Project 3: TCP over IP over UDP

Due: 11:59 PM, Nov 22, 2016

Contents

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 the transport layer and export a socket API similar to what

you used in Snowcast.

Each year, students report this assignment is an order of magnitude harder than its predecessor (seriously). But when you are done here, you will really understand TCP. We've given you a lot of time for this assignment – use it wisely!

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 pieces — 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 all of us to test your code.

Additionally, students taking this class for a capstone and choosing the IP/TCP additions will have to implement a congestion control algorithm and document its performance.

2.1   State Machine

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

The state machine is not as complicated as it may seem, but you should be sure that your TCP follows all state transitions properly, and doesn't do anything otherwise. For example, you need to send SYNs for connect, and FINs to close. You will be expected to follow RFC793[2] and RFC2525[3] precisely, except for the parts of the RFC that refer to PUSH, RST, urgent data, options, precedence, or security.

You can start coding by just using the diagram and getting connections to set up and close under ideal conditions. However, there are tons of less obvious cases that the diagram doesn't cover – 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

You need to implement the sliding window protocol that is the heart of TCP. Make sure you understand the algorithm before you start coding. Also 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.

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

---

[1]`http://ttcplinux.sourceforge.net/documents/one/tcpstate/tcpstate.html`
[2]`http://www.faqs.org/rfcs/rfc793.html`
[3]`http://www.faqs.org/rfcs/rfc2525.html`

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 should strictly adhere to the flow control window as specified in the RFC, e.g. don't send packets outside of your window, etc. You have to ensure reliability – all data must get to its destination in order, uncorrupted.

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

2.3 API

You must implement an API to your TCP implementation. This layer will use constructs appropriate for whichever language you are using, but the API functions/methods will be essentially the same. In C, you will create a mock sockets layer, using your own table and set of integers to allow connecting and listening, reading and writing into buffers, etc. In Go, you will provide types similar to net.TCPConn and net.TCPListener. In Java, you will provide a type similar to java.net.Socket.

