JSS MAHAVIDYAPEETA

JSS SCIENCE AND TECHNOLOGY UNIVERSITY

JSS Technical Institution Campus, MYSORE-570006

COMPUTER NETWORKS (IS620)

Project Title : Implementation and Simulation of Go-Back-N Protocol

Submitted by:

Dhanush Raj N	*01JST17IS016*
K Shrikrishna Hebbar	*01JST17IS023*
Kirthan M	*01JST17IS024*

6th sem,

Information Science and Engineering

24/05/2020

Table of Contents

1 Problem Statement

Implement Go-Back-N protocol for the transmission of multiple packets without waiting for the acknowledgment. Generate sequence number for sent packets and acknowledgment.

2 Introduction

Go-Back-N protocol, also called Go-Back-N Automatic Repeat request, is a data link layer protocol that uses a sliding window mechanism for reliable and sequential delivery of data frames or packets. The process of sending the packets continues although number of frames to be sent is specified without receiving acknowledgement. The transmit window size is N and receive window size is 1. Go-Back-N ARQ supports for sending multiple frames before receiving the acknowledgment for the first frame. The frames are sequentially numbered and a finite number of frames are present. The maximum number of frames or packets that can be sent depends upon the size of the sending window. If the acknowledgment of a frame is not received within specified time period, all frames starting from that frame are retransmitted.

This project implements client-server model with Go-Back-N ARQ as its foundation. It replicates all the internal functioning of Go-Back-N ARQ such as acknowledgements, appropriate time-outs and retransmissions. It is a specific instance of the automatic repeat request (ARQ) protocol, in which the sending process continues to send a number of frames or packets specified by a window size even without receiving an acknowledgement (ACK) packet from the receiver. It is a special case of the general sliding window protocol with the transmit window size of N and receive window size of 1. It can transmit N frames to the receiver before requiring an ACK.

The long round trip time can have major effects on the efficiency of the channel bandwidth utilization. In Go-Back-N protocol, the Sender transmits up to a specified number of frames without blocking. In this protocol, the Sender can continuously transmit frames for a time equal to the round-trip time without filling up the window. This ability of the Sender to fill the window without stopping is known as pipelining. Pipelining frames over an unreliable communication channel can cause some serious issues. Consider a situation where a frame in the middle of a long stream is damaged or lost, a large number of frames following it will be received by the Receiver before the sender even finds out that anything is wrong. The damaged frame received by the receiver is discarded. The serious issue here is what should be done with the large number of correctly received frames following it. This situation can be handled by the Go-Back-N protocol. Go-Back-N protocol deals with the above situation and handles errors when the frames are pipelined. In this protocol, the Receiver simply discards all the correctly received frames following the bad frame and sends no acknowledgements for the discarded frames. This protocol has a Receiver window of size 1. The pipeline will begin to empty, if the sender's window fills up before the timer expires. Eventually, the sender will time out and retransmit all 5 unacknowledged frames in order, starting with the damaged frame. This protocol can waste a lot of bandwidth if the packet loss rate is high.

The size of the sending window determines the sequence number of the outbound frames. If the sequence number of the frames is an n-bit field, then the range of sequence numbers that can be assigned is 0 to 2^n-1. Consequently, the size of the sending window is 2^n-1. Thus in order to accommodate a sending window size of 2^n-1, a n-bit sequence number is chosen.

Characteristics of Go-Back-N protocol:

The three main characteristic features of GBN are:

1. **Sender Window Size (WS)**

   It is the size 'N' itself. If we say the protocol is GB10, then Sender Window Size (WS) = 10. N should be always greater than 1 in order to implement pipelining. For N = 1, it reduces to Stop-and-Wait protocol.

   Efficiency of GBN = N/(1+2a)
   where a = Tp/Tt

   If B is the bandwidth of the channel, then

   Effective Bandwidth or Throughput

   = Efficiency * Bandwidth

   = (N/(1+2a)) * B

2. **Receiver Window Size (WR)**

   Receiver Window Size (WR) is always 1 in GBN.

3. **Acknowledgements**

   There are 2 kinds of acknowledgements namely:

   - **Cumulative ACK**: One acknowledgement is used for many packets. The main advantage is traffic is less. A disadvantage is less reliability as if one ACK is the loss that would mean that all the packets sent are lost.

   - **Independent ACK**: If every packet is going to get acknowledgement independently. Reliability is high here but a disadvantage is that traffic is also high since for every packet we are receiving independent ack.

3 Literature Survey

[1] D Towsley et al., (1979) presented the stutter Go Back-N Protocol consider the problem of designing a good ARQ protocol for a message transmission environment characterized by high error rates and/or long propagation delays. We derive the average queue length for an idealized ARQ protocol for such an environment. We also describe a modification that can be made to existing ARQ protocols which can significantly decrease queue lengths in such an environment. We show that for the case of low message traffic rates, the modified Go Back- N protocol approaches the idealized scheme in performance.

[2] D Towsley et al., (1985) discussed a performance analysis of an error control protocol operating in a point-to-multipoint environment. This protocol, called broadcast go-back- N [BGB(N)], is a generalization of the standard go-back- N point-to-point protocol. Expressions for the message throughput and expected message delay are obtained. Numerical results are provided for a satellite communications system.

[3] JL Wang et al., (1989) presented a delay minimization of adaptive go-back-N ARQ Protocols for point to multipoint communication so some data-link layer error control go-back-N ARQ (automatic repeat-request) protocols are studied that are suitable for point-to-multipoint communication over broadcast channels where data are delivered to the destinations in the order they are sent. A series of protocols differing in the way that the sender uses the outcomes of the previous transmissions are studied. The system delay, rather than the throughput, is the optimization measure. The optimal number of copies that the sender should transmit to minimize the time between when the sender first transmits a data frame and when the data frame is accepted by all the receivers is determined. The results show that sending the optimum number of copies of a data frame instead of just a single copy significantly improved the delay performance.

