

# CPS2008 - Operating Systems and Systems Programming 2 - Assignment Report

Daniel Cauchi

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Tasks</b>	<b>3</b>
2.1	The Clients . . . . .	3
2.2	The Server . . . . .	5
2.3	Helper Files . . . . .	6
2.3.1	Helper.c . . . . .	6
2.3.2	StringManipulator . . . . .	6
2.3.3	MyTime . . . . .	6
2.3.4	Job . . . . .	6
2.3.5	Network . . . . .	7
2.4	The functions . . . . .	8
2.4.1	Run . . . . .	8
2.4.2	Submit . . . . .	8
2.4.3	Chdir . . . . .	8
2.4.4	Status . . . . .	8
2.4.5	Copy . . . . .	8
2.4.6	Kill . . . . .	9
2.5	Limitations and Future Improvements . . . . .	10
<b>3</b>	<b>Conclusion</b>	<b>11</b>

# Chapter 1

## Introduction

The following is the report for the CPS2008 unit assignment, which includes the approach taken to the assignment, which parts were tackled and how the project was managed to maximise the amount completed as much as possible. The task is to implement a server client system where the client asks the server to perform commands for it remotely.

The tasks are found in chapter 2 in a structured order, each part in its own section. A proper example is given for each part of the implementation. A video demonstration is also provided, and can be found linked in the README file. No known bugs exist, since proper error handling and mutexing was implemented throughout.

The Appendix contains the plagiarism form. Disclaimer: Most of the code for sockets was taken from the notes given, but modified to the specific needs of the project.

# Chapter 2

## Tasks

### 2.1 The Clients

The clients are the users of the rex program. They interact with the server, sending it commands to execute. When it is executed, the arguments are analysed to confirm that a proper command has been sent, using the *computeArgs()* method. The commands are the following. Note: Underline means it is an argument, | means or, items enclosed in square brackets means optional

- run "Destination:command + parameters"
- submit "Destination:commands + parameters" [now | day/month/year hour:minute:second]
- copy File name on client "Destination:New file name on destination"
- copy "Destination:File name on destination" New file name on client
- chdir "Destination:New directory"
- kill JobID mode [grace period]
- status

