Network Lab Report

Shuvayan Ghosh Dastidar
001810501044
JU-BCSE - III

Submission Date: 23rd November,2020

Assignment 2:
Implement three data link layer protocols, Stop and Wait, Go Back N Sliding Window and Selective Repeat Sliding Window for flow control.

Theory

In data communications, flow control is the process of managing the rate of data transmission between two nodes to prevent a fast sender from overwhelming a slow receiver. It provides a mechanism for the receiver to control the transmission speed, so that the receiving node is not overwhelmed with data from transmitting node.

We have considered three flow control algorithms:

- Stop and Wait Protocol :
- GoBackN Protocol :
- SelectiveRepeat Protocol :

Implementation

Approach

The language used for this assignment is Java. The method chosen for simulating the protocols is multithreading.

Here are some reasons supporting the reason of choosing multithreading:

- In real world applications, all networks work simultaneously and proper synchronisation of the transmitted packets is very necessary at sending and receiving sides. Multithreading gives a great way to achieve synchronisation through various methods such as locks, semaphores and signals. Thus the main components of this assignment , the sender, the receiver and channel are all different threads.
- The channel is based on the concept of message queue. The message queue serves as a common channel between the sender and receivers. It is used for communication between sender and receiver threads.
- This method helps in proper synchronisation. Moreover Java API provides functions like DelayQueue which greatly helps in calculating time-outs for the packets.

Related methods

Some other methods in which this simulation could have been achieved are

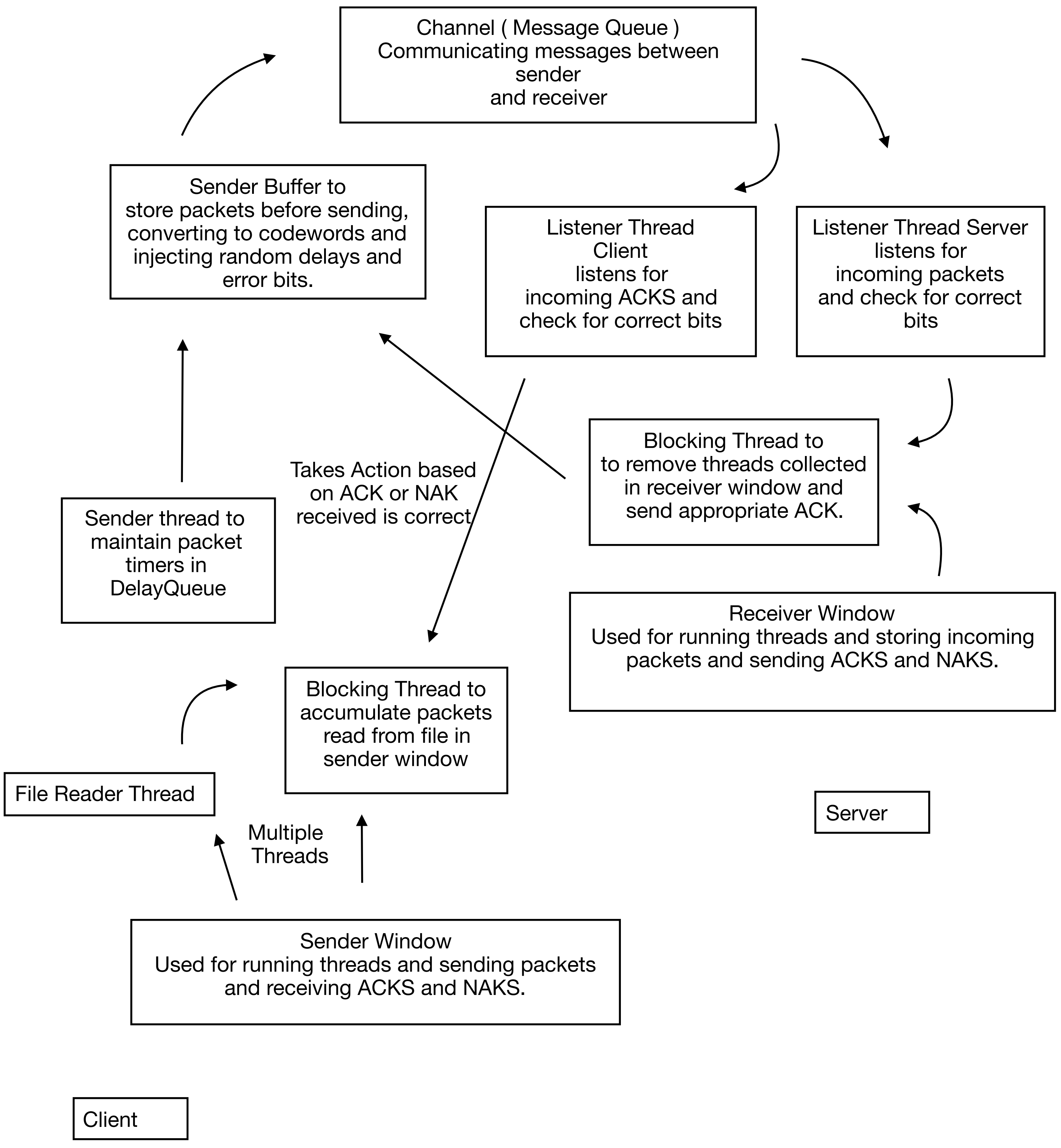
- Socket : Sockets are great for communicating between different senders and receivers. They communicate between different ports and handle all the communication through socket channels. Although sockets were a great choice, this method involving message queues greatly helps in communication of packets and all kinds of errors and inclusion of random noises are easy enough. However this method can be easily extended by a socket channel.
- File Channel : Another method that Java provides is a file channel. File channel involves a memory mapped file that can communicate between different senders and receivers. Although it sounds good, synchronising packet writes and reads in a file is difficult to obtain due to it's unavailability in program space and hence locks and other synchronization tools cannot be used for communication.

