Re-creating Mailinator in Python

November 11, 2011

Update: *February 21, 2012*
I've extended this concept into a framework for creating an intelligent email-based agent server, whereby email sent to designated inboxes get dynamic, or custom replies.

It's the same logic used by the TeamWork.io web service and I've decided to open source it on github: https://github.com/dpapathanasiou/intelligent-smtp-responder

Paul Tyma, the creator of Mailinator, once wrote about its architecture. He said that after starting with sendmail, he found it necessary to write his own SMTP server from scratch. While he never released the Java source code of his server, I wanted to see if I could re-create it using Python, since I also wanted to understand how state machines work in that language.

The Basic Server

To start, I needed some code that would listen on a specific port, and read and respond to clients. Python's SocketServer module makes this simple. Here, in a few lines, is a multi-threaded TCP server that listens on port 8888 of the local machine and echoes back what a connected client sends to it:

```
#!/usr/bin/python
```

```
import SocketServer
```

```
cr_lf = "\r\n"
```

```
class SMTPRequestHandler (SocketServer.StreamRequestHandler):
    def handle (self):
        try:
            while 1:
                client_msg = self.rfile.readline()
                self.wfile.write(client_msg.rstrip()+cr_lf) # a simple echo
        except Exception, e:
            print e
```

```
# server hostname and port to listen on
server_config = ('localhost', 8888)
```

```
if __name__ == '__main__':
    tcpserver = SocketServer.ThreadingTCPServer(server_config, SMTPRequestHandler)
    tcpserver.serve_forever()
```

Start it from a command line prompt (if the port number you choose is less than 1025, then you need to do this as root):

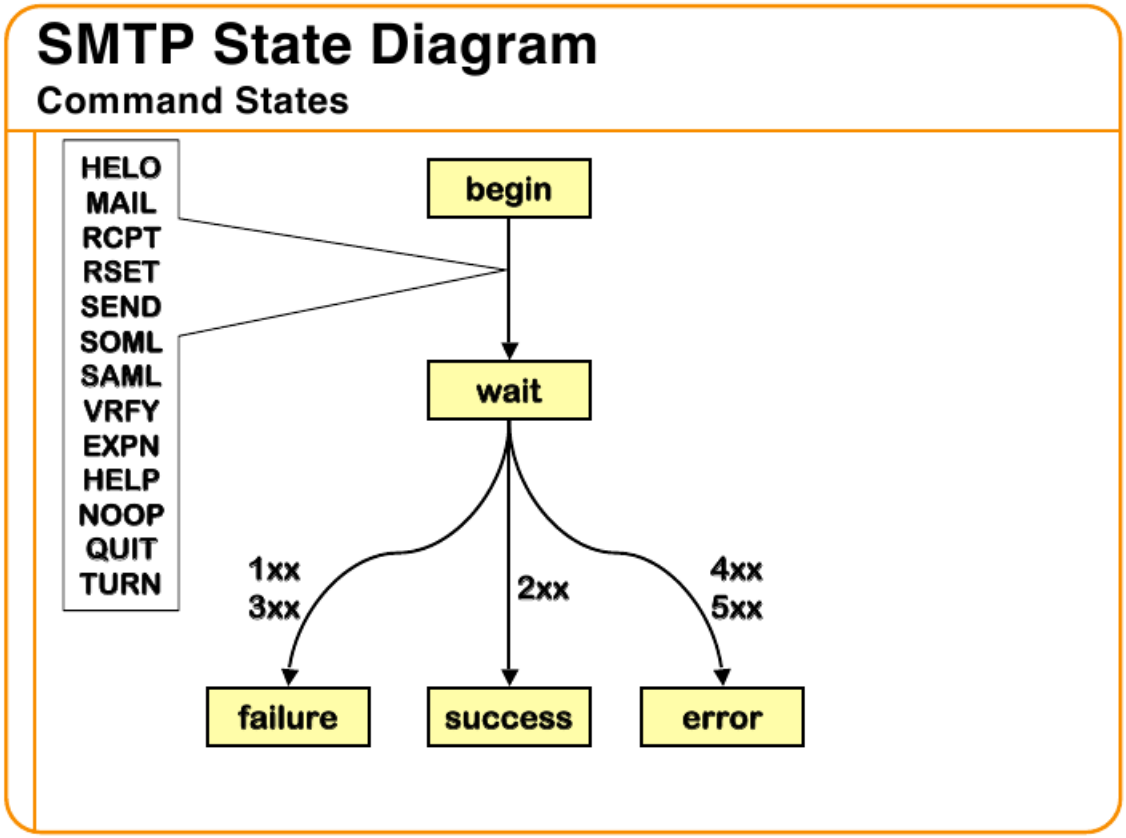
```
$ python server.py
```

And test it using telnet:

```
$ telnet localhost 8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
This is an echo
This is an echo
Ok, I get it
Ok, I get it
What next?
What next?
```

Handling SMTP

Now I needed to be able to understand and reply to SMTP requests. The protocol is fairly simple, with only a handful of commands. Each command consists of four letters, which appear at the start of the stream sent by the client, and terminated with "\r\n".



