Computer Network Lab Report

23020036096 Yuwei ZHAO
23th Computer Science and Technology

1 Combine code and LOG file analysis to illustrate the solution effect for each project.

1.1 RDT-1.0

RDT-1.0 simulates transmission over a reliable channel, and the data transfer between the sender and receiver is not affected by any network errors.

There is no need to modify any code in this project, as the reliable channel ensures that all packets are delivered correctly and in order. Therefore, the sender simply sends packets sequentially, and the receiver acknowledges each received packet as follows:

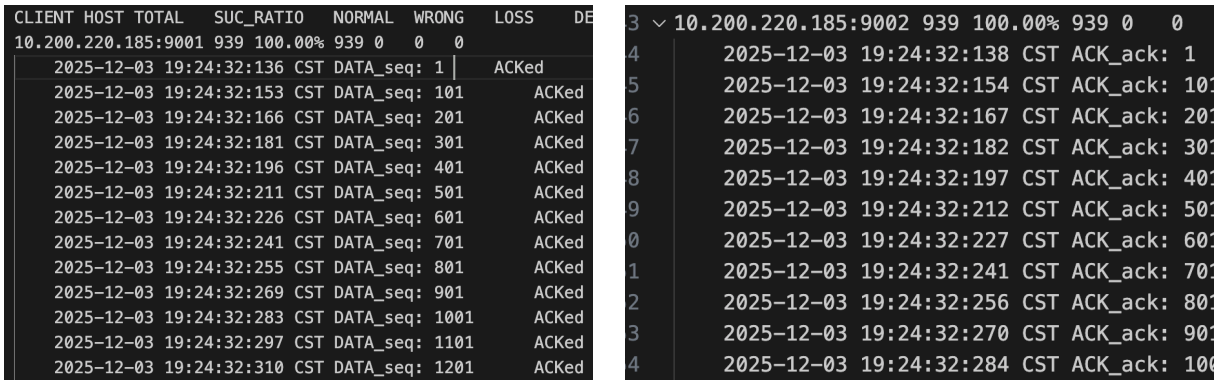


Figure 1: RDT-1.0 Sender/Receiver LOG & Code Snippet

1.2 RDT-2.0

RDT-2.0 simulates transmission over a channel that may introduce **bit errors**, and it uses checksums and acknowledgments to ensure reliable data transfer.

1.2.1 CheckSum.java

```java
public class CheckSum {
    public static short computeChkSum(TCP_PACKET tcpPack) {
        // Extract TCP header information
        TCP_HEADER header = tcpPack.getTcpH();

        // Calculate the checksum using the CRC32 algorithm
        CRC32 crc32 = new CRC32();
        crc32.update(header.getTh_seq());
```

```
        crc32.update(header.getTh_ack());

        // Traverse the data fields and update each data item to the checksum.
        for (int i : tcpPack.getTcpS().getData()) {
          crc32.update(i);
        }

        // Obtain the calculated checksum and convert it to the short data type for return
        return (short) crc32.getValue();
    }
}
```

1.2.2 TCP_Sender.java

```java
public void waitACK() {
    // Loop-check to confirm if there are any newly received ACKs in the confirmation number column.
    if (!this.ackQueue.isEmpty()) {
        int currentACK = this.ackQueue.poll();
        // System.out.println("CurrentAck: " + currentAck);
        if (currentACK == -1) {
            System.out.println();
            System.out.println("Retransmit: " + this.tcpPack.getTcpH().getTh_ack());
            System.out.println();

            udt_send(this.tcpPack);
            this.flag = 0;
        } else {
            System.out.println();
            System.out.println("Clear: " + currentACK);
            System.out.println();

            this.flag = 1;
            // break;
        }
    }
}
```

1.2.3 TCP_Receiver.java

```java
public void rdt_recv(TCP_PACKET recvPack) {
    // Received data packet - Check the checksum, and set the reply ACK message segment
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
        this.tcpH.setTh_ack(recvPack.getTcpH().getTh_seq());
        this.ackPack = new TCP_PACKET(this.tcpH, this.tcpS, recvPack.getSourceAddr());
        this.tcpH.setTh_sum(CheckSum.computeChkSum(this.ackPack));

```

```
8         System.out.println();
9         System.out.println("ACK: " + recvPack.getTcpH().getTh_seq());
10        System.out.println();
11
12        // Reply to ACK message segment
13        reply(this.ackPack);
14
15        // Insert the received correct and ordered data into the data queue, preparing for delivery
16        this.dataQueue.add(recvPack.getTcpS().getData());
17
18        // Deliver data (per 20 sets of data)
19        if (this.dataQueue.size() == 20)
20            deliver_data();
21    } else {
22        // Generate NACK message segment
23        this.tcpH.setTh_ack(-1);
24        this.ackPack = new TCP_PACKET(this.tcpH, this.tcpS, recvPack.getSourceAddr());
25        this.tcpH.setTh_sum(CheckSum.computeChkSum(this.ackPack));
26
27        System.out.println();
28        System.out.println("NACK: " + recvPack.getTcpH().getTh_seq());
29        System.out.println();
30
31        reply(this.ackPack);
32    }
33 }
```

The sender and receiver *LOGs* show that the checksum is calculated and verified correctly, ensuring reliable data transfer over a channel with bit errors:

```
NT HOST TOTAL    SUC_RATIO    NORMAL  WRONG    LOSS
50.201.164:9001 969 98.45%  954 15  0    0
2025-12-03 21:59:06:495 CST DATA_seq: 1      ACKed
2025-12-03 21:59:06:512 CST DATA_seq: 101        ACK
2025-12-03 21:59:06:528 CST DATA_seq: 201        ACK
2025-12-03 21:59:06:543 CST DATA_seq: 301        ACK
2025-12-03 21:59:06:558 CST DATA_seq: 401        ACK
2025-12-03 21:59:08:707 CST DATA_seq: 16601     ACKed
2025-12-03 21:59:08:720 CST DATA_seq: 16701 WRONG    NO
2025-12-03 21:59:08:721 CST *Re: DATA_seq: 16701
2025-12-03 21:59:08:734 CST DATA_seq: 16801     ACKed
2025-12-03 21:59:08:747 CST DATA_seq: 16901     ACKed
2025-12-03 21:59:08:760 CST DATA_seq: 17001     ACKed
```

```
10.150.201.164:9002 969 100.00% 969 0    0
    2025-12-03 21:59:06:498 CST ACK_ack: 1
    2025-12-03 21:59:06:513 CST ACK_ack: 101
    2025-12-03 21:59:06:529 CST ACK_ack: 201
    2025-12-03 21:59:06:544 CST ACK_ack: 301
    2025-12-03 21:59:06:558 CST ACK_ack: 401
    2025-12-03 21:59:06:571 CST ACK_ack: 501
    2025-12-03 21:59:08:693 CST ACK_ack: 16501
    2025-12-03 21:59:08:707 CST ACK_ack: 16601
    2025-12-03 21:59:08:720 CST ACK_ack: -1
    2025-12-03 21:59:08:721 CST ACK_ack: 16701
    2025-12-03 21:59:08:734 CST ACK_ack: 16801
```

Figure 2: RDT-2.0 Sender/Receiver LOG & Code Snippet

1.3 RDT-2.1

RDT-2.1 simulates transmission over a channel that may introduce **bit errors** and **packet loss**, and it uses sequence numbers, checksums, and acknowledgments to ensure reliable data transfer.

Therefore, in addition to the checksum implementation in RDT-2.0, sequence numbers are added to the sender and receiver code to distinguish between new and old packets. The sender alternates the sequence number between 0 and 1 for each packet sent, while the receiver checks the sequence number of each received packet to determine whether it is a new packet or a duplicate.

