Grid-FTP Client Server Application

Edoardo Predieri 1697836

SAPIENZA
UNIVERSITÀ DI ROMA

SAPIENZA
UNIVERSITÀ DI ROMA

Grid-FTP Client Server Application

Facoltà di ingegneria dell’informazione, informatica e statistica
Dipartimento di ingegneria informatica, automatica e gestionale “Antonio Ruberti”
Laurea Magistrale in Cybersecurity

Edoardo Predieri
Matricola 1697836

INDEX

CHAPTER 1: Introduction

1.1. PURPOSE OF THE PROJECT

The aim of the project is to build a distributed infrastructure based on the FTP protocol in order to guarantee a higher level of performance and reliability.

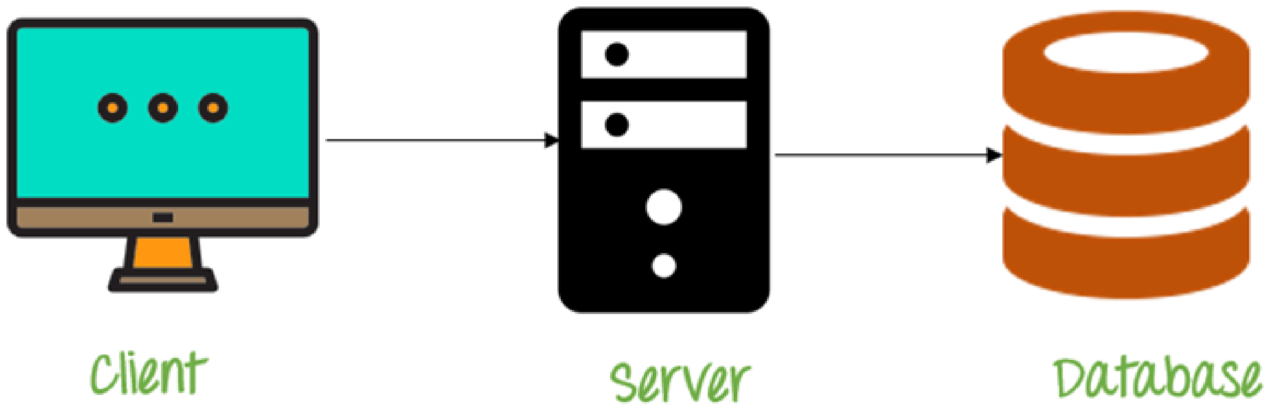
1.2. REFERENCE ARCHITECTURE

The architecture on which our application is based, in software engineering, is defined as: three-tier architecture, which consists of three "levels" (called tiers):

- *Presentation Tier,* which constitutes the level of interface with the client;
- *Business Tier,* which represents the application logic;
- *Data Tier,* which makes up the management of persistent data on the database.

These nodes interact with each other following the general lines of the client-server paradigm using well-defined interfaces, which allow the total independence and modularity of the various levels.

Using this technique, data cannot be accessed directly, but only through server-side processing.



1.3. PROJECT ORGANIZATION

Grid-FTP Client Server Application presents three different entities developed in C language (64 bits).

- *The Metadata Server*: is the true core of the application that manages client requests and organizes the management of the space of the various DRs
- *DRs servers*: that physically store and manage client files
- *Clients*

Very important note: socket C does not allow you to simulate fake IP on the same machine, so the whole project will not be based on IP but on Ports (a different one for each DRs, server and client). If you want to test the application in a real distributed system you can easily change the door parameters with the IP).

