Engineering project proposal: Garage door

Nick van der Merwe - s5151332 - nick.vandermerwe@griffithuni.edu.au

September 9, 2020

1 Problem Statement

A server-client system is required, where the server functions as a remote point to upload programs and run them. Additionally, it should be able to fetch file contents, return its CPU model, list the programs on the server and list the contents of the programs.

2 User Requirements

The user should be able to launch the client, (given the server is running separately, on its own) and connect via the IP and host. Once connected, the following commands should work:

1. "Put": Sends a file to the server
2. "Get": Fetches a file from the system and print it to the screen
3. or save onto a file
4. Run a program on the server side, with the option of additional arguments and downloading it to a localfile
5. List the program names, list the program's directories and optionally change to a long list view
6. sys, which grabs the server's operating system and version alongside its CPU model.

3 Software Requirements

There should be two Cygwin compiled executables, server and client. Out of these the server should be launched first, and the user should launch the client and connect to the port.

1. The client side should specify the server's IP address through the commndline arguments - ./client ¡IP¿
2. On the client side, the time for each read should be reported. An added detail is what the read is for, and a maximum timeout of 500ms. This is to prevent bugs in case the server crashes.
3. The client must be a shell format and given that it isn't currently running a command, must always be able to accept input.
4. The server should not block input from a client when it requests it. In other words, a single client must not hog the server and stop others from using it
5. For each request from a client, the server must split into a new process.
6. The server must be able to handle multiple clients at the same time

7. When the client types in *put*

$$progname$$

$$sourcefile(s)$$

$$-f$$

the server should download the specified files into a directory named the specified program name. If the -f option is given, it will delete everything in the directory before doing this.

8. When the client types in *get progname sourcefile* the source file should be printed to the client's screen 40 lines at a time with any keypress to continue

9. When the client types in *run progname*

$$args$$

$$-flocalfile$$

, it will run the specified program and pass the given arguments. if -f is specified, the following word is the file to download the output on the client's side to, otherwise it will be printed 40 lines at a time with a keypress in between.

10. When the client types in *list*

$$-l$$

$$progname$$

it returns the list of program names or the contents of the specified program name, staggered every 40 lines and continued with a keypress. If the -l parameter is given then it will print the view in a long view.

11. The sys command will return the CPU and operating system

12. Whenever an error occurs on the server side, the server will report what happened and kill the process. On the other side, every read times out after five seconds with a message for what the client was trying to read.

13. The process table must be properly cleared; zombie processes must be killed. These will be reported on the server side whenever one occurs.

4 Software design

In terms of our file organisation, both the server and client will have two major files. The first handles the sockets while the second manages each command. While this can result in reasonably larger files, a rule of thumb is 30 lines of code per function, and a maximum of 30 functions per class. Thus 900 lines of code per class; in this case we're using C, so we treat a file like this instead.

Our basic logical block diagram looks like this:

Functions

Due to the programming style there are a large number of functions, but they will be broken up into the client side, server side and major functions/minor functions.

