"Accio" File using TCP (Server Simplified)

Accio Server Simplified

The simplified Accio server is another relatively simple application that waits for clients to connect, accepts a connection, sends the `accio\r\n` command, **afterwards** receives confirmation, sends the `accio\r\n` command again, receives the second confirmation, and then receives binary file that client sends, counts the number of bytes received, and prints it out as a single number (number of bytes received not including the header size).

Revisions

Not yet

Simplified Server Specification

The server application MUST be implemented in `server-s.py` Python file, accepting two command-line arguments:

```
$ python3 server-s.py <PORT>
```

- `<PORT>`: port number on which server will listen on connections. The server must accept connections coming from any interface.

For example, the command below should start the server listening on port `5000`.

```
$ python3 server-s.py 5000
```

DO NOT open files in "text" mode. All the code you write should directly work with buffer and buffer strings like `b"foobar-I-am-a-buffer-string"`. Nowhere in your program you should use `.decode('utf-8')` or `.encode('utf-8')`. If you do, you probably not going to pass many of the tests

Requirements:

- The server must open a listening socket on the specified in the command line port number on all interfaces. In order to do that, you should hard-code `0.0.0.0` as a host/IP in `socket.bind` method.
- The server should gracefully process an incorrect port number and exit with a non-zero error code. In addition to exit, the server must print out on standard error (using `sys.stderr.write()`) an error message that starts with `ERROR:` string.
- The server should exit with code zero when receiving `SIGQUIT`, `SIGTERM`, `SIGINT` signal
- The server should be able to accept multiple connections sequentially (i.e., accept one connection, process it, close connection; accept another one, etc.)
- The server should be able to handle up toe 10 simultaneous connections in a graceful fashion: not rejecting next connection, but processing it as soon as done with the previous one. To fullfil this requirement, you don't need to use multithreading, just the correct parameter to `socket.listen` call.
    - To test, you can `telnet` to your server from multiple consoles. None should be rejected, but only one should display `accio` command at a time. As soon as you done with one telnet session, `accio\r\n` should appear in another console.
- The server must assume an error if no data received from the client for over `10 seconds`. It should abort the connection and write a single `ERROR` string instead of number of bytes read.
- The server should be able to accept large enough files (`100 MiB` or more) that do not fit in memory (so, you cannot receive all in memory and then calculate the length, you should calculate the length as you receive).

Approach to implement

- Using client implementation as an example, write the required command line processing
- Add `SIGQUIT`, `SIGTERM`, `SIGINT` signal processing using signal handlers.
    - To test, add

```
not_stopped = True
while not_stopped:
    time.sleep(1)
```

In your handler, you should declare `not_stopped` variable as global and set it to `False`. If everything works correctly, your handler should be executed and application gracefully terminated. **DO NOT** use sys.exit() call, as it is

NOT graceful termination of the application and points will be deducted if you are using it.

- Add routines to initiate server socket
- Add routines to accept a connection
    - To test, you can use your client or `telnet` application. For example, `telnet localhost port`. If your server correctly accepted connection, telnet should indicate that. If not, it will be stuck in "Connecting..." stage.
    - Note that to satisfy one of the requirement, you need to use appropriate value for `socket.listen`
- Add routines to send `accio\r\n` after connection established
    - To test, use telnet again. You should see `accio\r\n` appearing after you connected to the server.
- Add routines to receive data from the client and count the amount of data received
    - To test, use your client
- Add routine to print out the amount of received data just after connection is terminated.
- Ensure that the server can accept another connection after it is done with the first one.