

CPS2008 - Operating Systems and Systems Programming 2 - rex

Aidan Cauchi



Faculty of ICT
University of Malta

30/05/2019

Contents

1	Introduction	2
2	rex	3
2.1	Overview	3
2.2	rex to rexd Communication	3
3	rex	5
3.1	Overview	5
3.2	Job Registry	5
3.3	Job Scheduler	6
3.4	Servicing Clients	7
4	rex Commands	9
4.1	Run	9
4.2	Submit	9
4.3	Status	10
4.4	Kill	11
4.5	Chdir	11
5	Limitations And Known Bugs	13
5.1	Overview	13
5.2	rex	13
5.3	rex	13
6	Conclusion	15

Chapter 1

Introduction

This report provides a high level overview of the implementation for the 'rex' assignment. First, the report explains the structure of both the 'rex' client and 'rexd' server and then proceeds to delve deeper into how the separate commands are implemented. The goal is to give the reader an idea of how everything works, without going too much in detail. Should one want to look at it in more detail, the implementation is supplied with this report.

Chapter 2

rex

2.1 Overview

This chapter contains an overview of how the *rex* client is designed. The following text shows the basic layout of how to use it. It is important to note that the layout changes depending on the command supplied (as specified in the assignment). In the case where commands don't take any arguments (ex. status) the port must be that of the leader (on localhost).

```
./rex <port> <run|submit|chdir|kill> "<hostname>:<command>" [(date time) | now]
```

2.2 rex to rexd Communication

The *rex* client takes the arguments supplied and transforms them into a format that is understandable by the server. This is dependent on the type of command supplied. Before doing this, it first gets the details of the host from <hostname> and the port from <port>. The port is important here since the implementation was tested locally. Given this, it made sense to only use the hostname "**localhost**" for testing and use different ports to denote different servers (ideally one would have multiple servers communicating on one specific port, to reduce errors).

Since the message sent to the server depends on the command, the following list shows how certain commands are transformed:

- **run:** "run <command> ']"
- **submit:** "submit <command> ']' (<date> <time> | now)"
- **status:** "status" to rexd leader (more info in Chapter 3).
- **kill:** "kill <job-id> <mode> [grace-period]"
- **chdir:** "chdir <path>"

An important thing to note in the submit and run commands particularly is the extra character ']' that gets added after the command. This is essential as it acts as a delimiter, separating the actual linux command from the general rexd command. This allows the server to tokenise on ']' to store the command as a whole.

Once the command is formatted properly, *rex* then proceeds to connect to the server over TCP/IP, where it then sends the formatted command and awaits a response. Figure 2.1 gives a brief overview of how the explained process is carried out.

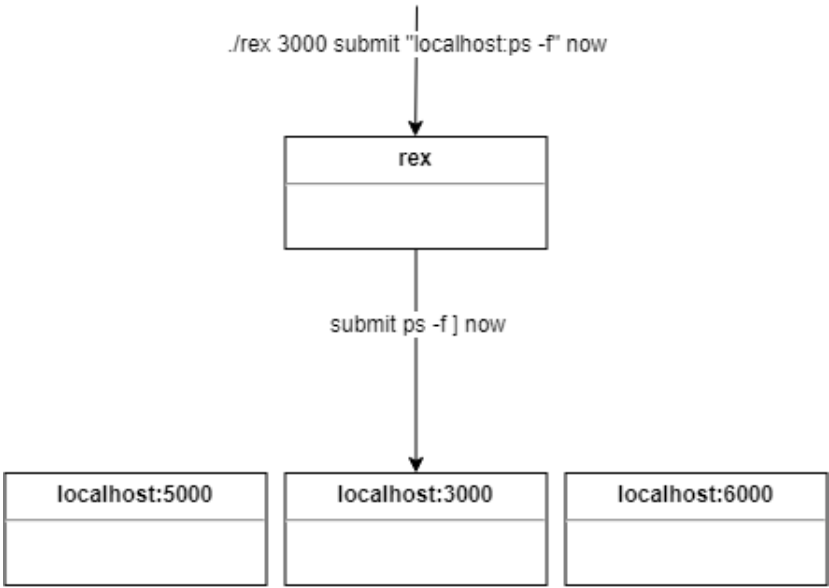


Figure 2.1: Diagram showing simple rex to rexd communication

Chapter 3

rexd

3.1 Overview

This chapter contains an overview of how the *rexd* server is designed. The following text shows the necessary arguments to initialise a server.

```
./rexd <name> <port> [L]
```

Once again the port is specified since testing was done on a local machine. The last argument specifies whether the server is a leader or not (more on this below). It is important that the leader is started first, otherwise it might lead to undefined behaviour.

