

Distributed File System

Project Report
Submitted by
Ishan Kapoor, Sarat Sreepathi.
{ikapoor, sssreepa}@ncsu.edu.

Description:

The Design on the Client Side:

System Call Overriding:

The client side loads a library before loading any of the other system libraries. Using the LD_PRELOAD environment variable allows us to accomplish this. This environment variable is used to specify the path of our library which is loaded first. This enables us to override the default system calls and we create hooks (pointers) to the real system call functions and use those functions whenever an operation involves the local file system.

If the operation involves the remote file system, then the client library initiates a connection to the server, sends the request, gets back the results and passes them back to the program that called the library system call.

Modular Design:

The client library is built using a modular design where the major functions are delegated to individual programs in the design phase itself, i.e., 'csock.c' takes care of the initial connection setup and termination, 'ccommand.c' is responsible for communicating the requested operations to the server and getting the results back. Due to the modular design and consistent interfaces, we believe that our system is extendable and amenable to improvements without a big overhaul.

The Server Model and Basic Design:

The final design is an **iterative stateful server**; here each client sets up a new TCP connection with the DFS file server. The first thing a client is expected to send out to the server is a COMMAND data structure which includes the following data definitions.

```
struct COMMAND{
enum      operation  cmd;
char      fname[MAX_FILENAME];
mode_t    mode;
int       desc;
int       size;
int       flags;
int       perms;
};
```

This structure is interpreted by the server and appropriate action is then taken to execute the clients command. If the command is 'open', 'close', 'mkdir' or 'rmdir' the results of those operations are sent back as received from the locally executed command.

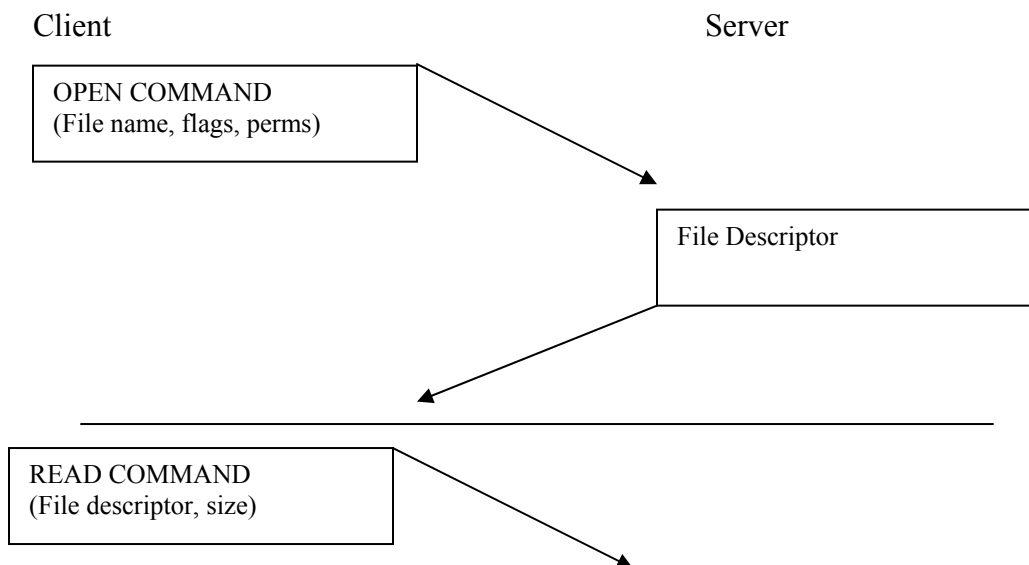
If the command is 'read' the client is passed the data read from the file descriptor mentioned in the `COMMAND` data structure along with the size in bytes to be read from the file. The server then passes along data to the client in the following data structure `DATA`.