These commands are implemented in **rex.c**

A typical user session might look like the following:

```

danielcauchi@Daniel: /mnt/c/Users/Daniel Cauchi/Google Drive/School/University/Second Year/Sem2/OS2/Assignment/rwx/code/build$ ./executables/rwx run "Daniel:ls -l"
total 52
-rwxrwxrwx 1 danielcauchi danielcauchi 11626 May  8 19:06 CMakeCache.txt
drwxrwxrwx 1 danielcauchi danielcauchi 4096 May  8 19:06 CMakeFiles
-rwxrwxrwx 1 danielcauchi danielcauchi  770 May  8 19:06 CMakeLists.txt
drwxrwxrwx 1 danielcauchi danielcauchi 4096 May  9 07:30 Jobs
-rwxrwxrwx 1 danielcauchi danielcauchi 31632 May  8 19:06 Makefile
-rwxrwxrwx 1 danielcauchi danielcauchi 1654 May  8 19:06 cmake_install.cmake
drwxrwxrwx 1 danielcauchi danielcauchi 4096 May  8 19:06 executables
danielcauchi@Daniel: /mnt/c/Users/Daniel Cauchi/Google Drive/School/University/Second Year/Sem2/OS2/Assignment/rwx/code/build$ ./executables/rwx submit "Daniel:echo Submitted job"
Acknowledged
danielcauchi@Daniel: /mnt/c/Users/Daniel Cauchi/Google Drive/School/University/Second Year/Sem2/OS2/Assignment/rwx/code/build$ ./executables/rwx chdir Jobs
ERROR: incorrect use. Please enter address after command: Success
danielcauchi@Daniel: /mnt/c/Users/Daniel Cauchi/Google Drive/School/University/Second Year/Sem2/OS2/Assignment/rwx/code/build$ ./executables/rwx chdir "Daniel:Jobs"
Done!
danielcauchi@Daniel: /mnt/c/Users/Daniel Cauchi/Google Drive/School/University/Second Year/Sem2/OS2/Assignment/rwx/code/build$ ./executables/rwx run "Daniel:ls -l"
total 0
-rwxrwxrwx 1 danielcauchi danielcauchi 14 May  9 07:30 Job_2.txt
-rwxrwxrwx 1 danielcauchi danielcauchi 131 May  9 07:31 Jobs.txt
danielcauchi@Daniel: /mnt/c/Users/Daniel Cauchi/Google Drive/School/University/Second Year/Sem2/OS2/Assignment/rwx/code/build$ ls
CMakeCache.txt  CMakeLists.txt  Makefile  cmake_install.cmake  executables
danielcauchi@Daniel: /mnt/c/Users/Daniel Cauchi/Google Drive/School/University/Second Year/Sem2/OS2/Assignment/rwx/code/build$ ls
CMakeCache.txt  CMakeLists.txt  Job_2.txt  Makefile  cmake_install.cmake  executables
danielcauchi@Daniel: /mnt/c/Users/Daniel Cauchi/Google Drive/School/University/Second Year/Sem2/OS2/Assignment/rwx/code/build$ cat Job_2.txt
Submitted job
danielcauchi@Daniel: /mnt/c/Users/Daniel Cauchi/Google Drive/School/University/Second Year/Sem2/OS2/Assignment/rwx/code/build$ echo "Appendage" >> Job2FileFromServer.txt
danielcauchi@Daniel: /mnt/c/Users/Daniel Cauchi/Google Drive/School/University/Second Year/Sem2/OS2/Assignment/rwx/code/build$ ./executables/rwx copy Job2FileFromServer.txt "Daniel:Job_2.txt"
danielcauchi@Daniel: /mnt/c/Users/Daniel Cauchi/Google Drive/School/University/Second Year/Sem2/OS2/Assignment/rwx/code/build$ ./executables/rwx run "Daniel: cat Job2FileFromServer.txt"
cat: Job2FileFromServer.txt: No such file or directory
danielcauchi@Daniel: /mnt/c/Users/Daniel Cauchi/Google Drive/School/University/Second Year/Sem2/OS2/Assignment/rwx/code/build$ ./executables/rwx run "Daniel: cat Job_2.txt"
Appendage
danielcauchi@Daniel: /mnt/c/Users/Daniel Cauchi/Google Drive/School/University/Second Year/Sem2/OS2/Assignment/rwx/code/build$ ./executables/rwx status
Job
Host      Command      Type      status      Date      Time
1         Daniel      ls -l      INTERACTIVE FINISHED   9/4/2019   7:30:51
2         Daniel      echo Submitted job  BATCH      FINISHED   9/5/2019   7:30:57
3         Daniel      ls -l      INTERACTIVE FINISHED   9/4/2019   7:31:46
4         Daniel      cat Job2FileFromServer.txt  INTERACTIVE FINISHED   9/4/2019   7:36:11
5         Daniel      cat Job_2.txt      INTERACTIVE FINISHED   9/4/2019   7:36:29