An independent thread in your program should be able to use this interface in almost the exact same way that 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, or thrown exceptions in Java). For C, make sure to use the official 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.

```
/* returns a new, unbound socket.
   on failure, returns a negative value */
int v_socket();

/* binds a socket to an address/port
   If addr is nil/not specified, bind to any available interface
   returns 0 on success or negative number on failure */
int v_bind(int socket, struct in_addr *addr, uint16_t port);

/* moves socket into listen state (passive OPEN in the RFC)
   bind the socket to a random port from 1024 to 65535 that is unused
   if not already bound
   returns 0 on success or negative number on failure */
int v_listen(int socket);

/* connects a socket to an address (active OPEN in the RFC)
   returns 0 on success or a negative number on failure */
int v_connect(int socket, struct in_addr *addr, uint16_t port);

/* accept a requested connection (behave like unix socket's \verb|accept|)
   returns new socket handle on success or negative number on failure */
```

```
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
   Note: you are not expected to implement non-blocking reads.
   All reads should return at least one data byte unless failure or eof occurs.*/
int v_read(int socket, unsigned char *buf, uint32_t nbyte);

/* write on an open socket (SEND in the RFC)
   return num bytes written or negative number on failure
   Note: you are not expected to implement non-blocking writes.  Writes should
   block until all bytes are in the send buffer*/
int v_write(int socket, const unsigned char *buf, uint32_t nbyte);

/* shutdown an open socket. 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 just fail, 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. */
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. */
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):

help Print this list of commands.

interfaces/li Print information about each interface, one per line.

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

sockets/ls List all sockets, along with the state the TCP connection associated with them is in.

down *integer* Bring an interface "down".

up *integer* Bring an interface "up" (it must be an existing interface, probably one you brought down)

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

connect/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`.

send/s/w *socket data* Send a string on a socket. This should block until write() returns.

recv/r *socket 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.

sendfile *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.**

recvfile *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.

window *socket* Print the socket's send / receive window size.

shutdown *socket 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.

close *socket* `v_close` on the given socket.

We actually provide a .c file which implements most of this functionality. See the Getting Started section below.

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 get to 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:

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

setcc *socket string* Sets the congestion control algorithm for the given socket. To disable congestion control, use the string: *none*

*sockets You should modify your *sockets* command to also list the congestion control algorithm (if any) the socket is using.

*window You should modify your *window* command to also list the congestion window size in addition to the sliding window sizes.

*sendfile 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 should use the TCP packet format, exactly as-is. You can 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.

- 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].

- 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. `bqueue`, and `pthread_cond` are your friends.

- Never send packets greater than the MTU(same as IP). 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 can ignore any options that you see in incoming packets (but don't blow up!).

- 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).

---

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

- 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.

- A non-exhaustive list of RFCs that you might find relevant include: RFC793, RFC2525, RFC6298, RFC2581, RFC1122

4 Tips

A few tips:

- 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.

- Debugging TCP can be very difficult, and sometimes your own logging isn't enough. Therefore, we suggest using Wireshark, an industrial-strength packet analyzer. It will allow you to collect packets travelling over your network cards and filter them by different queries. With some minor configuration, you can tell Wireshark that you are embedding IP packets inside of UDP packets.

- Take a look online for guides on profiling your language. For a lot of languages you can use gprof, CallGrind and KCacheGrind (for example, C, Go and OCaml all allow you to use this).

5 Grading

5.1 Milestone I – 5% (Nov 3)

Set up a meeting with your mentor TA for anytime by the milestone due date. You should demonstrate that your implementation can establish connections, properly following the TCP state diagram under ideal conditions. Note that connection teardown is **not** yet required for Milestone I.

It should work with itself and our reference implementation.

Also show that your TCP works even with another node in between the two endpoints. Your IP and routing should make this trivial. Note that this sounds redundant, but doing this early in the development of TCP will ensure you find any lingering bugs in your IP implementation.

5.2 Milestone II – 20% (Nov 10)

Set up a meeting with your TA for anytime by the milestone due date.

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. The final implementation, will, however require these functionalities be implemented correctly.

5.3   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.

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.

5.4   Performance and Documentation – 20%

Part of this grade will be your TCP's speed. Your implementation should achieve speeds of at least 8 megabytes/second (that's 64Mb/s) on two nodes connected directly to each other, both running on the same computer, with a perfect link. It should also not perform terribly if the link is slightly faulty.

We want you to understand how your design decisions affect your TCP's behavior, so another part of the grade will be a README stating all design decisions you made, and why.

The rest will be a packet trace. You should send a 1 megabyte file, as specified in the Driver section, and have your TCP record all the packets that are sent and received on each end, along with a nanosecond timestamp of when the packet was sent, the sequence number and size of the data in the packet, and the ack number of the packet. You should annotate the first few hundred of these packets with key events, saying why particular events occurred. Run your connection through a faulty node with a 2% drop rate.

You don't have to write too much about the packets that get to the other end safely. The interesting things happen when packets get dropped. When this happens, let us know how this affects your window sizes, how your implementation reacted and retransmitted the dropped packet, etc. Let us know which of your design decisions caused this behavior.

Capstone students will also need to provide performance comparisons and traces for their congestion control implementation. 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.

6   Getting Started

6.1   C Support Code

We've given some C support code, in `/course/cs168/pub/tcp`. If you are not using C, it is not important for you to read these files.

- `util/bqueue.c util/bqueue.h`: A blocking queue. See the header and source files for instructions on how to use it.

- `util/circular_buffer.c util/circular_buffer.h`: A circular buffer, which will probably be useful as you implement the sliding window buffer.

- `util/node.c`: Code for a driver that implements all functionality that only uses the TCP API functions, e.g. connect, sendfile, etc. It only has holes for implementation-specific functions such as up, down, routes, etc. Feel free to take this code and use it for your driver.

- `util/node_readline.c`: The same thing as `util/node.c` except that it uses lib readline. This means you can use up down arrow keys to scroll through your command history and tab to auto-complete file path. This will make your experience with your console much better. Compile it with -lreadline.

We've also modified the debugging macros to prepend messages with a timestamp. These are also in the `util` directory.

6.2   Lossy Network Node

We will provide you with an IP node that will drop a configurable amount of outgoing packets. This will be useful when testing your retransmission and timeout logic. The lossy node is available as `/course/cs168/pub/bin/ip_node_lossy`. 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.

6.3   TCP Reference Implementation

Available as `/course/cs168/pub/bin/tcp_node`. It should be emphasized that your node **MUST** be able to work with the reference node. Make sure you fix any lingering issues in IP preventing your node from working with the reference IP node!

7   Handing In and Interactive Grading

Before each milestone and before the final deadline, once you have completed the requirements for that part of tthe project, you should run the electronic handin script

```
cs168_handin tcp
```

to deliver us a copy of your code. 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.

8   Final Thoughts

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.

Don't tackle the RFC until you're sure that you have your head wrapped around the assignment. For any corner cases or small details, the RFC will be your best friend, and our reference implementation should come in handy. You should read it and consult the TA 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 other groups in the class.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS168 document by filling out the anonymous feedback form:

`https://piazza.com/class/isqj37mfnyz26r?cid=6`.