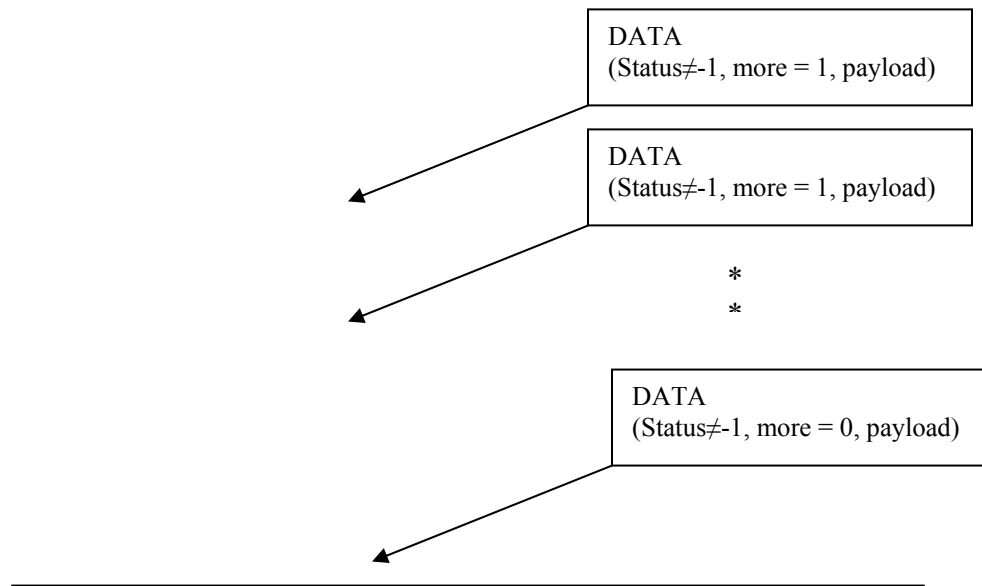
```
struct DATA{
int    length;
int    status;
char   more;
char   payload[MAX_PAYLOAD];
};
```

The server immediately mentions a -1 in the `status` indicating that the read was a failure (the client could pass an invalid descriptor). The `more` variable is set to 0 indicating there are no more data messages to be sent and the client quietly stops receiving. The client then sends a zero byte message to the server to terminate the connection. If the file is legitimate a read command is made to the local file. All reads are performed in units of `MAX_PAYLOAD`. The number of bytes read is accumulated until they match the size provided by the client or we read less than `MAX_PAYLOAD`, which indicates that end-of-file was arrived at. There is no response message associated with the read command since the client can determine the bytes received by monitoring the `more` variable.

If the command from the client was a write, the server begins to accept the `DATA` structure from the client. The server quietly listens and receives all of the data that client has to send. The client can signal the end of the data transmission when `more` is set to zero. The server keeps writing each chunk it receives from the client, down at the appropriate offset in the file. If it encounters an error it takes a note of it and sends the error back to the client.

A sample message exchange:





The horizontal line indicates a TCP connection being brought down

A similar scheme works for the writing part except there is one additional message sent over to the client by the server indicating what bytes actually got written.

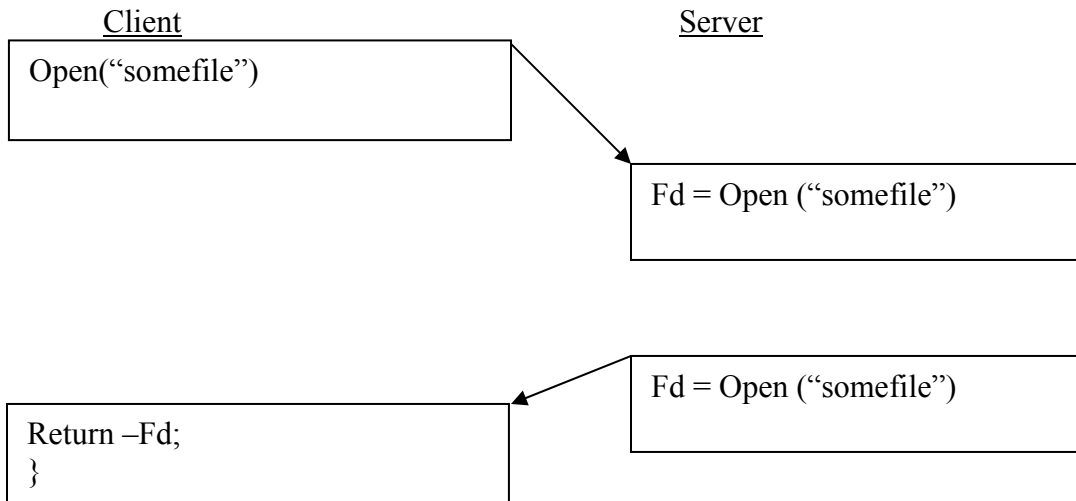
The Iterative Design:

In this case the server goes in a while loop accepting a new connection from the client in the beginning and interpreting the commands sent. The server then sends the slave socket descriptor and the command structure itself to a function that negotiates with the client until its time for the TCP connection to be broken down.

This design can be easily modified to support a concurrent server. Simply accept a new connection and fork a process, also closing the appropriate descriptors in the child/parent process.

The remote directory is a global namespace and is visible to all users who access our DFS. Everything written to a “/r” absolute path will get written/opened/read from the remote server.

