

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 2:00 am Wed 24th (i.e. late Tue night). Normally, assignments are due at midnight, but in this case we choose a weird time to try to spread out the load on the POP server. See our homework directory at `/usr/class/cs193i/hw` for starter files.

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 find the Post Office Protocol RFC. and read it. I won't tell you exactly which RFC, since I want you to have to hunt it down like a real Internet programmer (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. It's all about standards. 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.

Part (a) -- TopMail

TopMail takes as command line arguments 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 courses1.stanford.edu
password:
Connect courses1.stanford.edu
6 messages, 5064 bytes
-----
Message 6
Return-Path: <nick@cs.stanford.edu>
Received: from smtp2.Stanford.EDU (smtp2.Stanford.EDU [171.64.14.116])
    by courses1.Stanford.EDU (8.11.6/8.11.6) with ESMTTP id g3CGxs808787
    for <nick@courses1.stanford.edu>; Fri, 12 Apr 2002 09:59:59 -0700 (PDT)
Received: from smtp2.Stanford.EDU (localhost [127.0.0.1])
    by smtp2.Stanford.EDU (8.11.6/8.11.6) with ESMTTP id g3CGxsE11569
    for <nick@courses1.stanford.edu>; Fri, 12 Apr 2002 09:59:54 -0700 (PDT)
Received: from cs.stanford.edu (nick3.Stanford.EDU [171.64.64.167])
```

by smtp2.Stanford.EDU (8.11.6/8.11.6) with ESMTTP id g3CGxrw11564
 for <nick@courses1.stanford.edu>; Fri, 12 Apr 2002 09:59:53 -0700 (PDT)
 Message-ID: <3CB71284.DD9211F7@cs.stanford.edu>
 Date: Fri, 12 Apr 2002 09:59:48 -0700
 From: Nick Parlante <nick@cs.stanford.edu>
 X-Mailer: Mozilla 4.79 (Macintosh; U; PPC)
 X-Accept-Language: en
 MIME-Version: 1.0
 To: nick@courses1.stanford.edu
 Subject: last notice
 Content-Type: text/plain; charset=us-ascii
 Content-Transfer-Encoding: 7bit

This is the most recent message,
 gosh darn it!

Kindest regards,

Nick

Message 5
 Return-Path: <nick@cs.stanford.edu>
 Received: from smtp2.Stanford.EDU (smtp2.Stanford.EDU [171.64.14.116])
 by courses1.Stanford.EDU (8.11.6/8.11.6) with ESMTTP id g3CGui808782
 for <nick@courses1.stanford.edu>; Fri, 12 Apr 2002 09:56:49 -0700 (PDT)
 Received: from smtp2.Stanford.EDU (localhost [127.0.0.1])
 by smtp2.Stanford.EDU (8.11.6/8.11.6) with ESMTTP id g3CGuhE10343
 for <nick@courses1.stanford.edu>; Fri, 12 Apr 2002 09:56:43 -0700 (PDT)
 Received: from cs.stanford.edu (nick3.Stanford.EDU [171.64.64.167])
 by smtp2.Stanford.EDU (8.11.6/8.11.6) with ESMTTP id g3CGugw10331
 for <nick@courses1.stanford.edu>; Fri, 12 Apr 2002 09:56:42 -0700 (PDT)
 Message-ID: <3CB711C5.1FCCAE3B@cs.stanford.edu>
 Date: Fri, 12 Apr 2002 09:56:37 -0700
 From: Nick Parlante <nick@cs.stanford.edu>
 X-Mailer: Mozilla 4.79 (Macintosh; U; PPC)
 X-Accept-Language: en
 MIME-Version: 1.0
 To: nick@courses1.stanford.edu
 Subject: el testito
 Content-Type: text/plain; charset=us-ascii
 Content-Transfer-Encoding: 7bit

Behold !

Some Email !!

Regards,

Nick

Message 4
 Return-Path: <nick@cs.stanford.edu>
 Received: from smtp2.Stanford.EDU (smtp2.Stanford.EDU [171.64.14.116])
 by courses1.Stanford.EDU (8.11.6/8.11.6) with ESMTTP id g3CGuR808777
 for <nick@courses1.stanford.edu>; Fri, 12 Apr 2002 09:56:32 -0700 (PDT)
 Received: from smtp2.Stanford.EDU (localhost [127.0.0.1])
 by smtp2.Stanford.EDU (8.11.6/8.11.6) with ESMTTP id g3CGuBE10135
 for <nick@courses1.stanford.edu>; Fri, 12 Apr 2002 09:56:11 -0700 (PDT)

