POCSD Final Project Report

Author: Andrew Diaz

***Abstract*—**This paper gives an in-depth look into a client-server design that was implemented in order to develop a fault tolerant networking system. The core of the design was implementing a client in python programming language to interface with four servers that serve as a single storage unit from the perspective of the client. The design approach taken is similar to RAID1 where redundancy is used. This particular design uses a redundancy factor of two. If one of the servers go down, the client is able to fetch the data from another server. The overall design emulates a linux file system where the client provides an interactive window to the user with the capabilities to do the following file system commands: mkdir, create, mv, read, write, status, rm

I. Problem Statement

A storage system without fault tolerance has severe consequences. If all data is stored in a single location there can be issues if the system goes down (e.g., lightning strikes and power goes off) because if a user try to retrieve data from the server, the server is down and cannot provide that information. Another issue may arise in a non-fault tolerant system when a load-balancing strategy is not implemented in the design. This affects performance in that one server can be overloaded with request and the overall performance suffers. This paper discusses a design that aids to the problems previously mentioned. A single storage system is implemented through the use of four servers. Two of those servers hold the same information so in the case a server goes down, the information is able to be retrieved from the other server. Also, this design implements a load-balancing technique so that all four servers are used in a balanced way when receiving request from a user in the client.

II. Design of Servers

The servers are designed to interface with their local file system. We use the "SimpleXMLRPCServer" python package to assign a port number and run the server. A class called "MyFuncs" was created to perform file system operations for the server object. The following bullets discuss the functions implemented in the "MyFuncs" class:

- **def __init__(self) :** This function initializes the file system for the server object and allows the server to interface with the local file system.
- **def mkdir(self, path) :** Allows the server object to interface with the "mkdir" function in the local file system. This command makes new directories with the given path.
- **def create(self, path) :** Allows the server object to interface with the "create" function in the local file system. This command creates files.
- **def mv(self, old_path, new_path) :** Allows the server object to interface with the "mv" function in the local file system. This command moves file from one directory to another.
- **def read(self, path) :** Allows the server object to interface with the "read" function in the local file system. This command reads data from files.
- **def write(self, path, data) :** Allows the server object to interface with the "write" function in the local file system. This command writes data into files.
- **def status(self) :** Allows the server object to interface with the "status" function in the local file system. This command displays directories, files, and data that have been written to the file system. This command is used on the server end to track information and debugging purposes.
- **def rm(self, path) :** Allows the server object to interface with the "rm" function in the local file system. This command deletes files and directories.

The final lines of the server script register the "MyFuncs" class to the server object and then the server begins running. The first server has been created.

A four-server system is implemented by creating different server files containing the same script with the only difference being that each server file is assigned a different port number, but are all in sequence. A server file and all file system files are stored in the same directory. This is replicated for all four server files. Individual terminal threads need to be spawned in order to run each server.

III. Design of Client

A. Objective

This system implements a client that provides and interactive window and allows a user to remotely access a file system and perform file system operations. The client's objective is to provide a single storage system through the use of four servers, where two of the servers are used to hold mirrored data from another server. The client acts as a middle-man between all four servers and integrates them as if it were a single storage system. In addition, the client implements a load-balancing technique so that a single server is not overly worked and requested are not lob-sided. This design pairs server01 with server02 and server03 with server04. This means server01 mirrors server2 and server04 mirrors server04.

B. Client Setup

The client begins by printing in the interactive window the number of servers the system will run. For this system, four servers will run.

Next, the user is asked to provide the starting port number of the servers. The client saves each server's port number and concatenates it the local host http address. The servers are looked up over the network through the xmlrpc library and a proxy is created for each server. If the servers are not already up and running then there will be an error in the client. The

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

client prints “connection established from server<#>.py” in the interactive window after establishing contact with each individual server.

A ‘server<#>_not_avail_flag’ is created to keep track if a server goes down and is instantiated to zero. A variable ‘tracker’ is instantiated to zero to balance requests between servers. When ‘tracker’ is zero it will prioritize requests for [server01, server02], and when it is one it will prioritize requests for [server03, server04]. Another variable ‘sub_tracker’ is instantiated to zero and has the job of distributing requests between each server pair, so that back-to-back requests alternate between server01 and server02 or server03 and server04.

C. Client Main Loop

The main loop begins by accepting a command from the user. Once a command has been entered, it is mapped to the corresponding RPC. Through each iteration the client contacts the servers and keeps track of whether a server went down or not. if a server went down the interactive window will print “Connection lost to server<#>.py”. The following sections go through each command and how the corresponding RPC is distributed between servers.

