CS 164 – Winter 2009 Term Project
Writing an SMTP server and an SMTP client
(Receiver-SMTP and Sender-SMTP)
Due & Demo Date (Friday, March 13th)

YOUR ASSIGNMENT

Your assignment is to write an SMTP (Simple Mail Transfer Protocol) server and an SMTP client. You will use telnet to send a message to the SMTP server. The SMTP server will write all the messages it receives to a queue directory. Each message must result in a new file in that directory. All server activity must be written to a log file.
The server could (but not mandatory) run as a daemon, and handle the TERM signal correctly. The server should also have a maximum number of allowable simultaneous connections at any time-five is a reasonable number for a small server.
The SMTP server will then have to send the messages to a "real" SMTP server, like mail.cs.ucr.edu. For that reason, you will write an SMTP client. The SMTP client will have to pop the queue sequentially, parse the mail message and send it to mail.cs.ucr.edu.

DELIVERABLES:

You must submit the following:

All source code.
Your source code should compile without ANY warnings. (Use the –Wall option.) You will lose one point per warning. You should write the program in C. (Please contact the TA if you are interested in implementing the project in another programming language.)

Documentation.
There should be a file called README. It should include a general outline of how your program works, and any special comments or notes that you need to share. Please keep it straight forward and easy to read. The README file should be straight ASCII text.

GRADING

- **70% for program functionality/correctness**
- **20% for documentation**
- **10% for programming style**

BACKGROUND

You are probably using e-mail every day, but do you know how it works? After this assignment, you will have a fair understanding of the underlying protocol that makes e-mails possible: SMTP. The RFC for SMTP can be found at http://www.freesoft.org/CIE/RFC/821/

You don’t have to worry about things like mail relaying in this assignment-this is left for real SMTP servers (called MTAs, or Mail Transfer Agents). What you will have to worry about are the SMTP commands and building a Finite State Machine (FSM) that implements them, in both the server and the client.

You can see a real mail server in action, by doing a:

telnet mail.cs.ucr.edu 25

And you should get something like:

Trying 138.23.169.5...
Connected to barrichello.cs.ucr.edu.
Escape character is '^]'.
220 barrichello.cs.ucr.edu ESMTP Postfix

And from here on you can start issuing commands and see the replies.

Postfix is a good SMTP for UNIX, but, the most widely used SMTP server in the world is Sendmail. One of the nice features is that it is quite verbose on its replies (this is the very least of its nice features, actually, but anyway). To see a Sendmail server in action, type:

telnet achilles.noc.ntua.gr 25

And you should get something like

Trying 147.102.222.210...
Connected to achillles.noc.ntua.gr.
Escape character is '^]'.
220 photonics.ece.ntua.gr ESMTP Sendmail 8.10.0/8.10.0; Wed, 9 May 2001 08:35:38 +0300

(Yes, I know, it is in Greece so the connection might be a little slow…Also, if you tried photonics.ece.ntua.gr, it would not connect, this is due to the access lists on the NTUA campus border router)

Try issuing commands in both servers and see their replies.

THE TWO PARTS OF AN SMTP SERVER

An SMTP server is broken into two parts. The first part is the Receiver-SMTP. This part is the one that actually acts as a server- it is listening on port 25, and accepts connections as soon as they come. When a connection (session) terminates successfully, the R-SMTP will check to see the recipients of the message. If there is a local recipient, it will put the mail at that user’s mailbox. If not, it will put the message in the queue, leaving the forwarding job to the second part.
The second part is the Sender-SMTP. This is the part that acts as a client. The job of the S-SMTP is to get the files from the queue and send them to their final destination. The S-SMTP will connect to the R-SMTP of the remote network-the network where the recipient of the message is located- and send the message. The R-SMTP of the remote network will see that the message is for a local user and put it on the users’s mailbox.