File Structure Format

```
.
├── Channel
|   └── Channel.java
├── Client
|   ├── Client.java
|   └── SenderWindow.java
├── ErrorDetection
|   └── CRC.java
├── Generate
|   ├── Gen.java
|   └── Generate.java
├── Node
|   └── Sender.java
├── Packet
|   ├── ACKPacket.java
|   ├── Packet.java
|   └── TimedPacket.java
├── Report
|   └── assign2.pages
```

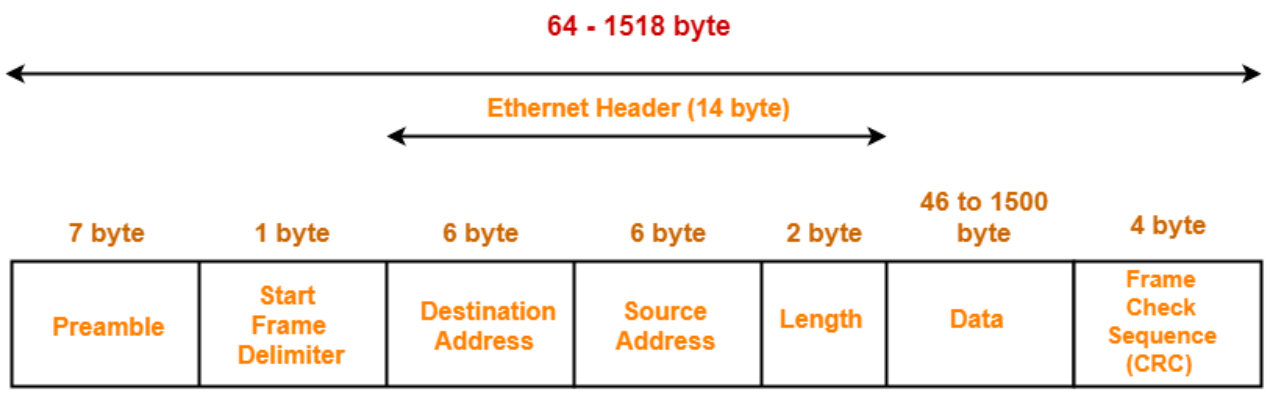
```
├── Run
│   ├── GoBackN.java
│   ├── SelectiveRepeat.java
│   └── StopNWait.java
├── Server
│   ├── ReceiverWindow.java
│   └── Server.java
└── Utils
    └── ArrayBlockingList.java
```

Schematic Diagram of WorkFlow(Heavy multithreading)


Channel (Message Queue)
Communicating messages between sender and receiver
Sender Buffer to store packets before sending, converting to codewords and injecting random delays and error bits.
Listener Thread Client listens for incoming ACKS and check for correct bits
Listener Thread Server listens for incoming packets and check for correct bits
Blocking Thread to to remove threads collected in receiver window and send appropriate ACK.
Takes Action based on ACK or NAK received is correct
Sender thread to maintain packet timers in DelayQueue
Receiver Window Used for running threads and storing incoming packets and sending ACKS and NAKS.
Blocking Thread to accumulate packets read from file in sender window
File Reader Thread
Server
Multiple Threads
Sender Window Used for running threads and sending packets and receiving ACKS and NAKS.
Client

Frame Format

The frame format used for this simulation is the IEEE 802.3 frame format.