D. Client “create” Command

Once the “create” command is entered by the user the client uses multiple exception handlers to prioritize which servers to create files in. If the “tracker” variable is zero then select an RPC to server01 and server02. The “sub_tracker_1_2” variable is then used to prioritize whether to create in server01 or server02 first. If sub_tracker_1_2 is zero then create in server01 first. If sub_tracker_1_2 equals one then create in server02 first. If the path is not found in server01 or server02 then an exception is thrown and the handler attempts to create in server03 and server04. Similarly, a “sub_tracker_3_4” variable is used to prioritize whether to create in server03 or server04 first. If sub_tracker_3_4 is zero then create in server03 first. If sub_tracker equals_3_4 one then create in server04 first.

If the “tracker” variable is one then select an RPC to server03 and server04. The “sub_tracker_3_4” variable is then used to prioritize whether to create in server03 or server04 first. If sub_tracker_3_4 is zero then create in server03 first. If sub_tracker equals one then create in server04 first. If the path is not found in server03 or server04 then an exception is thrown and the handler attempts to create in server01 and server02. Similarly, a “sub_tracker_1_2” variable is used to prioritize whether to create in server01 or server02 first. If sub_tracker_1_2 is zero then create in server01 first. If sub_tracker_1_2 equals one then create in server02 first.

E. Client “mkdir” Command

Once the “mkdir” command is entered by the user the client uses multiple exception handlers to prioritize which servers to make directories in. If the “tracker” variable is zero then select an RPC to server01 and server02. The “sub_tracker_1_2” variable is then used to prioritize whether to make directory in server01 or server02 first. If sub_tracker_1_2 is zero then make directory in server01 first. If sub_tracker_1_2 equals one then make directory in server02 first. If the path is not found in server01 or server02 then an exception is thrown and the handler attempts to make directory in server03 and server04. Similarly, a “sub_tracker_3_4” variable is used to prioritize whether to make directory in server03 or server04 first. If sub_tracker_3_4 is zero then make directory in server03 first. If sub_tracker_3_4 equals one then make directory in server04 first.

If the “tracker” variable equals one then select an RPC to server03 and server04. The “sub_tracker_3_4” variable is then used to prioritize whether to make directory in server03 or server04 first. If sub_tracker_3_4 is zero then make directory in server03 first. If sub_tracker_3_4 equals one then make directory in server04 first. If the path is not found in server03 or server04 then an exception is thrown and the handler attempts to make directory in server01 and server02. Similarly, a “sub_tracker_1_2” variable is used to prioritize whether to make directory in server01 or server02 first. If sub_tracker_1_2 is zero then make directory in server01 first. If sub_tracker_1_2 equals one then make directory in server02 first.

F. Client “mv” Command

Once the “mv” command is entered by the user the client uses an exception handler to prioritize which servers to make requests to. The “sub_tracker_1_2” variable is then used to prioritize whether to move file in server01 or server02 first. If sub_tracker is zero then move file in server01 first. If sub_tracker_1_2 equals one then move file in server02 first. If the path is not found in server01 or server02 then an exception is thrown and the handler attempts to move file in server03 and server04. Similarly, a “sub_tracker_3_4” variable is used to prioritize whether to move file in server03 or server04 first. If sub_tracker_3_4 is zero then move file in server03 first. If sub_tracker_3_4 equals one then move file in server04 first.

G. Client “write” Command

Once the “write” command is entered by the user the client uses multiple exception handlers to prioritize which servers to write in. If the “tracker” variable is zero then select an RPC to server01 and server02. The “sub_tracker_1_2” variable is then used to prioritize whether to write in server01 or server02 first. If sub_tracker_1_2 is zero then write in server01 first. If sub_tracker_1_2 equals one then write in server02 first. If the path is not found in server01 or server02 then an exception is thrown and the handler attempts to write in server03 and server04. Similarly, a “sub_tracker_3_4” variable is used to prioritize whether to write in server03 or server04 first. If sub_tracker_3_4 is zero then write in server03 first. If sub_tracker_3_4 equals one then write in server04 first.

If the “tracker” variable equals one then select an RPC to server03 and server04. The “sub_tracker_3_4” variable is then used to prioritize whether to write in server03 or server04 first. If sub_tracker_3_4 is zero then write in server03 first. If sub_tracker_3_4 equals one then write in server04 first. If the path is not found in server03 or server04 then an exception is thrown and the handler attempts to write in server01 and server02. Similarly, a “sub_tracker_1_2” variable is used to prioritize whether to write in server01 or server02 first. If

sub_tracker_1_2 is zero then write in server01 first. If sub_tracker_1_2 equals one then write in server02 first.