```

Figure 2.1: A screen shot of a client using the system

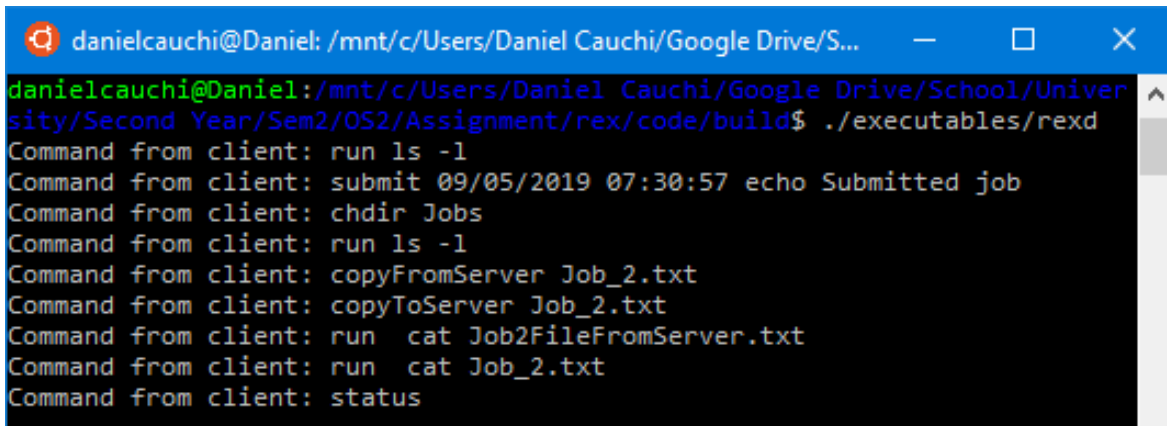
In the above, notice how proper error messages are displayed when a user incorrectly uses the system. Although the network buffer size is limited to 256, looping through the read method allows for longer messages to be *read()* such as *"ls -l"*. The server also sends back useful messages to tell the user whether it has been successful on executing the command properly or not, when the output of a command may not be so clear, such as *submit* or *chdir* or *kill*. Description of how each command works internally is discussed in a later section, however something which is important to know for now is that for each request, the client creates a new socket, makes a connection request to the server then sends the command as plain text.

## 2.2 The Server

The server can run on the system either as a foreground or background process. Four different types of threads appear on the server:

1. The main thread: sits listening on a port. When it receives a request, a new thread is started to deal with the request on a new socket, so that it can keep on listening for more requests
2. The child handler thread (multiple): Handles request made by the client by analysing the string input and performing the command requested
3. The polling thread: Looks at the (sorted) list of submitted jobs, and every second (using the *sleep()* function) polls to check if the current time is bigger than the time for the job, if it is, a child is forked and its output redirected to a file "Job\_*JobID*". A child handler thread is also started to wait for this child
4. Child Handler: Waits for a submitted forked child and updates the jobs file depending on how the child exited

When it receives a command, the server shows the command on screen, this was mostly used for debugging. For the previous session the following output is shown by the server:

A screenshot of a terminal window with a blue title bar. The title bar text is "danielcauchi@Daniel: /mnt/c/Users/Daniel Cauchi/Google Drive/S...". The terminal content shows a series of commands and their outputs. The prompt is "danielcauchi@Daniel:/mnt/c/Users/Daniel Cauchi/Google Drive/School/University/Second Year/Sem2/OS2/Assignment/rex/code/build\$". The commands and outputs are: 1. Command: ./executables/rexd. 2. Command from client: run ls -l. 3. Command from client: submit 09/05/2019 07:30:57 echo Submitted job. 4. Command from client: chdir Jobs. 5. Command from client: run ls -l. 6. Command from client: copyFromServer Job\_2.txt. 7. Command from client: copyToServer Job\_2.txt. 8. Command from client: run cat Job2FileFromServer.txt. 9. Command from client: run cat Job\_2.txt. 10. Command from client: status.

```
danielcauchi@Daniel: /mnt/c/Users/Daniel Cauchi/Google Drive/S...
danielcauchi@Daniel:/mnt/c/Users/Daniel Cauchi/Google Drive/School/University/Second Year/Sem2/OS2/Assignment/rex/code/build$ ./executables/rexd
Command from client: run ls -l
Command from client: submit 09/05/2019 07:30:57 echo Submitted job
Command from client: chdir Jobs
Command from client: run ls -l
Command from client: copyFromServer Job_2.txt
Command from client: copyToServer Job_2.txt
Command from client: run cat Job2FileFromServer.txt
Command from client: run cat Job_2.txt
Command from client: status
```

Figure 2.2: A screen shot of the server output after the user uses the system as in the previous example

The *resetOutputFileDirectory()* method creates a folder to output the Jobs.txt and submitted jobs. A note of the starting directory is kept so if the user executes the chdir command, the server does not get confused as to where it should output the files (Which are kept in a separate *Jobs* folder).

