VISVESVARAYA TECHNOLOGICAL UNIVERSITY,
BELAGAVI – 590 018

ACTIVITY-BASED ASSESSMENT REPORT ON

"Implementation of Stop and Wait Protocol with Socket Programming"

By

YASHWANTH S [4VV22CS189]
VENKATESHA Y K [4VV22CS181]
YESH GOWDA [4VV22CS190]

Submitted in the partial fulfillment of the requirement for the "COMPUTER NETWORKS (BCSCN501)" ABA marks

COURSE HANDLED BY

Mr. Nitesh A
Assistant Professor
Department of CS&E
VVCE, Mysuru

2024 - 2025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
VIDYAVARDHAKA COLLEGE OF ENGINEERING
MYSURU - 570 002

Implementation of Stop and Wait Protocol Using Socket Programming

1.Abstract

The Stop-and-Wait protocol is a fundamental mechanism used in reliable data communication to ensure error-free transmission of frames between a sender and receiver. It operates by sending one frame at a time and waiting for an acknowledgment (ACK) before transmitting the next frame. While this approach simplifies implementation and ensures reliability, it suffers from significant inefficiencies in modern high-speed networks. the key limitation arises from its sequential nature: only one frame can be transmitted at a time, which leads to poor utilization of network bandwidth, especially in scenarios with high latency (e.g., longdistance communication).

Additionally, in environments with frequent acknowledgment loss or transmission errors, the protocol's reliance on retransmission after a timeout increases delays, further impacting throughput. despite its simplicity and suitability for low-speed, error-prone networks, the Stop-and-Wait protocol becomes impractical in high-speed communication systems, where more advanced protocols like Go-Back-N or Selective Repeat are necessary to address its inefficiencies and improve performance. This work explores its principles, challenges, and real-world implications.

2. Primitives of Stop and Wait Protocol

Sender's Side

Rule 1: The sender sends one data packet at a time.

Rule 2: The sender only sends the subsequent packet after getting the preceding packet's acknowledgement.

Therefore, the concept behind the stop and wait protocol on the sender's end is relatively straightforward: Send one packet at a time and refrain from sending any additional packets until you have received an acknowledgement.

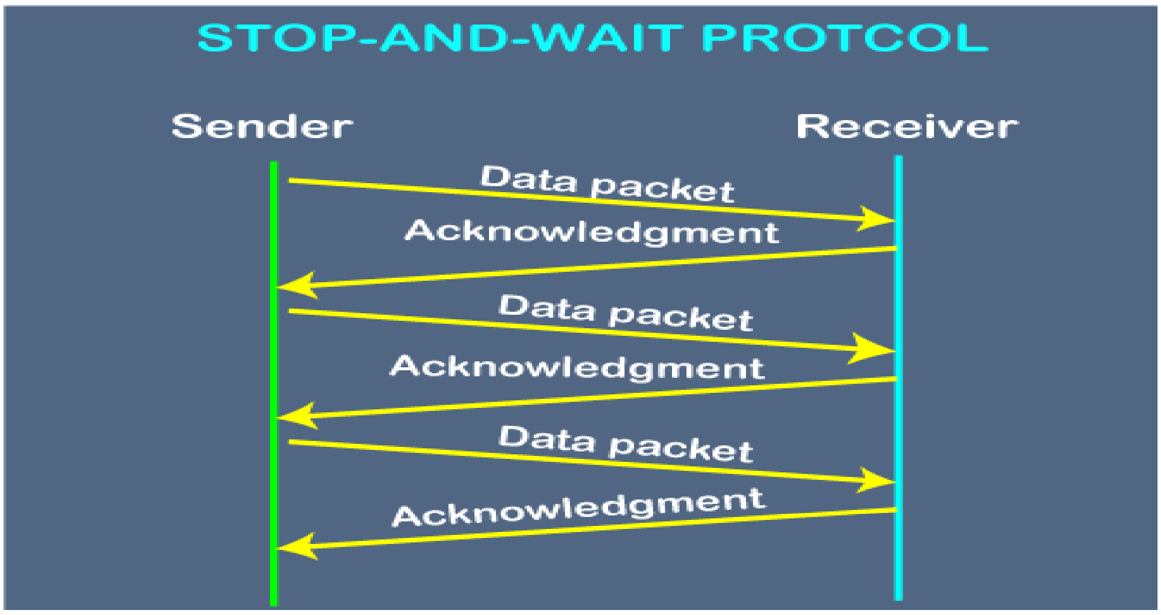
Receiver's Side

Rule 1: Receive the data packet, then consume it.

Rule 2: The receiver provides the sender with an acknowledgement after consuming the data packet.

As a result, the stop and wait protocol's basic tenet on the receiver's end is extremely straightforward: Ingest the packet, and after it has been consumed, send the acknowledgement. This is a mechanism for flow control.

