CS 464/564, Assignment 2

This assignment is *individual*,
is worth *two tokens*, and
is due on *11 February at 11:59 pm*

Clients are not terribly interesting, but it is still necessary to know how to build them. Additionally some new, subtle issues about TCP (and inter-process) communication will be revealed while completing this assignment.

The assignment does all of the above by expanding the Unix shell from Assignment 1 (and transforming it into a client). In solving this assignment you can start from either your solution for the previous assignment or mine. While I do not have any recommendation on this matter, you need to make sure that you are starting from a good code base. While the behaviour considered in Assignment 1 will not be tested again in details, several features of the previous solution will be required to test the new one. An incorrect code base might produce an incorrect behaviour for the new program, which will in turn cause otherwise preventable loss of marks.

The new shell behaves in the same manner as the one developed previously, with the following *modifications*:

- All the local commands are now prefixed with the character `!` followed by a blank. In other words, a command of the form

  `!` *command with arguments*

  shall now be processes just as the command

  command with arguments

  was processed by the previous incarnation of your shell. In particular, the command

  ```
  ! exit
  ```

  terminates the shell, and commands such as

  ```
  ! & /usr/local/bin/test464
  ```

  are possible.

- Commands *not* prefixed by `!` denote requests that are to be sent to a server running on a remote machine. The name of the remote machine as well as the port number are given in the configuration file by two new lines of the form:

`RHOST` *r*
`RPORT` *p*

where *r* is an arbitrary string (the machine name) and *p* is a string representing a positive number (the port number).

Once such a "remote" command has been sent, the shell awaits for, and prints everything that the server cares to send back as a response. No terminating marker can be assumed for the response received from the server.

The server response is printed in its entirety before the shell prompt is presented back to the user, except when a remote command is prefixed by the character `&` followed by a blank (case in which the first two characters are not sent to the server). In such a case, the prompt is presented immediately to the user, and all the pieces of the server response are printed to the terminal as soon as they become available (the shell does not wait for the whole response, it prints instead the received piece of the response as soon as this piece becomes available). An appropriate message must be printed by the shell as soon as the server response has been received in its entirety.

Many standard application protocols use for compatibility reasons `\r\n` as line terminator, so your shell should accept responses with *either* `\r\n` or the normal Unix `\n` as line terminators.

- There are two new local commands (that must be issued prefixed by a `!` to take effect):

  `keepalive`
  : Before such a command is issued, the shell opens a connection to the server each time the user types in a remote command, and closes the connexion as soon as the server response has been received. Once the command `keepalive` has been issued, the shell opens the connexion to the server when a remote command is typed in, and then keeps the connection open until a `close` command (see below) is issued, upon connection termination by the server (also see below), or upon termination. In other words, all the subsequent remote commands will use the same connection to the server.

  `close`
  : Closes the connection to the server in case such a connection is open, and reverts the behaviour to the initial one (the connection to the server is closed after the completion of each remote command).

Connection termination Many times the termination of the connection is initiated by the client. It is often the case however that the server is the one terminating the connection, possibly but not necessarily at the request of the client. Assume for example a SMTP interaction, where the client is done sending emails. Once this happens the client will send a `QUIT` command, and the connection is terminated as a result. However, the server is the one that terminate the connection (as a consequence of the `QUIT` command), not the client. In other words, the client is the one that terminates the connection from the point of view of the application protocol (the `QUIT` command). However, this is immaterial from the point of view of the transport layer (TCP); at this level the connection is terminated by the server.

One way or another once the connection is terminated by one side, the respective socket at the other end becomes unconnected. Nothing in the data structure maintained by the operating

system for that socket tells you that. It is instead your responsibility to keep track of which of your sockets are connected and which ones are not. You should detect the event beforehand (when you read from the socket) and keep track of that in your code. I personally like to close the socket and set the respective socket descriptor to `-1`, but I am sure that other possibilities exist. You need to be particularly careful about this for remote background commands.

Testing You may want to do your main testing using the standard HTTP server running on `osiris.ubishops.ca` and the SMTP server running on `linux.ubishops.ca` (note however that the latter is only accessible locally, so your client must be running on `linux.ubishops.ca` to access this SMTP server). SMTP in particular offers a rich enough collection of possible commands and responses to make for a good test bed. It is your responsibility to research the application protocol for these two services. It is also your responsibility not to DoS the servers.

You can also find the server running on port 8001 on `linux.ubishops.ca` mildly interesting. This server does not implement much of an application protocol; it just send back the received line twice, with a 2-second delay between the two copies. This server is only open to the local machine so your client must be running on `linux.ubishops.ca` to access it.

Test explicitly what happens if you issue "asynchronous" remote commands (i.e., commands prefixed by `&` only) in rapid succession (i.e., issue a new command before the completion of the previous one—you may want to use copy and paste here to insure the rapid succession). Do the answers come back interleaved with each other? Why (not)? How about the case in which the command `keepalive` is in effect? Explain the behaviour in the documentation accompanying your submission.

Note that answering the testing question is integral part of your submission, and marks will be subtracted for a poor or nonexistent discussion.

What to submit Submit the sources for your shell plus appropriate documentation as described on the course's Web site. The default target of your makefile must produce an executable named `rsshell` and residing in the root directory of your submission.