1.3.1 TCP_Sender.java

```java
public void rdt_recv(TCP_PACKET recvPack) {
    // Received data packet - Check the checksum, and set the reply ACK message segment
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
        this.tcpH.setTh_ack(recvPack.getTcpH().getTh_seq());
        this.ackPack = new TCP_PACKET(this.tcpH, this.tcpS, recvPack.getSourceAddr());
        this.tcpH.setTh_sum(CheckSum.computeChkSum(this.ackPack));

        System.out.println();
        System.out.println("ACK: " + recvPack.getTcpH().getTh_seq());
        System.out.println();

        // Reply to ACK message segment
        reply(this.ackPack);

        // Insert the received correct and ordered data into the data queue, preparing for delivery
        this.dataQueue.add(recvPack.getTcpS().getData());

        // Deliver data (per 20 sets of data)
        if (this.dataQueue.size() == 20)
            deliver_data();
    } else {
        // Generate NACK message segment
        this.tcpH.setTh_ack(-1);
        this.ackPack = new TCP_PACKET(this.tcpH, this.tcpS, recvPack.getSourceAddr());
        this.tcpH.setTh_sum(CheckSum.computeChkSum(this.ackPack));

        System.out.println();
        System.out.println("NACK: " + recvPack.getTcpH().getTh_seq());
        System.out.println();

        reply(this.ackPack);
    }
}
```

1.3.2 TCP_Receiver.java

```java
public void rdt_recv(TCP_PACKET recvPack) {
    // Received data packet - Check the checksum, and set the reply ACK message segment
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
```

```
4           this.tcpH.setTh_ack(recvPack.getTcpH().getTh_seq());
5           this.ackPack = new TCP_PACKET(this.tcpH, this.tcpS, recvPack.getSourceAddr());
6           this.tcpH.setTh_sum(CheckSum.computeChkSum(this.ackPack));
7
8           System.out.println();
9           System.out.println("ACK: " + recvPack.getTcpH().getTh_seq());
10          System.out.println();
11
12          // Reply to ACK message segment
13          reply(this.ackPack);
14
15          int currentSequence = (recvPack.getTcpH().getTh_seq() - 1) / 100;
16          if (currentSequence != this.lastSequence) {
17              this.lastSequence = currentSequence;
18
19              // Insert the received correct and ordered data into the data queue, preparing for delivery
20              this.dataQueue.add(recvPack.getTcpS().getData());
21
22              // Deliver data (per 20 sets of data)
23              if (this.dataQueue.size() == 20)
24                  deliver_data();
25          }
26      } else {
27          // Generate NACK message segment
28          this.tcpH.setTh_ack(-1);
29          this.ackPack = new TCP_PACKET(this.tcpH, this.tcpS, recvPack.getSourceAddr());
30          this.tcpH.setTh_sum(CheckSum.computeChkSum(this.ackPack));
31
32          System.out.println();
33          System.out.println("NACK: " + recvPack.getTcpH().getTh_seq());
34          System.out.println();
35
36          reply(this.ackPack);
37      }
38  }
```

The sender and receiver *LOGs* show that sequence numbers are used correctly to ensure reliable data transfer over a channel with bit errors and packet loss:

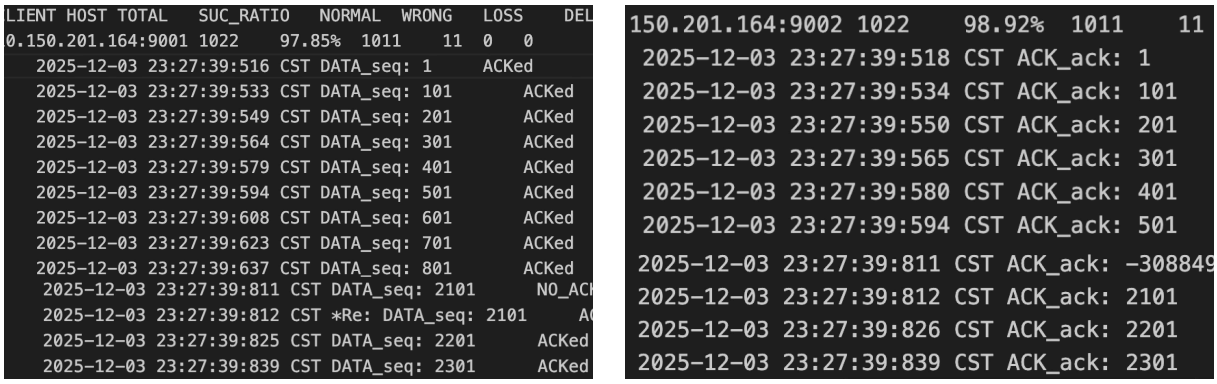


Figure 3: RDT-2.1 Sender/Receiver LOG & Code Snippet

1.4 RDT-2.2

RDT-2.2 simulates transmission over a channel that may introduce **bit errors** and **packet loss**, and it uses sequence numbers, checksums, and acknowledgments to ensure reliable data transfer. Its difference from RDT-2.1 lies in the cancel of negative acknowledgments (NACKs).

1.4.1 TCP_Receiver.java

Java

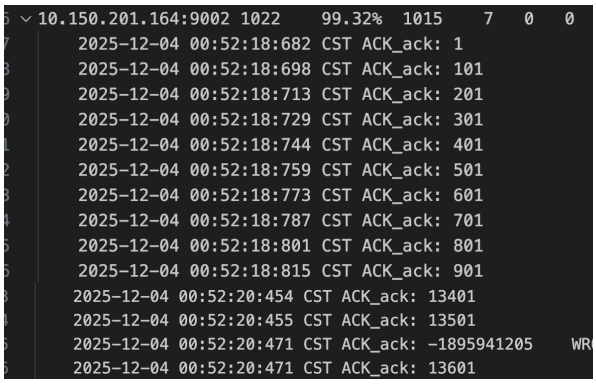
```java
public void rdt_recv(TCP_PACKET recvPack) {
    // Received data packet - Check the checksum, and set the reply ACK message segment
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
        this.tcpH.setTh_ack(recvPack.getTcpH().getTh_seq());
        this.ackPack = new TCP_PACKET(this.tcpH, this.tcpS, recvPack.getSourceAddr());
        this.tcpH.setTh_sum(CheckSum.computeChkSum(this.ackPack));

        System.out.println();
        System.out.println("ACK: " + recvPack.getTcpH().getTh_seq());
        System.out.println();

        // Reply to ACK message segment
        reply(this.ackPack);

        int currentSequence = (recvPack.getTcpH().getTh_seq() - 1) / 100;
        if (currentSequence != this.lastSequence) {
            this.lastSequence = currentSequence;

            // Insert the received correct and ordered data into the data queue, preparing for delivery
            this.dataQueue.add(recvPack.getTcpS().getData());

            // Deliver data (per 20 sets of data)
            if (this.dataQueue.size() == 20)
                deliver_data();
        }
    } else {
        // Generate NACK message segment
```

```
28          this.tcpH.setTh_ack(this.lastSequence * 100 + 1);
29          this.ackPack = new TCP_PACKET(this.tcpH, this.tcpS, recvPack.getSourceAddr());
30          this.tcpH.setTh_sum(CheckSum.computeChkSum(this.ackPack));
31
32          System.out.println();
33          System.out.println("ACK last sequence: " + this.lastSequence);
34          System.out.println();
35
36          reply(this.ackPack);
37      }
38  }
```

