**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**

**2. Objective**

**3. Methodology**

**4. Flowchart/ Data Flow Diagram**

**5. Code**

**6. Output**

**7. Observation**

**8. Conclusion**

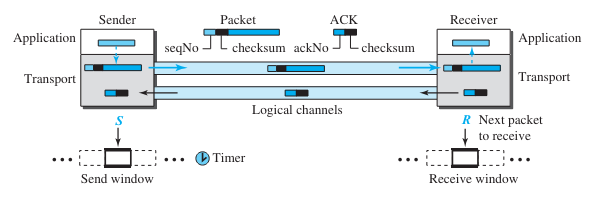
**7. Learning Outcomes**

**9. References**

**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.

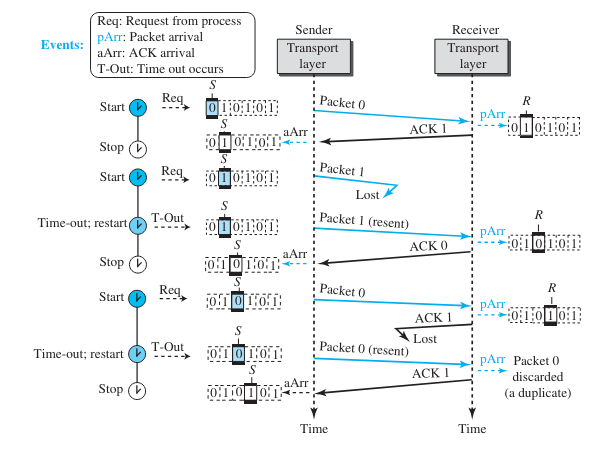
c. If a time-out occurs, the sender resends the only outstanding packet and restarts the timer.

**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**

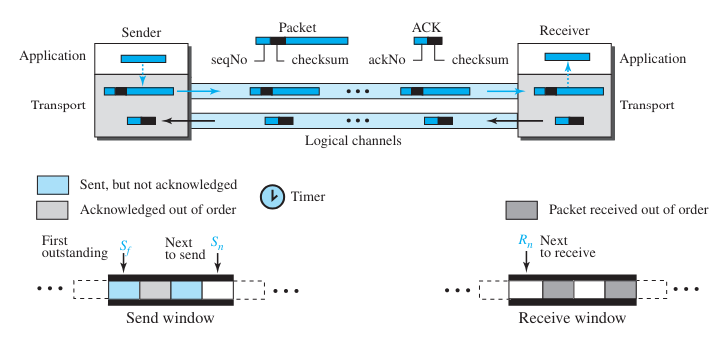
The Stop-and-Wait protocol is very inefficient if our channel is thick and long. By thick, we mean that our channel has a large bandwidth (high data rate); by long, we mean the round-trip delay is long. The product of these two is called the **bandwidth-delay** product. We can think of the channel as 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

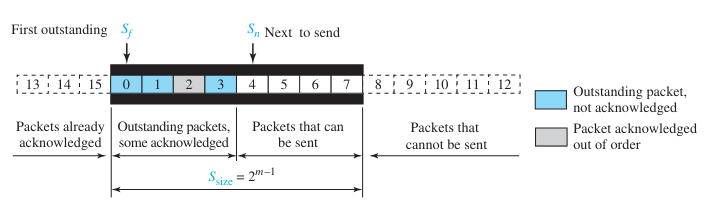
 **Fig 1.3**

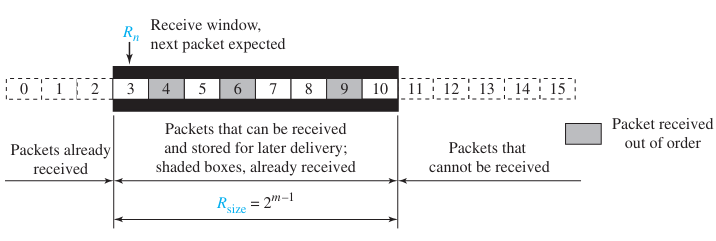
The Selective-Repeat protocol also uses two windows: a send window and a receive window. However, there are differences between the windows in this protocol and the ones in Go-Back-N. First, the maximum size of the send window is much smaller; it is 2m−1. Second, the receive window is the same size as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

