2803ICT Systems and Distributed Computing
Remote Execution System

Caleb Howard

September 9, 2020

1. Problem Statement

The problem objective is to create a remote execution system with a client/server model. The user can connect to a server via a client and query the server with certain commands that perform different tasks. The user commands are extensions of *put, get, run, list* and *sys*, refer to the *user requirements* section of this document for a detailed definition of each command.
The server can connect to multiple non-blocking clients.

2. User Requirements

The following outlines the user requirements for the program (for details on how to use client commands, refer to the *user instructions* section of this document):

- The user should be able to upload source files from the client to a given directory on the server using the *put* command. If the directory exists on the server, the user can add the optional flag *-f* to overwrite said directory.
- The user should be able to download source code to the client from a file in a given directory on the server and either print to console, or if given the *-f* flag, a file using the *get* command.
- Using the *run* command a user should be able to compile and execute source code in a given directory on the server. A source file should only be compiled if required. Additionally, the user should have the option to provide arguments for executing the sub-program. If the *-f* flag and an additional file name is provided, the output of the sub-program should be written to said file, otherwise it should be written to the client console.
- The user should be able to use the *list* command to retrieve list information from the server, that is, list all files on the server or all files in an optionally provided directory. Additionally, the user should have the option to use the *-l* flag to retrieve the long list information.
- The user should be able view the server operating system and CPU type by running the *sys* command.
- The user should be able to disconnect from the server and exit the client program using the *quit* command.

3. Software Requirements

1. Using a socket connection, the client program will query the remote server program that is listening on port 80.
2. The IP address of the server to be queried by the client shall be given to the client as a command line argument.
3. The client shall wait for the user to enter queries (via stdin), which it then forwards to the server in a loop until the user types ‘quit’. Any responses from the server are immediately displayed to the user.
4. The client will report the time taken for the server to respond to each query together with the server’s response.
5. The client is non-blocking. An infinite number of server queries may be outstanding.
6. The server will spawn a new process to execute each new request and must be able to accept multiple clients.
7. The server will be able to accept one or more source files and a ‘progname’ and place the files in a directory called ‘progname’. It will be able to compile the source files (if not previously compiled), run the executable with command line arguments provided from the client and return the result to the clients.
8. The following query commands (and options) are to be recognised by the server (anything within []s optional):
    A. put progname sourcefile[s] [-f] : upload sourcefiles to progname dir, -f overwrite if exists.
    B. get progname sourcefile : download sourcefile from progname dir to clients screen.
    C. run progname [args] [-f localfile] : compile (if req.) and run the executable (with args) and either print the return results to screen or given local file.
    D. list [-l] [progname] : list the prognames on the server or files in the given progname directory to the screen, –l = long list
    E. sys : return the name and version of the Operating System and CPU type.
9. The long list (-l) option of the list command will also return the file size, creation date and access permissions. If no progname is given, then the list of all available progname directories will be returned.
10. The get command will dump the file contents to the screen 40 lines at a time and pause, waiting for a key to be pressed before displaying the next 40 lines etc.
11. The put command will create a new directory on the server called ‘progname’ If the remote progname exists the server will return an error, unless -f has been specified, in which case the directory will be completely overwritten (old content is deleted). This command allows you to upload one or more files from the client to the server.
12. If a ‘-f localfile’ option is given to the run command a new file on the client will be created. If the localfile name exists, the file will be overwritten.

13. The run command will check to see if a ‘progname’ has been compiled, and if not will compile the relevant files as required. run will initiate a compile if there is no executable in the folder, or its creation date is older than the last modified date of a source file. It will then run the executable, passing to it any specified command line arguments, and the server will redirect output from the executed program to the client. If the program can’t be run (or compiled) an appropriate error will be returned to the client. You must not use the system() call to compile or run the ‘progname’.
14. If the server receives an incorrectly specified command it will return an error. If the server is unable to execute a valid command the server will return the error string generated by the operating system to the client.
15. All Zombie processes are terminated as required. There is to be no unwanted Zombie processes on either the client, or the server.

4. Software Design

List of Software Functions:
The client/server program has 3 modules (source files), a list of function definitions for each module is as follows.

Helper.c
allocateDirectory(char*, int)
- This function is used by the server’s put handling function. It takes a directory name and an overwrite flag and creates a new directory with the provided name. If the directory exists and the overwrite flag is false (0) then the function returns -1, else it returns 0 for success.

writeToFile(char*, char*)
- Takes a file path and and content string and writes the content to the file located at file path. Returns -1 if a failure occurred.

getFileContent(char*)
- Takes a file path and reads the file’s content, returns the content in a string buffer.

runSystemCommand(char*)
- Takes a system command and uses popen to run the command. The results of the command, or any errors that occured from running the system command are returned from the function.

updateDirMakeFile(char*)
- Takes a directory name as a parameter. The function scans the directory for any .c files, then proceeds to create a makefile for the directory that includes all c files from that directory. It returns -1 if any errors occurred, else it returns 0.

error(char*)
- Simply takes a char* and prints it as a perror.

