**A**

**Project Report**

**On**

**“Implementation and Analysis of Bi-directional Stop & Wait and Selective Repeat Protocols”**



**Department of Computer Science & Engineering**

**NATIONAL INSTITUTE OF TECHNOLOGY PATNA**

**University Campus, Bihar – 800005**

**Submitted By:**

|  |  |
| --- | --- |
| **Name** | **Roll No.** |
| **Gobinda Prasad Bag** | **2206005** |
| **Chandan Kumar** | **2206018** |
| **Rai Chirag Kumar** | **2206040** |

**Course Code: CS44113**

**Course Title: Computer Networks**

**Branch: B. Tech CSE - IVS**

**TABLE OF CONTENTS**

**S.No. Contents Page No.**

**1. Introduction 3**

**2. Code 13**

**3. Output 19**

**4. Observation 27**

**5. Conclusion 30**

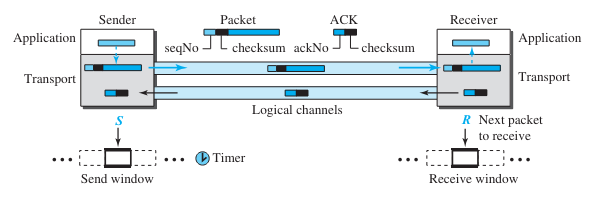
**6. Learning Outcomes 32**

**7. References 33**

**AIM OF THE EXPERIMENT**

Design and implement Bi-directional Stop & wait and Selective Repeat and demonstrate the following case studies:

1. Frame lost b. Acknowledgement lost.
2. **INTRODUCTION**
   1. **Stop and Wait**

**** Stop-and-Wait protocol is connection-oriented protocol, which uses both flow and error control. Both the sender and the receiver use a sliding window of size 1. The sender sends one packet at a time and waits for an acknowledgment before sending the next one. To detect corrupted packets, we need to add a checksum to each data packet. When a packet arrives at the receiver site, it is checked. If its checksum is incorrect, the packet is corrupted and silently discarded. The silence of the receiver is a signal for the sender that a packet was either corrupted or lost. Every time the sender sends a packet, it starts a timer. If an acknowledgment arrives before the timer expires, the timer is stopped and the sender sends the next packet (if it has one to send). If the timer expires, the sender resends the previous packet, assuming that the packet was either lost or corrupted. This means that the sender needs to keep a copy of the packet until its acknowledgment arrives. Figure 1.1 shows the outline for the Stop-and-Wait protocol. Note that only one packet and one acknowledgment can be in the channels at any time.

**Fig 1.1**

**Sender**

The sender is initially in the ready state, but it can move between the ready and blocking state. The variable S is initialized to 0.

**❑ Ready state.** When the sender is in this state, it is only waiting for one event to occur. If a request comes from the application layer, the sender creates a packet with the sequence number set to S. A copy of the packet is stored, and the packet is sent. The sender then starts the only timer. The sender then moves to the blocking state.

**❑ Blocking state.** When the sender is in this state, three events can occur:

a. If an error-free ACK arrives with the ackNo related to the next packet to be sent, which means ackNo = (S + 1) modulo 2, then the timer is stopped. The window slides, S = (S + 1) modulo 2. Finally, the sender moves to the ready state.

b. If a corrupted ACK or an error-free ACK with the ackNo ≠ (S + 1) modulo 2 arrives, the ACK is discarded.

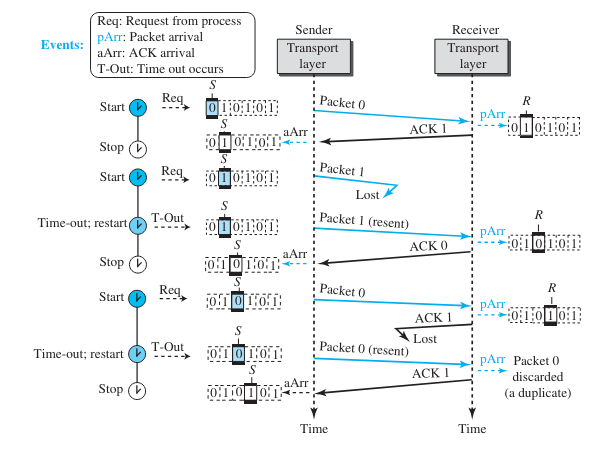
c. If a time-out occurs, the sender resends the only outstanding packet and restarts the timer.

**Receiver**

The receiver is always in the ready state. Three events may occur: a. If an error-free packet with seqNo = R arrives, the message in the packet is delivered to the application layer. The window then slides, R = (R + 1) modulo 2. Finally an ACK with ackNo = R is sent. b. If an error-free packet with seqNo ≠ R arrives, the packet is discarded, but an ACK with ackNo = R is sent. c. If a corrupted packet arrives, the packet is discarded.

**Efficiency**

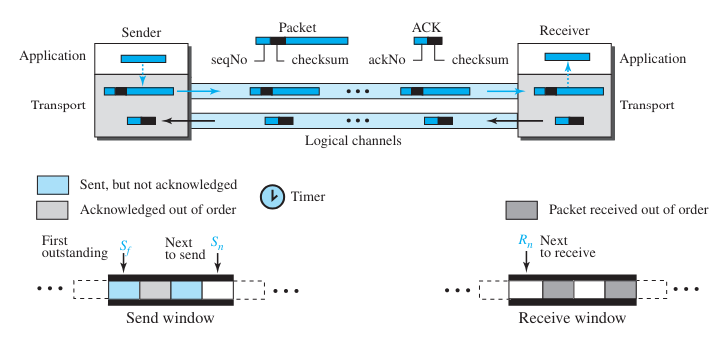
The Stop-and-Wait protocol is very inefficient if our channel is thick and long. By thick, we mean that our channel has a large bandwidth (high data rate); by long, we mean the round-trip delay is long. The product of these two is called the **bandwidth-delay** product. We can think of the channel as a pipe. The bandwidth-delay product then is the volume of the pipe in bits. The pipe is always there. It is not efficient if it is not used. The bandwidth-delay product is a measure of the number of bits a sender can transmit through the system while waiting for an acknowledgment from the receiver.



