Go, change the world

RV Educational Institutions ®

RV College of Engineering ®

Autonomous Institution Affiliated to Visvesvaraya Technological University, Belagavi

Approved by AICTE, New Delhi

Assignment Report on

GOBACK N ARQ AND SELECTIVE REPITITIVE ARQ

COMPUTER NETWORKS (22MCA13TL)

MASTEROFCOMPUTERAPPLICATIONS

By

1RV22MC069	**SANTHOSHA K**

1RV22MC086	**PRATHAMESH. M .HIREMATH**

UNDER FACULTY OF

PROF. CHANDRANI CHAKRAVORTY

DEPARTMENT OF COMPUTER APPLICATION
RV COLLEGE OF ENGINEERING
MYSORE ROAD BANGALORE
MAY 2023

INDEXPAGE

INTRODUCTION

GOBACK ARQ

The provided program is a simulation of a sliding window protocol for reliable data transmission. It demonstrates the behavior of a sender transmitting packets to a receiver over an unreliable channel. The program allows the user to specify the number of packets, window size, and timeout duration. It simulates packet loss and ACK loss to test the protocol's ability to handle these scenarios.

SELECTIVE REPEAT ARQ

The given program implements a simplified sliding window protocol for reliable data transmission. It simulates the transmission of packets from a sender to a receiver over an unreliable channel. The program uses a fixed-size window and incorporates timeout mechanisms to handle potential packet loss.

OBJECTIVES

GOBACK ARQ

Simulate Sliding Window Protocol: The program aims to simulate the sliding window protocol, which allows the sender to transmit multiple packets before receiving acknowledgments for them.

User-Defined Input: The program prompts the user to enter the maximum number of packets to generate, the window size, and the timeout duration. This allows customization of the simulation based on user requirements.

Packet Generation: The program generates packets with sequential sequence numbers and assigns data to each packet. The number of packets generated is based on the user-defined maximum number of packets.

Simulate Packet Transmission: The program simulates the transmission process by sending packets within the window size. It accounts for possible packet loss by randomly deciding whether to send or discard a packet.

Timeout Handling: The program includes a timeout mechanism that tracks the duration of packet transmission. If the timeout duration is exceeded without receiving an acknowledgment, the program resends the unacknowledged packets.

ACK Reception: The program simulates the reception of acknowledgments (ACKs) from the receiver. It considers the possibility of ACK loss and retransmits packets if an ACK is lost**.**

SELECTIVE REPAETIVE

Simulate the sliding window protocol: The program demonstrates the operation of a sliding window protocol, which allows the sender to transmit multiple packets without waiting for individual acknowledgments.

Ensure reliable data transmission: The program ensures that all packets are successfully transmitted and received by implementing error detection and retransmission mechanisms.

Handle packet loss: The program includes a timeout mechanism that retransmits packets if their acknowledgments are not received within a specified timeout duration.

Implement a fixed-size window: The program uses a fixed-size window, which limits the number of unacknowledged packets the sender can transmit at a given time.

Validate successful transmission: Upon successful transmission of all packets, the program displays a message indicating the completion of the transmission process.

SLIDING WINDOW PROTOCOL

The sliding window protocol is a method used in computer networking to ensure reliable and sequential data transmission over a communication channel that may have errors, congestion, or limited capacity. It is commonly used in protocols such as TCP (Transmission Control Protocol) for reliable data transfer.

The protocol involves dividing the data into small, fixed-size units called "frames" and assigning a sequence number to each frame. The sender maintains a "sliding window" of frames that can be transmitted without waiting for acknowledgment from the receiver. The size of the window determines the number of frames that can be sent before requiring an acknowledgment.

WORKING

- The sender divides the data into frames and assigns a sequence number to each frame. The sender then starts sending frames within the current window.

- The receiver receives the frames and sends acknowledgment (ACK) for the received frames. The ACK includes the sequence number of the next frame expected by the receiver.

- The sender receives the ACK and slides the window forward, acknowledging that the frames within the window have been successfully received. The sender then sends new frames for the next window.

- If the sender does not receive an ACK within a specified timeout period, it assumes that a frame or ACK has been lost. It retransmits the frames within the current window.

- The process continues until all frames have been successfully transmitted and acknowledged.