Tyma did not, however, implement the full list of SMTP commands, since RSET (Reset), VRFY (Verify), NOOP (No operation), and others are used by spammers to abuse or even take over a server, and are rarely required by legitimate email clients. The server needs to be able to handle the basic interaction, so HELO (Hello) / EHLO (Extended Hello), MAIL (Mail from), RCPT TO (Recipient To), and DATA all need to be supported. At first glance, it's tempting to try to implement it like this:

```
class SMTPRequestHandler (SocketServer.StreamRequestHandler):
    def handle (self):
        try:
            data = {}
            while 1:
                client_msg = self.rfile.readline()
                if client_msg.startswith('MAIL FROM:'):
                    data['sender'] = get_email_address(client_msg)
                elif client_msg.startswith('RCPT TO:'):
                    data['recipient'] = = get_email_address(client_msg)
```

```
                ...
```

```
                elif client_msg.startswith('QUIT'):
                    break
        except Exception, e:
            print e
```

Where `get_email_address()` *is defined as, for example, something like this:*

```
def get_email_address (s):
    """Parse out the first email address found in the string and return it"""
    for token in s.split():
        if token.find('@') > -1:
            # token will be in the form:
            # 'FROM:' or 'TO:'
            # and with or without the <>
            for email_part in token.split(':'):
                if email_part.find('@') > -1:
                    return email_part.strip('<>')
```

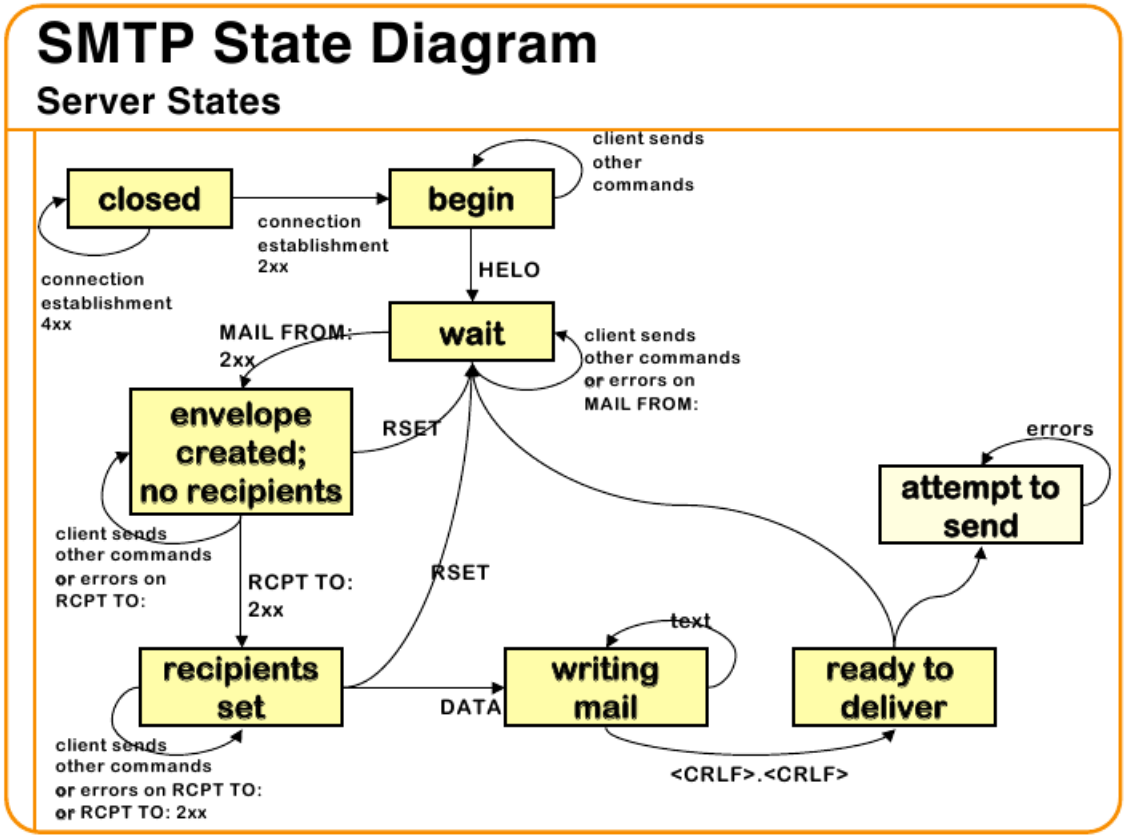
But this gets messy in a hurry. While some commands fit within the neat single-line `/^CMND rest of data\r\n/` *pattern, others do not. RCPT, for example, can be repeated multiple times, and once DATA is seen, every subsequent line must be collected until the final* `/^\.$/` *appears.*

