HKUST ELEC3120: Computer Communication Networks
Project 1: TCP in the Wild

TA: Xiaoyang YAN
Email: xyanaq@connect.ust.hk

Foreword from Prof. MENG: I borrow the course project from CMU's Computer Network course (15-441) – that's why there are a lot of "CMU"s in the starter code. I know that the C programming must be painful for some of you – keep in mind that we are not data structure or algorithm courses. The main part is how to implement the functionality. I'm sure you will learn a lot in this project – how to use basic coding tools, how to debug network issues, how to improve the network performance.

If you do have questions, feel free to reach out to the TA or me at any time. You can even bring your laptop to my office during the office hour so that we can debug together.

Checkpoint 1 due: 11:59 pm on October 27th, 2023
Checkpoint 2 due: 11:59 pm on November 17th, 2023

1 Introduction

TCP is a transport layer protocol that enables processes on different devices to communicate reliably. TCP's beauty stems from its ability to give applications a simple abstraction with strong guarantees, even though the underlying network only provides best-effort delivery. Best effort delivery means that packets can be lost, duplicated, reordered, and that there is no guarantee that the bandwidth available will remain the same (due to changes in the network itself or interference from other traffic). TCP on the other hand assures applications that their data will be delivered in order, uncorrupted, and that transmissions will be paced to avoid overwhelming the network (congestion control) or the application on the other end of the connection (flow control). It also aims to be fair, trying to give an equal share of bandwidth to multiple senders that share the same link.

In this project, your goal is to implement the underlying mechanisms that are the basis of TCP's strong guarantees. You will focus on implementing an algorithm that is very similar to TCP Reno. However, TCP has evolved since Reno, which is now just one of many congestion control algorithms (CCAs) in use. Companies use different CCAs depending on their context. For example, one version may be appropriate inside of datacenters while another version may be preferable for serving web content on the Internet. For web content, the most common algorithm—and the default in Linux—is called Cubic. Akamai, the largest content distribution network in the world, uses a proprietary TCP called FastTCP. In Checkpoint 2, you will have the option to implement your own CCA for bonus points.

With this project, you will gain experience in:

- Building non-trivial computer systems in a low-level language (C).
- Building interoperable programs for use on the Internet.
- Designing end-to-end systems when the underlying network is fundamentally unreliable.
- Analyzing an algorithm's performance, designing ways to improve these metrics, and testing those improvements.

This project is divided in two checkpoints. In the first, you will implement the TCP three-way handshake, RTT estimation, as well as TCP's sliding window algorithm. In the second, you will implement fast recovery, flow control, congestion control, and, optionally, your own congestion control algorithm.

2 Logistics

Starter code. We reserve the right to change the starter code as the project progresses to fix bugs and to introduce new features that will help you debug your code (we commit to making the changes as transparent as possible so as to not break your working implementation). You are responsible for checking Canvas to stay up-to-date on these changes. We will assume that all students in the class will read and be aware of any information posted to Canvas.

Reusing code. You are welcome to clone the project repository to stay up-to-date with changes (e.g., bug-fixes), but you are NOT allowed to push your codes to any public hosting platforms (e.g., GitHub). For example, if you're using GitHub to manage your project, make sure the repository is **private**. You agree that you will not publicize your solution in the future either. We take academic integrity violations *very* seriously, and you will be held just as liable for sharing code with someone as you will be for using it.

Testing. The starter code includes a suite of public test-cases to help you get started, but the autograder may use private test-cases to grade your solution (you will see the final score at the time of submission, though). Please come up with your own tests; it'll save you a lot of debugging pain in the long term!