DIAGRAMIC REPRESENTAION OF SLIDING WINDOW PROTOCOL

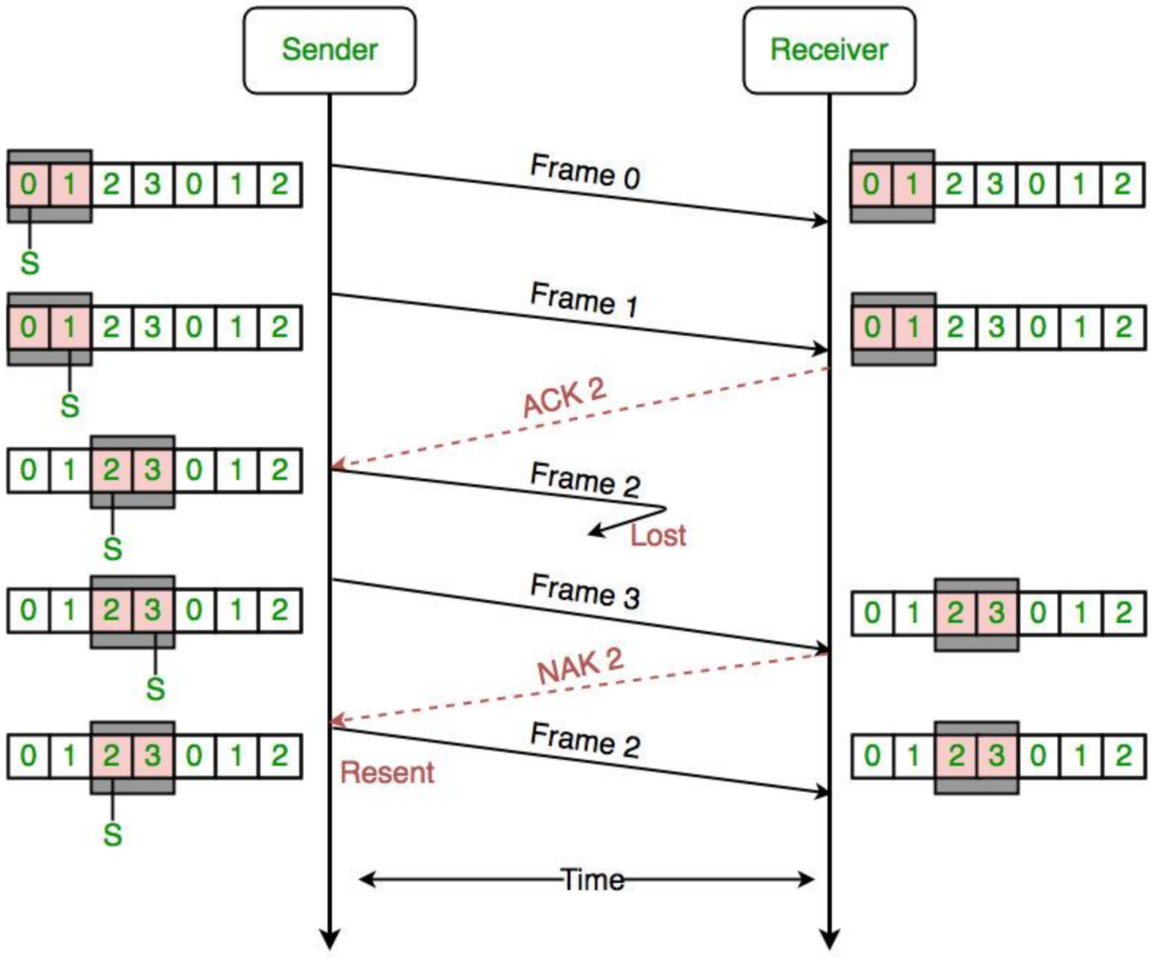


Fig 1.0 Sliding Window Protocol

USES IN COMPUTER NETWORKS

The sliding window protocol has several important uses in computer networks. Here are some of its main applications:

- Reliable Data Transfer:
- The sliding window protocol is primarily used to ensure reliable data transfer over unreliable communication channels. By using sequence numbers, acknowledgments, and selective retransmission, it allows for the reliable delivery of data, even in the presence of errors, packet loss, or network congestion.

- Flow Control
- Sliding window protocol also provides flow control mechanisms to prevent

the sender from overwhelming the receiver with a large number of frames. The receiver's acknowledgment acts as a signal to the sender to continue transmitting more frames within the window size, thus maintaining an optimal flow of data.

- Congestion Control
- Sliding window protocols play a role in managing network congestion. By limiting the number of unacknowledged frames in the window, the protocol ensures that the sender does not flood the network with excessive data, helping to prevent congestion and improve overall network performance.

- Efficient Bandwidth Utilization
- The sliding window protocol optimizes the utilization of available bandwidth by allowing the sender to transmit multiple frames before waiting for acknowledgments. This pipelining technique improves the efficiency of data transmission, as the sender can continuously transmit new frames while waiting for acknowledgments for previously sent frames.

- Error Detection and Recovery
- The protocol enables the detection of errors during data transmission. If a frame is damaged or lost, the receiver can request the retransmission of that specific frame using negative acknowledgments (NAKs). This selective retransmission approach minimizes unnecessary retransmissions and reduces network overhead.

- Multicast and Broadcast Applications
- The sliding window protocol can also be adapted for multicast or broadcast scenarios, where data is sent to multiple receivers simultaneously. Each receiver maintains its own sliding window and acknowledges the frames it successfully receives, allowing for reliable data transfer to multiple recipients.

Go-Back-N ARQ (Automatic Repeat Request)

Go-Back-N ARQ (Automatic Repeat Request) is an error control protocol used for reliable data transmission over a communication channel that may introduce errors or have limited capacity. It is a specific variation of the Automatic Repeat Request protocol and is widely used in computer networks.

In Go-Back-N ARQ, the sender is allowed to transmit a window of multiple frames without waiting for individual acknowledgments from the receiver. The receiver acknowledges the successful receipt of frames, and if any errors occur or frames are lost, it requests the sender to retransmit a specific range of frames

WORKING

- Sender Window Initialization: The sender divides the data into frames and assigns a sequence number to each frame. It also maintains a sending window, which represents the range of frames that can be sent without waiting for acknowledgments. The sender starts by sending frames within the initial window.

- Frame Transmission: The sender starts transmitting frames within the sending window. The frames are sent continuously, one after another, until the entire window is filled. The sender does not wait for individual acknowledgments after each frame.

- Frame Reception at the Receiver: The receiver receives the frames sent by the sender. It checks each frame for errors, using techniques like checksum or cyclic redundancy check (CRC). If a frame is error-free and arrives in order, the receiver sends an acknowledgment (ACK) for that frame which contains the sequence number of the next expected frame.

- Positive Acknowledgment (ACK) Reception: Upon receiving an ACK, the sender slides the sending window forward, acknowledging that the frames within the window have been successfully received. The sender then sends new frames for the next window, which can now include previously unacknowledged frames.

- Negative Acknowledgment (NAK) Reception: If the receiver receives a frame that is damaged, out of order, or missing, it discards the frame and sends a negative acknowledgment (NAK) to the sender. The NAK requests the retransmission of a specific range of frames, starting from the earliest unacknowledged frame.

- Retransmission: Upon receiving a NAK or a timeout event, the sender retransmits all the frames within the current sending window, starting from the earliest unacknowledged frame. This ensures that any lost or corrupted frames are sent again.

- Window Sliding: If the sender receives an ACK or NAK, it slides the sending window forward to the next unacknowledged frame. This allows for the continuous transmission of new frames while still maintaining the reliability of the data transfer.

- Process Continuation: The process continues until all frames have been successfully transmitted and acknowledged. The sender keeps sending frames within the window and slides the window forward based on the acknowledgments received from the receiver.

- Go-Back-N ARQ provides a simple and efficient way to ensure reliable data transmission. However, it may result in unnecessary retransmissions when a single frame is lost or corrupted. This limitation can be addressed by more advanced protocols like Selective Repeat ARQ, which allow for individual retransmission of lost frames.

USES AND PURPOSES

The Go-Back-N ARQ protocol serves several important uses and purposes in computer networks. Here are some of its main applications

- Reliable Data Transfer: The primary purpose of Go-Back-N ARQ is to ensure reliable data transfer over unreliable communication channels. By retransmitting a window of frames when errors occur, it enables the receiver to correctly receive and reconstruct the transmitted data.

- Error Detection and Recovery: The protocol incorporates error detection mechanisms, such as checksums or CRC, to identify damaged or corrupted frames. When errors are detected, the receiver can request the sender to retransmit the affected frames, ensuring data integrity.

- Flow Control: Go-Back-N ARQ provides flow control mechanisms to regulate the rate of data transmission. The receiver's acknowledgment acts as a signal to the sender to continue transmitting more frames within the window size. This helps prevent the receiver from being overwhelmed with a large number of frames.

- Bandwidth Utilization: The protocol optimizes the utilization of available bandwidth by allowing the sender to transmit multiple frames before waiting for acknowledgments. This pipelining technique improves the efficiency of data transmission by minimizing idle time.

- Simple Implementation: Go-Back-N ARQ is relatively simple to implement compared to more complex protocols like Selective Repeat ARQ. It offers a good balance between simplicity and reliability, making it suitable for scenarios where a moderate level of error control is required.

