15-441: Computer Networks
Project 2: TCP in The Wild

TAs: Ranysha Ware (Ray) <rware@andrew.cmu.edu>
Alex Bainbridge <abainbri@andrew.cmu.edu>

Assigned: February 19, 2019
Final version due: March 29, 2019

1 Overview

In class, we have been talking about TCP, the default transport protocol on the Internet. 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 class, we discussed TCP Reno. A variant of TCP Reno, NewReno used to be the standard TCP on the Internet. But these are just one of many congestion control algorithms (CCAs). Companies use different TCPs depending on the context, for example, one TCP for data centers and another for serving web content.

If you are curious learn more about TCPs in the real world, here is are some fun CCAs to read about: Microsoft and Google use very different approaches to TCP in their datacenters. DCTCP is a TCP for datacenters designed by Microsoft. TIMELY is a TCP for datacenters designed by Google. For web content, the most common algorithm – and the default in Linux servers – is called Cubic. Akamai, the largest content distribution network in the world, uses a proprietary TCP called FastTCP.

In this project, you will demonstrate your understanding of the TCP basics by implementing TCP Reno. You will then use your own engineering skills to design a modern TCP for the datacenter, the Internet, or Earth to Moon communication.

1.1 Checkpoints and Deadlines

The timeline for the project is below, including two checkpoints. To help you pace your work, remember that checkpoints represent a date by which you should easily have completed the required functionality. Given the timeline, you can see that this means you should get started now! The late policy is explained on the course website.

Date	Description
February 19	Project released.
March 4	**Checkpoint 1**: Data Serialization and transmission
March 22	**Checkpoint 2**: TCP Reno implementation
March 29	**Final Project Deadline by 11:59 P.M.**

2 TCP Re-Hash

TCP is a network layer protocol that enables different devices to communicate. There are a variety of different TCP protocols such as Reno, New Reno, Cubic, and more. For this project we are focusing on Reno. As a reminder the basic setup of Reno 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 initate the connection. Afterwards, they can perform read and writes 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.

In order to establish reliable data transfer, TCP has to manage lots of different variables and data structures. Here is an example of some of the details you'll need to track.

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, will forward as much data in order that it can to the applications 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.

3 Project specification

3.1 Background

This project will consist of three checkpoints. The first checkpoint will have you build the basics of TCP: reliable, in-order delivery and flow control. In the second checkpoint, you will implement the Reno CCA and demonstrate to us that it is correct. In the third and final checkpoint, you will design and implement your own CCA for one of three scenarios: a datacenter, the Internet, or Earth to Moon communication.

3.2 What are you actually turning in

You are implementing the cmu_tcp.h interface. Your code will be tested by us creating other C 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.c and server.c which utilize the sockets to send information back and forth. You can add additional helper functions to cmu_tcp 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.h 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 fluctuating the packet length, and the initial window variables.

Additionally, for each checkpoint you will need to provide a graph showing the number of packets in flight (or unacked packets) vs time. We have provided you with a samplefile in the test directory that you can transfer and graph. We have also provided a python file called gen_graph.py to help you generate the graph. It should be setup to monitor packets sent to and from the sender's perspective - you may need to change and update the gen_graph.py script in order to provide a quality graph.

4 Checkpoint 1

In this checkpoint, you will implement windowed transmission, where more than one packet can be "on the wire" at the same time. You will start by extending a simple "stop and wait" implementation that we provide.

4.1 Learning Objectives

With this section, you will learn to think about data packetization and reconstruction, TCP windowing, and packet retransmission. You will also continue to strengthen your C programming skills.

4.2 Starter Code

The following files have been provided for you to use:

- cmu_packet.h: this file describes the basic packet format and header. You are not allowed to modify this file until the final submission! The scripts that we provide to help you graph your packet traces rely on this file being unchanged.
- grading.h: these are variables that we will use to test your implementation, please do not make any changes here as we will be replacing it when running tests.
- server.c: this is the starter code for the server side of your transport protocol.
- client.c: this is the starter code for the client side of your transport protocol.
- cmu_tcp.c: this contains the main socket functions required of your TCP socket including reading, writing, opening and closing.
- backend.c: this file contains the code used to emulate the buffering and sending of packets. This is where you should spend most of your time. gen_graph.py: Python script that takes in a pcap file and graphs your sequence numbers by time.
- cmu_packet.h All the communication between your server and client will use UDP as the underlying protocol. All packets will begin with the common header described in cmu_packet.h as follows:
  - Course Number [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]
- Extension length [2 bytes]
- Extension Data [You Decide]