3 CMU-TCP

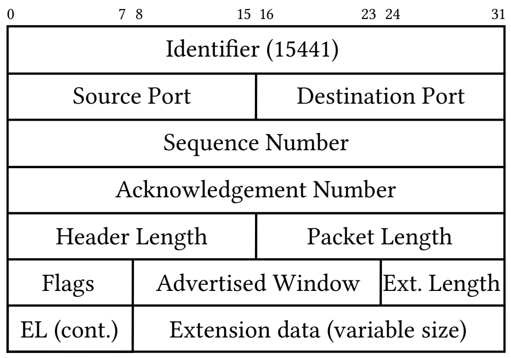


Figure 1: CMU-TCP header format.

You will implement CMU-TCP, a simple reliable transport protocol. Different from a traditional TCP implementation, CMU-TCP segments are carried inside UDP payloads, this allows you to implement CMU-TCP from user space using UDP sockets and benefit from UDP's existing multiplexing/demultiplexing capabilities. As in traditional TCP, CMU-TCP connects two end points, an initiator and a listener. While a CMU-TCP header shares many fields with traditional TCP, its structure is different. Figure 1 shows the structure of a CMU-TCP header, with each field described next:

- *Identifier*: All CMU-TCP packets start with an identifier that MUST be set to `15441`.
- *Source Port*: Identifies the source port (matches the UDP source port in the header below).
- *Destination Port*: Identifies the destination port (matches the UDP destination port in the header below).
- *Sequence Number*: The sequence number of the first byte in this segment, unless if `SYN` is set. If `SYN` is set, this number is the initial sequence number (ISN). The first data sent, will have ISN+1 as the sequence number.
- *Acknowledgment Number*: Valid when the `ACK` flag is set. Identifies the next sequence number that the sender expects.
- *Header Length*: Length of the CMU-TCP header in bytes (25 bytes with no extension data).
- *Packet Length*: Length of the CMU-TCP packet (payload + header) in bytes.
- *Flags*: Only 3 out of 8 bits are used (from left to right starting at 0):
  - `SYN` (bit 4): Synchronize sequence numbers. Used by both sides at the beginning of the connection.
  - `ACK` (bit 5): Acknowledgment number is significant.
  - `FIN` (bit 6): Sender has no more data to send.

- *Advertised Window*: Number of bytes that the sender can accept in its window.
- *Extension Length*: Length of the "Extension Data" field in bytes.
- *Extension Data*: Arbitrary extension data. (You may use this when developing your own CCA.)

As a starting point, you will receive starter code that implements CMU-TCP using stop-and-wait transmission. However, as we learned in class, stop-and-wait is very inefficient because it can only transmit a single packet per RTT. The starter code also lacks connection setup and termination, congestion control and flow control. In **Checkpoint 1**, you will augment the starter code with a handshake and window, so that multiple packets can be sent before the next ACK arrives. For Checkpoint 1, this window will be fixed and set at compile time. You will also implement RTT estimation to be able to recover from loss more efficiently. Optionally, you can also implement connection termination for bonus points. In **Checkpoint 2**, you will improve on the performance of your implementation by automatically choosing an appropriate window size. You will need to implement flow control and congestion control, using a basic version of TCP Reno as the congestion control algorithm. Optionally, you can also implement your own CCA. As in Project 1, both checkpoints also require that you run some experiments to test a hypothesis. You should make sure to finish your implementation early so that you have time to run the experiments.

You will build CMU-TCP using UDP sockets, crafting packets and transmitting them yourself. UDP will not re-transmit lost packets, and UDP has no control over when a packet is sent (i.e., the sending throughput). You must augment UDP with these features yourself. As both sides (initiator and listener) can send and receive, you will need to track a lot of data and information. It is important to write down everything that each side knows while working on your implementation and to utilize interfaces to keep your code modular and reusable. A practical guide to implementing TCP is found in the textbook in Sections 5.2 [1] and 6.3 [2]. Please read these textbook sections before starting the project! If in doubt or faced with ambiguity, prefer explanations from the textbook.

