Implementing Reliable Data Transmission with an N-bit Sliding Window Protocol

Done by: Thota Surya Vara Prasad Rao, CSE-G, AP22110011059,

suryavaraprasasdrao_thota@srmap.edu.in

Submitted on: 28-09-2024.

Objective

The primary objective of this project was to gain hands-on experience in implementing the N-bit Sliding Window Protocol for reliable data transmission between two processes using Python socket programming. Through this project, I learned to manage the complexities of frame transmission, acknowledgment handling, and error detection, which are crucial for ensuring efficient communication across unreliable networks. The outcomes are observable in the successful simulation of frame loss, out-of-order frames, and closed connections.

Problem Statement

In modern network communication, ensuring reliable and efficient data transfer between sender and receiver is critical, particularly in scenarios where data packets may be lost or delivered out of order. This project aimed to address this problem by implementing an N-bit Sliding Window Protocol, which allows multiple frames to be sent before receiving acknowledgments, while effectively handling packet loss, frame retransmission, and maintaining window synchronization. The challenge was to ensure both processes—sender and receiver—work in tandem to guarantee that data is transmitted accurately and efficiently, even in an unreliable network environment.

CODE:

1. **sender.py**

```
import socket
import time
import random

# Define constants
N = 4  # N-bit, so window size will be 2^N
window_size = 2**N
total_frames = 10  # Total number of frames to send
host = 'localhost'
port = 12345  # Port to connect to the receiver

# Sender class
class Sender:
    def __init__(self, sock):
        self.window_start = 0
        self.window_end = window_size - 1
        self.next_frame_to_send = 0
        self.sock = sock

    def send_frame(self):
        if self.next_frame_to_send <= self.window_end:
            print(f"Sender: Sending frame
{self.next_frame_to_send}")
            self.sock.sendall(str(self.next_frame_to_send).encode())
            return self.next_frame_to_send
        else:
            print("Sender: Window full, waiting for ACK...")
            return None

    def receive_ack(self):
        ack = self.sock.recv(1024).decode()
        print(f"Sender: Received ACK for frame {ack}")
        ack = int(ack)
        if self.window_start <= ack <= self.window_end:
            self.window_start = ack + 1
            self.window_end = self.window_start + window_size - 1
            self.next_frame_to_send = self.window_start
        else:
```

```
            print(f"Sender: ACK {ack} is out of window bounds")

def main():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((host, port))
        sender = Sender(sock)

        while sender.next_frame_to_send < total_frames:
            frame = sender.send_frame()
            if frame is not None:
                time.sleep(1)  # Simulate transmission delay

                # Randomly simulate frame loss
                if random.random() < 0.8:  # 80% chance of
successful transmission
                    sender.receive_ack()
                else:
                    print(f"Sender: Frame {frame} lost during
transmission.")

            sender.next_frame_to_send += 1
            time.sleep(1)  # Simulate delay before sending the next
frame

if __name__ == "__main__":
    main()
```

2. receiver.py

```python
import socket

# Define constants
host = 'localhost'
port = 12345  # Port to listen on

# Receiver class
class Receiver:
    def __init__(self, conn):
        self.expected_frame = 0
        self.conn = conn

    def receive_frame(self):
        frame = self.conn.recv(1024).decode().strip()

        # If frame is empty, the connection has likely closed
        if not frame:
            print("Receiver: Connection closed by the sender.")
            return False

        # Check if the frame received is valid
        try:
            frame = int(frame)
            if frame == self.expected_frame:
                print(f"Receiver: Received expected frame {frame},
sending ACK")
                self.conn.sendall(str(frame).encode())
                self.expected_frame += 1
            else:
                print(f"Receiver: Frame {frame} out of order,
expected {self.expected_frame}. Ignoring.")
                self.conn.sendall(str(self.expected_frame -
1).encode())
        except ValueError:
            print(f"Receiver: Received invalid frame data '{frame}',
ignoring.")

        return True
```

