Report - Networking

Comparison of implemented TCP with standard TCP

Connection Establishment

When the client and server are booted up, the **server waits for the client to join** before allowing any communication with it. This has been done using the `acceptClient()` function in server and `joinServer()` function in client. The server on booting up opens a receive window that registers a client only when it receives the string ***JOIN***. The client on booting up sends the same message to establish a connection. This is only a 2-way handshake.

- **Difference from standard TCP:**

  Standard TCP uses a 3-way handshake to establish a connection. It involves client sending a **SYN** bit, server acknowledging with a **SYN-ACK** packet and the client sending an **ACK** packet in return.

- **Similarities with standard TCP:**

  Some form of connection is being established between the server and client before transmission of data.

Data sequencing

The code splits the data into fixed size chunks of length 5. Each chunk is stored in a `struct dataChunk` which contains details like the **data**, **sequence number**, **acknowledgement bit** and the **total chunks** in the entire data to be sent. This sequencing is done by the `splitInput(char * input)` function which stores all the chunks in a `struct dataStruct` named `transferData`

- **Difference from standard TCP:**

  Standard TCP does not split the data into fixed size chunks. It rather divides the data into **variable sized segments** based on the **Maximum Segment Size** negotiated during the connection establishment. Each dataChunk struct in my code represents the **TCP header**, but does not contain all the information that the original TCP header contains - like error handling.

- **Similarities with standard TCP:**

  TCP sends data in packets after sequencing them and adding TCP headers to all packets. In my code, the similar attempt of dividing into chunks and adding

helper variables to the dataChunk struct resembles the TCP header to some extent.

Retransmission

The sender code has a function to `sendChunk()` and `receiveAck()`; and the receiver code has a function to `receiveChunk()` and `sendAck()`.

The sender uses another function `getNextNonAck()` to get the next non-acknowledged chunk in `transferData` . The function returns -1 when all chunks have been sent, otherwise returns the sequence number of the next non-acknowledged chunk.

For sending data, the sender has a while loop that runs till the `getNextNonAck()` function returns -1. In each iteration it sends the currentChuck which was returned by `getNextNonAck()` and then opens a non-blocking receive window only for `receiveAck()`. In case any previous acknowledgement had been received, that is updated in the function, otherwise the `recvfrom` call instantly returns -1 which is ignored. It does not wait for the acknowledgement message. Chunks are sent and retransmitted if acknowledgement has not yet been received, till all chunks have been acknowledged.

- **Difference from standard TCP:**

  In TCP, the sender would wait for acknowledgement for a specific timeout period before sending the next chunk. The sender maintains a sliding window for acceptable sequence numbers from the receiver. As an ack is received, the sliding window advances. If it is not acknowledged within a certain timeout period, it is retransmitted.

- **Similarities with standard TCP:**

  Out of order acknowledgements have been handled as the socket is non-blocking for the `receiveAck()` function. There is always retransmission until all chunks have been acknowledged to ensure reliable communication between the client and the server.

Marking End of Data

The code sends a special dataChunk, after all the chunks have been successfully sent, that has a sequence number -1 to indicate to the receiver that the entire data has been sent.

- **Difference from standard TCP:**

In TCP, the header for individual packets inform the receiver about the total number of packets to be received.

Flow Control

Flow control in standard TCP is a fundamental mechanism that prevents the sender from overwhelming the receiver with an excessive amount of data.

The receiver consistently communicates it’s capacity to accept data to the sender. When the receiver's buffer nears full capacity, the subsequent acknowledgment temporarily suspends the data transfer, allowing the existing data in the buffer to be processed before further transmission resumes. This process helps maintain a balanced and effective data flow.

The current code does not account for any sort of flow control. The main idea of adding flow control to the current code is to implement some sort of sliding window where data is sent and acknowledged.

Ideal additions to be made to account for Flow Control

1. Add a **concept of sliding window** in both the server and client which is defined before the data has begun its transmission. The sliding window size must be decided based on the data size, network, senders transmission rate and receivers acknowledgement rate
2. The server side window keeps track of **chunks which can be sent**. The receive acknowledgement for only these chunks must be open.
3. The receiver side window keeps track of the chunks which are **expected to be received**. Only received acknowledgement within the window should be sent.
4. The sliding window should **decrease** in the sender side by one or many chunks (decided by the message) when the acknowledgement message for the sent chunk received.
5. The sliding window should be **dynamically resizeable** for the receiver based on the requirements.

Modifications in my code

1. Add a struct for sliding window which should contain sequence numbers. The size of this window can be assumed to be a random number for basic implementation.

2. In the `getNextNonAck()` function, add a condition to get the next non acknowledged chunk part of the sliding window defined. Similarly, in the `receiveAck()` function, ignore ack for chunks not part of the sliding window.
3. In the `receiveData()` function, add a condition to ignore chunks whose sequence numbers don’t belong to the sliding window. The `sendAck()` function should also contain information about how the senders window should change.
4. Every time the sender receives acknowledgement for one or a couple of chunks, modify the sliding window base and bounds accordingly in the sender.
5. The receiver can change the window size based on its requirements, and the same can be conveyed to the sender in the acknowledgement message.