Client.c

putCommand(int, char[], char[][], int)

- Takes the socket file descriptor, the client buff (input from the user), the input array (client buff split into words) and the length of the input array. Handles all functionality of the put command; puts client files to the server and handles errors.

getCommand(int, char[], len)

- Takes the socket file descriptor, the client buff (input from the user), and the number of words in the client buff. Handles all functionality of the get command; gets server file content and handles errors.

runCommand(int, char[], char[][], int)

- Takes the socket file descriptor, the client buff (input from the user), the input array (client buff split into words) and the length of the input array. Handles all functionality of the run command; compiles and executes source code on the server, displays results to console or prints to file, and handles errors.

listCommand(int, char[])

- Takes the socket file descriptor and the client buff (user input). Handles all functionality of the list command; retrieves list information from server directories or the server itself, also handles errors.

sysCommand(int, char[])

- Takes the socket file descriptor and the client buff (user input). Handles all the sys command; retrieves operating system type and version, and CPU type of the server, from the server.

clientLoop(int)

- Takes the socket file descriptor. Gets the user input, parses it and calls the appropriate command function.

main(int, char**)

- Main program function, takes an IP address as a command line argument, connects to the server and runs the clientLoop function.

Server.c

putCommand(int, char[][], int)

- Takes the socket file descriptor, the message array (client command message split into words) and the length of the message array. Handles all functionality of the put command; sends the client -1 if directory exists an "-f" is not provided, otherwise accepts file content from the client.

getCommand(int, char[][])

- Takes the socket file descriptor and the message array (client command message split into words). Handles all functionality of the get command; reads a file's content and sends the content to the client.

runCommand(int, char[][], int)

- Takes the socket file descriptor, the message array (client command message split into words) and the length of the message array. Handles all functionality of the run command; compiles and executes source code in the directory "progname", sends results or any errors to the client.

listCommand(int, char[])

- Takes the socket file descriptor, the message array (client command message split into words) and the length of the message array. Handles all functionality of the list command; retrieves list information from server directories or the server itself, also handles errors.

sysCommand(int)

- Takes the socket file descriptor. Handles the sys command; retrieves operating system type and version, and CPU type of the server and sends the information to the client.

serverLoop(int)

- Takes the socket file descriptor. Receives the user input command from the client, parses it and calls the appropriate command function.

ZombieKill(int)

- Kills zombie processes.

main(int, char**)

- Main program function, waits to connect to one or more non-blocking clients and runs the serverLoop function for each client.

Data Structures used in program:

The data structures used in this program are arrays.

Pseudocode for the updateDirMakefile function:

```
BEGIN updateDirMakefile(dir_name)
        Open directory
        dir_content = load all the filenames from the directory

        IF (directory open failed)
                Return -1

        FOR all values v in dir_content:
                IF the last letter of v is “c”
                        Copy value into c_files array

        Open makefile in directory with write properties.
        IF open failed
                Return -1

        FOR the first two lines of the file; i=0 to 1
                IF i equals 0
                        Print “output: “ to file
                ELSE
                        Print “gcc “ to file
                FOR all values v in c_files:
                        obj_file = Create a new string with the last letter of v changed to “o”.
                        Print obj_file followed by a space to the file.
                IF i equals 0
                        Print a newline to the file
                ELSE
                        Print “-o output\n” to the file

        // Every two lines after the first two, for the every .o target
        FOR (i=0 to length of c_files -1)
                obj_file = Create a new string with the last letter of v changed to “o”.
                Print “obj_file: c_files[i]\n” to the file.  // First line
                Print “gcc -c c_files[i]\n” to the file.     // Second line

        Close the makefile
        Return 0 // Success
END
```

5. Requirement Acceptance Tests

Software Requirement No.	Test	Implemented (Full/Partial/ None)	Comments
1	Connect to the server via the client and issue a command.	Full	
2	Start the client with and without the IP argument. Ensure an error message is displayed if the IP is not provided.	Full	
3	Send the server two commands back to back, then send the *quit* command.	Full	
4	Test each command and ensure the time is printed to the client after each process is finished.	Full	
5	Test commands on multiple clients that are connected to the server simultaneously.	Full	
6	Covered by requirement test 5.	Full	
7	Covered by requirement test 11, 12, 13.	Full	
8	A. Covered by requirement test 11. B. Covered by requirement test 10. C. Covered by requirement test 11, 12. D. Covered by requirement test 9. E. Run sys on the client and ensure the displayed server OS and CPU information is accurate.	Full	
9	Test *list* by itself, test *list -l* for long list information, test *list progname* and *list -l progname* for directory list information.	Full	
10	Test *get prognme sourcefile* on a test file and directory. Ensure the content is dumped 40 lines at a time with each 40 coming after the ENTER key is pressed.	Full	
11	Test *put* without the *-f* overwrite flag on a directory that already exists on the server, ensure error messages are displayed on client. Test *put* with the *-f* flag on on a directory that does exist, then on one that does not and ensure the file arrives on the server. Test without the *-f* on a directory that does not exist on the server and ensure the files arrive on the server. Repeat the above with more than 1 file.	Full	
12	Test the run command using the *-f* flag for a file that does exist and for a file that does not, ensure that the run results are printed to the file in both cases.	Full	