3. Working of Stop and Wait Protocol



The stop and wait protocol's operation is depicted in the above figure. The sender sends the packet, referred to as a data packet if there is a sender and a receiver. After receiving a data packet, the receiver sends an acknowledgement. Without receiving acknowledgement for the first packet, the sender won't send the second packet. The sender sends the subsequent packet after receiving the acknowledgement.

This procedure keeps going until all of the packets have been sent. The stop and wait protocol's main benefit is simplicity, but there are some drawbacks as well. For instance, if 1000 data packets need to be delivered, they cannot all be sent at once since this protocol only sends one packet at a time.

4. Implementation of Stop and wait protocol in Python Code Snippet with Output

The implementation of the Stop-and-Wait protocol requires creating two separate files: one for the **Sender** and another for the **Receiver**. Each file contains specific code to handle its role in the protocol.

1. Sender File (sender.py)

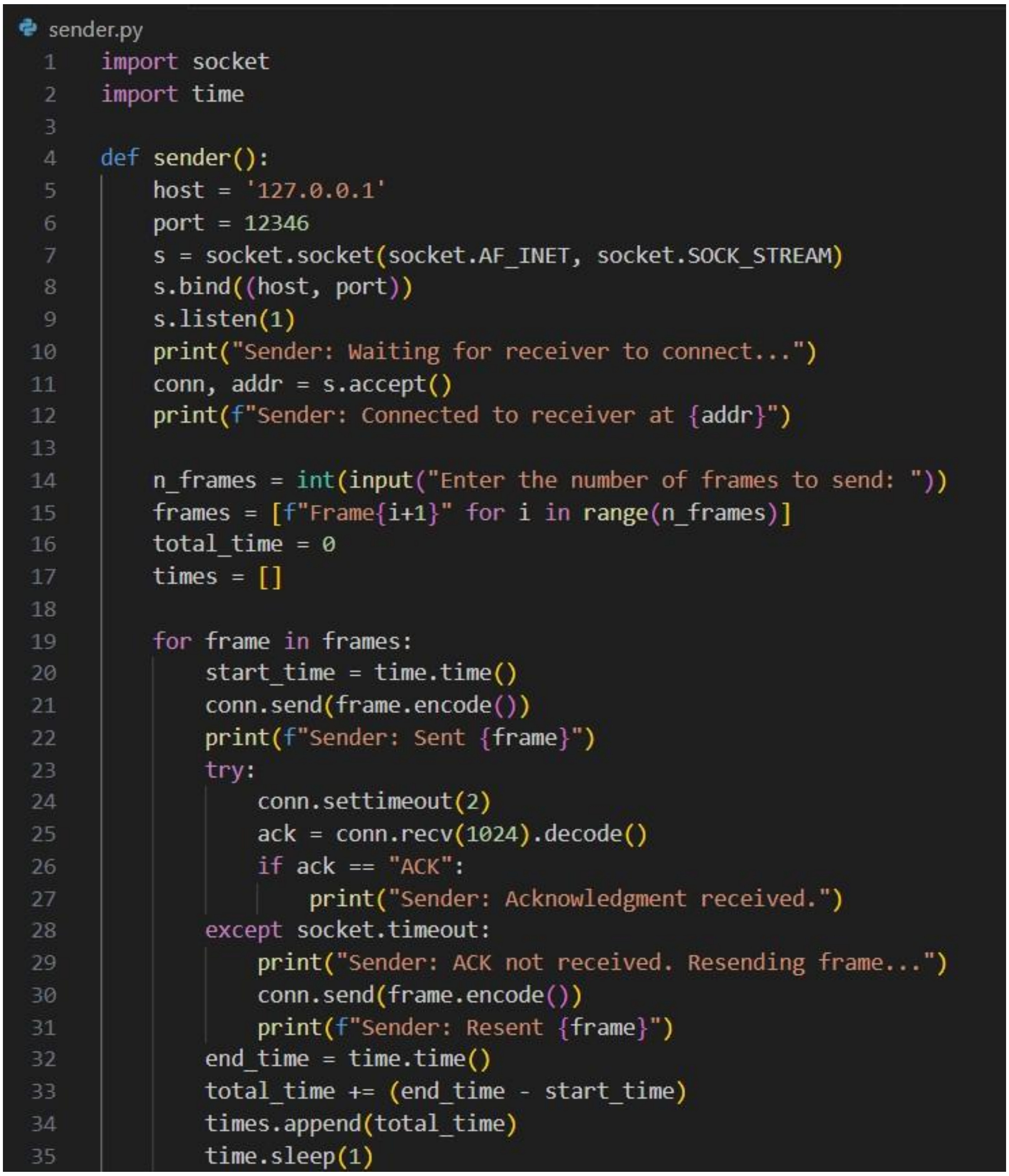
The **Sender** is responsible for sending frames to the receiver and waiting for acknowledgments before proceeding to the next frame.