**Fig 1.2**

* 1. **Selective Repeat**

The Selective-Repeat (SR) protocols, resends only selective packets, those that are actually lost. The outline of this protocol is shown in Figure 1.3



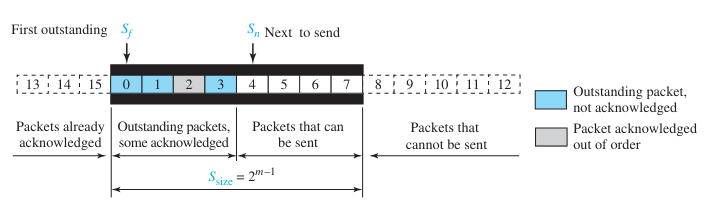
**Fig 1.3**

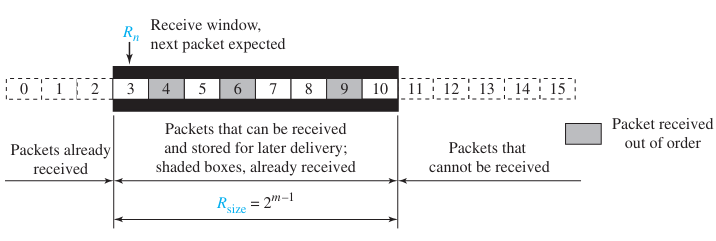
The Selective-Repeat protocol also uses two windows: a send window and a receive window. However, there are differences between the windows in this protocol and the ones in Go-Back-N. First, the maximum size of the send window is much smaller; it is 2m−1. Second, the receive window is the same size as the send window.

The send window maximum size can be 2m−1. For example, if m = 4, the sequence numbers go from 0 to 15, but the maximum size of the window is just 8 (it is 15 in the Go-Back-N Protocol). We show the Selective-Repeat send window in Figure 1.4 to emphasize the size.

The size of the receive window is the same as the size of the send window (maximum 2m−1). The Selective-Repeat protocol allows as many packets as the size of the receive window to arrive out of order and be kept until there is a set of consecutive packets to be delivered to the application layer. Because the sizes of the send window and receive window are the same, all the packets in the send packet can arrive out of order and be stored until they can be delivered. We need, however, to emphasize that in a reliable protocol the receiver never delivers packets out of order to the application layer. Figure 1.5 shows the receive window in Selective-Repeat. Those slots inside the

window that are shaded define packets that have arrived out of order and are waiting for the earlier transmitted packet to arrive before delivery to the application layer.

****

**Fig 1.4**

**Fig 1.5**

**Sender**

The sender starts in the ready state, but later it can be in one of the two states: ready or blocking. The following shows the events and the corresponding actions in each state.

**❑ Ready state.** Four events may occur in this case:

a. If a request comes from the application layer, the sender creates a packet with the sequence number set to Sn. A copy of the packet is stored, and the packet is sent. If the timer is not running, the sender starts the timer. The value of Sn is now incremented, Sn = (Sn + 1) modulo 2m. If the window is full, Sn = (Sf + S size) modulo 2m, the sender goes to the blocking state.

b. If an error-free ACK arrives with ackNo related to one of the outstanding packets, that packet is marked as acknowledged. If the ackNo = Sf, the window slides to the right until the Sf points to the first unacknowledged packet (all consecutive acknowledged packets are now outside the window). If there are outstanding packets, the timer is restarted; otherwise, the timer is stopped.

c. If a corrupted ACK or an error-free ACK with ackNo not related to an outstanding packet arrives, it is discarded.

d. If a time-out occurs, the sender resends all unacknowledged packets in the window and restarts the timer.

**❑ Blocking state.** Three events may occur in this case:

a. If an error-free ACK arrives with ackNo related to one of the outstanding packets, that packet is marked as acknowledged. In addition, if the ackNo = Sf, the window is slid to the right until the Sf points to the first unacknowledged packet (all consecutive acknowledged packets are now outside the window). If the window has slid, the sender moves to the ready state.

b. If a corrupted ACK or an error-free ACK with the ackNo not related to outstanding packets arrives, the ACK is discarded.

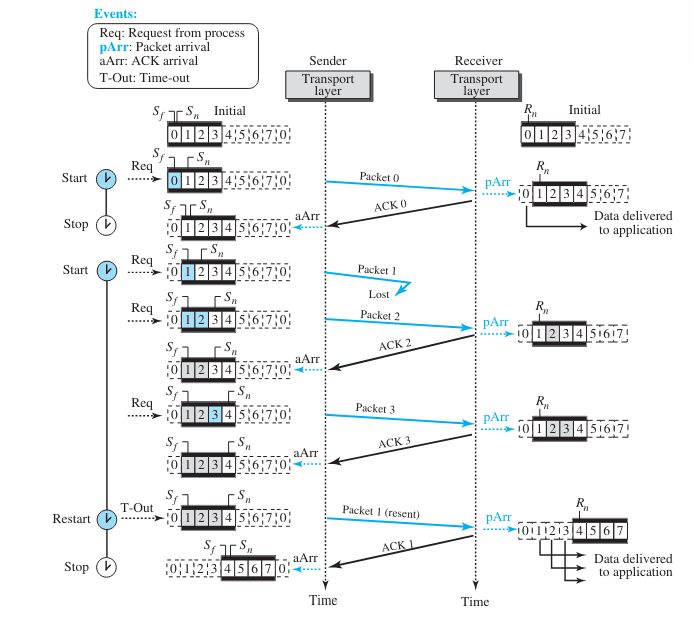
c. If a time-out occurs, the sender resends all unacknowledged packets in the window and restarts the timer.

**Receiver** The receiver is always in the ready state. Three events may occur:

a. If an error-free packet with seqNo in the window arrives, the packet is stored and an ACK with ackNo = seqNo is sent. In addition, if the seqNo = Rn, then the packet and all previously arrived consecutive packets are delivered to the application layer and the window slides so that the Rn points to the first empty slot.

b. If an error-free packet with seqNo outside the window arrives, the packet is discarded, but an ACK with ackNo = Rn is returned to the sender. This is needed to let the sender slide its window if some ACKs related to packets with seqNo < Rn were lost.

c. If a corrupted packet arrives, the packet is discarded.

****

**Fig 1.6**

At the sender, packet 0 is transmitted and acknowledged. Packet 1 is lost. Packets 2 and 3 arrive out of order and are acknowledged. When the timer times out, packet 1 (the only unacknowledged packet) is resent and is acknowledged. The send window then slides.

At the receiver site we need to distinguish between the acceptance of a packet and its delivery to the application layer. At the second arrival, packet 2 arrives and is stored and marked (shaded slot), but it cannot be delivered because packet 1 is missing. At the next arrival, packet 3 arrives and is marked and stored, but still none of the packets can be delivered. Only at the last arrival, when finally, a copy of packet 1 arrives, can packets 1, 2, and 3 be delivered to the application layer. There are two conditions for the delivery of packets to the application layer: First, a set of consecutive packets must have arrived. Second, the set starts from the beginning of the window. After the first arrival, there was only one packet and it started from the beginning of the window. After the last arrival, there are three packets and the first one starts from the beginning of the window. The key is that a reliable transport layer promises to deliver packets in order.