State Machines to the rescue

A state machine provides a much better way of handling SMTP requests. In his excellent article, David Mertz defines a state machine as:

> *a directed graph, consisting of a set of nodes and a corresponding set of transition functions. The machine "runs" by responding to a series of events. Each event is in the domain of the transition function belonging to the "current" node, where the function's range is a subset of the nodes. The function returns the "next" (perhaps the same) node. At least one of these nodes must be an end-state. When an end-state is reached, the machine stops.*

And that corresponds exactly to what happens when a client interacts with an SMTP server:



Brass Tacks

Creating a state machine in Python is simple, since Python allows you to pass functions as higher-order objects. The statemachine.py implementation in Mertz's article was done in just a few lines of code. To handle each SMTP node, I defined a series of functions, one for each server response or command. Here are the function prototypes, where the `cargo` parameter is a tuple, containing both the stream from/to requests are read and responses written, and a dict of data collected from the request:

```
def greeting (cargo):
def helo (cargo):
def mail (cargo):
def rcpt (cargo):
def data (cargo):
def process (cargo):
```

The state machine is defined within the SMTPRequestHandler class like this:

```
class SMTPRequestHandler (SocketServer.StreamRequestHandler):
    def handle (self):
        try:
            m = StateMachine()
            m.add_state('greeting', greeting)
            m.add_state('helo', helo)
            m.add_state('mail', mail)
            m.add_state('rcpt', rcpt)
            m.add_state('data', data)
            m.add_state('process', process)
            m.add_state('done', None, end_state=1)
            m.set_start('greeting')
```

```
            m.run((self, {}))
        except Exception, e:
            print e
```

So that each function knows how to recognize its assigned command, I defined and compiled these regular expressions. These are created as globals, since it's more efficient to initiate them once, and have each subsequent method call use the already-existing version.

```
import re
helo_pattern = re.compile('^HELO', re.IGNORECASE)
ehlo_pattern = re.compile('^EHLO', re.IGNORECASE)
mail_pattern = re.compile('^MAIL', re.IGNORECASE)
rcpt_pattern = re.compile('^RCPT', re.IGNORECASE)
data_pattern = re.compile('^DATA', re.IGNORECASE)
end_pattern = re.compile('^.$')
```

The `greeting()` function, which begins the interaction with the client, sends a simple message and passes control to the `helo()` function. It looks like this:

```
def greeting (cargo):
    stream = cargo[0]
    stream.wfile.write('220 localhost SMTP'+cr_lf)
    return ('helo', cargo)
```

Later in the sequence, the `mail()` function, which is the first node from which data is collected (in this case, the email address of the sender), is the first to save information in the cargo's dict. It looks like this:

```
def mail (cargo):
    stream = cargo[0]
    client_msg = stream.rfile.readline()
    if mail_pattern.search(client_msg):
        sender = get_email_address(client_msg)
        if sender is None:
            stream.wfile.write(bad_request+cr_lf)
            return ('done', cargo)
        else:
            email_data = cargo[1]
            email_data['sender'] = sender
            return ('rcpt', (stream, email_data))
    else:
        stream.wfile.write(bad_request+cr_lf)
        return ('done', cargo)
```

Here, if the request is not recognized or invalid, the client sees the `bad_request` message, and the connection is closed, since control passes to the `done` end-state. I followed Tyma's example and defined `bad_request` as "`550 No such user`" (which, as he notes, is ironic, since Mailinator accepts email sent to any user). It also doesn't conform to the protocol, since I'm supposed to give different error messages at different nodes, but since clients are always disconnected after any type of invalid request, it hardly matters what they see in that scenario. If a client is well-behaved, the final method called is `process()` which decides what to do with the client's email. The data dict will contain three parameters: 'sender' (the email address of the sender), 'recipients' (a list of email addresses), and 'data' (the contents which followed the DATA command ahead of the final '.').

```
def process (cargo):
    email_data = cargo[1]
    # do something with the email_data dict here
```

```
    return ('done', cargo)
```

Basically, this is where the data can be saved to disk/db (so that it can be served by a web browser later, e.g.), MIME-parsed (to remove attachments, etc.), or just trashed (if you have reason to believe the sender is a spambot or zombie network, e.g.). Tyma describes various measures for dealing with attacks from spambots and zombies which I haven't implemented here, but would be relatively easy to add to both the `data()` *and* `process()` *functions. Obtaining the ip address of the client is done using the* `stream.client_address[0]` *attribute.*

Archived from the original at http://denis.papathanasiou.org/
Bitcoin Donate: 14TM4ADKJbaGEi8Qr8dh4KfPBQmjTshkZ2