# 2803ICT Assignment 1 – Option A

Lochie Ashcroft – s5080439

## Contents



## 1. Problem Statement

Option A of assignment one is to write a simple remote execution system. It consists of a client and server which communicate over the network and allows the client to run a number of system commands on the server. The features include uploading C source files, compiling and running these files, listing directories/files on the server, viewing files on the server and also returning the system information of the server. The output of the server can either be stored locally in a file on the client or displayed to the screen depending on the command and flags used.

## 2. User Requirements

The following outlines the user requirements for the program:

- The server IP will be provided as a command line argument when running the client
- The user can execute a number of commands in parallel (non blocking)
- Commands are read from stdin
- Typing ‘quit’ will close the client
- Ability to upload source code files to the server
- Ability to read source files from the server
- Ability to run programs on the server and reviving the output (screen or local file)
- Ability to list the programs on the server or the source files of a specific programs
- Ability to view the servers Operating System version and CPU type

## 3. Software Requirements

The following outlines the software requirements for the client:

1. The server IP address will be given as a command line argument
2. Will communicate with the server over port 80 (I defined a port in common.h)
3. Will read user commands through stdin and forward to the server, until the user enters ‘quit’. Server responses are immediately displayed
4. The time taken for the server to respond will be returned as well as the server response
5. The client is non blocking
6. Compiled on both Windows/Unix
7. The get command will pause at 40 lines and continue scrolling when enter is pressed
8. If the local file flag is provided in the run command the output will be saved to a ‘local file’
9. The put command will upload local files
10. No zombie processes

The following outlines the software requirements for the server:

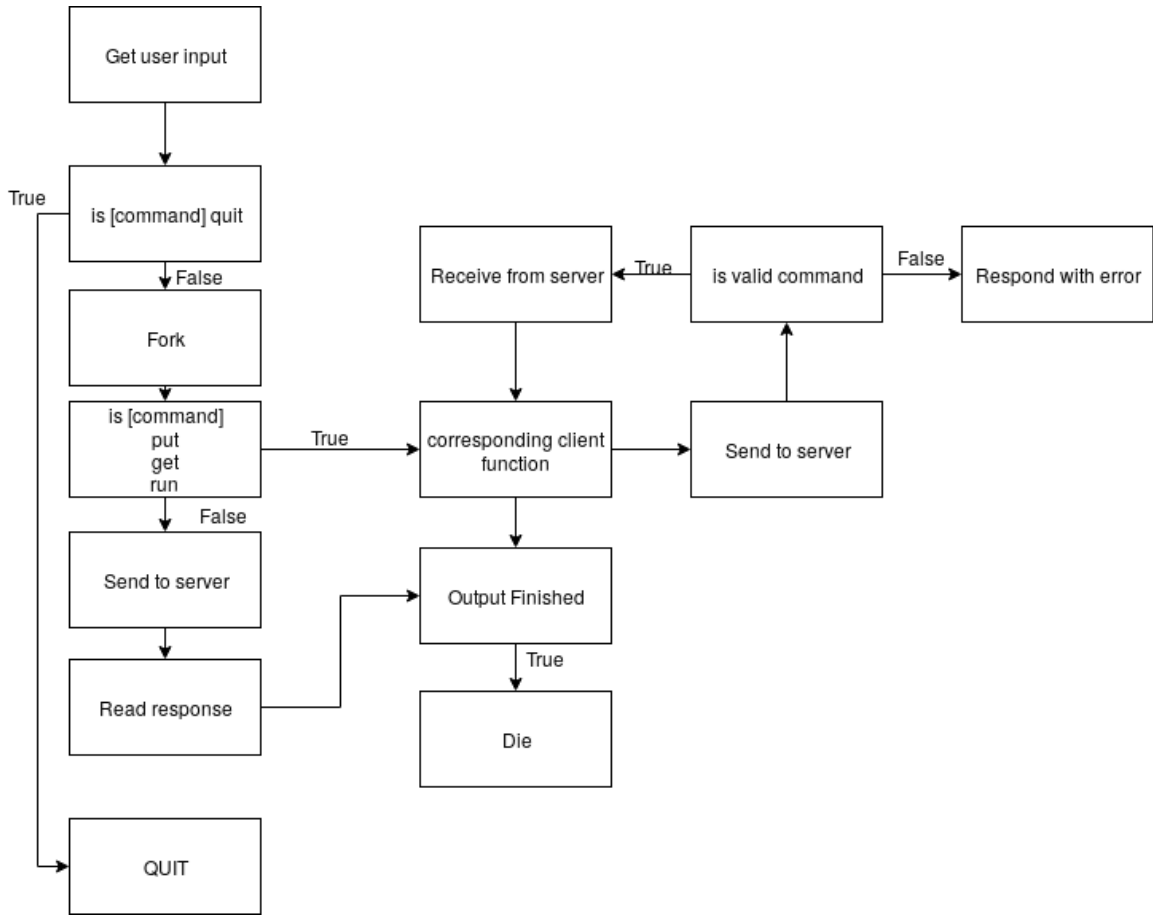
1. Will communicate to the client over port 80 (I defined a port in common.h)
2. The server will spawn a new process to execute each new request and must be able to accept multiple clients
3. The server will be able to accept one or more source files and a ‘progname’ and place the files in a directory 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 client.
4. Compiled for both windows/Unix
5. put progname sourcefile[s] [-f]: upload sourcefiles to progname dir, -f overwrite if exists
6. get progname sourcefile: upload sourcefile from progname dir to the client
7. run progname [args] [-f localfile]: compile (if req.) and run the executable (with args)
8. List [-l] [progname]: list the prognames on the server or files in the given progname directory
9. 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

