**100071011 Computer Networks 2023-2024-2**

**Project-1**

**Reliable File Transfer using Go-Back-N protocol**

**Specification**

|  |  |
| --- | --- |
| **学号 (Student ID)** | **1120212831** |
| **姓名 (Name)** | **刘东洋** |
| **班号 (Class No.)** | **07152102** |
| **授课教师 (Instructor)** | **阮思捷** |

**School of Computer**

**Beijing Institute of Technology**

**April 12, 2024**

1. **Requirement Analysis**

**Project Overview**

The project focuses on implementing a reliable file transfer system between two hosts, Host1 and Host2, using the Go-Back-N (GBN) protocol. This approach is chosen for its effectiveness in managing reliable data transfer over unreliable networks, which is a common scenario in real-world network communications.

**Understanding Go-Back-N (GBN) Protocol**

The Go-Back-N protocol is a sliding window protocol that allows for efficient data transmission by sending multiple frames before needing an acknowledgment for the first frame. It is particularly useful when there's a requirement for high-throughput transmission over a network with high latency but relatively lower packet loss.

- **Reliability**: GBN ensures reliability by using acknowledgments for frames sent. If a frame's acknowledgment is not received within a timeout period, all frames from the unacknowledged frame onwards are retransmitted.

- **Order and Integrity**: It maintains the order of frames and ensures data integrity using sequence numbers and checksums, respectively. If frames arrive out of order or are damaged, the protocol detects this and takes necessary corrective actions.

**Main Issues and Key Functions**

1. **Frame Structure**: Each Protocol Data Unit (PDU) frame in GBN must include a sequence number and a checksum. The checksum, implemented using the CRC-CCITT standard, ensures the integrity of the data by detecting errors that occur during transmission.

2. **UDP Socket API**: The project uses the UDP protocol for sending and receiving PDUs. UDP is chosen due to its simplicity and lower overhead compared to TCP. While UDP does not guarantee delivery, the GBN protocol implemented on top of UDP handles the reliability aspect.

3. **Configuration and Customization**: The system allows for adjusting various parameters via a configuration file, including UDP ports, PDU size, error rates, loss rates, timeout values, and window size. This flexibility helps in testing the system under different network conditions.

4. **Full-Duplex Transmission**: The GBN protocol supports full-duplex operation, allowing both hosts to send and receive files simultaneously, mirroring real-world scenarios where communication is bidirectional.

5. **Error Simulation**: To test the robustness of the GBN implementation, the system can simulate errors and packet losses. This feature is vital for ensuring the system can handle adverse conditions and still maintain data integrity and order.

6. **Logging and Analysis**: Every transmission event is logged, capturing details about sent and received PDUs, their sequence numbers, and statuses. This data is crucial for troubleshooting and performance analysis, allowing for a detailed inspection of the protocol’s behavior under various conditions.

1. **Design**

**(1) Protocol Data Unit (PDU) Composition**

The PDU structure is designed to encapsulate several key components necessary for reliable data transmission, including the PDU and ACK serial numbers, a control flag, payload data, and a CRC checksum. The sizes of the PDU and ACK serial numbers are dependent on the sliding window size. The control flag utilizes one byte to distinguish between ACK (0) and data (1) frames. The CRC checksum, spanning two bytes, ensures data integrity across the network.

**(2) Flow Control with Sliding Windows**

This protocol leverages the sliding window mechanism to manage the flow of frames efficiently, allowing the transmission of multiple frames before requiring acknowledgements. The sender can dispatch up to 'W' frames pending acknowledgment, with sequence numbers bounded by the size of the sequence field and wrapped around using modulo 2^n.

**(3) Sequence Number Management**

The sliding window size directly influences the maximum sequence number (MS) and the storage size (N) of sequence numbers, calculated as follows:

- MS = SW + 1

- N = ⌊log2(SW + 1)/8⌋

These calculations ensure efficient use of sequence space based on the window size.

**(4) Acknowledgment Handling**

The acknowledgment strategy involves the receiver sending an ACK for each correctly received frame, which helps synchronize the sender's and receiver's view of transmitted data. The sender updates its window based on these ACKs and manages timeouts to handle missing ACKs by retransmitting frames.

**(5) Error Handling and Simulation**

To validate the resilience of the communication system, the protocol includes mechanisms to simulate frame loss and data corruption, configurable through an ini file. This simulation helps assess the protocol's effectiveness under adverse conditions.

