Name: Labhansh Naik Reg ID: 221070041
TY CE Batch B

CN LAB EXPERIMENT 7

Aim: Develop a program to simulate the Go-Back-N (GBN) and Selective Repeat (SR) modes of the Sliding Window Protocol in a peer-to-peer communication setup. Use Wireshark Packet Analyzer to capture and analyze the packet traces in this mode.

Theory:

The Sliding Window Protocol is a key mechanism in network communication, employed for managing flow control and error correction during data transmission. It enables the sender to transmit multiple frames without waiting for individual acknowledgments for each. This method is integral to protocols at the Transport Layer, such as TCP (Transmission Control Protocol).

A "sliding window" is maintained by the protocol, representing the range of frame sequence numbers eligible for sending or receiving at any given time. Two notable variants of the Sliding Window Protocol include:

1. **Go-Back-N (GBN)**
2. **Selective Repeat (SR)**

These protocols differ in how they handle retransmissions and acknowledgments, making them suitable for varying use cases.

Peer-to-Peer Communication:

In peer-to-peer (P2P) systems, each participant can function both as a sender and receiver, in contrast to the client-server model, where roles are fixed. This experiment demonstrates P2P communication using Python socket programming, where one instance of the program acts as the sender while another acts as the receiver.

Go-Back-N (GBN) Protocol:

Overview:
The Go-Back-N protocol allows the sender to transmit up to *N* frames (defined by the window size) without waiting for acknowledgments for each one. However, if a frame encounters an error or is lost, all subsequent frames are retransmitted, regardless of whether they were successfully received. This approach, though straightforward to implement, can lead to redundant retransmissions.

Working Mechanism:

1. The sender maintains a window of *N* frames.
2. Frames within the window are transmitted sequentially without waiting for acknowledgement for each.

3. If a timeout occurs due to a lost or delayed acknowledgment, the sender retransmits the unacknowledged frame and all frames following it.
4. This process repeats until all frames are successfully acknowledged by the receiver.

Example:

- The sender transmits frames 0, 1, 2, and 3.
- If frame 2 is lost, the receiver does not acknowledge it.
- Upon timeout, the sender retransmits frame 2 and all subsequent frames (e.g., 2 and 3), even if frame 3 had previously arrived at the receiver.

This approach simplifies implementation but increases the retransmission overhead, especially in environments with high packet loss.

client.py:

```python
import socket
import time

# Configuration for the client
server_details = ('localhost', 8080)
win_size = 4
current_base = 0
sequence_number = 0
response_timeout = 2

# Initialize UDP socket
udp_client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udp_client_socket.settimeout(response_timeout)

def transmit_packet(seq):
    packet = str(seq)
    udp_client_socket.sendto(packet.encode(), server_details)
    print(f"Packet {seq} has been sent")

while current_base < 10:  # Send a total of 10 packets for this example
    while sequence_number < current_base + win_size and sequence_number < 10:
        transmit_packet(sequence_number)
        sequence_number += 1

    try:
        while current_base < sequence_number:
            acknowledgment, _ = udp_client_socket.recvfrom(1024)
```

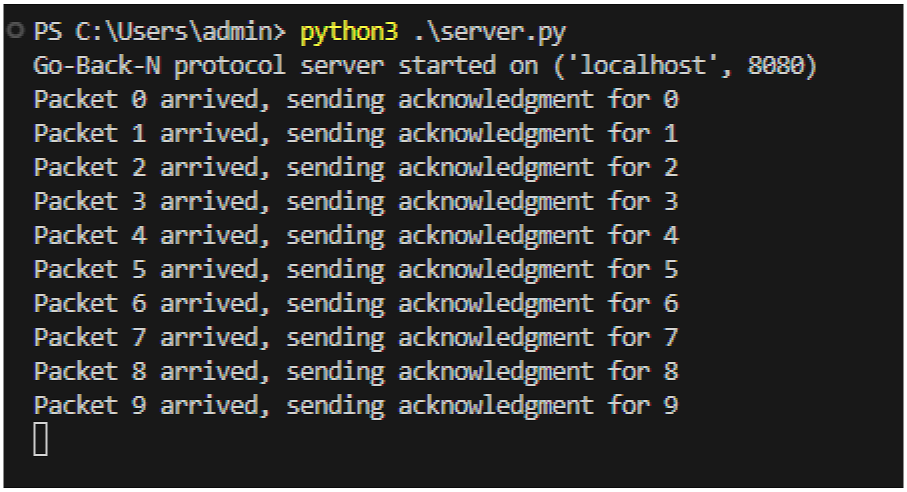
```
            ack_seq = int(acknowledgment.decode().split()[1])
            print(f"Received acknowledgment: {acknowledgment.decode()}")
            if ack_seq >= current_base:
                current_base = ack_seq + 1

    except socket.timeout:
        print("Timeout detected! Re-sending packets starting from current
base...")
        sequence_number = current_base

udp_client_socket.close()
```