CHAPTER 2: Collection of Requirements

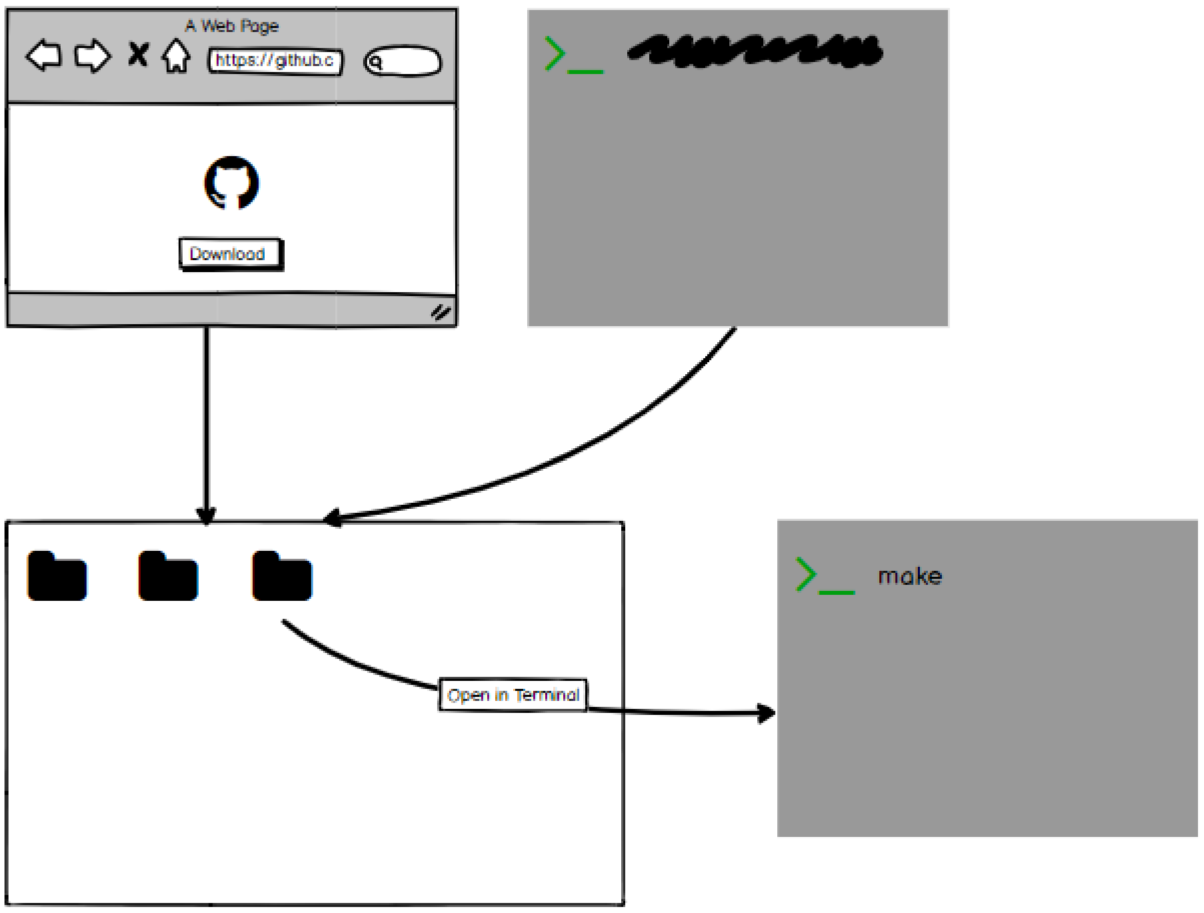
2.1. REALITY OF INTEREST

Grid-FTP Client Server Application is based on the FTP protocol, one of the first network protocols used for transferring data and files between hosts. The main idea behind my application is scalability, in fact, being a distributed system, the files of the various clients will be scattered among the various DRs, making the system lighter, faster and almost free of bottlenecks.

2.2. COMPLETE SPECIFICATION

2.2.1 DOWNLOAD AND START APPLICATION

The application can be easily downloaded from my GitHub site through the command $ git clone https://github.com/edoardoPredieri/Grid-FTP-Client-Server-Application or directly from the web page. Once downloaded you can start the various components via Makefile (Attention: clients must be started after the MDS server).



```
>_ make
```

2.2.2. REGISTRATION/ACCESS PHASE

Once the client is started, the MDS server will check through its id if the client is already registered or not showing, depending on the eventuality, these two messages:

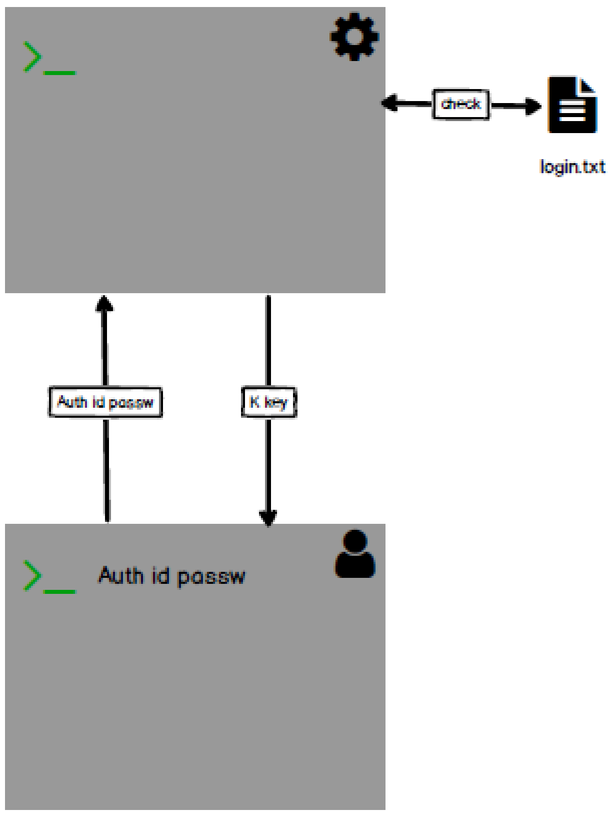
- Not registered case:

```
Connection established!
Welcome to Grid FTP Client-Server Application made by Edoardo Predieri. I w
ill stop if you send me QUIT, Press 'Send'
>
Server response: Welcome Client n:1, to register type: Auth <Userid> <Passw
ord>
```

- Registered case:

```
Connection established!
Welcome to Grid FTP Client-Server Application made by Edoardo Predieri. I w
ill stop if you send me QUIT, Press 'Send'
>
Server response: Welcome Client n:1, to login type: Auth <Userid> <Password
>
```

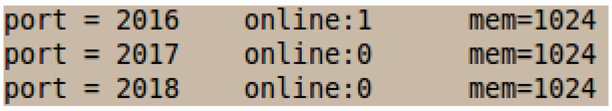
At this point the client will send its own command (in the form *Auth id passw*) and the server will generate a key and send it to the client as a reply (if the user is not registered he will save it in the login.txt file ").



2.2.3. GETDR COMMAND

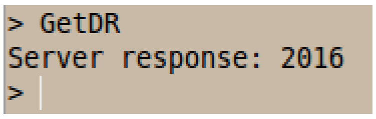
The GetDR command allows you to see which DRs servers are currently available.
Operation:

1) The client types GetDR
2) The MDS server reads the DR.txt file on the dr.txt file that knows
3) The MDS server will try to establish a connection with all DR and print the information on the screen: DR port, connection status and available space (in the following example only the DR1 is connected)

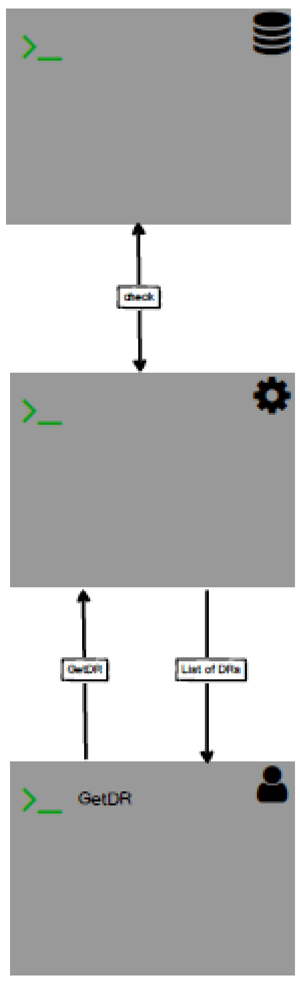


```
port = 2016    online:1      mem=1024
port = 2017    online:0      mem=1024
port = 2018    online:0      mem=1024
```