Key Components

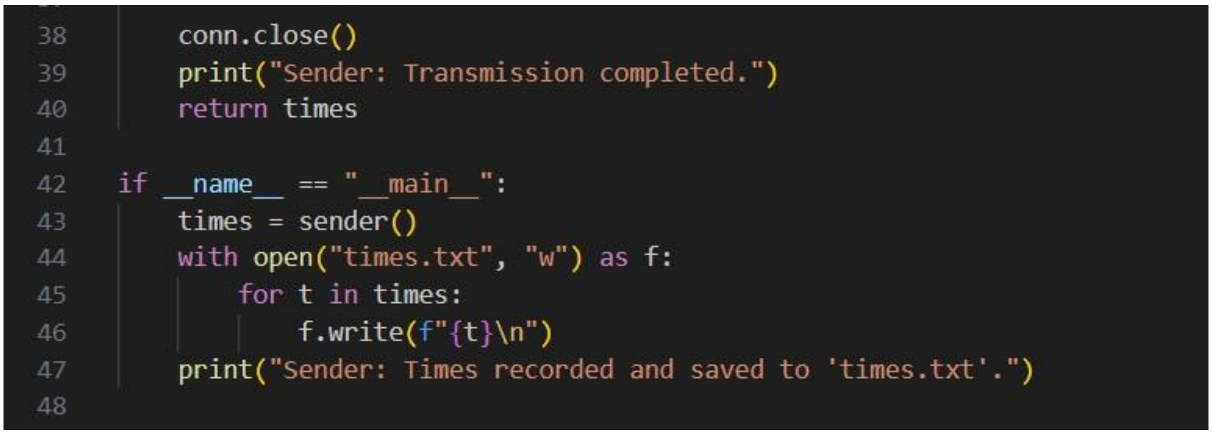
- Input: Number of frames is taken from the user via input().

- Frame Sending: Sends one frame at a time and waits for acknowledgment.
- Resending Logic: Retransmits a frame if acknowledgment isn't received within the timeout.
- Time Recording: Records the time taken for each frame (including retransmissions) and saves it to a file times.txt.

Contents of **sender.py:**



```
sender.py
import socket
import time

def sender():
    host = '127.0.0.1'
    port = 12346
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((host, port))
    s.listen(1)
    print("Sender: Waiting for receiver to connect...")
    conn, addr = s.accept()
    print(f"Sender: Connected to receiver at {addr}")

    n_frames = int(input("Enter the number of frames to send: "))
    frames = [f"Frame{i+1}" for i in range(n_frames)]
    total_time = 0
    times = []

    for frame in frames:
        start_time = time.time()
        conn.send(frame.encode())
        print(f"Sender: Sent {frame}")
        try:
            conn.settimeout(2)
            ack = conn.recv(1024).decode()
            if ack == "ACK":
                print("Sender: Acknowledgment received.")
        except socket.timeout:
            print("Sender: ACK not received. Resending frame...")
            conn.send(frame.encode())
            print(f"Sender: Resent {frame}")
        end_time = time.time()
        total_time += (end_time - start_time)
        times.append(total_time)
        time.sleep(1)
```

