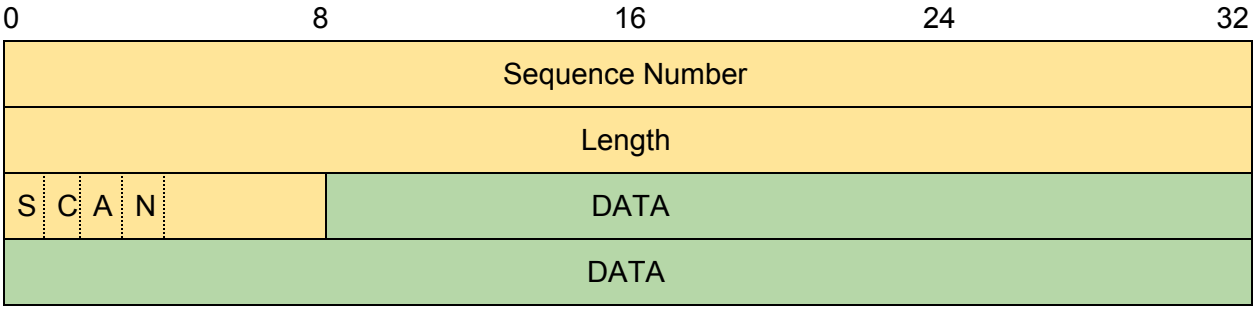
Reliable Data Transfer Protocol

Introduction

In this project, our task was to build a reliable data transfer protocol, over an unreliable channel, where packets can be dropped, lost, corrupted, or be delivered out of order. The sender and receiver have to communicate and recover from these losses. One more requirement of the project was to design a protocol that results in smaller packet size than TCP, and could theoretically be faster than TCP. The losses mentioned above are handled well by TCP/IP protocol, however, TCP can sometimes be slower. To ensure reliability, I have designed a Negative AcKnowledgement (NAK) based protocol, where the receiver sends NAKs, for the lost packets. The design and discussion are covered in more detail below.

Design

The figure below represents a packet structure of the protocol. Sequence number and Length are of 4 Bytes each, and another byte is allocated to various flags, which are explained below.



- *Sequence Number* - Sequence number represents the offset of the data received. The receiver can use this number to copy data in its receive buffer, starting from the index which is equal to this number. The sequence number always starts from zero, and its maximum value can be 2^{31} - 1. Thus, with this protocol, a maximum of 2^{31} - 1 bytes (2.14 GB) can be transferred on a single connection.

- *Length* - Length denotes the length of the data received. When the Start (S) flag is set to true, the value in the length field denotes the total length of the data that the sender will send. The receiver creates a buffer of size Length and accepts data until the buffer is full. For the length field, we don't really need 4 bytes, as UDP can only send 2^{16} - 1 bytes at a

time, so we can get away with using just 2 bytes. However, since at the start of the connection (when S is set to true), the length denotes the total length of the file to be sent, I decided to use 4 bytes.

- *Start (S)* - This flag is used to denote that the connection is starting. The sender sends a Start packet in the beginning, with total length of the data to be sent in the Length field, and the file name in the Data field.

- *Close (C)* - This flag is used to close the connection by the receiver, when it has received everything. The receiver knows from the start how much data it'll receive. Thus, the receiver initiates closing of the connection.

- *ACK (A)* - This flag is used to send Acknowledgement to the sender when the receiver receives a connection request. It is also used by the sender to send ACK at the end when it receives a Close packet from the receiver.

- *NAK (N)* - This flag is used by the receiver to communicate to the sender that it hasn't received a packet with a particular sequence number. When the sender receives this information, it resends the missing data.

Reliability

At the start of the connection, the sender keeps sending the *Start* packet, with start set to 1, length set to the total length of the file to be sent, and file name in the data field, until an ACK is received from the receiver. Even if a *Start* packet gets dropped, the sender won't start sending data until the receiver sends ACK. On receiving the connection request from the sender, the receiver creates a buffer of total size, computes the sequence numbers it expects, and then keeps sending ACK packets until the sender starts sending data packets. Thus, the connection is established between the sender and the receiver, and both of them can deal with loss of packets at this stage. Also, at this stage, both the sender and the receiver works similar to stop-and-wait, thus, packets are always delivered in order.

