
        i--;
    }
}
//IF NO TIMEOUT AND POSITIVEACK, SEND NEXT PACKET
```

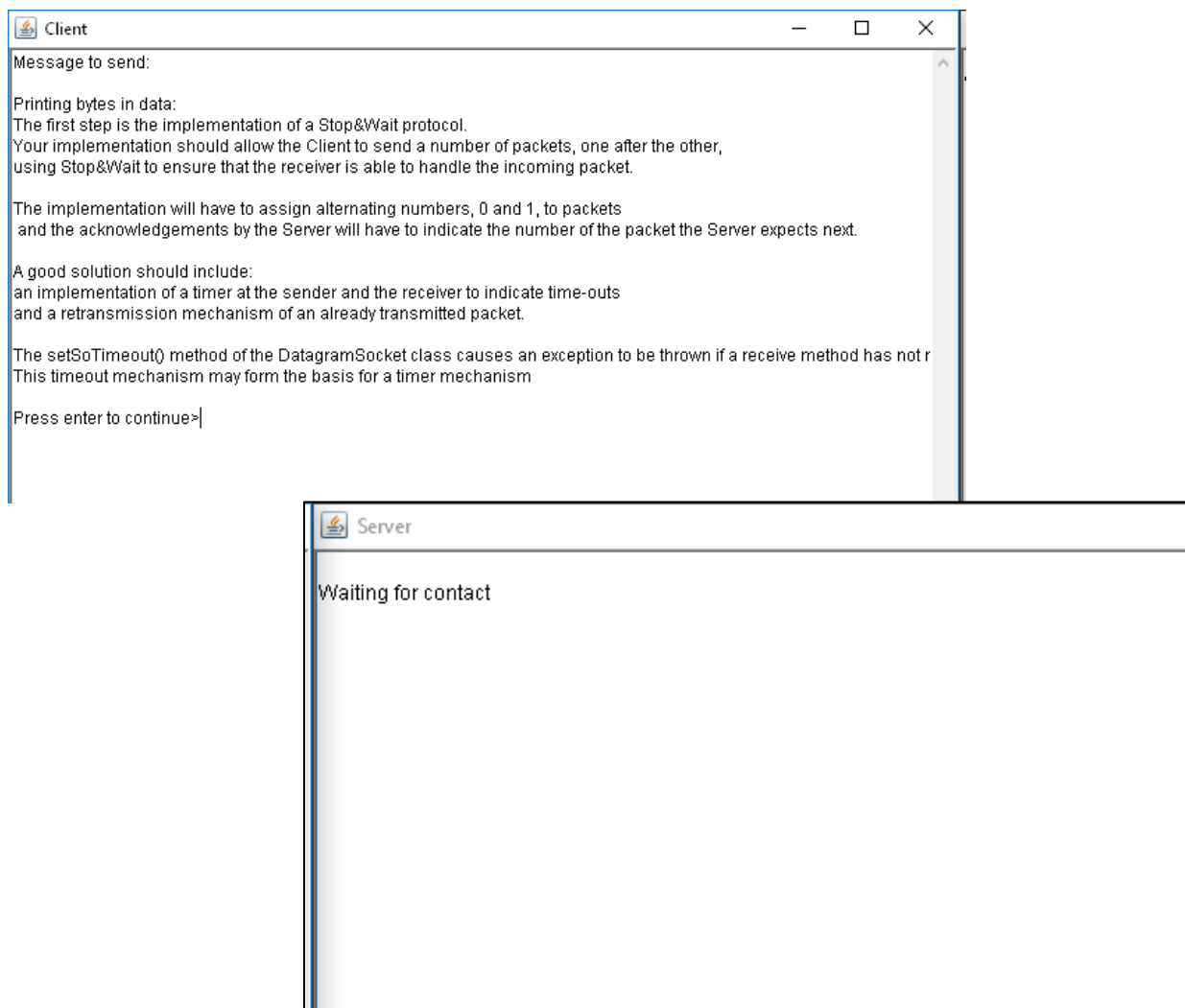
Server resend mechanism & Termination

The server onTimeout mechanism fully handles its resend mechanism, by calling the resend method.

The final packet sent by the client is always the “termination” packet. It is a single byte, with the value -1. My Stop&Wait protocol dictates that, when the server receives this single-byte packet containing -1, this indicates that the last packet has been sent, and the server exits the start method.

Examples of Normal Operation & Snapshots of Packet Details

Before sending, Client prints the string it is to send, then waits for user to commence sending.



Client begins sending packets. Debugging information is printed to the terminals, including the loop cycle, if a timeout occurred, and the contents of the packets being sent and received. Below, the terminals show a pause in the execution when the server attempts to cause a timeout in client by purposely failing to sending an acknowledgement for the third packet.

Client

```

resend==false
expectedACK==true

*****
For loop cycle:2
Sent Packet with string content : the sender and the receiver to indicate time-outs
and a retransmission mechanism of an already transmitted packet.