**1.3 Bi-directional Flow Control**

In previous protocols, data frames were typically transmitted in one direction, necessitating separate channels for data and acknowledgment traffic. However, achieving full-duplex transmission can be accomplished by running two instances of the same protocol, each utilizing a separate link for simplex data traffic in different directions. This setup often leads to underutilization of the reverse channel's capacity.

A more efficient approach involves using the same link for data transmission in both directions. By intermixing data frames and acknowledgment frames on the same link, the receiver can distinguish between them based on the kind field in the frame header. This model improves channel utilization compared to having two separate physical links.

An even more optimized method is piggybacking, where the receiver delays sending separate acknowledgment frames and instead attaches them to outgoing data frames. This technique reduces the number of frames sent, resulting in lighter processing loads at the receiver and better use of available channel bandwidth. The piggyback field in the frame header typically incurs minimal overhead, making it a cost-effective solution.

However, piggybacking introduces a complication regarding how long the data link layer should wait for a packet onto which to piggyback the acknowledgment. If the wait exceeds the sender's timeout period, it risks triggering unnecessary frame retransmissions. While an ideal solution would involve predicting when the next network layer packet will arrive, in reality, the data link layer resorts to ad hoc schemes, such as waiting a fixed duration before sending a separate acknowledgment frame if no new packet arrives within that time period.

**1.4 Bidirectional Stop & Wait Protocol:**

A bidirectional stop-and-wait protocol is a communication protocol used in computer networks where data can be transmitted in both directions between two communicating parties. In this protocol:

**Stop-and-Wait Mechanism:**

* Each party takes turns sending a single data frame and then waits for an acknowledgment (ACK) from the other party before sending the next frame.
* This ensures that data is transmitted reliably and in sequence, with minimal risk of data loss or corruption.

**Bidirectional Communication:**

* Both parties, often referred to as sender and receiver, can initiate data transmission independently.
* This allows for two-way communication, where each party can send and receive data packets to and from the other.

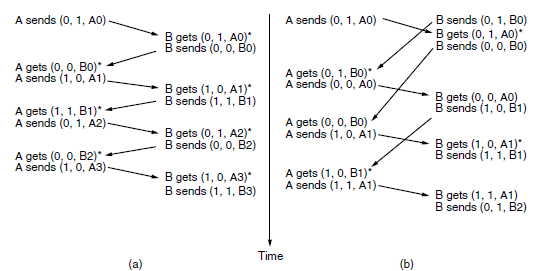
**Window Size 1:**

* The window size of 1 means that only one frame is allowed to be outstanding (unacknowledged) at any given time.
* After sending a frame, the sender waits for the corresponding acknowledgment before sending the next frame.
* This ensures simplicity and ease of implementation but may not fully utilize available bandwidth.

**Acknowledgment Handling:**

* Upon receiving a frame, the receiver sends an acknowledgment to confirm successful reception.
* If the sender does not receive an acknowledgment within a specified timeout period, it assumes that the frame was lost or corrupted and retransmits the frame.

**Reliable Data Transfer:**

* The protocol ensures reliable data transfer by confirming successful reception of each frame through acknowledgments.
* Retransmissions are employed to recover from lost or corrupted frames, ensuring data integrity.

**Fig 1.7**

**Key Components:**

* **Sender and Receiver Roles:** In the Bidirectional Stop-and-Wait ARQ protocol, both connected devices alternate between acting as the sender and receiver. This bidirectional communication allows for seamless data exchange between the devices.
* **Stop-and-Wait Mechanism:** Similar to the traditional Stop-and-Wait ARQ protocol, each device sends a single data frame and waits for an acknowledgment (ACK) from the other device before sending the next frame.
* **Acknowledgment and Retransmission:** Upon receiving a data frame, the receiving device sends an ACK to acknowledge successful reception. If the sending device does not receive an ACK within a specified timeout period, it retransmits the data frame. This process continues until the data frame is successfully acknowledged.

**Case Study**

* **When frame is lost:**

After sending the frame, the sender device waits for an acknowledgment (ACK) from the receiver device.

If the sender device does not receive an ACK within the timeout period, it retransmits the same data frame.

* **When acknowledgement is lost:**

After receiving the frame, the receiver sends the acknowledgement to the sender. If the ack is lost, then sender retransmits the frame after the timeout and the receiver will discard the duplicate frame and sends the acknowledgement with the frame.

* 1. **Bidirectional selective repeat protocol:**

In this protocol, both sender and receiver maintain a window of outstanding and acceptable sequence numbers, respectively. The sender's window size starts at 0 and grows to a predefined maximum, while the receiver's window is fixed and equal to this maximum. Each buffer in the receiver's window is associated with a bit indicating whether it's full or empty. When a frame arrives, its sequence number is checked to see if it falls within the receiver's window. If so, and if it hasn't been received before, it's accepted and stored. However, frames are only passed to the network layer after all lower-numbered frames have been correctly delivered.

Bidirectional Selective Repeat Protocol:

Bidirectional selective repeat is an enhanced version of the stop-and-wait protocol where both sender and receiver maintain a window of outstanding sequence numbers. Unlike stop-and-wait, selective repeat allows the sender to transmit multiple frames before waiting for acknowledgments. The receiver acknowledges each frame individually, allowing the sender to retransmit only the frames that are not acknowledged within a specified timeout period. This protocol improves bandwidth utilization and efficiency compared to stop-and-wait, especially in scenarios with higher network latency or variable traffic conditions.

**Window Management:**

* Both sender and receiver maintain a window of outstanding and acceptable sequence numbers, respectively.
* Sender's window size starts at 0 and grows to a predetermined maximum, while receiver's window remains fixed at the maximum size.
* Each receiver buffer corresponds to a sequence number within its window, marked by a bit indicating buffer status (full or empty).

**Frame Reception and Processing:**

* Frames are accepted and stored by the receiver if their sequence numbers fall within the window and have not already been received.
* Reception of frames is independent of whether they contain the next expected packet for the network layer.
* Frames are only passed to the network layer once all lower-numbered frames have been correctly delivered.

**Error Handling:**

* In case of frame losses or corruption, the sender eventually times out and retransmits the lost frame.
* However, retransmissions may lead to protocol failures if window management is not appropriately handled.

**Window Management Adjustment:**

* To prevent overlap after window advancement, the maximum window size should be at most half the range of sequence numbers.
* This ensures unambiguous distinction between retransmitted and new frames, preventing protocol failures.

**Buffer and Timer Management:**

* The number of buffers and timers needed is equal to the window size, not the range of sequence numbers.
* Each buffer and timer is associated with a specific sequence number, facilitating efficient protocol operation.