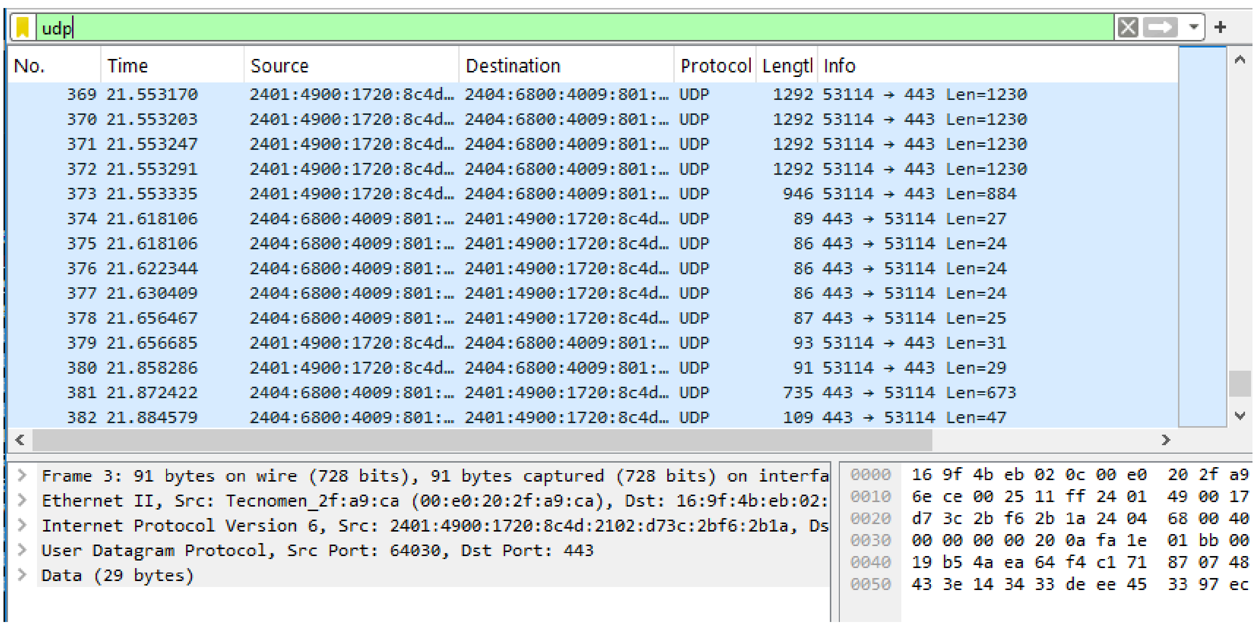
```
PS C:\Users\admin> python3 .\client.py
Packet 0 has been sent
Packet 1 has been sent
Packet 2 has been sent
Packet 3 has been sent
Received acknowledgment: ACK 0
Received acknowledgment: ACK 1
Received acknowledgment: ACK 2
Received acknowledgment: ACK 3
Packet 4 has been sent
Packet 5 has been sent
Packet 6 has been sent
Packet 7 has been sent
Received acknowledgment: ACK 4
Received acknowledgment: ACK 5
Received acknowledgment: ACK 6
Received acknowledgment: ACK 7
Packet 8 has been sent
Packet 9 has been sent
Received acknowledgment: ACK 8
Received acknowledgment: ACK 9
```

