Programming Project for Operating Systems @ TTPU (year 2021/22)

Write a client/server application in either C++ or Python that allows to:

- Make basic operations on the files contained in a given folder of your hard disk;
- Send/receive messages to the other users currently connected to the same server.

The server will wait for TCP connections on port 2021; most of the interactions from client and server happen on that port; instead, messages will be delivered by the server to the client listening on TCP port 2022. Both the client and server can be configured statically to use the above ports (but do not hardcode this number in the code; use a macro or similar) and to read/write from a given folder from the hard disk.

The project must be submitted on GitHub; share it with professor's accounts (frisso, yujboss). Intermediate submissions are mandatory, as detailed in the last part of this document; **the project is not valid if submitted only once at the end, even if it is done correctly.**

Deadline: end of October 2021.

Note that the project is rather challenging and it requires the student to carry out a lot of work by himself, with the supervision (and help) of professors. Students will learn a lot, but learning does not come for free, or from people who sleep most of the time. **If you are unsure, do not even start the project.**

Client

The client must have two threads:

- A "sending" thread that reads command from the keyboard, transforms them into proper protocol messages, send them to the server, and receives the server response (if any);
- A "receiving" thread that is used by the server to deliver messages received from other clients.

Be careful that the output of the two threads do not interfere with each on screen (i.e., use mutual exclusion).
When the client is launched, it waits for commands typed on the keyboard. The user can type multiple commands, one after the other. Available commands are the following:

Command	Example	Description

connect USERNAME SERVER_IP_ADDRESS	connect fulvio 192.168.1.3	Connect the given username (e.g., “fulvio”) to the server with the given IP address.The username is used to send messages to other people connected to the same server. Note that when this command is issued, the client must expect a new TCP connection request coming from the server toward its local port 2022.
disconnect		Disconnect from server.
lu	lu	Shows the list of users currently connected to the server. (lu = list users)
send USER “message”	send jalol “Hello from Fulvio”	Send the message written between quotes (i.e., “”) to the specified user. If the user is not online, an error has to be shown on screen.
lf	lf	Shows the list of files that are present in the server folder. (lf = list files).
read FILENAME	read text.txt	Take the given file on the server (e.g., “text.txt”), transfer and store in the folder configured on the client.If the client already contains that file, an error has to be shown on screen.
write FILENAME	write text.txt	Take the given local file (e.g., “text.txt”), transfer and store in the folder configured on the server.If the server already contains that file, an error has to be shown on screen.
overwrite FILENAME	overwrite text.txt	Take the given local file (e.g., “text.txt”), transfer and store in the folder configured on the server.If the server already contains that file, the new file replaces the old one.
overread FILENAME	overread text.txt	Take the given file on the server (e.g., “text.txt”), transfer and store it in the folder configured on the client. If the client already contains that file, the new file replaces the old one.
append “content” FILENAME	append “this data has to be appended” text.txt	Append the given “content” to a file (e.g., “text.txt”) that must already be present on the server. If the server does not contain that file, an error has to be shown on screen.
appendfile SRC_FILENAME DST_FILENAME	appendfile myfile.txt remotefile.txt	Append the content of the local file (e.g., “myfile.txt”) to a given file (e.g.,

		remotefile.txt) that must already be present on the server. If either the client does not contain the source file, or the server does not contain the destination file, an error has to be shown on screen.
quit	quit	Terminate (gracefully) all the connections and quit the program.

Server

The server will have one main thread that accepts incoming connections from clients; each time a new connection is received, the server will create a new thread dedicated to the above connection. The thread will terminate gracefully when the client disconnects. The main thread will always stay active to receive new connections from clients.
The server will have to intercept the SIGINT signal (signal 2, CTRL+C); all existing connections must be terminated gracefully and the process should be terminated.
The server needs to avoid any racing condition, e.g., two users trying to append data to the same file, or two users trying to send messages to the same user, using the proper primitives (e.g., mutual exclusion).

Protocol

Client and server will interact with the following client-server protocol. Each command has a direction (e.g., client to server, C → S) and is followed by a return message (either OK or ERROR) in the opposite direction.
Commands (including parameters) and return codes will be sent in standard ASCII (8 bits); all commands/return codes are uppercase. Messages will also be sent in ASCII, while files will be sent in binary.
All commands have the same format:

```
COMMAND PARAMETERS\n
RETURN_CODE\n
```

Parameters are always separated by a space; messages can contain spaces, but must be enclosed within quotation marks. Command and return codes are always terminated by a “new line” character (\n).