- Network Congestion Control: The protocol indirectly contributes to network congestion control by regulating the rate at which frames are transmitted. By limiting the number of unacknowledged frames in the sending window, it helps prevent the sender from overwhelming the network with excessive data.

- Support for Unidirectional Channels: Go-Back-N ARQ can be applied to unidirectional communication channels, where acknowledgments are not explicitly transmitted back to the sender. Instead, the receiver's NAKs trigger the sender to retransmit frames, ensuring reliable data transfer.

- Overall, Go-Back-N ARQ is a widely used error control protocol that provides reliable data transmission and efficient bandwidth utilization. Its simplicity and effectiveness make it suitable for various network scenarios, especially those with moderate error rates and moderate bandwidth capacities.

WHERE IT BE USED

- Local Area Networks (LANs): Go-Back-N ARQ can be used in LAN environments where there is a need for reliable data transfer between network devices. It ensures that data packets are successfully delivered across the LAN, even in the presence of errors or congestion.

- Wide Area Networks (WANs): WANs often involve long-distance communication over networks that may experience higher error rates or longer propagation delays. Go-Back-N ARQ can help ensure reliable data transmission in such scenarios, mitigating the effects of errors and improving overall data transfer performance.

- Wireless Networks: Wireless communication channels are susceptible to various sources of errors, such as interference and signal attenuation. Go-Back-N ARQ can be employed in wireless networks, such as Wi-Fi networks or cellular networks, to enhance the reliability of data transmission and maintain a high-quality connection.

- Satellite Communication: Satellite communication systems often encounter signal degradation due to factors like atmospheric conditions or signal interference. Go-Back-N ARQ is used to minimize the impact of these issues and achieve reliable data transfer over satellite links.

DIAGRAPHIC REPRESENTATION

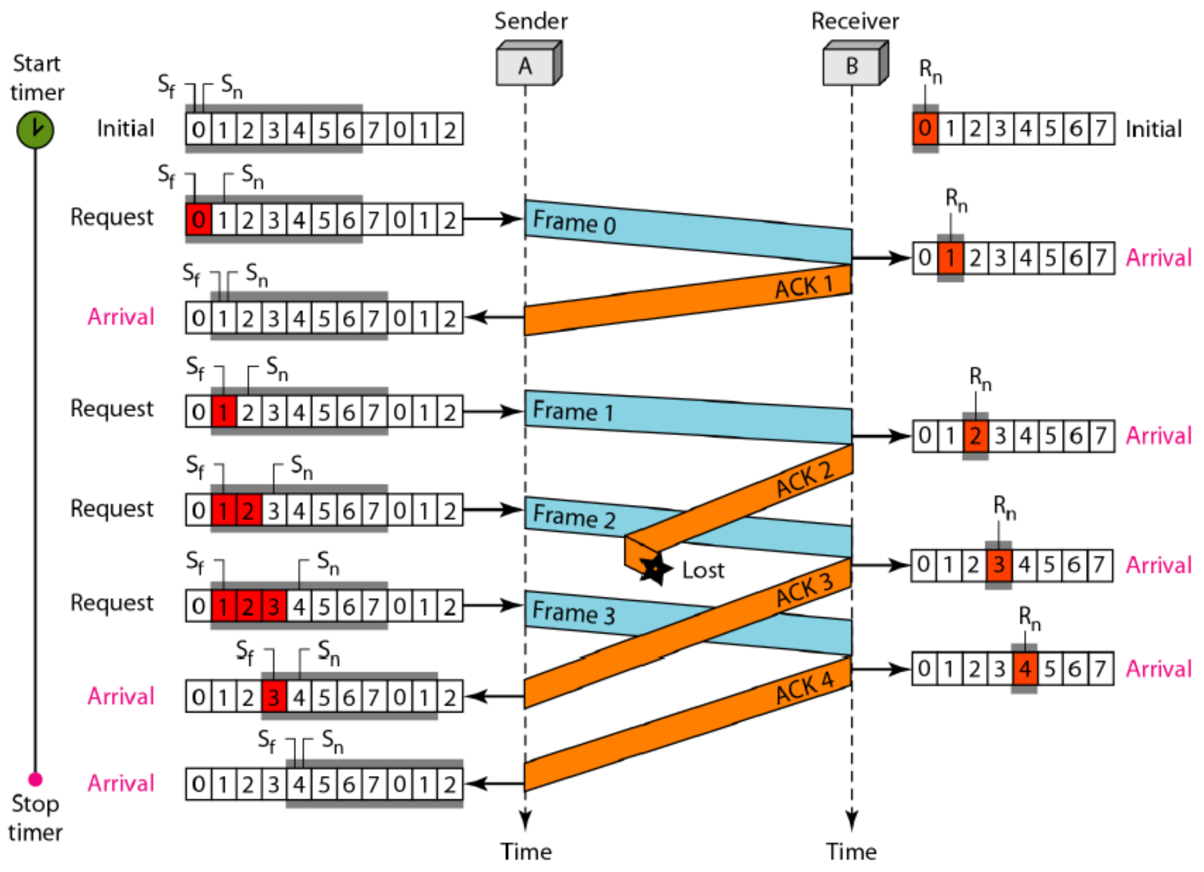


Fig 1.1 Go-Back-N ARQ

HOW IT IS IMPLEMENTED

We implement this Go back N Arq by C language
Here is the Detail Explain

- The code includes necessary header files: \`stdio.h\` for input/output operations, \`stdlib.h\` for memory allocation and random number generation, \`string.h\` for string operations, \`time.h\` for time-related functions, and \`stdbool.h\` for boolean data type.
- A constant \`MAX_PACKETS\` is defined to represent the maximum number of packets that can be generated.

- The code defines a structure called \`packet\`, which consists of an integer field \`seq_num\` representing the sequence number of the packet, and a character array \`data\` of size 20 to store the packet data.

- The \`main()\` function begins the execution of the program.

- Several variables are declared: \`packets\` as an array of \`packet\` structures to hold the generated packets, \`num_packets\` to store the user input for the maximum number of packets, \`window_size\` to store the user input for the window size, \`timeout_duration\` to store the user input for the timeout duration, \`base\` to track the base of the sending window, \`nextseqnum\` to track the next sequence number to be sent, \`expectedseqnum\` to track the expected sequence number to be received, \`start_timer\` to store the starting time of the timer, and \`timer_running\` to indicate whether the timer is currently running or not.

- The user is prompted to enter the maximum number of packets to generate, the window size, and the timeout duration using \`printf()\` and \`scanf()\`.

- A loop is initiated to generate the packets. It iterates \`num_packets\` times and assigns a sequence number and data to each packet using the \`sprintf()\` function. The generated packets are stored in the \`packets\` array.

