

HW1 Sockets

For this homework we'll build a couple Perl programs that use sockets. Part A is a simple Perl client for the POP email reading protocol. Part B is a simple client-server pair. Both parts are due midnight ending Tue 4-18-2000. (There'll be a separate handout on the mechanics of turning in your work online.)

P/NC

As an experiment, I'm going to allow P/NC people to work in teams of two on an assignment if they wish. P/NC people also do not need to do part B.

Part A — TopMail

For this part you will write some client-side socket code and read a real RFC to get an appreciation for how Internet services come together.

Find The RFC

The first step is to go to the IETF (Internet Engineering Task Force) home page at www.ietf.org, go to the Request for Comments (RFC) section, and search for the string "post" to find the POP RFC and read it. I can't tell you exactly which RFC to use because that would spoil the fun of hunting around for it (well ok, it's the 19xx revision). Every piece of Internet software begins with the humble but organizing act of locating and reading the appropriate RFC. We're writing a simplified client, so only about half of the RFC applies. Our client will only use the commands USER, PASS, STAT, RETR, and QUIT.

TopMail

TopMail takes as input the username and POP server to use, connects to the POP server, prints out some statistics about the mailbox, and then prints out the three most recent messages. Please use exactly the following format so as not to confuse the grading scripts...

```
% topmail.pl nick nick.pobox.stanford.edu
Connect nick.pobox.stanford.edu
341 messages 934245 bytes
```

```
Message 341
<all of the lines of message 341, header and body>
```

```
Message 340
<all of the lines of message 340, header and body>
```

```
Message 339
<all of the lines of message 339, header and body>
%
```

The program should connect to the given POP server using the appropriate port #, and if successful print the "Connect..." line. It should then try to log in the

given user. If the login is successful, it should print the status line giving the number of messages and number of bytes and then print out the three most recent messages, each preceded by a blank line. It should send the "QUIT" command to the server when it's done. If there are fewer than three messages, it should print whatever is available. The flow of control in the program is simplified by the fact that there is no user interaction after the initial input.

Password

TopMail could take your password on the command line along with the other arguments, but that's a little awkward since your password is on screen for all to see. Instead, we will put your password in an environment variable just for the duration of your testing. Use the following command to enter your password once in the shell where you will be running TopMail..

```
% setenv IPASSWORD yourpassword
% clear                                ## clears the screen
```

In your code, use the expression `$ENV{'IPASSWORD'}` to retrieve the password. When you log out, the environment variable is forgotten — you'll need to `setenv` it again each time.

To get started read the RFC and get going — just like a regular Internet programming citizen (this is more or less exactly how Marc Andreason got his start).

Details

Here are few implementation details to think about...

EOLN Handling

Unfortunately, our home planet has several different conventions for how the end of a line of text is marked. The most common end-of-line format is the two character sequence Carriage-Return Linefeed, aka CRLF, aka `"\r\n"`. Written with octal character constants that's `"\015\012"` (in decimal that's 13 followed by 10). (For maximum portability, in your code use the constants `"\015\012"` since some language/platform combinations re-map `"\n"` to be the local end-of-line character.) As a practical matter, Internet servers and clients will use one of the two end of line conventions: `"\r\n"` or `"\n"` (or more rarely `"\r"`). Or a server may use different conventions for different parts of the dialog. A well behaved piece of Internet software should be able to accept lines written with any of the conventions at any time. For this course, your programs should at least be able to deal with either the `"\r\n"` or `"\n"` conventions. Once a line is read in, whatever its end of line convention, it should be converted to end with a single `"\n"`. This will best enable the text to be saved or printed on the local machine.

Error Handling

As with most networked software, there are many error conditions which may occur during a run of the program. For error cases, TopMail should print a short error statement on its own line and then `exit(-1)`. Some of the error cases which need to be caught are: bad hostname, could not create socket, could connect, bad

username, bad password, STAT error (unlikely to ever happen), RETR error. Please use the following error statements (in reality you'd want something more descriptive, but we need to keep it terse for the grading scripts) ...

```
error hostname
error socket
error connect
error username
error password
error stat
error retr
```

For this assignment, we will not worry about read and write errors -- they are pretty rare once the connection is up.