4) The MDS server the list of online DRs to the client



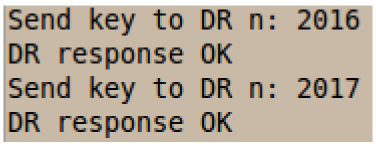
```
> GetDR
Server response: 2016
>
```



2.2.4. PUT COMMAND

The process of saving a file on the system follows the following steps:

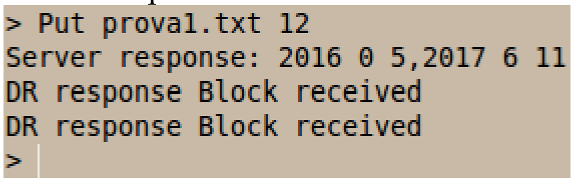
1) The client types the command: *Put fileName* size and saves the file size in fileSize.txt (it will later be used for the Get)
2) The MDS server receives the command and the key and verify which are the DRs currently online sending to the latter the key of the client.



```
Send key to DR n: 2016
DR response OK
Send key to DR n: 2017
DR response OK
```

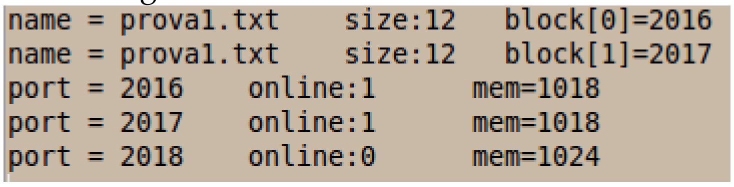
3) The DRs save the client's file in the key.txt file
4) The MDS depends on how many Dr are online calculates the number of blocks to send to each DR (the idea is to divide the file into blocks and save them in an equal number (also counting odd dimensions) in the various DR currently online) and sends it to the client in the following form: DR port initial block final block):

Client Response:



```
> Put prova1.txt 12
Server response: 2016 0 5,2017 6 11
DR response Block received
DR response Block received
>
```

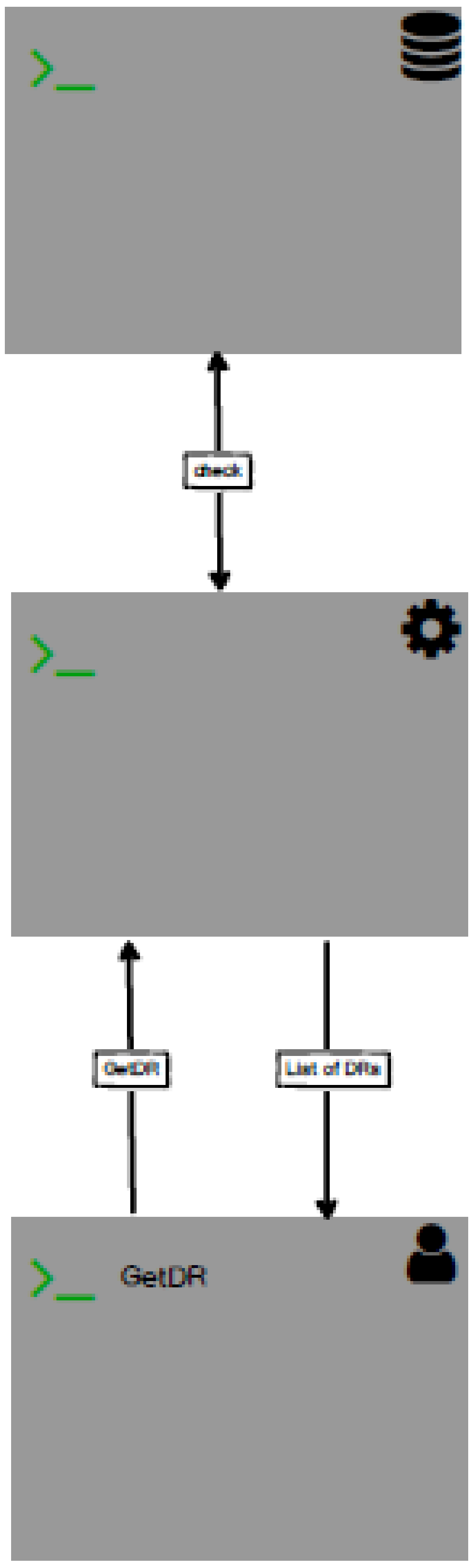
Server Log:



```
name = prova1.txt    size:12   block[0]=2016
name = prova1.txt    size:12   block[1]=2017
port = 2016    online:1      mem=1018
port = 2017    online:1      mem=1018
port = 2018    online:0      mem=1024
```