3.1 Starter Code and Implementation Details

The starter code is hosted on BitBucket: `https://bitbucket.org/computer-networks/project1_elec3120/src/master/`. See `README.md` for a description of the files in the starter code and instructions on how to run it.

The API exposed to applications is defined in `cmu_tcp.h` and consists of four core functions whose signatures should not be changed (`cmu_socket`, `cmu_close`, `cmu_read`, and `cmu_write`). An application requests a new CMU-TCP socket by calling the `cmu_socket` function. The application specifies whether it is a listener (e.g., a server) or an initiator (e.g., a client). For listener sockets, the application specifies the port to listen on. For initiator sockets, the application specifies the IP and port number of the *listener* socket. After initializing the socket, the application can use `cmu_write` to transmit data and `cmu_read` to receive data. `cmu_read` also accepts flags that define whether it should block when there is no data to be read (blocking) or return immediately with no data (non-blocking). When the application is done sending data, it is responsible for calling `cmu_close` to terminate the socket.

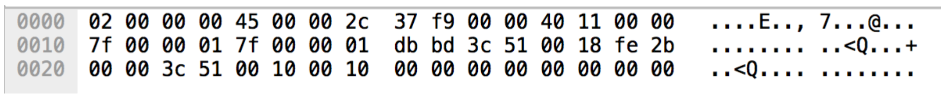
While the functions defined in the API are implemented in `cmu_tcp.c`, which runs on the same thread as the application, most of CMU-TCP's core logic actually runs in a separate backend thread. This is important as TCP must be able to work *independently* from the application (i.e., receiving, acknowledging and retransmitting packets). As such, the logic implemented in `cmu_tcp.c` is simply responsible for communicating with the backend thread and most of your implementation should be done at `backend.c`.

The starter code also includes two sample applications (`server.c` and `client.c`) that use CMU-TCP to communicate. You can use and adapt these applications to test our code. We also provide simple automatic tests written in Python that you can use as a basis to implement more refined tests.

We will use `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 that you are not hard-coding something elsewhere in your code. Namely, we will be changing the packet length and the initial window variables. Additionally, for CP2, we will be monitoring and reviewing your congestion window to ensure that it is consistent with TCP Reno or your own congestion control algorithm. This will be manually graded after the due date.

All multi-byte integer fields must be transmitted in network byte order. `ntoh()`, `hton()`, and similar functions are important. We also provide many helper functions to get and set different packet fields, automatically dealing with endianness. Refer to `cmu_packet.h` for their signatures and `cmu_packet.c` for their implementation. You must also be careful when comparing sequence number as they might wrap around. When comparing sequence numbers you may use one of the provided helper functions (`before`, `after`, `between`) in `cmu_packet.h`. You should also make sure that `IDENTIFIER` is always set to `15441`, even if you are taking 15-641 (the scripts rely on this). You are not allowed to change the structure of the header, with the exception of the extension data which you may want to modify when implementing your own CCA in CP2. Additionally, packet length (`plen`) cannot exceed `MAX_LEN` to prevent packets from being broken into parts.

You can verify that your headers are sent correctly using Wireshark or `tcpdump` (Section 5.3), which allow you to inspect packet data, including the full Ethernet frame. When viewing a packet, you should see something similar to the image below; in this case the payload starts at `0x0020`. The `IDENTIFIER` (`15441`) shows up in hexadecimal 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 Evaluation

This project is broken down in two checkpoints to help you maintain pace with required work. Your grade for each checkpoint will be based on two components: an *implementation* component, which is determined by manual evaluation as well as a suite of automated test cases on Gradescope; and an *experimental* component, where you will hypothesize about, experiment with, and answer questions relating to your CMU-TCP implementation. Thus, for each checkpoint, you need to submit both the source code and a PDF containing your answers. Note that both the implementation and the experimental component have equal weight. You should start your implementation early to also have time to run experiments and write the report.