After the connection has been established, the sender sends all the data it has in multiple packets, and waits either for NAK, heartbeat or a Close packet. Whenever the sender receives a NAK packet, it creates the required packet and sends it across to the receiver. From experiments, I have found that when the sender sends packets as fast as possible, the receiver gets swamped and drops many packets. Thus, I have added a small delay between sending packets of 5 ms - I arrived at this value after experimenting with other values. I found that if we keep larger delays,

e.g. 50 ms, almost all packets are received by the receiver, however, the throughput is much lower. 5 ms provides a good balance between the throughput and the number of dropped packets. On receiving a NAK, however, the sender sends the required packet without any delay.

The receiver processes the data as fast as it can. When it receives a data packet, it compares the received sequence number with the expected sequence number. If the received sequence number is less than the expected number, it simply ignores the packet as it already has the data stored in its receive buffer. If the received sequence is equal to the expected sequence, it stores the data in its buffer. However, if the received sequence is larger than the expected sequence, it first stores the data in its buffer in the correct location, then waits for a small amount of time (50 ms at the start, and then it increases with each NAK), giving the data receiver a chance to receive the data. After the delay, it checks if it has received the data. If it has, then it processes the data, otherwise, it keeps sending NAK to the sender (with some delay in between) until it receives the required packet. The value of delay between each NAK, 50 ms, was arrived at by doing experiments. I found that for smaller wait time, the receiver sends too many NAKs to the sender, and in response, the sender sends an equal number of packets, increasing the traffic, and for larger values, the receiver is too slow to send NAKs, reducing the throughput.

Since, the receiver keeps sending NAKs for a sequence number, until it receives the packet with that sequence number, even if a NAK gets dropped, at least one NAK will reach the sender. The sender will respond with the required data packet. If the data packet gets dropped, the receiver won't receive it and will keep sending NAK. Thus, it is guaranteed that the receiver will receive all data packets from the sender. Also, since the receiver waits for the buffer to get filled up correctly before writing the data to a file, it guarantees in order delivery.