13	Ensure the *run* command updates the progname's makefile on the server side and calls make in the progname's directory, then executes the program. Ensure the results or any errors are displayed to the client. This can be achieved by editing a source file manually and testing that the update is present after running the command. Another test is to manually include a syntax error in the source code and ensure the *run* command displays the intended error message.	Full	
14	Input a random sequence of text into the client and ensure the correct error message was displayed. Test variation of each command that are slightly incorrect and ensure the appropriate error message is displayed	Full	
15	Print to console in the ZombieKill() function and check that the print is displayed when exiting a client to ensure the function is being called correctly.	Full	

6. Detailed Software Testing

No	Test	Expected Results	Actual Results
1.0	**put command**		
1.1	Run the *put* command with less than 3 arguments.	“ERROR: put command requires minimum 3 arguments.”	As expected
1.2	Run *put* without the *-f* flag for a progname that already exists on the server.	"PERMISSION DENIED: cannot overwrite directory on server, to overwrite directory use the flag -f.”	
1.3	Run *put* without the *-f* flag for a progname that does not exist on the server.	The file will appear on the server in the progname directory.	
1.4	Run *put* with the *-f* flag for a progname that does not exist on the server.	The file will appear on the server in the progname directory.	
1.5	Run *put* with the *-f* flag and with multiple sourcefiles for a progname that already exists on the server.	The file will appear on the server in the progname directory.	
1.6	Run *put* without the *-f* flag and with multiple sourcefiles for a progname that does not exist on the server.	The file will appear on the server in the progname directory.	
1.7	Run *put* without the *-f* flag and with multiple sourcefiles for a progname that already exists on the server.	"PERMISSION DENIED: cannot overwrite directory on server, to overwrite directory use the flag -f.”	
2.0	**get command**		
2.1	Run the *get* command with less than 3 arguments.	“ERROR: get command requires minimum 3 arguments.”	As expected
2.2	Run the *get* command with a directory that does not exist on the server.	“ERROR: file does not exist.”	
2.3	Run the get command with a file in a directory on the server that has less than 40 lines.	File content will print to the client console.	
2.4	Run the get command with a file in a directory on the server that has greater than 40 lines. Press enter.	40 lines of the file content will print to the client console followed by “Press ENTER to view more”. After pressing enter 40 more lines (or less if that is all the content) will be displayed to the console.	
3.0	**run command**		
3.1	Run the *run* command with less than 3 arguments.	“ERROR: run command requires minimum 2 arguments.”	As expected
3.2	Run the *run* command with a directory that does not exist on the server.	“ERROR: failed to execute system command.”	

3.3	Run the *run* command with a directory that has multiple error free .c files in it. Have the main function print the message "Test".	"Test"	
3.4	Run the *run* command with a directory that has a .c file in it with syntax errors.	Client prints the accurate syntax error.	
3.5	Run the *run* command with a directory that has an error free .c file in it and provide "bob" and "jim" program arguments, have the .c file print all the arguments, each on a new line.	"./test/output bob jim"	
3.6	Run the *run* command with a directory that has an error free .c file in it and the *-f localfile* flag.	The results are not displayed to the client console and instead printed to a file with the specified *localfile* name.	
4.0	**list command**		
4.1	Run *list* by itself, confirm the results using the ls terminal command in the server directory.	*list* matches the output of *ls* for the same directory.	As expected.
4.2	Run *list progname* and confirm the results using the ls terminal command in the progname directory.		
4.3	Repeat the above but with the *-l* flag for both *list* and *ls*.	*list -l* matches the output of *ls -l* for the same directory.	
5.0	**sys command**		
5.1	Run the sys command. Use the terminal command *cat* to confirm the details match with the values in /proc/cpuinfo and /proc/version.	Client displays Linux Manjaro and Intel 4790k with other information such as cpu speed, kernel info, etc.	As expected

7. User Instructions

1. Launch the server if it is not already running.
2. Launch a client.
3. Repeatedly run one of the following commands (note: [ ] arguments are optional):
    - *put progname sourcefile[s]* : upload one or more sourcefiles to a directory (progname) on the server. Add the optional *-f* flag to overwrite an existing directory.
    - *get progname sourcefile* : download the contents of the sourcefile from the progname directory on the server to the client screen 40 lines per key press by the user.
    - *run progname [args] [-f localfile]* : compile the source code in the progname directory on the server and run the executable with the optionally provided *args*. The result (or error messages) of the executed program will either print to the client screen or you can optionally provide a file to print the results to by using the optional parameters *-f localfile*, where localfile is where the results will be stored.
    - *list [-l] progname* : list the file contents of the server or if the optional parameter *progname* is provided, the file contents list information will be from the progname

directory. Additionally the *-l* flag can be used to provide the directories content's file size, file creation date and access permission information.

- *sys* : displays the servers operating system version and CPU type to the client screen
- *quit* : disconnects from the server and exits the client program.