- The simulation of the transmission process begins with a while loop that continues until all expected packets have been received.

- Within the while loop, another while loop is used to send packets within the window. It sends packets as long as the \`nextseqnum\` is less than the base plus the window size and less than the total number of packets. The packets are sent by printing the packet number using \`printf()\`. The code simulates packet loss by generating a random number from 0 to 9 and only proceeding with sending the packet if the generated number is not 0. If a packet is lost, it is considered as a failed transmission, and the code prints a message indicating the loss.

- After sending the packets within the window, the code checks for a timeout condition. If the timer is running and the elapsed time (in seconds) since the start of the timer is greater than or equal to the specified `timeout_duration`, a timeout is considered to have occurred. The code prints a message indicating the timeout and resets the `nextseqnum` to the base value, indicating the retransmission of packets from the base.

- Another while loop is used to receive ACKs (acknowledgments) from the receiver. It continues until the `expectedseqnum` is less than the `nextseqnum`, meaning there are still packets to be acknowledged. The code prints a message indicating the expected ACK number. The code simulates ACK loss by generating a random number from 0 to 9 and only proceeds with acknowledging the packet if the generated number is not 0. If an ACK is lost, it is considered as a failed acknowledgment, and the code prints a message indicating the loss.

- The code updates the `expectedseqnum` by incrementing it if a valid ACK is received. If the timer is running and the `expectedseqnum` becomes equal to the `nextseqnum`, it means

EXAMPLE

Suppose the user enters the following inputs:
- Maximum number of packets to generate: **10**
- Window size**: 3**
- Timeout duration: **2**

The code will generate 10 packets with sequence numbers from 0 to 9 and store them in the **`packets`** array.

The simulation of the transmission process begins. Here's a step-by-step breakdown:

1. Initially, the **`base`** and **`nextseqnum`** are both 0, and the **`expectedseqnum`** is also 0.

2. The code enters the outer while loop since the **`expectedseqnum`** is less than the total number of packets (10).

3. In the first iteration of the outer while loop, the code enters the inner while loop to send packets within the window. Since the `**nextseqnum**` (0) is less than the `base` (0) plus the window size (3) and less than the total number of packets (10), it sends the packet with sequence number 0.

Output

Sending packet 0

4. The code checks for packet loss by generating a random number. Assuming the generated number is not 0, the transmission of packet 0 is considered successful.

5. Since the timer is not running, the code starts the timer and increments `**nextseqnum**` to 1.

6. In the next iteration of the inner while loop, the code sends packet 1.

Output

Sending packet 1

7. Assuming packet 1 is successfully transmitted, the code starts the timer (if not running) and increments `**nextseqnum**` to 2.

8. In the third iteration of the inner while loop, the code sends packet 2.

Output

Sending packet 2

9. Assuming packet 2 is successfully transmitted, the code starts the timer (if not running) and increments `**nextseqnum**` to 3.

10. Since `**nextseqnum**` (3) is equal to `base` (0) plus the window size (3), the inner while loop exits.

11. The code checks for a timeout. Assuming the elapsed time since the start of the timer is less than 2 seconds, no timeout occurs in this iteration.

12. The code enters the inner while loop to receive ACKs. Since the `**expectedseqnum**` (0) is less than `**nextseqnum**` (3), it waits for the acknowledgment of packet 0.

Output

Waiting for ACK 0

13. Assuming the ACK for packet 0 is received, the code increments `**expectedseqnum**` to 1 and checks if the timer is running. Since `**expectedseqnum**` (1) is not equal to `**nextseqnum**` (3), the timer continues to run.

14. The inner while loop iterates again to wait for the acknowledgment of packet

Output

Waiting for ACK 1

15. Assuming the ACK for packet 1 is received, the code increments `**expectedseqnum**` to 2 and checks if the timer is running. Since `**expectedseqnum**` (2) is not equal to `**nextseqnum**` (3), the timer continues to run.

16. The inner while loop iterates once more to wait for the acknowledgment of packet 2.

Output

Waiting for ACK 2

17. Assuming the ACK for packet 2 is received, the code increments `**expectedseqnum**` to 3 and checks if the timer is running.
Since `**expectedseqnum**` (3) is equal to `**nextseqnum**` (3), the timer is no longer needed and is stopped.

18. The code updates the `base` value to the `**expectedseqnum**` (3) since all packets up to the base have been successfully acknowledged.

19. The outer while loop continues as the `**expectedseqnum**` (3) is still less than the total number of packets (10).

20. The code re-enters the inner while loop to send packets within the window. Since the `**nextseqnum**` (3) is less than the `base` (3) plus the window size (3) and less than the total number of packets (10), it sends packet 3.

Output

```
Sending packet 3
```

21. Assuming packet 3 is successfully transmitted, the code starts the timer (if not running) and increments `**nextseqnum**` to 4.

22. In the next iteration of the inner while loop, the code sends packet 4.

Output

```
Sending packet 4
```

23. Assuming packet 4 is successfully transmitted, the code starts the timer (if not running) and increments `**nextseqnum**` to 5.

24. In the third iteration of the inner while loop, the code sends packet 5.

Output

```
Sending packet 5
```

25. Assuming packet 5 is lost, the code detects the loss and prints a message indicating the loss.

Output

```
Packet 5 lost, resending packet 5
```

26. The inner while loop continues to send the remaining packets within the window.

27. The code checks for a timeout. Assuming the elapsed time since the start of the timer is greater than or equal to 2 seconds, a timeout occurs.

Output

```
  Timeout occurred, resending packets 3 to 5
```

28. The `**nextseqnum**` is reset to the `base` value (3), indicating the retransmission of packets from the base.

29. The inner while loop continues to retransmit the packets that were lost.

30. The code receives the ACKs for the transmitted packets and updates the `**expectedseqnum**`.

31. The code updates the `base` value to the `**expectedseqnum**` since all packets up to the base have been successfully acknowledged.

32. The outer while loop continues until all expected packets have been received.

33. Once all packets have been successfully transmitted and acknowledged, the code exits the while loop and prints a success message.

Output

```
 All packets transmitted successfully!
```

This example demonstrates the operation of the Go-Back-N ARQ protocol where packets are sent within a sliding window, and in the event of a timeout or packet loss, the protocol retransmits the packets starting from the base. The process continues until all packets are successfully transmitted and acknowledged.

