TABLE OF CONTENTS

CHAPTER NO.	TITLE	PAGE NO.
1.	Introduction	1
2.	Software Requirement Specification	2
	2.1 Functional Requirements	2
	2.2 Non-Functional Requirements	3
	2.3 Hardware Requirements	3
3.	System Design	4
	3.1 High level Design	4
	3.2 Detailed Design	5
4.	System Implementation	9
	4.1 User Interface	9
	4.2 Screen Shots	12
5.	System Testing	13
6.	Conclusion & Future Enhancements	15
7.	References	16

1. INTRODUCTION

The main purpose of RudiFS is to build a rudimentary Client-Server oriented network file system. Here the Server serves out the files which are accessed by Clients over the network. It should provide the Clients with facilities to connect to the Server, know the files on Server, open a file, read a file, write to a file and close a file.

It is basically a ‘rudimentary’ network file system, which assumes no disruption on network path. This is an ideal project for those who begin the client-server concept study and implementations. It involves many other aspects that add value to the project & gives a brief knowledge of concepts involved to beginners.

It also involves many other file management concepts like Client-Server connection peculiarities, network utilization, Remote Procedure Calls, API or user defined Library creation, custom Marshalling and De-marshalling procedures, Multi-threading concepts, Critical Section managements and many more.

The project requirements are provided by Mr. Joseph Elvin Fernandes, Senior Software Engineer, RedHat.

2. SOFTWARE REQUIREMENT SPECIFICATION

Any project to be successful, it should start with requirement analysis. Each requirements of the project should be specifically organized in a document called Software Requirement Specification (SRS). The purpose of the document is to specify the external requirements of the system. It is meant for use by the developers and will be the basic of validating the final system.

2.1 <u>Functional Requirements</u>:

Functional requirements specify what the system is supposed to accomplish. It defines the functions of the system and its components.

The RudiFS system should provide the services to connect to the server, seek files on the server, open, read, write and close a file on the server.

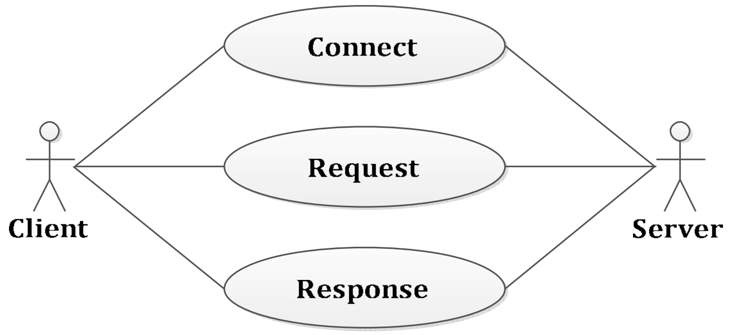


Fig 2.1: Use case Diagram of RudiFS

- ❖ <u>Connect to Server</u>: It should allow a user to connect to a specified Server with its IP address and the port.
- ❖ <u>List:</u> It should provide a list of files available on the connected server.
- ❖ <u>Open:</u> A client requested file should be opened.
- ❖ <u>Write:</u> User should be able to write to the opened file.
- ❖ <u>Close:</u> The opened file should be closed.

2.2 <u>Non-functional Requirements</u>:

The system has shared resources such as open file tables. So to maintain the integrity of the system, the access to these resources by multiple clients should be controlled.

As a basic system, it assumes no disruption on network path. Thus network errors are not considered here. There are many readymade frameworks for such cause.

Minimize the network traffics as much as possible is an important point of concern. So all the aspects of marshalling and communications are structured in such a way to minimize the size of data communicated.

The rudiFS system is not mission critical system, the importance is given more to the concept demonstration rather than performance measure such as response time, fault tolerance, etc.

2.3 <u>Hardware Requirements</u>:

<u>Hardware Requirements:</u>

✓	PROCESSOR:	1 GHz Processor or Above.
✓	RAM:	1 GB or Above.
✓	OTHER:	Network Connection (For 2 or more System connections)

<u>Software Requirements:</u>

✓	OPERATING SYSTEM:	Fedora.
✓	COMPILER:	GCC Compiler.
✓	EDITOR:	Codelite (Recommended)
✓	CODE VALIDATOR:	Checkpatch.pl

3. SYSTEM DESIGN

3.1 <u>High level Design</u>:

The High level design explains the architecture that would be used for developing the software product. It provides an overview of entire system, by identifying the main components that would be developed & the interface of the system.

