15-441/641: Computer Networks
Project 2: TCP in The Wild

TAs: Kartik Chitturi <kchittur@andrew.cmu.edu>
Ines Potier <ipotier@andrew.cmu.edu>

October 14, 2019

1 Introduction

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), or 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. In the first two checkpoints of this project, you will focus on implementing a transport algorithm that is very similar to TCP Reno.

However, Reno and NewReno are just two 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 on the Internet. 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 the final checkpoint, you will get creative and design your own, new congestion control algorithm for long file transfers on the Internet.

With this project, you will gain experience in:

- ... building more programs in C, and writing more programs which are compatible with standards for inter-operability.
- ... reasoning about designing end-to-end systems when the underlying network is fundamentally unreliable and disorderly.
- ... analyzing a program for performance and fairness, designing ways to improve it, and testing those improvements.

To guide your development process, we have divided the project into three checkpoints. Unlike P1, the grading and checkpoints for both 441 and 641 are the same.

CP	**Goal**	**% P1 Grade**		**Deadline**	
		441	641	441	641
1	Implement the three-way handshake and connection shutdown	33%	33%	Oct 18, 2019	Oct 18, 2019
2	Implement flow control and Reno-style congestion control.	33%	33%	Oct 30, 2019	Oct 30, 2019
3	Design, implement, and evaluate your own congestion control algorithm.	33%	33%	Nov 6, 2019	Nov 6, 2019

At each checkpoint (CP), we will assess your implementation to verify that it supports all of the goals for the current checkpoint and we will rerun some tests from the previous checkpoint(s) as well. This means that if you fix a bug you had in an earlier checkpoint, you will get credit for doing so in a later checkpoint. Of course, if you break a feature that was originally implemented correctly, you will lose some points, so you should return your tests from earlier checkpoints before you submit a later checkpoint.

2 CMU-TCP

TCP is a network layer protocol that enables different devices to communicate. There are a variety of different algorithms for TCP's *congestion controller* such as Reno, New Reno, Cubic, and more. For this project we are focusing on Reno in CP2; in CP3 you will get to design your own congestion control algorithms.

You will build your CMU-TCP using UDP sockets, crafting packets and transmitting them yourself. UDP will not re-transmit lost packets, and UDP has no controls on how fast you transmit: you will have to augment UDP with these features yourself.

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. A very practical guide to implementing TCP is found in the textbook in chapters 5.2 and 6.3. **Please read these sections before getting started!**

3 Project specification

3.1 Background

This project will consist of three checkpoints. You will receive starter code that implements Stop-and-Wait transmission, but it will be very slow. Furthermore, it does not have a handshake or a teardown, so if the very first packet is lost (or the very last one), it will not perform correctly. In CP1, you will augment the starter code with a handshake and teardown phase. You will also implement two features to make the Stop-and-Wait transport protocol recover from loss more efficiently. In the second checkpoint, we will get rid of Stop-and-Wait and use Windowed Sending. To choose an appropriate window size, you will need to implement flow control and congestion control. In this checkpoint, you will implement a basic version of Reno as your congestion control algorithm. In the third and final checkpoint, you will design and implement your own algorithm that is faster than your Reno implementation when used on the Internet.

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 checkpoints 2 and 3 you will need to provide a graph showing the number of packets in flight (or unacked packets) vs time. **This graph must demonstrate where your algorithm slows down due to congestion, and how your algorithm speeds up in an uncongested network.** We have provided you with a sample file 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 add handshaking, session termination, and RTT estimation to your implementation. The handshaking and session termination will make sure that even if the first or last packet are lost, the data will be transferred reliably. RTT estimation will make the Stop-and-Wait sender recover from loss more quickly.

