CS 118 TCP Reliable Data Transfer Project Report

Austin Vaday 104-566-193, Yingbo (Max) Wang 604-593-537

I. Implementation Description

a. Header Format

We are basically following the header design of the Transmission Control Protocol (TCP) to design the packet header for our reliable data transfer protocol. The header has 24 bits in total and includes the following fields that are useful in data transmission: Port of the source process(*src_port*), port of the destination process (*dst_port)*, sequence number of the packet(*seq_num)*, ACK number of the packet (*ack_num)*, the SYN/ACK/FIN bit that marks the type of the packet (*offset_reserved_ctrl)* and size of payload in the packet (*data_size)*. We also added the *window, checksum* and *urgent_pointer* fields in our protocol header for completeness, which are included in TCP headers. However they are not set in our current application.

Two fields that are particularly important for identifying if a package is successfully transmitted or not are the sequence number field and the ACK number field. The sequence number, which is meaningful in a normal data packet, specifies the number of current packet in a set of data packet streams. The ACK number, meaningful in an ACK packet, identifies the sequence number of the first data packet it is expecting to receive. Since our reliable data transfer protocol supports bi-directional data transmission, both the client and server should have these two fields. However, the server is mostly sending data (that is, the requested file) to the client particularly, the sequence number of server's packets and ACK number of client's packets are updated more often.

b) Connection Setup & Messages

The client first starts with a three-way handshake with the server similar to that in TCP, to establish a connection session when both the client and server program are executed on hosts. This process is done by the client first sending a SYN packet to the server, the server responding with a SYN-ACK packet to the client and the client responding with another ACK packet to the server. All of these customized packets are implemented above the transport layer User Datagram Protocol (UDP), which does not guarantee correct packet order or packet delivery. We are setting the sequence number and ACK number of these three packets to be (0, 0), (0, 1), (1, 1), but they are not particularly useful as the data transmission state has not been entered yet.

Then the client reads in the file request from *stdin* and sends a data packet encapsulating the file name as payload to the server. The server sends back an ACK packet after receiving the packet; the client retransmits this packet if it hasn't received the server's ACK after the data packet times out, which usually implies a packet loss in the network. Then the server starts transmitting the requested file to the client in data packets, which can be more than one for a large file. The client responds each data packet received with an ACK packet, with the sequence number and ACK number specified in Part (a).

The file is transmitted successfully after the client has ACKed for all packets transmitted from the server. At this time the client initiates to close the connection by sending an FIN packet to the server, which responds an ACK and another FIN packet. The client then sends back another ACK packet and enters the time-wait state for about 30 seconds before closing the connection; during this time it makes sure the second ACK is received by the server.

c) Window Based Protocol & Timeouts

To simulate the behavior of Selective Repeat Protocol in our design, both client and server keeps a local buffer for sending or receiving packets, to support packet pipelining of effective link utilization. The window is implemented as an array of the structure *WindowPacket,* with Maximum Window Size to be 5 packets according to the specification. Each *WindowPacket* contains a packet, the size of this packet, a variable that marks if the current packet has been ACKed by the client (on server) or it has been used to store data (on client), and a variable keeping track of the transmission time of the data packet on server used to detect its timeout.

In addition, we need to shift the packet buffer window to transmit more than MSS packets. On the server side, each *WindowPacket* is moved to its previous slot in the buffer array starting from the second one, if the first packet in the buffer is already ACKed by the client. The process is repeated until a unACKed packet remains in the first slot, and may be done multiple times when packets have been delivered out of order. Then the program reads the requested file to construct data packets to fill up the freed space in packet buffer, and eventually transmits these new packets to the client. Similar process is done on the client side, as the received data packets are shifted out of the buffer until the first slot remains empty. The payload of these packets are extracted and appended to the received file.

Data packet timeout and retransmission is handled by another thread on the server program, which repeatedly traverse through the packet buffer, compare each unACKed packet's last transmission time with current time. It then retransmits the packet if the difference is above a preset timeout threshold, which is 500 ms in our case.

II. Difficulties & Solutions

a) Packet Serialization and Deserialization

The packet structure cannot be sent directly through a UDP socket because it contains pointers to the payload's memory address. Therefore, each field in the packet header and payload should be serialized into a string to be sent on the sender side, and it is de-serialized at the receiver side to retrieve the data. Our de-serialization did not recover the correct data at first, because it mishandled large integer values storing in multiple bytes.

It turns out that we had not considered machine endianness carefully: when storing a variable with a larger size to one with a smaller size, the least significant bytes are copied

into the new variable in Little Endian machines that are typical in Intel x86-64 architectures. Also, we have to be careful when doing bitwise operations after storing a variable with smaller size to the one with larger size: bitwise AND operation with the result should be done with appropriate masks to make sure extra bits are set to zero.

b) Design of sequence and ACK numbers of packets

At first the client cannot receive a complete copy of the file when packet loss occurs, and we realized the problem is at the sequence and ACK numbers used in data and ACK packets: the ACK packet in this previous design does not correspond with the correct data packet, so it cannot be used for correct verification of successful data transmission. After playing around with some demonstrations of the Selective Repeat Protocol and reading some resources, we come up with the new design illustrated in Part 1, which works as expected.

c) Handling boundary conditions of packets

There would be situations when an ACK packet arrives at the server with its ACK number outside the range of its current packet buffer, or a data packet arrives at the client with its sequence number outside its buffer's range. This usually happens when the network delay is larger than usual, premature timeouts happen and unnecessary retransmission are triggered. Since each arriving packets would be handled inside the packet buffer, we add in extra conditional expressions to avoid unexpected behaviors for out-of-bound packets. Essentially the client sends an ACK to the out-of-range data packet it receives, and the server just ignores the out-of-range ACK packet, because these packets only contain redundant information.