**Error Recovery Strategies:**

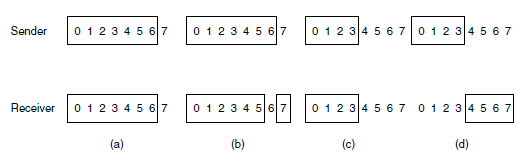
* The protocol employs strategies such as auxiliary timers, NAK (Negative Acknowledgement) frames, and resynchronization mechanisms to recover from errors and protocol failures.
* NAK frames are sent by the receiver to request retransmission of damaged or missing frames, enhancing reliability.

**Timeout and Retransmission Handling:**

* Timers associated with frames ensure timely retransmission of lost frames, while avoiding unnecessary retransmissions through appropriate timeout settings.
* The sender adjusts its timer based on the expected acknowledgment interval, optimizing retransmission efficiency.

**Determining Timeout Causes:**

* In complex scenarios with multiple outstanding frames, determining which frame caused a timeout may be challenging.
* The protocol employs administrative mechanisms to track the oldest timed-out frame for efficient error handling.



**Fig 1.8**

The bidirectional stop-and-wait protocol involves both sender and receiver maintaining windows of outstanding and acceptable sequence numbers, respectively. The sender's window starts at 0 and grows to a maximum, while the receiver's window remains fixed at the predetermined maximum. Each buffer at the receiver is associated with a bit indicating its status as full or empty. Frames arriving within the receiver's window are accepted regardless of their packet order, although they are only passed to the network layer once preceding frames have been delivered correctly.

However, non-sequential reception presents challenges. For instance, if acknowledgments are lost due to an external event like a lightning strike, the receiver may erroneously accept retransmitted frames as new ones, leading to protocol failure. This problem arises when the receiver's new window overlaps with the old one, making it impossible to distinguish between duplicates and new frames.

To address this, the maximum window size should be no more than half the sequence number range to prevent overlap. Each buffer at the receiver corresponds to the window size, and timers are associated with each buffer for retransmission purposes. Additionally, an auxiliary timer ensures a minimum rate of acknowledgments, independent of reverse traffic levels.

Protocol 6 also introduces a more efficient error-handling strategy. Upon suspecting an error, the receiver sends a negative acknowledgment (NAK) to request retransmission of the suspected frame. To avoid multiple requests for the same lost frame, the receiver keeps track of NAKs sent. If a NAK is lost, the sender eventually times out and retransmits the missing frame. The sender adjusts its timer based on the variability of acknowledgment intervals, optimizing retransmission efficiency.

Determining which frame caused a timeout is a challenge in Protocol 6, as it is not trivially determined by the oldest frame. Sophisticated mechanisms are required for timeout administration to ensure accurate frame retransmissions. Overall, Protocol 6 enhances reliability and efficiency in bidirectional stop-and-wait communication by addressing non-sequential reception and optimizing error-handling mechanisms.

**CODE**

**2.1 Code to implement Bidirectional stop and wait protocol**

import java.util.Random;

public class BidirectionalStopAndWaitARQ {

static int totalFrames = 5; // Total number of frames to be sent

static int Sf\_A = 0; // Sender A's frame sequence number

static int Sf\_B = 0; // Sender B's frame sequence number

static int Rn\_A = 0; // Receiver A's expected frame sequence number

static int Rn\_B = 0; // Receiver B's expected frame sequence number

static boolean Ack\_A = true; // Acknowledgement status for Sender A

static boolean Ack\_B = true; // Acknowledgement status for Sender B

static Random random = new Random();

public static void main(String[] args) {

while (Rn\_A < totalFrames || Rn\_B < totalFrames) {

// Sender A sends frame to Receiver B

if (Sf\_A < totalFrames && Ack\_A) {

SendFrameAtoB();

Ack\_A = false;

Sf\_A++;

sleep(1000); // Introduce a delay of 1 second

}

// Receiver B receives frame from Sender A

if (Rn\_A < totalFrames) {

if (random.nextDouble() < 0.9) { // Simulate frame loss with 10% probability

ReceiveFrameAfromB();

Rn\_A++;

Ack\_A = true;

sleep(1000); // Introduce a delay of 1 second

} else {

System.out.println("Receiver B: Frame " + Rn\_A + " from Sender A lost");

}

}

// Sender B sends frame to Receiver A

if (Sf\_B < totalFrames && Ack\_B) {

SendFrameBtoA();

Ack\_B = false;

Sf\_B++;

sleep(1000); // Introduce a delay of 1 second

}

// Receiver A receives frame from Sender B

if (Rn\_B < totalFrames) {

if (random.nextDouble() < 0.9) { // Simulate frame loss with 10% probability

ReceiveFrameBfromA();

Rn\_B++;

Ack\_B = true;

sleep(1000); // Introduce a delay of 1 second

} else {

System.out.println("Receiver A: Frame " + Rn\_B + " from Sender B lost");

}

}

}

}

static void SendFrameAtoB() {

System.out.println("Sender A: Sending frame " + Sf\_A + " to Receiver B");

}

static void ReceiveFrameAfromB() {

System.out.println("Receiver B: Received frame " + Rn\_A + " from Sender A");

if (random.nextDouble() < 0.9) { // Simulate acknowledgment loss with 10% probability

SendACKtoA();

sleep(1000); // Introduce a delay of 1 second

} else {

System.out.println("Receiver B: ACK for frame " + Rn\_A + " lost");

}

}

static void SendACKtoA() {

System.out.println("Receiver B: Sending ACK for frame " + Rn\_A + " to Sender A");

}

static void SendFrameBtoA() {

System.out.println("Sender B: Sending frame " + Sf\_B + " to Receiver A");

}

static void ReceiveFrameBfromA() {

System.out.println("Receiver A: Received frame " + Rn\_B + " from Sender B");

if (random.nextDouble() < 0.9) { // Simulate acknowledgment loss with 10% probability

SendACKtoB();

sleep(1000); // Introduce a delay of 1 second

} else {

System.out.println("Receiver A: ACK for frame " + Rn\_B + " lost");

}

}

static void SendACKtoB() {

System.out.println("Receiver A: Sending ACK for frame " + Rn\_B + " to Sender B");

}

static void sleep(int milliseconds) {

try {

Thread.sleep(milliseconds);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

**2.2 Code to implement Bidirectional selective repeat protocol:**

import java.util.Random;

public class BidirectionalSelectiveRepeatARQ {

static int Sw = (int) Math.pow(2, 2) - 1; // Window Size = 2^m - 1

static int Sf = 0;

static int Sn = 0;

static int Rn = 0;

static boolean NakSent = false;

static boolean AckNeeded = false;

static boolean[] Marked = new boolean[Sw + 1];

public static void main(String[] args) {

while (true) {

waitForEvent();

if (event("RequestToSend")) { // There is a packet to send

if (Sn - Sf >= Sw)

Sleep(); // If window is full

GetData();