IEEE 802.3 Ethernet Frame Format

The only change in the format is the addition of sequence number of size 1 byte which is added to the frame after the source address.

PREAMBLE - Ethernet frame starts with 7-Bytes Preamble. This is a pattern of alternative 0's and 1's which indicates starting of the frame and allow sender and receiver to establish bit synchronization
Start of frame delimiter (SFD) - This is a 1-Byte field which is always set to 10101011.
Destination Address - This is 6-Byte field which contains the MAC address of machine for which data is destined.

Source Address - This is a 6-Byte field which contains the MAC address of source machine. As Source Address is always an individual address (Unicast), the least significant bit of first byte is always 0.

Sequence Number : This is a 1-Byte field and is used to signify the bit number in the sliding window algorithms.

Length : Length is a 2-Byte field, which indicates the length of entire Ethernet frame. This 16-bit field can hold the length value between 0 to 65534, but length cannot be larger than 1500 because of some own limitations of Ethernet.

Data - This is the place where actual data is inserted, also known as Payload.

Type - This is the place where the type is stored - PACKET, NAK or ACK.

Cyclic Redundancy Check (CRC) - CRC is 4 Byte field. This field contains a 32-bits hash code of data, which is generated over the Destination Address, Source Address, Length, and Data field.

Explanation of code segments

First ...

Let’s discuss about delay queue in java and how it is used in making packet timers.

DelayQueue is a specialised Priority Queue that orders elements based on their delay time. It means that only those elements can be taken from the queue whose time has expired.

It helps in querying the packet with the least delay and just increasing the delay of a certain packet can be used as starting a timer.

These are as simple as

// These are implemented methods
packet.reset();
packet.repeat();

Let’s come to codes ..

Client.java: main client for all operations.

```
import Packet.Packet;
import java.util.concurrent.CountDownLatch;

public class Client {

    private final SenderWindow window;
    private String srcAddress;
    private String destAddress;
    private int windowLength;
    private int seqLength;
    private Channel ch;
    private Sender sender;
    private boolean gobackn;

    private CountDownLatch latch;
    Listener listener;

    public Client(String srcAddress, String destAddress, int windowLength, int
seqLength, Channel ch, boolean gobackn) {
        latch = new CountDownLatch(1);
        listener = new Listener();
        listener.setDaemon(true);
        listener.start();


        this.srcAddress = srcAddress;
        this.destAddress = destAddress;
        this.windowLength = windowLength;
        this.seqLength = seqLength;
        this.ch = ch;
        this.gobackn = gobackn;

        sender = new Sender(ch, false);
        window = new SenderWindow(windowLength, seqLength, srcAddress,
destAddress) {
            @Override
            public void sendPacket(Packet p) {
                sender.add(p);
            }
        };
        listener.go();
    }

    public void onReceive(Packet packet) {
        System.out.println("[INFO] ACK   : " +
String.valueOf(packet.getSeqNumber()));
        switch (packet.type) {
            case "ACK": {
                window.ack(packet.getSeqNumber()); // implement number
                System.out.println("[INFO] ACK REMOVED : " + 
String.valueOf(packet.getSeqNumber()));
            }
            case "NAK": {
                window.nak(packet.getSeqNumber());
            }

        }
    }


    class Listener extends Thread {


        public void go() {
            latch.countDown();
        }

        /*
         * Listen for incoming packets and inform receivers
         */
        @Override
        public void run() {
            try {
                latch.await();
                // Endless loop: attempt to receive packet, notify receivers, 
etc
                while (true) {
                    String content = ch.getPacketReceiver();

                    Packet p = Packet.toPacket(content);
                    System.out.println("[PACKET RECEIVED]" + p);
                    onReceive(p);
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

```
}
```

Sender.java : Used for storing packets as buffer and sending to the channel