If the client open several files locally and on the remote file server simultaneously there needs to be a mechanism in place which can distinguish between the two file descriptors. In our design we chose to have a mapping of every remote files descriptor to its corresponding negative value. This scheme greatly simplifies the task of identifying whether the file descriptor represents a remote or local file. However this may pose problems when the application checks for a file descriptor value less than zero for error.



Design Choices:

This is perhaps the most important part of the whole DFS design. There were many options before us for coding the server

1. A stateful iterative server.
2. A stateful concurrent server (using forked server processes)
3. A stateful concurrent server (using select)
4. A stateless concurrent server(using forked server processes)

The concurrency using forking is uncalled for in this case because, we're not really expecting a lot of concurrent load on our server. For our scale of operations it is more efficient to run the server iteratively for each request.

In some cases according to [1] the iterative server can actually be more efficient since forking involves a bit of system overhead that might not be warranted in every application

One way to build the file system is to transfer the whole file down to the client and perform read write operations and commit the changes by sending the modified copy back. If no changes were made the file need not be sent back to the server.

In our design we chose to do carry out the operations remotely and get the results back from the server. This operations was in coherence with our stateful iterative server design In short the iterative stateful was the most straightforward design.

All communication between the client and server uses TCP connections.

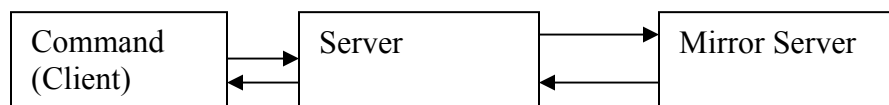
Additional Features:

Caching:

We plan to implement client side caching where the client reads ahead for data that might be requested by the user/application. This can be done by storing the last DATA packet and reading the cached block from local memory instead of sending a new request.

Replication :

Additionally we plan to replicate the entire server's state to a different machine as the user is issuing commands. This ensures robustness. The server may go down at any instant but since every operation is mirrored, we have a more robust system.



Security:

We also plan to incorporate security features using simple XOR based encryption as the baseline. Stronger algorithms may be used where security is critical. In encryption the user may wish to encrypt the files or data before sending it out to the server. This is done as follows as soon as the buffer is given by the user in the write call it buffer is first XOR'ed and then written off to the server. The modular structure of our framework allows us to easily extend and provide more functionality as needed. Hence a client who desires a higher level of security can utilize better cryptographic techniques to enforce security policy.

A Note on Postmark's operation:

In un-buffered mode the postmark opens numerous files, appends random text in each file and closes all of them. It later opens and reads those files back again closing them. It later deletes all files and directories created at the start.

We tried to work with caching in such a way that it enhances an access patterns similar to what is shown by postmark. So a read ahead cache is one simple way of getting a marked increase in performance of reads.

Testing:

The DFS server was tested using the postmark file system benchmark from NetApp. There are some important results we need to mention. We varied the size of the payload carried in each DATA structure that is sent between the client and sever and ran repeated tests with postmark. Here are some of the observations from those tests.

Common Statistics for all the runs:

The following are the results of running postmark using the default configuration. The options that had to be configured were:

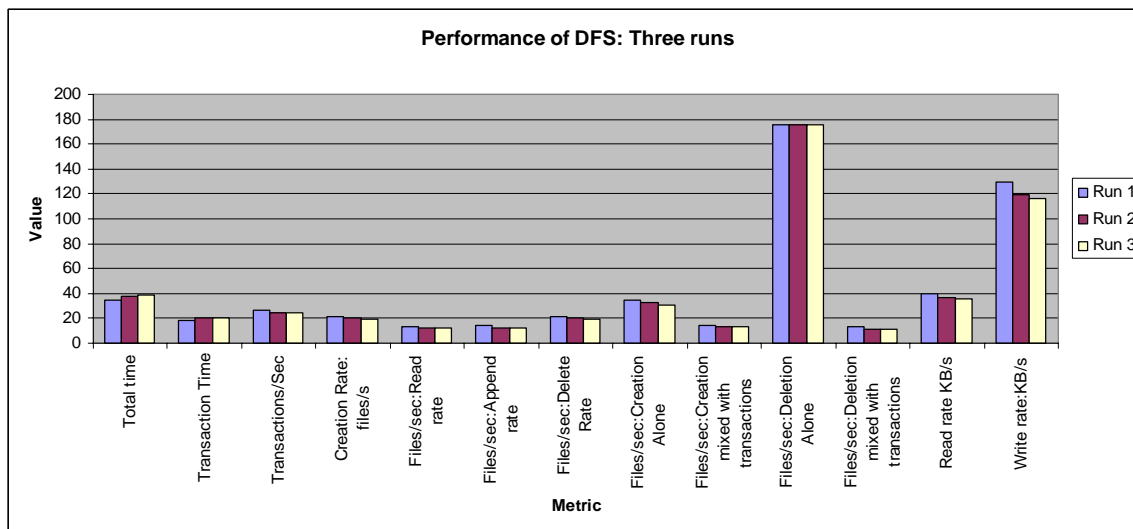
- Turn off buffering
- Set the location to '/r'