1.4.2 TCP_Sender.java

```java
1   public void recv(TCP_PACKET recvPack) {
2       if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
3           System.out.println();
4           System.out.println("Receive ACK Number: " + recvPack.getTcpH().getTh_ack());
5           System.out.println();
6
7           this.ackQueue.add(recvPack.getTcpH().getTh_ack());
8       } else {
9           System.out.println();
10          System.out.println("Receive corrupt ACK: " + recvPack.getTcpH().getTh_ack());
11          System.out.println();
12
13          this.ackQueue.add(-1);
14      }
15
16      // Process the ACK message
17      waitACK();
18  }
```

The Logs show that the receiver no longer sends NACKs, and the sender correctly handles duplicate ACKs to ensure reliable data transfer over a channel with bit errors and packet loss:



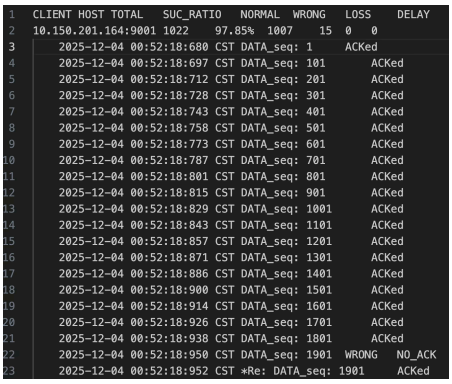


Figure 4: RDT-2.2 Sender/Receiver LOG & Code Snippet

1.5 RDT-3.0

RDT-3.0 simulates transmission over a channel that may introduce **bit errors** and **packet loss**, and it uses sequence numbers, checksums, acknowledgments, and timeouts to ensure reliable data transfer.

Therefore, in addition to the implementations in RDT-2.2, a timeout mechanism is added to the sender code. If an acknowledgment is not received within a specified time, the sender retransmits the packet.

1.5.1 TCP_Sender.java

Java

```java
public void rdt_send(int dataIndex, int[] appData) {
    // Generate the TCP data packet (set the sequence number, data field, and checksum), and pay attention to the order of packaging
    this.tcpH.setTh_seq(dataIndex * appData.length + 1); // Set the package number to the byte stream number
    this.tcpS.setData(appData);
    this.tcpPack = new TCP_PACKET(this.tcpH, this.tcpS, this.destinAddr);

    this.tcpH.setTh_sum(CheckSum.computeChkSum(this.tcpPack));
    this.tcpPack.setTcpH(this.tcpH);

    this.timer = new UDT_Timer();
    this.task = new UDT_RetransTask(this.client, this.tcpPack);
    this.timer.schedule(this.task, 3000, 3000);

    // Send TCP data packet
    udt_send(this.tcpPack);
    this.flag = 0;

    // Wait for the ACK message
    // waitACK();
    while (this.flag == 0) ;
}
```

1.5.2 TCP_Receiver.java

```java
public void rdt_recv(TCP_PACKET recvPack) {
    // Received data packet - Check the checksum, and set the reply ACK message segment
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
        this.tcpH.setTh_ack(recvPack.getTcpH().getTh_seq());
        this.ackPack = new TCP_PACKET(this.tcpH, this.tcpS, recvPack.getSourceAddr());
        this.tcpH.setTh_sum(CheckSum.computeChkSum(this.ackPack));

        System.out.println();
        System.out.println("ACK: " + recvPack.getTcpH().getTh_seq());
        System.out.println();

        // Reply to ACK message segment
        reply(this.ackPack);

        int currentSequence = (recvPack.getTcpH().getTh_seq() - 1) / 100;
        if (currentSequence != this.lastSequence) {
            this.lastSequence = currentSequence;

            // Insert the received correct and ordered data into the data queue, preparing for delivery
            this.dataQueue.add(recvPack.getTcpS().getData());

            // Deliver data (per 20 sets of data)
            if (this.dataQueue.size() == 20)
                deliver_data();
        }
    }
}
```

The Logs show that the sender retransmits packets when timeouts occur, ensuring reliable data transfer over a channel with bit errors and packet loss:

```
50.201.164:9002 1012    98.81%  1000    5
2025-12-04 01:14:43:758 CST ACK_ack: 1
2025-12-04 01:14:43:774 CST ACK_ack: 101
2025-12-04 01:14:43:789 CST ACK_ack: 201
2025-12-04 01:14:43:804 CST ACK_ack: 301
2025-12-04 01:14:43:819 CST ACK_ack: 401
2025-12-04 01:14:43:834 CST ACK_ack: 501
2025-12-04 01:14:47:421 CST ACK_ack: 4801
2025-12-04 01:14:47:435 CST ACK_ack: 4901
2025-12-04 01:14:47:449 CST ACK_ack: -19712
2025-12-04 01:14:50:454 CST ACK_ack: 5001
```

```
CLIENT HOST TOTAL   SUC_RATIO   NORMAL  WRONG   LOSS    DELAY
10.150.201.164:9001 1026    97.47%  1012    4   10  0
    2025-12-04 01:14:43:756 CST DATA_seq: 1     ACKed
    2025-12-04 01:14:43:773 CST DATA_seq: 101       ACKed
    2025-12-04 01:14:43:788 CST DATA_seq: 201       ACKed
    2025-12-04 01:14:43:803 CST DATA_seq: 301       ACKed
    2025-12-04 01:14:43:818 CST DATA_seq: 401       ACKed
    2025-12-04 01:14:43:833 CST DATA_seq: 501       ACKed
    2025-12-04 01:14:43:848 CST DATA_seq: 601       ACKed
    2025-12-04 01:14:43:862 CST DATA_seq: 701       ACKed
    2025-12-04 01:14:44:393 CST DATA_seq: 4601  LOSS    NO_ACK
    2025-12-04 01:14:47:394 CST *Re: DATA_seq: 4601     ACKed
    2025-12-04 01:14:47:407 CST DATA_seq: 4701      ACKed
    2025-12-04 01:14:47:421 CST DATA_seq: 4801      ACKed
    2025-12-04 01:14:47:435 CST DATA_seq: 4901      ACKed
    2025-12-04 01:14:47:448 CST DATA_seq: 5001      NO_ACK
    2025-12-04 01:14:50:453 CST *Re: DATA_seq: 5001     ACKed
```

Figure 5: RDT-3.0 Sender/Receiver LOG & Code Snippet

1.6 Go-Back-N

In the Go-Back-N protocol, the sender can send multiple packets before needing an acknowledgment for the first one, but if a packet is lost or corrupted, all subsequent packets are retransmitted.

The implementation involves maintaining a sliding window of packets that can be sent without waiting for an acknowledgment. If an acknowledgment is not received for a packet within a certain time frame, the sender retransmits that packet and all subsequent packets in the window.