H. Client "read" Command

Once the "read" command is entered by the user the client uses an exception handler to prioritize which servers to make requests to. The "sub_tracker_1_2" variable is then used to prioritize whether to read file in server01 or server02 first. If sub_tracker is zero then read file in server01 first. If sub_tracker_1_2 equals one then read file in server02 first. If the path is not found in server01 or server02 then an exception is thrown and the handler attempts to read file in server03 and server04. Similarly, a "sub_tracker_3_4" variable is used to prioritize whether to read file in server03 or server04 first. If sub_tracker_3_4 is zero then read file in server03 first. If sub_tracker_3_4 equals one then read file in server04 first.

I. Client "rm" Command

Once the "rm" command is entered by the user the client uses multiple exception handlers to prioritize which servers to remove files from. If the "tracker" variable is zero then select an RPC to server01 and server02. The "sub_tracker_1_2" variable is then used to prioritize whether to remove file in server01 or server02 first. If sub_tracker_1_2 is zero then remove file in server01 first. If sub_tracker_1_2 equals one then remove file in server02 first. If the path is not found in server01 or server02 then an exception is thrown and the handler attempts to remove file in server03 and server04. Similarly, a "sub_tracker_3_4" variable is used to prioritize whether to remove file in server03 or server04 first. If sub_tracker_3_4 is zero then remove file in server03 first. If sub_tracker_3_4 equals one then remove file in server04 first.

If the "tracker" variable equals one then select an RPC to server03 and server04. The "sub_tracker_3_4" variable is then used to prioritize whether to remove file in server03 or server04 first. If sub_tracker_3_4 is zero then remove file in server03 first. If sub_tracker_3_4 equals one then remove file in server04 first. If the path is not found in server03 or server04 then an exception is thrown and the handler attempts to remove file in server01 and server02. Similarly, a "sub_tracker_1_2" variable is used to prioritize whether to remove file in server01 or server02 first. If sub_tracker_1_2 is zero then remove file in server01 first. If sub_tracker_1_2 equals one then remove file in server02 first.

J. Main Data Structure: Hash Map

The main data structure used in the client (can be thought as the coordinating data structure) was a hash map. This is implemented using the 'dictionary' structure in python. The goal of this data structure is to save directories and files (the key), and map them to the corresponding server that holds the directory or file. The server number is the value for each key-value pair in the hash map. The hash map registers the absolute path of any directory or file. This data structure is integrated in the client and serves any command that attempts to access/create an absolute path to verify that it does not already exist. Also, if the directory or file already exists, the client can access the value in the hash map and immediately know which servers hold the data and contact those servers.

IV. TESTING

Testing was completed by running all four server files and single client file in separate terminal threads. The "status" command used in the interactive window displayed on the server terminals how data has change since the most recent command was processed. The following discusses how each command was tested.

A. Testing the "create" command

Testing the "create" command has two aspects. The first aspect is verifying if the path given already exist. If true the file is created on the servers where the path exist. If the path does not already exist then the client creates the file in any pair of servers that are scheduled to write data to next. Using the "status" command it can be seen which servers hold the file that was created.

B. Testing the "mkdir" command

Testing the "mkdir" command has two aspects. The first aspect is verifying if the path given already exist. If true the directory is created on the servers where the path exist. If the path does not already exist then the client creates the directory in any pair of servers that are scheduled to write data to next. Using the "status" command it can be seen which servers hold the directory that was created.

C. Testing the "mv" command

Testing the "mv" command has two aspects. The first aspect is verifying on which servers the 'old_path' and 'new_path' reside. After the command is executed the server that holds the 'new_path' is updated with the file in 'old_path'. This is verified by using the 'status' command and verifying the file has been moved to the correct path.

D. Testing the "write" command

Testing the "write" command has two aspects. The first aspect is verifying on which server pair the file that is written resides. The second aspect is after the command is executed, the 'status' command is used to verify if the data was written to the correct file (on the server end).

E. Testing the "read" command

Testing the "read" command has two aspects. The first aspect is verifying on which server pair the file that is read resides and logging the data that is stored in the file. The second aspect is after the command is executed, the client should display the data that was read and it should be the same as on the server end.

F. Testing the "rm" command

Testing the "rm" command has two aspects. The first aspect is verifying on which server pair the file/directory that will be deleted resides. The second aspect is after the command is executed, the 'status' command is used to verify if the file/directory was removed (on the server end).

V. CONCLUSION

This paper covers a client-server system design with fault tolerance through the use of redundancy. The client acts as a middleman between all four servers and coordinates

requests between them in a balanced way. When a server goes down, the server replica takes its place and handles all the load. When all servers are running each server is scheduled in a fair way so that no server is overloaded or requests are biased toward a server. The overall design emulates a linux file system where the client provides an interactive window to the user with the capabilities to do the following file system commands: mkdir, create, mv, read, write, status, rm. The goal of the overall system is (from the client perspective) to integrate multiple storage units, and implement them as a single storage unit with fault tolerance. This was successfully accomplished.