Project TCP

Due: 29 April 2022, 11:59pm

1 Introduction

In this project, you will implement a simple but RFC-compliant form of TCP on top of IP from your last assignment. You will build a working transport-layer and create your own socket API, similar to the one you have been using to interact with sockets in all of your previous assignments.

Students report that this assignment is very challenging, but also very rewarding. When you are done here, you will really understand TCP, and the challenges involved in building a real-world network protocol.

You have roughly 4 weeks to complete this assignment, with two intermediate milestone deadlines for you to implement part of the functionality and check in with the course staff. We *strongly* recommend that you **start early** and take full advantage of this time, as you will need it!

During this time, we remind you that we are here to help at *all* stages of the process! If you have questions at the design or implementation stage, we implore you to ask for help early so that we can make sure you are on the right track.

2 The Pieces

In this assignment you will use the library you wrote for IP as the underlying network.

Your TCP implementation will have four major components: the state machine that implements connection setup and teardown, the sliding window protocol that determines what data you are allowed to send and receive at any point, the API to your sockets layer, and a driver program that will allow you (and us) to test your code.

Additionally, capstone students must also implement a congestion control algorithm and document its performance.

As a guide for your implementation, we will refer you to various IETF RFCs that provide de-facto standard for TCP implementations. **For more details on how to find information on specific components inside the RFCs, see Section 5.3.** You are also encouraged to consult our class notes and textbooks, as well as any other online resources—so long as the code you write is your own work.

2.1 State Machine

You have to implement a basic TCP state machine that allows state transitions in your TCP. You can use this diagram [1] to help orient yourself.

[1] `http://ttcplinux.sourceforge.net/documents/one/tcpstate/tcpstate.html`

The state machine is not as complicated as it may seem at first. To begin, we recommend that you start coding by just using the diagram and getting connections to set up and close under ideal conditions. In general, you expected to follow the state machine described by RFC793[2] and RFC2525[3] , *except* for the parts of referring to:

- `PSH` flags
- Error-related edge cases involving `RST` packets
- Urgent data
- Any TCP options
- Precedence
- Security considerations

Once you have the rest of your implementation working, there are some edge cases to consider: for example, what happens when, after a call to `connect`, you've sent a SYN, but you receive a packet that has an incorrect ACK in it? Once your basic state diagram is working, we recommend that you look at the RFC for answers to questions such as these. In particular, pages 54 and on contain info on exactly what you should do in such scenarios.

2.2 Sliding Window Protocol

Your sliding window protocol controls how you send and receive data—this is the "heart" of your TCP stack.

Be sure that you can accept out-of-order packets. That is, a packet's sequence number doesn't have to be exactly the sequence number of the start of the window. It can be fully contained within the window, somewhere in the middle. The easiest way to handle such packets is to place them on a queue of potentially valid packets, and then deal with them once the window has caught up to the beginning of that segment's sequence number.

You are not required to implement slow start, but you should detect dropped or un'acked packets and adjust your flow accordingly.

You should strictly adhere to the flow control window as specified in the RFC, e.g. do not send packets outside of your window, etc. Similarly, you should implement zero window probing to ensure your sender can recover when the receiver's window is full. Overall, your goal is to ensure reliability—all data must get to its destination in order, uncorrupted.

As you implement your protocol, keep in mind how sliding windows will interact with the rest of TCP. For example, a call to CLOSE (`v_shutdown(s, 1)` in our API) only closes data flow in one direction. Because data will still be flowing in the other direction, the closed side will need to send acknowledgments and window updates until both sides have closed.

[2] `https://datatracker.ietf.org/doc/html/rfc793`
[3] `https://datatracker.ietf.org/doc/html/rfc2525`

2.3 Socket API

You will create an interface to your TCP stack by implementing a socket API, similar to the socket API in the language you are using. While the API will differ slightly for each language, the required elements described here should be essentially the same.

In C, you should create new connections as virtual sockets using your own socket table and virtual file descriptors to allow connecting and listening, reading and writing into buffers, etc. In Go, you should provide types similar to net.TCPConn and net.TCPListener.

An independent thread in your program should be able to use this your socket API in a similar way to how you would use the normal API in your language. These functions, on error, should return appropriate error codes (such as negative values in C, error values in Go, etc.). For C, you should use standard error codes (such as `EBADF`).

The functionality you need in the C socket API is shown below. Except for `v_socket` and `v_bind`, these functions (or some reasonably equivalent function, potentially with a different name or arguments) should be part of the API for any language other than C.