1.6.1 TCP_Sender.java

```java
public void rdt_send(int dataIndex, int[] appData) {
    // Generate the TCP data packet (set the sequence number, data field, and checksum), and pay attention to the order of packaging
    this.tcpH.setTh_seq(dataIndex * appData.length + 1); // Set the package number to the byte stream number
    this.tcpS.setData(appData);
    this.tcpPack = new TCP_PACKET(this.tcpH, this.tcpS, this.destinAddr);

    this.tcpH.setTh_sum(CheckSum.computeChkSum(this.tcpPack));
    this.tcpPack.setTcpH(this.tcpH);

    if (this.window.isFull()) {
        System.out.println();
        System.out.println("Sliding Window Full");
        System.out.println();

        this.flag = 0;
    }
    while (this.flag == 0) ;

    try {
        this.window.putPacket(this.tcpPack.clone());
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }

    // Send TCP data packet
    udt_send(this.tcpPack);
}
```

1.6.2 TCP_Receiver.java

```java
public void rdt_recv(TCP_PACKET recvPack) {
    // Received data packet - Check the checksum, and set the reply ACK message segment
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
        int currentSequence = (recvPack.getTcpH().getTh_seq() - 1) / 100;
        if (this.expectedSequence == currentSequence) {
```

```
6             this.tcpH.setTh_ack(recvPack.getTcpH().getTh_seq());
7             this.ackPack = new TCP_PACKET(this.tcpH, this.tcpS, recvPack.getSourceAddr());
8             this.tcpH.setTh_sum(CheckSum.computeChkSum(this.ackPack));
9
10            System.out.println();
11            System.out.println("ACK: " + recvPack.getTcpH().getTh_seq());
12            System.out.println();
13
14            // Reply to ACK message segment
15            reply(this.ackPack);
16
17            this.expectedSequence += 1;
18
19            // Insert the received correct and ordered data into the data queue, preparing for delivery
20            this.dataQueue.add(recvPack.getTcpS().getData());
21
22            // Deliver data (per 20 sets of data)
23            if (this.dataQueue.size() == 20)
24                deliver_data();
25        }
26    }
27 }
```

1.6.3 SenderSlidingWindow.java

Java

```
1  public class SenderSlidingWindow {
2      private Client client;
3      private int size = 16;
4      private int base = 0;
5      private int nextIndex = 0;
6      private TCP_PACKET[] packets = new TCP_PACKET[this.size];
7
8      private Timer timer;
9      private TaskPacketsRetransmit task;
10
11     public SenderSlidingWindow(Client client) {
12         this.client = client;
13     }
14
15     public boolean isFull() {
16         return this.size <= this.nextIndex;
17     }
18
19     public void putPacket(TCP_PACKET packet) {
20         this.packets[this.nextIndex] = packet;
21         if (this.base == this.nextIndex) {
22             this.timer = new Timer();
```

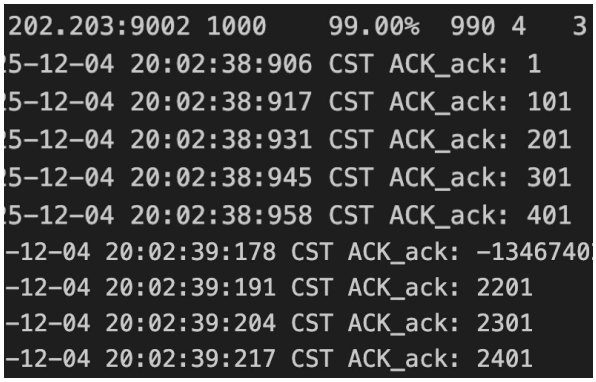
```
            this.task = new TaskPacketsRetransmit(this.client, this.packets);
            this.timer.schedule(this.task, 3000, 3000);
        }

        this.nextIndex++;
    }

    public void receiveACK(int currentSequence) {
        if (this.base <= currentSequence && currentSequence < this.base + this.size) {
            for (int i = 0; currentSequence - this.base + 1 + i < this.size; i++) {
                this.packets[i] = this.packets[currentSequence - this.base + 1 + i];
                this.packets[currentSequence - this.base + 1 + i] = null;
            }

            this.nextIndex -=currentSequence - this.base + 1;
            this.base = currentSequence + 1;

            this.timer.cancel();
            if (this.base != this.nextIndex) {
                this.timer = new Timer();
                this.task = new TaskPacketsRetransmit(this.client, this.packets);
                this.timer.schedule(this.task, 3000, 3000);
            }
        }
    }
}
```

1.6.4 TaskPacketsRetransmit.java

```java
public class TaskPacketsRetransmit extends TimerTask {
    private Client senderClient;
    private TCP_PACKET[] packets;

    public TaskPacketsRetransmit(Client client, TCP_PACKET[] packets) {
        super();
        this.senderClient = client;
        this.packets = packets;
    }

    @Override
    public void run() {
        for (TCP_PACKET packet : this.packets) {
            if (packet == null) {
                break;
            } else {
                this.senderClient.send(packet);
            }
```

```
19          }
20      }
21  }
```

The Logs show that the sender retransmits packets when timeouts occur, ensuring reliable data transfer using the Go-Back-N protocol:



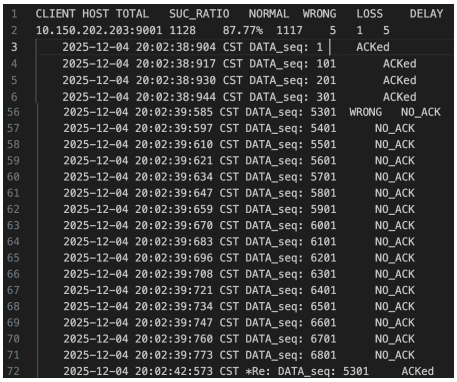


Figure 6: Go-Back-N Sender/Receiver LOG

1.7 Selective Repeat

In the Selective Repeat protocol, the sender can send multiple packets before needing an acknowledgment for the first one, and only the lost or corrupted packets are retransmitted.

The implementation involves maintaining a sliding window of packets that can be sent without waiting for an acknowledgment. If an acknowledgment is not received for a packet within a certain time frame, only that specific packet is retransmitted.

1.7.1 TCP_Receiver.java

```java
public void rdt_recv(TCP_PACKET recvPack) {
    // Received data packet - Check the checksum, and set the reply ACK message segment
    if (CheckSum.computeChkSum(recvPack) == recvPack.getTcpH().getTh_sum()) {
        int toACKSequence = -1;
        try {
            toACKSequence = this.window.receivePacket(recvPack.clone());
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }

        if (toACKSequence != -1) {
            this.tcpH.setTh_ack(toACKSequence * 100 + 1);
            this.ackPack = new TCP_PACKET(this.tcpH, this.tcpS, recvPack.getSourceAddr());
            this.tcpH.setTh_sum(CheckSum.computeChkSum(this.ackPack));

            // Reply to ACK message segment
```

```
            reply(this.ackPack);
        }
    }
}

public void deliver_data() { }
```

1.7.2 SenderSlidingWindow.java

```java
public class SenderSlidingWindow {
    private Client client;
    private int size = 16;
    private int base = 0;
    private int nextIndex = 0;
    private TCP_PACKET[] packets = new TCP_PACKET[this.size];
    private UDT_Timer[] timers = new UDT_Timer[this.size];

    public SenderSlidingWindow(Client client) {
        this.client = client;
    }

    public boolean isFull() {
        return this.size <= this.nextIndex;
    }

    public void putPacket(TCP_PACKET packet) {
        this.packets[this.nextIndex] = packet;
        this.timers[this.nextIndex] = new UDT_Timer();
        this.timers[this.nextIndex].schedule(new UDT_RetransTask(this.client, packet), 3000, 3000);

        this.nextIndex++;
    }

    public void receiveACK(int currentSequence) {
        if (this.base <= currentSequence && currentSequence < this.base + this.size) {
            if (this.timers[currentSequence - this.base] == null) {
                return;
            }

            this.timers[currentSequence - this.base].cancel();
            this.timers[currentSequence - this.base] = null;

            if (currentSequence == this.base) {
                int maxACKedIndex = 0;
                while (maxACKedIndex + 1 < this.nextIndex
                        && this.timers[maxACKedIndex + 1] == null) {
                    maxACKedIndex++;
```

```
            }

            for (int i = 0; maxACKedIndex + 1 + i < this.size; i++) {
                this.packets[i] = this.packets[maxACKedIndex + 1 + i];
                this.timers[i] = this.timers[maxACKedIndex + 1 + i];
            }

            for (int i = this.size - (maxACKedIndex + 1); i < this.size; i++) {
                this.packets[i] = null;
                this.timers[i] = null;
            }

            this.base += maxACKedIndex + 1;
            this.nextIndex -= maxACKedIndex + 1;
        }
    }
}
```

