CS305-2024Spring-FinalProject: Reliable Data Transfer upon UDP

11813203 段青洋 12111644 马鑫 12211825 周益贤

**Abstract:**

In this project we focuses on developing a reliable data transfer (RDT) protocol using UDP as the unreliable protocol. Specifically, this project requires us to use UDP as the transport layer protocol and implement an RDT protocol at the application layer. The primary objective is to design and implement a protocol that can address the challenges and limitations of data transfers in an unreliable network environment. The project aims to create a robust and efficient communication protocol by leveraging UDP and exploring various mechanisms for reliable transmission. The protocol will be designed to mitigate issues such as packet loss, delay, data corruption, and congestion, which commonly impact network communication.

**Format and demo:**

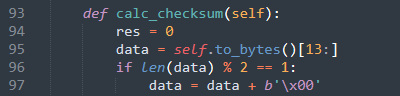
1. **Message Format**

The RDT header was construct as the template RDTHeader class, and it should contain the following data fields:

|  |  |  |
| --- | --- | --- |
| test\_case | 1 byte | Indicate the test case will be used |
| Source\_address | 6 bytes | IP address and port of sender |
| Target\_address | 6 bytes | IP address and port of receiver |
| SYN | 1 byte | SYN in TCP |
| FIN | 1 byte | FIN in TCP |
| ACK | 1 byte | ACK in TCP |
| SEQ\_num | 4 bytes | Sequence number in TCP |
| ACK\_num | 4 bytes | Acknowledge number in TCP |
| LEN | 4 bytes | Length of PYALOAD |
| RWND | 4 bytes | Size of Receiving window |
| CHECKSUM | 2 bytes | Chcksum in TCP |
| Reserved | 8 bytes | Reserved space for adding any additional fields you deem necessary |
| PAYLOAD | LEN bytes | Data |

One of the most important function of header.py is CHECKSUM Calculation:

1. **calc\_checksum()**



This method calculates a checksum for the data, which is typically used for error detection. It initializes a variable ‘res’ to store the intermediate checksum result. Converts the object to its byte representation and skips the first 13 bytes. This implies that the first 13 bytes are not part of the data that needs to be check summed, because they are a header or some fixed metadata.



Then pads the data with an extra zero byte if the length of the data is odd. This ensures that the data length is even, which is required for 16-bit word processing. Iterates over the data in 2-byte chunks, converts each chunk to an integer using big-endian byte order, and adds it to ‘res’.



Handles the carry bits by adding the overflow beyond 16 bits back into the lower 16 bits. This step is repeated to ensure all carries are properly accounted for. Applies a bitwise NOT to the result and masks it to 16 bits, returning the final checksum value

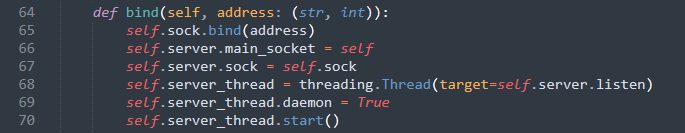
1. **set\_checksum()**



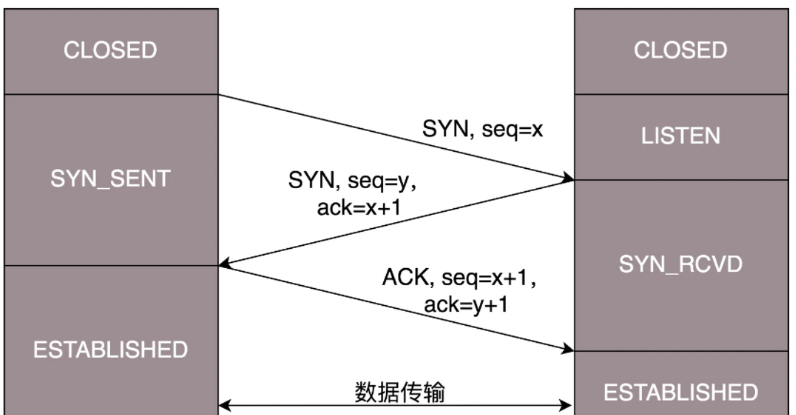
This method sets the checksum field of the object based on the calculated checksum.

1. **Reliable Data Transfer**
2. **Accept and establish a connection**

we need to implement the functions bind(), accept(), and connect().



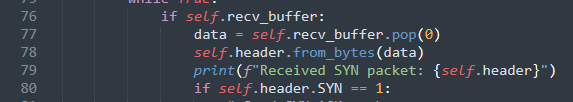
The function bind() could bind the current RDT socket to a specific address.