```

Received: from cs.stanford.edu (nick3.Stanford.EDU [171.64.64.167])
        by smtp2.Stanford.EDU (8.11.6/8.11.6) with ESMTP id g3CGu9w10124
        for <nick@courses1.stanford.edu>; Fri, 12 Apr 2002 09:56:09 -0700 (PDT)
Message-ID: <3CB711A5.CB8FBEDC@cs.stanford.edu>
Date: Fri, 12 Apr 2002 09:56:05 -0700
From: Nick Parlante <nick@cs.stanford.edu>
X-Mailer: Mozilla 4.79 (Macintosh; U; PPC)
X-Accept-Language: en
MIME-Version: 1.0
To: nick@courses1.stanford.edu
Subject: testing
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit

```

This that and,
the other.

Woo Hoo

Nick
>

The program should prompt the user for their password with something like the following so that the password does not echo on screen...

```

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# (our POP server uses port 110), 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: historically, "octet" means "byte" in ancient computer-speak of RFCs. Your client should send the "QUIT" command to the server when it's done. If there are fewer than three messages, it should print the messages that are available, if any. 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). Notice that you do not have to get anyone's permission or sign a non-disclosure agreement or anything to implement the RFC.

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 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 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 (this function is available in starter code).

```
# 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

    ## print "read on socket:$line"\n"; ## use for debugging

    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; it prints to much extraneous stuff. Some of the error cases which need to be caught are: could not lookup hostname, could not create socket, could not connect, bad username, bad password, STAT error (unlikely to ever happen), RETR error. Please use the following error phrases in your code (in reality you'd want something more descriptive, but we need to keep it simple 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 to not give bad guys a way to discover user names, but your program should follow the RFC anyway. For this assignment, we will not worry about low level read and write errors -- they are 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 the 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 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 `courses1.stanford.edu`. People registered for cs193i automatically have accounts on this server. Email `cs193i@cs` if you need an account created. The passwords on that server are all "foobar". You can send email messages to your account -- `user@courses1.stanford.edu`. This server only works from within the Stanford domain, so send your test messages and run `topmail` on campus.

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 aim for a common-sense, reasonable quality in your coding style. You may want to use a little decomposition for the socket operations so you can re-use that code on later assignments. There's some code provided for you in the starter files.

Miscellaneous Tips

- Whenever you call `$line = <F>;`, don't forget to handle the end-of-line char(s).
- Remember to set your sockets to be unbuffered -- if you are sending something on a socket and it never shows up at the other end, buffering is usually the problem.
- Never write this: `my($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 `print` 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.
- You can also use your topmail to look at your email on a real account that supports straight POP access. We are implementing the real RFC after all, and that really is what the world uses.

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.

Part (b) -- Quotes

HW1a is about writing a typical, RFC based client side application. Furthermore, it sent passwords in the clear. HW1b is just the opposite. HW1b will involve both the client and server sides of a simple protocol, and will not send passwords in the clear.

For simplicity, both the client and server code will be implemented in `quote.pl` and a command line argument will determine which role the program is taking. For the first milestone, we'll consider the program without any passwords. The server blocks on a port as usual. The client connects, sends a "target" string, and the server sends back quotes that include that string and closes the connection.

Perl

In order to use the MD5 Perl library, the top of your Perl file needs to look like this (the starter file is set up for you this way)...