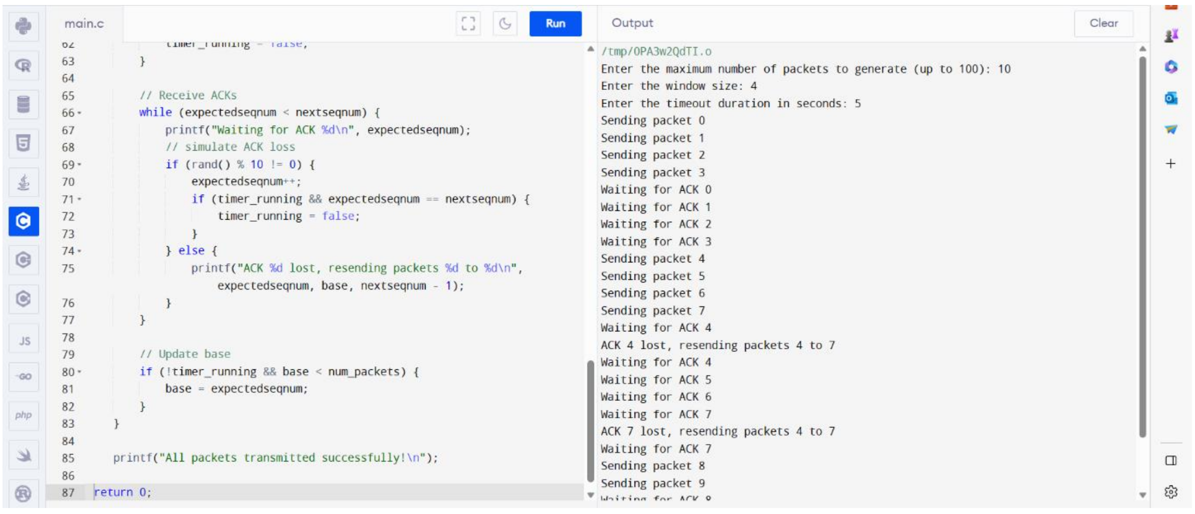
CODE

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <time.h>
#include <stdbool.h>

#define MAX_PACKETS 100

typedef struct {
    int seq_num;
    char data[20];
} packet;

int main() {
    packet packets[MAX_PACKETS];
    int num_packets = 0;
    int window_size = 0;
    int timeout_duration = 0;
    int base = 0;
    int nextseqnum = 0;
    int expectedseqnum = 0;
clock_tstart_timer = 0;
    bool timer_running = false;

    // Get user input
printf("Enter the maximum number of packets to generate (up to %d): ",
MAX_PACKETS);
scanf("%d", &num_packets);
printf("Enter the window size: ");
scanf("%d", &window_size);
printf("Enter the timeout duration in seconds: ");
scanf("%d", &timeout_duration);

    // Generating packets
    for (int i = 0; i<num_packets; i++) {
        packet p;
p.seq_num = i;
sprintf(p.data, "Packet %d", i);
        packets[i] = p;
    }

    // Simulation of transmission
    while (expectedseqnum<num_packets) {
        // Send packets within the window
        while (nextseqnum< base + window_size&&nextseqnum<num_packets) {
```

```
printf("Sending packet %d\n", nextseqnum);
            // simulate packet loss
            if (rand() % 10 != 0) {
                if (!timer_running) {
start_timer = clock();
timer_running = true;
                }
nextseqnum++;
            } else {
printf("Packet %d lost, resending packet %d\n", nextseqnum, nextseqnum);
            }
        }

        // Check for timeout
        if (timer_running&& (double)(clock() - start_timer) / CLOCKS_PER_SEC
>= timeout_duration) {
printf("Timeout occurred, resending packets %d to %d\n", base, nextseqnum -
1);
nextseqnum = base;
timer_running = false;
        }

        // Receive ACKs
        while (expectedseqnum<nextseqnum) {
printf("Waiting for ACK %d\n", expectedseqnum);
            // simulate ACK loss
            if (rand() % 10 != 0) {
expectedseqnum++;
                if (timer_running&&expectedseqnum == nextseqnum) {
timer_running = false;
                }
            } else {
printf("ACK %d lost, resending packets %d to %d\n", expectedseqnum, base,
nextseqnum - 1);
            }
        }

        // Update base
        if (!timer_running&& base <num_packets) {
            base = expectedseqnum;
        }
    }
printf("All packets transmitted successfully!\n");
```

```
  return 0;
}
```



```c
            timer_running = false;
        }

        // Receive ACKs
        while (expectedseqnum < nextseqnum) {
            printf("Waiting for ACK %d\n", expectedseqnum);
            // simulate ACK loss
            if (rand() % 10 != 0) {
                expectedseqnum++;
                if (timer_running && expectedseqnum == nextseqnum) {
                    timer_running = false;
                }
            } else {
                printf("ACK %d lost, resending packets %d to %d\n",
                    expectedseqnum, base, nextseqnum - 1);
            }
        }

        // Update base
        if (!timer_running && base < num_packets) {
            base = expectedseqnum;
        }
    }

    printf("All packets transmitted successfully!\n");

