CS168 Programming Assignment 3: TCP over IP over UDP

Assignment Out:	October 12, 2012
Milestone I:	October 25, 2012
Milestone II:	November 8, 2012
Assignment Due:	November 20, 2012, 11pm

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.

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. You don't have to handle the parts of the RFC that refer to urgent data, precedence, and security.

[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`

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 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 must calculate smooth round trip times and retransmission timeouts as specified in class[4]. Each separate segment you send should have its own timer – that is, if you've written 64kB of data, and the first packet you sent expires, you shouldn't immediately re-send all 64kB.

2.2.1 Graduate Credit: Congestion Control

If you are taking the course for graduate credit, you must implement slow start.

2.3 API

You must implement an API to your TCP implementation. This layer will use integers as handles into a table you maintain, much like unix sockets do, and allow reading and writing into the buffers associated with each connection. Think of it as the system call layer into your "kernel."

An independent thread in your program should be able to use this interface in almost the exact same way that you used the Socket API in your first assignment. These functions, on error, should return a negative value indicating an error code, e.g. `-EBADF` if an invalid socket was passed to a function. The man pages for the unix socket functions are a good source of error codes.

The functionality you need in your socket API is shown below:

[4]Website describing it: http://www.opalsoft.net/qos/TCP-10.htm

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

/* binds a socket to a port
   always bind to all interfaces - which means addr is unused.
   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 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 */
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 */
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);
```

Among the bookkeeping that will be required for this part of the project, you will have your own file-descriptor system. It's important to remember that the file descriptors, port numbers, etc. that are used in these functions will not be actual UNIX system values. They are your own creation, and you are free to instantiate, manage, and free these resources as you see fit.

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 continute 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.

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 whatever recv 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.

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 [5].
- 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. 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.

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

4 Grading

4.1 Milestone I – 5%

Set up a meeting with a 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 Milesone 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.

4.2 Milestone II – 20%

Set up a meeting with a 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.

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.

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 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 congestion window, how your implementation reacted and retransmitted the dropped packet, etc. Let us know which of your design decisions caused this behavior.

5 Getting Started

5.1 IP Reference Implementation Code

We've released the source code for our IP reference implementation. Feel free to look at it and take whatever you like from it. You can even completely use ours instead of your own, if you like it more. It's in `/course/cs168/pub/ipsrc`.

5.2 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.
- `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.
- `node_readline.c`: The same thing as `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.

5.3 TCP Reference Implementation

Available in `/course/cs168/pub/tcp` as `node`. You can specify a drop rate as the second parameter at the command line, to simulate lossy links. The drop rate should be a value between 0 and 100, where 100 means every packet will be dropped by the node.

5.4 Hand In

`/course/cs168/bin/cs168_handin tcp`

6 Final Thoughts

Although we expect compatibilty 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.