PART I-THE SERVER SIDE (Receiver-SMTP)

SMTP Commands

Your MTA should implement the following commands:

- HELO/EHLO
- MAIL
- RCPT
- DATA
- HELP
- RSET
- QUIT

Moreover, it should understand (not reply with "command unrecognized") but not implement, the following commands:

- EXPN
- VRFY
- SOML
- SAML

After receiving a command from the mail client, the MTA must reply using a reply code. The following replies must be implemented:

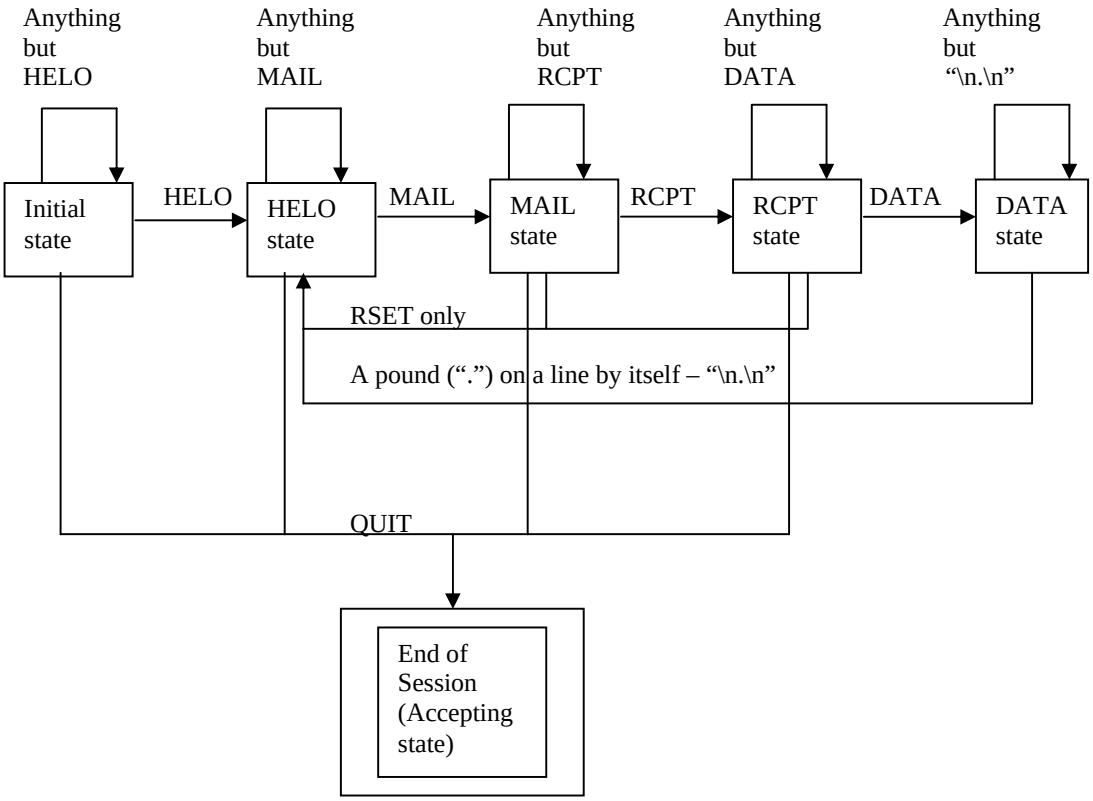
- 500 Syntax error, command unrecognized
  [This may include errors such as command line too long]
- 501 Syntax error in parameters or arguments
- 502 Command not implemented
- 503 Bad sequence of commands
- 214 Help message
  [Information on how to use the receiver or the meaning of a particular non-standard command; this reply is useful only to the human user]
- 220 <domain> Service ready
- 221 <domain> Service closing transmission channel
- 421 <domain> Service not available, closing transmission channel
  [This may be a reply to any command if the service knows it must shut down]
- 250 Requested mail action okay, completed

Simply speaking, a reply starting with a 4 or a 5 represents an error. A reply starting with a 2 represents success.

Sequence of commands-SMTP state diagram

If you paid attention to the reply codes, you would notice that there is a reply called "bad sequence of commands". What this means is that a mail client cannot just issue any command at any time (well, technically, it can but it will receive a "503 bad sequence"☺).