Functions accept() and connect() are used to establish a connection between the client and server. it is supposed to implement the 3-way handshake in these two functions.



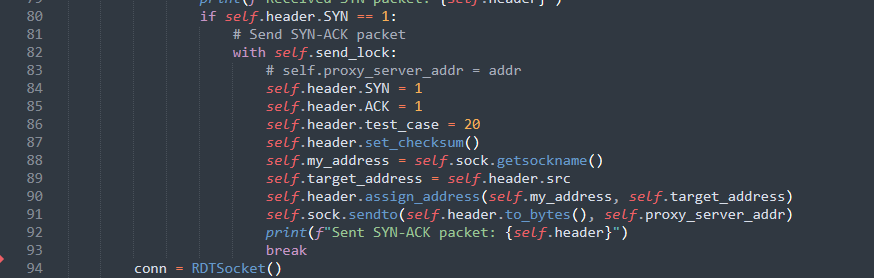
The method starts with an infinite loop (‘while True:’) that continuously checks for incoming data in the ‘recv\_buffer’.



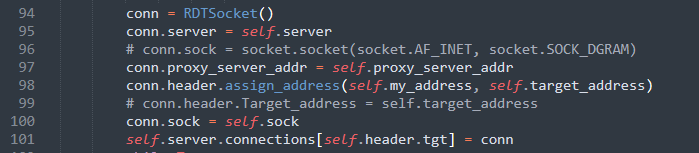
When data is available in the ‘recv\_buffer’, it is popped and processed as a SYN packet.

The ‘header’ object is populated with the data from the received packet using the ‘from\_bytes’ method.

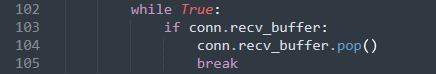
If the packet is a SYN packet (‘self.header.SYN == 1’), the method proceeds to send a SYN-ACK packet.



Sending SYN-ACK Packets.



Creating a New Connection



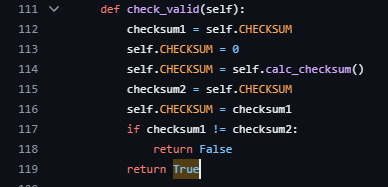
Waiting for ACK Packet



A message is printed to indicate that the connection has been accepted and the ACK packet has been received. The new connection object (`conn`) is returned.

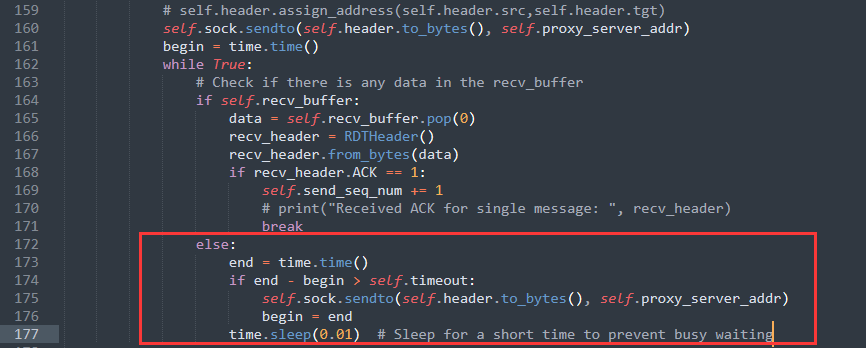
All in all, accept() is also supposed to demultiplex the data received from UDP to different sockets, implementing the Socket's multiplexing. In other words, when maintaining multiple sockets simultaneously, the buffer space for each socket should be isolated. Function accept() needs to support multithreading and be able to establish multiple socket connections. Messages from different sockets should be isolated from each other, requiring we to demultiplex the data received at the underlying UDP.

1. **Packet verification**



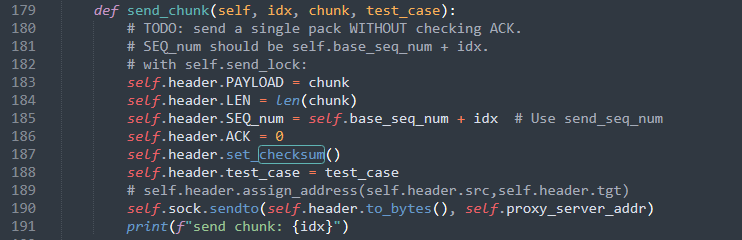
To ensure reliable data transmission, we are supposed to implement data validation functionality. Specifically, we need to calculate a 16-bit checksum for all fields in the packet except for test\_case, Source\_address, and Target\_address, and then fill in the CHECKSUM field. This ensures that the receiver can perform validation checks upon receiving the data. We defined a new function check\_valid() and invoke it when needed.