## 2.3 Helper Files

The following files were created to abstract away from the Commands Manager, where most of the functionality is provided. These abstract away most of the functionality of the server to make implementation easier.

### 2.3.1 Helper.c

This file is quite simple in nature. It is meant to be extendible to more functions however for this project all that was needed is an *error()* method which prints an error to *STDERR* and exits the program.

### 2.3.2 StringManipulator

This file is created to deal with strings, which c has limited support for. The header files with its comments should provide enough description for this class:

```
1 //Changes the contents of returnArgs to the string split by the
   delimiters delims
2 //Return args is an array of <bufferAmount> strings
3 void splitStringBy(char *delims, char* splitString, char** returnArgs,
   int bufferAmount);
4
5 //Parser with an entire string as delimiter
6 void splitStringByString(char *string, char* splitString, char**
   returnArgs, int bufferAmount, int bufferSize);
7
8 //Method to remove spaces from the beginning and end of a string
9 void removeSpaces(char* string, int bufferSize);
10
11 // check if string is just spaces
12 bool isEmptyString(char* string, int bufferSize);
13
14 // shift an array of strings by 1 to the left
15 void shiftStrings(char **args);
16
17 // concatenate the array of strings to one string separated by spaces
18 void concatenteStrings(char **strings, char *buffer, int bufferSize);
```

### 2.3.3 MyTime

This file was made to abstract away functions related to manipulating any time constructs. Although it is also made to be extendible, this class only contains one method *timeBiggerThan()* which compares two time structs and returns true if the first parameter is bigger than the second.

### 2.3.4 Job

This file contains functions to deal with the Job struct, and to organise the different lists of jobs. The following defines what a Job is:

```
1 typedef enum {WAITING, RUNNING, TERMINATED, FINISHED} JobState;
2 typedef enum {INTERACTIVE, BATCH} Type;
3
```

```

4 typedef struct{
5     int pid; // the actual pid of the process
6     int jid; // the job pid on the network
7     char host[STRING_BUFFER_SIZE];
8     char command[STRING_BUFFER_SIZE];
9     Type type;
10    JobState state;
11    struct tm dateTime;
12 } Job;

```

Alongside helper functions such as *jobToString()*, *stringToJob* (both used to transfer job info on networks, as well as transfer received jobs from the network to a job structure) and the list of *createJob* method, it contains methods to manipulate the 2 lists of jobs. These 2 lists are:

1. The main list of jobs, stored in the text file. This contains all the jobs in text format, with their details
2. The bath jobs list, which is sorted so that the last item is the next job. When a new bath job is added through the *addBatchJob()* method, it is compared to the last element and placed accordingly, either at the end of the list or behind it, like an insertion sort but adding one element.

**Semaphores** are used a lot in this file, in order to protect access to the jobs files and structures, since different threads may try to access these simultaneously. The semaphores are created and destroyed through the *jobs\_init()* and *jobs\_finish()* methods, respectively. The remove jobs methods remove jobs from the bath jobs list. *AddJob()* and *changeJob* modify the job within the file to show the correct status, based on the Job ID given.

### 2.3.5 Network

This file abstracts network operations, to simplify communication between client and server. It is in charge of creating sockets, sending messages, listening on specific ports, accepting users on sockets as well as connecting clients to ports of a server. It also consists of storing some of the network options such as the network buffer size, and the port number used by the server, as well as the name of the master server (which in this case should be the same as the rexd name since the network has no master, since only one rexd server was implemented). The *Network.h* contains all the method descriptions as comments for any clarifications needed.



