CIS 505: Software Systems

Spring 2017

Assignment 2: Email servers

MS1 due February 15, 2017, at 10:00pm EST
MS2+3 due March 3, 2017, at 10:00pm EST

1 Overview

For this assignment, you will build two simple multithreaded servers: a SMTP server, which can be used to send emails using a normal mail client, and a POP3 server, which can be used to receive emails. SMTP and POP3 are both relatively simple, text-based protocols, and we are going to leave out some of the more complicated features (such as encryption and secure authentication) to reduce the amount of coding that will be necessary.

The assignment consists of three milestones. For the first milestone, you will write a simple multithreaded server that implements only two very basic commands. Then, for the second and third milestone, you will create two variants of this server that implement SMTP and POP3, respectively. This should save you some work because the two protocols share the same basic structure, so all the basic infrastructure (connection handling, buffering, threading, etc.) should be the same. Note: The first milestone is easier than the others, so you should keep working once you finish it – don't wait for the MS1 deadline before you start on MS2!

As with the previous assignment, we have put together a small amount of code to help you get started. As in the first assignment, you can download this code by doing a `git pull` in your `git` directory; this should produce a new directory called HW2, which should contain a basic Makefile, a README file, and placeholders for the servers, as well as a `test` folder that contains the testers to help you test your servers. As before, we strongly encourage you to periodically check your modifications into git and to push them to the git server.

2 Milestone 1: Multithreaded echo server

As a first step, you will build a simple multithreaded server. Your server should open a TCP port and start accepting connections. When a client connects, the server should send a simple greeting message and then start a new pthread, which should read commands from the connection and process them until the user closes the connection. For now, you only need to support two very simple commands:

1. `ECHO <text>`, to which the server should respond `+OK <text>` to the client; and
2. `QUIT`, to which the server should respond `+OK Goodbye!` to the client and then close the connection.

Each command is terminated by a carriage return (`\r`) followed by a linefeed character (`\n`). The greeting message should have the form `+OK Server ready (Author: Linh Thi Xuan Phan / linhphan)`, except that you should fill in your own name and SEAS login.

One important challenge with building a server like this is the fact that the input from the client can arrive in small pieces, and not necessarily as an entire line. For instance, if the client sends `ECHO foo`, the first `read()` call in your server could return `ECH`, the next one `O fo`, and the final one `o`. Also, the linefeed character will not necessarily occur at the end of the data returned by `read()` - it could occur in the middle. To handle this, your server should maintain a buffer for each connection; when new data arrives, it should be added to the corresponding buffer, and the server should then check whether it has received a full line. If so, it should process the corresponding command, remove the line from the buffer, and repeat until the buffer no longer contains a full line.

Your server should support multiple concurrent connections. You may assume that there will be no more than 100 concurrent connections; however, your server should not limit the number of sequential connections. Commands should be treated as case-insensitive; if the server receives a command it does not understand, it should return `-ERR Unknown command`. If the server limits the length of a command, the limit should be at least 1,000 characters. The server should also properly clean up its resources, e.g., by terminating pthreads when their connection closes; if the user terminates the server by pressing Ctrl+C, the server should write `-ERR Server shutting down` to each open connection and then close all open sockets before terminating.

The server should support three command-line options: `-p <portno>`, `-a`, and `-v`. If the `-p` option is given, the server should accept connections on the specified port; otherwise it should use port 10000. If the `-a` option is given, the server should output your full name and SEAS login to `stderr`, and then exit. If the `-v` option is given, the server should print debug output to `stderr`. At the very least, the debug output should contain four kinds of lines:

1. `[N] New connection` (where `N` is the file descriptor of the connection);
2. `[N] C: <text>` (where `<text>` is a command received from the client and `N` is as above);
3. `[N] S: <text>` (where `<text>` is a response sent by the server, and `N` is as above); and
4. `[N] Connection closed` (where `N` is as above).

You should feel free to output other kinds of debug output (but only if the -v option is given!).

2.1 Testing your server with Telnet

To test your server, you can open a terminal and connect to it using the telnet program. For instance, if the server is running on its default port (10000), you could run `telnet localhost 10000` and then start typing commands. Below is an example transcript that was produced by a server with the `-v` option, with two different connections from two different `telnet` instances:

```
[4] New connection
[4] C: ECHO Hello world
[4] S: +OK Hello world
[4] C: TEST unsupported
[4] S: -ERR Unknown command
[5] New connection
[5] C: ECHO Nice to meet you!
[5] S: +OK Nice to meet you!
[5] C: QUIT
[5] S: +OK Goodbye!
[5] Connection closed
[4] C: QUIT
```

```
[4] S: +OK Goodbye!
[4] Connection closed
```