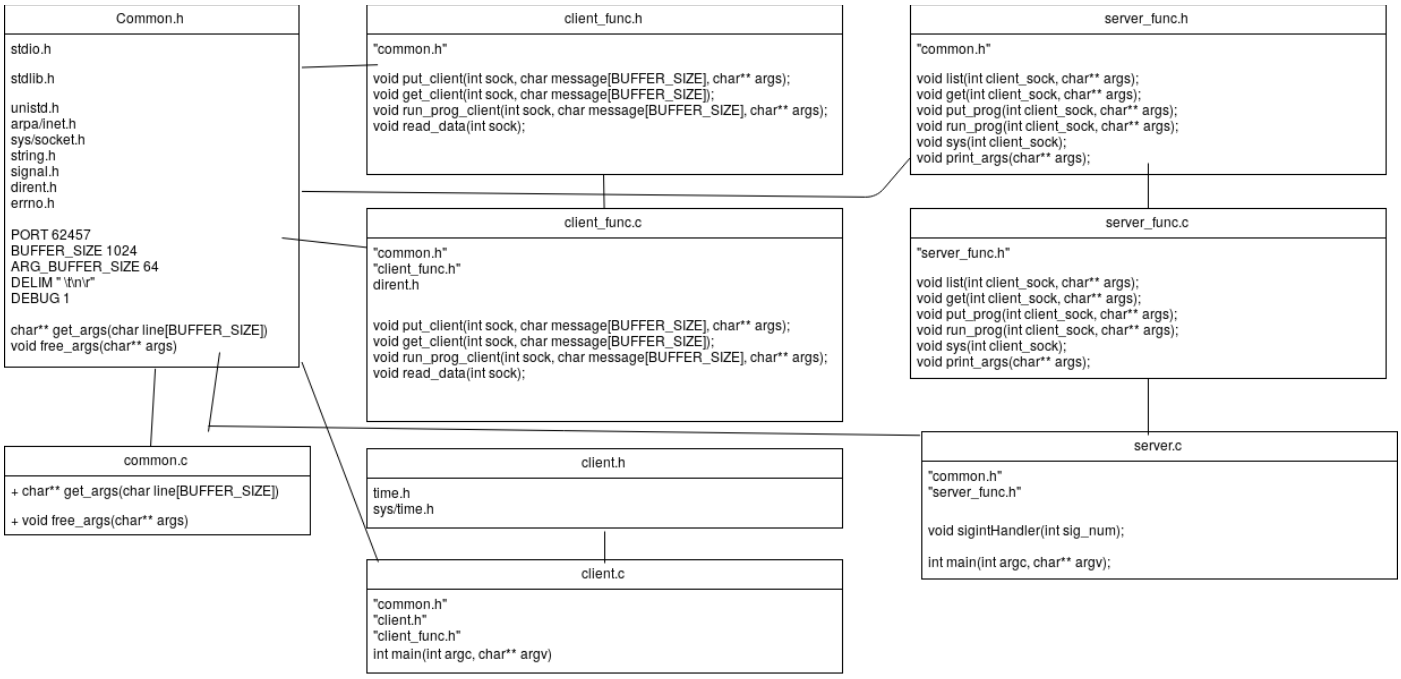
10. The run command will check to see if a ‘progname’ has been compiled, and if not will compile the relevant files as required. Run will initiate a compile if there is no executable in the folder,or its creation date is older than the last modified date of a source file. It will then run the executable, 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’.
11. 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.
12. No zombie processes

## 4. Software Design

## 5. High Level Design – Logical Block Diagram



## Structure Chart



## List of all functions in the software.

### common.h

char** get_args(char line[BUFFER_SIZE])
- Given a line of text split into arguments (tokenize)
- Input is a string
- returns 2D string array of the tokens which is dynamically allocated

void free_args(char** args)
- Given a 2D string array
- frees each string and than the array