[4] M Yoshimoto et al., (1991) discussed a performance analysis of automatic repeat request (ARQ) protocols. In our model, each message arriving at a transmitter is divided into several packets, which are continuously transmitted to a receiver according to Go-Back-N ARQ. Because of the assumption that messages are served on FCFS basis, transmission of a message is commenced after the completion of the previous message's transmission.

[5] M Zorzi, RR Rao et al., (1995) proposed an ARQ Go-Back-N protocol with a time-out mechanism is studied. Transmissions on both the forward and the reverse channels are assumed to be subject to Markovian errors. An approach based on renewal theory is further extended and the steady state number of packets in the ARQ system is evaluated. This quantity is one of two components that contribute to the delay in the overall system. Simulation results, that confirm the analysis, are also presented. Based on the delay analysis, the trade-off involved in the choice of the time-out parameter is identified and discussed especially in the context of a mobile radio channel.

[6] L Freiberg et al., (1997) demonstrated a systematic and efficient method to concurrently optimize a multiplicity of design variables for an adaptive Go-Back-N ARQ strategy both in noiseless and noisy feedback channels. For this continuous ARQ protocol, we adapt the number of identical message blocks sent in each transmission dynamically to the estimated channel condition. The channel state information is obtained by counting the contiguous ACK and NACK messages. Exploiting the asymptotic properties of the steady-state probability expressions, we show analytically that the optimum solution indeed lies in the infinite space. Subsequently, a simple method to estimate the suboptimal design parameters is suggested. Our approach of minimizing the mean square error (MSE) function also yields to a quantitative study of the appropriateness of the selected parameters. The results provide fundamental insights into how these key parameters interact and determine the system performance.

[7] MG Gouda et al., (2001) demonstrated a communication protocol is stabilizing if and only if starting from any unsafe state (i.e. one that violates the intended invariant of the protocol), the protocol is guaranteed to converge to a safe state within a finite number of state transitions. Stabilization allows the processes in a protocol to reestablish coordination between one another whenever coordination is lost due to some failure. The authors identify some important characteristics of stabilizing protocols; they show in particular that a stabilizing protocol is nonterminating, has an infinite number of safe states, and has timeout actions. Finally, they discuss how to redesign a number of well-known protocols to make them stabilizing; these include the sliding-window protocol and the two-way handshake.

[8] D Hercog et al., (2002) presented a generalised sliding window protocol that is a new retransmission strategy of the basic sliding window protocol is proposed. The stop-and-wait, go-back-n and selective-repeat protocols are special cases of this generalised protocol. The protocol is correct for any combination of send and receive window widths. The protocol efficiency increases with the receive window width.

[9] DA Chkilaev et al., (2011) presented a Sliding Window Protocol (SWP) which provides reliable and e-client transmission of data over unreliable channels. It seems quite important to give a formal proof of correctness for the SWP, especially because the high degree of parallelism in this protocol creates a significant potential for errors. However, the efforts to provide a deductive verification for the SWP had only a limited success so far. To fill this gap, we follow a new approach, in which the protocol is specified by a state machine in the language of the verification system PVS. We also formalize its safety property and prove it using the interactive proof checker of PVS.

[10] K Derom et al., (2012) proposed a system and method for data communications involves receiving a data frame in a receiver using a sliding receive window protocol, where a sliding receive window of the sliding receive window protocol is identified by sliding receive window information, and determining whether the data frame is a retransmitted data frame using the sliding receive window information.

4 Protocol and Standards followed

We implement Go-Back-N protocol in java using ARQ protocol, sliding window protocol and following the specification in RFC 768 (User Datagram Protocol).

- Go-Back-N ARQ is a specific instance of the **automatic repeat request (ARQ) protocol**, in which the sending process continues to send a number of frames specified by a window size even without receiving an acknowledgement (ACK) packet from the receiver.
- It is a special case of the general **sliding window protocol** with the transmit window size of N and receive window size of 1. It can transmit N frames to the peer before requiring an ACK. Sliding window protocol is a feature of packet-based data transmission protocols. Sliding window protocols are used where reliable in-order delivery of packets is required, such as in the data link layer (OSI layer 2) as well as in the Transmission Control Protocol (TCP).
- We use an unreliable channel and **UDP protocol** to implement Go-Back-N protocol. UDP is defined in **RFC 768**. RFC 768 is the User Datagram Protocol which is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) is used as the underlying protocol. This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications which require ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP).

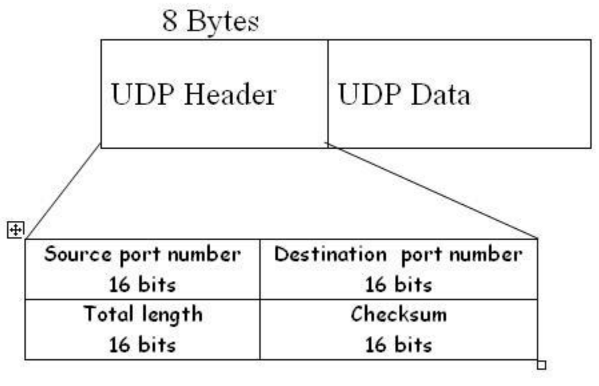


Fig. UDP Segment Structure

5 Algorithm

Receiver side:

Step-1: Packet received from the sender's side. Unpack the packet to get sequence number, checksum, header and data.
Step-2: If the packet is header consists of all 1's, then it's the last packet sent by the sender. Exit the program.
Step-3: Calculate the probability for packet loss. If the probability is less than the threshold, then the packet is considered lost and proper message is printed for simulation.
Step-4: Calculate the checksum for the packet. If the checksum does not match, then the packet is corrupted. Print appropriate message.
Step-5: If the expected packet is received, then send the acknowledgement for that packet, send the last received flag to the sequence number of the current packet received.
Step-6: If the packet receive is not expected, then it is considered out of order and is discarded.