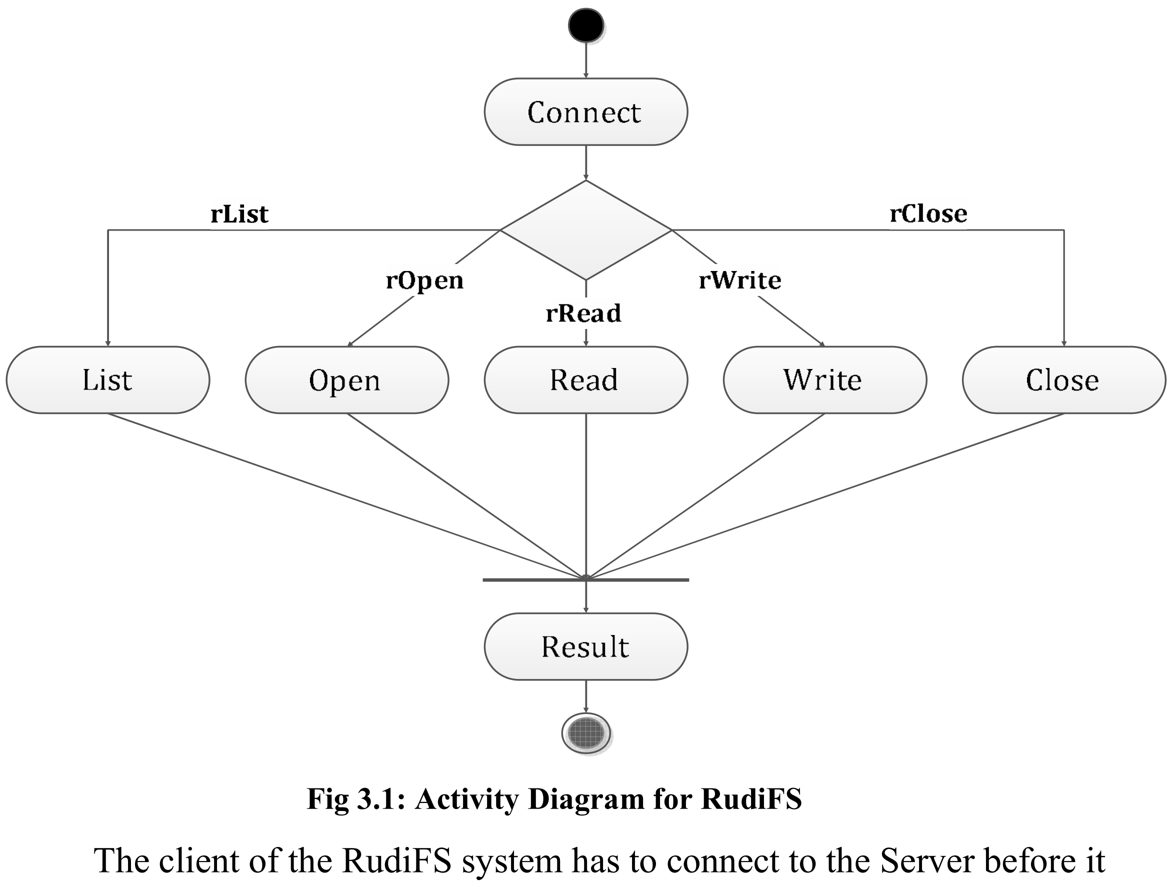


Fig 3.1: Activity Diagram for RudiFS

The client of the RudiFS system has to connect to the Server before it can make any request. Once it is connected, the client can request the server for the services it provides.

3.2 **Detailed Design:**

In detailed design, the internal design of each of the modules is specified. The detailed design phase is often called as Logic Design, because it deals with the internal design of the modules or how the specification of the module can be satisfied, is decided. During the detailed design, developer can concentrate on one module at a time.

There are mainly four major modules in RudiFS.

The four major modules of RudiFS are-

❖ Client API Layer:

Client API provides an interface to the client to access the system. It provides a list of functions which is similar to the default system calls. Thus it is easy to understand and user friendly.

Client API design -

- ✓ init_rClient (host_name , port)
- ✓ rList (file_list)
- ✓ rOpen (filename , mode , file_Structure)
- ✓ rRead (file_Structure, buffer, size)
- ✓ rWrite (file_Structure, buffer, size)
- ✓ rClose (file_Structure)

❖ Client RPC Layer:

The Client RPC layer responsible to capturing the client call and provides the custom marshalling and de-marshalling model. This includes packing bits into a buffer in some discipline & sending to the server through the network. Also, unpacking the resultant buffer received from the server.