A simplified state diagram of SMTP follows:



Basically, what the above state machine means is the following:

1. After connecting to the server, the client must send the HELO command to move the server to the HELO state. Any other command will keep the server on the Initial state
2. After HELO, the client issues the MAIL command that specifies the sender of the message, to move the serve to the MAIL state. Any other command will leave it to the HELO state.
3. After the MAIL command, the client will issue the RCPT command, specifying the recipient of the message in order to move the server to the RCPT state. Any other command will keep the server at the MAIL state.
4. After RCPT, the client sends the DATA command and moves the server to the DATA state, any other command will keep the server on RCPT state.
5. The DATA state is different from the rest. In the DATA state, the client sends the actual message, including the headers, subject and body of the message. The message terminates with a pound (".") on a line by itself (that is, "\n.\n", or "\r\n.\r\n"). When the client sends the special termination character, the server puts the message on the queue, and gets back on the HELO state.
6. At any state, except the DATA and Initial states, a RSET command will put the server back to the HELO state.
7. At any state, except the DATA state, the QUIT command will terminate the session, thus moving the server to the "accepting" state-and terminating the connection.
8. The HELP command will **never** move the server to a new state.
9. The server should terminate the connection ONLY after it has received a QUIT command. The server must also reply to a QUIT command.

Based on the above state diagram, the return code that the server sends to the client after every command becomes clear:

- When a client connects to the server, the server should send a “220 Service ready” greeting message. In essence, the server initiates the dialogue.
- Any successful command must return a”250 Ok <optional verbose data>”
- Any command that is out of order, like a RCPT before a MAIL, or a DATA before a MAIL, or even a HELO after a HELO must return a “503 Bad sequence of commands”
- Any syntax error on the command’s parameters must return a “501”.
- Any unrecognized command must return a “500”.
- All unimplemented commands must return a “502”.
- The HELP command always returns “214”.
- The QUIT commands always returns “221”.
- The server returns “421 Service unavailable” when the maximum number of simultaneous connections have been reached. The server terminates the connection immediately after a 421 message-it needs not wait for a QUIT command.

So now we can see that a “250” reply will move the SMTP state machine to the next state-with the exception of 221 that moves the machine to the accepting state and terminates the session. (There is also another exception, revealed in the next section). Any other return code will leave the machine at the same state. This information is also useful when designing the client’s state machine.

Commands

Generally speaking, commands can be issued only once, at the lifetime of a session, unless the machine has been reset, one way or another (that is, either with an explicit call to RSET or a successful completion of the DATA command). The only exception to this rule is the RCPT command. When the server is at the RCPT state, subsequent RCPT commands will not return an error-rather than that, the server will reply with a “250 Recipient OK”. This happens because a mail message can have multiple recipients. However, to keep things simple, you should only handle the first recipient-the rest can be silently discarded.

The correct syntax of the commands that you will implement is the following:

- HELO <SP> <domain> <CRLF>
- MAIL <SP> FROM:<full_address> <CRLF>
- RCPT <SP> TO:<full_address> <CRLF>
- DATA <CRLF>
- RSET <CRLF>
- HELP <CRLF>
- QUIT <CRLF>

<SP> denotes a blank space. <CRLF> is the “carriage return-line feed” character sequence. It basically means that you should terminate your commands with “\r\n”, although “\n” will be enough, for UNIX. <full_address> means that you should give the address of the sender and the recipient in the form: “thanos@cs.ucr.edu”. In essence, you must include the “@”.
All commands are case INSENSITIVE. The only case-sensitive portion of the commands is the <full_address> field.
The HELP command should just print the list of available commands in your server.

I STRONGLY suggest you telnet to an SMTP server (see “Background” section) and issue those commands that you have to implement. See what the server returns. Use an incorrect syntax and see what it returns. Experiment extensively.

Buffers and writing to the queue

A fully functional SMTP server has a few buffers, but you only need to worry about 3 of them in this assignment. The three buffers are: The “sender” buffer, the “receiver” buffer, and the “data” buffer. You don’t need to worry about forwarding paths and reverse paths.

If a DATA command completes succesfully, you have all three buffers filled. In that case, you should write all three buffers to a file. The format should be the following:

From: <sender_address>
To: <recepient_address>
(a new line by itself)
<the contents of the DATA buffer>

This will make your file easy to parse by your client.