return 0;
```

```
/tmp/OPA3w2QdTI.o
Enter the maximum number of packets to generate (up to 100): 10
Enter the window size: 4
Enter the timeout duration in seconds: 5
Sending packet 0
Sending packet 1
Sending packet 2
Sending packet 3
Waiting for ACK 0
Waiting for ACK 1
Waiting for ACK 2
Waiting for ACK 3
Sending packet 4
Sending packet 5
Sending packet 6
Sending packet 7
Waiting for ACK 4
ACK 4 lost, resending packets 4 to 7
Waiting for ACK 4
Waiting for ACK 5
Waiting for ACK 6
Waiting for ACK 7
ACK 7 lost, resending packets 4 to 7
Waiting for ACK 7
Sending packet 8
Sending packet 9
Waiting for ACK 8
```

SELECTIVE REPETIVE ARQ

Selective Repeat ARQ (Automatic Repeat Request) is a protocol used in computer networks to ensure reliable data transmission between a sender and a receiver. It is an extension of the Go-Back-N ARQ protocol that provides additional functionality and efficiency.

In Selective Repeat ARQ, the sender maintains a larger sending window compared to the Go-Back-N protocol. The window size determines the number of packets that can be sent without waiting for acknowledgments. Unlike Go-Back-N, Selective Repeat ARQ allows individual packets within the window to be selectively retransmitted upon encountering errors or timeouts, instead of retransmitting the entire window.

WORKING OF SELECTIVE REPETIVE ARQ

The working of Selective Repeat ARQ can be explained step-by-step as follows:

- Sender Side
  - The sender divides the data into packets and assigns a unique sequence number to each packet.
  - The sender maintains a sending window that can hold multiple packets.
  - It starts a timer for each packet that is sent and awaits

acknowledgments.

- Packet Sending
    - The sender sends packets within the sending window to the receiver.
    - The packets are transmitted over the network to the receiver.
    - The sender keeps track of the sequence numbers of the packets it has sent and waits for acknowledgments.

- Receiver Side
    - The receiver receives the packets sent by the sender.
    - It checks the received packets for errors using techniques like checksum or CRC (Cyclic Redundancy Check).
    - If a packet is error-free, it sends an acknowledgment (ACK) with the sequence number of the received packet back to the sender.
    - If a packet has errors, the receiver discards the packet and does not send an ACK for that packet.

- Packet Retransmission
    - Upon receiving an ACK, the sender marks the corresponding packet as successfully acknowledged.
    - If the sender does not receive an ACK within a specified timeout duration for a particular packet, it assumes the packet was lost or corrupted.
    - In Selective Repeat ARQ, instead of retransmitting the entire window, the sender selectively retransmits only the unacknowledged packets within the sending window.
    - The sender retransmits the unacknowledged packets, starting from the earliest unacknowledged packet.

- Receiver Buffering
    - The receiver buffers the received packets, even if they arrive out of order.
    - It maintains a receive window that can store multiple packets.
    - Once all the packets up to a certain sequence number (known as the window size) are received, the receiver delivers the packets to the higher layer in the correct order.

- Acknowledgment Handling
    - The sender receives the ACKs from the receiver, indicating the successful receipt of packets.
    - If an ACK is received for a specific packet, the sender marks that packet as acknowledged.

    - If duplicate ACKs are received or if a timeout occurs for a specific packet, the sender knows that the corresponding packet was lost or corrupted and needs to be retransmitted.

- Window Management
    - Both the sender and receiver maintain their respective windows.
    - The sender adjusts the size of the sending window dynamically based on network conditions and feedback from the receiver.
    - The receiver adjusts the size of the receive window based on its buffer capacity and flow control mechanisms.

- The process of sending, receiving, and selectively retransmitting packets continues until all the packets are successfully transmitted and acknowledged. Selective Repeat ARQ provides more efficient retransmission by retransmitting only the necessary packets, resulting in improved network efficiency and throughput compared to Go-Back-N ARQ.

Overall, Selective Repeat ARQ ensures reliable data transmission by using selective retransmission and receiver buffering, allowing for error recovery and out-of-order packet delivery in computer networks.

USES AND PURPOSE

Here are some of the main uses and purposes of Selective Repeat ARQ:

- Reliable Data Transfer: The primary purpose of Selective Repeat ARQ is to ensure reliable data transfer between a sender and receiver in the presence of errors or packet loss. By selectively retransmitting only the necessary packets, it improves the reliability of data transmission.

- Error Recovery: Selective Repeat ARQ provides efficient error recovery mechanisms. When a packet is lost or corrupted, the protocol allows for the retransmission of only the affected packet(s), minimizing the overhead of retransmitting the entire window or sequence of packets.

- Congestion Control: Selective Repeat ARQ contributes to congestion control in network communication. By selectively retransmitting only the necessary packets, it helps in reducing unnecessary retransmissions and network congestion.

- Out-of-Order Packet Delivery: The receiver in Selective Repeat ARQ buffers out-of-order packets until all the preceding packets are received. This allows for the delivery of packets in the correct order to the higher layer at the receiver, even if they arrive out of order over the network.

- Efficient Network Utilization: Selective Repeat ARQ optimizes network utilization by utilizing the available bandwidth efficiently. It avoids retransmitting packets that have already been successfully received and acknowledged, reducing redundant transmissions and maximizing network throughput.

- Flexibility in Window Size: Selective Repeat ARQ allows for flexibility in choosing the window size. The sender can adjust the window size based on network conditions, optimizing the trade-off between reliability and efficiency according to the specific requirements of the network.