Sender side:

Step-1: Sender prepares the message to be sent as per the input given by the user.
Step-2: Checksum is calculated for the message to be sent. A packet is created using sequence number, checksum, header, and the message.
Step-3: Packet is appended into the send buffer and the timeout value is set for the packet.
Step-4: Bit error probability is calculated, and if the probability appears to be less than the threshold set, then an error is induced into the message just before it is sent.
Step-5: Meanwhile, a thread has already been started to look for acknowledgements from the receiver. Upon receiving an acknowledgment, it is unpacked and acknowledgement number is obtained.
Step-6: Acknowledgement lost probability is calculated. If it is less than the specified threshold, then the acknowledgement is lost and appropriate message is printed.
Step-7: If the acknowledgement is received for the latest packet sent, then appropriate message is printed and the send buffer and timeout value for the corresponding packet and all the packet preceding it is set to reflect the receipt of the acknowledgement.
Step-8: If the acknowledgement is received for some other packet, then appropriate message is printed and the send buffer and timeout value for the corresponding packet is set to reflect the receipt of the acknowledgement.
Step-9: If all the packets are sent and all the acknowledgements are received, then a flag is set to indicate the same. The sender will send a final packet to indicate to the receiver that there are no more packets to send.

6 Implementation

Requirement is to implement sliding window protocol – Go-Back-N (GBN) using an unreliable channel and UDP protocol. Checksum of the segment to be sent needs to be calculated and placed in the packet which is sent from sender to the receiver. The receiver needs to parse the checksum and check for bit errors.

For simulation purposes, the sender and receiver program must include some faulty network behaviors to test the implementation and observe the robustness of the protocol. The different cases are as follows:

1. **Bit/Checksum error:** It means alter the sending message just before sending the packet so that a checksum error will be detected at the receiver end. The response from the receiver needs to be recorded here to observe how the receiver reacts to a faulty packet. Probability of occurrence of checksum error can be fixed at 0.1 (10 segments out of 100 sent can have this issue). We can also modify this probability value in the program.

2. **Lost Packet:** In this scenario, receiver will treat a packet as lost, even though it is received perfectly. The response from the receiver needs to be recorder here to observe how the receiver reacts to a lost packet. Probability of loss of packet can be fixed at 0.1 (10 segments out of 100 gets lost). We can also modify this probability value in the program.

3. **Lost Acknowledgements:** This is similar to lost packet, but this occurs at the sender. An acknowledgement to a packet received at the receiver's end is sent perfectly by the receiver, but it gets lost at the sender's end. Response from the sender for the lost packet need to be recorded here to observe how the sender reacts when it does not receive an acknowledgement for a packet under the stipulated time. Probability of the acknowledgment to get lost can be fixed at 0.05 (5 ACK's out of 100 gets lost). We can also modify this probability value in the program.

Some additional classes are used to implement the program. These are used to carry out various operations in the program.

The additional classes used other than main sender and receiver class are:

1. **AckData:** Its object is used to send the details about the acknowledgement.
2. **InitiateTransfer:** Used to send the initial SYN data which includes packet size, window size, number of packets to the receiver.
3. **SegmentData:** Used to send the data, here we are sending a character as data. We are randomly generating the character data.
4. The **Data.txt** is the input file which contains the protocol type, Packet size, window size and timeout between each packet transmission. So we can test for Go-Back-N protocol on top of Data.txt file and record the observations.

Program environment and execution details:

1. Implemented using JAVA programming language.

2. First compile the Receiver.java and Sender.java files. We get a bytecode class file using which we can execute the program.
   javac <Receiver filename>
   javac <Sender filename>
   Eg.,: $ javac Receiver.java
   $ javac Sender.java

3. For the receiver program, we need to pass the port number as argument. To run the receiver program, use the following command format:
   java <Receiver bytecode file> <port number>
   For example:
   $ java Receiver 7780

4. For the sender program, we need to pass a file which holds the configuration information like protocol type, Packet size, window size and timeout between each packet transmission. We also need to pass the same port number as specified on the receiver, and the number of packets (The message to be sent is hardcoded in the code. We can increase the size of this message by using this parameter. Say, number of packets is 10 and the message to be sent is AB. The final message will be prepared before the sending starts, which will be "ABABABABABABABABABAB". This is done for illustration purposes). To run the sender program, use the following command format:
   java <sender bytecode file> <configuration filename> <port number> <number of packets>
   For example:
   $ java Sender Data.txt 7780 10

Pseudo code of the program:

```
N  = window size
Sn = sequence number
Sb = sequence base
Sm = sequence max
ack = ack number
nack = first non acknowledged

Receiver:
Do the following forever:
  Randomly accept or reject packet

  If the packet is received and the packet is error free
        Accept packet
        Send a positive ack for packet
  Else
        Refuse packet
        Send a negative ack for packet

Sender:
Sb = 0
Sm = N – 1
ack = 0
Repeat the following steps forever:
  Send packet with ack
  If positively ack is recieved:
         ack++
         Transmit a packet where Sb <= ack <= Sm.
         packets are transmitted in order
  Else
          Enqueue the nack into the queue
  //check if last packet in the window is sent
  if(ack == Sm)
    if(queue is not empty)
          // start from the first nack packet
         nack = queue.front();
         empty the queue
         ack = nack

    Sm = Sm + (ack – Sb)
    Sb = ack
```

