Matt Forbes

Assignment 5

March 4, 2011

1 Server Documentation

My server implements all elements declared in the RFA protocol. It passes both my tests and tests written by Dr. Nelson. The server relies on the functions defined in netlib.h, which along with netlib.c, should be included in the directory. There are a few ways that the code could be cleaned up, but I'd rather not break it in the process.

1.1 Overview

The bulk of the program is in the function handle_client, which is called by a forked process after a client connection is made. In this function, an array of all the open file descriptors is held. Rather than handing back actual file descriptors to the client, indexes to this array are passed. I'm not sure why I did this.

When a client makes a request, an action is performed based on the first character of the message. Currently, the request string is parsed character by character which is something that could be cleaned up.

1.2 Structures

Unfortunately, I didn't use any interesting data structures except for a linked list. It's main usage was in loading up the authorized hostnames; it made it easy to just read the .rfahosts line-by-line and throw it in to the list.

1.3 Tests

Unlike my client code, I didn't do much testing until the program was complete. After getting the Open procedure working and parsing down, the rest of the functionality just fell in to place. Besides running the 'try' test script, I played around with ncat to make sure files were being written and read correctly.

1.4 Possible Improvements

One thing that really bothers me about my code is how request strings are being parsed. It works just fine, but the same code is duplicated on multiple lines in multiple functions. My initial attempt was to use the strtok function from the string library, but that breaks when filenames have spaces. In hindsight, I should have used that method for all the requests except for Open, which is a special case.

2 Client Documentation

I took a much different approach to the client library. Rather than blindly forge ahead and code out the whole project, I spent some time thinking about how the parts would fit together beforehand. Coupled with unit testing, this program was smooth sailing. As far as functianality goes, it passed all my unit tests, and Dr. Nelson's test scripts. All seems well.

2.1 Overview

As the assignment entails, there are really just four functions that need to be provided: open, read, write, and close – the stand file manipulation routines. They are prototyped in a way that they seemlessly mask the corresponding system calls and provide drop-in network availability.

2.2 Structures

The three structures I built were as follows:

1) **host:** Hold server information so that connections can be reused. They are uniquely identified by their hostname/port combination. An array of these are stored globally and are checked for existence when a server connection is requested. Multiple files can be opened with just a single connection to an RFA server.

2) **file_d:** I'm keeping an array of all the remote file descriptors opened, each represented as a **file_d**. When a remote file is opened, the first open index in this global array of remote file descriptors is added to 1,000,000 in order to guarantee an invalid local file descriptor. Indexes to this array can be found by simply subtracting 1,000,000 from the file desriptor. **file_d**s hold a pointer to the corresponding host, so it's easy to just grab the corresponding socket.

3) **pathinfo:** Pathnames are specified in the format: [[port@]machine:]path. A **pathinfo** breaks this string in to the associated pieces, making it much easier to decide how to handle the opening of the file. The decision to use a structure for this task was motivated by my less-than-optimal solution in the server.

2.3 Tests

Knowing that small mistakes can completely throw things off (especially in C), I decided to incrementally implement each function with a "suite" of unit tests. Compared to some of the really nice unit test frameworks out there, mine are really just quick checks through printf. My goal was to represent all possible use cases of each function print out the results, which I just compared by hand as there weren't too many. In a larger project, I'd definitely want to be able to automatically detect if tests were passed/failed rather than just inspecting them manually. Developing this way was much more stress-free, when I made a change to the program, I could be comfortable that nothing broke.