Command	Direction	Example	Description
CONNECT USERNAME	C → S	CONNECT fulvio	Similar to the client command. This command should be the first one of the connection; if

			not, any other command will return an error. Returns: OK if the command succeeded, ERROR otherwise (e.g., the user already exists).
DISCONNECT	C → S	DISCONNECT	Client asks to disconnect. Returns: OK if the command succeeded, ERROR otherwise.
DISCONNECT	S → C	DISCONNECT	Server asks the client to disconnect. The connection toward the client is immediately terminated. No return code. This command is issued on the second TCP connection (port 2022).
LU	C → S	LU	Similar to the client command. Returns: the list of users, separated by spaces, terminated by a new line.
MESSAGE USER\n MSGSIZE MSGDATA	C → S	MESSAGE jalol\n 23456 datacontent	Similar to the client command. After the first line (terminated by \n") the client sends the size of the message, a space, then the message content. The message can contain any character; the transfer finishes when the declared number of bytes have been transferred (without any trailing "\n"). Returns: the server will reply either with OK or ERROR (e.g., user does not exist).
MESSAGE\n MSGSIZE MSGDATA	S → C	MESSAGE\n 23456 datacontent	Similar to the previous message, but used by the server to send the data to the client. Same protocol. Returns: nothing.
LF	C → S	LF	Similar to the client command. Returns: the list of files, separated by spaces, and terminated by a new line. File names cannot contain spaces.
READ FILENAME	C → S	READ text.txt	Similar to the client command. Returns: OK if the command succeeded, ERROR otherwise (e.g., the file does not exist).

			The file is returned after the response (see next line)
FILESIZE FILEDATA	S → C	23456 filecontent	After the return code, the server returns the size of the file, a space, then the file content. The file can contain any characters; the transfer finishes when the declared number of bytes have been transferred (without any trailing “\n”).
WRITE FILENAME	C → S	WRITE text.txt	Similar to the client command. Returns: OK if the command succeeded, ERROR otherwise (e.g., the file already exists). The file is sent by the client after getting the return code (see next line).
FILESIZE FILEDATA	C → S	23456 filecontent	After the return code, the client sends the size of the file, a space, then the file content. The file can contain any characters; the transfer finishes when the declared number of bytes have been transferred (without any trailing “\n”). The server will finish the transfer either with OK or ERROR (e.g., disk full).
OVERWRITE FILENAME	C → S	OVERWRITE text.txt	Similar to the client command. The protocol used is the same as the one used for WRITE.
OVERREAD FILENAME	C → S	OVERREAD text.txt	Similar to the client command. The protocol used is the same as the one used for READ.
APPEND FILENAME	C → S	APPEND text.txt	Similar to the client command. Returns: OK if the command succeeded, ERROR otherwise (e.g., the file does not exist). The data is sent by the client after getting the return code (see next line).
DATASIZE DATA	C → S	23456 filecontent	After the return code, the client sends the size of the data, a space, then the actual content. The content can contain any characters; the transfer finishes when the

			declared number of bytes have been transferred (without any trailing “\n”). The server will finish the transfer either with OK or ERROR (e.g., disk full).
APPENDFILE SRC_FILENAME DST_FILENAME	C → S	APPENDFILE myfile.txt remotefile.txt	Similar to the client command. Returns: OK if the command succeeded, ERROR otherwise (e.g., the file does not exist). The data is sent by the client after getting the return code in a way equivalent to the APPEND command.

Intermediate submissions

To keep our work in sync with professors, we ask to perform the following intermediate submissions, which include intermediate versions of the program.
This is the list of required intermediate steps:

1. Client program that can parse minimal command line commands.
2. Client and server, which interact through a single network connection using sockets (no HTTP or similar).
3. Client must support multithreading, interacting with the server server through two TCP connections.
4. Enhance the server with the capability to create and handle a dedicated working thread each time a new client connects to it, hence supporting multiple connections at the same time.
5. Evolve the client/server application supporting a few commands (e.g., CONNECT and READ) and the related return codes.
6. Improve the client/server application with support for all the commands, but without taking care of racing conditions and/or mutual exclusions;
7. Deliver the final client/server application, which includes the support for concurrent connections and commands.

Finally, remember that the final program will be tested using the software provided by other students; in client/server environments, interoperability is a fundamental requirement.