1.7.3 ReceiverSlidingWindow.java

Java

```
public class ReceiverSlidingWindow {
    private Client client;
    private int size = 16;
    private int base = 0;
    private TCP_PACKET[] packets = new TCP_PACKET[this.size];
    Queue<int[]> dataQueue = new LinkedBlockingQueue();

    private int counts = 0;

    public ReceiverSlidingWindow(Client client) {
        this.client = client;
    }

    public int receivePacket(TCP_PACKET packet) {
        int currentSequence = (packet.getTcpH().getTh_seq() - 1) / 100;

        if (currentSequence < this.base) {
            // ACK [base - size, base - 1]
            int left = this.base - this.size;
            int right = this.base - 1;
            if (left <= 0) {
                left = 1;
            }

            if (left <= currentSequence && currentSequence <= right) {
                return currentSequence;
```

```
27            }
28        } else if (this.base <= currentSequence && currentSequence < this.base + this.size) {
29            this.packets[currentSequence - this.base] = packet;
30
31            if (currentSequence == this.base) {
32                this.slid();
33            }
34            return currentSequence;
35        }
36        return -1;
37    }
38
39    private void slid() {
40        int maxIndex = 0;
41        while (maxIndex + 1 < this.size
42                && this.packets[maxIndex + 1] != null) {
43            maxIndex++;
44        }
45
46        for (int i = 0; i < maxIndex + 1; i++) {
47            this.dataQueue.add(this.packets[i].getTcpS().getData());
48        }
49
50        for (int i = 0; maxIndex + 1 + i < this.size; i++) {
51            this.packets[i] = this.packets[maxIndex + 1 + i];
52        }
53
54        for (int i = this.size - (maxIndex + 1); i < this.size; i++) {
55            this.packets[i] = null;
56        }
57
58        this.base += maxIndex + 1;
59
60        if (this.dataQueue.size() >= 20 || this.base == 1000) {
61            this.deliver_data();
62        }
63    }
64
65    public void deliver_data() {
66        // Check the `this.dataQueue` and write the data to the file
67        try {
68            File file = new File("recvData.txt");
69            BufferedWriter writer = new BufferedWriter(new FileWriter(file, true));
70
71            while (!this.dataQueue.isEmpty()) {
72                int[] data = this.dataQueue.poll();
73
74                // Write data to a file
```

```
75              for (int i = 0; i < data.length; i++) {
76                  writer.write(data[i] + "\n");
77              }
78
79              writer.flush(); // Clear out Caches
80          }
81
82          writer.close();
83      } catch (IOException e) {
84          e.printStackTrace();
85      }
86  }
87  }
```

The Logs show that the sender retransmits only the lost or corrupted packets, ensuring reliable data transfer using the Selective Repeat protocol:

```
.203:9002 1009    99.11%  1000    4
2-04 20:34:42:309 CST ACK_ack: 1
2-04 20:34:42:324 CST ACK_ack: 101
2-04 20:34:42:338 CST ACK_ack: 201
2-04 20:34:42:353 CST ACK_ack: 301
2-04 20:34:42:367 CST ACK_ack: 401
2-04 20:34:42:382 CST ACK_ack: -103
2-04 20:34:45:387 CST ACK_ack: 501
2-04 20:34:45:402 CST ACK_ack: 601
```

```
T TOTAL   SUC_RATIO   NORMAL  WRONG   L
.203:9001 1014    98.62%  1009    2   3
2-04 20:34:42:308 CST DATA_seq: 1     A
2-04 20:34:42:323 CST DATA_seq: 101
2-04 20:34:42:337 CST DATA_seq: 201
2-04 20:34:42:352 CST DATA_seq: 301
2-04 20:34:42:367 CST DATA_seq: 401
2-04 20:34:42:381 CST DATA_seq: 501
2-04 20:34:45:387 CST *Re: DATA_seq: 50
2-04 20:34:45:401 CST DATA_seq: 601
```

Figure 7: Selective Repeat Sender/Receiver LOG

1.8 TCP

In the TCP protocol, reliable data transfer is achieved through a combination of sequence numbers, acknowledgments, checksums, timeouts, and congestion control mechanisms.

Therefore, in addition to the implementations in Selective Repeat, congestion control algorithms such as TCP Tahoe and TCP Reno are implemented to manage network congestion and optimize data transmission rates.

1.8.1 ReceiverSlidingWindow.java

```java
public int receivePacket(TCP_PACKET packet) {
    int currentSequence = (packet.getTcpH().getTh_seq() - 1) / 100;

    if (currentSequence >= this.expectedSequence) {
        putPacket(packet);
    }
```

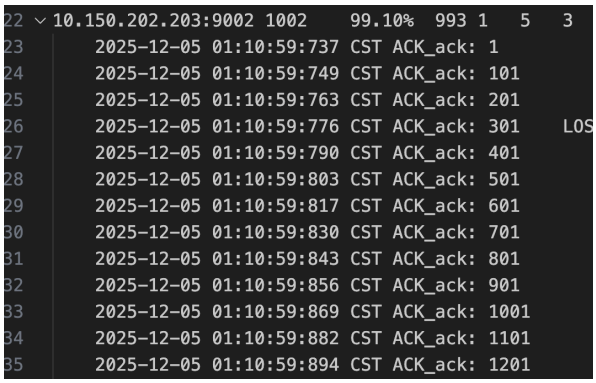
```
7       slid();
8
9       return this.expectedSequence - 1;
10  }
11
12  private void putPacket(TCP_PACKET packet) {
13      int currentSequence = (packet.getTcpH().getTh_seq() - 1) / 100;
14
15      int index = 0;
16      while (index < this.packets.size() && currentSequence > (this.packets.get(index).getTcpH().getTh_seq() - 1) / 100)
        {
17          index++;
18      }
19
20      if (index == this.packets.size() || currentSequence != (this.packets.get(index).getTcpH().getTh_seq() - 1) / 100) {
21          this.packets.add(index, packet);
22      }
23  }
24
25  private void slid() {
26      while (!this.packets.isEmpty() && (this.packets.getFirst().getTcpH().getTh_seq() - 1) / 100 ==
        this.expectedSequence) {
27          this.dataQueue.add(this.packets.poll().getTcpS().getData());
28          this.expectedSequence++;
29      }
30
31      if (this.dataQueue.size() >= 20 || this.expectedSequence == 1000) {
32          this.deliver_data();
33      }
34  }
```