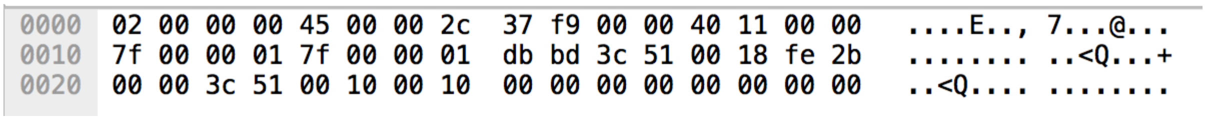
4.1 Starter Code

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: an application using the server side of your transport protocol. We may test your code using a different server program, so do not keep any variables or functions here that are necessary for your protocol to use.
- client.c: an application using the client side of your transport protocol. We may test your code using a different client application, so do not keep any variables or functions here that are necessary for your protocol to use.
- 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.... ........
```

4.2 Checkpoint 1 Tasks

Your server MUST:

1. Implement the TCP Handshake and Teardown - Implement TCP starting handshake and teardown handshake before data transmission starts and ends [1]. This should happen in the constructor and destructor for cmu_socket.

2. Implement improved 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].

4.3 Checkpoint 1 Grading

The breakdown of grading for Checkpoint 1 is below.

Task	Weight	Subcriteria
Format	10%	*Assigned by human grader:* • Correct turnin – Makefile, compilation, properly tagged repo, TA's don't have to edit or search for files (10%) • *Code style and commenting will be assigned in CP3.*
Initiator Handshake	20%	*Assigned in AutoLab:* • Establishes connection with 'good' listener (10%) • Establishes connection even when packets are lost (5%) • Rejects invalid packets from misbehaving listeners (2.5%) • Does not crash (2.5%).
Listener Handshake	20%	*Assigned in AutoLab:* • Establishes connection with 'good' initiator (10%) • Establishes connection even when packets are lost (5%) • Rejects invalid packets from misbehaving initiators (2.5%) • Does not crash. (2.5%)
Teardown	20%	*Assigned in AutoLab:* • Closes connection correctly with 'good' other endpoint (12.5%). • Retransmits FIN when FIN or FIN/ACK is lost (5%). • Does not crash (2.5%)
RTT Estimation	20%	*Assigned in AutoLab:* • Does not re-transmit slower than 3 RTTs (10%) • Does not re-transmit faster than one RTT (5%). • Handles multiple retransmissions if the same packet is lost more than once (5%)
File Transfer	10%	*Assigned in AutoLab:* • Correctly transmits a file end to end, despite loss. (10%)

5 Checkpoint 2

Once you have implemented the basics, you can add windowing. To set your window size will require Flow Control a Congestion Control Algorithm (CCA). 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 graphs from real connections that your TCP Reno implementation uses Additive Increase under normal operation, and Multiplicative Decrease under loss.

5.1 Checkpoint 2 Tasks

Your implementation MUST:

1. Implement Windowing: Extend your CP1 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 [2]. You do not need to implement Nagle's algortihm.

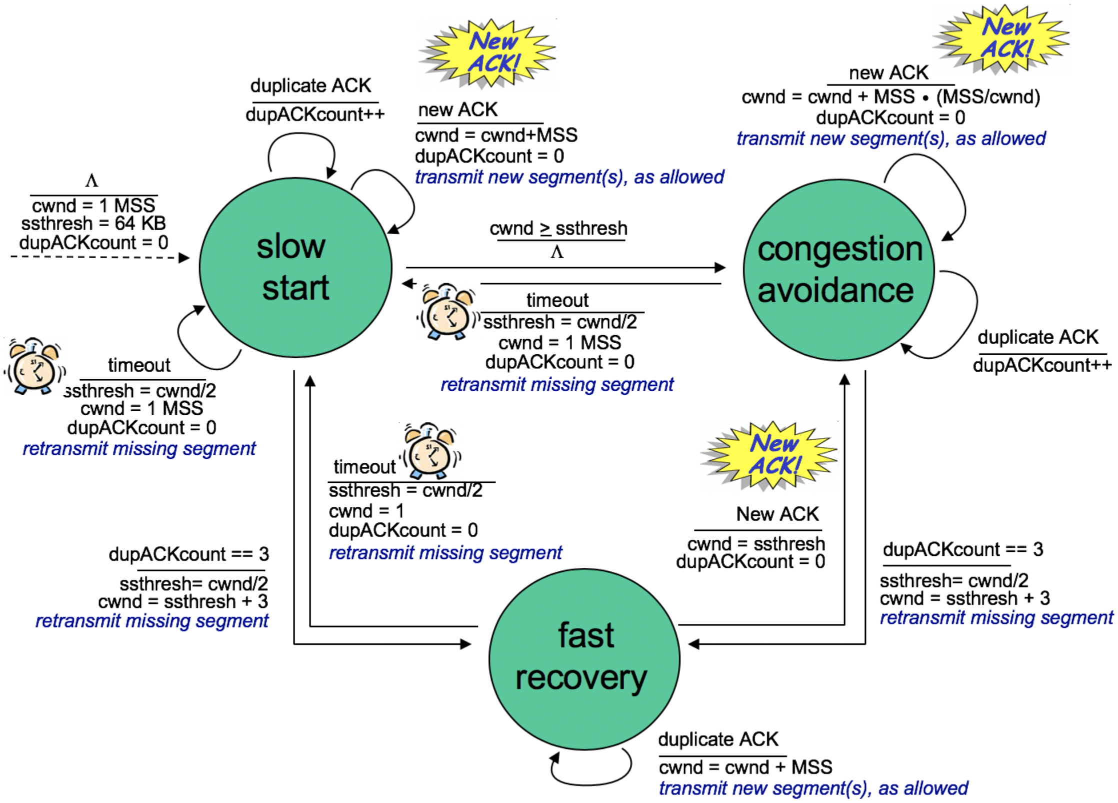
2. Implement duplicate ACK Retransmission - Another reason loss recovery is slow is the starter code relies on timeouts to detect packet loss. One way to recover more quickly is to retransmit whenever you see triple duplicate ACKs. Implement retransmission on the receipt of 3 duplicate ACKs.

3. Implement Flow Control: Update your code to use a the receiver's AdvertisedWindow as your maximum window size; this field is contained in the CMUTCP Header. You will ned to update your receiver to update the AdvertisedWindow as it receives data.

4. Implement Congestion Control: you will need to add a new parameter, the congestion window: `cwnd`. The size of your sending window should now be the minimum of `cwnd` and the advertised window. You should additionally maintain that the total amount of data buffered for the application (unread data, both ordered and unordered bytes) should be less than `MAX_NETWORK_BUFFER`.

Make sure to test your new features with many different network settings using `tcconfig`. You should transmit a large file again using your TCP implementation, like you did in Checkpoint 1, set the bandwidth small in relation to the size of your file (ex: transferring 100Mb file, 1Mbps bandwidth) and add packet loss (ex: 5%) in order to see the TCP sawtooth pattern.