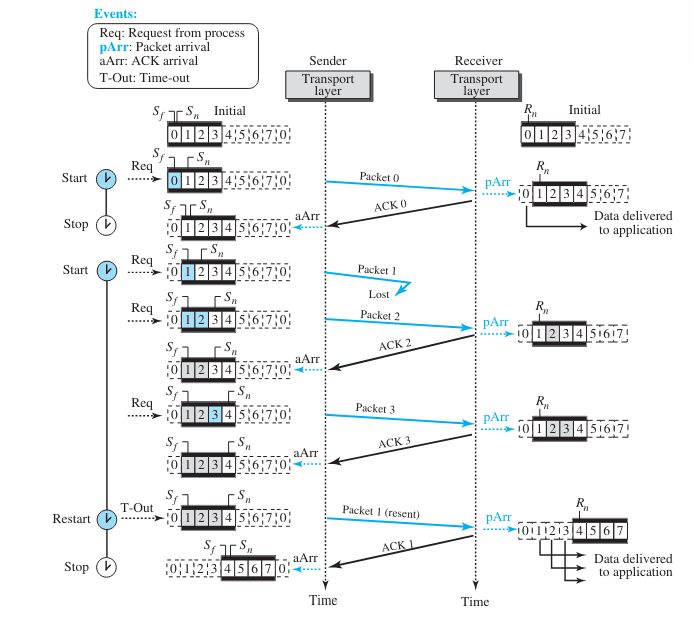
c. If a time-out occurs, the sender resends all unacknowledged packets in the window and restarts the timer.

**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.

**Bi-directional Flow Control**

In the previous protocols, data frames were transmitted in one direction only. In most practical situations, there is a need to transmit data in both directions. One way of achieving full-duplex data transmission is to run two instances of one of the previous protocols, each using a separate link for simplex data traffic (in different directions). Each link is then comprised of a ‘‘forward’’ channel (for

data) and a ‘‘reverse’’ channel (for acknowledgements). In both cases the capacity of the reverse channel is almost entirely wasted.

A better idea is to use the same link for data in both directions. After all, in protocols 2 and 3 it was already being used to transmit frames both ways, and the reverse channel normally has the same capacity as the forward channel. In this model the data frames from *A* to *B* are intermixed with the acknowledgement frames from *A* to *B*. By looking at the *kind* field in the header of an incoming

frame, the receiver can tell whether the frame is data or an acknowledgement.

Although interleaving data and control frames on the same link is a big improvement over having two separate physical links, yet another improvement is possible. When a data frame arrives, instead of immediately sending a separate control frame, the receiver restrains itself and waits until the network layer passes it the next packet. The acknowledgement is attached to the outgoing data frame (using the *ack* field in the frame header). In effect, the acknowledgement gets a free ride on the next outgoing data frame. The technique of temporarily delaying outgoing acknowledgements so that they can be hooked onto the next outgoing data frame is known as **piggybacking**.

The principal advantage of using piggybacking over having distinct acknowledgement

frames is a better use of the available channel bandwidth. The *ack* field in the frame header costs only a few bits, whereas a separate frame would need a header, the acknowledgement, and a checksum. In addition, fewer frames sent generally means a lighter processing load at the receiver. In the next protocol to be examined, the piggyback field costs only 1 bit in the frame header. It rarely

costs more than a few bits.

However, piggybacking introduces a complication not present with separate acknowledgements. How long should the data link layer wait for a packet onto which to piggyback the acknowledgement? If the data link layer waits longer than the sender’s timeout period, the frame will be retransmitted, defeating the whole purpose of having acknowledgements. If the data link layer were an oracle and could foretell the future, it would know when the next network layer packet was going to come in and could decide either to wait for it or send a separate acknowledgement immediately, depending on how long the projected wait was going to be. Of course, the data link layer cannot foretell the future, so it must resort to some ad hoc scheme, such as waiting a fixed number of milliseconds. If a new packet arrives quickly, the acknowledgement is piggybacked onto it. Otherwise, if no new packet has arrived by the end of this time period, the data link layer just sends a separate acknowledgement frame.