If you are using a language other than C, you should implement a socket interface similar to that provided by the language, but you are only required to support enough functionality to provide the basic socket operations described here. For example, in Go, you should implement an interface similar to `net.TCPConn` and `net.TCPListener`, but you need not implement functions like `SetReadDeadline`.

```
/* creates a new socket, binds the socket to an address/port
If addr is nil/0, bind to any available interface
After binding, moves socket into LISTEN state (passive OPEN in the RFC)
returns socket number on success or negative number on failure
Some possible failures: ENOMEM, EADDRINUSE, EADDRNOTAVAIL
(Note that a listening socket is used for "accepting new
connections") */
int v_listen(struct in_addr *addr, uint16_t port);

/* creates a new socket and connects to an address (active OPEN in the RFC)
returns the socket number on success or a negative number on failure
You may choose to implement a blocking connect or non-blocking connect
Some possible failures: EAGAIN, ECONNREFUSED, ENETUNREACH, ETIMEDOUT */
int v_connect(struct in_addr *addr, uint16_t port);

/* accept a requested connection from the listening socket's connection
 queue
returns new socket handle on success or error on failure.
if node is not null, it should fill node with the new connection's address
 accept is REQUIRED to block when there is no awaiting connection
Some possible failures: EBADF, EINVAL, ENOMEM */
int v_accept(int socket, struct in_addr *node);
```

```
/* read on an open socket (RECEIVE in the RFC)
   return num bytes read or negative number on failure or 0 on eof and
   shutdown_read
   nbyte = 0 should return 0 as well
   read is REQUIRED to block when there is no available data
   All reads should return at least one data byte unless failure or eof occurs
   Some possible failures : EBADF, EINVAL */
int v_read(int socket, void *buf, size_t nbyte);

/* write on an open socket (SEND in the RFC)
   return num bytes written or negative number on failure
   nbyte = 0 should return 0 as well
   write is REQUIRED to block until all bytes are in the send buffer
   Some possible failures : EBADF, EINVAL, EPIPE */
int v_write(int socket, const void *buf, size_t nbyte);

/* shutdown an connection. If type is 1, close the writing part of
   the socket (CLOSE call in the RFC. This should send a FIN, etc.)
   If 2 is specified, close the reading part (no equivalent in the RFC;
   v_read calls should return 0, and the window size should not grow any
   more). If 3 is specified, do both. The socket is NOT invalidated.
   returns 0 on success, or negative number on failure
   If the writing part is closed, any data not yet ACKed should still be
   retransmitted.
   Some possible failures : EBAF, EINVAL, ENOTCONN */
int v_shutdown(int socket, int type);

/* Invalidate this socket, making the underlying connection inaccessible to
   ANY of these API functions. If the writing part of the socket has not been
   shutdown yet, then do so. The connection shouldn't be terminated, though;
   any data not yet ACKed should still be retransmitted.
   Some possible failures : EBADF */
int v_close(int socket);
}
```

2.4 Driver

Your driver should support the following commands ("command/cmd" means that typing both "command" and "cmd" should have the same effect). Note that you do not need to have "up" or "down" functionality for this project (as TCP sockets are rarely well defined with interfaces brought down), but we recommend keeping the code for that.

`h` Print this list of commands.

`li` Print information about each interface, one per line.

`lr` Print information about the route to each known destination, one per line.

`ls` List all sockets, along with the state the TCP connection associated with them is in, and their window sizes (one should be the socket's receiving window size, and the other should be the peer's receiving window size)

`a <port>` Open a socket, bind it to the given port on any interface, and start accepting connections on that port. **Your driver must continue to accept other commands.**

`c <ip> <port>` Attempt to connect to the given IP address, in dot notation, on the given port. Example: `c 10.13.15.24 1056`. This command should return an ID number that is used to refer to the socket for other commands (0, 1, 2, ...).

`s <socket ID> <data>` Send a string on a socket. This should block until write() returns.

`r <socket ID> <numbytes> <y|n>` Try to read data from a given socket. If the last argument is y, then you should block until numbytes is received, or the connection closes. If n, then don't block; return whenever and whatever read() returns. Default is n.

`sd <socket ID> <read|write|both>` `v_shutdown` on the given socket. If read or r is given, close only the reading side. If write or w is given, close only the writing side. If both is given, close both sides. Default is write.