7 Program

- **Receiver:**

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.util.ArrayList;
import java.util.Collections;

public class Reciever {

    public static final double LOST_PACK_PROBABILITY = 0.1;

    public static void main(String[] args) {

        DatagramSocket socket = null;
        int portNumber = 0;
        if (args.length == 1) {

            portNumber = Integer.parseInt(args[0]);

        } else {
            System.out.println("Invalid Parameters");
        }

        byte[] incomingData = new byte[1024];
        try {

            socket = new DatagramSocket(portNumber);
            System.out.println("Reciver Side is Ready to Accept Packets at PortNumber: " + portNumber+ "\n");

            DatagramPacket initialPacket = new DatagramPacket(incomingData, incomingData.length);
            socket.receive(initialPacket);
            byte[] data1 = initialPacket.getData();
            ByteArrayInputStream inInitial = new ByteArrayInputStream(data1);
            ObjectInputStream isInitial = new ObjectInputStream(inInitial);
            InitiateTransfer initiateTransfer = (InitiateTransfer) isInitial.readObject();
            System.out.println("\nInitial Data Recieved = " + initiateTransfer.toString() + "\n");

            int type = initiateTransfer.getType();
            InetAddress IPAddress = initialPacket.getAddress();
            int port = initialPacket.getPort();
            initiateTransfer.setType(100);

            ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
            ObjectOutputStream os = new ObjectOutputStream(outputStream);
            os.writeObject(initiateTransfer);

            byte[] replyByte = outputStream.toByteArray();
            DatagramPacket replyPacket = new DatagramPacket(replyByte, replyByte.length, IPAddress, port);
            socket.send(replyPacket);

            if (type == 0) {
                initiateTransfer.setType(0);
                gbnTransfer(socket, initiateTransfer);
            }
        } catch (Exception e) {

            e.printStackTrace();
        }

    }
    private static void gbnTransfer(DatagramSocket socket, InitiateTransfer initiateTransfer)
            throws IOException, ClassNotFoundException {

        ArrayList<SegmentData> received = new ArrayList<>();
        boolean end = false;
        int waitingFor = 0;
        byte[] incomingData = new byte[1024];

        while (!end) {
            DatagramPacket incomingPacket = new DatagramPacket(incomingData, incomingData.length);
            socket.receive(incomingPacket);
            byte[] data = incomingPacket.getData();
            ByteArrayInputStream in = new ByteArrayInputStream(data);
            ObjectInputStream is = new ObjectInputStream(in);
            SegmentData segmentData = (SegmentData) is.readObject();
            System.out.println(" \n Packet Received  = " + segmentData.getSeqNum());

            char ch = segmentData.getPayLoad();
            int hashCode = ("" + ch).hashCode();
            boolean checkSum = (hashCode == segmentData.getCheckSum());

            if (!checkSum) {
                System.out.println("Error Occured in the Data");
            }

            if (segmentData.getSeqNum() == waitingFor && segmentData.isLast() && checkSum) {

                waitingFor++;
                received.add(segmentData);
                System.out.println("Last packet received");

                end = true;

            } else if (segmentData.getSeqNum() == waitingFor && checkSum) {
                waitingFor++;
                received.add(segmentData);
            }

            else if (!checkSum) {
                System.out.println("Checksum Error");
                segmentData.setSeqNum(-1000);
            }

            else {
                System.out.println("Packet discarded (not in order)");
                segmentData.setSeqNum(-1000);
            }

            InetAddress IPAddress = incomingPacket.getAddress();
            int port = incomingPacket.getPort();

            AckData ackData = new AckData();
            ackData.setAckNo(waitingFor);

            ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
            ObjectOutputStream os = new ObjectOutputStream(outputStream);
            os.writeObject(ackData);

            byte[] replyByte = outputStream.toByteArray();
            DatagramPacket replyPacket = new DatagramPacket(replyByte, replyByte.length, IPAddress, port);

            if (Math.random() > LOST_PACK_PROBABILITY && segmentData.getSeqNum() != -1000) {
                String reply = "Sending Acknowledgment Number :" + ackData.getAckNo() + "\n";
                System.out.println(reply);
                socket.send(replyPacket);
            } else if (segmentData.getSeqNum() != -1000) {
                int length = received.size();
                System.out.println("Packet Lost");
                received.remove(length - 1);
                waitingFor--;
                if (end) {
                    end = false;
                }
            }
        }
    }
}
```

- **Sender:**

```
import java.io.BufferedReader;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketTimeoutException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Random;

public class Sender {
    public static int TIMER = 3000;
    public static final double LOST_ACK_PROBABILITY = 0.05;
    public static final double BIT_ERROR_PROBABILITY = 0.1;
    public static void main(String[] args) {
        BufferedReader br = null;
        String fileName = "";
        int portNumber = 0;
        int numPackets = 0;
        String type = "";
        int sequenceNumBits = 0;
        int windowSize = 0;
        long timeOut = 0;
        long sizeSegment = 0;
        |
        if (args.length == 3) {
            fileName = args[0];
            portNumber = Integer.parseInt(args[1]);
            numPackets = Integer.parseInt(args[2]);
        } else {
            System.out.println("Invalid Parameters");
        }
        try {
            br = new BufferedReader(new FileReader(fileName));
            String line = br.readLine();
            int i = 0;
            while (line != null) {
                if (i == 0) {
                    type = line.trim();
                } else if (i == 1) {
                    sequenceNumBits = Integer.parseInt(line.charAt(0) + "");
                    windowSize = Integer.parseInt(line.charAt(2) + "");
                } else if (i == 2) {
                    timeOut = Long.parseLong(line);
                } else if (i == 2) {
                    timeOut = Long.parseLong(line);
                } else if (i == 3) {
                    sizeSegment = Long.parseLong(line);
                }
                i++;
                line = br.readLine();
            }
            br.close();
        } catch (Exception e) {