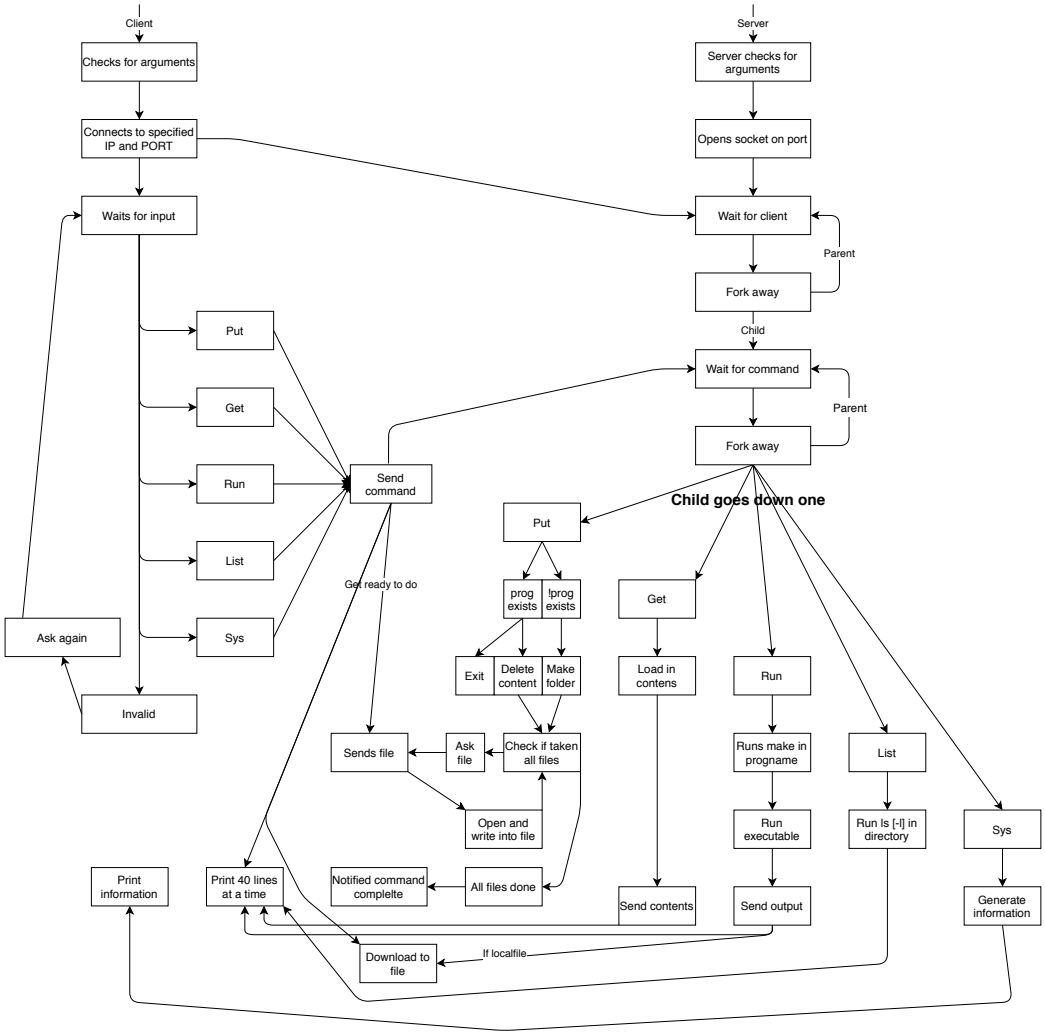

Figure 1: High level design of the interactions

Client side

1. *main(argc, argv)* - Runs defineSocketToServer() and chatWithServer()
2. *defineSocketToServer(char* host, int portNumber).* Defines the socket, specifies the host and port and connects to it
3. *chatWithServer* - Grabs input and runs handleCommand()
   (a) *handleCommand(char*command, int commandSize, int socketFD)* - Interprets the command and runs the appropriate function
4. Timer functions, major = timeOutRead
   (a) *startTimer()* - Begins the timer for reportTime();
   (b) *reportTime(char*task)* - Reports the time and prints what task it was
   (c) *timeOutRead(int socket, char *buffer, int sizeOfBuffer, char *task)* - Allows a maximum of five seconds for a read to run and reports how long it took with the task.
5. Word splitting, major = splitIntoWords
   (a) *countWords(char * string)* - counts how many words there are in the given string, split only by space characters
   (b) *splitIntoWords(char *string, char ***toFill)* - Fills the given triple pointer with the space seperated tokens of the string. Note that it requires a free on the toFill
6. Loading and measuring files
   (a) *loadFileIntoString(char * toFill, FILE * filePointer)* - loads the file into the specified string
   (b) *measureFileBytes(FILE * filePointer)* - Counts how many letters are in a file
7. Put command
   (a) *transferRequestedFile(char *fileRequest, int socketFD)* - sends the file that's asked for
   (b) *putCommand(char * command, int commandSize, int socketFD)* - Interacts with the server to run the put command
8. Printing large strings
   (a) *waitForKeypress()* since C doesn't have a distinct command for a single keypress, this was written so that not only the enter key can be pressed
   (b) *printXLinesAtATime(char*string, int stringSize, int numberOfLines)* - Prints the lines in a neat format for the user to read.
9. Get command
   (a) *getCommand(char * command, int commandSize, int socketFD)* - Handles the get command
   (b) *saveFile(char * fileString, int fileSize, char *fileName)* - Saves a file into the specified file relative to the current working directory
10. *runCommand(char * command, int commandSize, int socketFD)* handles the run command
11. *listCommand(char * command, int commandSize, int socketFD)* - handles list command
12. *sysCommand(char * command, int commandSize, int socketFD)* - handles system command

Server side

Server Driver

1. *main()* - Parses the port and runs signal(killZombieProcess), openTCPSocket() and host()
2. *killZombieProcess(int signal)* - Kills any dead forks - ran with signal()
3. openTCPSocket
   (a) *defineTCPSocket(int * socketFD)* - Calls the socket() command with error checking
   (b) *specifySockAddress_in(struct sockaddr_in *const* socketAddress, int portNumber) - fills the sockaddr_in with the required constants and port number
   (c) *bindSocketAddress_inToSocket(const int * socketFD, const struct sockaddr_in *socketAddress)* - Runs bind on the sockaddr and socket with error checking
   (d) *listenToSocket(int const * socketFD, int const backlog)* - runs listen() on the socket with error checking
   (e) *openTCPSocket(int const portNumber, int const backlog)* - Opens a socket by running all the minor commands here