- Increased Throughput: By selectively retransmitting only the necessary packets and avoiding unnecessary retransmissions, Selective Repeat ARQ increases the throughput of the network. It reduces the impact of packet loss on overall data transmission, leading to improved network performance.

WHERE IT IS USED

- Selective Repeat ARQ (Automatic Repeat Request) is used in various network communication scenarios where reliable and efficient data transmission is required. Some common applications and use cases of Selective Repeat ARQ include:

- Wireless Networks: Selective Repeat ARQ is widely used in wireless communication systems, including cellular networks (e.g., 3G, 4G, 5G), Wi-Fi networks, and satellite communication. These networks often experience packet loss, interference, and fading, and Selective Repeat ARQ helps to ensure reliable transmission in such environments.

- Internet Protocols: Selective Repeat ARQ is used in transport layer protocols, particularly in TCP (Transmission Control Protocol). TCP utilizes Selective Repeat ARQ to provide reliable, in-order data delivery over IP networks. It is a fundamental mechanism for error recovery and congestion control in TCP.

- Multimedia Streaming: Selective Repeat ARQ is employed in multimedia streaming applications, such as video streaming and real-time audio transmission. These applications require continuous and uninterrupted data delivery. Selective Repeat ARQ helps to minimize interruptions and deliver a smooth streaming experience by selectively retransmitting lost or corrupted packets.

- File Transfer Protocols: Selective Repeat ARQ is utilized in file transfer protocols like FTP (File Transfer Protocol) and TFTP (Trivial File Transfer Protocol). These protocols ensure the reliable transfer of files over networks, and Selective Repeat ARQ plays a crucial role in recovering lost packets during the transfer process.

- Network Switching and Routing: Selective Repeat ARQ can be employed in network switching and routing protocols to handle packet losses and ensure reliable data delivery between network nodes. It helps to overcome transmission errors and efficiently recover lost packets in the network infrastructure.

- Remote Sensing and Telemetry Systems: Selective Repeat ARQ is used in remote sensing and telemetry systems, such as environmental monitoring, remote data collection, and industrial control systems. These systems often rely on wireless communication links, and Selective Repeat ARQ aids in ensuring the accurate and reliable transmission of sensor data.

- High-Speed Networks: Selective Repeat ARQ is beneficial in high-speed networks where data transmission occurs at a rapid rate. It helps to manage potential packet losses, minimize retransmissions, and maintain efficient throughput in these networks.

- These are just a few examples of the applications and use cases where Selective Repeat ARQ is used. The protocol's ability to provide reliable data transmission, handle errors, and optimize network performance makes it a crucial component in various network communication scenarios.

DIAGRAPHIC REPRESENTAION

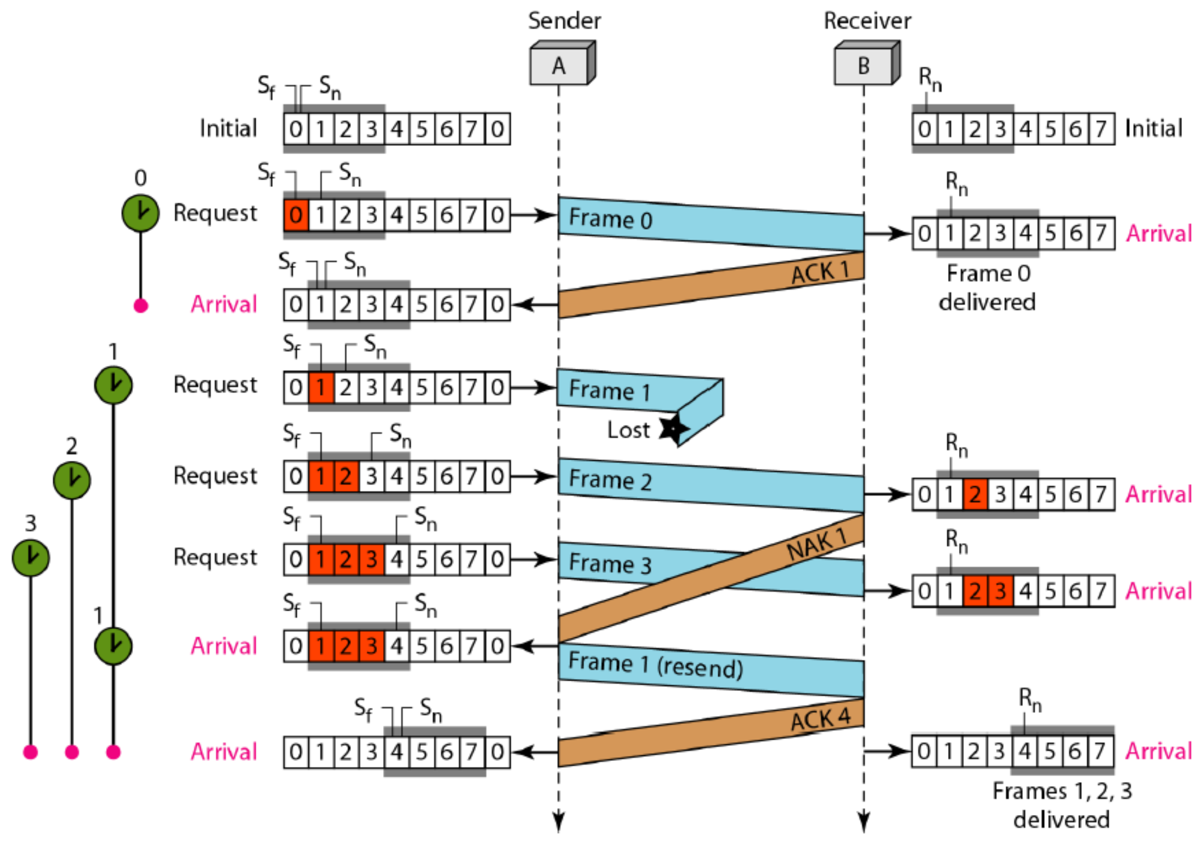


Fig 1.1 Selective repeat arq

HOW IT IS IMPLEMENTED

We implement this Selective Repeat ARQ by C language
Here is the Detail Explain:

Header Files

The necessary header files `stdio.h`, `stdlib.h`, and `time.h` are included for input/output operations, memory allocation, and time-related functions.

Constants

Several constants are defined:
`WINDOW_SIZE`: Represents the size of the sliding window, indicating the maximum number of packets that can be sent without waiting for acknowledgments.
`PACKET_SIZE`: Represents the size of each packet's data.
`TOTAL_PACKETS`: Indicates the total number of packets to be transmitted.

`TIMEOUT`: Specifies the timeout duration (in seconds) for retransmitting packets that have not received acknowledgments.

Packet Structure

The `Packet` structure represents a packet and contains the following fields:
`seq_num`: Stores the sequence number of the packet.
`data`: Holds the data content of the packet.
`ack_received`: Indicates whether the acknowledgment for this packet has been received (0 for not received, 1 for received).
`send_time`: Records the time at which the packet was sent.

Packet Array Initialization

An array of `Packet` structures named `packets` is declared with a size of `TOTAL_PACKETS`.
The packets are initialized with sequence numbers, acknowledgment flags set to 0, and the remaining fields uninitialized.

`send_packet` Function

The `send_packet` function takes a sequence number as input and simulates sending the corresponding packet.
It prints a message indicating the packet being sent and sets the send time for the packet.

`receive_ack` Function

The `receive_ack` function simulates the reception of an acknowledgment from the receiver.
It prompts the user to enter an ACK number and returns it as an integer.

Main Function

The `main` function initializes the base and next sequence numbers to 0.
It enters a while loop that continues until all packets have been transmitted and acknowledged.

Sending Packets

Within the while loop, the code sends packets within the sliding window that have not received acknowledgments.
It iterates from the base packet to the next sequence number plus the window size (limited by `TOTAL_PACKETS`).If a packet has not received an acknowledgment and has exceeded the timeout duration, it calls the `send_packet` function for retransmission.

Receiving ACKs

After sending the packets, the code enters a nested while loop to receive acknowledgments.

It prompts the user to enter an ACK number using the `receive_ack` function.

If the ACK number falls within the current sliding window range, the corresponding packet's `ack_received` flag is set to 1.

If the received ACK number is equal to the base, indicating the acknowledgment for the oldest unacknowledged packet, the code slides the window by incrementing the base until the next unacknowledged packet.

The next sequence number is updated to be the base plus the window size.

Completion and Termination

The while loop continues until all packets have been successfully transmitted and acknowledged.

Once the loop exits, a message is printed indicating the successful transmission of all packets.

The program terminates by returning 0.

EXAMPLE

Assuming the following values for the constants:
- WINDOW_SIZE: 4
- PACKET_SIZE : 8
- TOTAL_PACKETS : 12
- TIMEOUT : 3 seconds

Example Execution