2.2 Testing your server with the tester

There are certain subtle bugs, such as servers sending some extra zero bytes with each response, that you cannot easily reproduce with telnet. To help you better test your solutions, we have put together a little tester, which has been checked into your git repository (in the `HW2/test` folder). When you run it, the tester will try to connect to your server on port 10000, so be sure to run your server with `./echoserver -p 10000`. If all goes well, you should see something like this:

```
S: +OK Server ready (Author: Linh Thi Xuan Phan / linhphan)<CR><LF> [OK]
C: ECHO Hello world!<CR><LF>
S: +OK Hello world!<CR><LF> [OK]
C: BLAH<CR><LF>
S: -ERR Unknown command<CR><LF> [OK]
C: ECH
C: O blah<CR><LF>EC
S: +OK blah<CR><LF> [OK]
C: HO blubb<CR><LF>ECHO xyz<CR><LF>
S: +OK blubb<CR><LF> [OK]
S: +OK xyz<CR><LF> [OK]
C: QUIT<CR><LF>
S: +OK Goodbye!<CR><LF> [OK]
```

If something goes wrong, the tester will either abort with an error message or, in the case of smaller bugs or typos, put a little annotation in the brackets instead of the `OK`.

It is very important that you run this tester on your echo server *at least* once, and that you fix any problems it reveals, before you submit your solution. Since the echo server is meant as a foundation for the next two milestones, which use the same server structure but more complex protocols, having a working echo server will help save you a lot of time and headaches with the other milestones. Please note also that the tester is meant only as a starting point – you should add your own tests! For instance, you may want to test cases with multiple connections, opening and closing lots of connections (to see whether the cleanup works), etc.

3 Milestone 2: SMTP server

Next, you should make a copy of your server and change it to implement the SMTP protocol. The goal of this milestone is to develop a SMTP server that works with Thunderbird. The full protocol specification is in RFC 821 (https://tools.ietf.org/html/rfc821), but for the purposes of this assignment, we are going to simplify it a little.

Basically, SMTP works just like the simple echo server you implemented above, but it has different commands, and its responses have a slightly different format. In SMTP, the server's responses start with a three-digit code (e.g., 250) followed by a textual response (e.g., 'OK'). When the client first opens a connection, the server should send a 220 'service ready' response, which starts with the domain name (in our case, localhost) and a greeting message. It should then accept one of the following commands:

- `HELO <domain>`, which starts a connection;
- `MAIL FROM:`, which tells the server who the sender of the email is;

- `RCPT TO:`, which specifies the recipient;
- `DATA`, which is followed by the text of the email and then a dot (.) on a line by itself;
- `QUIT`, which terminates the connection;
- `RSET`, which aborts a mail transaction; and
- `NOOP`, which does nothing.

For anything else (in particular, `EHLO`), your server should return an error code.

On the command line, your server should accept the name of a directory that contains the mailboxes of the local users. Each file in this directory should store the email for one local user; a file with the name `user.mbox` would contain the messages for the user with the email address `user@localhost`. The files should initially be empty (size zero; use the UNIX `touch` command to create these!), and new emails should be appended at the end of the file. The file should follow the mbox format - that is, each email should start with a line `From <email> <date>` (Example: `From <linhphan@localhost> Mon Aug 22 23:00:00 2016`), and after that, the exact text that was sent by the client (but without the final dot). You do not need to worry about emails that contain lines that start with `"From "`.

As before, your server should support the `-p`, `-a`, and `-v` options. The debug output should be as specified above, but the default port number should now be 2500. (The default for SMTP is 25, but port numbers below 1024 require root privileges to access.) Also, as before, your server should support multiple concurrent connections, use pthreads, and clean up all resources properly when it is terminated.

You are explicitly *not* required to implement authentication, encryption, or mail forwarding (in the sense of delivering mail to a non-local user). If the `RCPT TO:` command specifies a recipient whose email address ends in something other than `@localhost`, your server may return an error message.

3.1 Testing your solution

The VM image contains the Thunderbird email client, which you can use to test your server. You should configure an email account with server name `localhost`, port number 2500, connection security "None", and authentication method "No authentication". You can then write an email to another user on localhost and send it while the server is running. If the server does not receive the connection at all, verify that the server name is really localhost (and not 'localhost' or something like that), and check the port number. If the server is using unusual commands that are not mentioned above, check the security and authentication method settings. You may see an attempt to use the `EHLO` command at the beginning, but Thunderbird should fall back to `HELO` once your server returns the appropriate error code.

To create an 'empty' `.mbox` file for testing, you can use the Unix `touch` command. For instance, you could run the following commands:

```
mkdir mailtest
touch mailtest/linhphan.mbox
touch mailtest/bcpierce.mbox
touch mailtest/zives.mbox
```

This would create a directory called `mailtest` that contains three empty `.mbox` files for users `linhphan`, `bcpierce`, and `zives`. Then, if you run the SMTP server with this directory as an argument, it should accept mail for linhphan@localhost, bcpierce@localhost, and zives@localhost, but reject everything else. (The incoming mail would then be appended to the corresponding `.mbox` file.)