4.1 Checkpoint 1

In this checkpoint, you will implement windowing so that multiple packets can be 'in the pipe' simultaneously. To do so, you also need to implement the TCP three-way handshake. This ensures that even if the first packet is lost, the data will be transferred reliably. In addition, you will implement RTT estimation to improve timeout selection, which allows the sender to recover from losses more quickly. Optionally, you can also implement connection teardown for bonus points.

4.1.1 Implementation Component

Your implementation should:

1. Perform the TCP three-way handshake [3].
2. Initialize sequence numbers randomly and synchronize them during the handshake.
3. Use 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 using either the Jacobson/Karels or Karn/Partridge algorithms [5].
4. Transmit multiple packets before an ACK by using a fixed window size (you must set the window size to `WINDOW_INITIAL_WINDOW_SIZE` bytes) [4].
5. [Optional] Perform connection teardown [3].

You should use the provided script (`prepare_submission.sh`) to create an archive with your submission (`handin.tar.gz`). Make sure that all the files that are relevant to the submission are committed before you run the script.[1] You can then submit this archive to Gradescope. Your implementation will be automatically graded using the following rubric:

[1]You may also consider tagging [13] your commit once you are done with the checkpoint, so that you can easily compare your CP1 solution with your CP2 solution.

Category	Weight	Criteria (points)
Initiator handshake	20%	*Autograded:* • Sends the first SYN packet properly • Rejects malformed SYN-ACK packet from listener • Receives data packets properly after handshake finishes
Listener handshake	30%	*Autograded:* • Does not respond to invalid SYN packets • Responds to valid SYN packets with a valid SYN-ACK packet • Retransmits SYN-ACK packets on loss • Properly handles invalid ACK packets after a SYN-ACK • Receives data packets properly after handshake finishes
Reliability	30%	*Autograded:* • Retransmits data packets on timeout • Retransmits data packets within 1–3 estimated RTTs • Transfers a file reliably under lossless conditions • Transfers a file reliably under lossy conditions
Windowing, Sequence numbers	20%	*Autograded:* • Sender transmits multiple data packets at a time • Initiator correctly synchronizes sequence number • Listener correctly synchronizes sequence number • Initiator initializes sequence number randomly • Listener initializes sequence number randomly
Teardown (Optional)	+5%	*Autograded:* • Sends FIN packet correctly • Sends a valid ACK packet after receiving a FIN-ACK • Retransmits FIN packet on timeout • One side can still send data after the other side starts teardown

4.1.2 Experimental Component

For the experimental component, you will measure the 'flow completion time' (or 'FCT': how long it takes to finish transferring a file) under different loss rates (0%, 0.1% and 0.01%). You should fix the base rate at 10 Mbps and set the delay to 20 ms. Apply these settings to both the server and the client. For each one of the different loss rates, you will transfer files of a range of sizes: 512 bytes, 4 KiB, 32 KiB, 256 KiB, and 2 MiB. Hence, you will measure 15 total scenarios: five different file sizes $\times$ three loss rates.

Please create a PDF with your experiment report and submit it to Gradescope with the following information.

Hypothesis. Given the implementation in the starting code (baseline) and your complete CP1 implementation, before running any experiment, we ask that you hypothesize about your expected results.

(1) Derive an equation to predict how long (in number of RTTs) it will take for the initial implementation to transfer a file of size f with loss rate p. Recall that the throughput is fixed (10 Mbps) and the RTT is also fixed at 40 ms (20 ms delay on each end).

(2) Derive an equation to predict how long (in number of RTTs) it will take for your CP1 implementation to transfer a file of size f with loss rate p. (Don't forget the handshake and, if you implemented it, teardown time!)

(3) Using these two equations, predict in which scenarios your CP1 implementation would outperform the baseline.

Similarly, predict when the baseline implementation would outperform your CP1 implementation. Provide a chart or table with an entry for every testing scenario predicting how many RTTs it will take to complete the file transfer using either the baseline or your new implementation.

