Lab 3

Purpose:

- To use the TCP Transmission Layer protocol
- To implement a simple Application Layer protocol
- To implement data streaming

Handin:

You are to hand in the following files to the Lab 3 folder on the D2L Dropbox page:

- `client.py`
- `server.py`

Specification:

You are to implement a simple, streaming file transfer protocol using TCP. You must create both the client and server programs for this protocol.

The Protocol:

Our file transfer protocol consists of simple command messages, with acknowledgements sent as required. Commands shown as text are to be sent as UTF-8-encoded strings, while file sizes are to be sent as 8-byte, big-ending ("network order"), unsigned integers.

1. `GET <filename>` - this command is used by the client to retrieve the specified file from the server:
    a. ***server***: `READY`
    b. ***client***: `GET <filename>`
    c. ***server:*** `OK`
    d. ***client:*** `READY`
    e. ***server***: `<# bytes>`
    f. ***client***: `OK`
    g. ***server***: sends bytes
    h. ***server***: `DONE`

2. `PUT <filename>` - this command is used by the client to send the specified file to the server:
    a. ***server***: `READY`
    b. ***client***: `PUT <filename>`
    c. ***server***: `OK`
    d. ***client***: `<#bytes>`
    e. ***server***: `OK`
    f. ***client***: sends bytes
    g. ***server***: `DONE`

3. `DEL <filename>` - this command is used by the client to delete the specified file from the server:
    a. ***server:*** `READY`
    b. ***client:*** `DEL <filename>`
    c. ***server:*** `DONE`

Note that any kind of file should be able to be transferred -- text (like code files) or binary (like images), it shouldn't matter, you are just transmitting the raw bytes from one system to the other. The only limit on the file *size* should be on the 8-byte size value that is part of the protocol (although that's also probably the limit on your operating system, so really there's no *practical* size restriction).

Because this is a "streaming" protocol, for PUT and GET you must only send file data in 1024-byte blocks. You should never hold the entire file in memory, on either client or server sides!

The Client:

The client -- called `client.py` -- receives the action and filename the user wants to perform on the commandline. It connects to the server, performs the protocol, and exits.

The client also retrieves the server's address and port on the command line, as the first two arguments. For example, to send a local file called "`mytest.txt`" to a server on `deepblue` listening to port 12345, you would issue the command:

```
python lab2client.py deepblue.cs.camosun.bc.ca 12345 PUT mytest.txt
```

Once the client gets to the point in the protocol where it's about the send the bytes, print:

```
client sending file <filename> (<NNN> bytes)
```

Once the client receives the "DONE" message from the server, the client prints (on a separate line) "Complete" and exits. (Obviously you replace <filename> with the actual name of the file, and <NNN> with the actual number of bytes you are transferring.)

Output is similar for the GET command: just before the client is about to receive the bytes, it outputs:

```
client receiving file <filename> (<NNN> bytes)
```

And again it outputs "Complete" once it receives the "DONE" message from the server.

For the DEL command, the client just prints:

```
client deleting file <filename>
```

And once again, once it receives the "DONE" message from the server it prints "Complete" and exits.

The Server:

The server -- called server.py -- sets up a TCP streaming socket on a port specified on the commandline. When a connection occurs, it sends the "READY" message to the client, and waits for the specific command.

The server takes an optional "-v" commandline argument. If this command is given (as the second argument, after the port number), the server will give some output as it executes:

- "server waiting on port <port#>" - when listening for a connection
- "server connected to client at <address>:<port>" - when a connection is made; print the client's address and port number as specified, as in:

  ```
  server connected to client at 127.0.0.1:12345
  ```

  (Note that the actual client IP could be represented differently; just output whatever the first index of the "addr" variable is as returned from the accept() method.)

- "server receiving request: <cmd>" - when the server receives the command string such as "GET myfile.txt". Display the entire command as sent by the client, including both the command name and the filename.

- "`server receiving <NNN> bytes`" - when the server is about to receive the bytes for a file after receiving a "`PUT`" command.

- "`server sending <NNN>` bytes" - when the server is about the send the bytes for a file after receiving a "`GET`" command.

- "`server deleting file <filename>`" - when the server is about the delete the specified file.

If the server does not receive the "`-v`" command-line argument, then *no* output is to be given.

The server is intended to serve just a single client at a time. use `sock.listen(0)` before entering the `accept()` loop; and run the `accept()` loop forever, until the user kills the process externally.

Handling Errors:

Your server needs to tell the client if anything goes wrong with the request. In particular, your server needs to send the following messages:

- "`ERROR: <filename> does not exist`" - for `GET` and `DEL` commands, if the specified filename is not on the server.

- ** OPTIONAL ** "`ERROR: unable to delete <filename>`" - for the `DEL` command, if the file exists on the server, but for some reason the program was unable to delete it (for example, the server process did not have permission).

- "`ERROR: unable to create file <filename>`" - for the `PUT` command, if for some reason the server process was unable to create the file. Note that if the file already exists, it is simply overwritten (it is *not* an error); this error is more for permission-type errors or other reasons why the operating system is preventing the server process from creating this file.

The server would send this message *instead* of the `OK` or `DONE` message in that part of the protocol. That is, with `GET` and `PUT` it would come after the client sends the command and the filename. With `DEL` it would send it instead of `DONE` at the end of the protocol.

The client must handle the error messages; when received, it prints out "`server error: `" plus the error message. For example:

```
server error: file myfile.txt does not exist
```

(Note that the "`ERROR: `" prefix has been removed from the server's message when the client reports it.)

The client closes the socket connection and exits immediately upon receiving one of these errors.

If there's an error on the client side -- e.g. for some reason the client process can't create the file requested by a `GET` command -- the client must report the error to the user, close the socket (if it's been opened at this point) and exit. The error text must be one of:

- `client error: unable to create file <filename>`
- `client error: <filename> does not exist`

Note that these are error *caused by the client*; they are ***not*** errors reported by the server, because something went wrong on the server side! Also, some errors can be detected *before* the server socket has been opened; in these cases you should report the error and exit the client program without ever attempting the server connection.

No other errors need to be explicitly handled; if they occur, they are allowed to just crash the program with the generic python error messages.

Python File Access:

Python includes a way to check your permissions on a file before you go ahead and try to open/create it:

```
import os
if os.access(filename, os.R_OK):
     # I have read access
elif os.access(filename, os.F_OK):
     # file exists, but apparently I can't read from it. . .
```

etc. There's a bunch of different constant that you can check, and the `os.access()` method just returns `True` or `False` depending on whether you have the specified access or not. Do this *before* you try to open the file, so you can send the appropriately-formatted error message if access is not allowed.

(The above example is for illustrative purposes and is not necessary the exact way you need to use the `os.access()` method in this lab!)