            System.out.println("Error occured while reading file");
        }

        System.out.println("Type: " + type + " Number of Seq bits: " + sequenceNumBits + " Window Size " + windowSize
                + " Timeout: " + timeOut + " Segment Size: " + sizeSegment);
        TIMER = (int) timeOut;

        try {
            sendData(portNumber, numPackets, type, sequenceNumBits, windowSize, timeOut, sizeSegment);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

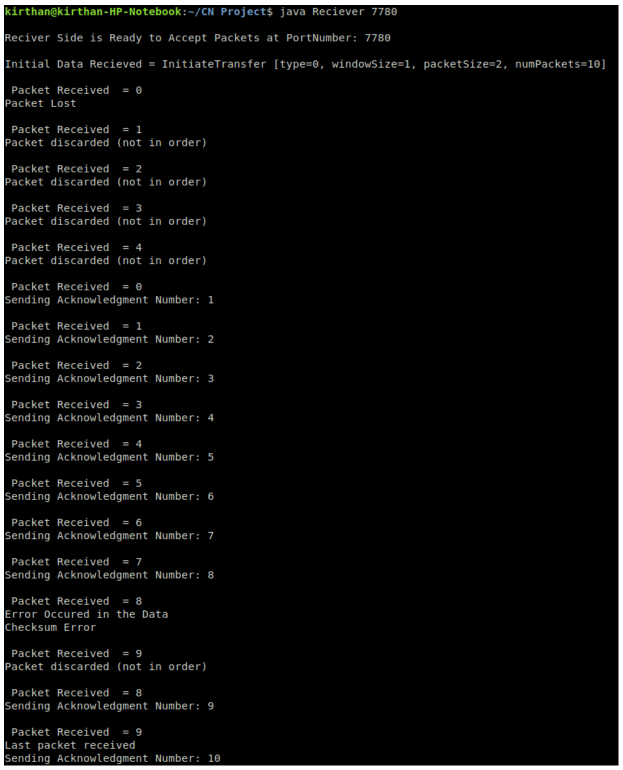
```
private static void sendData(int portNumber, int numPackets, String type, int sequenceNumBits, int windowSize,
        long timeOut, long sizeSegment) throws IOException, ClassNotFoundException, InterruptedException {
    ArrayList<SegmentData> sent = new ArrayList<>();

    int lastSent = 0;

    int waitingForAck = 0;
    String alphabet = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    int N = alphabet.length();
    DatagramSocket Socket = null;
    if (type.equalsIgnoreCase("gbn")) {

        byte[] incomingData = new byte[1024];
        InitiateTransfer initiateTransfer = new InitiateTransfer();
        initiateTransfer.setType(0);
        initiateTransfer.setNumPackets(numPackets);
        initiateTransfer.setPacketSize(sizeSegment);
        initiateTransfer.setWindowSize(1);

        Socket = new DatagramSocket();
        InetAddress IPAddress = InetAddress.getByName("localhost");

        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        ObjectOutputStream os = new ObjectOutputStream(outputStream);
        os.writeObject(initiateTransfer);
        byte[] data1 = outputStream.toByteArray();

        DatagramPacket initialPacket = new DatagramPacket(data1, data1.length, IPAddress, portNumber);
        System.out.println("Sending Initial Data : " + "\n");
        Socket.send(initialPacket);

        DatagramPacket initialAck = new DatagramPacket(incomingData, incomingData.length);
        Socket.receive(initialAck);
        byte[] dataImp = initialAck.getData();
        ByteArrayInputStream inReturn = new ByteArrayInputStream(dataImp);
        ObjectInputStream isReturn = new ObjectInputStream(inReturn);
        InitiateTransfer initiateTransfer2 = (InitiateTransfer) isReturn.readObject();

        if (initiateTransfer2.getType() == 100) {
            while (true) {
                while (lastSent - waitingForAck < windowSize && lastSent < numPackets) {
                    if (lastSent == 0 && waitingForAck == 0) {
                        System.out.println("Timer Started for Packet:  " + 0);
                    }
                    Random r = new Random();
                    char ch = alphabet.charAt(r.nextInt(N));
                    int hashCode = ("" + ch).hashCode();
                    SegmentData segmentData = new SegmentData();
                    segmentData.setPayLoad(ch);
                    segmentData.setSeqNum(lastSent);
                    segmentData.setCheckSum(hashCode);
                    if (lastSent == numPackets - 1) {
                        segmentData.setLast(true);
                    }
                    if (Math.random() <= BIT_ERROR_PROBABILITY) {
                        segmentData.setPayLoad(alphabet.charAt(r.nextInt(N)));
                    }
                    outputStream = new ByteArrayOutputStream();
                    os = new ObjectOutputStream(outputStream);
                    os.writeObject(segmentData);
                    byte[] data = outputStream.toByteArray();
                    DatagramPacket sendPacket = new DatagramPacket(data, data.length, IPAddress, portNumber);
                    System.out.println("Sending Packet : " + segmentData.getSeqNum());
                    sent.add(segmentData);
                    Socket.send(sendPacket);
                    lastSent++;
                    Thread.sleep(2500);
                }
```

```
                    DatagramPacket incomingPacket = new DatagramPacket(incomingData, incomingData.length);
                    try {
                        Socket.setSoTimeout(TIMER);
                        Socket.receive(incomingPacket);
                        byte[] data = incomingPacket.getData();
                        ByteArrayInputStream in = new ByteArrayInputStream(data);
                        ObjectInputStream is = new ObjectInputStream(in);
                        AckData ackData = (AckData) is.readObject();

                        if (Math.random() > LOST_ACK_PROBABILITY) {
                            System.out.println("Received ACK for :" + (ackData.getAckNo() - 1) + "\n");
                            waitingForAck = Math.max(waitingForAck, ackData.getAckNo());
                            if (!(waitingForAck == numPackets)) {
                                System.out.println(
                                        "Timer Started for Packet:  " + ackData.getAckNo());
                            }
                        } else {
                            System.out.println("Acknowledgment Lost for :" + (ackData.getAckNo() - 1)
                                    + "\n");
                        }
                        if (ackData.getAckNo() == numPackets) {
                            break;
                        }
                    } catch (SocketTimeoutException e) {
                        System.out.println(
                                "Timeout Occured for Packet " + waitingForAck);
                        for (int i = waitingForAck; i < lastSent; i++) {
                            SegmentData segmentData = sent.get(i);
                            char ch = segmentData.getPayLoad();
                            int hashCode = ("" + ch).hashCode();
                            segmentData.setCheckSum(hashCode);
                            if (Math.random() <= BIT_ERROR_PROBABILITY) {
                                Random r = new Random();
                                segmentData.setPayLoad(alphabet.charAt(r.nextInt(N)));
                            }
                            outputStream = new ByteArrayOutputStream();
                            os = new ObjectOutputStream(outputStream);
                            os.writeObject(segmentData);
                            byte[] data = outputStream.toByteArray();
                            DatagramPacket sendPacket = new DatagramPacket(data, data.length, IPAddress, portNumber);
                            System.out.println("Re Sending Packet :" + segmentData.getSeqNum());
                            Socket.send(sendPacket);
                            Thread.sleep(3000);
                        }

                    }

                }
            }

        }

    }

}
```

