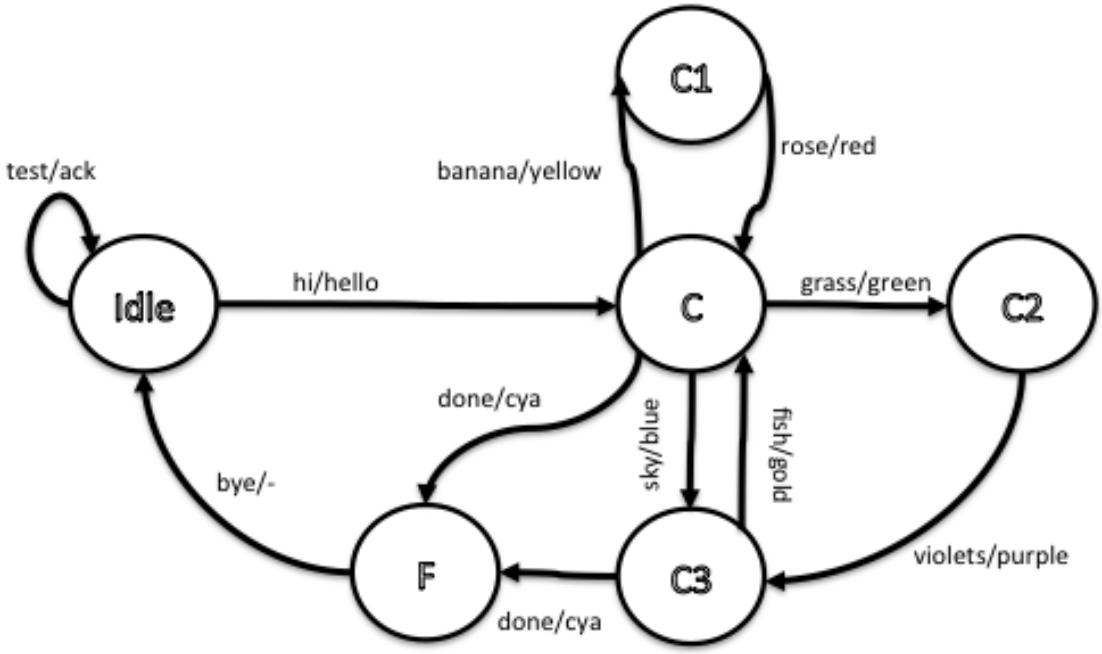

Figure 1: FSM for server

1 Assignment

Write a client which can talk to the server FSM depicted in Figure 1. An implementation of a server which speaks this protocol is hosted on eig.isi.edu, port 63681.

Your client should read a list of arguments from the commandline. Each argument is separated by white space. Example:

```
% python3.2 client.py b r b r g v f s d
```

These arguments should be interpreted as a list of local commands (input) for the client protocol as such:

Command	**Action**
b	Send “banana”
r	Send “rose”
g	Send “grass”
v	Send “violets”
f	Send “fish”
s	Send “sky”
d	Send “done”

All other actions needed by your client to talk to the server need to be done automatically. For example, when your client starts and gets a list of commands, it should establish communication with the server (by sending “hi”).

Your client code should attempt to handle errors and continue communication if possible. If the list of commands does not follow the protocol (as derived from the Server's FSM in Figure 1), or the command is not known (e.g. you get the command "x"), you should print an error message to stdout, and attempt to keep going.

For example, if the list of commands given to your client is: "b g r s d" Your client should ignore the "g" command, and then continue with the "r" command and send "rose" to the server.

At any point during the protocol you can send "done" to the server, though not all arcs indicating this are shown in Figure 1. If a series of commands does not follow protocol (for example, it does not end in "d"), your client should still attempt to follow protocol by sending a "done" and "bye" to the server.

Remember Postel's Law/the Robustness Principle: *Be conservative in what you send, be liberal in what you accept.* If you get messages from the server which do not follow the protocol (e.g. if after sending "grass" you get the message "pink" which is not in the protocol or you get the message "red" which is in the protocol, but not a valid response to "grass") you should print an error message to stdout, but still attempt to continue. In other words, your client should re-attempt sharing state by sending "grass" again.

The server will lose messages. Your client needs to handle this loss (resend messages after a timeout).

You can assume all messages will be received (at both the client and server) in the order of when they were sent. (No reordering of messages).

2 Turn In Instructions

You must submit a ZIP (.zip) or GZIP'ed TAR file (.tar.gz). These will be the *only* formats accepted. YOU MUST NAME YOUR ARCHIVE FILE USING YOUR USC USERNAME: E.G. {YOUR USC USERNAME}.tar.gz or {YOUR USC USERNAME}.zip.

Email your .zip or .tar.gz file to bartlett@isi.edu BEFORE 8pm on the deadline.

Included in your ZIP or TAR(.GZ) file should be the code for your client and a README file (see below). *Do not turn in any binaries - only text files (eg no compiled code/executables).*

Your README file is worth points, so leave yourself time to complete the README. This README file should be plain text (no other format will be accepted) and should include the following headings/sections. Each of these items should be *clearly marked*:

1. AUTHOR: Your NAME and EMAIL.
2. BUILD: Instructions on how to compile your code under UNIX. Use a Makefile if possible. If you use Python (or some other scripting language), simply put "None" for this item in your README.
3. RUN: Instructions on how to run your code. If you use the provided Python client stub, this will only be:

   ```
   % python3.2 client.py {LIST OF COMMANDS}
   ```

4. BUGS: List things you know you were not able to complete or do correctly.