`cl <socket ID>` `v_close` on the given socket.

`sf <filename> <ip> <port>` Connect to the given IP and port, send the entirety of the specified file, and close the connection. **Your driver must continue to accept other commands.**

`rf <filename> <port>` Listen for a connection on the given port. Once established, write everything you can read from the socket to the given file. Once the other side closes the connection, close the connection as well. **Your driver must continue to accept other commands.** Hint: give `/dev/stdout` as the filename to print to the screen.

`q` Quit cleanly, closing all open sockets

2.5 Congestion Control (Capstone only)

Each student taking cs168 for capstone, and choosing the IP/TCP additions route, will be responsible for implementing one of the following congestion control algorithms. Your TCP design should be able to selectively enable and disable any congestion control module that is available, and only 1 congestion control algorithm can be enabled per tcp socket at any given time. If you would like to implement a different congestion control algorithm than the two provided below (since there are many more out there), first seek approval from the TAs.

The algorithms you may choose from are:

- **TCP Tahoe:** Slow Start, Congestion Avoidance, Fast Retransmit
- **TCP Reno:** TCP Tahoe + Fast Recovery

If there are two students both taking cs168 for capstone, they may not share parts of their code for the congestion control algorithm with each other. All code for each congestion control algorithm must be written individually. Your TCP driver must implement the following commands to demonstrate your congestion control algorithm:

`lc` Prints the available congestion control algorith names: *reno*, *tahoe*, ...

`sc <socket ID> <string>` Sets the congestion control algorithm for the given socket. To disable congestion control, use the string: *none*

`*s` You should modify your *sockets* command to also list the congestion control algorithm (if any) the socket is using, and the congestion window size as well.

`*sf` You should modify your *sendfile* to optionally take in a congestion control algorithm, with the options being: *reno*, *tahoe*, ... The default for no argument is *none*.

Lastly, you will be required to provide trace files as well as a summary of how your congestion control algorithm fared against your implementation just using flow control.

3 Implementation

A few notes:

- You must use the TCP packet format, exactly as-is. If you are using C, you should use the header found in `netinet/tcp.h`, although technically, you can use anything, since the TCP packet format is not exposed in the API.
- There are several places in the RFCs that leave room for flexibility in implementation. We extend the same flexibility to your projects, as long as you can justify your design decisions (in a README). A good rule of thumb is to be liberal in what you accept but conservative in what you output. For example, you are not required to implement Nagle's algorithm (which will be covered in class), but you should be able to operate with implementations that do.
- TCP uses a pseudo-header in its checksum calculation. Make sure you understand how TCP checksumming works to ensure interoperability with the TA binary. You may consult online resources as needed [4].

[4] `http://www.tcpipguide.com/free/t_TCPChecksumCalculationandtheTCPPseudoHeader-2.htm`

- You should **not** use arbitrary sleeps in your code. For example, you might have a thread which takes care of all your transmission. You shouldn't have this thread check whether there is something to be sent every 1 ms, because 1 ms is an eternity on a fast LAN connection. Mutexes and Conditions are your friends.
- As in the IP assignment, never send packets greater than the MTU. Even if you implemented fragmentation in your IP, you should assume that fragmentation is not supported.
- You don't have to handle any TCP options. You should ignore any options that you see in incoming packets, but your program shouldn't crash if you encounter them.
- When should v_connect() timeout? A good metric is after 3 re-transmitted SYNs fail to be ACKed. The idea is that if your connection is so faulty that 4 packets get dropped in a row, you wouldn't do very well anyway. How long should you wait in between sending SYNs? You can have a constant multi-second timeout, e.g. 3 seconds. Or, you can start off at 2 seconds, and double the time with each SYN you retransmit.
- The RFC states that a lower bound for your RTO should be 1 second. This is way too long! A common RTT is 350 microseconds for two nodes running on the same computer. Use 1 millisecond as the lower bound, instead. By a similar principle, you do not need to be overzealous in precisely measuring RTT; it is reasonable to tolerate small processing delays (1-10ms).
- Debugging TCP can be very difficult—we strongly recommend using Wireshark to observe your TCP connections. Wireshark has many tools to help analyze TCP connection state, which may prove useful—we will discuss a few of these in class. If you have questions how to use wireshark to accomplish a particular task, please feel free to ask!
- Log as much as you can, and make it possible to filter out what you care about. For example, you may only want to log information related to a specific connection, or you may only want to see logs from TCP, and not IP.