5) the client divides the file and sends the various blocks as suggested by the MDS server
6) the DRs receive the block and the key and, if the key is present in key.txt, save the block (the name of block file is a simple hash (nameFile, key) -> key||nameFile

This system guarantees a high level of security, in fact only a client can interact with a DR only if it has first authenticated on an MDS server.

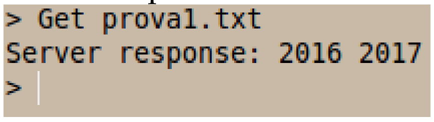

check
GetDR
List of DRs
GetDR

2.2.5 GET COMMAND

To recover a file from the system, proceed as follows:

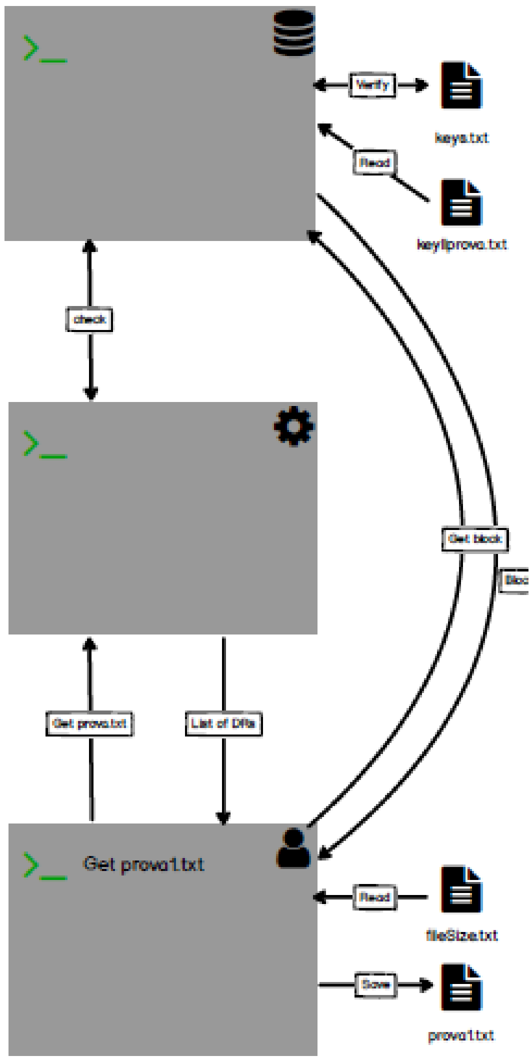
1) The client types the command Put fileName
2) The MDS server returns the ports of the DR in which the client has previously saved the file

   Client Response:



```
> Get prova1.txt
Server response: 2016 2017
>
```

3) The client reads the file fileSize.txt the file size and starts reclaiming the file blocks to the various DRs (sending it the key)
4) The DRs at each request get check if the key is known and resolves the hash function
5) Received all the blocks re-compiles them into a single file



2.2.6. REMOVE COMMAND

To delete a file on the system, follow this procedure:

1) The client sends the Remove fileName command to the MDS server and the key
2) The MDS server sends to all DRs that have blocks of those files the message Remove and calculate the new available space for each DRs.