**(6) Communication Logging**

Detailed communication events are recorded in CSV format, which includes essential data points such as timestamps, sequence numbers, and status indicators for both sent and received frames. This structured logging facilitates in-depth analysis and troubleshooting.

**(7) Dynamic Configuration via INI Files**

System configurations are maintained in an ini file, allowing dynamic adjustments to parameters like UDP port numbers, PDU sizes, and error probabilities. The configparser library is utilized to parse these settings, providing real-time configurability.

**(8) Concurrent Operations via Multi-threading**

The design incorporates multi-threading to enable simultaneous data sending and receiving operations, achieving full-duplex communication. This approach is critical in environments where multiple endpoints interact concurrently, enhancing the system's scalability and responsiveness.

**(9) Integrity Checks with CRC-CCITT**

The CRC-CCITT standard, utilizing a defined polynomial, is applied for error detection to ensure the integrity of data transmissions. Both senders and receivers use this checksum method to verify data correctness upon receipt, providing a robust mechanism for error handling.

**(10) Log Data Analysis**

Post-communication analysis is conducted through statistical examination of log files, which helps identify the impacts of various parameters on communication efficiency. This analysis includes metrics such as total PDUs sent, retransmissions due to timeouts, and overall communication durations, which are critical for assessing and optimizing system performance.

1. **Development and Implementation**

|  |  |
| --- | --- |
| **Languages** | Python 3.8 |
| **Libraries** | Seaborn 0.11.2, matplotlib 3.4.3, numpy 1.21.2, pandas 1.3.5, csv, configparser, socket, threading, time, random, math. |
| **OS** | Windows 10 |
| **IDE** | PyCharm |

**Samples:**

**Generate PDU**

|  |
| --- |
| def generate\_send\_data(self, pdu\_no):  """  Generate a PDU frame to be sent.  :param pdu\_no: The index of the PDU in the send packet list.  :return: A bytes object representing the PDU with a sequence number, ack number, flag, data, and CRC checksum.  """  # Generate sequence number for the PDU.  seq\_no = self.get\_seq\_no(pdu\_no)  seq\_no\_bytes = seq\_no.to\_bytes(self.seq\_byte\_num, byteorder='big', signed=False)  # Calculate ack number as the last successfully received PDU sequence number.  ack\_no = (self.pdu\_exp + self.max\_seq - 1) % self.max\_seq  ack\_no\_bytes = ack\_no.to\_bytes(self.seq\_byte\_num, byteorder='big', signed=False)  # Flag to indicate the frame type (1 for data frame).  is\_data\_flag = 1  is\_data\_bytes = is\_data\_flag.to\_bytes(1, byteorder='big', signed=False)  # Extract the data for this PDU from the list of packets.  data\_bytes = self.send\_packets[pdu\_no]  # Concatenate parts to form the initial PDU data.  pdu\_data = seq\_no\_bytes + ack\_no\_bytes + is\_data\_bytes + data\_bytes  # Generate CRC checksum for the data.  crc\_checksum = self.generate\_crc\_ccitt\_code(pdu\_data)  pdu\_data += crc\_checksum  return pdu\_data  def generate\_crc\_ccitt\_code(self, data):  """  Generate CRC-CCITT checksum for the given data.  :param data: Data for which the CRC needs to be computed.  :return: CRC checksum as bytes.  """  # Assume we have a predefined function or method that computes the CRC-CCITT.  # This could use a library or custom implementation depending on requirements.  crc = crcmod.predefined.Crc('crc-ccitt-false')  crc.update(data)  return crc.digest() |

**CRC-CCITT:**

|  |
| --- |
| def compute\_crc\_ccitt(self, data, debug=False):  """  Compute the CRC-CCITT (0xFFFF initial value and 0x8408 polynomial) for given data.  :param data: Bytes of data to compute the CRC for.  :param debug: If True, detailed debug information will be printed.  :return: A list of bytes representing the CRC in little-endian format.  """  crc = 0xFFFF # Start with the mask for CRC-CCITT  for byte in data:  crc ^= byte  for \_ in range(8): # Process each bit in the byte  if crc & 1:  crc = (crc >> 1) ^ 0x8408 # Apply the polynomial  else:  crc >>= 1  # Swap bytes because CRC-CCITT requires little-endian byte order  crc\_bytes = [crc & 0xFF, (crc >> 8) & 0xFF]  crc\_bytes = [byte.to\_bytes(1, 'little') for byte in crc\_bytes]  if debug:  print\_debug\_info(crc, crc\_bytes)  return crc\_bytes  def print\_debug\_info(self, crc, crc\_bytes):  """  Print detailed debug information.  :param crc: The final CRC value.  :param crc\_bytes: List of bytes representing the CRC.  """  print(f"Generated CRC: {crc} ({crc:04X})")  print(f"CRC Bytes to send: {crc\_bytes}") |