1. **Retransmitting**

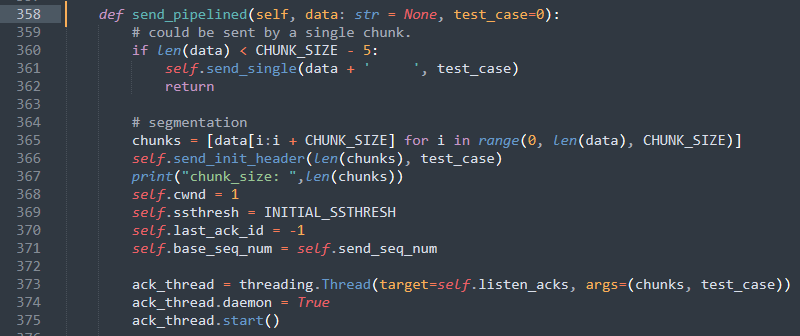


Generally, a packet may be lost, corrupted, or delayed in an unreliable network environment. Hence, it is necessary to retransmit these error packets when the above error happens. Specifically, when some packets that have been sent do not receive a response with ACK=1, which means that they have not been received successfully, it is necessary to resend those packets. We should implement this feature following the TCP timeout retransmit. It was realized in send\_single().

1. **Data segmentation**



We need to transmit large amounts of data that cannot be sent simultaneously. In such cases, we are supposed to implement a basic data chunking and sorting method to divide a large dataset into multiple small CHUNKs that can be transmitted directly. Additionally, it should ensure that the receiver can reassemble these data chunks into the original large dataset upon receipt.



And it was used in listen\_ack()

1. **Pipeline manner**

Call two threads when sending: send and listen-ack.



Send first sends a single packet indicating the number of slices, and then attempts to fill the sending queue with slices. If the sending queue is full, it stops and waits.

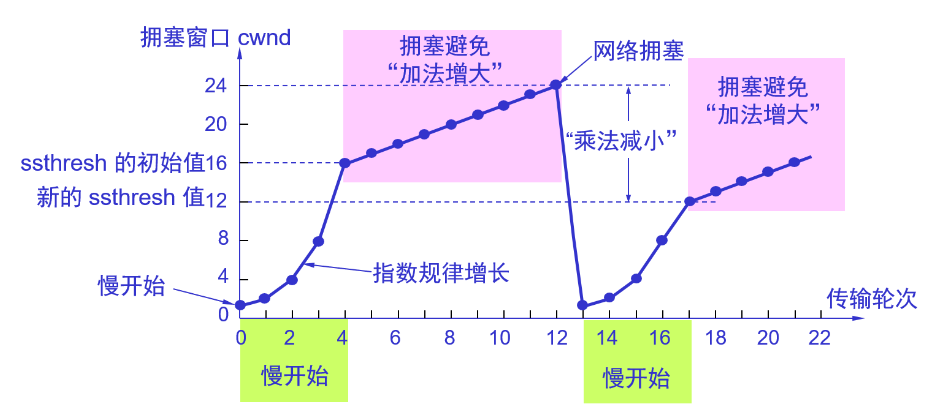


Listen.ack processes the returned ack packet and maintains the last packet that was previously acked, last-ack\_id. If the last-ack\_id is received three times, it is considered lost and the packet is resent.

At the same time, check the sending time in the queue that has been sent regularly. If the task has been ack, delete it. Otherwise, determine whether a timeout retransmission is required.

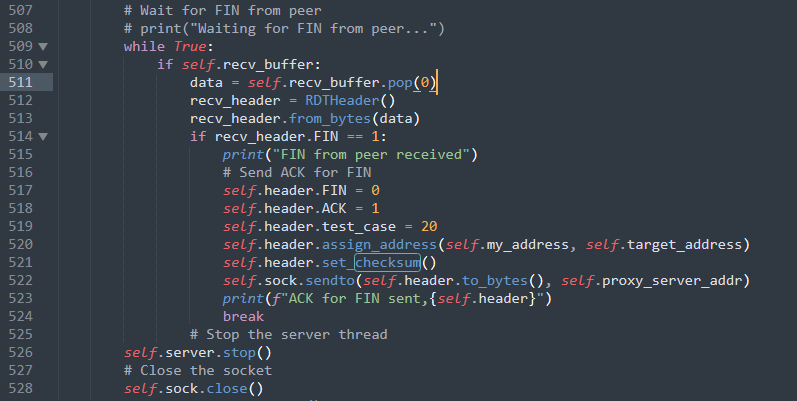
The receiving end first receives the number of slices, and then receives the slices and rearranges the order based on SEQ\_2um. Simultaneously maintaining cur indicates that all packets before this number have been received, and returning the corresponding ACKnum based on cur.