* 1. **Bidirectional Stop & Wait Protocol:**

Before tackling the general case, let us examine a sliding window protocol with a window size of 1. Such a protocol uses stop-and-wait since the sender transmits a frame and waits for its acknowledgement before sending the next one.

Now let us examine protocol to see how resilient it is to pathological scenarios. Assume that computer *A* is trying to send its frame 0 to computer *B* and that *B* is trying to send its frame 0 to *A*. Suppose that *A* sends a frame to *B*, but *A*’s timeout interval is a little too short. Consequently, *A* may time out repeatedly, sending a series of identical frames, all with *seq* = 0 and *ack* = 1*.*

When the first valid frame arrives at computer *B*, it will be accepted and *frame expected* will be set to a value of 1. All the subsequent frames received will be rejected because *B* is now expecting frames with sequence number 1, not 0. Furthermore, since all the duplicates will have *ack* = 1 and *B* is still waiting for an acknowledgement of 0, *B* will not go and fetch a new packet from its network

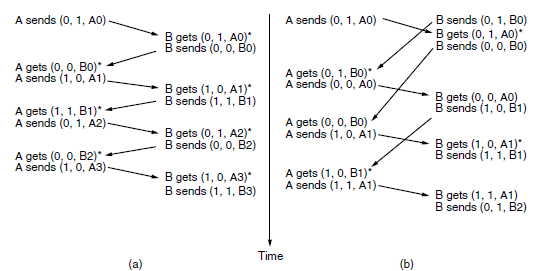
layer.

After every rejected duplicate comes in, *B* will send *A* a frame containing *seq* = 0 and *ack* = 0. Eventually, one of these will arrive correctly at *A*, causing *A* to begin sending the next packet. No combination of lost frames or premature timeouts can cause the protocol to deliver duplicate packets to either network layer, to skip a packet, or to deadlock. The protocol is correct.

However, to show how subtle protocol interactions can be, we note that a peculiar situation arises if both sides simultaneously send an initial packet. This synchronization difficulty is illustrated by Fig. 1.7. In part (a), the normal operation of the protocol is shown. In (b) the peculiarity is illustrated. If *B* waits for *A*’s first frame before sending one of its own, the sequence is as shown in (a), and

every frame is accepted.

However, if *A* and *B* simultaneously initiate communication, their first frames cross, and the data link layers then get into situation (b). In (a) each frame arrival brings a new packet for the network layer; there are no duplicates. In (b) half of the frames contain duplicates, even though there are no transmission errors. Similar situations can occur as a result of premature timeouts, even when one side

clearly starts first. In fact, if multiple premature timeouts occur, frames may be sent three or more times, wasting valuable bandwidth. 

**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 out at 0 and grows to some predefined maximum. The receiver’s window, in contrast, is always fixed in size and equal to the predetermined maximum. The receiver has a buffer reserved for each sequence number within its fixed window.

Associated with each buffer is a bit (*arrived* ) telling whether the buffer is full or empty. Whenever a frame arrives, its sequence number is checked by the function *between* to see if it falls within the window. If so and if it has not already been received, it is accepted and stored. This action is taken without regard to whether or not the frame contains the next packet expected by the network layer. Of course, it must be kept within the data link layer and not passed to the network layer until

all the lower-numbered frames have already been delivered to the network layer in the correct order. A protocol using this algorithm is given in Fig. 3-21. Nonsequential receive introduces further constraints on frame sequence numbers compared to protocols in which frames are only accepted in order. We can illustrate the trouble most easily with an example. Suppose that we have a 3-bit sequence number, so that the sender is permitted to transmit up to seven frames before being required to wait for an acknowledgement. Initially, the sender’s and receiver’s windows are as shown in Fig. 3-22(a). The sender now transmits frames 0 through 6. The receiver’s window allows it to accept any frame with a sequence number between 0 and 6 inclusive. All seven frames arrive correctly, so

the receiver acknowledges them and advances its window to allow receipt of 7, 0, 1, 2, 3, 4, or 5, as shown in Fig. 3-22(b). All seven buffers are marked empty. It is at this point that disaster strikes in the form of a lightning bolt hitting the telephone pole and wiping out all the acknowledgements. The protocol should operate correctly despite this disaster. The sender eventually times out and retransmits frame 0. When this frame arrives at the receiver, a check is made to see if it falls within the receiver’s window. Unfortunately, in Fig. 3-22(b) frame 0 is within the new window, so it is accepted as a new frame. The receiver also sends a (piggybacked) acknowledgement for frame 6, since 0 through 6 have been received.