The setSoTimeout() method of the DatagramSocket class causes an exception to be thrown if a receive method has not r
entered setSoTimeoutOn:

socket.isClosed():false
printed before setSoTimeout
printed after setSoTimeout
socket.getSoTimeout()==1000

Main thread: waiting for acknowledgement

onReceipt(): received packet with string content: ACK Received expected
onReceipt finished,
notifying...

onReceipt: notified main thread

Main thread: Wait Over

Main thread: while loop


Main thread: exited while loop
resend==false
expectedACK==true

*****
For loop cycle:3
Sent Packet with string content : a given time.
This timeout mechanism may form the basis for a timer mechanism

entered setSoTimeoutOn:

socket.isClosed():false
printed before setSoTimeout

```

 Search Windows

Server

```

data received so far:
The first step is the implementation of a Stop&Wait protocol.
Your implementation should allow the Client to send a number of pac
using Stop&Wait to ensure that the receiver is able to handle the incoi

The implementation will have to assign alternating numbers, 0 and 1,
and the acknowledgements by the Server will have to indicate the nur

A good solution should include:
an implementation of a timer at

Received packet
the sender and the receiver to indicate time-outs
and a retransmission mechanism of an already transmitted packet.

The setSoTimeout() method of the DatagramSocket class causes an
onReceipt(): packet.getLength() ==251
onReceipt(): packet.getData()[0] ==32
onReceipt(): packet.getData()[packet.getLength()] ==0
onReceipt(): getID(packet) ==0
onReceipt(): nextID ==0
onReceipt(): nextID ==1

sending ACK1

Finished responding to receipt
*****

data received so far:
The first step is the implementation of a Stop&Wait protocol.
Your implementation should allow the Client to send a number of pac
using Stop&Wait to ensure that the receiver is able to handle the incoi

The implementation will have to assign alternating numbers, 0 and 1,
and the acknowledgements by the Server will have to indicate the nur

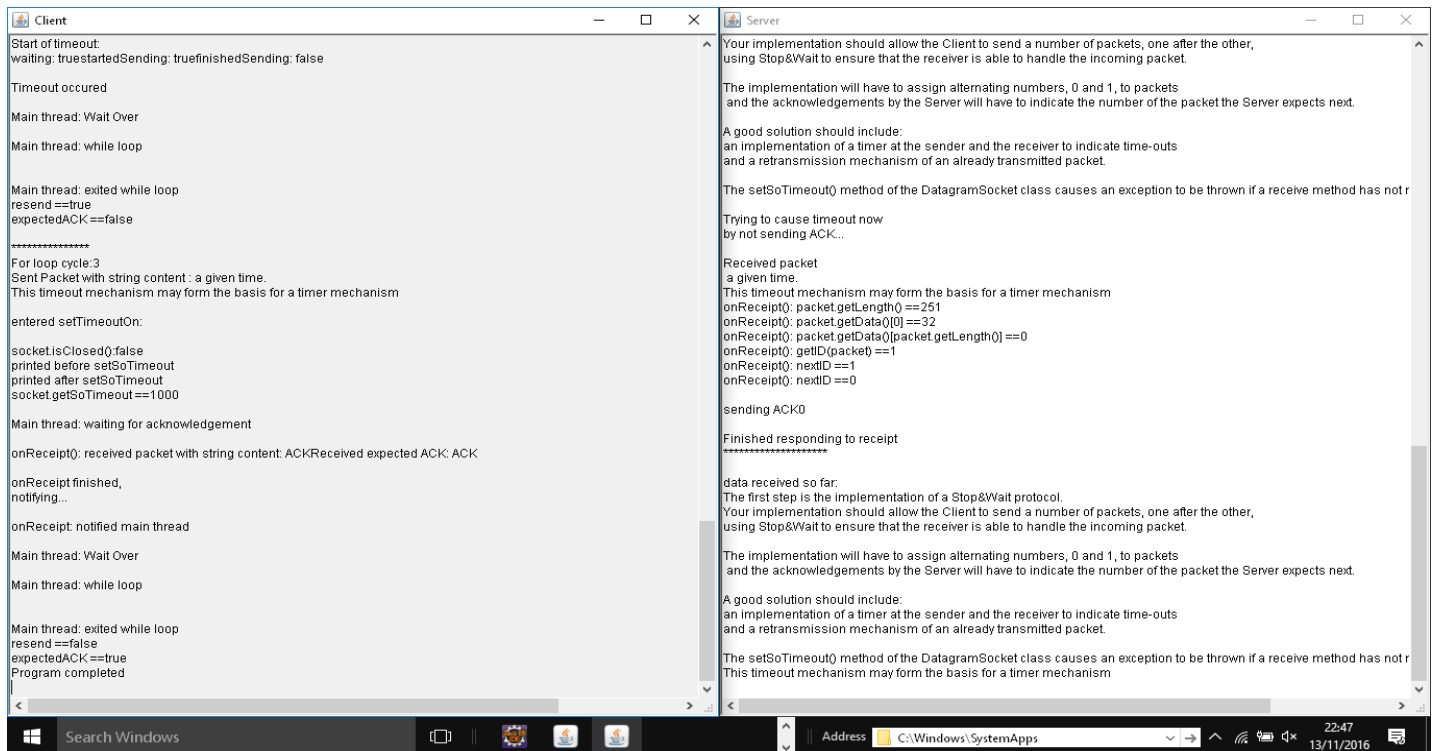
A good solution should include:
an implementation of a timer at the sender and the receiver to indicat
and a retransmission mechanism of an already transmitted packet.

