Implementing a 1-bit Stop-and-Wait Protocol for Reliable Data Transmission

Done by: Thota Surya Vara Prasad Rao, CSE-G, AP22110011059,
suryavaraprasasdrao_thota@srmap.edu.in
Submitted on: 28-09-2024

Objective Statement

The objective of this project is to implement a reliable data transmission protocol at the Data Link Layer using the 1-bit Stop-and-Wait mechanism. This protocol ensures that each frame sent by the sender is acknowledged by the receiver before sending the next frame, thereby providing a mechanism for handling lost or corrupted frames in an unreliable network environment. The primary learning outcome is understanding how a fundamental ARQ (Automatic Repeat reQuest) protocol operates in ensuring reliable communication and how to handle retransmissions and errors in data transmission.

Problem Statement

In network communication, data transmission is often subject to packet loss, corruption, and acknowledgment failures. A reliable protocol is required to ensure that all frames are transmitted successfully. The Stop-and-Wait ARQ (Automatic Repeat reQuest) protocol solves this issue by having the sender wait for an acknowledgment (ACK) for each frame sent, before proceeding with the next frame. If an ACK is not received within a specific time, the sender retransmits the frame. The challenge lies in handling network unreliability, timeouts, and frame corruption, while maintaining the simplicity of communication using alternating frame numbering.

Code:

```
import time

import random


# Constants to simulate network conditions and frame transmission

TIMEOUT_DURATION = 2  # Time in seconds to wait for ACK before resending

FRAME_COUNT = 5  # Number of frames to be sent

PACKET_LOSS_PROBABILITY = 0.1  # Probability of losing a packet

ACK_LOSS_PROBABILITY = 0.1  # Probability of ACK loss

FRAME_CORRUPTION_PROBABILITY = 0.1  # Probability of frame corruption


# Sender function implementing the Stop-and-Wait ARQ protocol

def sender():

    frame_number = 0  # Start with frame 0

    sent_frames = 0  # Track successfully sent frames


    # Loop until all frames are successfully transmitted

    while sent_frames < FRAME_COUNT:
```

```
print(f"Sender: Sending frame {frame_number}")

# Simulate random transmission delay
time.sleep(random.uniform(0.5, 1.5))

# Simulate potential frame corruption
if random.random() < FRAME_CORRUPTION_PROBABILITY:
    print(f"Sender: Frame {frame_number} corrupted. Retransmitting.")
    continue

# Timer starts to await acknowledgment
start_time = time.time()
ack_received = False  # Flag to track if ACK was received

while not ack_received:
    # Call receiver to check for ACK
    if receiver(frame_number):
        print(f"Sender: Received ACK for frame {frame_number}")
        frame_number = 1 - frame_number  # Toggle frame number (0 or 1)
```

```
            sent_frames += 1  # Increment the count of successfully sent frames
            ack_received = True  # ACK received, proceed to next frame
        else:
            # Check if timeout occurred
            if time.time() - start_time > TIMEOUT_DURATION:
                print(f"Sender: Timeout! Resending frame {frame_number}.")
                break  # Retransmit the frame after timeout

    print("Sender: All frames successfully transmitted.")


# Receiver function simulating the receipt of frames
def receiver(expected_frame):
    # Simulate random packet loss
    if random.random() < PACKET_LOSS_PROBABILITY:
        print(f"Receiver: Frame {expected_frame} lost in transmission.")
        return False

    # Simulate frame corruption during transmission
    if random.random() < FRAME_CORRUPTION_PROBABILITY:
```

```
        print(f"Receiver: Frame {expected_frame} received but corrupted.")
        return False

    print(f"Receiver: Frame {expected_frame} received correctly.")

    # Simulate delay in acknowledgment transmission
    time.sleep(random.uniform(0.5, 1.0))

    # Simulate ACK loss
    if random.random() < ACK_LOSS_PROBABILITY:
        print(f"Receiver: ACK for frame {expected_frame} lost.")
        return False

    print(f"Receiver: Sending ACK for frame {expected_frame}")
    return True

if __name__ == "__main__":
    sender()  # Start the sender process
```

OUTPUTS:

```
PS C:\uni\cn lab> python -u "c:\uni\cn lab\week5\stop_and_wait.py"
Sender: Sending frame 0
Receiver: Received frame 0 successfully.
Receiver: ACK for frame 0 lost in transmission.
Receiver: Received frame 0 successfully.
Receiver: Sending ACK for frame 0
Sender: Received ACK for frame 0
Sender: Sending frame 1
Receiver: Frame 1 lost in transmission.
Receiver: Received frame 1 successfully.
Receiver: Sending ACK for frame 1
Sender: Received ACK for frame 1
Sender: Sending frame 0
Receiver: Received frame 0 successfully.
Receiver: Sending ACK for frame 0
Sender: Received ACK for frame 0
Sender: Sending frame 1
Receiver: Received frame 1 successfully.
Receiver: Sending ACK for frame 1
Sender: Received ACK for frame 1
Sender: Sending frame 0
Receiver: Received frame 0 successfully.
Receiver: Sending ACK for frame 0
Sender: Received ACK for frame 0
Sender: All frames transmitted successfully.
PS C:\uni\cn lab>
```

Problems Faced During Development and Implementation

During the implementation, one of the key challenges was simulating real-world network conditions such as packet loss, ACK loss, and frame corruption. Another challenge was setting appropriate timeout intervals to ensure retransmissions occur efficiently without unnecessary delays. Fine-tuning the timeout and balancing the probability of errors helped achieve a more realistic simulation of the protocol. Handling frame numbering and ensuring proper toggling of bits (0 or 1) was also critical to the Stop-and-Wait mechanism.

Conclusion (Knowledge, Skills, Attitude Acquired)

Through this project, I gained a deeper understanding of how fundamental ARQ protocols like Stop-and-Wait function at the Data Link Layer. I learned to manage retransmissions, handle network unreliability (packet loss, corruption), and implement timeouts for ensuring reliable communication. This experience also helped improve my problem-solving skills, especially in scenarios where multiple components of the system (sender, receiver, and network) interact under imperfect conditions. Finally, I acquired hands-on experience with simulating network protocols and optimizing their performance.