3.2 Job Registry

The **rexd** server maintains a single consistent job registry across all server instances. This essentially contains a history of all the jobs in the system, whether they are running, scheduled or terminated. The job registry is implemented as a dynamic array, where the index of the job in the array denotes its unique id. Moreover, entries in the job registry can either be added or updated, but never removed. Implementing it as a dynamic array gives the server the luxury of constant access time, thus reducing the access time of a thread updating the registry.

In order to keep a single consistent job registry, it was designed such that this registry is stored on the "Leader". Any amendments would have to go through the leader. As a result, there needed to be some way for the non-leaders to communicate with the leader. Given the fact that these are separate servers, it made sense that the non-leaders would communicate with their leader via a predetermined port. Any time a non-leader wants to update a registry, they simply send a message to the port 10000 (localhost), with the details of the update. This could be either adding a new entry to the registry (denoted by 'a') or updating an existing entry (denoted by a 'u'). This process is described Figure 3.1.

If the server is a leader, it spawns a thread that will run *void* child_listener()*, which blocks until a connection is accepted from another server. So in this case the servers themselves are treated as 'special' clients. The thread then services the request and waits again until another request is received and so on. Thus it can only handle one remote registry update at a time. Figure 3.2 shows the job registry after inputting the commands specified in Figure 3.1.

