Project 3: TCP Implementation

Introduction

TCP serves many purposes: it provides reliable, in-order delivery of bytes, it makes sure the sender does not send too fast and overwhelm the receiver (flow control), and it makes sure the sender does not send too fast and overwhelm the network (congestion control). It also aims to be fair: when multiple senders share the same link, they should receive roughly the same proportion of bandwidth. In this project, you will demonstrate your understanding of the TCP basics by implementing the TCP protocol.

Be prepared: this is a single-person project, and your skills will be exercised. So start early and feel more than welcome to ask questions. However, please note that the TAs are not allowed to debug your code for you during their office hours. They cannot touch your keyboard, and they will only spend a maximum of 10 minutes reading your code. This helps them to assist more students effectively. Therefore, it is recommended that you visit their office hours with specific questions instead of vague statements like "my code doesn't work." For debugging, logging or tools like Wireshark[1] can be helpful. There are many ways to do this; be creative.

TCP Recap

TCP is a transport layer protocol that enables different devices to communicate. As a reminder the basic setup is the following.

You have an initiator (client) and a listener (server). The listener is waiting to receive a connection from the initiator. After a connection is received, they perform a TCP handshake to initiate the connection. Afterward, they can read and write to each other. From the application layer, reads and writes to the socket are buffered before being sent over the network. This means that multiple reads or writes might be combined into a single packet or the opposite, that a single read or write to a socket might be split into many packets.

To establish reliable data transfer, TCP must manage many different variables and data structures. Here are some examples of the details you'll need to track.

[1] https://www.wireshark.org/

Let’s say we have sockets A and B, and that socket A wants to start sending data to socket B. A will store the data in a buffer that it will pull from for sending packets. Socket A will then create packets using data from the buffer and send as many as it is allowed to send based on the congestion control algorithm used. As Socket B receives packets, it stores the data transmitted in a buffer. This buffer helps with maintaining the principle of in-order data transmission. B sends ACKs as responses to A to notify A that various bytes have been received up to a certain point. B is also tracking the next byte requested by the application reading from the socket, and when it receives this byte, it will forward as much data in order that it can to the application buffer. For example, if B is looking for byte number 400, it will not write any other bytes into the application’s buffer until it receives byte number 400. After receiving byte number 400, it will write in as many other bytes as it can. (If B had bytes 400-1000 then all would be written to the application buffer at the time of receiving the packet with byte 400. As packets are ACK’d, socket A will release memory used for storing data as they no longer need to hold onto it. Finally, either side can initiate closing the connection where the close handshake begins.

As both sides (initiator and listener) can both send and receive, you’ll be tracking a lot of data and information. It’s important to write down everything each side knows while writing your implementation and to utilize interfaces to keep your code module and re-usable.

Project Specification

You are implementing the interfaces in tcpSock.py, such as `case_socket` and `case_close`. Your code will be tested by us creating other Python files that will utilize your interface to perform communications. The starter code has an example of how we might perform the tests, we have a client.py and server.py which utilize the sockets to send information back and forth. You can add additional helper functions to tcpSock or change the implementation of the 4 core functions (`socket`, `close`, `read`, and `write`). However, you cannot change the function signature of the 4 core functions. Further, we will be utilizing grading.py to help us test your code. We may change any of the values for the variables present in the file to make sure you aren’t hard-coding anything. Namely, we will be varying the packet length and the initial window variables.

Starter Code

The following files have been provided for you to use:

- `packet.py`: this file describes the basic packet format and header. You are not allowed to modify this file.

- `grading.py`: these are variables that we will use to test your implementation; please do not make any changes here, as we will replace them when running tests.

- `server.py`: this is the starter code for the server side of your transport protocol.

- `client.py`: this is the starter code for the client side of your transport protocol.

- `tcpSock.py`: this contains the main socket functions required of your TCP socket, including reading, writing, opening, and closing.