```
#!/usr/pubsw/bin/perl5.6.1 -I/usr/class/cs193i/pm -w

use Socket;
use FileHandle;

use Digest::MD5 qw(md5 md5_hex md5_base64);
```

The first line says to use the perl5.6.1 version on leland (it works on the elaines; I have not tried the other machines). This is required because the older `/usr/bin/perl` is not compatible with the `Digest::MD5` module. The `-I` directive points to the `Digest::MD5` module which is in the "pm" directory in the 193i directory. You must run your perl program as `./quote.pl` (or simply `quote.pl` if you have "." (dot) in your `$PATH` set in your `.cshrc` file.). This requires that the "execute" bit is set on the file -- see the instructions in the Essential Perl handout. By running the program in this way, the `#!...` line can specify the Perl options we want. The `Digest::MD5` module is available as open source from www.cpan.org if you'd like to make your own install, but if you just run on the elaines, we have it all installed already.

If you are having problems getting the MD5 module to work, it's possible to do the assignment without it, but it's not as neat.

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. If the target is not present on the command line, the target is the empty string. The client should connect to the server, send the target string on a line (if the target is the empty string, this amounts to just sending the end-line character), and read back and print out all the lines the server sends back. The quote protocol will use a simple `"\012"` as the line ending convention in both directions. Your quote client and server should implement the quote protocol faithfully enough to interact correctly with another program that implements the quote protocol.

```
% ./quote.pl -c localhost 61784 "fish"
Even a fish could stay out of trouble if it would just learn to keep its mouth shut.
% ./quote.pl -c localhost 61784 "gas"
We don't have time to stop for gas -- we're already late.
% ./quote.pl -c localhost 61784
This time for sure!
%
```

In the above 'localhost' is a special DNS name that points to the local machine, and "./quote.pl" is a way to run a program in your current directory.

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. Pick a random port number to use. When it starts, the server should read the file a single time into an array. The server should wait for incoming connections on that port. When the client connects, the server sends a "hello" greeting on a line by itself. The client ignores this greeting. The client sends the target on a line by itself. The server iterates through the quotes, sends back all the quotes that contain the target as a substring anywhere in the quote. You may use the `index()` function to do the search, or a regular expression match (`$line =~ m/$target/i`). If the target string is the empty string, then the server should instead select a quote at random and send that back.

```
% ./quote.pl -s 61784 quotes.txt
quote server 61784
connection from localhost
target 'fish'
connection from localhost
target 'gas'
connection from localhost
target ''
^C
```

The server should print out the initial "Quote server *port*" 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 to standard output 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 retrieve a previous command without re-typing it.

Put lots of print statements throughout the client and server code so you can see what is going on. You can comment them out later.

Milestone

Get the basic no-password version working reliably before trying the next step.

Authentication

We can't let just anyone look through our valuable quote database! We'll need to do some authentication to make sure that each party is who they say they are. The scheme below will allow the client and the server to each authenticate that the other knows what the password is. This is a little stronger than what POP does, where the client must authenticate, but the server does not. This scheme has some theoretical weaknesses vs. a determined bad guy, but it works well enough and it's a nice demonstration of 2-way challenge-response authentication that avoids sending the password over the network.

The server will control whether authentication is used. We'll add a `-p password` command line argument that follows the `-s`. If `-p` is present, the server will insist on authentication. In that case, the server sends the string "auth" as its initial greeting instead of "hello". The client can have a `-p password` command line argument after the `-c`. The client will use that password if the server insists on authentication. If the `-p` command is not present, the password defaults to the empty string.

1. The server sends "auth" as its greeting line, followed by T and R1, each on a line by itself. T is the current GMT time as a string. In Perl, use the function `gmtime()` to get this string. R1 is a 4 digit random number (use `substr(rand(), 2, 4)`). These two pieces of information are the "challenge" to the client.
2. The client responds to the server challenge. The client makes a two line response: `hash(T,R1,P)`, where "hash" is the one way function described below, called on the T, R1, P strings. P is the password, which is known to the client. The second line of the client response is R2, a random number computed by the client.
3. The server responds to the client challenge by computing and sending `hash(T, R2, P)` on a line.

4. After exchanging challenge/response pairs, the client and server each print a "had:local-hash got:remote-hash" to local output comparing the hash expected vs. the hash received.
5. Now, the client and server can each decide if they are "satisfied" with the response they got -- did it match the hash expected. Each party must be satisfied to proceed with the quote protocol. If the server is not satisfied, it prints "error: bad client password", sends "-quit" as the first quotation, and closes the connection. If the client is not satisfied, it prints "error: bad server password", sends "-quit" as the target string, and exits. Each end should detect if the other is sending "-quit", and close the connection gracefully and without printing the "-quit". Usually, both will be unhappy at the same time, but it's possible for one to be happy and one not if one is a "bad guy" program, so make sure you test that case.
6. If both are satisfied, the quote protocol proceeds as it did without authentication. The client sends a target string, the server responds with quotes.

The hash function takes three strings and combines them into a token. The hash() function is present in the starter file. For initial testing, the hash function should just concatenate the three strings, separated by "*" characters as you can see in the have:/got: lines below. This is a terrible hash function, but it works and it's perfect for testing.