1.8.2 SenderSlidingWindow.java

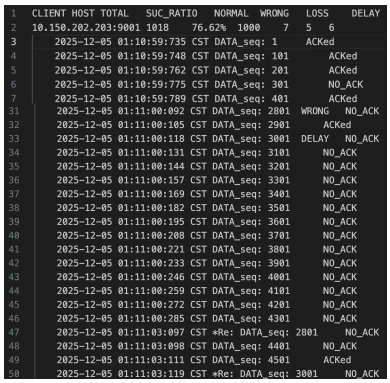
```
1   public class SenderSlidingWindow {                                                    Java
2       private Client client;
3
4       private int windowSize = 16;
5
6       private Hashtable<Integer, TCP_PACKET> packets = new Hashtable<>();
7       private Hashtable<Integer, UDT_Timer> timers = new Hashtable<>();
8
9       private int lastACKSequence = -1;
10
11      public SenderSlidingWindow(Client client) {
12          this.client = client;
13      }
14
```

```
15      public boolean isFull() {
16          return this.packets.size() >= this.windowSize;
17      }
18
19      public void putPacket(TCP_PACKET packet) {
20          int currentSequence = (packet.getTcpH().getTh_seq() - 1) / 100;
21          this.packets.put(currentSequence, packet);
22          this.timers.put(currentSequence, new UDT_Timer());
23          this.timers.get(currentSequence).schedule(new RetransmitTask(this.client, packet, this), 3000, 3000);
24      }
25
26      public void receiveACK(int currentSequence) {
27          if (currentSequence > this.lastACKSequence) {
28              for (int i = this.lastACKSequence + 1; i <= currentSequence; i++) {
29                  this.packets.remove(i);
30                  if (this.timers.containsKey(i)) {
31                      this.timers.get(i).cancel();
32                      this.timers.remove(i);
33                  }
34              }
35
36              this.lastACKSequence = currentSequence;
37              System.out.println("Window slides. Current base: " + (this.lastACKSequence + 1));
38          }
39      }
40  }
41
42  class RetransmitTask extends TimerTask {
43      private Client client;
44      private TCP_PACKET packet;
45      private SenderSlidingWindow window;
46
47      public RetransmitTask(Client client, TCP_PACKET packet, SenderSlidingWindow window) {
48          this.client = client;
49          this.packet = packet;
50          this.window = window;
51      }
52
53      @Override
54      public void run() {
55          System.out.println("Timeout! Retransmitting packet seq: " + packet.getTcpH().getTh_seq());
56          this.client.send(this.packet);
57      }
58  }
```

The Logs show that the sender and receiver correctly implement TCP mechanisms, ensuring reliable data transfer with congestion control:



```
10.150.202.203:9002 1002    99.10%  993 1   5   3
    2025-12-05 01:10:59:737 CST ACK_ack: 1
    2025-12-05 01:10:59:749 CST ACK_ack: 101
    2025-12-05 01:10:59:763 CST ACK_ack: 201
    2025-12-05 01:10:59:776 CST ACK_ack: 301    LOS
    2025-12-05 01:10:59:790 CST ACK_ack: 401
    2025-12-05 01:10:59:803 CST ACK_ack: 501
    2025-12-05 01:10:59:817 CST ACK_ack: 601
    2025-12-05 01:10:59:830 CST ACK_ack: 701
    2025-12-05 01:10:59:843 CST ACK_ack: 801
    2025-12-05 01:10:59:856 CST ACK_ack: 901
    2025-12-05 01:10:59:869 CST ACK_ack: 1001
    2025-12-05 01:10:59:882 CST ACK_ack: 1101
    2025-12-05 01:10:59:894 CST ACK_ack: 1201
```



```
CLIENT HOST TOTAL   SUC_RATIO   NORMAL  WRONG   LOSS    DELAY
10.150.202.203:9001 1018    76.62%  1000    7   5   6
    2025-12-05 01:10:59:735 CST DATA_seq: 1     ACKed
    2025-12-05 01:10:59:748 CST DATA_seq: 101       ACKed
    2025-12-05 01:10:59:762 CST DATA_seq: 201       ACKed
    2025-12-05 01:10:59:775 CST DATA_seq: 301       NO_ACK
    2025-12-05 01:10:59:789 CST DATA_seq: 401       ACKed
    2025-12-05 01:11:00:092 CST DATA_seq: 2801  WRONG   NO_ACK
    2025-12-05 01:11:00:105 CST DATA_seq: 2901      ACKed
    2025-12-05 01:11:00:118 CST DATA_seq: 3001  DELAY   NO_ACK
    2025-12-05 01:11:00:131 CST DATA_seq: 3101      NO_ACK
    2025-12-05 01:11:00:144 CST DATA_seq: 3201      NO_ACK
    2025-12-05 01:11:00:157 CST DATA_seq: 3301      NO_ACK
    2025-12-05 01:11:00:169 CST DATA_seq: 3401      NO_ACK
    2025-12-05 01:11:00:182 CST DATA_seq: 3501      NO_ACK
    2025-12-05 01:11:00:195 CST DATA_seq: 3601      NO_ACK
    2025-12-05 01:11:00:208 CST DATA_seq: 3701      NO_ACK
    2025-12-05 01:11:00:221 CST DATA_seq: 3801      NO_ACK
    2025-12-05 01:11:00:233 CST DATA_seq: 3901      NO_ACK
    2025-12-05 01:11:00:246 CST DATA_seq: 4001      NO_ACK
    2025-12-05 01:11:00:259 CST DATA_seq: 4101      NO_ACK
    2025-12-05 01:11:00:272 CST DATA_seq: 4201      NO_ACK
    2025-12-05 01:11:00:285 CST DATA_seq: 4301      NO_ACK
    2025-12-05 01:11:03:097 CST *Re: DATA_seq: 2801     NO_ACK
    2025-12-05 01:11:03:098 CST DATA_seq: 4401      NO_ACK
    2025-12-05 01:11:03:111 CST DATA_seq: 4501      ACKed
    2025-12-05 01:11:03:119 CST *Re: DATA_seq: 3001     NO_ACK
```

Figure 8: TCP Sender/Receiver LOG & Code Snippet

1.9 TCP Tahoe

In TCP Tahoe, the congestion control mechanism includes slow start, congestion avoidance, and fast retransmit. When packet loss is detected, the congestion window size is reset to one segment, and the slow start phase begins again.

The implementation involves monitoring acknowledgments and adjusting the congestion window size based on network conditions. When a timeout occurs or three duplicate ACKs are received, the congestion window size is reset, and the slow start phase is initiated.

1.9.1 SenderSlidingWindow.java

Java

```java
public void receiveACK(int currentSequence) {
    if (currentSequence == this.lastACKSequence) {
        this.lastACKSequenceCount++;
        if (this.lastACKSequenceCount == 4) {
            TCP_PACKET packet = this.packets.get(currentSequence + 1);
            if (packet != null) {
                this.client.send(packet);
                this.timers.get(currentSequence + 1).cancel();
                this.timers.put(currentSequence + 1, new UDT_Timer());
                this.timers.get(currentSequence + 1).schedule(new RetransmitTask(this.client, packet, this), 3000, 3000);
            }

            slowStart();
        }
    } else {
        for (int i = this.lastACKSequence + 1; i <= currentSequence; i++) {
            this.packets.remove(i);

            if (this.timers.containsKey(i)) {
                this.timers.get(i).cancel();
                this.timers.remove(i);
            }
        }
```

```
24
25          this.lastACKSequence = currentSequence;
26          this.lastACKSequenceCount = 1;
27
28          if (this.cwnd < this.ssthresh) {
29              this.cwnd++;
30              System.out.println("########### window expand ############");
31          } else {
32              this.count++;
33              if (this.count >= this.cwnd) {
34                  this.count -= this.cwnd;
35                  this.cwnd++;
36                  System.out.println("########### window expand ############");
37              }
38          }
39      }
40  }
41
42  public void slowStart() {
43      System.out.println("00000 cwnd: " + this.cwnd);
44      System.out.println("00000 ssthresh: " + this.ssthresh);
45      this.ssthresh = this.cwnd / 2;
46      this.cwnd = 1;
47      System.out.println("11111 cwnd: " + this.cwnd);
48      System.out.println("11111 ssthresh: " + this.ssthresh);
49  }
```