A sample SMTP session

What follows is a sample SMTP session.

```
Trying 147.102.222.210...
Connected to achilles.noc.ntua.gr.
Escape character is '^]'.
220 ntua.gr ESMTP Sendmail 8.9.3/8.9.3; Sat, 12 May 2001 05:15:11 +0300 (EET DST
)
HELO localhost
250 ntua.gr Hello hill.ucr.edu [138.23.169.9], pleased to meet you
MAIL FROM: thanos@cs.ucr.edu
250 thanos@cs.ucr.edu... Sender ok
RCPT TO: thanos@photonics.ece.ntua.gr
250 thanos@photonics.ece.ntua.gr... Recipient ok
DATA
354 Enter mail, end with "." on a line by itself
Testing 1,2,3
.
250 FAA17710 Message accepted for delivery
QUIT
221 ntua.gr closing connection
```

Notice the sequence of commands and the verbosity of the server’s replies. Although you do not have to be so explicit in your replies- after all, you rarely telnet to the SMTP server directly to send your mails-, it is nice to do so.

PART II- THE SMTP CLIENT

The SMTP client (Sender-SMTP) takes over after the SMTP server has done its job. The role of the client is to send the mails that cannot be delivered locally, to the final destination. For the purpose of this assignment, we assume that ALL mails will have to be forwarded- in other words, you do not have to parse the recipient buffer in order to decide whether the recipient is a local user or not.

Once you have completed the R-SMTP, making the S-SMTP is easy- you know exactly what to send to the server! However, this does not mean that the S-SMTP is stateless-it also has a state machine. The FSM (finite state machine) of the S-SMTP is fairly simple, and quite similar to the one of the R-SMTP. However, the machine state changes as a result of the reply codes. So the client sends the command, and based on the reply code, it either moves to a new state or it stays on the same state, and sends the correct command this time. Remember that all reply codes that start with 4 or 5 are errors-the client FSM should not change state when it gets those reply codes. The reply codes starting with 2 are successes. The client FSM should change state, unless the command issued was the HELP command, in which case, the server has not changed its state, so the client should not change state either.

For the purpose of this assignment, the S-SMTP part should be a single process. This process will have to sequentially pop messages out of the queue (remember that messages are written to a directory after a successful SMTP session). It should parse the messages and find the sender part, receiver part and the data part-these will be useful when the S-SMTP has to issue SMTP commands to the remote R-SMTP. The next step is to connect to the R-SMTP of the remote host. In our case, this will ALWAYS be mail.cs.ucr.edu, regardless of the address of the recipient. Simply speaking, we are using mail.cs.ucr.edu as a **relay**.

POINTS OF CONCERN

Some points of concern for both parts of the assignment:

- The R-SMTP should be a **concurrent** server. All the techniques that you used on the second assignment apply here (fork(), wait() etc)
- You must keep a **logfile** for all sessions. This includes both the R-SMTP and the S-SMTP part. The logfile should contain a timestamp (the date), an identifier (like R-SMTP or S-SMTP), the client's IP address or name for the R-SMTP messages - or the server's IP address or name for the S-SMTP messages. It should also include information on whether the transaction was successful or not. All errors should go to the logfile as well. Please keep in mind that you should use a **single** logfile, not two separate ones. There are issues of synchronization and race conditions, so pay particular attention.
- The R-SMTP must have a maximum number of allowable simultaneous connections. For example, if there are five connections to the server currently active, the maximum number is five, and a sixth connection arrives, the R-SMTP should reply with a "421 Service unavailable" error, and then terminate the session immediately. This means that you **still have to accept the connection, send the error message and then terminate it.**
- In all other cases besides the previous one, the R-SMTP part must terminate a session only after EXPLICITLY receiving a QUIT command from the client side. When receiving a QUIT command, the server must still send a response, then terminate the connection. This also means that the S-SMTP part must ALWAYS send QUITs. Don't close the socket without sending a QUIT!
- If the queue is empty, S-SMTP must go to sleep. It should also awaken in specific intervals and check the status of the queue. A good interval is one minute, but you can use whatever you like (just don't make it an hour or so, remember you still have to demo your program to me). If the S-SMTP detects that the queue is not empty, it should process it-that means the entire queue, not just one file. It should then go back to sleep. Remember that new files can be created while the S-SMTP is processing the queue. Just because you had one file in the queue when you started processing it does't mean that there will be no more files left after you are done! Also, for the sleeping part, use sleep-don't use an infinite loop! Infinite loops waste CPU cycles.