8 Output

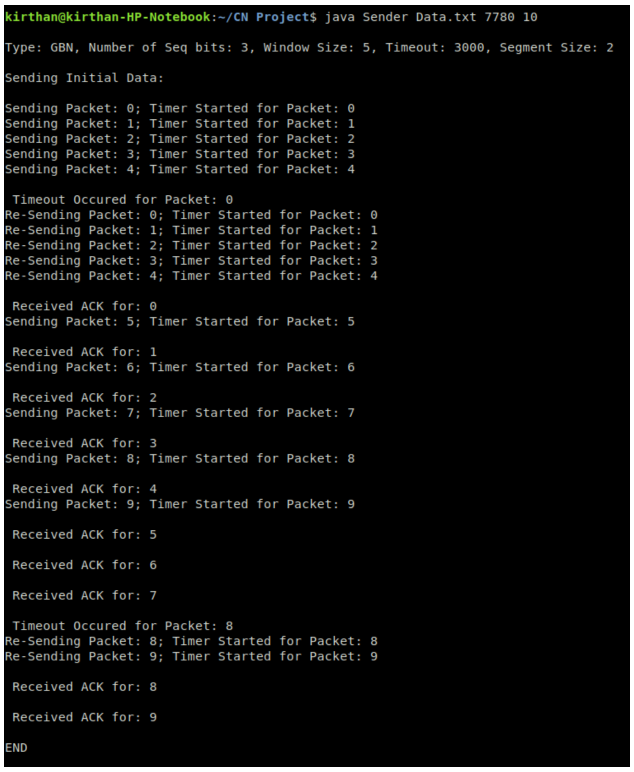
Screenshot 1: Illustrating Lost Packet and Checksum Error at the Receiver's side

Receiver:



```
kirthan@kirthan-HP-Notebook:~/CN Project$ java Reciever 7780

Reciver Side is Ready to Accept Packets at PortNumber: 7780

Initial Data Recieved = InitiateTransfer [type=0, windowSize=1, packetSize=2, numPackets=10]

 Packet Received  = 0
Packet Lost

 Packet Received  = 1
Packet discarded (not in order)

 Packet Received  = 2
Packet discarded (not in order)

 Packet Received  = 3
Packet discarded (not in order)

 Packet Received  = 4
Packet discarded (not in order)

 Packet Received  = 0
Sending Acknowledgment Number: 1

 Packet Received  = 1
Sending Acknowledgment Number: 2

 Packet Received  = 2
Sending Acknowledgment Number: 3

 Packet Received  = 3
Sending Acknowledgment Number: 4

 Packet Received  = 4
Sending Acknowledgment Number: 5

 Packet Received  = 5
Sending Acknowledgment Number: 6

 Packet Received  = 6
Sending Acknowledgment Number: 7

 Packet Received  = 7
Sending Acknowledgment Number: 8

 Packet Received  = 8
Error Occured in the Data
Checksum Error

 Packet Received  = 9
Packet discarded (not in order)

 Packet Received  = 8
Sending Acknowledgment Number: 9

 Packet Received  = 9
Last packet received
Sending Acknowledgment Number: 10
```

Sender:



```
kirthan@kirthan-HP-Notebook:~/CN Project$ java Sender Data.txt 7780 10

Type: GBN, Number of Seq bits: 3, Window Size: 5, Timeout: 3000, Segment Size: 2

Sending Initial Data:

Sending Packet: 0; Timer Started for Packet: 0
Sending Packet: 1; Timer Started for Packet: 1
Sending Packet: 2; Timer Started for Packet: 2
Sending Packet: 3; Timer Started for Packet: 3
Sending Packet: 4; Timer Started for Packet: 4

 Timeout Occured for Packet: 0
Re-Sending Packet: 0; Timer Started for Packet: 0
Re-Sending Packet: 1; Timer Started for Packet: 1
Re-Sending Packet: 2; Timer Started for Packet: 2
Re-Sending Packet: 3; Timer Started for Packet: 3
Re-Sending Packet: 4; Timer Started for Packet: 4

 Received ACK for: 0
Sending Packet: 5; Timer Started for Packet: 5

 Received ACK for: 1
Sending Packet: 6; Timer Started for Packet: 6

 Received ACK for: 2
Sending Packet: 7; Timer Started for Packet: 7

 Received ACK for: 3
Sending Packet: 8; Timer Started for Packet: 8

 Received ACK for: 4
Sending Packet: 9; Timer Started for Packet: 9

 Received ACK for: 5

 Received ACK for: 6

 Received ACK for: 7

 Timeout Occured for Packet: 8
Re-Sending Packet: 8; Timer Started for Packet: 8
Re-Sending Packet: 9; Timer Started for Packet: 9

 Received ACK for: 8

 Received ACK for: 9

END
```

As seen in the above screenshot, the left window is the receiver and the right window is the sender. We can see that the window size is set to 5, the maximum segment size is set to 2, the timeout period is set to 3000ms, the number of sequential bits is set to 3 and the total number of packets to be sent is entered as 10.