1. **Congestion control & Flow control**



In this part, we are supposed to implement the congestion and flow control mechanisms following the TCP Reno.

1. **Close connection**



It was supposed to implement the 4-way handshake to close a reliable connection in close() function. Once FIN from peer get received, socket will shut down.

Testing

We use proxy server and “Test\_case” to test our RDTSocket.

文本

描述已自动生成

Sender sends message to “fromSenderAddr”, and the target address in header is “receiver address”. Receiver sends to “fromReceiverAddr”, and target address in header is “sender\_address”, because the messages should go proxy server firstly.

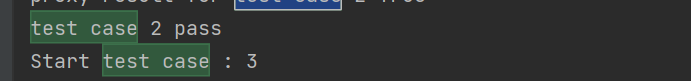
For test cases, we passed the first 7 cases.

图形用户界面, 文本, 应用程序, 聊天或短信

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成



图形用户界面, 文本

描述已自动生成

一些文字和图案

描述已自动生成

图形用户界面, 文本, 应用程序, 聊天或短信

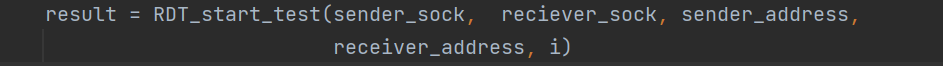
描述已自动生成

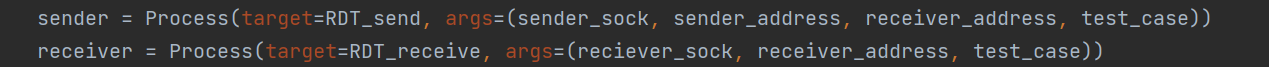
手机屏幕的截图

描述已自动生成

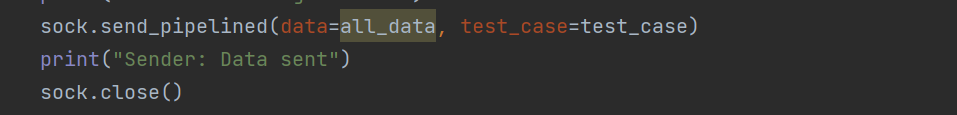
图形用户界面, 文本, 应用程序

描述已自动生成

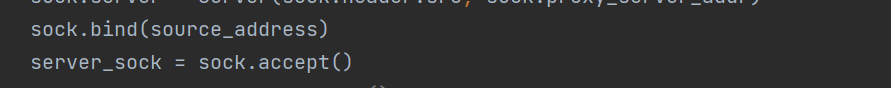
When testing, first comes from “RDT\_test\_case”:  


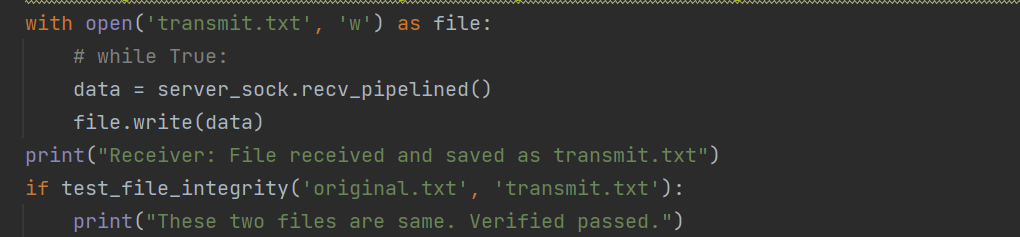
Init a sender and a receiver:  


For sender, bind and connect, the send messages, close:  

For receiver, bind, accept a connection, then receives messages, add the messages together, write to file, then validates files. Finally close.





**Conclusion.**

While UDP is not designed for reliable data transfer, it is entirely feasible to build a robust and efficient reliable data transfer protocol on top of it. This involves implementing mechanisms for packet acknowledgment, retransmission, sequencing, error detection, and flow control. Such custom protocols can offer the benefits of UDP's low overhead and flexibility while ensuring reliable data delivery, making them suitable for a wide range of applications, from real-time communications to custom network services.

Through this experiment, our group learned commonly used tools in the transport layer such as segmentation, congestion control, flow control, and pipeline. Added experience in Python programming and enhanced team collaboration skills.