

CS168 VAN – Virtually Active Network TCP

Assignment Out: Oct 18, 2007
Assignment Due: November 12, 2007 (11:59 pm)

1 Introduction

The second part of the VAN project builds on what you wrote for IP. You will build a transport layer (like TCP) with error detection and export a socket API similar to what you used in Snowcast, on top of your implementation of IP. Chapter 6 of the book should be especially helpful.

2 The Pieces

In this assignment you will use the library you wrote for IP as the underlying network. Needless to say, if you didn't get it working properly, the first thing you should do is finish it off.

Unlike in the last assignment, you need to use the drop and garble rates in the netconfig file. You cannot assume that the network is reliable. Therefore, you must implement the acknowledgment methods discussed in class and in the texts and a checksumming algorithm which we provide (`/course/cs168/pub/checksum.[ch]`). You might want to start this assignment by adding checksumming to your IP code.

You have to implement a state machine that allows state transitions in your TCP. The state machine is not complicated, but you should be sure that your TCP can follow all state transitions properly, and doesn't do anything otherwise. For example, you need to send SYN's for connect, and FIN's to close.

You also need to implement the sliding window protocol. 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 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. Furthermore, think about the possibility of receiving old packets from a previous connection.

In summary, you will need to implement checksumming, your TCP state machine, sliding window with acknowledgements, your own file descriptor system, and your own API.

3 The API

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

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

/* binds a socket to a port
   returns 0 on success or negative number on failure */
int v_bind(int socket, int node, short port);

/* moves socket into listen state
   returns 0 on success or negative number on failure */
int v_listen(int socket, int backlog /* optional */);

/* connects a socket to an address
   returns 0 on success or a negative number on failure */
int v_connect(int socket, int node, short port);

/* accept a requested connection
   returns new socket handle on success or negative number on failure */
int v_accept(int socket, int *node);

/* read on an open socket
   return num bytes read or negative number on failure or 0 on eof */
int v_read(int socket, unsigned char *buf, int nbyte);

/* write on an open socket
   return num bytes written or negative number on failure */
int v_write(int socket, const unsigned char *buf, int nbyte);

/* close an open socket
   returns 0 on success, or negative number on failure */
int v_close(int socket);
```

These functions should all look familiar to you, since they were the core set used in Snowcast. 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.

Finally, of course, you should build some test code on top of the system. This will be useful both to make sure your code is working and to demo it to your TA. In addition to your IP test

driver, you should have some mechanism of delivering large files over TCP. An ideal way to do this is to port your Snowcast code over to your version of TCP/IP. You can then transmit large data files (like ps files) and diff the original with the one received over the VAN. You will have to demonstrate that your TCP implementation is capable of reliably transmitting files. They should be *identical* if your TCP is working properly, even if the garble and drop rates are set very high.

4 Hand In / Demo

Hand in: Before the due date, you'll need to handin using the typical method:
`/course/cs168/bin/cs168_handin tcp`

Demo: Next, you'll need to send your Mentor TA email and set up a demo time. Beforehand, you should design a simple network and demo program to show that you can indeed send data (intact) over garbled links.

5 Final Thoughts

Do not depend on the RFC. It is much more important that you understand how TCP works on an algorithmic/abstract level. Design your project based on the books and JJ's slides. 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, however. You should read consult it and 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.

Start early and good luck. Seriously, this assignment is an order of magnitude harder than its predecessor (no seriously, it is).