Results:

These are the attributes common to the runs that we had conducted. We are presenting the results of three such runs which are representative of the experiments that we conducted.

Total Data Read	1.36 MB
Total Data Written	4.45 MB
Total number of files created	764
Number of files read	243
No of files appended	257
No of files deleted	764
Files: Creation Alone	500
Files: Creation mixed with transactions	264
Files: Deletion Alone	528
Files: Deletion mixed with transactions	236

The following bar chart signifies that our Distributed file system had a consistent performance throughout all the phases. The overall read rate is consistent around **37 KB/s** with a peak performance of **39.92 KB/s** over a number of runs. Similarly the write rate ranged from **120 KB/s** to a peak performance of **130.08 KB/s**. There is no significant variation in the values of any of the metrics.



The results of our experiments are tabulated as follows:

<u>Attribute</u>	<u>Run 1</u>	<u>Run 2</u>	<u>Run 3</u>
Total time (s)	35	38	39
Transaction Time (s)	18	20	20
Transactions/Sec	27	25	25
Creation Rate: files/s	21	20	19
Files/sec:Read rate	13	12	12
Files/sec:Append rate	14	12	12
Files/sec>Delete Rate	21	20	19
Files/sec:Creation Alone	35	33	31
Files/sec:Creation mixed with transactions	14	13	13
Files/sec:Deletion Alone	176	176	176
Files/sec:Deletion mixed with transactions	13	11	11
Read rate KB/s	39.92	36.77	35.83
Write rate:KB/s	130.08	119.81	116.74

Evaluation:

In our design, we had the provision to configure the Payload size used during the transfer of data between client and server. An interesting observation that we had discovered during our experiments with varying the payload size is that the payload size of 512 bytes gives the best performance for running the Postmark benchmark. Since the block size used in the Postmark benchmark is 512 bytes, the optimal performance could be expected when the payload size is equal to the block size being used.

Our experiments also confirmed our premise and hence we configured our DFS to use 512 bytes as the payload size. This is chosen by careful observation and after due experimentation and not merely by accident.

The following are the results that were obtained using a payload size of 256 bytes.

<u>Attribute</u>	<u>Run with Max Payload=256 Bytes</u>	
Total time		65
Transaction Time		6
Transactions/Sec		1
Creation Rate: files/s	53 created (0 per second)	
Files/sec: Read rate	4 read (0 per second)	
Files/sec: Append rate	6 appended (1 per second)	
Files/sec: Delete Rate	53 deleted (0 per second)	
Files/sec: Creation Alone	50 files (0 per second)	
Files/sec: Creation mixed with transactions	3 files (0 per second)	
Files/sec: Deletion Alone	46 files (46 per second)	
Files/sec: Deletion mixed with transactions	7 files (1 per second)	
Read rate KB/s	17.63 kilobytes read (0.27 kilobytes per second)	
Write rate: KB/s	317.88 kilobytes written (4.89 kilobytes per second)	

As it can be observed from the above table, even completing the Postmark on a smaller number of files takes more time than that observed using a payload size of 512 KB. Also since Postmark uses 512 bytes as its block size, there is no additional leverage that could be obtained using a higher payload size than 512 bytes. Moreover the higher payload size may result in wastage of network bandwidth too. Hence we concluded that 512 bytes is the optimal size for our design.

Extendibility:

As already stated, due to the modular design and well defined and consistent interfaces, we are able to seamlessly integrate new features into our existing design. Even the data structures we had can be used to provide more functionality including support for additional system calls.

We believe that this would empower us to explore new approaches and integrate new features which would prove even more valuable in the coming days.

References:

[1] Internetworking With TCP/IP Volume III: Client-Server Programming and Applications, Linux/POSIX Socket Version (with D. Stevens), 2000. 0-13-032071-4

[2] Advance Programming in the UNIX Environment: W. Richard Stevens.

[3] Professional Linux Programming: Neil Matthew et. al.

[4] Papers on the Google File System, NFS and AFS and User level Distributed File Systems.