### server_func.h

void list(int client_sock, char** args)
- given a client descriptor and 2D list of strings
- lists files/directories as stated in the requirements with the arguments
- sends this information to the client though the client socket

void get(int client_sock, char** args)
- given a socket descriptor and 2D list of strings
- sends the content of the file which is given in the args to to the client 40 lines at a time and resumes when the client sends a response

void put_prog(int client_sock, char** args)
- given a socket descriptor and 2D list of strings
- accepts a program name and source file(s) from the client as defined in requirements, stores the file locally
- client sends these files to the server and the server writes them locally

void run_prog(int client_sock, char** args)
- given a socket descriptor and 2D list of strings
- attempts to run a program specified in the args
- will compile if necessary
- sends output of program to client

void sys(int client_sock)
- given a socket descriptor
- gets the system information as defined in the requirements
- send this information to the client

void print_args(char** args)
- given a 2D list of strings
- print out all the arguments in args

### client_func.h

void put_client(int sock, char message[BUFFER_SIZE], char** args)
- given a socket descriptor (server), a string (user input) and args (tokenized user input)
- read local file(s) and upload file(s) to the server

void get_client(int sock, char message[BUFFER_SIZE])
- given a socket descriptor (server), a string (user input)
- read a file from the server 40 lines at a time

void run_prog_client(int sock, char message[BUFFER_SIZE], char** args)
- given a socket descriptor (server), a string (user input) and args (tokenized user input)
- run a program on the server
- either print to the screen or save to file the output the server sends

void read_data(int sock)
- given a socket descriptor (server)
- receive all data from the server until the end character is send (back tick)

### server.c

void sigintHandler(int sig_num)
- takes in a signal
- called when CTRL+C is pressed
- closes the server socket and exits the programs

int main(int argc, char ** argv)
- entry point of the application
- sets up variables
- infinite loop and forks on every new client connection

### client.c

int main(int argc, char ** argv)
- entry point of the application
- sets up variables
- infinite loop and forks on every new user input

## List of all data structures in the software. (eg linked lists, trees, arrays etc)

There is just so many, most are duplicate though, so the key ones will be described.

**Server:**

Socket descriptor: used to interface with the client over the network

sockaddr_in: stores information about a client connection

socklen_t: used to store the length of a socket descriptor

char buffer[BUFFER_SIZE]: general use buffer

char** args: 2D array of client arguments

pid_t: used when forking the server

**Server_func:**

char* : dynamically allocated strings, used a lot when appending user arguments onto system commands

FILE*: used when opening files and using popen()

base_command: const char* which holds a base command, user arguments are append to it

**Client:**

char message[] : stores user input

char reply[] : stores server responses

char** : stores the tokenized user input

clock_t: used to store CPU times – CPU time taken displayed after each command

timeval: used to store system time - time taken displayed after each command

sockaddr_in: used to store server information

size_t: used to store socket size

pid_t: used when forking the client

FILE*: used when opening files

## Detailed Design – Pseudocode

There is a lot of functions so the main infinite loops of the client and server will be shown.

**Client:**

```
while (1)
    get user input from stdin
    args = tokenize the input
    go to start of loop if args[0] is NULL

    if args[0] is “quit”:
            send quit message to server
            exit program

    FORK HERE
            create new socket descriptor
            connect with this socket descriptor to the server
            get CPU time and system time

            if args[0] is “put”:
                    run put_client() function
            else if args[0] is “get”:
                    run get_client() function
            else if args[0] is “run”:
                    run run_prog_client() functions
            else:
                    send user input to server
                    read_data()

            get CPU time and system time
            print CPU time duration and system time duration

            memset the message and reply
            send quit message to server
            close socket to server
            exit

close socket to server
return 0
```

**Server:**

```
while (1)
    client_sock = accept new client

    if client_sock < 0:
            go to start of loop


    FORK HERE
    close server socket
    receive data from client
    args = tokenize the data

    if args[0] is “quit”:
            print client disconnected
            break

    if args[0] is “list”:
            run list()
    else if args[0] is “put):
            run put_prog()
    else if args[0] is “run”:
            run run_prog()
    else if args[0] is “get”:
            run get()
    else if args[0] is “sys”:
            runs sys();
    else:
            send invalid command to client

    close client socket
    exit

close client socket

return 0
```