4 Grading

4.1 Milestone I – 5% (by April 15)

Similar to the IP milestone, you will complete this part by scheduling a meeting with the course staff (preferably your mentor TA) on or before **Friday, April 15**.

For this meeting, your implementation should be able to demonstrate the following:

- Establishing new connections by properly following the TCP state diagram under *ideal conditions*. Connection teardown is **NOT** required for this milestone.
- When creating new connections, you should allocate a data structure pertaining to the socket—be prepared to discuss what you need to include in this data structure for the rest of your implementation

- To test establishing new connections, you should implement the **a**, **c**, and (partially) `ls` commands in your TCP driver to listen for, create, and list connections, respectively. For the `ls` command, you need not list the window sizes for the milestone.
- Interoperation with both your implementation and the provided reference implementation as endpoints
- Correct operation even with another node in between the two endpoints. This should already be possible if your IP implementation is implementing forwarding properly—but you should check this at this stage to help rule out any lingering bugs.

In addition, try to consider how you will tackle these problems, which we will discuss:

- What does a ”SYN” packet or a ”FIN” packet do to the receiving socket (in general)?
- What data structures/state variables would you need to represent each TCP socket?
- What types of *events* do you need to consider that would affect each socket?
- How will you implement retransmissions?
- In what circumstances would a socket allocation be deleted? What could be hindering when doing so? Note that the state CLOSED would not be equivalent as being deleted.

4.2 Milestone II – 20% (by April 21)

You should schedule a subsequent milestone checkin with the course staff on or before **Thursday, April 21**

For this meeting, students should have the send and receive commands working over non-lossy links. That is, send and receive should each be utilizing the sliding window and ACKing the data received to progress the window. This also means that sequence numbers, circular buffers, etc. should be in place and working.

Retransmission, connection teardown, packet logging and the ability to send and receive at the same time are not yet required.

4.3 Basic Functionality – 55%

As usual, most of your grade depends on how well your implementation adheres to these specifications. Some key points:

- Properly follow the state diagram.
- Adhere to the flow control window.
- Re-transmit reasonably. Calculate SRTT and RTO.
- Send data **reliably**. Files sent across your network should arrive at the other end *identical* to how they were sent, even if the links in between the two nodes are lossy.

- Follow the RFC in corner cases. You may ignore error-related edge cases that normally require RST packets, as well as any of the exceptions listed in Section 2.1. If you have questions on whether you need to handle a certain edge case, just ask! Overall, we don't want you to be bogged down handling more edge cases than necessary to have a working implementation that can tolerate lossy links.

The idea is that having full basic functionality means that any existing valid TCP implementation should be able to talk with yours and eventually get data across, regardless of how faulty the link is.

4.4 Documentation and Performance – 20%

We want you to understand how your design decisions affect your TCP's behavior. In your README, you should document your major design decisions and your reasoning for using them.

Another part (5%) of this grade will be based on performance. Since we mostly test using a single machine, the performance of any implementation is dependent on CPU speed. To get a baseline for performance, run two reference nodes connected directly to each other with no packet loss and compare the time to send a file of a few megabytes in size (you can also directly measure the throughput in Wireshark). Your implementation should have performance on the same order of magnitude as the reference under the same test conditions. In addition, your implementation should also not perform terribly if the link is slightly faulty—again, compare to the reference implementation for a baseline.

Finally, you should submit a packet capture of a 1 megabyte file transmission between two of your nodes. To do this, run two of your nodes in the ABC network with the lossy node in the middle, configured with a 2% drop rate.

After filtering your packet capture to show only one side of the transmission, you should "annotate" the following items in the capture file:

- The 3-way handshake
- One example segment sent and acknowledged
- One segment that is retransmitted
- Connection teardown

To do this, list the frame numbers for each item in your README with a description. For each annotation, you should evaluate if your implementation is responding appropriately per the specification. If you notice any issues, you should document them accordingly.

An example packet capture will be demonstrated in class before the deadline.

Capstone students will also need to provide packet captures for their congestion control implementation. For these you should run your congestion control algorithm in a drop-free network, and also run it with the faulty node in the middle. Similarly, you should run your algorithm with no other competition, as well as run multiple instances of your algorithm intermixed with simple flow control TCP streams simultaneously. In your write up you should explain the behaviour of your node in all these situations, and try to explain the strengths and weaknesses of your algorithm.

5 Getting Started

Since this project is a continuation of your work from the IP assignment, you should continue development in your IP repository.