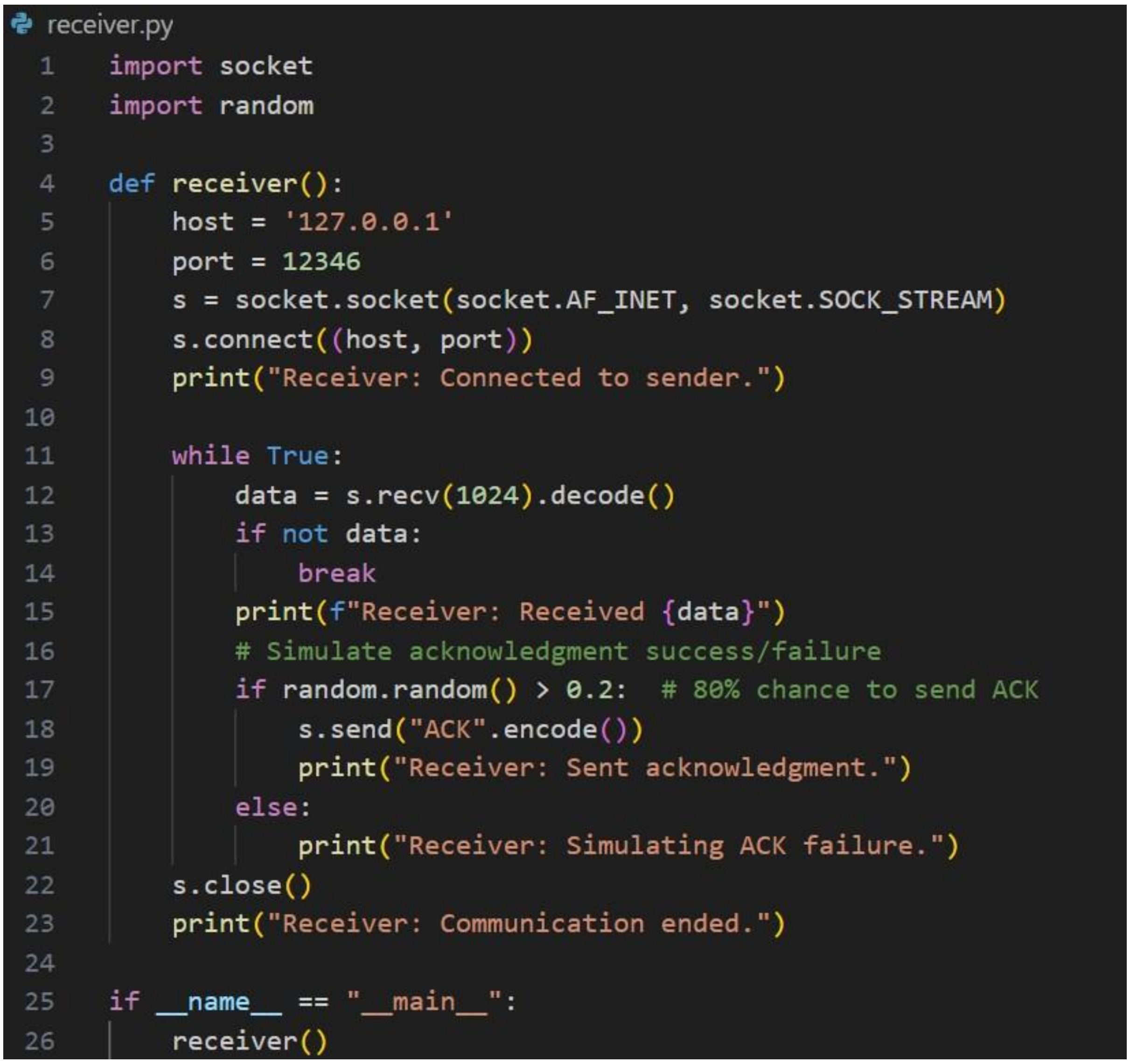


```python
    conn.close()
    print("Sender: Transmission completed.")
    return times

if __name__ == "__main__":
    times = sender()
    with open("times.txt", "w") as f:
        for t in times:
            f.write(f"{t}\n")
    print("Sender: Times recorded and saved to 'times.txt'.")

```

2. Receiver File (receiver.py)

The Receiver is responsible for receiving frames from the sender, processing them, and sending back acknowledgments.