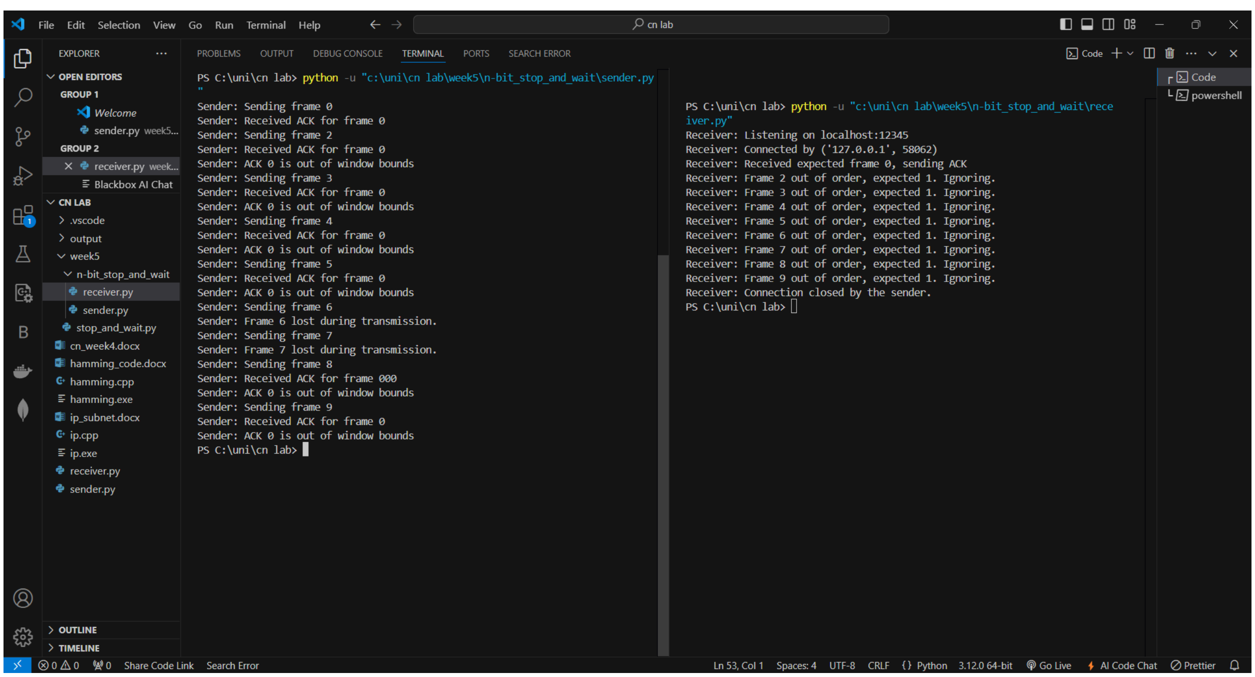
```python
def main():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.bind((host, port))
        sock.listen(1)
        print(f"Receiver: Listening on {host}:{port}")

        conn, addr = sock.accept()
        with conn:
            print(f"Receiver: Connected by {addr}")
            receiver = Receiver(conn)

            # Continue receiving frames until connection is closed
            while receiver.receive_frame():
                pass  # The loop continues until the connection is closed

if __name__ == "__main__":
    main()
```

OUTPUTS:



Problems Faced

One major challenge during development was managing the connection closure. Initially, the receiver process would continuously print error messages after the sender closed the connection. This issue was resolved by detecting an empty string returned by the recv() function, which signified the closed connection. Additionally, synchronizing the sliding window was challenging, particularly when simulating lost frames. I implemented an acknowledgment system that ensured that the sender would resend unacknowledged frames without duplicating already acknowledged frames, leading to a robust solution.

Conclusion

Through this project, I significantly enhanced my understanding of network communication protocols, particularly the sliding window technique used for reliable data transfer. I developed practical skills in Python socket programming and learned how to handle real-world network issues like packet loss, frame retransmission, and connection closure. Debugging the protocol deepened my understanding of error-handling in distributed systems, and I acquired the patience and analytical skills needed to resolve complex communication issues. Overall, this project fostered both technical knowledge and problem-solving resilience.