Each of the API functions have their own way of packing and unpacking.

Function code	Length of File name	File Name	File Open Mode	r_file Structure

Fig 3.2.1: Marshalling of rOpen call

Above figure (Fig 3.2.1) shows the marshaled rOpen call, which is sent through the network. Here r_file structure will hold the resultant file representative for further references. Similarly,

Function code	Length of Size	Size

Fig 3.2.2: Marshalling of rRead call

Function code	Length of Size	Size	Buffer

Fig 3.2.3: Marshalling of rWrite call

Function code	r_file Structure

Fig 3.2.4: Marshalling of rClose call

❖ Server RPC Layer:

The Server RPC layer is responsible for receiving the network stream of bytes sent from the client, de-marshal it and reform the call and marshal the result of call back to the client.

The marshalling in this case is different for succeeding call & failed call.

Success indicator	Open call result file reference structure

Fig 3.2.5: Marshalling on Open Call Success.

Failure indicator	Error code

Fig 3.2.6: Marshalling on Open Call failure.

The same pattern is followed throughout for other calls as well.

❖ Server API Layer:

It contains set of functions which fulfill the client request on the server side. They are the remote procedures, called by client with service request through the RPC mechanism.

It includes the Multi-threaded concept for request handling and locking mechanism for critical sections.

The operation on server side begins by checking whether an entry for the requested file exist in open file table. Based on its results, in combination with the requested operation, tasks are performed.

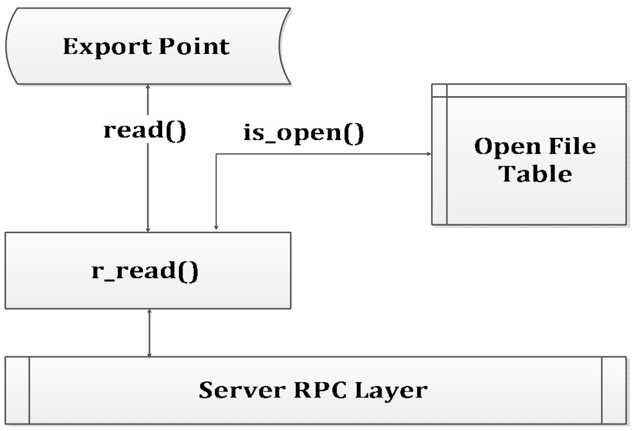


Fig 3.2.7: File read operation on the server side.

The write operation also follows the similar format for writing into the file in Export Point.

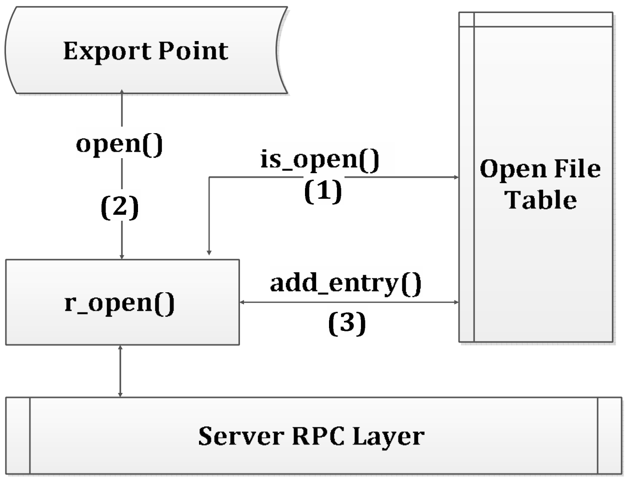


Fig 3.2.8: File open operation on the server side.

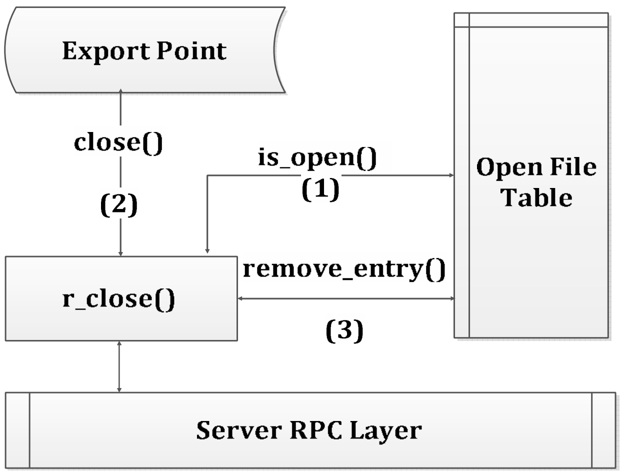


Fig 3.2.9: File close operation on the server side.