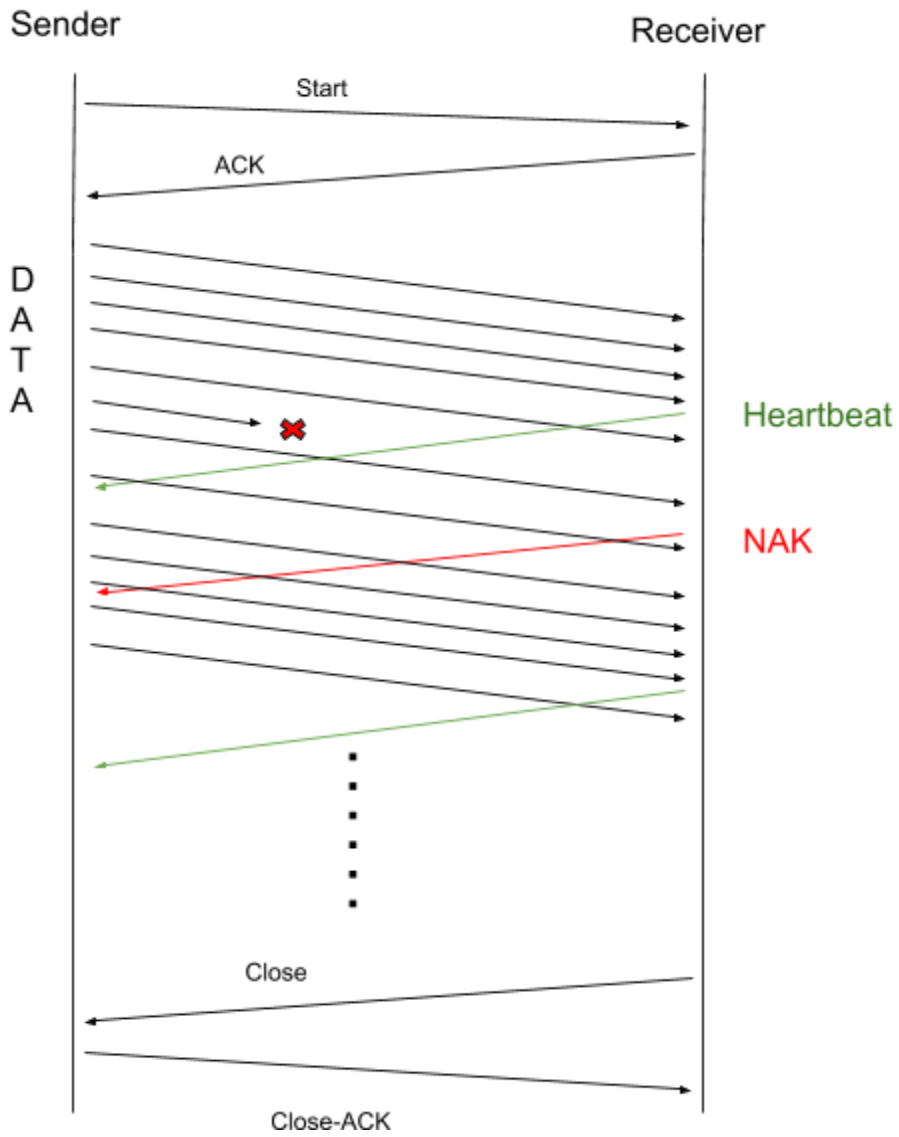
For connection teardown, the receiver sends a Close packet to the sender when it has filled up its receive buffer. The receiver maintains a pointer (expected sequence number) such that it has received and stored data in-order to the left of the pointer. Thus, when the pointer is greater than the length of the receiver buffer, it knows that it has received everything from the sender and asks the sender to close the connection. The receiver keeps sending the Close packet until it receives an ACK from the sender. The sender, on receiving the Close packet, responds with a Close-ACK packet. In this stage, the Close packets sent by the receiver can get dropped, however, since it keeps sending Close packet (at an interval of 100 ms) and waits for an ACK from the sender, it is guaranteed that the teardown will be successful.

Also, both sender and receiver has a timeout. If data is not received during timeout, the program terminates. Thus, the sender keeps sending heartbeats at an interval of 1/4 the timeout duration, which is 10 seconds. One thing that I considered is that the heartbeats can also get dropped because of the noisy channel. But if the channel is noisy, then the data packets will also get

dropped, and in that case the receiver will send NAKs. And if the channel is not very noisy, then the heartbeat packets will also not get dropped very frequently. Because of this, I don't think I the sender needs to send an ACK for the heartbeats.

Since UDP drops packets which are corrupted, although using Internet Checksum, which cannot deterministically say if the packet was corrupted or not, I am still relying on UDP to handle data corruption. Probability of data corruption is quite low these days because of full-duplex wires, and some amount of error checking happens in the lower layers too. Moreover, not implementing additional error checking/correcting techniques like CRC saves computation and results in higher throughput.

Data Flow



Results

With the implementation of the protocol detailed above, I am able to transfer data between two nodes reliably. I am using a window size of 65,000, i.e., a maximum of 65,000 data bytes and 9 header bytes are sent in each packet, which provides an effective throughput of around 11 MB/s for large video files of around 250 MB on my machine. Testing for image files of around 5 MB gives similar results.

The number of NAK packets sent from the receiver to the sender is around 3 % of the total packets sent, i.e., the sender is resending around 3 extra packets for every 100 packets it's supposed to send. Especially at the beginning of the transfer, the receiver sends out more duplicate NAKs (multiple NAKs for the same sequence number), but later on the number of NAKs reduces, and sometimes completely stops. For 250 MB file, around 45 NAKs are sent on average, however, for smaller files, the results may vary.

When a file **xyz.abc** is sent, the received file is named **xyz-received.abc.**