```
public class Sender {

    private Channel ch;
    private Boolean server;
    private LinkedBlockingQueue<String> bh;
    private final ExecutorService executor;
    public Sender(Channel ch, Boolean server) {
        this.server = server;
        this.ch = ch;
        bh = new LinkedBlockingQueue<>();
        executor = Executors.newSingleThreadExecutor();
        executor.execute(() -> sender());
    }
    public void add(Packet packet) {
        String content = packet.toCodeWord();
        bh.add(content);
    }
    private void sender() {
        ReentrantLock lock = new ReentrantLock();
        Condition sendTime = lock.newCondition();

        while (true) {
            lock.lock();
            try {
                String content = bh.take();
//                sendTime.awaitUntil(nextSendTime());

                if (server) {
                    ch.addReceiver(content);
                } else {
                    ch.addSender(content);
                }
            } catch (InterruptedException e) {
                System.out.println("Sender failed");
            } finally {
                lock.unlock();
            }
        }
    }

    /*
     *   Used for sending packets with random delay
     *   ranging from 0 seconds to 3 seconds.
     */
    private Date nextSendTime() {
        int delaySeconds = (int) (Math.random() * 4);
        return new Date(System.currentTimeMillis() +
TimeUnit.SECONDS.toMillis(delaySeconds));
    }
}
```

SenderWindow : Used for communicating with the client and channel and sending and storing packets

```
public abstract class SenderWindow {

    private final ArrayBlockingList<TimedPacket> windowPackets;
    private final DelayQueue<TimedPacket> windowTimers;
    private final LinkedBlockingQueue<Packet> packets;
    private final ExecutorService pool;
    private int sequenceLength;
    private int windowLength;
    private int bufferNumber;
    private int windowStart;
    private String srcAdd;
    private String destAdd;
    private int totalCount;

    public SenderWindow(int windowLength, int sequenceLength, String srcAdd, String
destAdd) {
        this.windowLength = windowLength;
        this.sequenceLength = sequenceLength;
        this.srcAdd = srcAdd;
        this.destAdd = destAdd;
        bufferNumber = 0;
        windowStart = 0;
        totalCount = 0;

        windowPackets = new ArrayBlockingList<>(windowLength);
        windowTimers = new DelayQueue<>();
        packets = new LinkedBlockingQueue<>();
        getPacket();
        pool = Executors.newFixedThreadPool(2);
        pool.execute(() -> inputToWindow());
        pool.execute(() -> windowWorker());
//        pool.execute(() -> getPacket());

    }

    private void inputToWindow() {
        while (true) {
            try {
                windowPackets.awaitNotFull();
//                packets.awaitNotEmpty();  // Blocking

                Packet packet = packets.take();
                packet.setSeqNumber(bufferNumber);
                bufferNumber = nextNumber(bufferNumber);

                // Blocking
                TimedPacket timedPacket = new TimedPacket(packet);
                //Add packet to both of the window components:
                windowPackets.add(timedPacket);
                windowTimers.add(timedPacket);
            } catch (InterruptedException e) {
                System.out.println("Terminating feeder"); //--> DEBUG
                return;
            }
        }
    }

    public abstract void sendPacket(Packet p);
    private void windowWorker() {
        while (true) {
            try {
                // Wait for next packet to expire

                TimedPacket timedPacket = windowTimers.take(); // Blocking

                sendPacket(timedPacket.getPacket());



                // renew the delay on the packet and feed back into schedule
                windowTimers.put(timedPacket.repeat());
            } catch (InterruptedException e) {
                System.out.println("Terminating scheduleHandler"); //DEBUG
                return;
            }
        }
    }



    private void getPacket() {
        try {
            String filename = "input.txt";
            List<String> lines = Files.readAllLines(new File(filename).toPath());
            for (String line : lines.subList(0, 100)) {
                try {
//                      packets.awaitNotFull(); // Blocking

//                      System.out.println("[DEBUG] ADDING : " + line);

                    Packet p = new Packet(srcAdd, destAdd, line);
                    packets.put(p);
                } catch (InterruptedException e) {