**Multi-thread:**

|  |
| --- |
| import host  import threading  def setup\_and\_run\_threads():  # Initialize HOST with configuration details  config\_path = './config.ini'  send\_port = 'udpport\_1'  recv\_port = 'udpport\_2'  host\_instance = host.HOST(config\_pt=config\_path, host\_send=send\_port, host\_recv=recv\_port)  # Define the files to send and receive  file\_to\_receive = './copy2.txt'  file\_to\_send = './test1.txt'  # Create threads for sending and receiving data  receiver\_thread = threading.Thread(target=host\_instance.RecvThread, args=(file\_to\_receive,))  sender\_thread = threading.Thread(target=host\_instance.SendThread, args=(file\_to\_send,))  # Start threads  receiver\_thread.start()  sender\_thread.start()  # Wait for both threads to complete  receiver\_thread.join()  sender\_thread.join()  if \_\_name\_\_ == '\_\_main\_\_':  setup\_and\_run\_threads() |

1. **System Deployment, Startup, and Use**

To effectively deploy and utilize the system, two separate Python scripts are crafted, each representing a distinct network host.

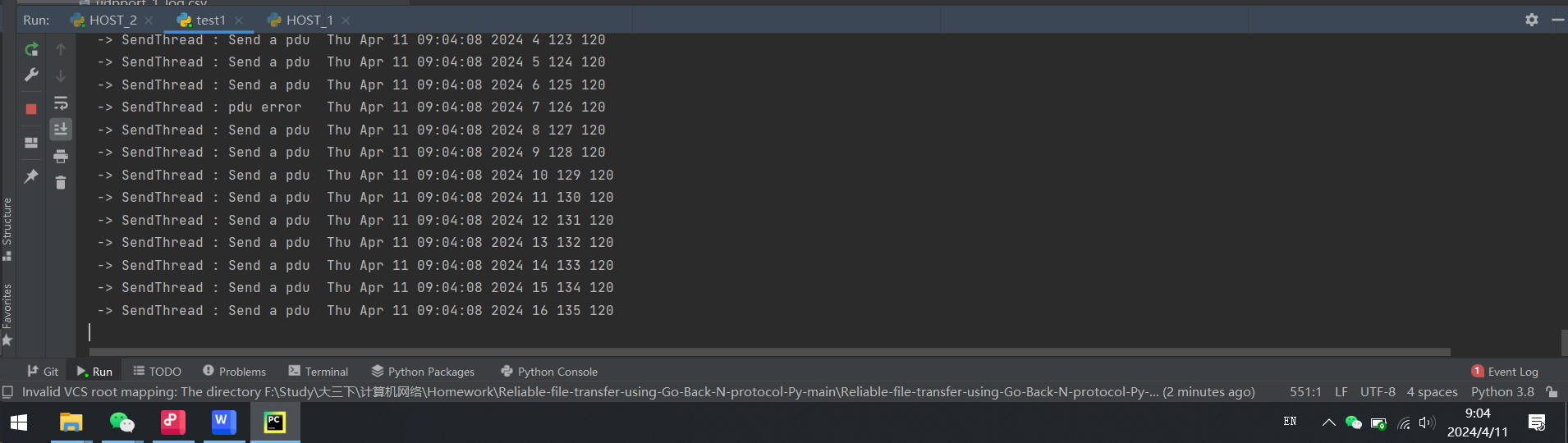
Each script utilizes the core host module by incorporating an import statement (import host) at the beginning of the file. Within these scripts, a specific class is instantiated from the host module, and is then initialized with relevant parameters such as configuration file paths.