Contents of **receiver.py:**



```python
import socket
import random

def receiver():
    host = '127.0.0.1'
    port = 12346
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((host, port))
    print("Receiver: Connected to sender.")

    while True:
        data = s.recv(1024).decode()
        if not data:
            break
        print(f"Receiver: Received {data}")
        # Simulate acknowledgment success/failure
        if random.random() > 0.2:  # 80% chance to send ACK
            s.send("ACK".encode())
            print("Receiver: Sent acknowledgment.")
        else:
            print("Receiver: Simulating ACK failure.")
    s.close()
    print("Receiver: Communication ended.")

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

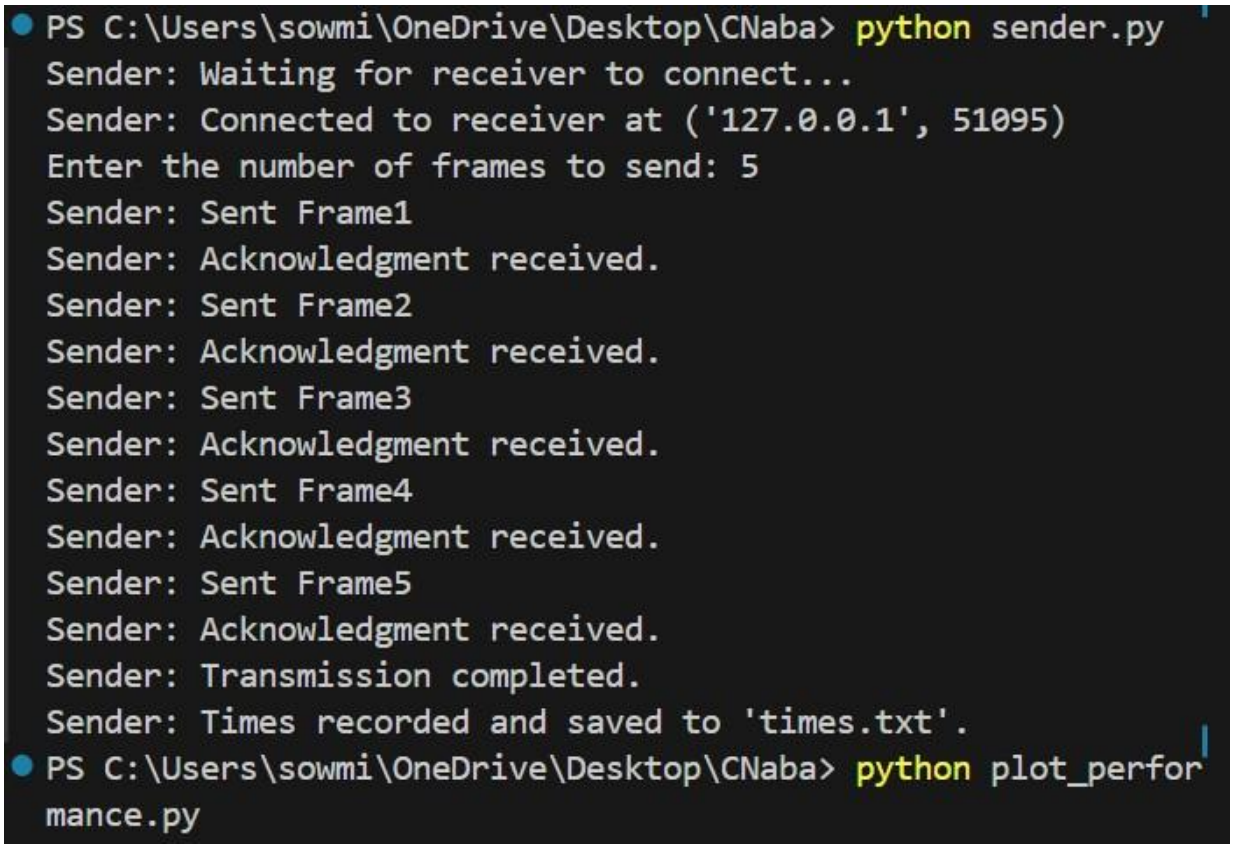
Key Components

- Frame Reception: Waits for frames sent by the sender.
- Acknowledgment Logic: Sends "ACK" back to the sender if the acknowledgment is simulated to be successful.
- Simulated Failure: Introduces randomness to simulate an acknowledgment failure (20% chance).

OUTPUT:

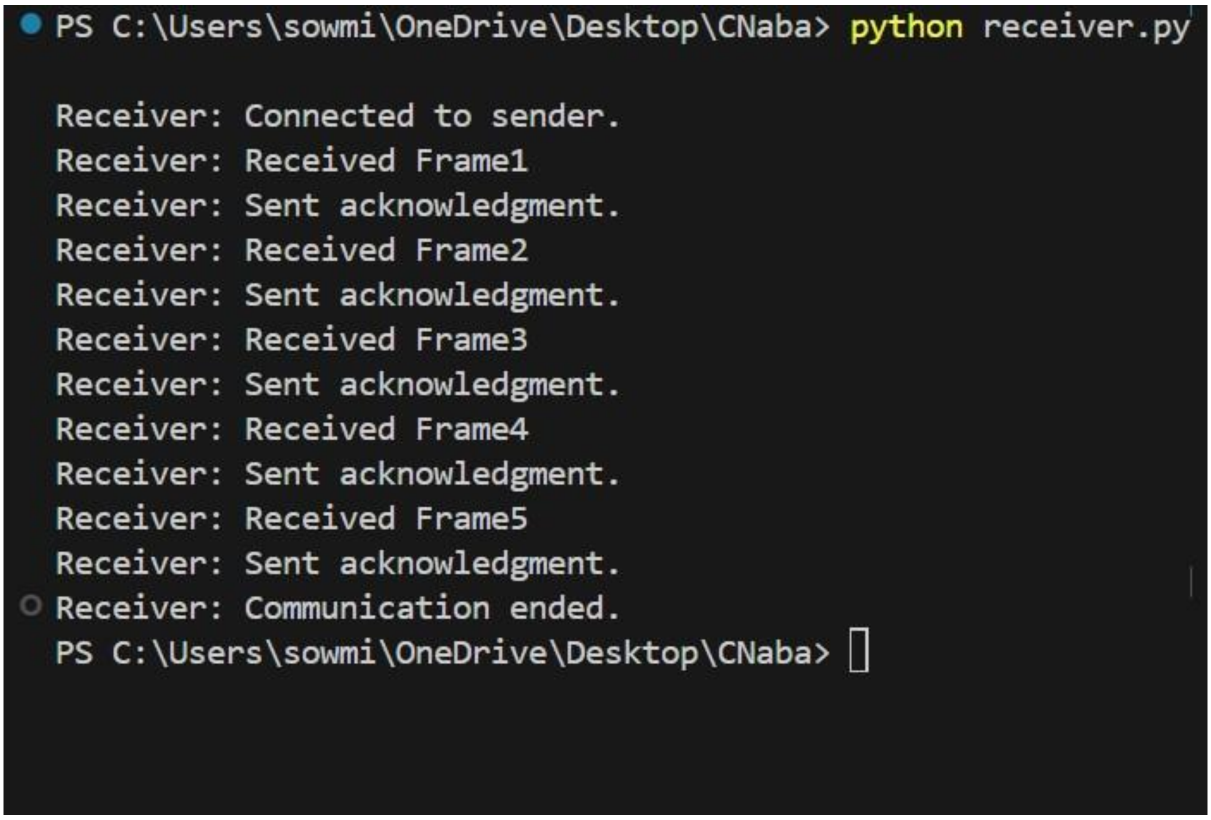
Firstly, we need to run the **sender.py** file, followed by **receiver.py** file, then provide the frame numbers to send to the receiver end and finally each frame will be sent to the receiver end and acknowledgement will be sent to the sender from the receiver. Once the sender receives the acknowledgement only then the next frame will be sent to the receiver or else Re-transmission of the same frame need to be done.

Initially Compile and run the **sender.py** file as shown:



```
PS C:\Users\sowmi\OneDrive\Desktop\CNaba> python sender.py
Sender: Waiting for receiver to connect...
Sender: Connected to receiver at ('127.0.0.1', 51095)
Enter the number of frames to send: 5
Sender: Sent Frame1
Sender: Acknowledgment received.
Sender: Sent Frame2
Sender: Acknowledgment received.
Sender: Sent Frame3
Sender: Acknowledgment received.
Sender: Sent Frame4
Sender: Acknowledgment received.
Sender: Sent Frame5
Sender: Acknowledgment received.
Sender: Transmission completed.
Sender: Times recorded and saved to 'times.txt'.
PS C:\Users\sowmi\OneDrive\Desktop\CNaba> python plot_perfor
mance.py
```

Then, Compile and run the **receiver.py** file as shown:



```
PS C:\Users\sowmi\OneDrive\Desktop\CNaba> python receiver.py

Receiver: Connected to sender.
Receiver: Received Frame1
Receiver: Sent acknowledgment.
Receiver: Received Frame2
Receiver: Sent acknowledgment.
Receiver: Received Frame3
Receiver: Sent acknowledgment.
Receiver: Received Frame4
Receiver: Sent acknowledgment.
Receiver: Received Frame5
Receiver: Sent acknowledgment.
Receiver: Communication ended.
PS C:\Users\sowmi\OneDrive\Desktop\CNaba> 
```

5. Time-Complexity Analysis with the Graph.

The time complexity of the Stop-and-Wait protocol is influenced by the number of frames, network latency, and retransmissions. For the provided graph, we analyse the total time complexity as a function of these factors.

1.Ideal Case (No Retransmissions): Time to send single frame and receive the ACK.

$$T_{\text{ideal}} = n \cdot (T_{\text{transmission}} + T_{\text{ack}})$$

The time complexity in this case is **O(n)**, where n is the number of frames

2.Worst Case (With Retransmissions): “r” denotes number of retransmissions.

$$T_{\text{worst}} = \sum_{i=1}^{n} ((T_{\text{transmission}} + T_{\text{ack}}) \cdot (1 + r_i))$$

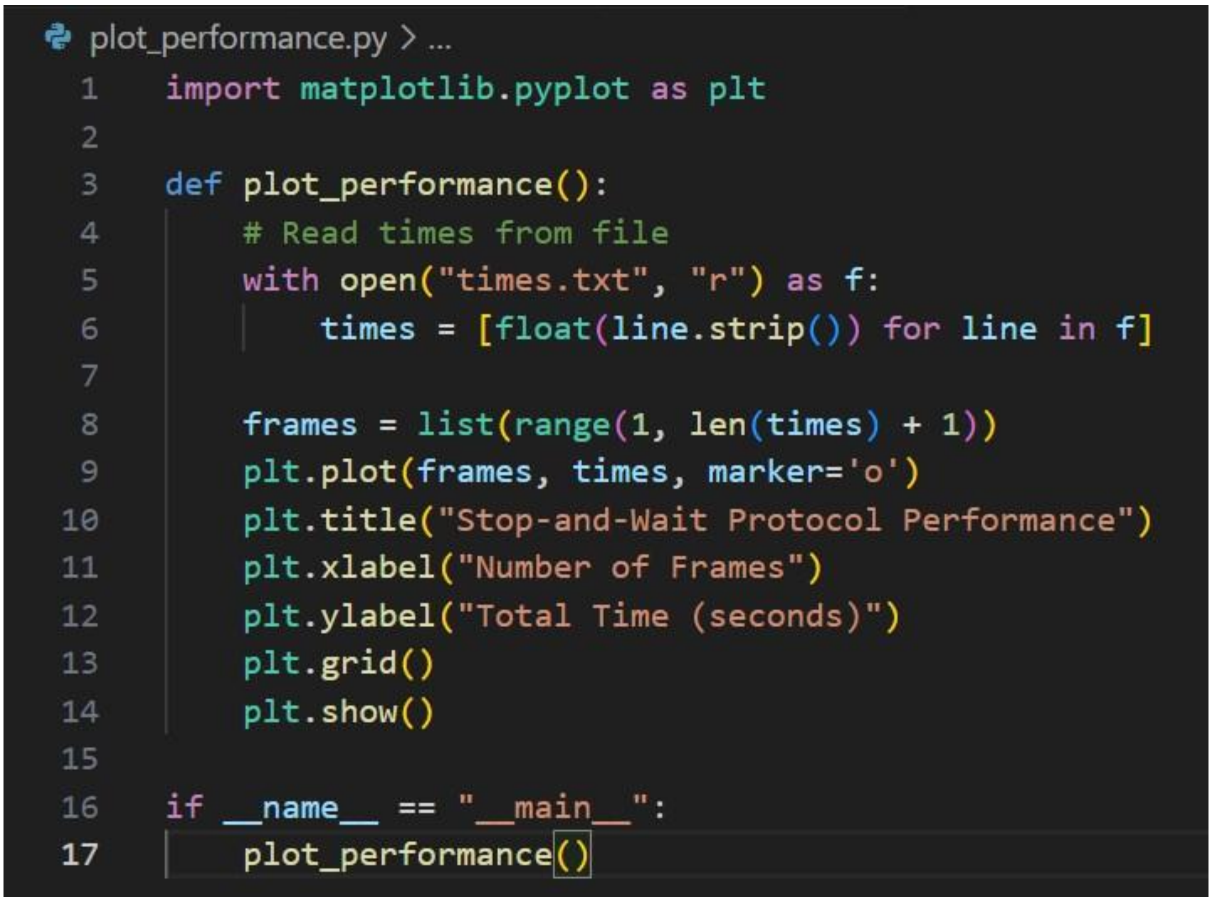
The time complexity remains O(n) since retransmissions are bounded and depend on the error rate, but the constant factor increases significantly with r.

To Plot the Graph, we have added one more file named as **plot_performance.py** file.

Key Components

- Reads times.txt generated by the sender.
- Plots the total time for each frame on a graph.

Contents of **plot_performence.py:**



plot_performance.py > ...