The sender is happy to learn that all its transmitted frames did actually arrive correctly, so it advances its window and immediately sends frames 7, 0, 1, 2, 3, 4, and 5. Frame 7 will be accepted by the receiver and its packet will be passed directly to the network layer. Immediately thereafter, the receiving data link layer checks to see if it has a valid frame 0 already, discovers that it does, and passes the old buffered packet to the network layer as if it were a new packet. Consequently, the network layer gets an incorrect packet, and the protocol fails.

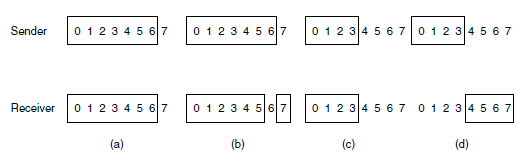
The essence of the problem is that after the receiver advanced its window, the new range of valid sequence numbers overlapped the old one. Consequently, the following batch of frames might be either duplicates (if all the acknowledgements were lost) or new ones (if all the acknowledgements were received). The poor receiver has no way of distinguishing these two cases.

The way out of this dilemma lies in making sure that after the receiver has advanced its window there is no overlap with the original window. To ensure that there is no overlap, the maximum window size should be at most half the range of the sequence numbers. This situation is shown in Fig. 3-22(c) and Fig. 3-22(d). With 3 bits, the sequence numbers range from 0 to 7. Only four unacknowledged frames should be outstanding at any instant. That way, if the receiver has just accepted frames 0 through 3 and advanced its window to permit acceptance of frames 4 through 7, it can unambiguously tell if subsequent frames are retransmissions (0 through 3) or new ones (4 through 7). In general, the window size for protocol 6 will be (*MAX SEQ* 1)*/*2.

An interesting question is: how many buffers must the receiver have? Under no conditions will it ever accept frames whose sequence numbers are below the lower edge of the window or frames whose sequence numbers are above the upper edge of the window. Consequently, the number of buffers needed is equal to the window size, not to the range of sequence numbers. In the preceding example of a 3-bit sequence number, four buffers, numbered 0 through 3, are needed. When frame *i* arrives, it is put in buffer *i* mod 4*.* Notice that although *i* and (*i* 4) mod 4 are ‘‘competing’’ for the same buffer, they are never within the window at the same time, because that would imply a window size of at least 5.

For the same reason, the number of timers needed is equal to the number of buffers, not to the size of the sequence space. Effectively, a timer is associated with each buffer. When the timer runs out, the contents of the buffer are retransmitted.

Protocol 6 also relaxes the implicit assumption that the channel is heavily loaded. We made this assumption in protocol 5 when we relied on frames being sent in the reverse direction on which to piggyback acknowledgements. If the reverse traffic is light, the acknowledgements may be held up for a long period of time, which can cause problems. In the extreme, if there is a lot of traffic in one direction and no traffic in the other direction, the protocol will block when the sender window reaches its maximum.



**Fig 1.8**

To relax this assumption, an auxiliary timer is started by *start ack timer* after an in-sequence data frame arrives. If no reverse traffic has presented itself before this timer expires, a separate acknowledgement frame is sent. An interrupt due to the auxiliary timer is called an *ack timeout* event. With this arrangement, traffic flow in only one direction is possible because the lack of reverse data frames onto which acknowledgements can be piggybacked is no longer an obstacle. Only one auxiliary timer exists, and if *start ack timer* is called while the timer is running, it has no effect. The timer is not reset or extended since its purpose is to provide some minimum rate of acknowledgements.

It is essential that the timeout associated with the auxiliary timer be appreciably shorter than the timeout used for timing out data frames. This condition is required to ensure that a correctly received frame is acknowledged early enough that the frame’s retransmission timer does not expire and retransmit the frame.