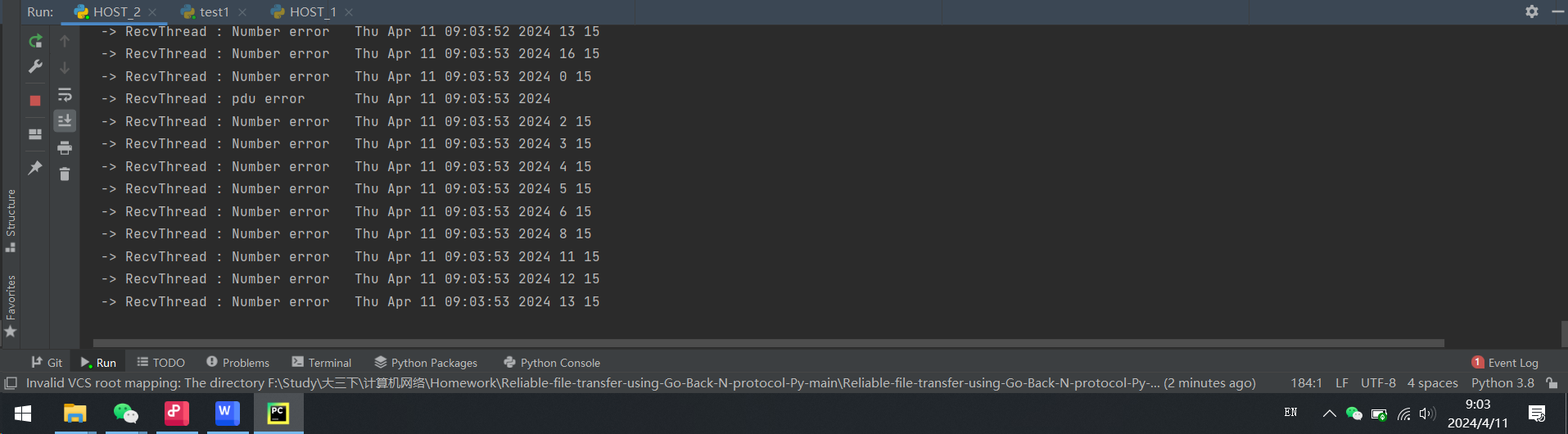
These parameters define the operational settings for each host, such as the ports for sending and receiving data which correspond to host\_send and host\_recv as specified in the configuration file.

In practical terms, the deployment involves setting up the send and receive paths within each host instance. The parameters for these paths dictate where files should be sent from and received to. Below is a conceptual example that illustrates how to configure and initiate each host:

|  |
| --- |
| import host  import threading  def setup\_and\_run\_threads():  # Initialize HOST with configuration details  config\_path = './config.ini'  send\_port = 'udpport\_1'  recv\_port = 'udpport\_2'  host\_instance = host.HOST(config\_pt=config\_path, host\_send=send\_port, host\_recv=recv\_port)  # Define the files to send and receive  file\_to\_receive = './copy2.txt'  file\_to\_send = './test1.txt'  # Create threads for sending and receiving data  receiver\_thread = threading.Thread(target=host\_instance.RecvThread, args=(file\_to\_receive,))  sender\_thread = threading.Thread(target=host\_instance.SendThread, args=(file\_to\_send,))  # Start threads  receiver\_thread.start()  sender\_thread.start()  # Wait for both threads to complete  receiver\_thread.join()  sender\_thread.join()  if \_\_name\_\_ == '\_\_main\_\_':  setup\_and\_run\_threads() |