```
Server:
% ./quote.pl -s -p foo 61784 quotes.txt
quote server 61784
connection from localhost
have:Sat Apr 21 16:24:16 2001*3329*foo got:Sat Apr 21 16:24:16 2001*3329*foo
target ''
connection from localhost
have:Sat Apr 21 16:24:33 2001*8635*foo got:Sat Apr 21 16:24:33 2001*8635*foo
target 'gas'
connection from localhost
have:Sat Apr 21 16:24:52 2001*3586*foo got:Sat Apr 21 16:24:52 2001*3586*bar
error: bad client password
^C
```

```
Client:
% ./quote.pl -c -p foo localhost 61784
have:Sat Apr 21 16:24:16 2001*4301*foo got:Sat Apr 21 16:24:16 2001*4301*foo
Beware the lollipop of mediocrity -- lick it once and you suck forever.
% ./quote.pl -c -p foo localhost 61784 "gas"
have:Sat Apr 21 16:24:33 2001*2293*foo got:Sat Apr 21 16:24:33 2001*2293*foo
We don't have time to stop for gas -- we're already late.
% ./quote.pl -c -p bar localhost 61784 "gas"
have:Sat Apr 21 16:24:52 2001*8312*bar got:Sat Apr 21 16:24:52 2001*8312*foo
error: bad server password
%
```

Once the '*' version is working, set the constant `hash_md5` to 1. This changes the `hash()` function to use the real MD5 hash "one way" function -- it hashes the strings in a way that is believed to be very difficult to invert. That is, if a bad guy sees the hash, they will not be able to compute the (T, R, P) that made it (or at least it will be very computationally expensive to do so). So essentially, a bad guy observing the quote session will not be able to sniff the password. Here's what the interaction looks like with real hashing (also, in this case I was running on two different machines, so the host names are more interesting)...

Sever:

```
elaine9:~/193i/quote> quote.pl -s -p foo 3465 quotes.txt
quote server 3465
connection from elaine0.Stanford.EDU
have:1Bdlhad4P565C/yzCqvjvg got:1Bdlhad4P565C/yzCqvjvg
target ''
connection from elaine0.Stanford.EDU
have:tYg5JA1XitX4vnCPbCKQjQ got:tYg5JA1XitX4vnCPbCKQjQ
target 'fish'
^C
```

Client:

```
elaine0:~/193i/quote> quote.pl -c -p foo elaine9 3465
have:XUmeAH9/wD9JilH4gXJ3DQ got:XUmeAH9/wD9JilH4gXJ3DQ
Even a fish could stay out of trouble if it would just learn to keep its mouth shut.
elaine0:~/193i/quote> quote.pl -c -p foo elaine9 3465 fish
have:xLBAR2HAuJldmonrS7pzLg got:xLBAR2HAuJldmonrS7pzLg
Even a fish could stay out of trouble if it would just learn to keep its mouth shut.
```

Please switch back to the "*" hash version for what you hand in -- the md5 version is harder to deal with for debugging and grading.

Deliverables

You can build your assignments on the machine of your choice, but you should do a little testing on leland since that's where we will test them. See the instructions in the `/usr/class/cs193i/bin/README.submit` for instructions on running the submit program to submit your hw1 directory.