The setSoTimeout() method of the DatagramSocket class causes an

Trying to cause timeout now
by not sending ACK...

```

Finally the terminals indicate that the sending has terminated. The server prints the full message, which has been stored in an ArrayList.



```
Client
Start of timeout:
waiting: truestartedSending: truefinishedSending: false

Timeout occurred

Main thread: Wait Over

Main thread: while loop

Main thread: exited while loop
resend ==true
expectedACK ==false

*****
For loop cycle:3
Sent Packet with string content : a given time.
This timeout mechanism may form the basis for a timer mechanism

entered setTimeoutOn:
socket.isClosed():false
printed before setTimeout
printed after setTimeout
socket.getSoTimeout ==1000

Main thread: waiting for acknowledgement

onReceipt(): received packet with string content: ACKReceived expected ACK: ACK

onReceipt finished,
notifying...

onReceipt: notified main thread

Main thread: Wait Over

Main thread: while loop

Main thread: exited while loop
resend ==false
expectedACK ==true
Program completed

Server
Your implementation should allow the Client to send a number of packets, one after the other,
using Stop&Wait to ensure that the receiver is able to handle the incoming packet.

The implementation will have to assign alternating numbers, 0 and 1, to packets
and the acknowledgements by the Server will have to indicate the number of the packet the Server expects next.

A good solution should include:
an implementation of a timer at the sender and the receiver to indicate time-outs
and a retransmission mechanism of an already transmitted packet.

The setTimeout() method of the DatagramSocket class causes an exception to be thrown if a receive method has not r

Trying to cause timeout now
by not sending ACK...

Received packet
a given time.
This timeout mechanism may form the basis for a timer mechanism
onReceipt(): packet.getLength() ==251
onReceipt(): packet.getData()[0] ==32
onReceipt(): packet.getData()[packet.getLength()] ==0
onReceipt(): getID(packet) ==1
onReceipt(): nextID ==1
onReceipt(): nextID ==0

sending ACK0

Finished responding to receipt
*****

data received so far:
The first step is the implementation of a Stop&Wait protocol.
Your implementation should allow the Client to send a number of packets, one after the other,
using Stop&Wait to ensure that the receiver is able to handle the incoming packet.

The implementation will have to assign alternating numbers, 0 and 1, to packets
and the acknowledgements by the Server will have to indicate the number of the packet the Server expects next.

A good solution should include:
an implementation of a timer at the sender and the receiver to indicate time-outs
and a retransmission mechanism of an already transmitted packet.

The setTimeout() method of the DatagramSocket class causes an exception to be thrown if a receive method has not r
This timeout mechanism may form the basis for a timer mechanism
```


Section 2: Sliding Window: Go Back N

In this report I describe the protocol I developed to implement the Sliding window flow control mechanism. I decided to implement the “Go Back N” version.

The following is the pseudo-code algorithm I developed to send the frames using a sliding window protocol:

```
public void sendFramesInSlidingWindow(byte[][] frames)
{
    int[] windowACKs = new int[WINDOW_SIZE];
    int[] windowSentFrames = new int[WINDOW_SIZE];
    int[] windowTimeouts = new int[WINDOW_SIZE];
    initialiseToZero(windowACKs);
    initialiseToZero(windowSentFrames);
    initialiseToZero(windowTimeouts);

    int headIndex = 0; //absolute index of the current start of the frame (ie: the index of that frame in frames)
    int tailIndex = WINDOW_SIZE - 1;

    boolean allFramesAcknowledged = false;
    while(!allFramesAcknowledged)
    {
        while(!frameAcknowledged(headIndex))
        {
            for(int frameIndex = headIndex; frameIndex <= tailIndex ; frameIndex++) //for frame in window
            {
                if(!frameSent(frameIndex) | (!frameAcknowledged(frameIndex) & frameTimedOut(frameIndex)) )
                {
                    sendFrame(frameIndex);
                    setTimeoutOn(frameIndex);
                    resetTimeoutBit(frameIndex);
                    setFrameSentBit(frameIndex);
                }
            }

            slideWindow(windowACKs, windowSentFrames, windowTimeouts);
            headIndex++;
            if(tailIndex >= frames.length){ tailIndex = frames.length-1; }
            else{tailIndex++;}
            if(frameAcknowledged(headIndex) & headIndex == frames.length-1)
            {
                allFramesAcknowledged = true;
            }
        }
    }
}
```

The boundaries of the sliding window are defined by the integer variables headIndex and tailIndex.

The three arrays declared at the top hold information about the frames currently within the scope of the window. They hold bits, which are set or reset depending on the state of the frame that element represents.

For example, each element in windowACKs is 1 if the frame with the same index in the window has been acknowledged, and 0 if the frame hasn't been acknowledged.

These arrays are used by the main for loop to determine what action to take for each frame in the window, as described below.

The elements of the arrays are initially 0. The program enters the main while loop, and will stay within this loop until all frames have been sent. A nested loop is then entered for every time the

window slides by one place (ie: this while loop is only exited when the head frame of the window is acknowledged).

The program provides a method for each array which, when given a frame index (ie: the index of an element in the input 2D byte array “frames”), accesses the data in the arrays and returns the status of that frame. For example, `frameAcknowledged(int frameIndex)` returns 1 if the frame is acknowledged, and 0 if unacknowledged. How this method works is described in the next section.

While the head frame of the window is unacknowledged, the program iterates over each frame in the window, sending the frame if:

A) the frame is unsent. This is determined by the program checking the relevant bit in the `windowSent` array, using the `frameSent` method, which is explained below.

B) the frame is unacknowledged and the frame timed out. Again, these values are determined by checking the data in the `windowACKs` and `windowTimeouts` arrays.

The frames are sent using the `sendFrame` method, which sends a datagram packet using the `DatagramPacket` and `DatagramSocket` libraries.

This for loop is repeatedly entered until the first frame in the current window (the frame with index `headIndex`) is acknowledged.

The lines outside of the containing while loop perform functions which effectively slide the window:

The values in the data arrays are shifted one to the left by the `slideWindow` method, so that they continue to represent the frames in the window as it slides.

The head index and tail index are incremented (unless they are equal to the length of the frames array), effectively “sliding” the window. This is performed by the `slideWindow` method below:

```
public void slideWindow(int[] windowACKs, int[] windowSentFrames, int[] windowTimeouts)
{
    for(int i = 0; i < WINDOW_SIZE; i++)
    {
        windowACKs[i] = windowACKs[i+1];
        windowACKs[i+1] = 0;
        windowSentFrames[i] = windowSentFrames[i+1];
        windowSentFrames[i+1] = 0;
        windowTimeouts[i] = windowTimeouts[i+1];
        windowTimeouts[i+1] = 0;
    }
}
```

Finally, unless the head index is the last element of the frames array, and the last element has been acknowledged, the program continues to loop and send the frames within the window.

With this protocol, the sender cannot progress past the tail index frame until the head index frame has been acknowledged. This allows for greater efficiency than a Stop-and-Wait protocol, yet still prevents the sender from sending faster than the receiver can receive.

Getting Data on the Frames in the Window from the Sliding Arrays

The main sending mechanism requires methods to determine whether each frame in the window has been sent, has been acknowledged and whether it has timed out.

This is provided by three methods which follow a similar format, as shown in the example below:

```
public boolean frameAcknowledged(int frameIndex, int framesLength, int headIndex, int[] windowACKs)
{
    int relativeIndex = (frameIndex % WINDOW_SIZE) - (headIndex % WINDOW_SIZE);
    if(windowACKs[relativeIndex] == 1 )
    {
        return true;
    }
    return false;
}
```

The frames are held in a single array, and the program doesn't create a new array of frames for each window. So a method is needed to map the actual index of the frame, to the index of the element in the window array which represents that frame. This is determined by the formula on the first line in the method above.

Why the equation works should be evident :

- Modulo the frame index to bring it within the window size (also the size of the arrays it needs to be mapped to).
- Subtract from this value the modulo of the head index, to negate the offset of a head index which isn't a multiple of the window size.

Once the correct element index is known, the program simply checks if the bit is set, returning false otherwise.