The Logs show that the sender correctly implements TCP Tahoe congestion control mechanisms, ensuring reliable data transfer with congestion management:

```
10.150.201.25:9002  982 98.78%  970 3   2   7
    2025-12-05 20:37:40:481 CST ACK_ack: 1
    2025-12-05 20:37:40:495 CST ACK_ack: 101
    2025-12-05 20:37:40:509 CST ACK_ack: 201
    2025-12-05 20:37:40:522 CST ACK_ack: 301
    2025-12-05 20:37:40:536 CST ACK_ack: 401
    2025-12-05 20:37:40:549 CST ACK_ack: 501
    2025-12-05 20:37:40:562 CST ACK_ack: 601
    2025-12-05 20:37:40:575 CST ACK_ack: 701
    2025-12-05 20:37:40:588 CST ACK_ack: 801
    2025-12-05 20:37:40:601 CST ACK_ack: 901
    2025-12-05 20:37:40:614 CST ACK_ack: 1001
    2025-12-05 20:37:40:627 CST ACK_ack: 1101
```

```
CLIENT HOST TOTAL   SUC_RATIO   NORMAL  WRONG   LOSS
10.150.201.25:9001  995 93.57%  982 3   7   3
    2025-12-05 20:37:40:480 CST DATA_seq: 1     ACKed
    2025-12-05 20:37:40:494 CST DATA_seq: 101       ACKe
    2025-12-05 20:37:40:508 CST DATA_seq: 201       ACKe
    2025-12-05 20:37:40:522 CST DATA_seq: 301       ACKe
    2025-12-05 20:37:40:535 CST DATA_seq: 401       ACKe
    2025-12-05 20:37:40:549 CST DATA_seq: 501       ACKe
    2025-12-05 20:37:40:561 CST DATA_seq: 601       ACKe
    2025-12-05 20:37:40:575 CST DATA_seq: 701       ACKe
    2025-12-05 20:37:40:588 CST DATA_seq: 801       ACKe
    2025-12-05 20:37:40:601 CST DATA_seq: 901       ACKe
    2025-12-05 20:37:40:614 CST DATA_seq: 1001      ACKe
    2025-12-05 20:37:40:627 CST DATA_seq: 1101      ACKe
```

Figure 9: TCP Tahoe Sender/Receiver LOG & Code Snippet

1.10 TCP Reno

In TCP Reno, the congestion control mechanism includes slow start, congestion avoidance, fast retransmit, and fast recovery. When packet loss is detected through three duplicate ACKs, the congestion window size is halved, and the fast recovery phase begins.

The implementation involves monitoring acknowledgments and adjusting the congestion window size based on network conditions. When three duplicate ACKs are received, the congestion window size is halved, and the fast recovery phase is initiated.

1.10.1 SenderSlidingWindow.java

```java
public void putPacket(TCP_PACKET packet) {
    int currentSequence = (packet.getTcpH().getTh_seq() - 1) / 100;
    this.packets.put(currentSequence, packet);

    if (this.timer == null) {
        this.timer = new UDT_Timer();
        this.timer.schedule(new RetransmitTask(this), 3000, 3000);
    }
}

public void receiveACK(int currentSequence) {
    if (currentSequence == this.lastACKSequence) {
        this.lastACKSequenceCount++;

        if (this.lastACKSequenceCount == 4) {
            TCP_PACKET packet = this.packets.get(currentSequence + 1);
            if (packet != null) {
                this.client.send(packet);

                if (this.timer != null) {
                    this.timer.cancel();
                }
                this.timer = new UDT_Timer();
                this.timer.schedule(new RetransmitTask(this), 3000, 300);
            }

            fastRecovery();
        } else if (this.lastACKSequenceCount > 4) {
            this.cwnd++;
            System.out.println("--- Fast Recovery Inflate: cwnd " + (this.cwnd - 1) + " -> " + this.cwnd);
        }
    } else {
        List sequenceList = new ArrayList(this.packets.keySet());
        Collections.sort(sequenceList);
        for (int i = 0; i < sequenceList.size() && (int) sequenceList.get(i) <= currentSequence; i++) {
            this.packets.remove(sequenceList.get(i));
        }

        if (this.timer != null) {
            this.timer.cancel();
        }

        if (this.packets.size() != 0) {
```

```
44              this.timer = new UDT_Timer();
45              this.timer.schedule(new RetransmitTask(this), 3000, 300);
46          }
47
48          this.lastACKSequence = currentSequence;
49          this.lastACKSequenceCount = 1;
50
51          if (this.isFastRecovery) {
52              System.out.println("--- Fast Recovery Exit ---");
53              this.cwnd = this.ssthresh;
54              this.isFastRecovery = false;
55              System.out.println("cwnd reset to ssthresh: " + this.cwnd);
56          } else {
57              if (this.cwnd < this.ssthresh) {
58                  this.cwnd++;
59                  System.out.println("########### Slow Start: window expand ############");
60              } else {
61                  this.count++;
62                  if (this.count >= this.cwnd) {
63                      this.count -= this.cwnd;
64                      this.cwnd++;
65                      System.out.println("########### Congestion Avoidance: window expand ############");
66                  }
67              }
68          }
69      }
70  }
71
72  public void slowStart() {
73      System.out.println("--- Slow Start ---");
74      System.out.println("00000 cwnd: " + this.cwnd);
75      System.out.println("00000 ssthresh: " + this.ssthresh);
76
77      this.ssthresh = this.cwnd / 2;
78      if (this.ssthresh < 2) {
79          this.ssthresh = 2;
80      }
81      this.cwnd = 1;
82      this.isFastRecovery = false;
83
84      System.out.println("11111 cwnd: " + this.cwnd);
85      System.out.println("11111 ssthresh: " + this.ssthresh);
86  }
87
88  public void fastRecovery() {
89      System.out.println("--- Fast Recovery ---");
90      System.out.println("00000 cwnd: " + this.cwnd);
91      System.out.println("00000 ssthresh: " + this.ssthresh);
```

```
92
93          this.ssthresh = this.cwnd / 2;
94          if (this.ssthresh < 2) {
95              this.ssthresh = 2;
96          }
97
98          this.cwnd = this.ssthresh;
99          this.isFastRecovery = true;
100
101         System.out.println("11111 cwnd: " + this.cwnd);
102         System.out.println("11111 ssthresh: " + this.ssthresh);
103     }
104
105     public void retransmit() {
106         this.timer.cancel();
107
108         List sequenceList = new ArrayList(this.packets.keySet());
109         Collections.sort(sequenceList);
110
111         for (int i = 0; i < this.cwnd && i < sequenceList.size(); i++) {
112             TCP_PACKET packet = this.packets.get(sequenceList.get(i));
113             if (packet != null) {
114                 System.out.println("retransmit: " + (packet.getTcpH().getTh_seq() - 1) / 100);
115                 this.client.send(packet);
116             }
117         }
118
119         if (this.packets.size() != 0) {
120             this.timer = new UDT_Timer();
121             this.timer.schedule(new RetransmitTask(this), 3000, 3000);
122         } else {
123             System.out.println("0000000000000000000 no packet");
124         }
125     }
```

The Logs show that the sender correctly implements TCP Reno congestion control mechanisms, ensuring reliable data transfer with congestion management:

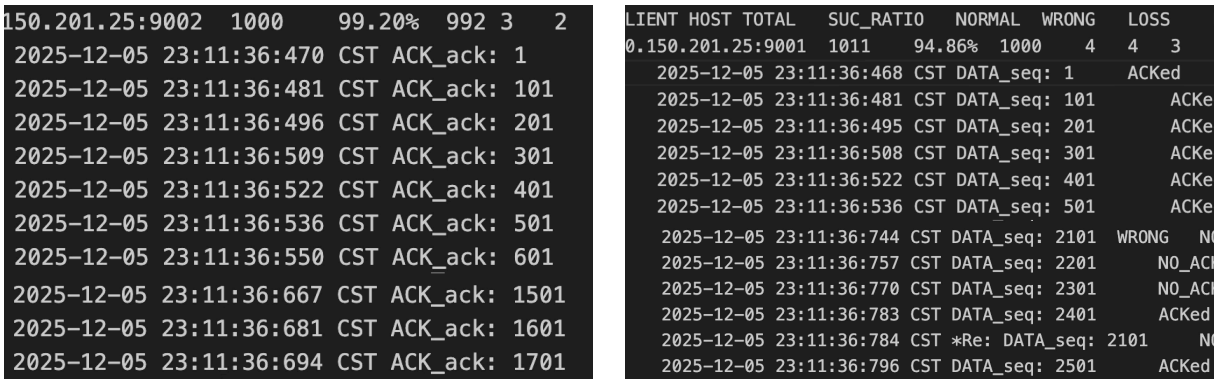


Figure 10: TCP Reno Sender/Receiver LOG & Code Snippet

2 Incomplete projects, indicating key difficulties in completion and possible solutions.

Based on the current progress, all required protocols (RDT 1.0 through TCP Reno) have been implemented. However, compared to a full-featured industrial TCP implementation, the current **TCP Reno** implementation has room for refinement, specifically regarding **Selective Acknowledgment (SACK)** and **RTT estimation**.

1. TCP NewReno / SACK (Not Implemented)
    1. **Difficulty**: The current Reno implementation falls back to retransmitting only the packet triggered by 3-duplicate ACKs or Timeout. If multiple packets are lost in a single window, Reno performance degrades significantly (often resulting in a timeout). Implementing SACK requires changing the TCP header structure to carry blocks of received bytes and modifying the sender's data structure to maintain a "scoreboard" of acknowledged gaps.
    2. **Proposed Solution:** Extend the TCP_HEADER class to support options fields. On the receiver side, track non-contiguous received blocks. On the sender side, modify the retransmission logic to prioritize gaps indicated by SACK blocks rather than just the base sequence.
2. Dynamic RTO Calculation (Karn's Algorithm)
    1. **Difficulty:** The current implementation uses a hardcoded timeout interval (3000ms). In a real network, RTT varies. Implementing dynamic RTO requires measuring SampleRTT, calculating EstimatedRTT and DevRTT. The difficulty lies in accurately matching sent packets with their ACKs when retransmissions occur (the retransmission ambiguity problem).
    2. **Proposed Solution:** Implement Karn's Algorithm. Timestamp packets upon sending. Update RTT estimates only for non-retransmitted packets. Use the formula $\text{RTO} = \text{EstimatedRTT} + 4 \times \text{DevRTT}$.

3 Explain the advantages or problems of using iterative development in the experiment process.

Advantages:

1. **Reduced Cognitive Load:** The experiment starts with an ideal channel (RDT 1.0) and incrementally adds channel impairments (bit errors *to* packet loss). This allows focusing on one problem at a time. For instance, we solved data corruption using Checksum in RDT 2.0 before worrying about sequence numbers for duplicates in RDT 2.1.
2. **Simplified Debugging:** If a bug appears in RDT 3.0 (Timeout), we can be reasonably confident that the Checksum logic (inherited from RDT 2.0) and Sequence Number logic (inherited from RDT 2.1) are correct. This isolation makes troubleshooting significantly faster.
3. **Code Reusability:** Core classes like CheckSum, TCP_PACKET, and TCP_HEADER were defined once and reused across all versions. The structure of rdt_send and rdt_recv evolved naturally, making the transition from Stop-and-Wait to Sliding Window (GBN/SR) a structural update rather than a complete rewrite.

Problems:

1. **Structural Refactoring Costs** Moving from RDT 3.0 (Stop-and-Wait) to GBN/SR (Pipelining) required a major architectural change. The simple state machine of "wait for ACK" had to be replaced by buffering queues, window variables (base, nextSeqNum), and multiple timer management. This "jump" was much harder than the incremental steps between 2.0 and 2.2.
2. **Legacy Code Overhead:** In early iterations, some temporary variables or print statements might be left over. As the logic becomes more complex (e.g., adding Congestion Control to SR), keeping the code clean and ensuring old logic doesn't interfere with new state variables (like cwnd) requires constant vigilance.

4 Summarize the main problems that have been solved in the process of completing the big homework and the corresponding solutions taken by myself.

1. **Problem 1: Object Reference vs. Deep Copy in Buffering**
   - **Issue:** In the Go-Back-N and TCP sender implementation, when putting a packet into the SenderSlidingWindow, I initially passed the tcpPack object directly. Since Java passes references by value, modifying the packet for retransmission (or if the application layer modified the array) sometimes caused inconsistency in the buffered packets.
   - **Solution:** As shown in the code snippet for GBN, I used this.window.putPacket(this.tcpPack.clone());. Implementing the Cloneable interface for TCP_PACKET ensured that the sliding window stored a snapshot of the packet as it was when sent, preventing unintended side effects.

2. **Problem 2: Timer Management in Selective Repeat**

- **Issue:** In SR, each packet needs its own logical timer. Creating a new Java `Timer` thread for every single packet caused high resource consumption and race conditions when cancelling timers during rapid ACK bursts.
- **Solution:** I utilized the provided `UDT_Timer` wrapper efficiently. Instead of creating global timers, I maintained an array of timers `timers[]` corresponding to the sequence numbers in the window. Crucially, in `receiveACK`, I added checks `if (this.timers[...] != null)` before cancelling to avoid `NullPointerException` when duplicate ACKs arrived.

3. **Problem 3: Fast Recovery Logic in TCP Reno**
   - **Issue:** Initially, my Reno implementation simply halved `cwnd` upon 3-dup ACKs but failed to implement the "Window Inflation" (adding 3 to `cwnd` and incrementing for subsequent dup ACKs). This caused the throughput to drop more than necessary, resembling Tahoe behavior.
   - **Solution:** I introduced a boolean flag `isFastRecovery`.
     - When `lastACKSequenceCount == 4`: set `ssthresh = cwnd / 2`, `cwnd = ssthresh + 3`, set `isFastRecovery = true`, and retransmit.
     - When `lastACKSequenceCount > 4`: `cwnd++` (inflate window).
     - When a **new** ACK arrives: if `isFastRecovery` is true, set `cwnd = ssthresh` (deflate window) and set `isFastRecovery = false`.

4. **Problem 4: Integer Division in Congestion Avoidance**
   - **Issue:** In Congestion Avoidance, `cwnd` should increase by $\frac{1}{\text{cwnd}}$ per ACK. Since `cwnd` is an integer, `1 / cwnd` becomes 0.
   - **Solution:** I used a counter variable `count`. `count` increments by 1 for each ACK. When `count >= cwnd`, I increment `cwnd` by 1 and reset `count`. This simulates the linear growth of adding 1 MSS per RTT.

5 Ask questions or suggestions about the experimental system.

Suggestions:

1. **Visualization Interface:** It would be very helpful if the system provided a graphical visualization of the packet flow, specifically showing the Sliding Window moving in real-time. Debugging purely via text logs (as shown in the figures) is effective but hard to intuitively grasp the timing of "flight" packets.
2. **Jitter Simulation:** The current channel simulates loss and bit errors well. Adding "Jitter" (random variation in delay) would make the RTT estimation and Timeout settings more challenging and realistic, better highlighting the need for dynamic RTO.