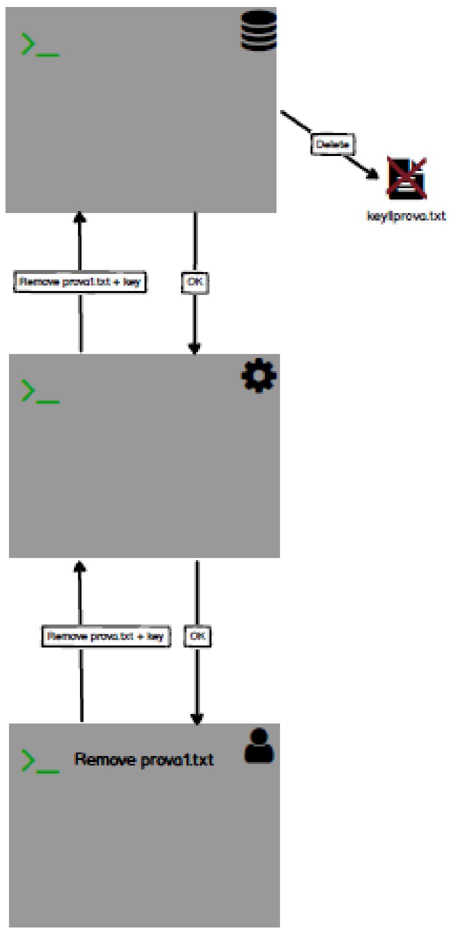
Server log:

```
Send Remove msg to DR n: 2016
DR response OK
name = prova1.txt    size:12   block[0]=2016
name = prova1.txt    size:12   block[1]=2017
Send Remove msg to DR n: 2017
DR response OK
port = 2016    online:1      mem=1024
port = 2017    online:1      mem=1024
port = 2018    online:0      mem=1024
```

3) The DRs check the key and resolve the hash and delete the file.

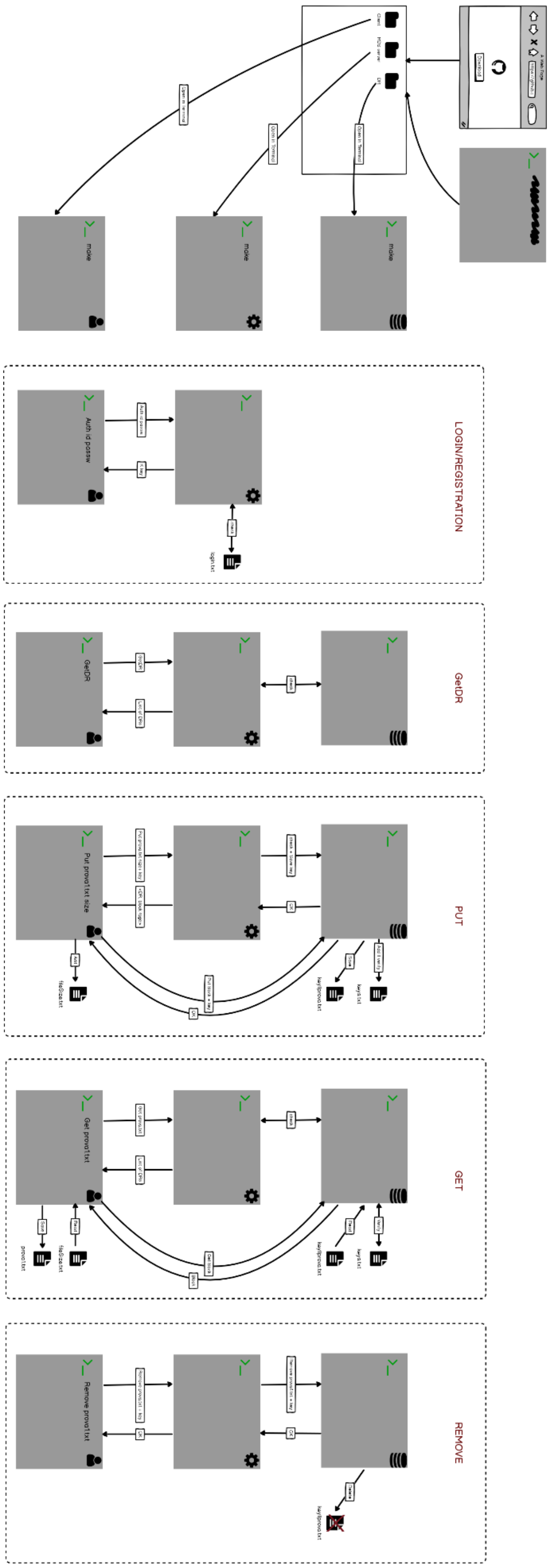
Client response:

```
> Remove prova1.txt
Server response: OK
>
```