Experiment. In the Vagrant environment you used to develop your code, measure the time it takes to transfer different files with increasing size: 512 bytes, 4 KiB, 32 KiB, 256 KiB, and 2 MiB; under different loss ratios: 0%, 0.1% and 0.01%, all with a base rate of 10 Mbps and an RTT of 40 ms (20 ms delay configured on each side). Refer to Section 5.2 for instructions on how to artificially set the loss ratio, throughput, and delay using `tcconfig`. Repeat this experiment for both the baseline implementation as well as your CP1 implementation. For the CP1 experiments, the transfer duration should include the handshake and, if you implemented it, the teardown time.

Create a separate bar plot for each loss ratio (3 plots). In each plot indicate the transfer size in the x axis and the transfer duration in milliseconds in the y axis. There should be four bars for each transfer size: one with transfer duration for the baseline implementation, one with the predicted duration given your equation from (1), one the transfer duration for your CP1 implementation, and one with the predicted duration given your equation from (2). For your measured results, each bar should indicate the median of 10 runs (i.e., you should repeat the same transfer 10 times under the same conditions and report the median). You should also report the standard deviation of each data point through error bars. Make sure to label and include the unit in both axis and include a legend to identify which bar is which.

Inference. Based on the results of your experiments, we ask that you answer the following questions:

- How closely does your equation (1) match the real results for the baseline implementation? Provide some ideas for what might be the source of the gap between the two, and how you might improve your equation.
- How closely does your equation (2) match the real results for the baseline implementation? Provide some ideas for what might be the source of the gap between the two, and how you might improve your equation.
- How accurate were your predictions as to whether or not the baseline implementation or your CP1 implementation would perform better for a given scenario? If your predictions were ever in error, please hypothesize as to why your predictions were wrong.

5 Resources

This section describes some of the recommended tools for developing and testing CMU-TCP.

5.1 Development Environment

For this project, you can do all of your development on your own machine using containers. You are also welcomed to keep using Azure VM for your development if that is more convenient. You should install Docker [12] and Vagrant [11] on the machine you plan to develop on. We have provided a `Vagrantfile` specifying two containers, a client and a server. The Vagrantfile is configured to map the repository's directory to the containers' `/vagrant/` directory. This means that any change you make to this directory, either from your computer or inside the containers will be propagated everywhere else. You can build and launch the containers by running `vagrant up --provider=docker` from the `cmu-tcp` directory. Once the containers finish starting up, you can log into any of the two containers using `vagrant ssh {client | server}`. The server and client are connected via a virtual network and have IP addresses `10.0.1.1` and `10.0.1.2`, respectively. But note that your code should work even if these IP addresses change.

The following sections describe the tools that are already installed on the containers to help you test your code. You can also install any additional tools you need.

5.2 Control network characteristics with `tcconfig`

`tcconfig` [6] is installed on the containers to let you artificially control the characteristics of the virtual network that interconnects the containers. The initial default settings on the containers are a 20 ms delay on both containers (so the total RTT is 40 ms), and 100 Mbps bidirectional bandwidth with no loss. Running `tcshow $IFNAME` on the containers will show you these settings. You can set additional `tc` parameters by using `tcset`, for example `sudo tcset $IFNAME --rate 10Mbps --delay 20ms --loss 0.1% --overwrite` will set the rate to 10 Mbps, single-way delay to 20 ms, and loss probability to 0.1%.[2]

Refer to the references for more information on how to use `tcconfig` to simulate different network characteristics that will be useful for testing your code, including packet loss, reordering, and corruption.

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

`tcpdump` [7] and Wireshark (and its equivalent command line program: `tshark` [8]) are installed on the containers to allow you to capture packets sent between the containers and analyze them later. We provide the following files in the directory `project-1_elec3120/utils/` to help with packet analysis (feel free to modify these if you want):

- `capture_packets.sh`: This is a simple program showing how to 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. See Section 5.4 for an example.
- `tcp.lua`: This ia a Lua plugin so that Wireshark can dissect our custom CMU-TCP packet format [9]. `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 should add this file to Wireshark's plugin folder [10].