```python
import matplotlib.pyplot as plt

def plot_performance():
    # Read times from file
    with open("times.txt", "r") as f:
        times = [float(line.strip()) for line in f]

    frames = list(range(1, len(times) + 1))
    plt.plot(frames, times, marker='o')
    plt.title("Stop-and-Wait Protocol Performance")
    plt.xlabel("Number of Frames")
    plt.ylabel("Total Time (seconds)")
    plt.grid()
    plt.show()

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

The file will provide the graph that visually represents the performance of the Stop-and-Wait protocol based on the number of frames transmitted versus the total time taken for transmission, including retransmissions.

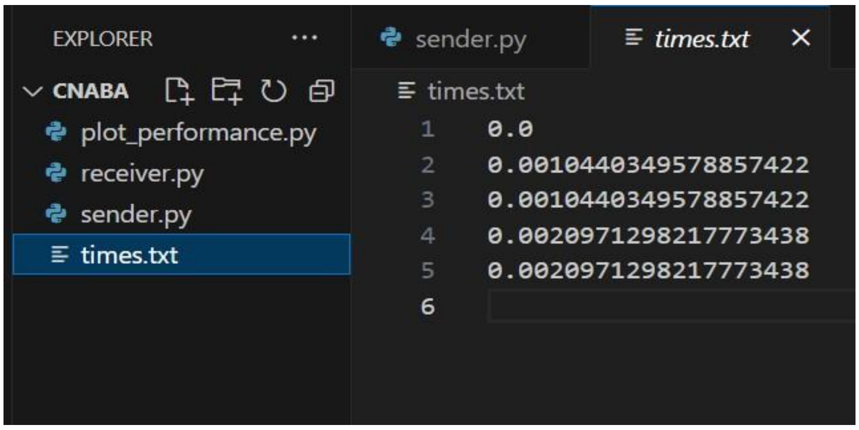
Key Observations from the Graph

X-Axis (Number of Frames): This represents the sequential frame numbers (1 to n). Each frame is sent one at a time following the Stop-and-Wait protocol.

Y-Axis (Total Time in Seconds): This indicates the cumulative time required to successfully transmit each frame. The time includes retransmissions if the acknowledgment (ACK) was not received.