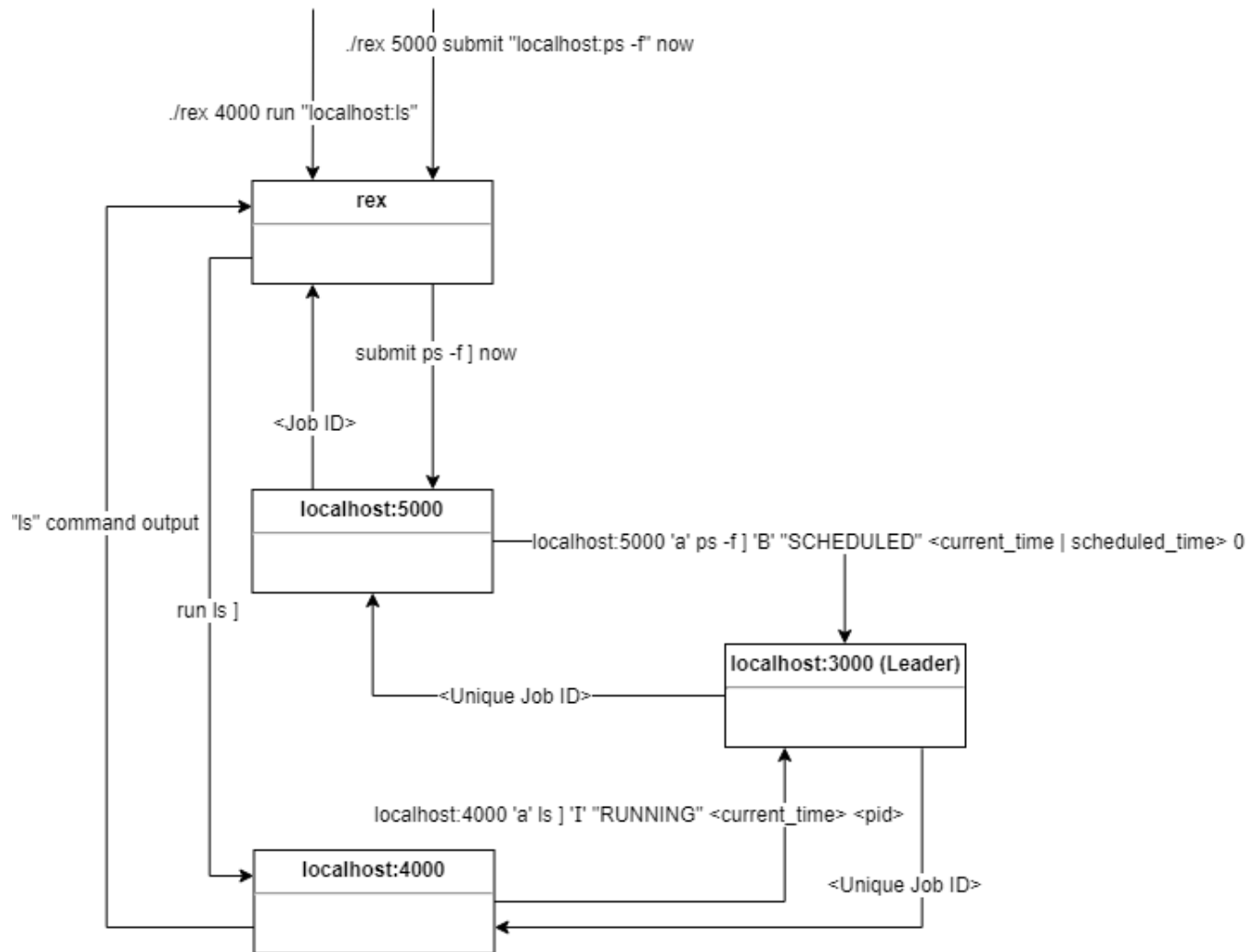


Figure 3.1: How servers communicate with the leader

```
$ ./rex 3000 status
```

Job	Host	Command	Type	Status	Date	Time
0	localhost:5000	ps -f	B	TERMINATED	05/30/19	15:13:08
1	localhost:4000	ls	I	TERMINATED	05/30/19	15:14:36

Figure 3.2: Resulting Job Registry

3.3 Job Scheduler

The *rex* server also makes use of a Job Scheduler. The Job Scheduler is simply an ordered linked list, where the jobs are ordered from earliest time to latest time. When adding an entry to the job scheduler, it is made sure that the order is kept. This is why a linked list was favoured as opposed to an array. To update the linked list, only two pointers need to be changed and order is kept. With an array, almost the

entire array would need to get resorted on every addition.

In order to execute scheduled jobs, the server makes use of a thread executing the *void* job_scheduler()* method. This method simply polls the first job in the job scheduler and compares the current time with its scheduled time. If the scheduled time is less than the current time then the process is executed by first calling *fork()*. In here, the process opens the necessary file based on the job ID of the job, and uses the fork+exec pattern to store the result in the file. File naming uses the convention "jobID.output". Furthermore, after forking, the thread waits for the child to finish to update its status afterwards. It is noted that this is not very efficient. Another thing to note is that the thread sleeps for 100ms after each loop to prevent hogging up the job schedule due to mutexes.

```

1  while(1){
2      sm_wait(&job_sch_access);
3      if(job_sch->head != NULL){
4          sm_signal(&job_sch_access);
5          time(&current_time);
6          //if current time has exceeded scheduled time
7          sm_wait(&job_sch_access);
8          if(difftime(current_time, job_sch->head->sch_time) > 0){
9              int jId = job_sch->head->jId;
10             char* command = job_sch->head->command;
11             sm_signal(&job_sch_access);
12             pid=fork();
13             if(pid < 0){
14                 perror("Error executing submitted job");
15             }
16             //child
17             if(pid==0){
18                 char filename[30];
19                 sprintf(filename,"%d.output",job_sch->head->jId); //to redirect output to file
20                 int filedesc = open(filename,O_CREAT | O_WRONLY);
21                 if(filedesc < 0){
22                     perror("Error creating opening file");
23                     exit(EXIT_FAILURE);
24                 }
25                 process_command(filedesc,command);
26             }
27             else{
28                 sm_wait(&job_sch_access);
29                 delete_next_job(job_sch);
30                 sm_signal(&job_sch_access);
31                 modify_registry('u',arg->job_registry, jId,NULL,NULL,0,"RUNNING",current_time,pid);
32
33                 waitpid(pid,&status,0);
34                 modify_registry('u',arg->job_registry, jId,NULL,NULL,0,"TERMINATED",current_time
35                 ,-1);
36             }
37         }
38         else
39             sm_signal(&job_sch_access);
40     }
41     else
42         sm_signal(&job_sch_access);
43     usleep(10000); //no need to poll on every cycle

```

Listing 3.1: Job Scheduler Loop

3.4 Servicing Clients

Initially, the server starts setting up the required sockets and binds to the supplied port (and port 10000 if it is a leader). It then enters an infinite loop where it calls *accept()* and blocks until a *rex* client connects. After a client is accepted, the server spawns a thread running the *void* handle_client()* method. After spawning the thread, the server instantly goes back to accepting clients to reduce waiting times as much as possible.