server.py:

```python
import socket
import random

# Server configuration
```

```
server_details = ('localhost', 8080)
win_size = 4
next_expected_seq = 0

# Initialize the UDP server socket
udp_server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udp_server_socket.bind(server_details)
print(f"Go-Back-N protocol server started on {server_details}")

while True:
    try:
        data, client_info = udp_server_socket.recvfrom(1024)
        received_seq = int(data.decode())

        # Introduce packet loss simulation with a 10% probability
        if random.random() < 0.1:
            print(f"Simulating loss of packet with sequence number
{received_seq}")
            continue

        if received_seq == next_expected_seq:
            print(f"Packet {received_seq} arrived, sending acknowledgment for
{received_seq}")
            ack_message = f"ACK {received_seq}"
            udp_server_socket.sendto(ack_message.encode(), client_info)
            next_expected_seq += 1
        else:
            print(f"Packet {received_seq} out of order, waiting for
{next_expected_seq}")
            ack_message = f"ACK {next_expected_seq - 1}"
            udp_server_socket.sendto(ack_message.encode(), client_info)

    except KeyboardInterrupt:
        print("Shutting down the server gracefully.")
        break

udp_server_socket.close()
```





Selective Repeat (SR) Protocol

Overview:

The Selective Repeat protocol enhances transmission efficiency by ensuring that only the frames that are lost or corrupted are retransmitted, rather than all subsequent frames. This method reduces redundant transmissions, but it requires more sophisticated handling of sequence numbers and acknowledgments by both the sender and the receiver.

Working Mechanism:

1. **Window Management:**
   - Both the sender and receiver maintain a sliding window of valid sequence numbers.
   - The receiver can accept frames that arrive out of order and temporarily buffer them until any missing frames are received.

2. **Acknowledgments:**
    - The receiver sends an individual acknowledgment (ACK) for each correctly received frame, regardless of the order.
3. **Retransmissions:**
    - The sender only retransmits frames for which it has not received an acknowledgment after the timeout expires.

Example:

- Suppose the sender transmits frames 0, 1, 2, and 3.
- If frame 2 is lost, the receiver still accepts frames 0, 1, and 3, storing frame 3 in a buffer.
- Upon detecting the loss of frame 2 (e.g., through a timeout), the sender retransmits only frame 2, which the receiver can then process, completing the sequence.

This approach optimizes network efficiency, particularly in environments with higher error rates, by minimizing unnecessary retransmissions. However, the added complexity of managing buffers and acknowledgments makes it more challenging to implement compared to the Go-Back-N protocol.

client.py:

```python
import socket
import time

# Client configuration
server_details = ('localhost', 8081)
win_size = 4
packet_tracker = {}
timeout_interval = 2

# Set up a UDP client socket
udp_client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udp_client_socket.settimeout(timeout_interval)

def transmit_packet(sequence_num):
    packet_data = str(sequence_num)
    udp_client_socket.sendto(packet_data.encode(), server_details)
    packet_tracker[sequence_num] = time.time()
    print(f"Packet {sequence_num} sent")

sequence_number = 0
while sequence_number < 10:  # Example: Send a total of 10 packets
```

```
    while sequence_number < sequence_number + win_size and sequence_number < 10:
        transmit_packet(sequence_number)
        sequence_number += 1

    try:
        while packet_tracker:
            acknowledgment, _ = udp_client_socket.recvfrom(1024)
            ack_sequence = int(acknowledgment.decode().split()[1])
            print(f"Received acknowledgment: {acknowledgment.decode()}")

            if ack_sequence in packet_tracker:
                del packet_tracker[ack_sequence]

    except socket.timeout:
        print("Timeout detected! Resending all unacknowledged packets...")
        for seq_num in list(packet_tracker):
            if time.time() - packet_tracker[seq_num] >= timeout_interval:
                transmit_packet(seq_num)

udp_client_socket.close()
```

```
PS C:\Users\admin> python3 .\client.py
Packet 0 sent
Packet 1 sent
Packet 2 sent
Packet 3 sent
Packet 4 sent
Packet 5 sent
Packet 6 sent
Packet 7 sent
Packet 8 sent
Packet 9 sent
Received acknowledgment: ACK 0
Received acknowledgment: ACK 1
Received acknowledgment: ACK 2
Received acknowledgment: ACK 3
Received acknowledgment: ACK 4
Received acknowledgment: ACK 5
Received acknowledgment: ACK 6
Received acknowledgment: ACK 7
Received acknowledgment: ACK 8
Received acknowledgment: ACK 9
```