MakeFrame(Sn, Rn);

StoreFrame(Sn, Rn);

SendFrame(Sn, Rn);

System.out.println("Sender A : Data sent: Frame " + Sn);

System.out.println("Sender A : Frame created: Frame " + Sn);

System.out.println("Sender A : Frame stored: Frame " + Sn);

System.out.println("Sender A : Frame sent: Frame " + Sn + " to Receiver B");

StartTimer(Sn);

Sn = Sn + 1;

}

if (event("Arrival Notification")) {

Receive();

if (FrameType() == "NAK") {

if (corrupted("frame")) Sleep();

if (nakNo("frame") >= Sf && nakNo("frame") <= Sn) {

resend(nakNo("frame"));

StartTimer(nakNo("frame"));

}

}

if (FrameType() == "Data") {

if (corrupted("frame") && !NakSent) {

SendNAK(Rn);

NakSent = true;

Sleep();

}

if (ackNo("frame") >= Sf && ackNo("frame") <= Sn) {

while (Sf < ackNo("frame")) {

Purge(Sf);

StopTimer(Sf);

System.out.println("Sender A : Timer stopped. (ACK received)");

Sf = Sf + 1;

}

if(Sf==Rn && !AckNeeded){

System.out.println("Window slide forward by 1");

System.out.println("current window size:"+ Sw);

}} if ((seqNo("frame") >= Sf) && (seqNo("frame") <= Sn) && !Marked[seqNo("frame")]) {

StoreFrame(seqNo("frame"), Rn);

Marked[seqNo("frame")] = true;

while (Marked[Rn]) {

DeliverData(Rn);

Purge(Rn);

Rn = Rn + 1;

AckNeeded = true;

if(Sf==Rn && !AckNeeded){

System.out.println("Window slide forward by 1");

System.out.println("current window size:"+ Sw);

}

}

}

if ((seqNo("frame") >= Sf) && (seqNo("frame") <= Sn) && !Marked[seqNo("frame")]){

StoreFrame(seqNo("frame"), Rn);

Marked[seqNo("frame")] = true;

while (Marked[Rn]) {

DeliverData(Rn);

Purge(Rn);

Rn = Rn + 1;

AckNeeded = true;

}}}}

if (event("TimeOut(t)")) {

StartTimer(Sf);

SendFrame(Sf, Rn);

}}}

static boolean event(String event) {

// Simulating event

Random random = new Random();

return random.nextBoolean();

}

static void waitForEvent() {

// Simulating waiting for event

try {

Thread.sleep(1000); // Delaying 1 second

} catch (InterruptedException e) {

e.printStackTrace();

}}

static void Sleep() {

// Simulating sleep

try {

Thread.sleep(1000); // Delaying 1 second

} catch (InterruptedException e) {

e.printStackTrace();

}}

static void GetData() {

// Simulating getting data

System.out.println("Sender A : Getting data");

}

static void MakeFrame(int Sn, int Rn) {

// Simulating making frame

System.out.println("Sender A : Making frame");

}

static void StoreFrame(int Sn, int Rn) {

// Simulating storing frame

System.out.println("Sender A : Storing frame");

}

static void SendFrame(int Sn, int Rn) {

// Simulating sending frame

if (Sn % 2 == 0) {

System.out.println("Sender A : Sending frame: Frame " + Sn + " to Receiver B");

System.out.println("Receiver B : Frame " + Sn + " received");

Sf++;

Sw--;

System.out.println("Window Size changed: " + Sw);

if (Sn == 2) { // Simulating frame loss

System.out.println("Sender A : Frame 2 lost!");

return;

}

} else {

System.out.println("Sender B : Sending frame: Frame " + Sn + " to Receiver A");

System.out.println("Receiver A : Frame " + Sn + " received");

}

}

static void StartTimer(int Sn) {

// Simulating starting timer

System.out.println("Sender A : Starting Timer for frame " + Sn);

}

static void Receive() {

// Simulating receiving frame

if (Sn % 2 == 0) {

System.out.println("Receiver B : Frame received");

} else {

System.out.println("Receiver A : Frame received");

}

}

static boolean corrupted(String frame) {

// Simulating checking if frame is corrupted

Random random = new Random();

return random.nextBoolean();

}

static void resend(int nakNo) {

// Simulating resending frame

System.out.println("Sender A : Resending frame: " + nakNo);

}

static void SendNAK(int Rn) {

// Simulating sending NAK

if (Sn % 2 == 0) {

System.out.println("Receiver B : NAK sent");

} else {

System.out.println("Receiver A : NAK sent");

}

}

static void StopTimer(int Sn) {

// Simulating stopping timer

System.out.println("Sender A : Timer stopped for frame " + Sn);

}

static void DeliverData(int Rn) {

// Simulating delivering data

if (Sn % 2 == 0) {

System.out.println("Receiver B : Data delivered: Frame " + Rn);

} else {

System.out.println("Receiver A : Data delivered: Frame " + Rn); }}

static void Purge(int Rn) {

// Simulating purging frame

System.out.println("Receiver B : Frame " + Rn + " purged");

}

static String FrameType() {

// Simulating frame type

Random random = new Random();

return random.nextBoolean() ? "Data" : "NAK";

}

static int nakNo(String frame) {

// Simulating NAK number

Random random = new Random();

return random.nextInt(Sw + 1);

}

static int ackNo(String frame) {

// Simulating ACK number

Random random = new Random();

return random.nextInt(Sw + 1);

}

static int seqNo(String frame) {

// Simulating sequence number

Random random = new Random();

return random.nextInt(Sw + 1);

}

static void startTimer(int frameNumber) {

System.out.println("Sender A : Starting Timer for frame " + frameNumber);

}

static void stopTimer(int frameNumber) {

System.out.println("Sender A : Timer stopped for frame " + frameNumber);