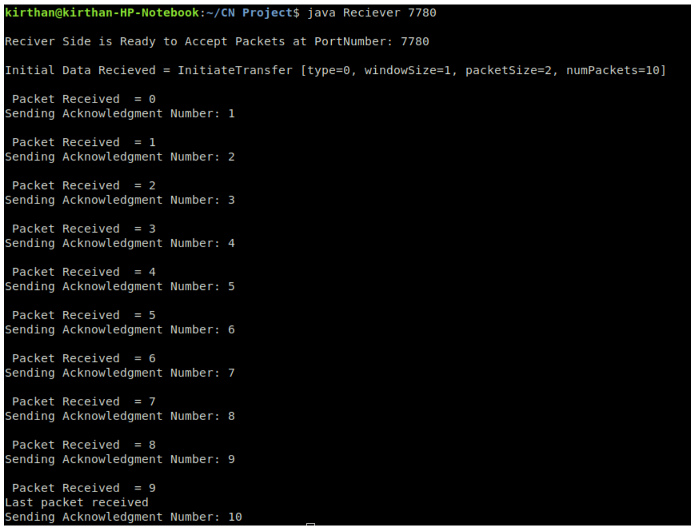
Execution explained:

1. The initial data such as window size, segment size and timeout is sent by the sender to the receiver. The window size is set to 5.
2. Packets 0 to 4 are sent in sequence (because window size is 5) by the sender and the timer is started.
3. While sending the packet 0, the packet gets lost and does not reach the receiver.
4. Hence no ACK is sent by the receiver.
5. The sender keeps on sending the remaining packets which are present in the window (i.e., 1, 2, 3 & 4).
6. But the receiver discards these packets since they are out of order.
7. Due to this, a timeout occurs for packet 0 on the sender's side.

8. The sender re-sends all the packets present in the window (i.e., 0,1,2,3 & 4) again without waiting for any ACKs.
9. Receiver receives packets from 0 through 4, and all are received correctly. Hence receiver sends ACKs for packets from 0 through 4.
10. After receiving acknowledgement for packet 0, the sender window moves by 1 and sends the packet 5.
11. In this way the sender sends the packets till packet 7 and the receiver sends the ACKs till packet 7 since there is no problem.
12. But when the sender sends the packet 8 to the receiver, a checksum error is detected on the receiver’s side.
13. A checksum error is detected for packet 8 and hence it is discarded. No ACK is sent for packet 8 by the receiver.
14. Packet 9 received is out of order, and hence discarded.
15. On the sender’s side, timeout for packet 8 has occurred and the packets 8 and 9 are resent since they are the only remaining packets available to send.
16. Packet 8 will be received correctly on the receiver’s side and receiver sends an ACK for that packet.
17. Sender will send the final packet 9 and it is correctly sent to the receiver which identifies it as the last packet and sends an ACK and finally the communication is stopped.

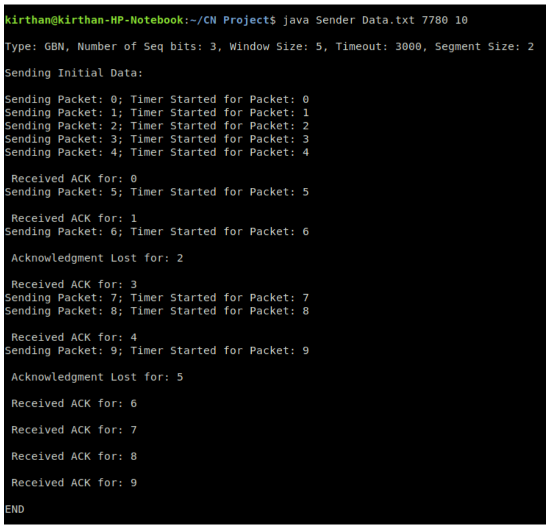
Screenshot 2: Illustrating Lost Acknowledgement at the Sender's side

Receiver:



```
kirthan@kirthan-HP-Notebook:~/CN Project$ java Reciever 7780

Reciver Side is Ready to Accept Packets at PortNumber: 7780

Initial Data Recieved = InitiateTransfer [type=0, windowSize=1, packetSize=2, numPackets=10]

 Packet Received  = 0
Sending Acknowledgment Number: 1

 Packet Received  = 1
Sending Acknowledgment Number: 2

 Packet Received  = 2
Sending Acknowledgment Number: 3

 Packet Received  = 3
Sending Acknowledgment Number: 4

 Packet Received  = 4
Sending Acknowledgment Number: 5

 Packet Received  = 5
Sending Acknowledgment Number: 6

 Packet Received  = 6
Sending Acknowledgment Number: 7

 Packet Received  = 7
Sending Acknowledgment Number: 8

 Packet Received  = 8
Sending Acknowledgment Number: 9

 Packet Received  = 9
Last packet received
Sending Acknowledgment Number: 10
```