                }

            }
        } catch (IOException e) {
            System.out.println("Cannot read input file");
        }
    }


    public boolean ack(int number) {
        boolean result;
        int relative = relativeNumber(number);
        System.out.println("[ACK] :" + String.valueOf(number) + "   " +
String.valueOf(relative));
        if (0 < relative && relative < windowLength + 1) {
            for (int i = 0; i < relative; i++) {
                TimedPacket timedPacket = windowPackets.remove();
                windowStart = nextNumber(windowStart);
                windowTimers.remove(timedPacket);
            }
            totalCount += relative;
            System.out.println("[INFO] PACKETS SUCCESSFULLY SENT AND ACK RECEIVED COUNT
:" + String.valueOf(totalCount));
            result = true;
        } else {
            result = false;
        }
        return result;
    }


    public void nak(int number) {
        int relative = relativeNumber(number);
        if (relative < windowLength) {
            TimedPacket timedPacket = windowPackets.get(relative);
            windowTimers.remove(timedPacket);

            // expire the timer for the packet and send it immediately
            windowTimers.add(timedPacket.reset());
        }
    }


    public int nextNumber(int number) {
        return (number + 1) % sequenceLength;
    }

    public int relativeNumber(int number) {
        return (number - windowStart + sequenceLength) % sequenceLength;
    }

}
```

Sliding Window Algorithms

For flow control, we have considered three protocols such as stop and wait, gobackN and selective Repeat. Stop and Wait is a special case of gobackN protocol with sender window size =1.

Coming to pros and cons of Stop and Wait vs Sliding Window algorithms.

(Stop and Wait.) Pros.

The only advantage is the simplicity of it’s implementation.

(Stop and Wait) Cons.

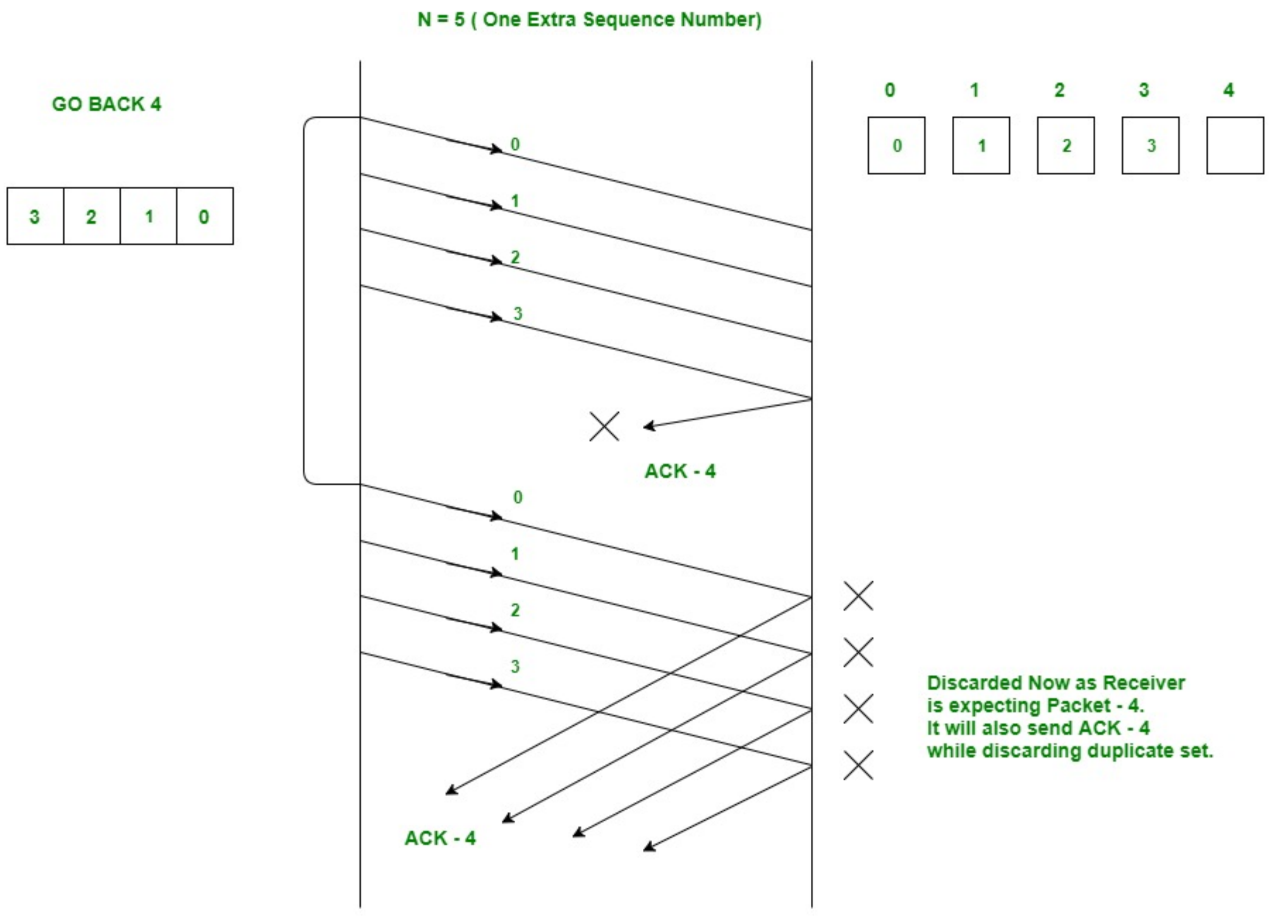
The sender needs to wait for the ACK everytime before sending a frame. This is a source of inefficiency and is particularly bad when propagation delay is much higher.

(Sliding Window) Pros.

- Performs much better than stop and wait algorithm.
- The efficiency is much high as multiple frames can be sent one after another.

(Sliding Window) Cons.

The implementation is fairly difficult.


N = 5 (One Extra Sequence Number)
GO BACK 4
3 2 1 0
0 1 2 3 4
0 1 2 3
0
1
2
3
ACK - 4
0
1
2
3
Discarded Now as Receiver is expecting Packet - 4. It will also send ACK - 4 while discarding duplicate set.
ACK - 4

Go-BackN protocol

Receiver Side