}

}

**OUTPUT**

* 1. **Bidirectional stop and wait**

**OUTPUT 1:**

Sender A: Sending frame 0 to Receiver B

Receiver B: Received frame 0 from Sender A

Receiver B: Sending ACK for frame 0 to Sender A

Sender B: Sending frame 0 to Receiver A

Receiver A: Received frame 0 from Sender B

Receiver A: Sending ACK for frame 0 to Sender B

Sender A: Sending frame 1 to Receiver B

Receiver B: Received frame 1 from Sender A

Receiver B: Sending ACK for frame 1 to Sender A

Sender B: Sending frame 1 to Receiver A

Receiver A: Received frame 1 from Sender B

Receiver A: Sending ACK for frame 1 to Sender B

Sender A: Sending frame 2 to Receiver B

Receiver B: Frame 2 from Sender A lost

Sender B: Sending frame 2 to Receiver A

Receiver A: Frame 2 from Sender B lost

Receiver B: Received frame 2 from Sender A

Receiver B: Sending ACK for frame 2 to Sender A

Receiver A: Received frame 2 from Sender B

Receiver A: Sending ACK for frame 2 to Sender B

Sender A: Sending frame 3 to Receiver B

Receiver B: Received frame 3 from Sender A

Receiver B: Sending ACK for frame 3 to Sender A

Sender B: Sending frame 3 to Receiver A

Receiver A: Received frame 3 from Sender B

Receiver A: Sending ACK for frame 3 to Sender B

Sender A: Sending frame 4 to Receiver B

Receiver B: Received frame 4 from Sender A

Receiver B: Sending ACK for frame 4 to Sender A

Sender B: Sending frame 4 to Receiver A

Receiver A: Received frame 4 from Sender B

Receiver A: Sending ACK for frame 4 to Sender B

**OUTPUT 2:**

Sender A: Sending frame 0 to Receiver B

Receiver B: Received frame 0 from Sender A

Receiver B: Sending ACK for frame 0 to Sender A

Sender B: Sending frame 0 to Receiver A

Receiver A: Received frame 0 from Sender B

Receiver A: Sending ACK for frame 0 to Sender B

Sender A: Sending frame 1 to Receiver B

Receiver B: Received frame 1 from Sender A

Receiver B: Sending ACK for frame 1 to Sender A

Sender B: Sending frame 1 to Receiver A

Receiver A: Received frame 1 from Sender B

Receiver A: Sending ACK for frame 1 to Sender B

Sender A: Sending frame 2 to Receiver B

Receiver B: Received frame 2 from Sender A

Receiver B: Sending ACK for frame 2 to Sender A

Sender B: Sending frame 2 to Receiver A

Receiver A: Received frame 2 from Sender B

Receiver A: Sending ACK for frame 2 to Sender B

Sender A: Sending frame 3 to Receiver B

Receiver B: Received frame 3 from Sender A

Receiver B: Sending ACK for frame 3 to Sender A

Sender B: Sending frame 3 to Receiver A

Receiver A: Received frame 3 from Sender B

Receiver A: Sending ACK for frame 3 to Sender B

Sender A: Sending frame 4 to Receiver B

Receiver B: Received frame 4 from Sender A

Receiver B: Sending ACK for frame 4 to Sender A

Sender B: Sending frame 4 to Receiver A

Receiver A: Received frame 4 from Sender B

Receiver A: Sending ACK for frame 4 to Sender B

**OUTPUT 3:**

Sender A: Sending frame 0 to Receiver B

Receiver B: Received frame 0 from Sender A

Receiver B: Sending ACK for frame 0 to Sender A

Sender B: Sending frame 0 to Receiver A

Receiver A: Received frame 0 from Sender B

Receiver A: Sending ACK for frame 0 to Sender B

Sender A: Sending frame 1 to Receiver B

Receiver B: Received frame 1 from Sender A

Receiver B: ACK for frame 1 lost

Sender B: Sending frame 1 to Receiver A

Receiver A: Received frame 1 from Sender B

Receiver A: Sending ACK for frame 1 to Sender B

Sender A: Sending frame 2 to Receiver B

Receiver B: Received frame 2 from Sender A

Receiver B: Sending ACK for frame 2 to Sender A

Sender B: Sending frame 2 to Receiver A

Receiver A: Received frame 2 from Sender B

Receiver A: Sending ACK for frame 2 to Sender B

Sender A: Sending frame 3 to Receiver B

Receiver B: Received frame 3 from Sender A

Receiver B: Sending ACK for frame 3 to Sender A

Sender B: Sending frame 3 to Receiver A

Receiver A: Received frame 3 from Sender B

Receiver A: Sending ACK for frame 3 to Sender B

Sender A: Sending frame 4 to Receiver B

Receiver B: Frame 4 from Sender A lost

Sender B: Sending frame 4 to Receiver A

Receiver A: Received frame 4 from Sender B

Receiver A: Sending ACK for frame 4 to Sender B

Receiver B: Received frame 4 from Sender A

Receiver B: Sending ACK for frame 4 to Sender A

* 1. **Bidirectional Selective repeat**

**OUTPUT1:**

Sender A : Starting Timer for frame 0

Sender A : Sending frame: Frame 0 to Receiver B

Receiver B : Frame 0 received

Window Size changed: 2

Sender A : Getting data

Sender A : Making frame

Sender A : Storing frame

Sender A : Sending frame: Frame 0 to Receiver B

Receiver B : Frame 0 received

Window Size changed: 1

Sender A : Data sent: Frame 0

Sender A : Frame created: Frame 0

Sender A : Frame stored: Frame 0

Sender A : Frame sent: Frame 0 to Receiver B

Sender A : Starting Timer for frame 0

Sender A : Getting data

Sender A : Making frame

Sender A : Storing frame

Sender B : Sending frame: Frame 1 to Receiver A

Receiver A : Frame 1 received

Sender A : Data sent: Frame 1

Sender A : Frame created: Frame 1

Sender A : Frame stored: Frame 1

Sender A : Frame sent: Frame 1 to Receiver B

Sender A : Starting Timer for frame 1

Sender A : Starting Timer for frame 2

Sender A : Sending frame: Frame 2 to Receiver B

Receiver B : Frame 2 received

Window Size changed: 0

Sender A : Frame 2 lost!

Sender A : Getting data

Sender A : Making frame

Sender A : Storing frame

Sender A : Sending frame: Frame 2 to Receiver B

Receiver B : Frame 2 received

Window Size changed: -1

Sender A : Frame 2 lost!