```
Sending packet 0...
Sending packet 1...
Sending packet 2...
Sending packet 3...
Enter ACK number: 0
Enter ACK number: 2
Enter ACK number: 1
Enter ACK number: 3
Enter ACK number: 5
Enter ACK number: 6
Enter ACK number: 7
Enter ACK number: 4
Enter ACK number: 8
Enter ACK number: 9
Enter ACK number: 10
Enter ACK number: 11
All packets successfully transmitted!
```

Explanation

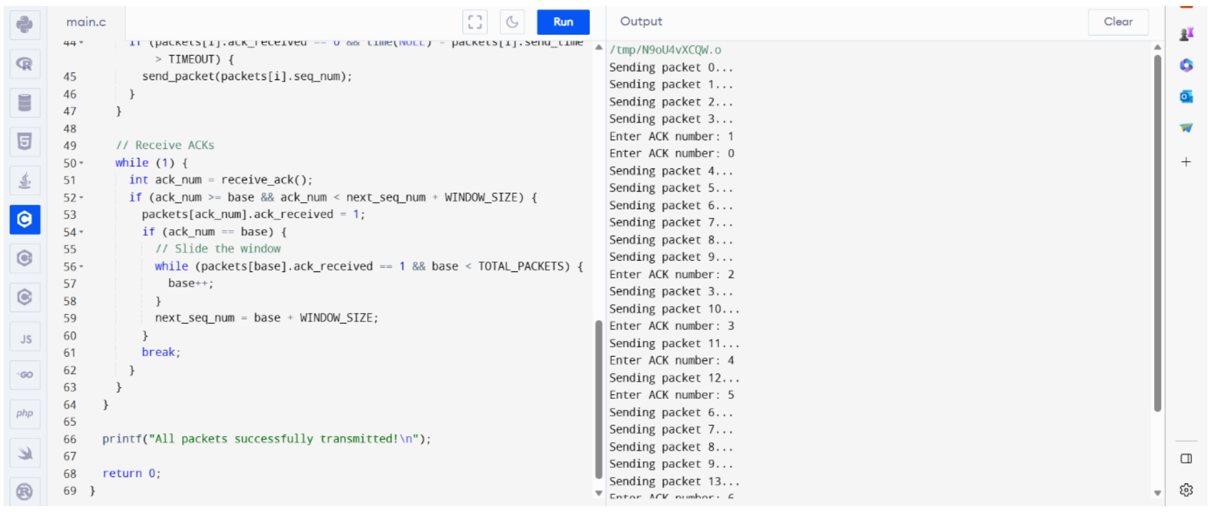
- Initially, the base and next sequence numbers are both 0.
- The code sends packets 0 to 3 since the window size is 4.
- The receiver sends acknowledgments (ACKs) based on the prompts.
- The first ACK received is for packet 0. It slides the window, so the base is incremented to 1.
- The next sequence number becomes base + window size, i.e., 5.
- Packet 2 receives an acknowledgment next followed by packet 1 and packet3.
- The window continues to slide, and packets are acknowledged in sequence.
- Eventually, all packets are transmitted successfully, and the code prints the message "All packets successfully transmitted!"

Let's analyze the progress of the sliding window protocol based on the ACKs received:

- ACK 0: Acknowledgment for packet 0. It slides the window, and the base becomes 1.
- ACK 2: Acknowledgment for packet 2.
- ACK 1: Acknowledgment for packet 1.
- ACK 3: Acknowledgment for packet 3. The window slides again, and the base becomes 2.
- ACK 5, 6, 7, 4, 8, 9, 10, 11: Acknowledgments for the remaining packets. The window slides accordingly.
- Finally, all packets are successfully transmitted, and the program terminates.

This example demonstrates how the sliding window protocol ensures reliable transmission by using acknowledgments to track the progress of sent packets and sliding the window based on the received acknowledgments.

CODE:
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define WINDOW_SIZE 4
#define PACKET_SIZE 8
#define TOTAL_PACKETS 16
#define TIMEOUT 5 // in seconds

typedef struct Packet {
  int seq_num;
  char data[PACKET_SIZE];
  int ack_received;
time_tsend_time;
} Packet;

Packet packets[TOTAL_PACKETS];
void send_packet(int seq_num) {
printf("Sending packet %d...\n", seq_num);
  packets[seq_num].send_time = time(NULL);
}

int receive_ack() {
  int ack_num;
printf("Enter ACK number: ");
scanf("%d", &ack_num);
  return ack_num;
}

int main() {
  int base = 0, next_seq_num = 0;
  int i;

  // Initialize packets
  for (i = 0; i< TOTAL_PACKETS; i++) {
    packets[i].seq_num = i;
    packets[i].ack_received = 0;
  }
while (base < TOTAL_PACKETS) {
    // Send new packets
    for (i = base; i<next_seq_num + WINDOW_SIZE &&i<
TOTAL_PACKETS; i++) {
      if (packets[i].ack_received == 0 && time(NULL) - packets[i].send_time>
TIMEOUT) {
send_packet(packets[i].seq_num);
      }
    }

    // Receive ACKs
    while (1) {
      int ack_num = receive_ack();
      if (ack_num>= base &&ack_num<next_seq_num + WINDOW_SIZE) {
        packets[ack_num].ack_received = 1;
        if (ack_num == base) {
          // Slide the window
          while (packets[base].ack_received == 1 && base < TOTAL_PACKETS)
{
            base++;
          }
next_seq_num = base + WINDOW_SIZE;
        }
        break;
      }
    }
  }
printf("All packets successfully transmitted!\n");
return 0;}
```



```
      if (packets[i].ack_received == 0 && time(NULL) - packets[i].send_time
          > TIMEOUT) {
        send_packet(packets[i].seq_num);
      }
    }

    // Receive ACKs
    while (1) {
      int ack_num = receive_ack();
      if (ack_num >= base && ack_num < next_seq_num + WINDOW_SIZE) {
        packets[ack_num].ack_received = 1;
        if (ack_num == base) {
          // Slide the window
          while (packets[base].ack_received == 1 && base < TOTAL_PACKETS) {
            base++;
          }
          next_seq_num = base + WINDOW_SIZE;
        }
        break;
      }
    }
  }

  printf("All packets successfully transmitted!\n");

  return 0;
}
```

```
/tmp/N9oU4vXCQW.o
Sending packet 0...
Sending packet 1...
Sending packet 2...
Sending packet 3...
Enter ACK number: 1
Enter ACK number: 0
Sending packet 4...
Sending packet 5...
Sending packet 6...
Sending packet 7...
Sending packet 8...
Sending packet 9...
Enter ACK number: 2
Sending packet 3...
Sending packet 10...
Enter ACK number: 3
Sending packet 11...
Enter ACK number: 4
Sending packet 12...
Enter ACK number: 5
Sending packet 6...
Sending packet 7...
Sending packet 8...
Sending packet 9...
Sending packet 13...
Enter ACK number: 6
```

CONCLUSION

In short, the Go-Back-N ARQ protocol and Selective Repeat ARQ protocol are both used in computer networks to ensure reliable data transmission over unreliable channels.

Go-Back-N ARQ allows the sender to transmit multiple packets before receiving acknowledgments. If an acknowledgment is not received within a timeout period, the sender retransmits all unacknowledged packets in the window. It is simpler to implement but may lead to unnecessary retransmissions.

Selective Repeat ARQ also allows the sender to transmit multiple packets before receiving acknowledgments. If an acknowledgment is not received within a timeout period, the sender retransmits only the lost or damaged packets. It is more efficient and avoids unnecessary retransmissions of correctly received packets.

Go-Back-N ARQ is suitable for channels with higher error rates or limited receiver buffer space, while Selective Repeat ARQ is more efficient and performs better in channels with lower error rates or larger receiver buffer space.

Ultimately, the choice between the two protocols depends on the specific network conditions and requirements of the application.

REFERENCES
1)Computer Networks By Tanenbaum & Wetherall
2)Neso Academy (1) Go-Back-N ARQ - YouTube (1) selective repetivebyneso academy - YouTube
3)Scalar Academy(Website)