```
public void gobackN(Packet packet) {
        int seqNumber = packet.getSeqNumber();

        if (seqNumber == expectedNumber) {
            // append to window for outputing
            window.set(0, packet);
            expectedNumber = nextNumber(expectedNumber);
        }

        // else silently discard
    }
```

Sender Side (GobackN and Selective Repeat)

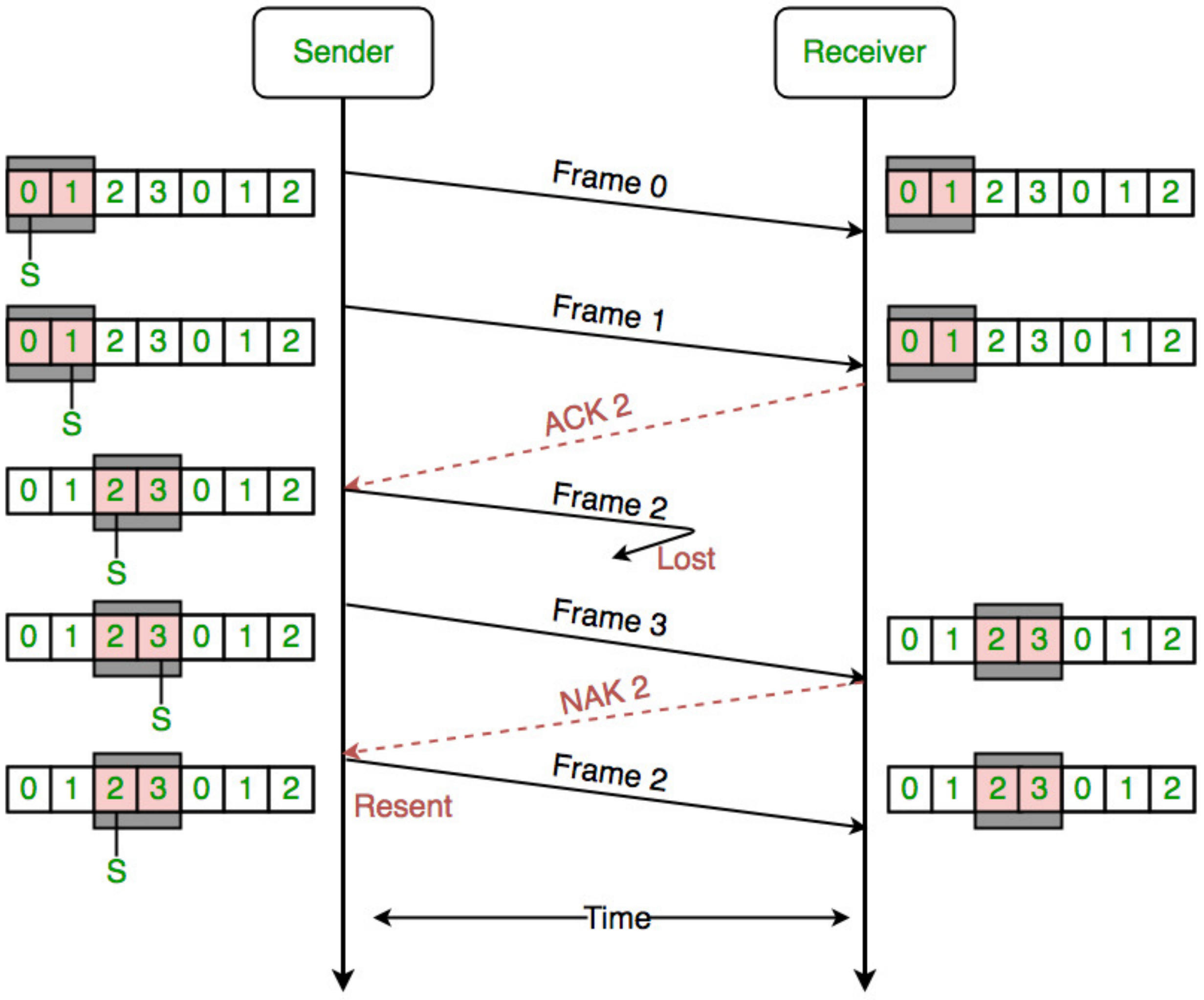
```
private void windowWorker() {
        while (true) {
            try {
                // Wait for next packet to expire