Sender A : Data sent: Frame 2

Sender A : Frame created: Frame 2

Sender A : Frame stored: Frame 2

Sender A : Frame sent: Frame 2 to Receiver B

Sender A : Starting Timer for frame 2

Sender A : Starting Timer for frame 4

Sender A : Sending frame: Frame 4 to Receiver B

Receiver B : Frame 4 received

Window Size changed: -2

Sender A : Getting data

Sender A : Making frame

Sender A : Storing frame

Sender B : Sending frame: Frame 3 to Receiver A

Receiver A : Frame 3 received

Sender A : Data sent: Frame 3

Sender A : Frame created: Frame 3

Sender A : Frame stored: Frame 3

Sender A : Frame sent: Frame 3 to Receiver B

Sender A : Starting Timer for frame 3

Receiver B : Frame received

Sender A : Getting data

Sender A : Making frame

Sender A : Storing frame

Sender A : Sending frame: Frame 0 to Receiver B

Receiver B : Frame 0 received

Window Size changed: 2

Sender A : Data sent: Frame 0

Sender A : Frame created: Frame 0

Sender A : Frame stored: Frame 0

Sender A : Frame sent: Frame 0 to Receiver B

Sender A : Starting Timer for frame 0

Sender A : Starting Timer for frame 1

Sender B : Sending frame: Frame 1 to Receiver A

Receiver A : Frame 1 received

Sender A : Starting Timer for frame 1

Sender B : Sending frame: Frame 1 to Receiver A

Receiver A : Frame 1 received

Sender A : Getting data

Sender A : Making frame

Sender A : Storing frame

Sender B : Sending frame: Frame 1 to Receiver A

Receiver A : Frame 1 received

Sender A : Data sent: Frame 1

Sender A : Frame created: Frame 1

Sender A : Frame stored: Frame 1

Sender A : Frame sent: Frame 1 to Receiver B

Sender A : Starting Timer for frame 1

Receiver B : Frame received

Sender A : Starting Timer for frame 1

Sender B : Sending frame: Frame 1 to Receiver A

Receiver A : Frame 1 received

Sender A : Getting data

Sender A : Making frame

Sender A : Storing frame

Sender A : Sending frame: Frame 2 to Receiver B

Receiver B : Frame 2 received

Window Size changed: 1

Sender A : Frame 2 lost!

Sender A : Data sent: Frame 2

Sender A : Frame created: Frame 2

Sender A : Frame stored: Frame 2

Sender A : Frame sent: Frame 2 to Receiver B

Sender A : Starting Timer for frame 2

Receiver A : Frame received

Receiver A : NAK sent

Sender A : Getting data

Sender A : Making frame

Sender A : Storing frame

**OUTPUT2:**

Sender B : Sending frame: Frame 3 to Receiver A

Receiver A : Frame 3 received

Sender A : Data sent: Frame 3

Sender A : Frame created: Frame 3

Sender A : Frame stored: Frame 3

Sender A : Frame sent: Frame 3 to Receiver B

Sender A : Starting Timer for frame 3

Receiver B : Frame received

Sender A : Starting Timer for frame 2

Sender A : Sending frame: Frame 2 to Receiver B

Receiver B : Frame 2 received

Window Size changed: 0

Sender A : Frame 2 lost!

Sender A : Getting data

Sender A : Making frame

Sender A : Storing frame

Sender A : Sending frame: Frame 4 to Receiver B

Receiver B : Frame 4 received

Window Size changed: -1

Sender A : Data sent: Frame 4

Sender A : Frame created: Frame 4

Sender A : Frame stored: Frame 4

Sender A : Frame sent: Frame 4 to Receiver B

Sender A : Starting Timer for frame 4

Receiver A : Frame received

Sender A : Starting Timer for frame 4

Sender A : Sending frame: Frame 4 to Receiver B

Receiver B : Frame 4 received

Window Size changed: -2

Sender A : Getting data

Sender A : Making frame

Sender A : Storing frame

Sender B : Sending frame: Frame 5 to Receiver A

Receiver A : Frame 5 received

Sender A : Data sent: Frame 5

Sender A : Frame created: Frame 5

Sender A : Frame stored: Frame 5

Sender A : Frame sent: Frame 5 to Receiver B

Sender A : Starting Timer for frame 5

Receiver B : Frame received

**OUTPUT3:**

Sender A : Getting data

Sender A : Making frame

Sender A : Storing frame

Sender A : Sending frame: Frame 0 to Receiver B

Receiver B : Frame 0 received

Window Size changed: 2

Sender A : Data sent: Frame 0

Sender A : Frame created: Frame 0

Sender A : Frame stored: Frame 0

Sender A : Frame sent: Frame 0 to Receiver B

Sender A : Starting Timer for frame 0

Sender A : Starting Timer for frame 1

Sender B : Sending frame: Frame 1 to Receiver A

Receiver A : Frame 1 received

Receiver A : Frame received

Sender A : Starting Timer for frame 1

Sender B : Sending frame: Frame 1 to Receiver A

Receiver A : Frame 1 received

Sender A : Getting data

Sender A : Making frame

Sender A : Storing frame

Sender B : Sending frame: Frame 1 to Receiver A

Receiver A : Frame 1 received

Sender A : Data sent: Frame 1

Sender A : Frame created: Frame 1

Sender A : Frame stored: Frame 1

Sender A : Frame sent: Frame 1 to Receiver B

Sender A : Starting Timer for frame 1

Sender A : Getting data

Sender A : Making frame

Sender A : Storing frame

Sender A : Sending frame: Frame 2 to Receiver B

Receiver B : Frame 2 received

Window Size changed: 1

Sender A : Frame 2 lost!

Sender A : Data sent: Frame 2

Sender A : Frame created: Frame 2

Sender A : Frame stored: Frame 2

Sender A : Frame sent: Frame 2 to Receiver B

Sender A : Starting Timer for frame 2

Receiver A : Frame received

Sender A : Getting data

Sender A : Making frame

Sender A : Storing frame

Sender B : Sending frame: Frame 3 to Receiver A

Receiver A : Frame 3 received

Sender A : Data sent: Frame 3

Sender A : Frame created: Frame 3

Sender A : Frame stored: Frame 3

Sender A : Frame sent: Frame 3 to Receiver B

Sender A : Starting Timer for frame 3

Receiver B : Frame received

Receiver B : NAK sent

Sender A : Starting Timer for frame 2

Sender A : Sending frame: Frame 2 to Receiver B

Receiver B : Frame 2 received

Window Size changed: 0

Sender A : Frame 2 lost!

Sender A : Getting data

Sender A : Making frame

Sender A : Storing frame

Sender A : Sending frame: Frame 4 to Receiver B

Receiver B : Frame 4 received

Window Size changed: -1

Sender A : Data sent: Frame 4

Sender A : Frame created: Frame 4

Sender A : Frame stored: Frame 4

Sender A : Frame sent: Frame 4 to Receiver B

Sender A : Starting Timer for frame 4

Receiver A : Frame received

**OBSERVATIONS**

**4.1. Flow Control Protocol Implementation:**

The project incorporated bidirectional flow control protocols to enable simultaneous data transmission in both directions, optimizing communication efficiency.

This implementation allows the sender and receiver to independently regulate data flow, preventing either side from becoming overwhelmed.

Bidirectional flow control enhances network bandwidth utilization by facilitating concurrent data transmission in both directions, thereby improving reliability and efficiency in real-world networking scenarios.

Moreover, it reduces network congestion occurrences and enhances overall communication system throughput.

In scenarios where acknowledgment is lost, bidirectional flow control streamlines the retransmission process, ensuring reliable data delivery with minimal latency.

**4.2. Case Studies Demonstration**:

Integrating flow control protocols with a bidirectional approach adds an extra layer of efficiency and reliability to the frame transfer process.

**i) Frame Lost Scenario:**

**Bidirectional Stop & Wait:**

Upon frame loss, the sender initiates retransmission after not receiving acknowledgment within the timeout period.