5.1 Development Environment

Similar to IP, you should work on this project using the provided container environment. For information on the container environment, see:
`https://cs.brown.edu/courses/csci1680/s22/pages/container_setup/`

Otherwise, you can still work on this assignment on the department machines if you want, but this is not recommended as you will not have access to Wireshark. When submitting your work, please indicate which environment you used in your README.

5.2 Reference Implementations

We have provided a few additional reference binaries for this assignment, which are available here:
`https://github.com/brown-csci1680/ip-tcp-starter/tree/main/tcp_tools`

Please copy these files into your IP repository—there is no need to start a new repository for this assignment.

M1 mac users: Please use the binaries located in the `arm64` directory instead.

5.2.1 Lossy Network Node (`ip_node_lossy`)

The starter repository contains an IP node called `ip_node_lossy` that can be configured to drop a fraction of outgoing packets. This will be useful when testing your retransmission and timeout logic. You can specify the drop rate with the command "lossy". The drop rate should be a value between 0.0 and 1.0, where 1.0 means every packet will be dropped by the node.

5.2.2 TCP Reference Implementation (`ref_tcp_node`)

The starter repository also contains a reference TCP implementation.

We must emphasize that your node **MUST** be able to operate with the reference node, so please test using this node frequently!

In addition, make sure you fix any lingering issues in IP preventing your node from working with the reference IP nodes! If you have questions on how to do this, please contact the course staff.

Note that the reference implementation does not implement congestion control.

5.2.3 IP Reference implementation (by request only)

If you do not feel confident in extending your work from the previous assignment to support TCP, we will provide a reference implementation for IP (available as a C static library) that you can use to implement this assignment.

To request a reference implementation, or if you need help deciding if this is right for your team, please email the TAs list.

5.3 Relevant RFCs

We found that implementing this project required a lot of parsing through the RFCs. To make this less time consuming, below is a list of relevant RFCs and what they contain. This is not necessarily complete, but a lot of what you'll need you can probably find here! If you find anything else that you think belongs in this list, let us know!

Links:

- RFC 793
- RFC 1122
- RFC 5681
- Beej's Guide to Network Programming

5.3.1 Introduction

RFC 793: pp. 4-5, 2.6, 2.7 and 2.10.

5.3.2 State Machine

RFC 793: pp. 27-28(Initial Sequence Number Selection), 3.4, 3.5, 3.8, 3.9 and RFC 1122: section 4.2.2.9, 4.2.2.10

5.3.3 Sliding Window Protocol

RFC 793: section 3.2, 3.3, 3.7, 3.9 and RFC 1122: section 4.2.2.16, 4.2.2.17, 4.2.2.20, 4.2.2.21

5.3.4 Congestion Control

RFC 5681

5.3.5 API

RFC 793: section 3.8; Beej's Guide to Network Programming: Ch. 5

5.3.6 TCP Header format

RFC 793: section 3.1 and RFC 1122: section 4.2.2.3

5.3.7 RTT and Re-transmission Timeout

RFC 793: p. 41 and RFC 1122: section 4.2.3.1

6 Handing In and Interactive Grading

Before each milestone and before the final deadline, once you have completed the requirements for that part of the project, you should commit and push your git repository.

Your mentor TA will arrange to meet with you for each interactive grading session (milestones and final demo) to demonstrate the functionality of your program and grade the majority of it. This meeting will take place at some point shortly after the project deadline.

Between the time you've handed in and the final demo meeting, you can continue to make minor tweaks and bug fixes. However, the version you've handed in should be nearly complete since it could be referenced for portions of the grading.

7 Final Thoughts

Once again, we highly recommend getting started on this assignment soon—don't wait to start until a few days before the deadlines! If you have questions, or need help with your implementation, always feel free ask for help or clarification or debugging. This is a hard assignment, but we are here to help you and provide resources, and we will be most effective at doing so if you ask early!

Although we expect compatibility between your TCP implementation and our own, do not get bogged down in the RFC from the start. It is much more important that you understand how TCP works on an algorithmic/abstract level and design the interface to your buffers from your TCP stack and from the virtual socket layer. For any corner cases or small details, the RFC will be your best friend, and our reference implementation should come in handy. Always feel free to consult the course staff if you have any questions about what you are required to do, or how to handle corner cases. It is **not OK** to just make assumptions as to how things will work, because we will be testing your code for interoperability with the reference node and other groups in the class.