Lab 3 Part 1: Build Your Own Transport

1. Introduction

In this lab I wrote code for a sliding window transport layer protocol on top of the IP layer. This code is a minimal stripped-down version of TCP called cTCP and it runs at the user level (instead of kernel level where the regular TCP usually runs). I created both the client and the server components of cTCP.

In a traditional OS with regular TCP in the Linux kernel, when the kernel received a TCP segment for a cTCP connection, it does not recognize the connection and immediately sends a reset (segment with RST bit set) in response, telling the other side to terminate the connection. The kernel has been modified in the VM we are currently using to prevent such behavior and the kernel does not send resets in response to unrecognized TCP segments and so allows cTCP to run.

The key elements to understand as part of this implementation was how TCP sliding window works. Key understandings were:

a) Dealing with lost segments, truncated segments and corrupted segments.
b) Data structures to handle segments that need to be sent and that need to be acknowledged.
c) Keeping a track of sender and receiver window sizes based on the sequence number and the acknowledgement number and shifting the window when necessary.
d) Keeping track of when EOF is received and when a FIN needs to be sent.

2. Program Structure, Design and Tools Used

A. Program Structure

a) ctcp.c – This file contains the main cTCP implementation code. The functions that I needed to write as part of the code was ctcp_init(), ctcp_destroy(), ctcp_read(), ctcp_receive(), ctcp_output and ctcp_timer(). I also wrote a few helper functions like send_available_segments(), send_single_segment(), check_unacked_segments() and send_only_ack() to aid in reducing repeated code and improve code readability at the same time.

B. Design Details

a) The crucial part of the implementation was to choose data structures which needed to be a part of the ctcp_state structure which is the main part of the code as the ctcp_state is the structure which revolves around the entire cTCP implementation.
b) ctcp_config will store all the configuration information. last_ackno_recv will store the acknowledgment number received while reading. last_seqno_recv will store the last sequence number received while reading. last_seqno_sent will record the highest sequence number that was sent via the output. last_seqno_accepted will track the sequence number while sending the data. recv_EOF will turn true if the EOF is received

at any end. recv_FIN will turn true if the FIN is received at any end. I also have a linked list of unacked_segments which will keep a track of the segment which have been sent and have not been acknowledged yet. The segments_to_be_sent linked list will keep a track of the segments which have been prepared and are ready to be sent once space on the network becomes available.

c) First, I am initializing all the data at the ctcp_init() function and setting all the state config values to the values obtained from at one of the arguments of the ctcp_init() function.
d) Then I am also initializing all the other values declared inside my state structure to zero.
e) Once reading is started, I am calling conn_input() and storing all the data that was read in a buffer and also calculate the total size of the data that was read to keep a track of the sequence numbers.
f) I then copied the data that was read in to buffer to the actual segment to be sent.
g) If there is no data available, I will not do anything since it is neither a data segment or an EOF indication.
h) If conn_output() function returns a -1 then it indicates an EOF and I am building the EOF segment to be sent.
i) After checking all the conditions of conn_output() I am then calling a helper function send_available_segments() to send the segment.
j) In send_available_segments() I am first checking if that segment already exists in my list of unacknowledged segments.
k) If it is not present and it was also not transmitted before, I then call another helper function send_single_segment() and send the segment.
l) If the segment is already present in the unacknowledged list, then we need to resend it. To resend it I first check if the retransmission timeout expired and only call the helper function send_single_segment() if the RTO expired.
m) In send_single_segment() I am checking if it exceeded more than 6 transmission and if a segment has, I discarded it since the receiver might not be responsive.
n) If it was not transmitted more than 6 times, then I created the segment to be sent and recalculated a new checksum and then called conn_send() to send the segment. I also incremented the sequence number and recorded the new timestamp required for RTO.
o) If conn_send() returns an error, then I just destroy the segment, since it could be an error because of some issue in the transmission and we will again successfully be able to send in another retransmission.
p) In ctcp_receive() I first check If the length of the segment received is less than the length it should actually be and return if it is smaller.
q) Then make sure we received the correct checksum and continue to other parts of receive.
r) If valid data was received, but it does not belong to the current receive window, then I discard the packet and send the ack with the expected value of sequence number.
s) If the segment received was an ACK, then we need to increment the acknowledgement number to the correct number of bytes that was successfully received.

t) If the data was valid and belongs to the current window size, then add to the segments to be sent list and later call ctcp_output() at the end.
u) Loop through the list again and again and keep a check that the segments are sorted
v) If a FIN was received, then I send an ACK accordingly by calling ctcp_output().
w) After calling ctcp_output for any segment that needs to be sent, I also call a helper function which will check my list of unacknowledged segments and clear any segments for which acknowledgement has already been received.
x) In ctcp_output() function, I am first calling conn_bufspace() to make sure there is buffer space that can be written into.
y) If there is no space, I will not send anything and perform flow control that way.
z) If there is some place to send, I will send those many bytes accordingly.
aa) Meanwhile if I receive a FIN, I also send out an acknowledgement for that corresponding FIN that was received.
bb) In ctcp_timer() I am iterating through all the states to check for any segments that need to be sent again. I am calling helper function send_available_segments() which takes care of sending retransmission if required.
cc) If an EOF is received and if a FIN is received and there are no unacknowledged segments and segments to be sent, I will then call ctcp_destroy() function to destroy the state.

C. Tools Used

a) GDB to identify memory corruption bugs and isolate bugs in code
b) Git and Github used as version control systems to continuously save the code progress on the local repository and the cloud repository respectively
c) Sublime Text editor to write and edit code

3. Implementation Challenges

The major implementation challenge I faced was identifying which variables to use in the ctcp_state structure. I was able to understand which variables to use as and when I went ahead with the implementation. I also declared some local variables inside variables to track the various sequence number and acknowledgement numbers. Also managing the sorted order of the receive window was a major challenge and needed to maintain a separate linked list to keep track of the out of order received segments.

4. Testing and Extra Tests

a) Basic TCP functionality:
The test sometimes shows some unpredictable behavior, and, in most cases, more than 15 test cases pass out of the required 17 tests. The Google and Bing test never pass, which needs to be discarded as per the post on piazza. Please run the test at least 5 times and then consider the highest score as per the piazza post.

b) Dumbell Topology:
The dumbbell topology passes all the required cases and I am able to get 240/240 points.

c) I2 Topology:
All the path and the cget tests pass which are a part of this and I can get a full score here.

5. Remaining Bugs and Unexpected Cases

a) I have not handled the case when I receive an ACK or FIN along with data segment. This is the case when there is piggybacking of segments.