POP Message Format

As described in the RFC, POP sends the lines of the message one after the other. The end of the message is marked by a line which contains a single period character: ".\r\n". POP uses a "byte-stuffing" technique in the case that a line in the middle of the message happens to begin with a period — in that case POP adds an extra period to the start of the line so that it does not look like the end-of-message marker. Your POP client needs to deal with all this and present the message as the sender intended.

Leland POP Hosts

It happens that the hostname to use for leland based POP email retrieval is username.pobox.stanford.edu. This convention may be to spread the POP load of the thousands of Stanford POP users across several POP servers. The local DNS system has been tweaked to pull the username off the front of the DNS address and return the IP address of the correct POP server. If you have your email forwarded from leland, go to stanfordyou.stanford.edu to temporarily set it to leave a copy at leland for TopMail to find. Running elm or mail on a leland machine immediately empties the POP mailbox, so you may need to send yourself more test messages to have something to test with.

Other Commands

POP defines other commands, such as DELE and APOP — we are not worrying about these. In particular, if your code does not mention the command DELE anywhere, then it should be impossible for your program to delete any of your email!

Decomposition and Style

We mostly grade on correctness. So you should use a common-sense quality coding style. You will want to decompose out functions at least for socket creation, reading, and writing. Partly because its the right thing to do and will help keep your engineering soul from getting all dirty. And partly because we're going to write other socket homeworks, and you're going to want to re-use that code.

Part B — Quote Server

For Part B you will build both the client and server sides of a simple quote exchange protocol. For simplicity, both the client and server code will be implemented in `quote.pl` and a command line switch will determine which role the program is taking. The client connects to the server and sends a "target" string. The server sends back quotes that include that string.

Client Side

If the first command line argument is "-c" then the program should run in client mode. In client mode, the three other arguments are the name of the machine to connect to, the port to use, and (optional) the target string to use. The client should connect to the server, send the target string on a line (the target may be the empty string), and read back and print out all the lines the server sends back.

```
elaine25:~/193i> quote.pl -c elaine34 2915 fish
Even a fish could stay out of trouble if it would just learn to keep its mouth shut.
elaine25:~/193i> quote.pl -c elaine34 2915 gas
We don't have time to stop for gas -- we're already late.
elaine25:~/193i>
```

Server Side

If the first command line argument is "-s" then the program should run in server mode. In server mode, the program should take as input a port # to use and the name of a file with quotes on each line. The server should wait for incoming connections on that port. When the client connects, the client will send a single "target" string on a single line. The server should iterate through the quotes, send back every quote that contains the target (each on its own line), and then close the connection.

```
elaine34:~/193i> quote.pl -s 2915 quotes.txt
Quote server ready on port '2915' with file 'quotes.txt'
Connection from 171.64.15.100 44578
Client string:'fish'
Connection from 171.64.15.100 44581
Client string:'gas'
^C
elaine34:~/193i>
```

The server should print out the initial "server ready" notice and then the two status messages for each connection. The server should read the quotes file into a global array when it starts up so the necessary data is all in memory when the connection comes in.

Error Handling

As with part A, for error conditions, the program should print a standard, terse message and then `exit(-1)`. Please use the following error statements...

```
error hostname
error socket
error connect
```

```
error sockopt
error bind
error listen
error accept
```

Mechanics

The easiest way to test the program is with two terminal windows logged in to two machines, but in the same directory. Start the server in one terminal. Use ctrl-c to kill it when necessary. Run the client in the other terminal. When you edit quote.pl, you still need to kill and re-start the server to get it to use the new code. Use the "up-arrow" feature of the shell to run things without having to type it all out each time.