4. SYSTEM IMPLEMENTATION

4.1 <u>User Interface</u>:

List of coding standards are provided. Implementation is according to these rules, in order to make it available to large group of Open-Source community.

<u>Coding Guidelines:</u> ***(The Ten Commandments)***

- All the structures / functions / #includes / #defines etc. which are common to client and server should be in rudi_common.h
- The Client API should be exposed only via rudi_client.h and all the other functions should be either static or not exposed to the consumer of the API.
- The code should be compiled using Makefiles only.
- To test the Client API's a test program should be written.
- Code should be shared only and only via github.com
- All the .c and .h files should pass the **checkpatch.pl** test. Checkpatch.pl checks for coding standards like indentation, extra spaces etc.

  ```
  $> ./checkpatch.pl -f <.c or .h file_path>
  ```

  There should be no warning or errors when the code is inspected by checkpatch.pl.
- Code should be well commented and documented. Also the code should be well indented. Each indentation is 8 spaces. Codelite is a good IDE for this purpose.
- Should have appropriate variable names. Names like i, j, k, temp, a, b are strictly not allowed.

- Code should be modular in nature with no memory leaks and core dumps. i,e should be tested well with all scenarios. Automated tests are welcomed.
- The Code should be submitted incrementally. i.e. in incremental patches and not in huge commits.

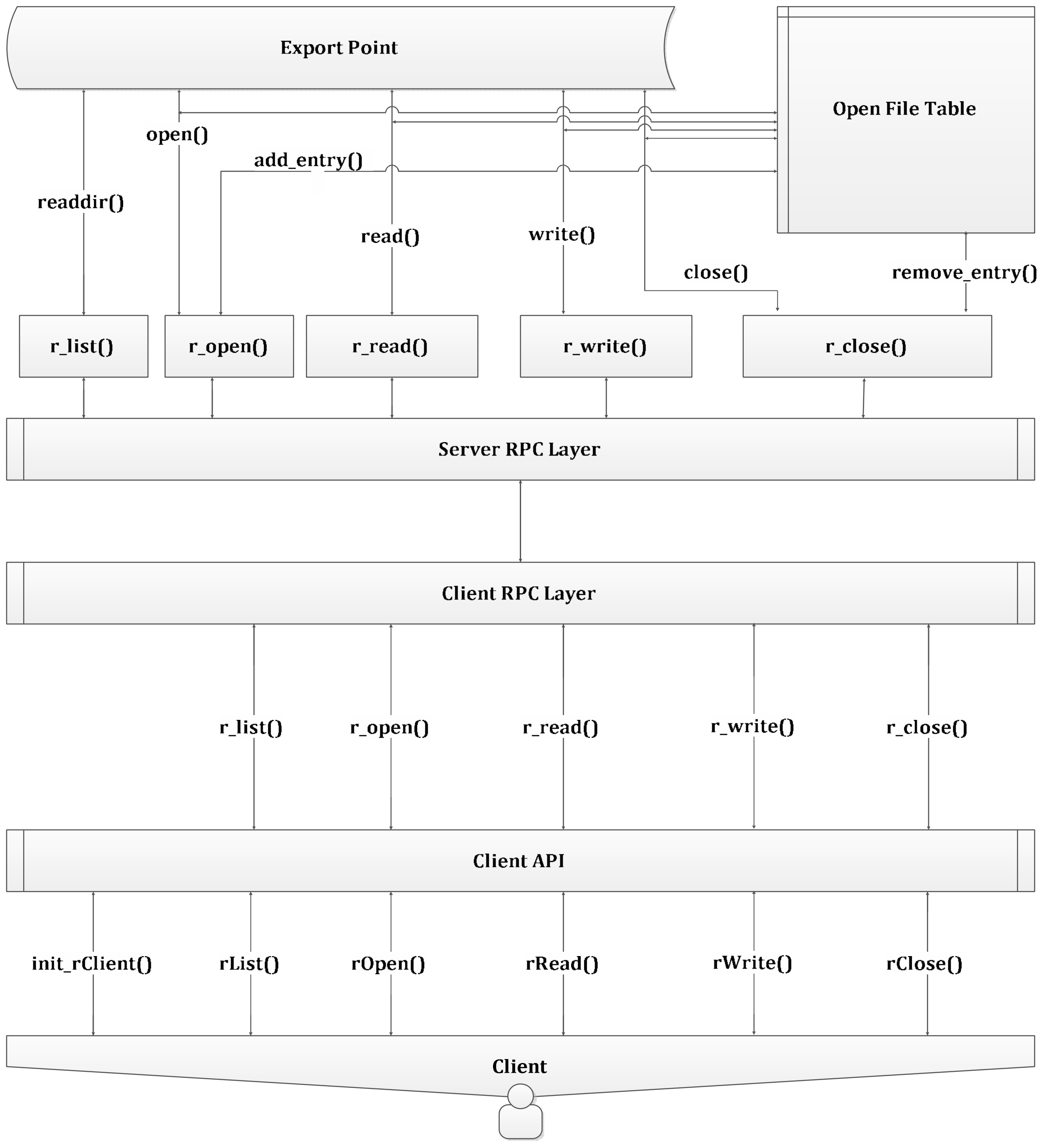
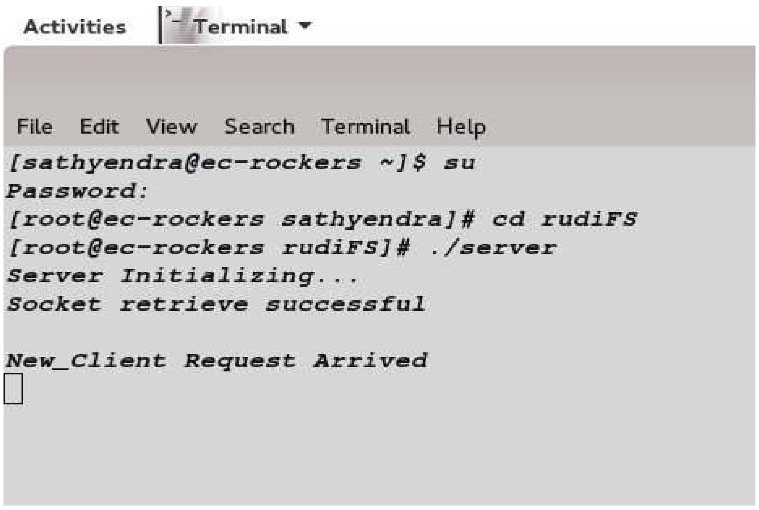