The reason a thread is spawned is to separate the handling of a client from accepting new clients. The thread then reads the command supplied by the client, by tokenising and splitting it as necessary.

Moreover, it decides what to do depending on the supplied command. The commands available by *read* are explained later in the report.

The following listing shows the code for the infinite server loop that keeps accepting clients. Note that *thread_param* is simply a struct used to pass arguments to threads.

```
1  while(1){
2      //Accept actual connection from the client
3      newsockfd = accept(sockfd, (struct sockaddr*) &cli_addr, &clilen);
4      if(newsockfd < 0){
5          close(sockfd);
6          error("ERROR on ACCEPT");
7      }
8
9      pthread_t client;
10     thread_param* t = malloc(sizeof(thread_param));
11     if(t != NULL) {
12         //If not null, create thread
13         t->job_registry = job_registry;
14         t->socket = newsockfd;
15         t->sch_jobs = job_sch; //Job schedule
16         t->hostname = hostname;
17         if (pthread_create(&client, NULL, handle_client, (void *)t) != 0) {
18             perror("Error creating client thread");
19         }
20     }
21     else{
22         error("Error passing argument to thread");
23     }
24 }
```

Listing 3.2: Main server loop

It is noted that spawning a thread for every client might not be the most optimal solution, as it add a lot of threads during busy periods, leading to more mutex usage, thus increasing service times (and also switching between threads). Another option could have been to have a pool of threads (ex. 5) which then the server calls one of these threads on every client. This reduces the overhead for creating threads, as they would already be created and at any given time the server is aware of how many threads are present in the system. The downside is that if there are more clients than threads, clients will have to wait.

Chapter 4

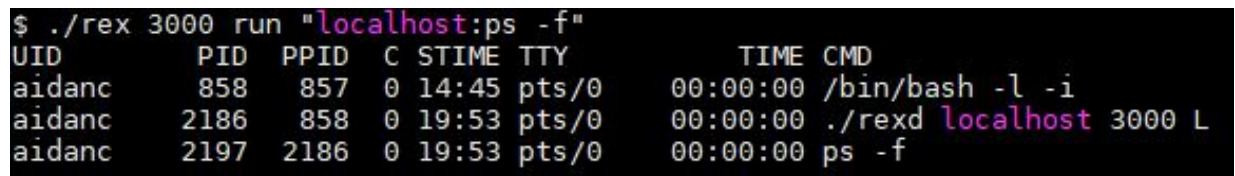
rexd Commands

4.1 Run

The 'run' command takes the following format:

```
./rex <port> run "<hostname>:<command>"
```

The 'run' command takes the command supplied in `<command>` and executes it immediately. The `run_command()` method is responsible for this. All this method does is it forks a process and this process calls `process_command()` where the command is split into an array of arguments. The forked process then iterates over the `PATH` environment variable, calling `execv` on the paths until either the command is found or it exits and informs the user that the command was not found. The reason the process is forked and not executed in the thread is so that the `stdout` and `stderr` of the process can be redirected to the client. Figure 4.1 shows the result of calling "ps -f".



```
$ ./rex 3000 run "localhost:ps -f"
UID      PID  PPID  C  STIME TTY          TIME CMD
aidanc    858   857   0  14:45 pts/0        00:00:00 /bin/bash -l -i
aidanc   2186   858   0  19:53 pts/0        00:00:00 ./rexd localhost 3000 L
aidanc   2197  2186   0  19:53 pts/0        00:00:00 ps -f
```

Figure 4.1: Remotely calling "ps -f"

4.2 Submit

The 'submit' command takes the following format:

```
./rex <port> submit "<hostname>:<command>" [(<date> <time>) | now]
```

The 'submit' command takes the command specified in `<command>`. It also takes the date and time specified and converts them to a `time_t` data type, so it would be easier for the job scheduler to compare times. These details along with some other information are then added to the job registry. Furthermore, the command and date are added to the correct place in the job scheduler. The job ID returned from the job registry is then sent to the user.