                TimedPacket timedPacket = windowTimers.take(); // Blocking

                if (timedPacket.getPacket().getSeqNumber() == windowStart && gobackn &&
flag) {
                    for (int i = windowStart; i < windowPackets.size(); i++) {
                        TimedPacket tp = windowPackets.get(i);
                        windowTimers.remove(tp);
                        windowTimers.add(tp.reset());
                    }
                    flag = false;
                }
                if (timedPacket.getPacket().getSeqNumber() == windowStart && !flag) {
                    flag = true;
                }
                sendPacket(timedPacket.getPacket());


                // renew the delay on the packet and feed back into schedule
                windowTimers.put(timedPacket.repeat());


            } catch (InterruptedException e) {
                System.out.println("Terminating scheduleHandler"); //DEBUG
                return;
            }
        }
    }
```



Selective Repeat

Receiver Side(Selective Repeat)

```
public void selectiveRepeat(Packet packet) {
        int packetNum = packet.getSeqNumber();
        int numberPos = posInWindow(packetNum);
        int expectedPos = posInWindow(expectedNumber);

        if (numberPos < expectedPos) {
            window.set(numberPos, packet);
        } else if (numberPos == expectedPos) {
            window.set(numberPos, packet);
            expectedNumber = nextNumber(expectedNumber);
        } else if (numberPos < windowLength) {
            while (expectedNumber != packetNum) {
                Packet nak = Packet.makeNak(expectedNumber, srcAdd, destAdd);
                sendPacket(nak);
                expectedNumber = nextNumber(expectedNumber);
            }
            window.set(numberPos, packet);
            expectedNumber = nextNumber(expectedNumber);
        } else {
            Packet ack = Packet.makeAck(windowStart, srcAdd, destAdd);
            sendPacket(ack);
        }


    }
```

Analysis

The bandwidth x RTT product represents the amount of data that can be sent before the first response is received. It plays a large role in the analysis of transport protocols.

There is a tradeoff, with a large window size, the RTT simply grows, due to queueing delays .

Due to heavy multithreading, results may be a little skewed and sometimes results in deadlocks due to involvement of several locks due to synchronise several message queues for communication between the sender and the receiver.

Due to NAKs sent in Selective Repeat and individual ACKs compared to cumulative in gobackN, it results in faster transmission of packets than gobackn and stop and wait protocols.

For Selective Repeat Protocol,
The comparison for time of transmission of 100 and 50 packets vs window size in a noisy channel.

packets/ Window size	2	4	8
100	8291ms	4123ms	255ms
50	4222ms	4074ms	254ms

For goBack N protocol, sending 100 packets takes almost 80 seconds under the same conditions.

The number of average tries required to send packets under the condition of adding noise in the channel due to which packets get lost by a certain probability.

Packets/Tries	Stop and Wait	Go Back N	Selective Repeat
100(window size=4)	5	20	8
100(window size=8)	5	40	8

Comments

The assignment greatly helped in understanding network flow control algorithms. This was a relatively hard assignment and so there are some caveats in the implementation however, it mostly covers all aspects of the flow control protocols.