5.4 Analyzing 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 large file, capturing the packets, and plotting the congestion window over time. You can use the utilities described in Section 5.3 to create `capture.pcap` and `graph.pdf` by running the following commands:

(In the server container) Start `tcpdump` and the server:

[2]Note that `$IFNAME` is a variable with the name of the interface that we will use with CMU-TCP, it is automatically initialized by the Vagrant scripts that we provide. When running on Azure, make sure to set it the the name of the interface that you want to use.

```
vagrant@server:/vagrant/project-1_elec3120$ make
vagrant@server:/vagrant/project-1_elec3120$ ./utils/capture_packets.sh start capture.pcap
vagrant@server:/vagrant/project-1_elec3120$ ./server
```

(In the client container) Start the client:

```
vagrant@client:/vagrant/project-1_elec3120$ ./client
```

(In the server container) When the transfer finishes, stop the packet capture and generate a graph:

```
vagrant@server:/vagrant/project-1_elec3120$ ./utils/capture_packets.sh stop capture.pcap
vagrant@server:/vagrant/project-1_elec3120$ ./gen_graph.py
```

5.5 Running tests with `pytest`

There are many ways that you can write tests for your code. To help get you started, `pytest` [14] is installed on the containers and we provide some examples of 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. However, be sure to update the Makefile so that `make test` must still runs your tests. As in Project 1, you should also use standard C debugging tools including `gdb` and `Valgrind`, which are also installed on the containers.

5.6 Automatic code formatting and analysis

We provided a `pre-commit` [15] configuration file that you can use to automatically format and statically check your code whenever you run `make format` inside one of the containers. You can check `.pre-commit-config.yaml` to see all the hooks that we run. You can optionally also install it in your development machine. This will ensure that it always runs all the hooks whenever you create a new commit. If you want to install `pre-commit` in your local machine, run `pip3 install pre-commit`. Then run `pre-commit install` in the root directory of the `cmu-tcp` repository to install the pre-commit hooks.

References

[1] Textbook Section 5.2 Reliable Byte Stream (TCP):
`https://book.systemsapproach.org/e2e/tcp.html` 3

[2] Textbook Section 6.3 TCP Congestion Control:
`https://book.systemsapproach.org/congestion/tcpcc.html` 3

[3] TCP connection establishment and termination:
`https://book.systemsapproach.org/e2e/tcp.html#connection-establishment-and-termination` 1, 5

[4] TCP sliding window:
`https://book.systemsapproach.org/direct/reliable.html#sliding-window`
`https://book.systemsapproach.org/e2e/tcp.html#sliding-window-revisited` 4

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

[6] `tcconfig`: `https://github.com/thombashi/tcconfig` 5.2

[7] `tcpdump`: `https://linux.die.net/man/8/tcpdump` 5.3

[8] `tshark`: `https://www.wireshark.org/docs/man-pages/tshark.html` 5.3

[9] 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` 5.3

[10] Wireshark plugin folder:
`https://www.wireshark.org/docs/wsug_html_chunked/ChPluginFolders.html` 5.3

[11] Vagrant: `https://www.vagrantup.com/intro/getting-started/index.html` 5.1

[12] Docker Desktop: `https://docs.docker.com/desktop/` 5.1

[13] Git Basics – Tagging: `https://git-scm.com/book/en/v2/Git-Basics-Tagging` 1

[14] `pytest`: `https://docs.pytest.org/en/latest/` 5.5

[15] `pre-commit`: `https://pre-commit.com/` 5.6

[16] James Kurose and Keith Ross. 2016. *Computer networking: A top-down approach* (7th ed.). Pearson, Upper Saddle River, NJ.