Protocol 6 uses a more efficient strategy than protocol 5 for dealing with errors. Whenever the receiver has reason to suspect that an error has occurred, it sends a negative acknowledgement (NAK) frame back to the sender. Such a frame is a request for retransmission of the frame specified in the NAK. In two cases, the receiver should be suspicious: when a damaged frame arrives or a frame other than the expected one arrives (potential lost frame). To avoid making multiple requests for retransmission of the same lost frame, the receiver should keep track of whether a NAK has already been sent for a given frame. The variable *no nak* in protocol 6 is *true* if no NAK has been sent yet for *frame expected*. If the NAK gets mangled or lost, no real harm is done, since the sender will eventually

time out and retransmit the missing frame anyway. If the wrong frame arrives after a NAK has been sent and lost, *no nak* will be *true* and the auxiliary timer will be started. When it expires, an ACK will be sent to resynchronize the sender to the receiver’s current status.

In some situations, the time required for a frame to propagate to the destination, be processed there, and have the acknowledgement come back is (nearly) constant. In these situations, the sender can adjust its timer to be ‘‘tight,’’ just slightly larger than the normal time interval expected between sending a frame and receiving its acknowledgement. NAKs are not useful in this case.

However, in other situations the time can be highly variable. For example, if the reverse traffic is sporadic, the time before acknowledgement will be shorter when there is reverse traffic and longer when there is not. The sender is faced with the choice of either setting the interval to a small value (and risking unnecessary retransmissions), or setting it to a large value (and going idle for a long

period after an error). Both choices waste bandwidth. In general, if the standard deviation of the acknowledgement interval is large compared to the interval itself, the timer is set ‘‘loose’’ to be conservative. NAKs can then appreciably speed up retransmission of lost or damaged frames.

Closely related to the matter of timeouts and NAKs is the question of determining which frame caused a timeout. In protocol 5, it is always *ack expected*, because it is always the oldest. In protocol 6, there is no trivial way to determine who timed out. Suppose that frames 0 through 4 have been transmitted, meaning that the list of outstanding frames is 01234, in order from oldest to youngest.

Now imagine that 0 times out, 5 (a new frame) is transmitted, 1 times out, 2 times out, and 6 (another new frame) is transmitted. At this point the list of outstanding frames is 3405126, from oldest to youngest. If all inbound traffic (i.e., acknowledgement-bearing frames) is lost for a while, the seven outstanding frames will time out in that order.

To keep the example from getting even more complicated than it already is, we have not shown the timer administration. Instead, we just assume that the variable *oldest frame* is set upon timeout to indicate which frame timed out.

**CODE**

**1.Code to implement Bidirectional stop and wait 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);

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("ArrivalNotification")) {

Receive();

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

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("ArrivalNotification")) {

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 ((seqNo("frame") != Rn) && !NakSent) {

SendNAK(Rn);

NakSent = true;

}

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 = 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);

}

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

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

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

}}

// 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);

}

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

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

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

**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 ((seqNo("frame") != Rn) && !NakSent) {

SendNAK(Rn);

NakSent = true;

}

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);

}

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

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

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

}}

**OUTPUT**

1. **Bidirectional stop and wait**

/\*OUTPUT 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 B : Frame created: Frame 1

Sender B : Frame stored: Frame 1

Sender B : Frame sent: Frame 1 to Receiver A

Sender B : Starting Timer for frame 1

Receiver A : Frame 1 received

Sender A : ACK received for Frame 0

Sender A : Timer stopped for Frame 0

Receiver B : Frame 0 received

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 \*/

/\*OUTPUT 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 B : Frame created: Frame 1