When the job scheduler thread notices that it has to execute the command, it forks a process and calls `process_command()`. This works exactly like the `run` command specified before however instead of being redirected to the user, the output is redirected to a file on disk. The file naming convention is the following: `jobId.output`. Figure 4.3 shows the result of submitting an "ls" command.

```

./rex 3000 submit "localhost:ls" 30/05/2019 20:31:30
Job "ls" submitted successfully; Unique ID is: 0
aidanc@MSI-AIDAN:/mnt/d/UNIVERSITY COMPUTING SCIENCE/OS 2/Networks/os-2-assignment/cmake-build-debug$
./rex 3000 status
Job      Host      Command  Type  Status  Date      Time
0      localhost:3000      ls      B  SCHEDULED  05/30/19  20:31:30

aidanc@MSI-AIDAN:/mnt/d/UNIVERSITY COMPUTING SCIENCE/OS 2/Networks/os-2-assignment/cmake-build-debug$
./rex 3000 status
Job      Host      Command  Type  Status  Date      Time
0      localhost:3000      ls      B  TERMINATED  05/30/19  20:31:31

```

Figure 4.2: Submit "ls"

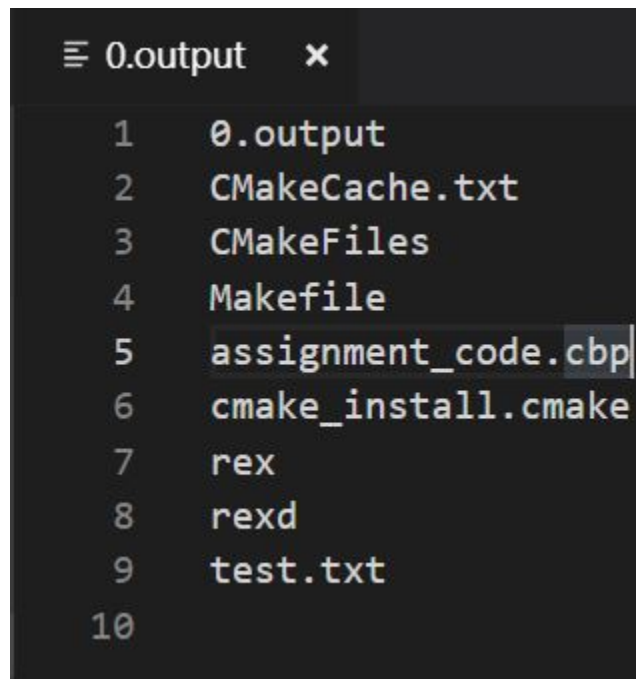


Figure 4.3: Output file

4.3 Status

The 'status' command takes the following format, where <port> is the port where the leader is running on the localhost.

```
./rex <port> status
```

All this command does is it iterates over the each entry in the job registry and outputs every entry. This can be seen by Figure 4.4.

```
$ ./rex 3000 status
```

Job	Host	Command	Type	Status	Date	Time
0	localhost:5000	ls	I	TERMINATED	05/30/19	20:49:23
1	localhost:4000	ps -f	B	SCHEDULED	05/31/19	17:00:00
2	localhost:5000	sort	I	RUNNING	05/30/19	20:50:40
3	localhost:3000	submit-assignment	B	SCHEDULED	05/31/19	23:59:59

Figure 4.4: Status command

4.4 Kill

The 'kill' command takes the following format, where <port> is the port where the leader is running on the localhost.

```
./rex <port> kill <job-ID> <mode> [grace-period]
```

The 'kill' command takes the job ID supplied by the user and performs a lookup in the registry. If the process is running, then its pid should be stored in the job registry as well. The **kill_command()** method takes this pid and sends a signal to it depending on the <mode>. If mode is set to hard, it instantly sends a SIG_KILL. Otherwise if it is set to soft sends a SIG_TERM. If it set to nice, it sends a SIG_TERM, and waits for an amount of seconds denoted by the grace period. If process is still alive by then, then it sends a SIG_KILL. Besides this, if the job is not currently running, then the job scheduler is checked for the same job Id, and is deleted if it exists.

```
aidanc@MSI-AIDAN:/mnt/d/UNIVERSITY COMPUTING SCIENCE/OS 2/Networks/os-2-assignment/cmake-build-debug$ ./rex 3000 status
```