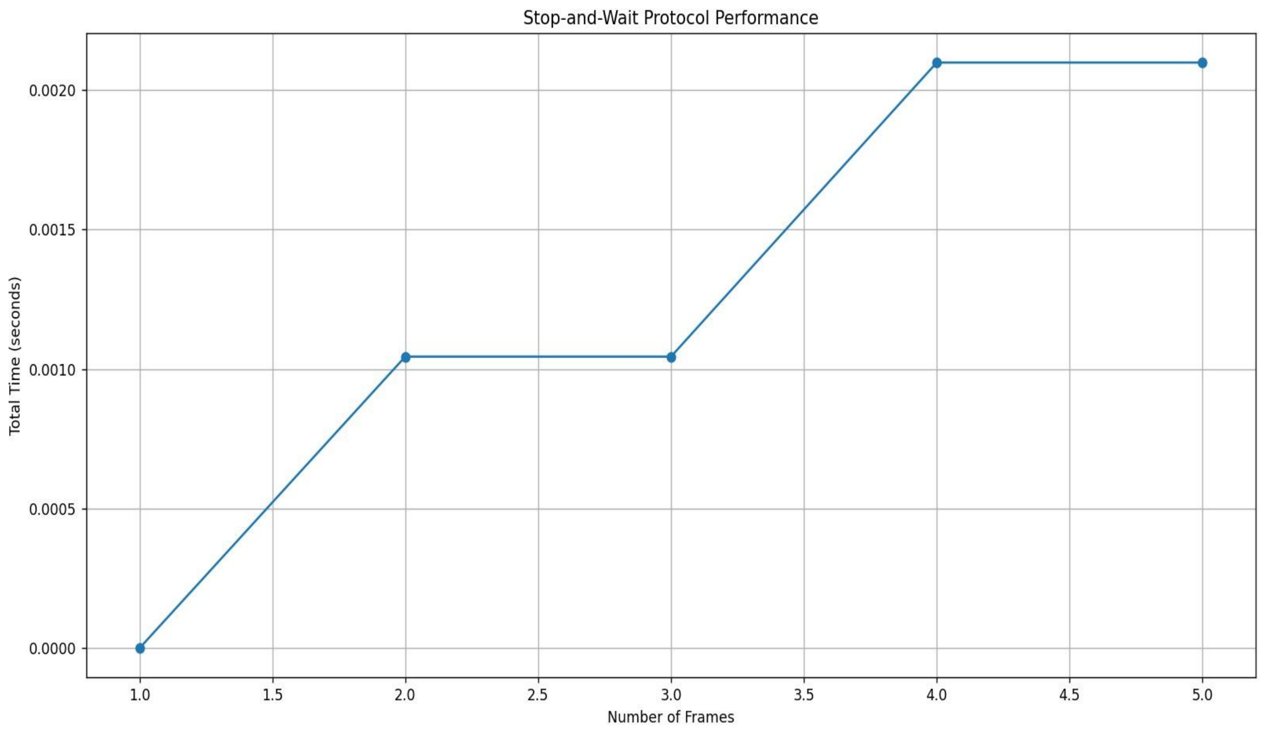
times.txt: Created by the sender to store total transmission times for each frame. **graph**: Displays the performance of the protocol based on the data from times.txt.

Contents of **times.txt** file generated after sending frames from sender to receiver:



```
0.0
0.0010440349578857422
0.0010440349578857422
0.0020971298217773438
0.0020971298217773438
```

Graph obtained based on the above **times.txt** file generated by sending frames in sequential order to receiver:



Graph-Specific Analysis

- There are 5 frames transmitted.

- Frame 3 incurs a retransmission due to acknowledgment failure, resulting in additional delay.

Observed Complexity:

- Frames 1, 2, 4, and 5 follow the ideal case-Transmission time grows linearly.
- Frame 3 experiences a retransmission-The spike reflects the additional time taken, contributing an extra transmission and receiving ACK

Thus, the total time for 5 frames is a mix of the ideal and retransmission cases. While the graph reflects **O(n)** complexity overall, the retransmission introduces a localized increase in time for specific frames.

6. Applications of Stop and Wait Protocol.

- Low-Speed Networks: Used in environments with low bandwidth or error-prone channels (e.g., satellite communication, low-frequency radio systems) where reliability is more critical than speed.
- Simple Communication Systems: Employed in basic communication systems or embedded systems where the simplicity of implementation is a priority over performance.
- Error Handling: Used in situations where ensuring the integrity of each individual frame is crucial, as it can detect and handle errors using retransmissions.
- Wireless Communication: Often found in low-power or short-range wireless communication protocols (e.g., Zigbee, Bluetooth low energy) where devices need to conserve energy and handle occasional transmission errors.
- The Stop-and-Wait protocol is commonly used in **satellite communication**, where ensuring reliable data transmission over long distances is critical due to high latency and error-prone channels caused by atmospheric interference

7. Conclusion

The Stop-and-Wait protocol is a straightforward and reliable method for ensuring accurate data transmission, utilizing acknowledgment and retransmission mechanisms to handle errors effectively. Its simplicity makes it well-suited for low-speed or unreliable networks, where maintaining data integrity is crucial. However, its design, which allows only one frame to be sent at a time, results in significant inefficiencies in high-speed or long-latency networks, leading to poor utilization of available bandwidth.

Furthermore, retransmissions due to acknowledgment timeouts or errors can introduce delays, further reducing the protocol's practicality in modern systems. Despite its drawbacks, the Stop-and-Wait protocol plays an important role as a foundational concept in networking, laying the groundwork for more advanced protocols. Its core principles are extended in protocols like Go-Back-N and Selective Repeat, which improve efficiency and performance while preserving reliability.