All multi-byte integer fields must be transmitted in network byte order. ntoh, hton, and friends will be very important functions for you to call! All integers must be unsigned, and the course number should be set to 15441 (the scripts rely on this). You are not allowed to change any of the fields in the header, with the exception of the extension data which you may want to modify in Checkpoint 3. Additionally, plen 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 0x0020. The course number - 15441- shows up in hex as 0x00003C51.

```
0000  02 00 00 00 45 00 00 2c  37 f9 00 00 40 11 00 00   ....E.., 7...@...
0010  7f 00 00 01 7f 00 00 01  db bd 3c 51 00 18 fe 2b   ........ ..<Q...+
0020  00 00 3c 51 00 10 00 10  00 00 00 00 00 00 00 00   ..<Q.... ........
```

4.3 Checkpoint 1 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 cmu_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]. You do not need to implement Nagle's algortihm.

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 with Jacobson/Karels Algorithm or using the Karns/Partridge algorithm [3].

4. Duplicate ACK Retransmission - Another reason loss recovery is slow is 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 on the receipt of 3 duplicate ACKs.

5 Checkpoint 2

Once you have implemented the basics, you can add a Congestion Control Algorithm (CCA) to handle overloading in the network. You will implement TCP Reno, as discussed in class. Hence, the number of outstanding (unACKed) packets will now be be min(window size, congestion window size). You will have to demonstrate to us using traces of real connections that your TCP Reno implementation uses Additive Increase under normal operation, and Multiplicative Decrease under loss.

5.1 Learning Objectives

With this section, you will learn how TCP manages to maintain a view of the network state, as well as how the TCP state machine works. The TCP state machine can be found **here** for reference. Implementing TCP reno will allow you to transfer data faster as you utilize more available bandwith for data transmission. Additionally, fast recovery will allow you to retransmit packets that have been lost.

5.2 Checkpoint 2 Tasks

1. Basics - Review and understand the TCP state machine in depth!
2. Update the advertised window the receiver sends to be the difference between MAX_NETWORK_BUFFER and the total amount of data currently buffered.
3. Congestion Control - Implement the slow start and congestion avoidance features of the state machine, this should make your implementation like TCP Tahoe.
4. Fast Recovery - Implement fast recovery, this should move your implementation from Tahoe to Reno.

6 Checkpoint 3

We are releasing the rest of checkpoint 3 at a later date as we finalize the wording and expectations.

7 Testing Your Code

7.1 Virtual Machine

For this project, you will do all of your development on your own machine using VMs. You should install VirutalBox [10] and Vagrant [9] on your own machine. We have provided a `Vagrantfile` for two VM's, client and server. If you keep the `Vagrantfile` in the same directory as the `15-441-project-2`, folder then you can edit code on your machine, or on either VM, and Vagrant will automatically sync it to both VMs `/vagrant` directory. Refer to references for more information on how to install and use VirtualBox and Vagrant.

The server and client are connected via a private network with IP addresses `10.0.0.1` and `10.0.0.2` respectively. (Note: your code should work even if these IP addresses were changed). The interface name for these addresses is `eth1` on both machines.

The following sections we describe the tools that are already installed on the VMs to help you (and us) test your code. You can also install any additional tools you need. Please document any additional tools you install to run tests in the test.txt file.

7.2 Control the network characteristics with `tcconfig`

`tcconfig` [4] is installed on the VMs to enable you to control the network characteristics for traffic between the VMs. The initial default settings on the VMs are a 20ms delay on both machines (so the total RTT is 40ms), and 100Mbps bidirectional bandwidth. Running `tcshow eth1` on the VMs will show you these settings. Refer to the references for more information on how to use `tcconfig` to simulate different network characteristics including packet loss, reordering, and corruption which will be useful for testing your code.

7.3 Capture and analyze packets with `tcpdump` and `tshark`

`tcpdump` [7] and Wireshark (terminal program: `tshark` [5]) are installed on the VMs to enable you to capture packets sent between the VMs and analyze them. We provide the following files in the directory `15-441-project-2/utils/` to help with packet analysis (feel free to modify these if you want):

- `utils/capture_packets.sh`: A simple program showing how you can start and stop packet captures, as well as analyze packets using `tshark`. The `start` function starts a packet capture in the background. The `stop` function stops a packet capture. Lastly, the `analyze` function will use `tshark` to output a CSV file with header information from your TCP packets.
- `utils/tcp.lua`: A Lua plugin so Wireshark can dissect our custom cmu packet format [6]. `capture_packets.sh` shows how you can pass this file to `tshark` to parse packets. To use the plugin with the Wireshark GUI on your machine, you add this file to Wireshark's plugin folder [8].

