15-441 Project 1 : SMTP client

Dates of Interest

Assigned : Tuesday, Jan 23rd, 2007
Due : Friday, Feb 8th, 2006, 11:59pm

Interesting RFCs

RFC 821: Simple Mail Transfer Protocol (SMTP)
RFC 822: Standard For the Format of ARPA Internet Text Messages
RFC 976: Mail Routing and the Domain System

You can find any RFC here:

http://www.faqs.org/rfcs/

Bboard

Andrew: academic.cs.15-441

Introduction

The purpose of this project is to give you experience in developing a network applications, specifically a *Simple Mail Transfer Protocol (SMTP)* client. It also serves as an introduction to Requests for Comment (RFCs), the mechanism for proposing and publishing new Internet standards.

SMTP is the most widely used mail transmission protocol on the Internet. As its name suggests, it is reasonable straight-forward. None-the-less, there is subtlety involved in transferring mail. So, although it should be reasonably quick to implement a client that can send mail in the simplest cases, additional time will be spent on some of the subtlety.

Logistics

You are to work in groups of two (2) on this project. You are free to discuss the project and references with your classmates, but your solution must be your own, or that of you and a single partner.

You must code your server in C (not C++). While other languages may provide features to make network programming easier, most kernel-level networking code is written in C and future assignments will require implementing such code in a simulated environment.

You may develop & test your code on any operating system you wish, however, we will grade your code on Andrew Linux machines, so please make sure your code compiles and runs as expected on them.

Your Assignment

We'd like you to write, in C, two different SMTP clients: *smtpverify* and *smtpsend*.

smtpverify

smptverify is the more straight-forward of the two. It should accept a single argument and an option server flag, for testing. This argument should use SMTP's *EXPN* (RFC 821) to verify the correctness of the mailbox or mailing list provided. In the case of a mailbox, it should print, one per line, each fully-specified mailbox in the list. It should treat the case of a mailbox as a mailing list containing only one mailbox: In other words, it should print the fully-qualified mailbox on its own line. If the mailbox or mailing list is unknown, "Unknown user" should be printed. *smptverify* should return 0 if, and only if, it successfully queries the server and gets a response, regardless of the response, and non-0 otherwise.

smtpverify [-server=*blah.blah.bla*] *emailaddress@hostspec*

smtpsend

smtpsend which, as its name suggests, is used to send mail via an SMTP server, is a bit more sophisticated than *smtpverify*. It is invoked as below:

smtpsend [-server=*blah.blah.bla*] *some.domain local-part-1* [*local-part-2...*] < *messagefile*

- *some.domain* is the domain part of an e-mail address, e.g., andrew.cmu.edu,
- one or more *local-part* parameters indicate users inside that domain (e.g., "gkesden"
- *messagefile* is an e-mail message formatted according to RFC 822.
- The –*server* flag can be used to force the client to talk to a specific SMTP server for testing. Since *courseweb* relays for andrew, this might be helpful to send yourself mail without having to talk to an andrew server.

If the program can successfully submit the mail for all recipients, it should exit with status zero; if not, it should exit with a non-zero status and print on standard error one line indicating the first recipient for which there was a problem and one line indicating the nature of the problem. If a "-v" (verbose) flag is provided before the domain parameter, the program should print each SMTP command (one line) and the corresponding response from the SMTP server (which may not be exactly one line, see RFC 821).

For full credit you should handle the case where the mail domain does not designate a machine running an SMTP server, as explained in RFC 974. For example, while there is a machine called "andrew.cmu.edu", it is not running an SMTP server and e-mail for the "andrew.cmu.edu" domain must be submitted to other machines. You do not need to do WKS queries as described in the "Interpreting the List of MX RRs" section of RFC 974; you may assume that every designated mail exchanger does support the SMTP protocol.

We expect you to use the socket API as discussed in class. A complete solution will probably need to use the DNS resolver library as documented in the resolver(3) manual page.

The RFC leaves several details up to the implementation. You should justify the significant choices you make in your README file.

Testing

The SMTP server on *courseweb.sp.cs.cmu.edu* has been configured to relay for the *andrew.cmu.edu* domain, but not any other. It has some user mailboxes that can receive mail, and some mailing lists that you can expand. We'll provide more details in a couple of days or so, once you've had some time to digest the RFCs.