Sender B : Frame stored: Frame 1

Sender B : Frame sent: Frame 1 to Receiver A

Sender B : Starting Timer for frame 1

Receiver A : Frame 1 received

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 B : Frame 2 received

Sender A : ACK received for Frame 2

Sender A : Timer stopped for Frame 2

Sender A : ACK received for Frame 1

Sender A : Timer stopped for Frame 1

Receiver B : Frame 1 received

Sender B : ACK received for Frame 1

Sender B : Timer stopped for Frame 1

Receiver B : Frame lost: Frame 0

Sender A : Timer timeout for Frame 0

Sender A : Resending frame: 0

Sender A : Starting Timer for frame 0

Receiver B : Frame 0 received

Sender A : ACK received for Frame 0

Sender A : Timer stopped for Frame 0

Window slide forward by 1

current window Size: 3\*/

/\* OUTPUT 3

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 B : Frame created: Frame 1

Sender B : Frame stored: Frame 1

Sender B : Frame sent: Frame 1 to Receiver A

Sender B : Starting Timer for frame 1

Receiver A : Frame 1 received

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 B : Frame 2 received

Sender A : ACK received for Frame 2

Sender A : Timer stopped for Frame 2

Sender A : ACK received for Frame 1

Sender A : Timer stopped for Frame 1

Receiver B : Frame 1 received

Sender B : ACK received for Frame 1

Sender B : Timer stopped for Frame 1

Receiver B : ACK lost for Frame 0

Sender A : Resending frame: 0

Sender A : Starting Timer for frame 0

Receiver A : Frame 0 received

Sender A : ACK received for Frame 0

Sender A : Timer stopped for Frame 0

Window slide by 1 forward

Current Window Size: 3 \*/

**2)Bidirectional Selective repeat**

/\*OUTPUT 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 B : Frame created: Frame 1

Sender B : Frame stored: Frame 1

Sender B : Frame sent: Frame 1 to Receiver A

Sender B : Starting Timer for frame 1

Receiver A : Frame 1 received

Sender A : ACK received for Frame 0

Sender A : Timer stopped for Frame 0

Receiver B : Frame 0 received

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 \*/

/\*OUTPUT 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 B : Frame created: Frame 1

Sender B : Frame stored: Frame 1

Sender B : Frame sent: Frame 1 to Receiver A

Sender B : Starting Timer for frame 1

Receiver A : Frame 1 received

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 B : Frame 2 received

Sender A : ACK received for Frame 2

Sender A : Timer stopped for Frame 2

Sender A : ACK received for Frame 1

Sender A : Timer stopped for Frame 1

Receiver B : Frame 1 received

Sender B : ACK received for Frame 1

Sender B : Timer stopped for Frame 1

Receiver B : Frame lost: Frame 0

Sender A : Timer timeout for Frame 0

Sender A : Resending frame: 0

Sender A : Starting Timer for frame 0

Receiver B : Frame 0 received

Sender A : ACK received for Frame 0

Sender A : Timer stopped for Frame 0

Window slide forward by 1

current window Size: 3 \*/

/\* OUTPUT 3

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 B : Frame created: Frame 1

Sender B : Frame stored: Frame 1

Sender B : Frame sent: Frame 1 to Receiver A

Sender B : Starting Timer for frame 1

Receiver A : Frame 1 received

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 B : Frame 2 received

Sender A : ACK received for Frame 2

Sender A : Timer stopped for Frame 2

Sender A : ACK received for Frame 1

Sender A : Timer stopped for Frame 1

Receiver B : Frame 1 received

Sender B : ACK received for Frame 1

Sender B : Timer stopped for Frame 1

Receiver B : ACK lost for Frame 0

Sender A : Resending frame: 0

Sender A : Starting Timer for frame 0

Receiver A : Frame 0 received

Sender A : ACK received for Frame 0

Sender A : Timer stopped for Frame 0

Window slide by 1 forward

Current Window Size: 3

\*/

**OBSERVATIONS**

**1. Bidirectional flow control protocol implementation**

The implementation of bidirectional flow control protocols in the project facilitated simultaneous data transmission in both directions, optimizing communication efficiency.The protocol ensures that the sender and receiver can independently control the flow of data, preventing either side from being overwhelmed. This bidirectional approach is essential for enhancing the utilization of the network bandwidth by allowing data transmission in both directions concurrently. It significantly enhances the reliability and efficiency of data transfer in real-world networking scenarios, making it a valuable addition to communication protocols. extra{ The implementation of bidirectional flow control reduces the occurrence of network congestion and improves the overall throughput of the communication system.

In scenarios where acknowledgment is lost, bidirectional flow control facilitates a more efficient retransmission process, ensuring reliable data delivery with minimal latency. }

**2.Case studies Demonstration**:

The integration of the flow control protocols with the bidirectional approach added an extra layer of efficiency and reliability to the frame transfer process.