The `test` directory contains another automated tester for SMTP, similar to the one that was described in Section 2.2. This tester accepts a single command-line argument, which specifies the port number that your SMTP server is running on. The output is similar to what was described in Section 2.2.

4 Milestone 3: POP3 server

Now you should make another copy of your echo server and change it to implement the POP3 protocol. The full protocol specification is again available as an RFC - in this case, RFC 1939 (https://tools.ietf.org/html/rfc1939) - but we are again going to simplify things a little bit, so it'll be easier for you to implement.

The POP3 mode of operation is very similar to SMTP - the client issues four-letter commands, the server sends responses in a specific format, etc. - but the details are different. With POP3, the responses do not start with three-digit codes, but rather with `+OK` or `-ERR`, depending on whether the command succeeded or failed. The initial greeting message should be `+OK POP3 ready [localhost]`, and you should support the following commands:

- `USER`, which tells the server which user is logging in;
- `PASS`, which specifies the user's password;
- `STAT`, which returns the number of messages and the size of the mailbox;
- `UIDL`, which shows a list of messages, along with a unique ID for each message;
- `RETR`, which retrieves a particular message;
- `DELE`, which deletes a message;
- `QUIT`, which terminates the connection;
- `LIST`, which shows the size of a particular message, or all the messages;
- `RSET`, which undeletes all the messages that have been deleted with `DELE`; and
- `NOOP`, which does nothing.

Some of these commands have required or optional parameters, and some of them return additional information after the initial `+OK` or `-ERR` line; for details, please see the RFC. If your server sees a command that is not in the above list (e.g., `CAPA`), it should return `-ERR Not supported`.

On the command line, your server should accept the name of a directory that contains the users' mbox files, in the same format and with the same naming convention as above. Also, the `-p`, `-a`, and `-v` options should be supported, but the default port should now be 11000. The unique IDs that are needed for the `UIDL` command should be computed as MD5 hashes over the text of the corresponding message. (For this, please use the `MD5_Init`, `MD5_Update`, and `MD5_Final` functions from `libcrypto`; example code should be included in your `HW2` directory.) The password of each user should be `cis505`.

Note that the POP3 protocol has been extended many times, e.g., with the `CAPA` command (RFC 2449), and SASL for authentication and security (RFC 5034). You are not required to implement these extensions, but you may find that some mail clients other than Thunderbird (e.g., Apple Mail) do not work properly without them. So please do use Thunderbird for testing!

4.1 Testing your solution

For initial testing, it is best to use `telnet` and the POP3 tester in the `test` directory, rather than a full email client. When your server is running, you should be able to connect to it using `telnet localhost`

11000, and you can then issue commands manually. Once you are satisfied that your server (mostly) works, you should start using Thunderbird for testing. You should configure an account with email address <yourlogin>@localhost (for me, it would be linhphan@localhost), server name localhost, server port 11000, username <yourlogin> (for me, linhphan), connection security "None", and authentication method "Password, transmitted insecurely". If your SMTP and POP3 servers are both running (and configured with the same directory), you should now be able to send an email to yourself and see it appear in your inbox. We strongly suggest that you keep the -v option on during testing, so you can see what the email client is saying to your server, and how your server is responding.

5 Submitting your solution

The submission process is the same for MS1 and for MS2+MS3 – the only difference is that for MS1, you only need to finish the echo server (echo.cc), whereas for MS2+3, you need to finish the other two servers (pop3.cc and smtp.cc). We will not re-grade your echo server for MS2+MS3.

Before you submit your solution, please make sure that:

- ☐ Your solution compiles properly.
- ☐ Your code contains a reasonable amount of useful documentation (required for style points).
- ☐ You have completed *all* the fields in the README file.
- ☐ You have checked your final code into the Git repository *before* you submit.
- ☐ You are submitting a .zip file (e.g., one created by running make pack from the HW2 folder)
- ☐ Your .zip file contains all of the following:
  - ☐ all the files needed to compile your solution (especially all .cc files!);
  - ☐ a working Makefile; and
  - ☐ the README file, with all fields completed
- ☐ Your .zip file is smaller than 250kB. Please do not submit binaries or input files!

As with HW1, you must submit your .zip file online, using the web submission system - we will not accept submissions via email, or in any other way. If you need an extension and still have jokers available, you can use the submission system to extend the deadline.

6 Extra credit

6.1 Mail forwarding (+15%)

For this extra-credit task, you should extend your SMTP server to accept mail for non-local users. These emails should be stored in a file called mqueue within the mailbox directory. Also, you should write an additional program that reads the contents of this file and attempts to deliver the emails within it. This program should accept the name of the mailbox directory on the command line, and it should support the -v and -a options as above. (Notice that you need to look up the name of the destination's mail server in DNS, using a MX record; you can use the res_query function from the libresolv library to do that.) Once invoked, your program should attempt to deliver each mail in mqueue; if an email is successfully delivered, it should be removed from the mqueue file, so that only the undeliverable emails remain in the file after the program terminates.