2.3. MOCK-UP

Below is the entire Mock-Up scheme of the application:



CAPTHER 3: Technologies and methodologies used

3.1. TECHNOLOGIES

The Grid-FTP Client Server application is totally based on C (64 bits) language.

3.1.1. COMMUNICATION PART

The bone of communication system is based on the library: <*sys/socket.h*> which permits to create and manage sockets in easy way.

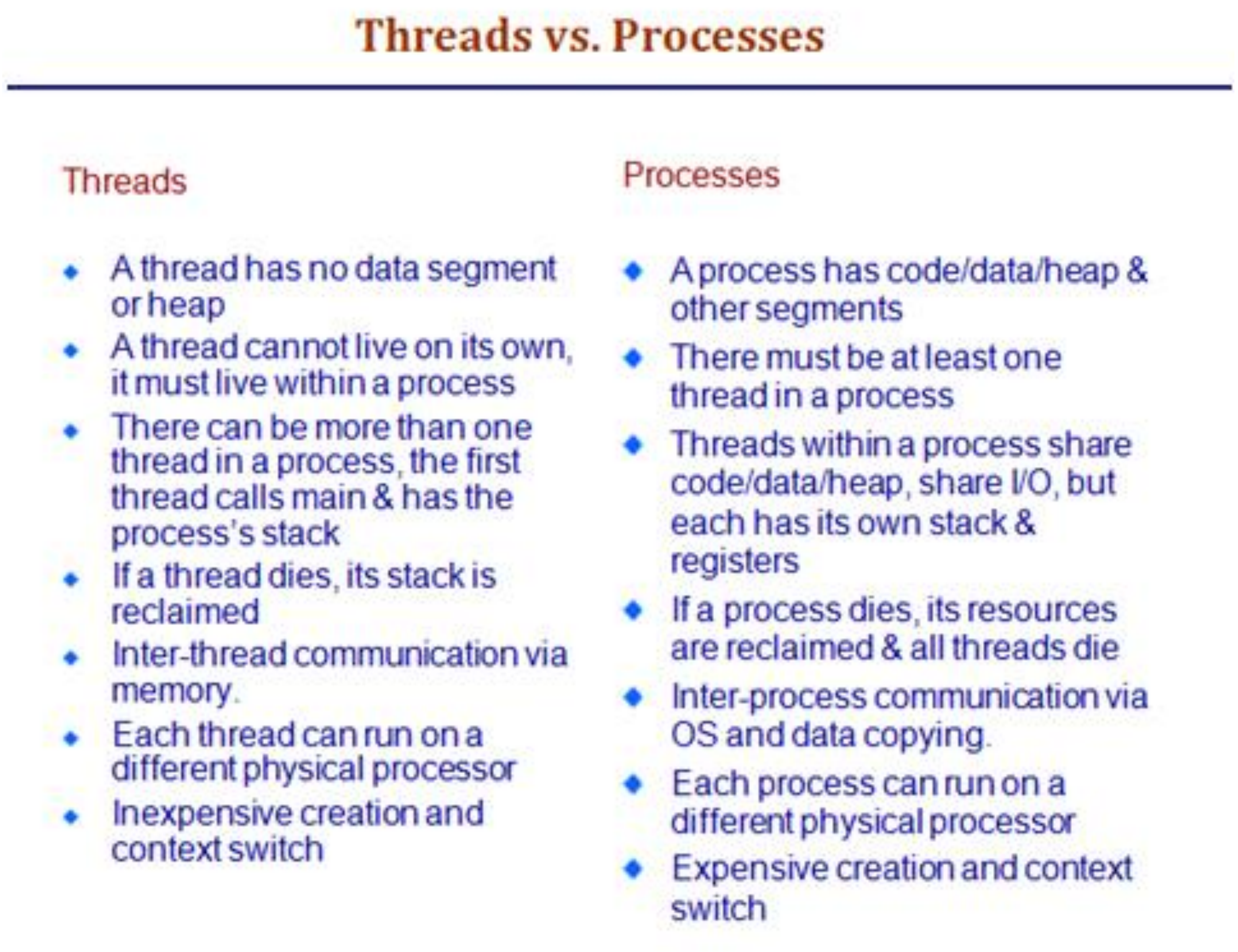
3.1.1.1. FOCUS ON THREADS

For my application I have decided to use Threads instead Processes for these reasons:

- Threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.
- Context switching between threads is generally less expensive than in processes
- Threads are faster to create, delete and manage
- The overhead (the cost of communication) between threads is very low relative to processes.

3.1.1.2. SEMAPHORES

To guarantee mutual exclusion I used Posix semaphores (being my Multithreaded application and not Multiprocess).
These semaphores are based on the *semaphore.h* library and are managed by the primitives: sem_init, sem_wait and sem_post.



Threads vs. Processes

Threads

- A thread has no data segment or heap
- A thread cannot live on its own, it must live within a process
- There can be more than one thread in a process, the first thread calls main & has the process's stack
- If a thread dies, its stack is reclaimed
- Inter-thread communication via memory.
- Each thread can run on a different physical processor
- Inexpensive creation and context switch

Processes

- A process has code/data/heap & other segments
- There must be at least one thread in a process
- Threads within a process share code/data/heap, share I/O, but each has its own stack & registers
- If a process dies, its resources are reclaimed & all threads die
- Inter-process communication via OS and data copying.
- Each process can run on a different physical processor
- Expensive creation and context switch

CAPITOLO 7: DEPLOYMENT AND VALIDATION

This last chapter we will show, first of all, how to install and use the application and, subsequently, we will specify how the test phase took place.

7.1. DEPLOYMENT

The entire project is available on the GitHub page:
https://github.com/edoardoPredieri/Grid-FTP-Client-Server-Application

7.2. TEST AND VALIDATION