Name: Labhansh Naik
TY CE

Reg ID: 221070041
Batch B

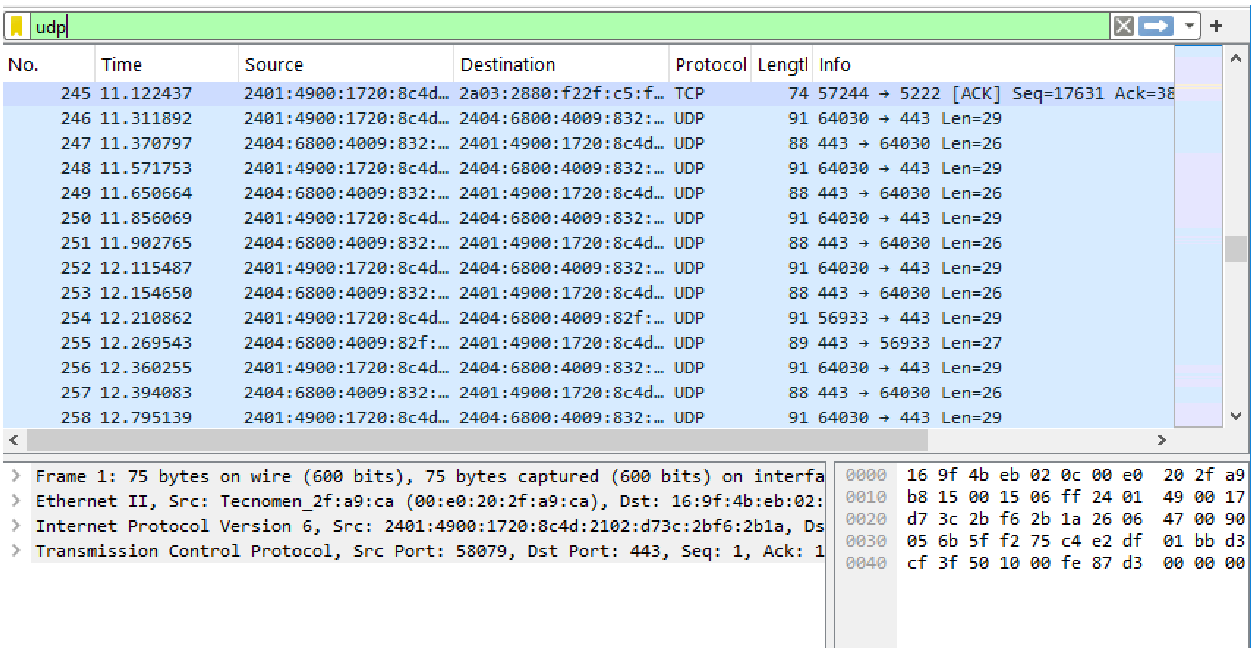
server.py:

```python
import socket
import random

# Server configuration
server_details = ('localhost', 8081)
win_size = 4
received_packets = {}

# Initialize a UDP server socket
udp_server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udp_server_socket.bind(server_details)
print(f"Selective Repeat Protocol Server started on {server_details}")

while True:
    try:
        data, client_info = udp_server_socket.recvfrom(1024)
        packet_seq_num = int(data.decode())

        # Simulating random packet loss with a 10% probability
        if random.random() < 0.1:
            print(f"Simulated loss for packet with sequence number
{packet_seq_num}")
            continue

        print(f"Packet {packet_seq_num} received successfully")
        received_packets[packet_seq_num] = True

        acknowledgment_message = f"ACK {packet_seq_num}"
        udp_server_socket.sendto(acknowledgment_message.encode(), client_info)

    except KeyboardInterrupt:
        print("Gracefully shutting down the server.")
        break

udp_server_socket.close()
```

Name: Labhansh Naik
TY CE

Reg ID: 221070041
Batch B

```
PS C:\Users\admin> python3 .\server.py
Selective Repeat Protocol Server started on ('localhost', 8081)
Packet 0 received successfully
Packet 1 received successfully
Packet 2 received successfully
Packet 3 received successfully
Packet 4 received successfully
Packet 5 received successfully
Packet 6 received successfully
Packet 7 received successfully
Packet 8 received successfully
Packet 9 received successfully
```



Conclusion:

This experiment provided an understanding of how the Go-Back-N (GBN) and Selective Repeat (SR) protocols handle flow control and error correction in a peer-to-peer communication setup. The GBN protocol offers simplicity in implementation but sacrifices efficiency by retransmitting all packets following a loss. In contrast, the SR protocol improves efficiency by retransmitting only the lost packets, albeit at the cost of greater complexity in managing sequence numbers and acknowledgments.