Please do not test using any other SMTP server which is not your own or any mailboxes which are not yours.

Time Line

We strongly recommend you start early. To help "guide" you through this lab, we have provided you some key milestones!

M	T	W	R	F	S	S
	23	24	25	26	27	28
29	30	31	1	2	3	4
5	6	7	8	9	10	

- Jan 23rd - Assignment is released. Start reading, and planning out your server
- Jan 24th - Discussion of the project in recitation, please complete what is listed under 1/23 by then.
- Jan 29th – *smtpverify* should be functioning
- Feb 2nd – *smtpsend* should be functioning, except for the DNS stuff
- Feb 6th - Your server should be complete. This allows you two days to relax, test, document, clean-up, and make it a project "That you're proud of"
- Feb 8th 11:59pm - Your code should be handed in.
-

Evaluation

This section, at present, is purely tentative. We'll let you know promptly if anything firms up differently.

- Core networking : 30 points

  This grade is intended to reflect your ability to write the "core" networking code that deals with creating sockets, and reading/writing from them. We will look mostly at *smtpverify* for this credit. Credit can be received here even if the *smtpverify* doesn't produce useful results, as long as it produces results on the client that originated on the server.

- *smtpverify* functionality: 15 points

  In this section, we will check how well you read, interpreted, and implemented the SMTP protocol as necessary for EXPN, as well as the functionality of *smtpsend*. As a result, good code documentation will help ensure you get full credit for this section.

- SMTP protocol as necessary for *smtpsend* to send a message: 15 points

  In this section, we will check how well you read, interpreted, and implemented the SMTP protocol as necessary to properly send a message with *smtpsend*. Any message will do. As a result, good code documentation will help ensure you get full credit for this section.

- *smtpsend* functionality: 15 points

  In this section, we will grade all of the details of a properly functioning mail client, including –the –v flag and the like, except for the functionality involving DNS.

- *smtpsend* DNS-related functionality: 15 points

  In this section, we will grade the DNS-related functionality, most notably that described by RFC 976.

- Style/Robustness/Qualitative Quality: 10 points

  This section is designed to catch all of the "small-ish" stuff. It includes minor bugs that don't significantly reduce the value of the tool or impeach the understanding or skill of its authors. It includes the presentation of the code, including internal and external documentation and formatting. It also includes sub-optimal, decisions about the strategy or techniques within the implementation.

Hand-In

We'll discuss this in recitation during the week it is due. Until then, keep in mind that we'll want your source files, makefile, and README file. Please also ensure that the executables and arguments match those discussed in this document.

Building

Your project must include a *makefile* called *Makefile*. We will build a binary from your source code using the *makefile* and GNU *make*. The *makefile* for this project should be simple. If you need help creating the *makefile*, everything you need to know (and much much more) about GNU make can be found in the GNU make manual. We will build your program by executing "make clean", and then "make". Please test this before handing in to avoid unnecessary deductions.

Your makefile should run gcc with option -Wall and should produce no errors.

Hints

Depending on your previous experience, this project may be substantially larger than your previous programming projects. You can expect up to 750-1000 lines of code, possibly more, possibly less. With that in mind, this section gives suggestions for how to approach the project. Naturally, other approaches are possible, and you are free to use them.

- Start early. Don't wait until the last week to start working.
- Read the RFC. RFCs are written in a style that you may find unfamiliar. However, it is wise for you to become familiar with it, as it is similar to the styles of many standards organizations. You may well need to reread critical sections a few times for the meaning to sink in.
- Get started by writing a simple program that receives messages from a single socket and repeats it back to the sender. It won't do anything useful, but it will get you used to working with `socket`, `bind`, `connect` and friends.
- You should familarize yourself with the concept of blocking syscalls. Basically, calls like `read, write, recv, send, printf`, and many others, block, or halt, your program until the action the call is performing completes. For example, `read` will wait until it has received at least some data to return. If the other side never sends your program any packets, `read` will never return.
- Break your code up into multiple files based on functionality. This makes testing, grading, and coding easier.
- Follow the timeline, but feel free to move faster than it. If you are running behind, please come to office hours for help.
- Even simple protocols such as SMTP can illustrate complex and counter-intuitive problems. For example, see the discussion of the "Sorcerer's Apprentice" bug in Section 4.2.3.1 of RFC 1123.