An Pham
9477698118

CSCI 551 Lab 3 Part 1 Report

Overview: For lab 3a we implemented our own version of the transport layer using protocols such as: stop-and-wait, sliding window, piggyback acknowledgement, selective arq, etc. This implementation took place at the user level; however, with the support of ctcp_sys_internal.c/h, we are able to interoperate with actual tcp socket connections, which takes place at the kernel level. Our program was able to use http to get webpages.

Tools & Environment:

- Virtual Box: Custom CSCI 551 image, which supports tcp changes at the user level
  - We use this environment to test our code
- Visual Studio Code:
  - Text editor with various functionalities, providing ease of development
  - Ssh-remote: to avoid using VM's GUI
- GDB:
  - Allows us to debug our code
- Valgrind:
  - Allows us to detect memory leaks within our code

Program Structure and Design: To keep things simple, the whole implementation took place within ctcp.c. By isolating our changes to this specific file, we didn't have to change other source files or change the Makefile, which affects the way the executable is built.

Design Goal & Implementations:

1. Avoid overwhelming the receiver
   a. Config.send_window & Config.recv_window to know the initial window size of sender and receiver
   b. Utilize a current_window_size variable to track how much buffer space we currently have
2. Reliable transmission
   a. Utilize checksum() function & segment->cksum variable to check for packet corruption
   b. Utilize config.rt_timeout to handle packet delay
3. In-order deliver
   a. Cumulative Ack allows us to know which segments have already been acknowledged and outputted
   b. Utilize the curr_ackno and prev_ackno variables to keep track of packet order

Function implementations:

ctcp_init(): Here we initialize the member variables of ctcp_state. Linked list member variables are initialized via: "ll_create()," curr_seqno & curr_ackno are initialized to 1, and everything else is initialized to 0.

ctcp_destroy(ctcp_state * state): Here we clear memory for a specific connection state. Specifically, we traverse any linked_list members within the state, remove the nodes, free() the objects, and finally free() the list itself.

ctcp_read(): This function keeps using conn_input, until either 0 or -1 is detected.

- If read_result > 0, we package these input into segments
- If read_result == -1, we set our EOF_FLAG to true, meaning the host stops packaging & sending segments, then the sends a TH_FIN segment to the receiver to tell the receiver that it's ready to close down on its part

ctcp_receive(): This function processes segments with data as well as segments without data but has flags to control the communication between 2 hosts.

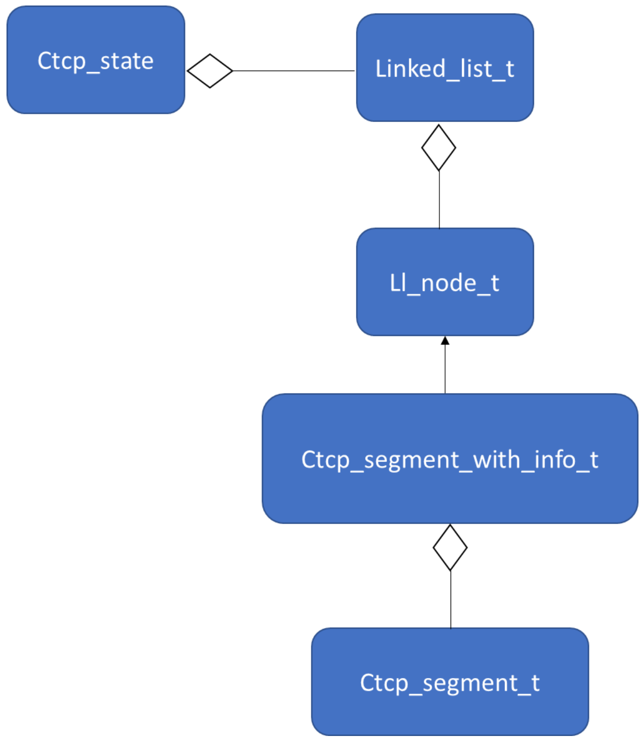
- We start out the function by doing various checks on the incoming segment & do not proceed in acknowledging the segments if:
    - If the segment->len < length: this indicates that the segment has been truncated
    - If the checksum doesn't match: the packet is corrupted
    - If the segment isn't in the receiving windows range
- If everything passes we send an acknowledgement back for the incoming segment

ctcp_output(): This function outputs segments that are acknowledged and ready to be outputted; however, it starts by checking if there is enough buffer space to output such a segment. We do not output segments that doesn't have data, such as ACK & FIN segments.

ctcp_timer(ctcp_state * state): The function traverses all the conn_states & then traverses all the unacknowledged segments of the curr_state. The inner loop checks for timeouts & retransmission numbers, while the outer loop checks the entire conn_state if all conditions are met to be ctcp_destroy(curr_state).

Implementation Challenges: One of the biggest challenges in implementing ctcp was defining the necessary data structures & adding the right fields into ctcp_state_t to support the lab. Changing the struct's member variables,causes code refactor that propagates to different functions and sections of the code. For example, after defining my "ctcp_segment_with_info_t" struct, I had to propagate the changes to ctcp_state class & the different functions that use it. The following picture shows the dependencies of classes.

Complex Class Diagram & Dependencies:



I introduced a new class called "ctcp_segment_with_info_t," which is a wrapper class over the ctcp_segment_t class. This wrapper added extra information to keep track of the number of retransmits & timestamp of each ctcp_segment_t. This design was necessary, since none of the variables in the original ctcp_segment_t struct keeps track of there information.

Testing:

1. Basic TCP Functionality: For these test cases, I can get anywhere from 15/17-17/17 test cases correct. It's unpredictable how the results will come out
2. Dumbbell topology: Unable to receive files over mini-net; however, we're passing the valgrind test for memory leaks
3. Mini-Internet topology: Unable to cget over the I2 network topology

Remaining bugs: I believe my remaining bug is having the client receive file from the server, while the client doesn't send anything (meaning it has an EOF from the very beginning & shouldn't shut down)