**i) Frame lost:**

**Bidirectional stop & wait:**

In the event of frame loss, the sender does not receive an acknowledgment within the timeout period, leading to frame retransmission. The sender resends the lost frame, and the receiver acknowledges it upon successful reception. This process continues until the sender receives an acknowledgment. Successful data delivery is achieved with an increase in latency due to retransmissions.

**Bidirectional Selective Repeat:**

In the event of frame loss, only the lost frame is retransmitted instead of the entire window of frames. The receiver, upon receiving the retransmitted frame, acknowledges it, and data transfer proceeds. This process significantly reduces latency and ensures efficient data delivery, especially in scenarios with a high probability of frame loss.

**ii) Acknowledgement loss:**

**Bidirectional stop & wait:**

In the event of acknowledgment loss, the sender does not receive it within the timeout period, leading to frame retransmission. The sender resends the frame, and the receiver re-acknowledges it upon successful reception. This process continues until the sender receives an acknowledgment. Successful data delivery is achieved with an increase in latency due to retransmissions.

**Bidirectional Selective Repeat:**

In the event of acknowledgment loss, the sender does not receive it within the timeout period, leading to selective retransmission of the lost acknowledgment.

The receiver, upon receiving the retransmitted acknowledgment, continues data transfer. This process minimizes latency and ensures efficient data delivery, especially in scenarios with a high probability of acknowledgment loss.

**3.Code implementation:**

The code implementation of the project demonstrated a clear understanding of flow control protocols and the bidirectional technique. The two way communication was effectively established, and the protocols were implemented robustly using bidirectional technique. Additionally, the integration was seamless, providing more efficient and reliable transmission.

**4.Observations from execution**

During the execution of the project, several observations were made:

The flow control ensured that data is transferred at an optimal rate, preventing overwhelming the receiver or network.

The bidirection process is resilient to loss, significantly reducing latency and ensuring efficient data delivery, especially in scenarios with a high probability of frame or acknowledgment loss.

The project highlighted the importance of bidirectional stop & wait and selective repeat protocols in ensuring reliable data delivery, even in scenarios of acknowledgment loss.

**5. Limitations and future considerations**

While the project successfully implemented bidirectional stop and wait and selective protocol, there are opportunities for further enhancement:

Implementing congestion control mechanisms to manage network traffic more effectively and improve overall performance.

Conducting performance optimization by refining the protocol implementation and minimizing redundant processes.

Utilizing load balancing algorithms to distribute network traffic evenly and prevent congestion. Overall, the project provided valuable insights into bidirectional flow control, protocols, and channel bandwidth considerations, laying a solid foundation for future developments in network applications.

**CONCLUSION**

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

**Key Achievements:**

**Successful Implementation:**

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

**Optimized Bandwidth Utilization:**

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

**Reliability:**

Ensured reliable data delivery, even in scenarios of acknowledgment loss, 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 and outlined the potential limitations and oversights, offering insights into overcoming scalability issues and improving error handling.

**Future Directions:**

**Performance Enhancement:**

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

**Scalability and Adaptability:**

Investigate the implementation of congestion control mechanisms to manage network traffic more effectively, ensuring scalability and adaptability.

**Optimization:**

Conduct performance optimization by refining the 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 a robust error recovery mechanism to handle various types of errors effectively.

In conclusion, the project has provided a solid foundation in bidirectional flow control, protocols, channel utilisation techniques, laying the groundwork for future developments in efficient and reliable communication.

**LEARNING OUTCOME**

Following are the learning outcomes of mine by this mini project that is "Design and implementation of bi-directional stop & wait and selective repeat" :

Understanding role of flow control in ensuring reliable data transmission.

Familiarity with bidirectional flow control and protocols

Experience in implementing bidirectional stop & wait and selective repeat for efficient communication.

Enhanced skills in error handling and flow management in network applications:

Insight into the importance of two way transmission.

Ability to apply theoretical knowledge of networking concepts to practical programming tasks.

Proficiency in using Java for network applications.

Understanding the challenges involved in data transmission.

Knowledge of potential optimisations for improving the efficiency of communication.

Appreciation of the complexities and nuances of real world networked applications.

**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

William Stallings, Data and Computer Communications (Vol.10), Pearson, 2013, 978-1292014388