

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 (separate handout). Both parts are due 2:00 am Fri the 27th (normally, assignments are due at midnight, but in this case we choose a weird time to try to spread out the load on the server).

P/NC

P/NC may work in teams of two on an assignment if they wish. P/NC people also do not need to do part (b).

For the first 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. You can sprinkle print statements through your program for debugging, but please comment them out for your final submission.

```
% topmail.pl nick pobox6.stanford.edu
password:
Connect pobox6.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 prompt the user for their password with something like the following...

```
system("stty -echo");    ## turn off echo
print "password:";
my($password);
$password = <STDIN>;    ## this needs to be on a separate line from the "my"
chomp($password);      ## remove the end-of-line char
system("stty echo");
print "\n";
```

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 as above, and then print out the three most recent messages as above. Note: "octet" means "byte". 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.

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

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 in Internet protocols 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 written out locally with the "\n" ending. This will best enable the text to be saved or printed on the local machine.

This subroutine reads a line of text from a file handle, and discards the line ending chars, whatever they are (the code is available in the /usr/class/cs193i/hw directory).

```
# Readline('SOCK') -- given a file handle,
# reads and returns a single text line
# with the EOLN char(s) removed. Returns false on EOF.
sub Readline {
    my ($sock) = @_ ;
    my($line);
```

```

$line = <$sock>;          ## read a single text line
if (!$line) { return $line; } ## check if there's nothing (EOF case)
$line =~ s/[\r\n]//g;    ## delete all the \r \n's
return($line);
}

```

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).

```

if (some error condition) {
    print "this error happened\n";
    exit(-1);
}

```

Do not call die. 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

```

Some servers will never report a username error, but your program should just follow the RFC anyway. 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 exactly as the sender sent it.

Password "in the clear"

When using the ordinary POP protocol, the password is transmitted on the socket "in the clear" -- it is not encrypted. This makes it possible (although it's rare), for someone to "sniff" the password by watching network traffic. The ultimate solution is to use a more complex protocol than POP. (Kerberos KPOP is one such protocol -- I looked in to using it, but it's too complex for hw1). For homework purposes, POP is fine, and in fact it's currently the world's most widely used email access protocol. Fortunately, the nice leland people have set up a special POP server for cs193i students to test against. The server is pobox6.stanford.edu. People registered for cs193i automatically have accounts on

this server. On that server, the plan is that password for a user is the same as the username (if the server admin wants to use a different password convention, we'll put a notice on the course page). You can send email messages to your pobox6 account -- user@pobox6.stanford.edu.

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.

Miscellaneous Tips

- Whenever you call `$line = <F>;`, don't forget to handle the end-of-line char(s).
- Rember to set your sockets to be unbuffered -- if you are sending something on a socket and it never shows up at the other end, buffering is usually the problem.
- Never write this: `my($line) = <F>;`
This is a Perl gotcha: written on one line, the `my($line)` is an array context, so the `<F>` tries to read **all** the lines. Write the `$line = <F>;` on a separate line.
- Use prints statements liberally in your program so you can see what it's doing; that's how Perl debugging is traditionally done. Comment them out for your final submission.
- In most unix shells, the up-arrow key scrolls back through recent commands. Use this to run your program again without re-typing the arguments.
- Just for fun, you could use your topmail to look at your real leland email account, but this has the password sniffing problem above. Topmail will only see email before you have downloaded it to your directory. Also, leland is going to discontinue POP access pretty soon.

The submit program will have its own instructions. Basically, you'll have your hw1 files in a directory on leland and run submit to copy them up into the submit directory.