Fig 4.1: An Over View of RudiFS system implementation.

Above figure (Fig 4.1) shows the overall implementation structure of the system.

4.2 Screen Shots:

The server program provides notification for every connection established. It is shown in the figure 4.2.1.

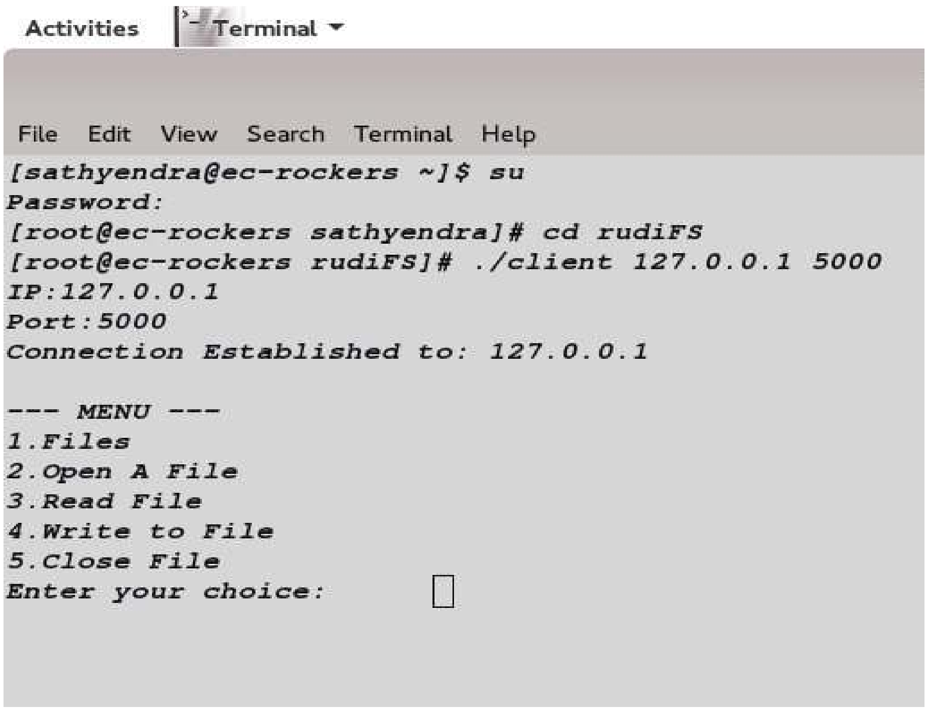


```
[sathyendra@ec-rockers ~]$ su
Password:
[root@ec-rockers sathyendra]# cd rudiFS
[root@ec-rockers rudiFS]# ./server
Server Initializing...
Socket retrieve successful

New_Client Request Arrived
```