7.4 Running tests with `pytest`

There are many ways you can write tests for your code for this project. To help get you started, `pytest` [11] is installed on the VMs and we provide example basic tests in `test/test_cp1.py`. Running `make test` will run these tests automatically. You should expand these tests or use a different tool to test your code (but `make test` should still run your tests). As in Project 1, you should also use standard C debugging tools including `gdb` and `Valgrind` which are also installed on the VMs.

7.5 Running a large file transfer

The starter code `client.c` and `server.c` will transmit a small file, `cmu_tcp.c` between the client and server. You should also test your code by transmitting a larger file (ex: 100MB file), capturing the packets, and plotting the number of unacked packets vs. time. You will turn in a PDF of this graph and this PCAP file.

You can use the utilities described in 7.3 to create `submit.pcap` by running the following commands:

Start tcpdump and the server:

```
vagrant@server:/vagrant/15-441-project-2$ make
vagrant@server:/vagrant/15-441-project-2$ utils/capture_packets.sh start submit.pcap
vagrant@server:/vagrant/15-441-project-2$ ./server
```

Start the client:

```
vagrant@client:/vagrant/15-441-project-2$ ./client
```

When the client and server code finishes running, stop the packet capture on the server:

```
vagrant@server:/vagrant/15-441-project-2$ utils/capture_packets.sh stop submit.pcap
```

7.6 Autolab

There will be no tests in Autolab. Autolab will only be used to submit your code. We will pull everyones code and run our tests in the VMs. We are supplying these VMs to you to aid in your development and so that you can also test your own code.

Make sure that you cover the basics such as no seg-faults, reliable data transfer, and error handling. Then ensure that your code manages flow control and congestion control according to the given checkpoint.

8 Hand-In

As in Project 1, code submission for checkpoint and the final deadline will be done through Autolab (autolab.cs.cmu.edu). Every checkpoint will be a git tag in the code repo. To create a tag, run

`git tag -a checkpoint-<num> -m <message> [<commit hash>]`

with appropriate checkpoint number and custom message filled in. (Put whatever you like for the message — git won't let you omit it.) The optional commit hash can be used to specify a particular commit for the tag; it you omit it, the current commit is used. For the checkpoint, you will be expected to have a working Makefile, and whatever source needed to compile a working binary. **Checkpoints that do not compile will NOT be graded.** To submit your code, make a tarball file of your repo after you tag it. Then login to autolab website, choose `15-441: Computer Networks (S19) -> project2cp<N>`, and then upload your tarball. The submitted tarball should contain a directory named `15-441-project-2`, which has the following files that implement all required functionality:

- Makefile: Make sure all the variables and paths are set correctly such that your program compiles in the hand-in directory. Running `make test` should run your testing code.
- All of your source code files and test files. (files ending in .c, .h, etc. only, no .o files and no executables)
- readme.txt: File containing a thorough description of your design and implementation. If you use any additional packet headers, please document them here.
- tests.txt: File containing documentation of your test cases and any known issues you have.
- submit.pcap: Your PCAP submission file from running the functionality code in server.c and client.c from the starter code (for a larger file transfer).
- graph.pdf: Your graph of the currently unacked packets in flight vs time computed from submit.pcap.

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`

[4] TCConfig: `https://github.com/thombashi/tcconfig`

[5] tshark: `https://www.wireshark.org/docs/man-pages/tshark.html`

[6] Creating a wireshark dissector in Lua: `https://mika-s.github.io/wireshark/lua/dissector/2017/11/04/creating-a-wireshark-dissector-in-lua-1.html`

[7] tcpdump: `https://linux.die.net/man/8/tcpdump`

[8] Wireshark plugin folder: `https://www.wireshark.org/docs/wsug_html_chunked/ChPluginFolders.html`

[9] Vagrant: `https://www.vagrantup.com/intro/getting-started/index.html`

[10] VirutalBox: `https://www.virtualbox.org/`

[11] pytest: `https://docs.pytest.org/en/latest/`