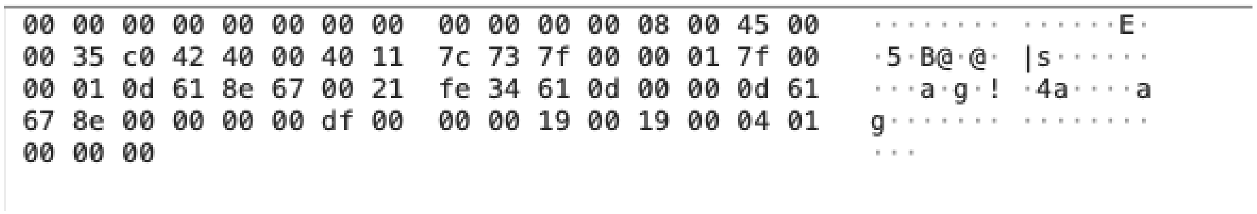
- `backend.py`: this file contains the code used to emulate the buffering and sending of packets. This is where you should spend most of your time.

All the communication between your server and the client will use UDP as the underlying protocol. All packets will begin with the common header described in `packet.py` as follows:

- Identifier [4 bytes]
- Source Port [2 bytes]
- Destination Port [2 bytes]
- Sequence Number [4 bytes]
- Acknowledgement Number [4 bytes]
- Header Length [2 bytes]
- Packet Length [2 bytes]
- Flags [1 byte]
- Advertised Window [2 bytes]

All multi-byte integer fields must be transmitted in network byte order. `socket.ntoh` and `socket.hton` and other related functions will be very important for you to call. All integers must be unsigned, and the identifier should be set to 3425. You are not allowed to change any of the fields in the header. Additionally, packet length cannot exceed 1400 in order to prevent packets from being broken into parts.

You can verify that your headers are sent correctly using wireshark or tcpdump. You can view packet data sent including the full Ethernet frames. When viewing your packet, you should see something similar to the below image; in this case, the payload starts at 0x0035. The identifier - 3425 - shows up in hex as 0x0d61.



```
00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········ ······E·
00 35 c0 42 40 00 40 11  7c 73 7f 00 00 01 7f 00   ·5·B@·@· |s······
00 01 0d 61 8e 67 00 21  fe 34 61 0d 00 00 0d 61   ···a·g·! ·4a····a
67 8e 00 00 00 00 df 00  00 00 19 00 19 00 04 01   g······· ········
00 00 00                                           ···
```

Implementation Tasks

1. TCP Handshakes - Implement TCP start and end handshakes before data transmission starts and ends [1]. This should happen in the constructor and destructor for case tcp socket.

2. Flow Control - You will notice that data transfer is very slow. That is because the starter code is using the Stop-and-Wait algorithm, transmitting one packet at a time. You can do much better by using a window of outstanding packets to send on the network. Extend the implementation to: 1) Change the sequence numbers and ACK numbers to represent the number of bytes sent and received (rather than segments) 2) Implement TCP's sliding window algorithm to send a window of packets and utilize the advertised window to limit the amount of data sent by the sender [2].

3. RTT Estimation - You will notice that loss recovery is very slow! One reason for this is the starter code uses a fixed retransmission timeout (RTO) of 3 seconds. Implement an adaptive RTO by estimating the RTT [3].

4. Duplicate ACK Retransmission - another reason loss recovery is slow is that the starter code relies on timeouts to detect packet loss. One way to recover more quickly is to retransmit whenever you see triple duplicate ACKs. Implement retransmission upon receipt of three duplicate ACKs.

For 325 students, you only need to implement Tasks 1 and 2; you will get extra credits for implementing all 4 tasks. For 425 students, you will need to implement all of them.

References

[1] TCP connection establishment and termination:
https://book.systemsapproach.org/e2e/tcp.html#connection-establishment-and-termination

[2] TCP sliding window:
https://book.systemsapproach.org/direct/reliable.html#sliding-window
https://book.systemsapproach.org/e2e/tcp.html#sliding-window-revisited

[3] Adaptive retransmission:
https://book.systemsapproach.org/e2e/tcp.html#adaptive-retransmission

Useful Links

Programming in Python

The 8th edition of the textbook (and what we've discussed in class) uses sockets in Python. A Python socket tutorial is http://docs.python.org/howto/sockets.html

It may also help by reading the system implementation of epoll at https://codebrowser.dev/glibc/glibc/sysdeps/unix/sysv/linux/sys/epoll.h.html. You will better understand the epoll events.

REMINDERS

- Submit your code to Canvas in a zip file
- If your code does not **run**, you will not get credits.
- Document your code (by inserting comments in your code)
- DUE: 11:59 pm, Friday, April 26th.