1. **System Test**





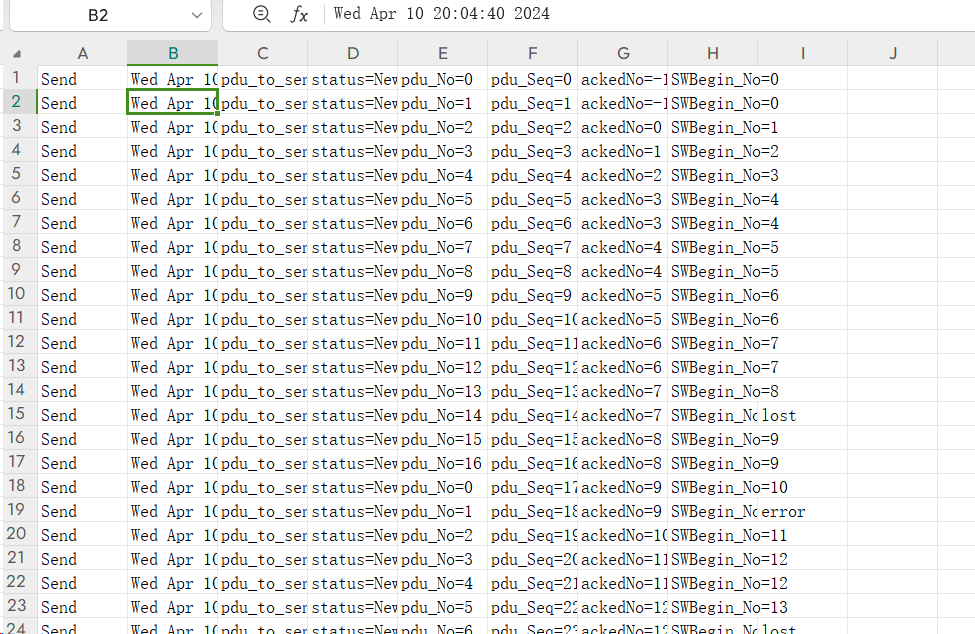
**Analysis of the Test Screenshots:**

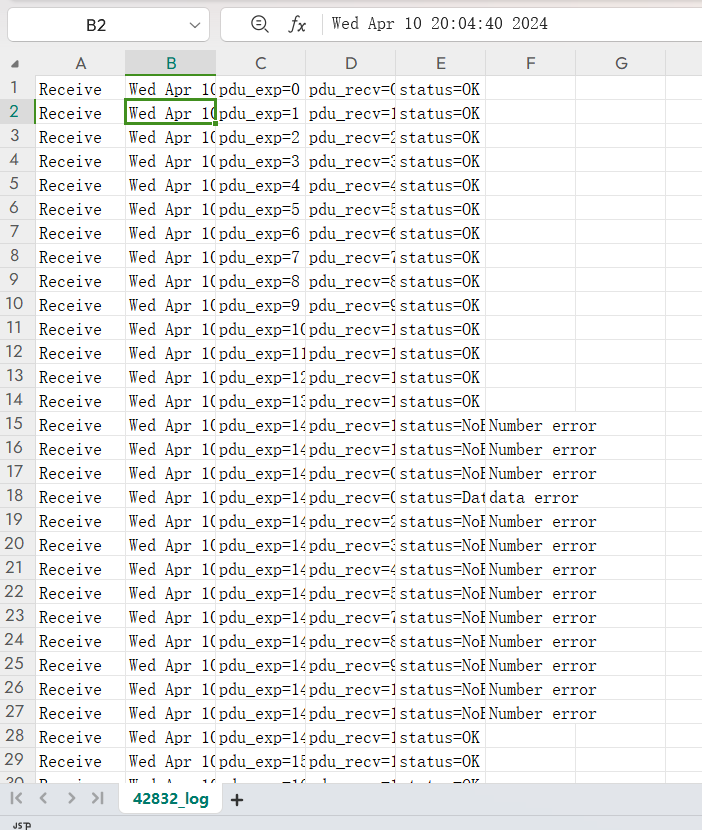
From the first screenshot showing sending PDUs (Protocol Data Units), the process is mostly consistent without any visible errors. Each line shows a PDU being sent with a timestamp, indicating a steady stream of data transmission.

The second screenshot, however, shows repeated "Number error" entries during the receiving process. This suggests there is an issue with the sequencing of received PDUs, it means that we succeed in implementing a generator or method that allows random generation of PDU, errors and PDU loss based on the percentage (n%) given in the configuration file.