For your info, Here is a copy of the full state machine. Section 6.3 of the textbook is also very helpful!



Here are the values from `grading.h` you must use in your code for this checkpoint. We will test your code (and you should too!) by changing these values. All of these values are in bytes.

1. WINDOW_INITIAL_WINDOW_SIZE: Initial window size for slow start. In slow start, you should initially set `cwnd` = WINDOW_INITIAL_WINDOW_SIZE.

2. WINDOW_INITIAL_SSTHRESH: `ssthresh` value for congestion control. In slow, start, you should initially set `ssthresh` = WINDOW_INITIAL_SSTHRESH.

3. MAX_LEN: Max packet length of any packet - including header. This value will not change and will always remain fixed.

4. MAX_NETWORK_BUFFER: Maximum number of bytes that the TCP implementation can hold/buffer for the application. (This includes unread, ordered and unordered bytes received on the network, and received by the application). Thus, the size of your `sending_buf` should be set to MAX_NETWORK_BUFFER and the size of `received_buf` should be set to MAX_NETWORK_BUFFER.

5.2 Checkpoint 2 Grading

The breakdown of grading for Checkpoint 2 is below.

Task	Weight	Subcriteria
Format	5%	*Assigned by human grader:* • Correct turnin – Makefile, compilation, properly tagged repo, TA's don't have to edit or search for files (5%) • *Code style and commenting will be assigned in CP3.*
Basics	25%	*Assigned in AutoLab:* • File can be successfully transferred, with and without loss (15%) • Handshake and teardown still execute properly (5%) • Does not crash (5%)
Windowing & Flow Control	40%	*Assigned in AutoLab:* • Transmits multiple packets in flight at once (10%) • Transmission never exceeds receiver's AdvertisedWindow (10%) • Retransmissions are performed after three duplicate ACKs (10%) • Window reaches as high as BDP or higher in long-lived connections with a maximum-sized AdvertisedWindow (10%).
Congestion Control	30%	*Assigned in AutoLab:* • `graph.pdf` Illustrates slow start, additive increase, and multiplicative decrease within a connection (10%) • Sender uses multiplicative decrease until reaching ssthresh or loss (5%). • ACKs received without loss increase window linearly (5%). • Window halves on triple duplicate ACKs (5%). • Sender returns to slow start upon timeout loss (5%).

6 Checkpoint 3

In this section, you will implement your own congestion control algorithm. More info coming soon.

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. You can log into a VM by using `vagrant up`, then `vagrant ssh <NAME>`. 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 tests.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. You can set additional tc variables by using `tcset`. 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 capture.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 capture.pcap
```

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. 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)
- graph.pdf: (CP2 and CP3 only) Your graph of the currently unacked packets in flight vs time computed from a packet capture of a large file transfer using your implementation.

There are a few requirements in order for your code to work well with the autograder.

- Your top level directory must be named *15-441-project-2*
- *15-441-project-2* must contain the following files and directories
  - *Makefile*: **Please make sure that you compile you code with the gcc flag "-fPIC".**
  - *readme.txt*
  - *tests.txt*: A description of all your tests
  - *src/*: For all your source files
  - *inc/*: For all your header files
  - *build/*: Make should place all object files within this directory
  - *tests/*: For all your testing scripts and files
  - *utils/*: Utility files to help you capture/analyze packets
- Your submission should not contain any files starting with the word "grader"

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] VirtualBox: https://www.virtualbox.org/

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

[12] TCP Congestion Control: https://intronetworks.cs.luc.edu/current/html/reno.html#tcp-reno-and-congestion-management