The sender resends the lost frame, and upon successful reception, the receiver acknowledges it, albeit with increased latency due to retransmissions.

**Bidirectional Selective Repeat:**

In the event of frame loss, only the lost frame is retransmitted, rather than the entire window of frames.

Upon receiving the retransmitted frame, the receiver acknowledges it, significantly reducing latency and ensuring efficient data delivery, particularly in high probability frame loss scenarios.

**ii) Acknowledgement Loss Scenario:**

**Bidirectional Stop & Wait:**

- If acknowledgment is lost, the sender triggers retransmission after not receiving it within the timeout period.

- The sender resends the frame, and upon successful reception, the receiver re-acknowledges it.

- This process repeats until the sender finally receives acknowledgment, achieving successful data delivery albeit with increased latency due to retransmissions.

**Bidirectional Selective Repeat:**

- In case of acknowledgment loss, the sender initiates selective retransmission of the lost acknowledgment.

- Upon receiving the retransmitted acknowledgment, the receiver continues data transfer without retransmitting the entire frame.

- This selective approach minimizes latency and ensures efficient data delivery, especially in scenarios prone to acknowledgment loss.

**4.3. Code Implementation:**

- The project's code implementation showcased a comprehensive understanding of flow control protocols and the bidirectional technique.

- Two-way communication was successfully established, and the protocols were robustly implemented using the bidirectional approach.

- The integration of protocols was seamless, resulting in more efficient and reliable transmission of data.

**4.4. Observations from Execution:**

**- Flow Control Efficiency:**

- The implemented flow control ensured optimal data transfer rates, preventing receiver or network overload.

**- Resilience to Loss:**

- The bidirectional process demonstrated resilience to loss, significantly reducing latency and ensuring efficient data delivery, especially in scenarios with a high probability of frame or acknowledgment loss.

**- Importance of Protocols:**

- The project underscored the importance of bidirectional stop & wait and selective repeat protocols in guaranteeing reliable data delivery, even in scenarios involving acknowledgment loss.

**4.5. Limitations and Future Considerations:**

**- Congestion Control Mechanisms:**

- There is an opportunity to enhance the project by implementing congestion control mechanisms. These mechanisms can effectively manage network traffic, thereby improving overall performance and reliability.

**- Performance Optimization:**

- Future iterations of the project could focus on performance optimization. Refining the protocol implementation and minimizing redundant processes can lead to more efficient data transmission and resource utilization.

**- Load Balancing Algorithms:**

- Implementing load balancing algorithms can further improve network performance. These algorithms distribute network traffic evenly across resources, helping to prevent congestion and optimize throughput.

**- Overall Insights and Future Developments:**

- While the project successfully implemented bidirectional stop-and-wait and selective repeat protocols, it provides valuable insights into bidirectional flow control and channel bandwidth considerations.

- These insights lay a solid foundation for future developments in network applications, guiding advancements in protocol design and network optimization.

**Conclusion:**

The implementation of bidirectional flow control with stop-and-wait and selective repeat protocols in this project has been a valuable learning experience, offering insights into network communication and bandwidth optimization techniques.

**Key Achievements:**

**- Successful Implementation:**

- Implemented Bidirectional Stop & Wait and Selective Repeat protocols, showcasing efficient data delivery even in scenarios of acknowledgment loss.

**- Optimized Bandwidth Utilization:**

- Demonstrated enhanced network bandwidth utilization through simultaneous data transmission in both directions, ensuring optimal use of available resources.

**- Reliability:**

- Ensured reliable data delivery, even in acknowledgment loss scenarios, by facilitating efficient retransmission processes.

- **Contributions to Knowledge:**

**- Improved Efficiency:**

- Showcased the efficiency of Bidirectional Stop & Wait and Selective Repeat protocols in data delivery, contributing to the enhancement of communication protocols.

**- Error Handling:**

- Explored potential limitations and offered insights into overcoming scalability issues and improving error handling, contributing to more robust communication systems.

**Future Directions:**

**- Performance Enhancement:**

- Implement protocols using different programming languages to assess performance and efficiency variations.

**- Scalability and Adaptability:**

- Investigate congestion control mechanisms to manage network traffic effectively, ensuring scalability and adaptability.

**- Optimization:**

- Conduct performance optimization by refining protocol implementation and minimizing redundant processes.

**- Enhanced Error Handling:**

- Implement error correction codes, such as Reed-Solomon codes, to detect and correct burst errors efficiently.

- Integrate robust error recovery mechanisms to handle various types of errors effectively.

In conclusion, the project has laid a solid foundation in bidirectional flow control, protocols, and channel utilization techniques, paving the way for future developments in efficient and reliable communication systems.

**Learning Outcome**

Through the design and implementation of the bi-directional stop & wait and selective repeat mini-project, the following learning outcomes have been achieved:

**- Understanding the Role of Flow Control:**

- Gained insights into the critical role of flow control in ensuring reliable data transmission within a network environment.

**- Familiarity with Bidirectional Flow Control and Protocols:**

- Acquired familiarity with bidirectional flow control concepts and protocols, such as stop & wait and selective repeat, for optimizing communication efficiency.

**- Experience in Protocol Implementation:**

- Developed hands-on experience in implementing bidirectional stop & wait and selective repeat protocols to facilitate efficient data transfer.

**- Enhanced Skills in Error Handling and Flow Management:**

- Improved skills in error handling and flow management within network applications, understanding the importance of two-way transmission for robust communication.

**- Application of Theoretical Knowledge:**

- Successfully applied theoretical knowledge of networking concepts to practical programming tasks, enhancing problem-solving abilities in network communication scenarios.

**- Proficiency in Java for Network Applications:**

- Enhanced proficiency in using Java for developing network applications, gaining practical experience in implementing communication protocols.

**- Understanding Data Transmission Challenges:**

- Acquired knowledge of the challenges involved in data transmission, including acknowledgment loss and network congestion, and learned strategies to address them effectively.

**- Optimization Techniques:**

- Explored potential optimizations for improving communication efficiency, such as implementing congestion control mechanisms and refining protocol implementations.

**- Appreciation of Real-World Networked Applications:**

- Developed an appreciation for the complexities and nuances of real-world networked applications, understanding the importance of efficient communication protocols in modern networking environments.

**REFERENCES**

Behrouz A. Forouzan, Data Communications And Networking (Vol.4), McGraw-Hill Forouzan Networking Series, 2007, 978-0070634145.

Andrew S. Tanenbaum, Computer Networks (Vol.4) , Pearson, 2011, 978-0130661029

Kurose James F., Ross Keith W., Computer Networking: A Top-Down Approach(Vol.6), Pearson, 2017, 978-9332585492