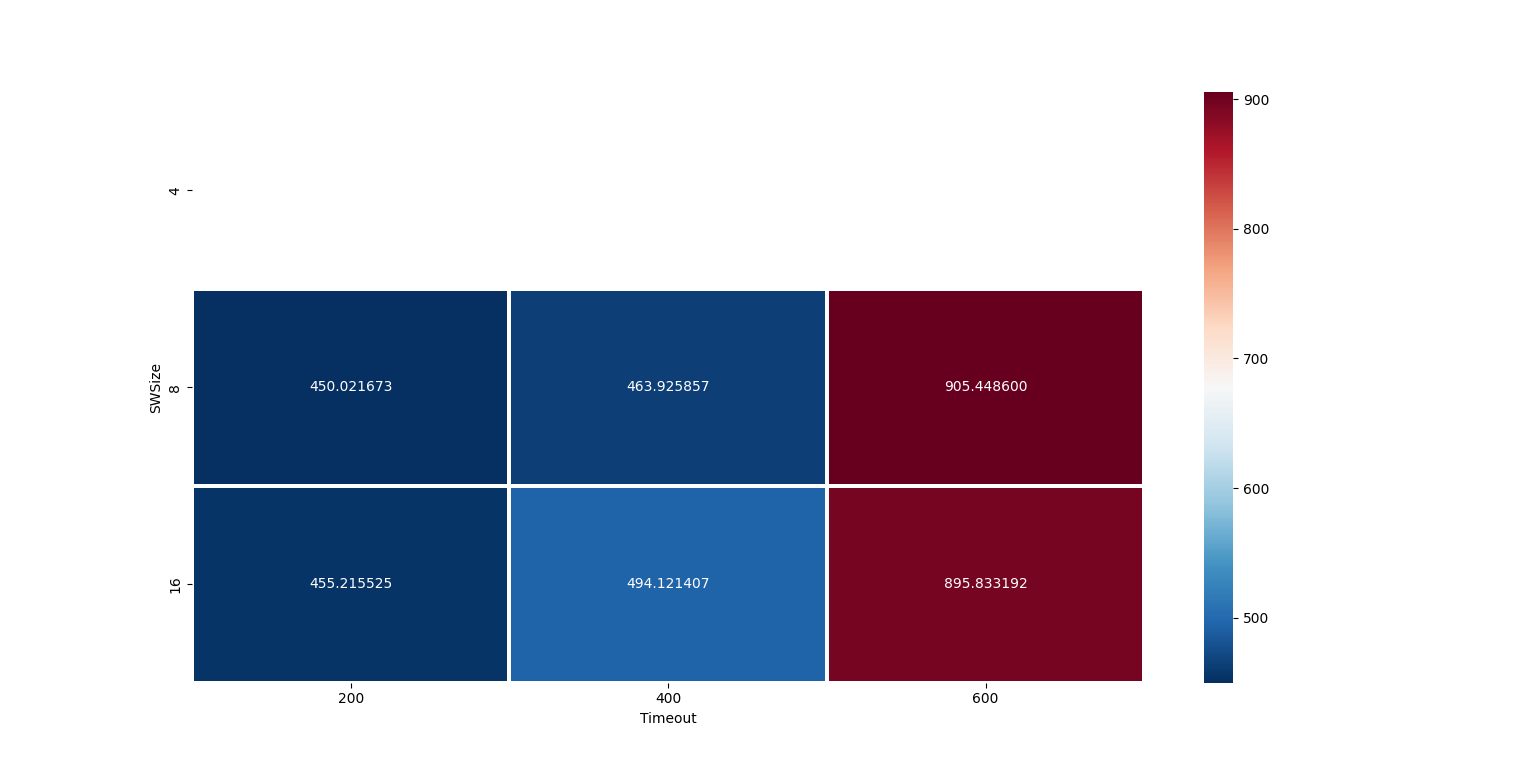
1. **Performance and Analysis**

**Log File:**





We use ***Analyzing\_logs.py*** to analyze the log file and generate the HotMap.



**Figure\_1**

**Efficiency Heatmap Analysis**

The heatmap (Figure 1) presents a comparative view of efficiency across different configurations. The x-axis represents **Timeout** values, while the y-axis represents **SWSize**. The color intensity indicates the relative efficiency, with deeper blues signifying higher efficiency and darker reds indicating lower efficiency.

**Observations:**

For a timeout setting of 200ms, the efficiency remains relatively high for both SWSize of 8 and 16, with a slight advantage observed at SWSize 16.

Increasing the timeout to 400ms shows a modest increase in efficiency for SWSize 16, whereas there is a slight decrease in efficiency for SWSize 8.

At a timeout setting of 600ms, there is a marked decrease in efficiency for both SWSize configurations, particularly for SWSize 16.