Job	Host	Command	Type	Status	Date	Time
0	localhost:3000	sort	I	RUNNING	05/30/19	21:09:02
1	localhost:3000	bash	I	RUNNING	05/30/19	21:09:40

```
aidanc@MSI-AIDAN:/mnt/d/UNIVERSITY COMPUTING SCIENCE/OS 2/Networks/os-2-assignment/cmake-build-debug$ ./rex 3000 kill 0 hard
Job 0 was successfully terminated
aidanc@MSI-AIDAN:/mnt/d/UNIVERSITY COMPUTING SCIENCE/OS 2/Networks/os-2-assignment/cmake-build-debug$ ./rex 3000 kill 1 nice 5
Job 1 was successfully terminated
aidanc@MSI-AIDAN:/mnt/d/UNIVERSITY COMPUTING SCIENCE/OS 2/Networks/os-2-assignment/cmake-build-debug$ ./rex 3000 status
```

Job	Host	Command	Type	Status	Date	Time
0	localhost:3000	sort	I	TERMINATED	05/30/19	21:10:01
1	localhost:3000	bash	I	TERMINATED	05/30/19	21:10:12

Figure 4.5: Kill command

4.5 Chdir

The 'chdir' command takes the following format:

```
./rex <port> chdir "hostname:directory"
```

This command simply takes the directory indicated by <directory> and uses the inbuilt *chdir()* method. If this doesn't exist, an error is sent back to the client. Otherwise the new directory is sent back to the client for confirmation. This changes the current working directory and is confirmed by the server as it returns runs *getcwd()*, which gets the current working directory, and forwards the result to the client.

```
aidanc@MSI-AIDAN:/mnt/d/UNIVERSITY COMPUTING SCIENCE/OS 2/Networks/os-2-assignment/cmake-build-debug$ ./rex 3000 run "localhost:ls"
0.output
CMakeCache.txt
CMakeFiles
Makefile
assignment_code.cbp
cmake_install.cmake
rex
rex.d
test.txt

aidanc@MSI-AIDAN:/mnt/d/UNIVERSITY COMPUTING SCIENCE/OS 2/Networks/os-2-assignment/cmake-build-debug$ ./rex 3000 chdir "localhost:.."
/mnt/d/UNIVERSITY COMPUTING SCIENCE/OS 2/Networks/os-2-assignment
aidanc@MSI-AIDAN:/mnt/d/UNIVERSITY COMPUTING SCIENCE/OS 2/Networks/os-2-assignment/cmake-build-debug$ ./rex 3000 run "localhost:ls"
CMakeLists.txt
README.md
build.sh
cmake-build-debug
dynArray.c
dynArray.h
job_scheduler.c
job_scheduler.h
jobs.c
jobs.h
rex.c
rex.d
rex.d
semaphore.c
semaphore.h
```

Figure 4.6: Chdir command

Chapter 5

Limitations And Known Bugs

5.1 Overview

This section contains information about the various parts of the program whose functionality is either limited or has some known bugs.

5.2 rex

- Very little error reporting: The rex client makes very little effort to inform the user about badly structured commands and crashes instead.
- Output is never redirected to stdout of client. The **read()** command is inside a loop which then prints the buffer line by line, mimicking the effect of redirection.
- 'run' commands don't redirect stdin. Commands like 'sort' and 'cat' (without any arguments) will not work on rexd.
- Commands that don't take a hostname (ex. status) have "localhost" hardcoded within the client, thus assuming the leader will always reside on localhost.

5.3 rexd

- Due to the having almost everything in threads, sometimes the rexd client doesn't function as expected. Exact cause is unknown.
- Mutexes have been utilized as needed, however there might be some unprotected accesses due to negligence.
- The 'kill' command doesn't work on non-leaders as they have no way of getting the pid of the process from the registry.
- Although the assignment specification didn't specify this, the port has to be included with every command due to the host being always localhost since this was tested only locally. Ports mimic the idea of multiple servers.
- The job scheduler waits for the currently executing job to finish before continuing. While this was done to prevent spawning more threads, it can halt job scheduling if the child hangs.
- The 'copy' command is not implemented, even though it is specified in the assignment.

- Non-leaders always assume the leader is on the localhost at port 10000. There is no way of checking whether this is actually the case and can result in undefined behaviour if leader is not online.
- If a leader crashes, it is up to the user to re-elect another one.

Chapter 6

Conclusion

This assignment highlighted the many challenges and obstacles present when working with network sockets and also when dealing with concurrency. Obstacles such as how to protect accessible variables from other threads by the use of mutexes, how to debug a program that deals with lots of threads and forked processes. Moreover, there was also the decision of what data structures to use, in order to represent the state of the job registry as well as that of the job scheduler. Additionally, one also had to look at how 'rex' and 'rexd' communicate with each other, how they convert and format certain messages for ease of use etc.