Fig 4.2.1: Server Interface.

The client program provides the list of services that the server provides. The client interface is as shown in figure 4.2.2.



```
[sathyendra@ec-rockers ~]$ su
Password:
[root@ec-rockers sathyendra]# cd rudiFS
[root@ec-rockers rudiFS]# ./client 127.0.0.1 5000
IP:127.0.0.1
Port:5000
Connection Established to: 127.0.0.1

--- MENU ---
1.Files
2.Open A File
3.Read File
4.Write to File
5.Close File
Enter your choice:
```

Fig 4.2.2: Client Interface.

5. SYSTEM TESTING

✓ Tests are done with multiple clients requesting the server at a time.
✓ One thread is used for each of the clients.

Test	Description	Result
Multiple clients opening the different files.	This regulates the open file table to contain multiple inode entry. This has been tested for different opening & closing orders for files.	Irrespective of the order of opening, open files were closed & respective entries are deleted.
Multiple clients opening the same file.	This regulates the open file table to contain multiple fd entry for each inode. This has been tested for different opening, closing orders for files & the inode entry deletion for the only one open file.	Success. Each client's fd is maintained separately.
Reading a non-opened file.		Appropriate error message is displayed.
Closing a non-opened file.		Appropriate error message is displayed.
Multiple clients reading the same file.	Different clients read the file part by part, by giving read size.	Each of the clients received their respective read request with correct reading. (Exact content that is expected) As each client has unique fd.

Test	Description	Result
Read size more than content.	Client requests with the read size, which is more than the available content of the file.	Server returns the available content & a message stating the total read (only available) size.
Writing to a file opened in read mode		Error indicating the file open mode is shown.
Writing to a file with large amount of data.	The data is split in the background & chunk by chunk, is sent to the server and written to file.	The write operation success in shown.
Opening a non-existing file		Error indicating file not found is displayed.
Opening more than one file	The system allows only one file to be opened at a time.	Error message regarding an open file is displayed.
Reading more than the file content.	Client after reaching end of the file, again sends a read request.	A message stating the end of file is the result.
Request after closing the connection. (client side)	Once an open file is closed, if one more request is made.	A new connection to the server is established & the request is passed to the server.
Request after closing the connection.(server side)	The server is not up, if a request is made.	As no one is there to listen, the error message regarding the connection is displayed.
List of files.	Client requests for the available files.	Server responds with available file. (If export point is empty, then an error message is shown)
Invalid read request.	Invalid read size provided.	Invalid requests are filtered & error messages are shown.

6. CONCLUSIONS AND FUTURE ENHANCEMENTS

The project titled “RudiFS” has been designed and developed in “Fedora 21”. Many concepts are added, that are beneficial for user understandability. Many complex concepts are presented here for the better understandability and their usage.

It also provides the knowledge and best practices of coding. The coding standard is very helpful and must in case of Open source community.

Currently only basic operations are implemented for the system. Many other concepts such as file ownership, copy & paste, upload & download and many more can be added to the existing system.

The reader can add his own ideas into this and make it a large learning repository. This makes a way to learning by sharing & contributing to the system, which will make the system a handy tool for a beginners.

7. REFERENCES

1. Code Repository: https://github.com/Sathyendra12/Assignements
2. System call Reference: http://www.tutorialspoint.com/unix_system_calls/
3. Linux Command Reference: http://linux.die.net/man/
4. C Language reference: “Advance Linux Programming” by Mark Mitchell, Jeffrey Oldham and Alex Samuel.