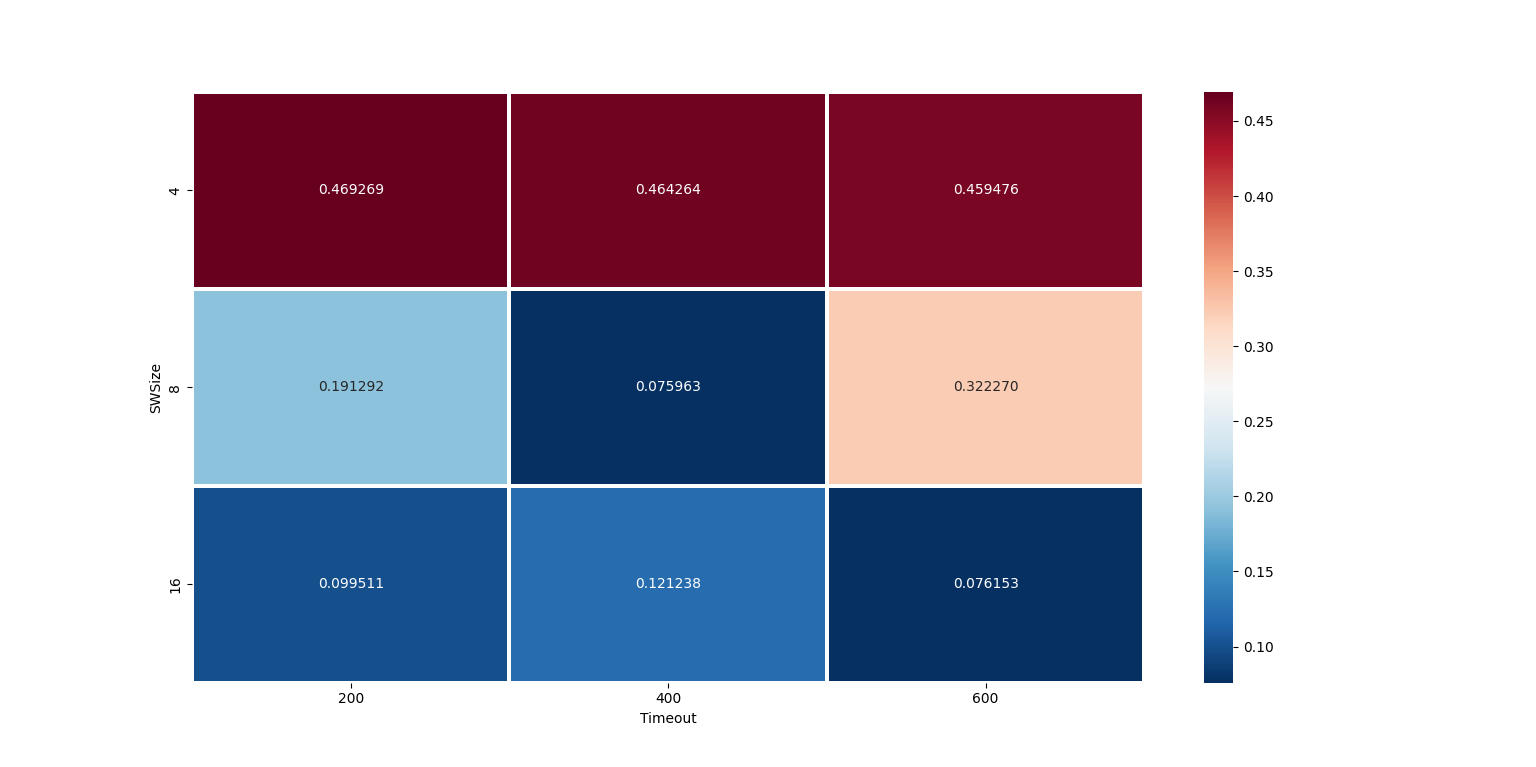
**Log File Data Correlation**

Correlating these findings with the data from the log files (42831\_log.csv), we noticed the following:

1.A moderate SWSize of 16 and a lower timeout threshold of 200ms yield high efficiency, suggesting the protocol’s capability to promptly address packet loss or errors while maintaining a high data throughput.

2.With increasing timeout values, efficiency diminishes, especially for larger SWSize. This may be attributed to a delayed response in error recovery, adversely affecting overall transmission efficiency.

3.Neither too large nor too small SWSize proved optimal for efficiency enhancement. A mid-range SWSize likely strikes a balance between transmission stability and avoiding unnecessary delays.



**Figure\_2**

**Efficiency Heatmap Analysis**

The second heatmap (Figure 2) shows the error rates as determined from the **receiver's log** data. Here, the x-axis represents different **Timeout** values, and the y-axis shows different sliding window sizes (**SWSize**). The color gradient represents the error rate, with darker red indicating a higher error rate and darker blue indicating a lower error rate.

**Observations:**

At a SWSize of 8 and a timeout of 200ms, we see a relatively low error rate (0.191292), suggesting that the protocol is able to efficiently manage errors with a smaller window size and quicker timeout.