Server tools

1. Word splitting, major = splitIntoWords
   (a) *countWords(char * string)* - counts how many words there are in the given string, split only by space characters
   (b) *splitIntoWords(char *string, char ***toFill)* - Fills the given triple pointer with the space seperated tokens of the string. Note that it requires a free on the toFill
2. Directory commands
   (a) *deleteDirectoryContents, char * path* - Deletes the contents of the specified directory
   (b) *getProgramsDirectory(char * fullDirectory)* - Fills the full directory with the path to the server's programs directory
   (c) *getProgrnameDirectory(char *fullDirectory, char *folder)* - Gets the full directory to the given program
   (d) *getProgramFileDirectory(char * fullDirectory, char *directory, char * fileName)* - fills the full directory with the absolute path to a specific file in a program
   (e) *makeDirectoryOrDeleteIfForced(char * dirName, int forced)* - Makes a directory or deletes it if its forced
3. putCommand
   (a) *downloadFile(int socketFD, char * directory, char *fileName, int fileNameLen, int forced)* - Downloads the specified file into the given directory
   (b) *putCommand(char * command, int socketFD)* - Handles the put command
4. Get command
   (a) *measureFileBytes(FILE * filePointer)* - Measures how many bytes are in a file
   (b) *loadFileIntoString(char * toFill, FILE * filePointer)* - Loads a file into the given string
   (c) *sendFile(File * fp, int socketFD)* - Sends the given file to the socket
   (d) *getCommand(char * command, int socketFD)* - Handles the get command
5. Run command

```
# ================ Client side ==============
def sendFile(int socketFD, char * fileText):
    sendToServer(socketFD, strlen(fileText))
    sendToServer(socketFD, fileText)

# ============== Server side ================
def recieveFile(int socketFD, char * toFill):
    int fileSize
    readFromClient(socketFD, fileSize)
    toFill = char[fileSize]
    readFromClient(socketFD, toFill)
```

Figure 2: Psuedocode for sending large pieces of texts, typically files

(a) *extractRunCommand(char * extractInto, char ** splitCommand, int endOfSplitCommand)* - Finds what the actual command that should be ran with popen() is in the passed command

(b) *compileFile(char * directory)* - Compiles the directory

(c) *loadInPipe(char ** loadInto, FILE * pipePointer)* - loads the pipe into the string. Note that the string needs to be freed afterwards.

(d) *sendPipe(FILE * pipePointer, int socketFD)* - Sends the pipe to the socket

(e) *runFileAndSendOutput(int socketFD, char * progName)* - runs the file and sends its output

(f) *runCommand(char * command, int socketFD)* - handles the run command

6. List command

(a) *defineListVariables(char * command, int *wasProgNameRequested, int *longList, char *progname)* - Defines all the variables needed for the list command from the passed command

(b) *openListCommand(const int wasProgNameRequested, const int longList, char * progName)* - popens() on the list command

(c) *listCommand(char * command, int socketFD)* - Runs the list command

7. *sysCommand(int socketFD)* - Sends the system information to the socket.

For more information on these, read the code. There are documentation comments everywhere.

Data structures

Arrays were used.

Algorithms

The only moderately complicated algorithm was sending commands. Otherwise the code basically reads like the high level design with added error checking. To send a file, the side that's sending it posts how large it is in bytes, the other side allocates that, then the source sends the file itself.

5 Requirement Acceptance Tests

Everything works.

Requirement	Test	Implemented	Pass/fail
Insert IP address as argument to client	Insert IP address as argument to client	Yes	Pass
Client reports time	Any command	Yes	Pass
Client loop	Run two commands	Yes	Pass
Multiple clients	Connect multiple clients	Yes	Pass
Spawn process for new requests	Check if spawns process	Yes	Pass
Basic put command	put progname file	yes	pass
Overwrite put command	put progname file -f into directory with multiple files	yes	pass
Put command allows multiple files	put progname file1 file2	yes	pass
Get command runs	Run get command	Yes	Pass
Run command compiles	Check if it compiles when run command is ran	yes	Pass
Run command runs	Check if it runs	Yes	Pass
Run command uses arguments	run progname args	Yes	pass
Run command downloads to localfile	Run progname -f localfile	Yes	Pass
List commands works	list	yes	pass
Long list command works	List -l	yes	pass
List command works in progname	list progname	yes	pass
Long list command works in progname	list -l progname OR list progname -l	yes	pass
sys command works	sys	yes	pass
List command prints 40 lines at a time	List on big directory	Yes	pass
Get command prints 40 lines at a time	Get on big file	Yes	Pass
Server reports errors to itself and client can insert invalid commands	Run invalid command	Yes	Pass
Zombies are removed	Run multiple commands and see if the server ever reports killing a zombie	Yes	Pass

6 Detailed Software Testing

Everything works.

Test	Expected output	Output
Random word; pie	Unknown command	Unknown command
put progname invalid file	Failed to open file	Failed to open file
put fileThatExists sourcefiles	File already exists	File already exists
get directoryThatDoesn'tExist file	Directory doesn't exist	Directory doesn't exist
get directory FileDoesn'tExist	File doesn't exist	File doesn't exist
run progNameDoesn'tExist	progName doesn't exist	progName doesn't exist
run progname invalidArguments	progName fails	progName fails
run -f progname args file	Invalid command	Invalid command
list progNamedoesn'tExist	progName doesn't exist	progName doesn't exist
sys any text afterwards	runs fine	runs fine

7 User Instructions

Run server, run client. Use commands in the user requirements section. The only special thing is that files in the put command must be in the exact directory as the command - it can't be outside that.