Lab3-Part1-Report

Qiushi Xu

qiushixu@usc.edu

In Part1 of lab3, I implemented the logic of cTCP with required functionality of In-order delivery, Multiple clients and Sliding window.

1. Program Structure and Design

For better maintenance of the TCP transmit-receive window, I created two data structures to separately represent these two states:

send_state_t: records transmission state of the current TCP connection, including the last received ACK number from the other side, the last byte read from input, the last byte sent, whether I have read an EOF or error from input, and a linked list to store the segments that have been sent but not ACKed yet(for convenience I call it unacknowledged_list in the rest of report).

receive_state_t: records receiving state of the current TCP connection, including the last sequence number I received, the statistics of corrupted segments(Truncated, out of receive_window and invalid checksum) I received since TCP connection is established, whether I received FIN from the other side and a linked list to store the segments that I received inside the range of current receive_window but not been outputted yet(for convenience I call it unoutputted_list in the rest of report).

to_send_ctcp_segment_t: More specifically, in order to keep track of the segment I send, I also write this new data structure, which wraps a ctcp_segment_t inside, and adds two more fields to record the last time segment was sent and how many times this segment already been sent.

In addition, I added several helper functions to make code more clearer and cleaner:

void ctcp_send(ctcp_state_t *): is called at the end of ctcp_read(), and will send the segments stored in unacknowledged_list as many as much possible, as long as which are in the current transmit window.

void ctcp_send_segment(ctcp_state_t *, to_send_ctcp_segment_t *): set the remaining fields of the specific segment and sends it using conn_send().

void ctcp_send_ack_segment(ctcp_state_t *): send the ACK segment to the other side with no data payload. The segment just contains control messages, indicating the last byte received and the current window size.

void clean_list(linked_list_t *): clean out the entire list with every object freed.

void clean_acked_segments(ctcp_state_t *): clean all the segments in unacknowledged_list which are already been acknowledge by the other side.

For the other existing ctcp functions which need to be completed, I just followed the instructions and use the helper functions. There's not much to talk about. For greater details and

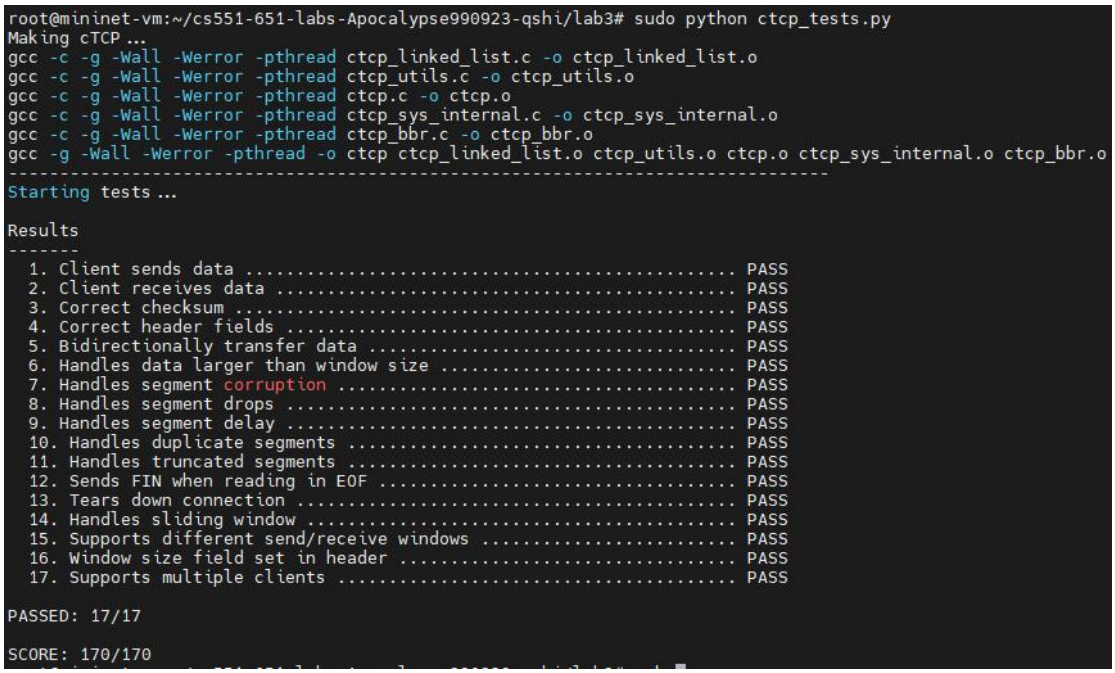
better understanding, I made **README_part1.md** file which provides the comprehensive explanation of the data structures as well as the ctcp functions.

2. Implementation Challenges

One challenge during implementation is **Sliding Window**. At the first stage, I just implemented the Stop-and-Wait protocol, where I simply call send segment in ctcp_read() function. But later I found that in order to meet the functionality of Slide Window, it's difficult to implement inside ctcp_read() without the help of more data structures and functions to keep track of the state of transmit window and receive window. So I created the more data structures such as **send_state_t** and **receive_state_t**. Also I encapsulated the whole process of sending segments based on Sliding Window as a separate function **void ctcp_send()**.

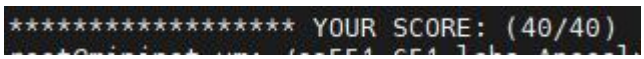
3. Testing

I already tested the code with tester **ctcp_tests.py**, and it passed



One thing I had to mention is that when I was debugging the test cases in ctcp_tests.py, I wrote some print statements. But don't worry, now they are all been commented out, so you may see there's commit change to ctcp_test.py because of that.

In addition, I also tested the code through **Dumbbell topology** and **Mini-Internet topology**, and it passed them all.



When it comes to my general strategy to debug and test, that is manually build and run the code in both client and server mode with commands:

Server: sudo ./ctcp -s -p [server port]

Client: sudo ./ctcp -c [server]:[server port] -p [client port]

For edge cases that sending segments may overwhelm the window, I tested with large window and large file input.

Server: sudo ./ctcp -p 8888 -s > new_file.txt

Client: sudo ./ctcp -p 9999 -c localhost:8888 < file.txt

And I also tested Memory leaks with command:

sudo valgrind --leak-check=full --show-leak-kinds=all ./ctcp -p 8888 -s

```
root@mininet-vm:~/cs551-651-labs-Apocalypse990923-qshi/lab3# sudo valgrind --leak-check=full --show-leak-kinds=all ./ctcp -p 8888 -s
==15114== Memcheck, a memory error detector
==15114== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==15114== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==15114== Command: ./ctcp -p 8888 -s
==15114==
[INFO] Cleaning up old connections ... done!
[INFO] Server started
==15114==
==15114== HEAP SUMMARY:
==15114==     in use at exit: 0 bytes in 0 blocks
==15114==   total heap usage: 6 allocs, 6 frees, 1,942 bytes allocated
==15114==
==15114== All heap blocks were freed -- no leaks are possible
==15114==
==15114== For counts of detected and suppressed errors, rerun with: -v
==15114== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Also I used fprintf() function to print debug information to STDERR, but now they are unable to compile due to macro definition and will not show any debugging output any more.

4. Remaining Bugs

In ctcp_tests.py, occasionally it may fail in case4(Correct header fields), but this situation is extremely rare.