Sender:



```
kirthan@kirthan-HP-Notebook:~/CN Project$ java Sender Data.txt 7780 10

Type: GBN, Number of Seq bits: 3, Window Size: 5, Timeout: 3000, Segment Size: 2

Sending Initial Data:

Sending Packet: 0; Timer Started for Packet: 0
Sending Packet: 1; Timer Started for Packet: 1
Sending Packet: 2; Timer Started for Packet: 2
Sending Packet: 3; Timer Started for Packet: 3
Sending Packet: 4; Timer Started for Packet: 4

 Received ACK for: 0
Sending Packet: 5; Timer Started for Packet: 5

 Received ACK for: 1
Sending Packet: 6; Timer Started for Packet: 6

 Acknowledgment Lost for: 2

 Received ACK for: 3
Sending Packet: 7; Timer Started for Packet: 7
Sending Packet: 8; Timer Started for Packet: 8

 Received ACK for: 4
Sending Packet: 9; Timer Started for Packet: 9

 Acknowledgment Lost for: 5

 Received ACK for: 6

 Received ACK for: 7

 Received ACK for: 8

 Received ACK for: 9

END
```

As seen in the above screenshot, the left window is the receiver and the right window is the sender. We can see that the window size is set to 5, the maximum segment size is set to 2, the timeout period is set to 3000ms, the number of sequential bits is set to 3 and the total number of packets to be sent is entered as 10.

Execution explained:

1. The initial data such as window size, segment size and timeout is sent by the sender to the receiver. The window size is set to 5.
2. Packets 0 to 4 are sent in sequence (because window size is 5) by the sender and the timer is started.
3. The receiver receives the packets 0 through 4 correctly and sends ACKs for the respective packets.
4. As soon as ACK for packet 0 is received at the sender, it moves the sender's window by 1 and sends the packet 5. Similarly, it sends packet 6.
5. But when the receiver sends the ACK for packet 2, the ACK is lost and does not reach the sender.
6. But we have received ACK for packet 3. Hence as per the cumulative ACK of Go-Back-N protocol, it means we have also received packet 2 even though we have lost ACK for packet 2.
7. Hence we send packet 7 and 8 since packets 2 and 3 are sent successfully.
8. Same case occurs for ACK 5 and since ACK 6 is received, we conclude that packet 5 is also received by the receiver.
9. In this way, sender sends the remaining packets present in the window and receives as ACK since they are sent successfully.
10. The sender sends the last packet and receives an ACK and finally the communication is stopped.

9 Conclusion

We can conclude by saying that Go-Back-N ARQ overcomes the short coming of Stop-and-Wait ARQ and also increases the efficiency of transmission by utilizing the bandwidth to the maximum extent possible. Implementation of Go-Back-N is comparatively simpler i.e., its algorithm is easy to understand and more efficient.

Go-Back-N ARQ is a more efficient use of a connection than Stop-and-wait ARQ, since unlike waiting for an acknowledgement for each packet, the connection is still being utilized as packets are being sent. In other words, during the time that would otherwise be spent waiting, more packets are being sent. However, this method also results in sending frames multiple times – if any frame was lost or damaged, or the ACK acknowledging them was lost or damaged, then that frame and all following frames in the send window (even if they were received without error) will be re-sent.

The advantages of Go-Back-N are:

- The sender can send many frames at a time.
- Timer can be set for a group of frames.
- One ACK can acknowledge more than one frame.
- Efficiency is more.

The disadvantages of Go-Back-N are:

- Buffer requirement.
- Transmitter needs to store the last N packets.
- Scheme is inefficient when delay is large and data transmission rate is high.
- Unnecessary retransmission of many error-free packets.

10 References

[1] D. Towsley, "The stutter go-back-N protocol", *IEEE Trans. Commun.*, vol. COM-27, pp. 869-875, June 1979.

[2] D. Towsley, " An Analysis of a Point-to-Multipoint Channel Using a Go-Back-N Error Control Protocol", *IEEE Trans. Commun.*, vol. COM-33, pp. 282-285, March 1985.

[3] J. L Wang, " Delay minimization of the adaptive go-back-N ARQ protocols for point to multipoint communication", *IEEE Trans. Commun.*, vol. COM-48, pp. 354-363, August 1989.

[4] M Yoshimoto, " Waiting time and queue length distributions for go-back-N and selective-repeat ARQ protocols".1991 pp. 247-260, August, June 1991.

[5] M Zorzi, R.R.Rao " Throughput analysis of Go-Back-N ARQ in Markov channels with unreliable feedback", *IEEE Trans. Commun.*, vol. COM-59, pp. 569-578, June 1995.

[6] L Freiberg , " Analysis and optimization of an adaptive go-back-N ARQ protocol for time-varying channels". pp. 685-696, August 1997.

[7] W Turin, " Throughput analysis of the Go-Back-N Protocol in fading radio channels", *IEEE Trans. Commun.*, vol. COM-17, pp. 881-887, March 1999.

[8] D Hercog , " Generalised sliding window protocol", *IEEE Trans. Commun.*, vol. COM-38, pp. 1067-1068, August 2002.

[9] D Chkliaev, " Verification and improvement of the sliding window protocol ", TACAS vol 2619, pp. 113-127, February 2011.

[10] K Derom, " System and method for data communications using a sliding window protocol with selective retransmission", *IEEE Commun. Technol*, vol. 22, pp. 5-17, Dec. 2012.

[11] Behrouz A. Forouzan, “Data Communications and Networking”, (third edition), published by McGraw-Hill Education, 2004.

[12] T. H. Beeforth, R. L. Grimsdale, F. Halsall, D. J. Woolons, "Proposed Organisation for Packet Switched Data-Communication Network," Proc. IEE, vol. 119, no. 12, Dec 1972, pp. 1677-1682.