## 2.4 The functions

The following are the functions implemented and how the client and server interact when a user uses the system. This functionality is found in the *CommandsManager* file.

### 2.4.1 Run

The run command is executed through the user executing the rex with the arguments 'run' and 'Destination:command+parameters'. The client processes this command and executes *clientRun()*. The client sends the command to the destination if it exists, and then keeps reading from the server until the socket closes.

The server, **on a new thread**, processes the command, and executes *serverRun()*. This command forks a child, and while the child simply executes the given command, the parent adds the job to the job list, and sets its state to running. Afterwards, it waits for the child until it stops. If it was signalled to stop, ie stopped abnormally, then its state is set to '*TERMINATED*', otherwise it is set to '*FINISHED*'. The standard input and output of the forked child are set to the socket which is communicating with the client. When the process stops, the socket is also closed on the parent's child. Important to note is that when manipulating the jobs file, **semaphores** are used so that no 2 threads can be reading or writing at the same time.

### 2.4.2 Submit

The user can submit batch jobs through the submit command. They can get also have a specified time when to execute. Their output is put into Job\_X.txt file. The client is given the command submit with the destination and command+parameters and an optional date or the word 'now' or nothing to specify that the command should be executed at the current time of the client, which may differ from that of the server. The client processes this request, gets current time if needed and sends the server a submit request with the command.

On the server side, a job is added whose status is set to waiting, and a batch job is also added so it can be polled by the server's batch job polling thread. If successful, an 'Acknowledged' message is sent to the client's socket, and the socket is then closed.

### 2.4.3 Chdir

This command changes the directory of a rex server. A destination is specified as well as a new path, by the user of the rex. The client then sends 'chdir *destination*' to the server.

The server then changes its working directory using c's chdir command, after checking that it is a valid one. A confirmation is sent to the client.

### 2.4.4 Status

When the user needs to check which jobs are currently active or which have finished, this command may be executed. The destination would be the master rex if it was the case that there were multiple servers, hence why this command's destination is hard-coded.

The server reads the jobs file, and only returns the necessary data back to the client, who prints this data on screen.

### 2.4.5 Copy

2 versions of this function can occur. Copying a file from a server to the client, or copying a file from a client to the server. In each case, a socket is opened from the client and connects to the server specified. In each case, one side reads from the file specified and sends this over the socket,

meanwhile the other end opens a new file and continuously writes to it after reading the data from the socket.

### 2.4.6 Kill

If a process stalls or is no longer needed, this command can be used by a user to end it. The process number, mode and grace period (if needed) is given. In a multi-server system, the master would check in the jobs file to see who the host of the process is and sends it a message to end the process.

In the case of a submitted process, this can be simply removed from the list of batch jobs being polled, then the job in the jobs text file is set to terminated. If a process is running, a *SIGTERM* is sent to the process if 'soft' is specified, or *SIGINT* if 'hard' is specified. In the case of nice, the thread sleeps for the specified grace period before sending a 'SIGINT'. An appropriate error message is sent to the user. If the process stops, the status of the job is changed to *TERMINATED*.

## 2.5 Limitations and Future Improvements

While great focus was given so that implementation is free of bugs or crashes and proper error handling was made, the cost of this is that some features were not implemented or not fully implemented. These are the following:

- Multiple rexd servers with master - This could be implemented by having the servers be 'clients' to other servers, meaning they would send certain server-to-server only commands such as 'AddJob' from a normal rexd to the master or 'Kill' from the master to the server upon which the specific process would reside.
- Support for commands with user input such as 'sort'. In this case the stdin of the executed command would also be mapped to a socket.
- File transfer of more than just text files. At the moment only text files are properly transmitted over the network, binary files get corrupted

## Chapter 3

# Conclusion

This concludes the report for the Operating Systems and Systems Programming 2 assignment. This report showed the implementation of a remote server and client system. The approach taken was to try and put as many features as possible, while having the drawback of not fulfilling the master slave server functionality.