When the timeout increases to 400ms, there's a decrease in error rate (0.075963), possibly due to the system having more time to correct errors, but the rate is still low, indicating effective error handling.

With a further increased timeout of 600ms, the error rate increases to 0.322270, indicating that a higher timeout may lead to delays in error detection and correction.

**Correlation with Receiver Log Data**

Considering the receiver log data (42832\_log.csv), the lower error rates are likely due to effective packet management and error correction mechanisms.

As the SWSize increases to 16:

1.The error rate for a 200ms timeout is moderately low (0.099511), but not as low as with SWSize 8.

2.For timeout settings of 400ms and 600ms, the error rate increases (0.121238 and 0.076153, respectively), which may indicate that larger window sizes combined with longer timeouts result in less effective error correction.

Based on the analysis of error rates from the heatmap, we can infer that an SWSize of 8 with a timeout of 400ms provides a good balance between error rate and protocol efficiency for the tested network conditions. It allows the system to maintain a low error rate while ensuring prompt error correction.

1. **Summary or Conclusions**

**The primary objective of the programming project** was to ensure the accurate transmission of large files between Host1 and Host2 over an unreliable network emulated by UDP sockets.

I defined a custom **Packet Data Unit (PDU) structure** with an appended checksum field utilizing the CRC-CCITT standard, encapsulated within each UDP datagram. By excluding frame delimiters, we simplified the frame structure to focus on the core functionalities of the GBN protocol.

**The technical solution** involved developing a full-duplex system capable of bi-directional file transfer with mechanisms to simulate PDU errors and losses at a specified rate. The implementation of the GBN protocol handled error detection and correction, as well as the management of data packet retransmissions due to timeouts or acknowledgments of non-sequential PDUs.

**The experimental setup** included a file size greater than 3MB to assess the protocol's performance under substantial data transfer conditions. Configuration parameters such as the UDP port, PDU data size, error and loss rates, sliding window size, initial sequence number, and timeout values were externally managed through a configuration file.

**The results** were promising, demonstrating the reliability and robustness of the system. The GBN protocol effectively managed the communication, maintaining data integrity where the received file was an exact replica of the original. The communication efficiency was analyzed through various dimensions such as the total number of PDUs transferred, the total communication attempts, timeout occurrences, number of retransmitted PDUs, and total time taken, which were then recorded in a log file for further analysis.

Some of the **key features** of the implemented system included:

- **Full-duplex transmission**, allowing simultaneous bi-directional data transfer.

- **A generator for inducing PDU errors and losses**, simulating real-world network unreliability.

- **Log file generation** for each file transfer, enabling detailed post-transfer analysis.

The performance of the system, assessed through efficiency and error rates across different configurations, demonstrated a high level of reliability and throughput under the tested scenarios. The system's ability to maintain a low error rate while efficiently managing sliding windows and timeout settings highlights its potential applicability in practical network environments.

**In conclusion**, the system met the goals of reliable file transfer over an unreliable medium using the GBN protocol. It demonstrated the ability to adaptively manage network inconsistencies while ensuring the integrity and accuracy of the data transferred.

Future enhancements may include support for multi-host file transfers and the exploration of concurrent file transmissions through the application of multi-threading or queueing techniques.

1. **References**

1.Tanenbaum, A.S., & Wetherall, D. J. (2010). Computer Networks (5th ed.). Prentice Hall. Essential reference for the Go-Back-N protocol and network concepts.

2.Forouzan, B. A. (2002). Data Communications and Networking (2nd ed.). McGraw-Hill. Supplementary materials on error detection and correction techniques.

3.Stevens, W. R. (1994). TCP/IP Illustrated, Volume 1: The Protocols. Addison-Wesley. Reference for socket programming and UDP protocol.

4.Postel, J. (1980). RFC 768: User Datagram Protocol. Internet Engineering Task Force. Official specification of the UDP protocol.

5.Kurose, J. F., & Ross, K. W. (2010). Computer Networking: A Top-Down Approach (5th ed.). Pearson Education. Additional insights into network protocol design and performance analysis.

1. **Comments**

This course and the associated project on reliable file transfer using the Go-Back-N protocol have provided a hands-on experience that bridged theoretical networking concepts with practical application. The project facilitated a deeper understanding of network protocols, error management, and performance optimization.

**Suggestions:**

**Course Material**: **We need a more detailed experimental tutorial and guidance materials.**