Below we list various stress tests to which my application can be submitted and for each will undermine the result and the process to be able to perform it:

- Client errs access credentials: no problem the MDS will send you the request again (for a maximum of two) PASS (Open the client and type random access credential (after a registration)
- Client make a mistake or write random words: the MDS will respond with NOK message PASS (Open the client and write GetDRR)
- Client do a Get of an "not Put file" or an inexistent file: the MDS will respond with an error message PASS (Open the client and type Get prova44.txt)
- Client do a Remove of an "not Put file" or an inexistent file: the MDS will respond with an error message PASS (Open the client and type Remove prova44.txt)
- Client do a Put of an inexistent file: the MDR will respond with an error but reduces the size of DRs : almost PASS (Open the client and type Put sss 12)
- Client crashes or quits: no problem, when it will reconnect his session will be restored and all the files that previously saved on the DR will be available: PASS (Open a Client and type Put prova1.txt 12, after type QUIT or ctr c, re-open the client and type Get prova1.txt or Remove prova1.txt)
- If a DR crashes: no problem, the files it contained will not be available, but as soon as it comes back online it will be again : PASS (Open a client and a DR, in the client type Put prova1.txt 12, Crash and Restart the DR, and in the client type Get prova1.txt).
- If there are the size of file is odd and the number of online DR is even: no problem the system manages these cases and divides the extra blocks without problems. PASS (open 3 DR and in a client type Put prova3.txt 20)

- If the MDS server crash: it is a problem the clients connected lost their file history and must be reactivate, but the DRS continue work normally: almost PASS
- If the number of DRs change a client can be re-upload a file without problem: almost PASS (no problem in practice but the size of DRs after the remove can be wrong) (open 1 DR and on client Type Put prova1.txt 12, Open DR2 and in the client Type Put prova1.txt 12)

BIBLIOGRAPHY

Course material System Programming: Prof Richelli.
Course material Sistemi di Calcolo: Prof Demethrescu.