## 6. Requirement Acceptance Tests

Client

| Software Requirement No | Test | Implemented (Full /Partial/ None) | Test Results (Pass/ Fail) | Comments (for partial implementation or failed test results) |
|---|---|---|---|---|
| 1 | Query the server on port 80 | F | P | |
| 2 | Server IP from command line argument | F | P | |
| 3 | Get input from stdin, forward to server, in a loop until ‘quit’, output displayed immediately | F | P | |
| 4 | Time taken to get response from server is reported | F | P | |
| 5 | Non-blocking | F | P | |
| 6 | Windows/Unix compiled | P | F | A lot of posix calls, but no effort was made to make it compilable for windows |
| 7 | Put progname sourcefiles[s] [-f] | F | P | |
| 8 | get progname sourcefile | F | P | Have to spam enter a few times when >= 40 lines |
| 9 | run progname [args] [-f localfile] | F | P | Wont print to screen will only save to file |
| 10 | list [-l] [progname] | F | P | |
| 11 | sys | F | P | |
| 12 | No zombies | F | P | |

Server

| Software Requirement No | Test | Implemented (Full /Partial/ None) | Test Results (Pass/ Fail) | Comments (for partial implementation or failed test results) |
|---|---|---|---|---|
| 1 | Listen on port 80 | F | P | |
| 2 | Spawn new process for every query and accept multiples clients | F | P | |
| 3 | Put command | F | P | |
| 4 | Get command | F | P | |
| 5 | Run command | F | P | |
| 6 | list command | F | P | |
| 7 | Sys command | F | P | |
| 8 | Return error for invalid command | F | P | |
| 9 | No Zombies | F | P | |

## 7. Detailed Software Testing

| No | Test | Expected Results | Actual Results |
|---|---|---|---|
| **1.0** | **Put** | | |
| | *Put progname file1 file2 file3* | Upload the files into the progname directory | As expected |
| | Put usedname file1 file2 file3 | Program directory already exists | As expected |
| | Put usedname file1 file2 file3 -f | Delete usedname, upload the files into the usedname directory | As expected |
| | Put progname file1 non_existent_file | non_existent_file does not exist | As expected |
| **2.0** | **Get** | | |
| 2.1 | Get valid_prog valid_file | Display the contents of ./programs/valid_prog/valid_file | As expected |
| | Get valid_prog invalid_file | File not found | As expected |
| | Get invalid_prog invalid_file | File not found | As expected |
| | Get valid_prog | Invalid usage | As expected |
| | Get | Invalid usage | As expected |
| **3.0** | **Run** | | |
| | Run progname | Print progname output | No output on client |
| | Run progname arg1 arg2 | Print progrname output | No output on client |
| | Run programe -f localfile | Save progrname output to localfile | Output saved in localfile |
| | Run progname arg1 arg2 -f localfile | Save progrname output to localfile | Output saved in localfile |
| | Run uncompiled_program | Compile program, run, print output | No output on client |
| | Run uncompiled_program -f localfile | Compile program, run, save output to localfile | Output saved in localfile |
| | Run non_existent_program | Program directory does not exist | Program directory does not exist |
| **4.0** | **List** | | |
| | list | List of all programs | As expected |
| | List -l | Long list of all programs | As expected |
| | List valid_prog | List of source files in valid_prog | As expected |
| | List -l valid_prog | Long list of source files in valid_prog | As expected |

| No | Test | Expected Results | Actual Results |
|---|---|---|---|
| | List invalid_prog | Nothing | As expected |
| **5.0** | **Sys** | | |
| | sys | System information | As expected |

## 8. User Instructions

**Compiler setup (Fedora 27, kernel 4.17.17-100.fc27.x86_64):**
**OUTPUT OF "gcc -v"**
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/libexec/gcc/x86_64-redhat-linux/7/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-redhat-linux
Configured with: ../configure --enable-bootstrap --enable-languages=c,c++,objc,obj-c++,fortran,ada,go,lto --prefix=/usr --mandir=/usr/share/man --infodir=/usr/share/info --with-bugurl=http://bugzilla.redhat.com/bugzilla --enable-shared --enable-threads=posix --enable-checking=release --enable-multilib --with-system-zlib --enable-__cxa_atexit --disable-libunwind-exceptions --enable-gnu-unique-object --enable-linker-build-id --with-gcc-major-version-only --with-linker-hash-style=gnu --enable-plugin --enable-initfini-array --with-isl --enable-libmpx --enable-offload-targets=nvptx-none --without-cuda-driver --enable-gnu-indirect-function --with-tune=generic --with-arch_32=i686 --build=x86_64-redhat-linux
Thread model: posix
gcc version 7.3.1 20180712 (Red Hat 7.3.1-6) (GCC)

**Compile instructions**
use the included make file

"make", "make all", "make client", "make server"

No windows instructions

**Client instructions**
$ make client && ./client IP

type commands, put/get/run/list/sys

**Server instructions**
$ make server && ./server