Fully Distributed Replicated File System

Files submitted:

Makefile
File to make all the goodies. Type ‘make’ to make all the executables

server.h
The header file for server.h

server.c
This is the server program. The server is multithreaded for good performance. All files opened by the client are closed only when a client makes an explicit request to close it.

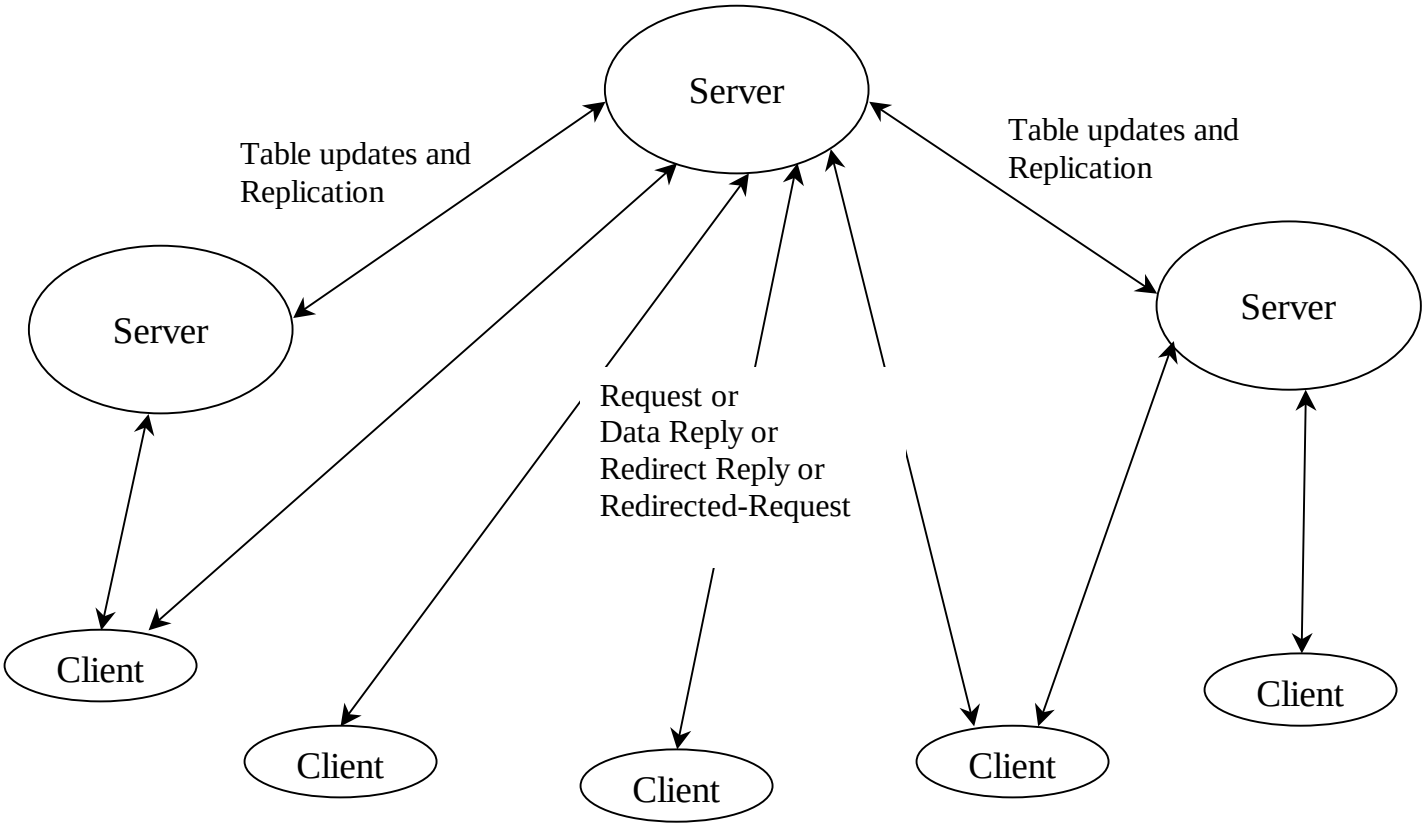
client.c
This is the sample client program that connects to the server and opens a file in various modes and then operates on the file, like read(), write(), lseek() and close().

perf_test.c
This is the testing file. See Last section for more details.

rfs.h
The header file needed by all the above programs.

librfs.c
The library file that is needed by the client. This file has functions to transmit the request, which frees the user from having to know about the internal protocol.

Design



Communication between servers:

- ?? A main server is started
- ?? Any new servers started are given names of all running servers
- ?? New servers contact the running servers and registers itself
- ?? When the global table changes in any server, it propagates it to all servers
- ?? When a file is too busy, it is replicated to another server
- ?? When a file opened in write mode is closed, it is updated to all relevant servers

Communication between client and servers:

- ?? Client knows only one server to contact
- ?? When client makes a request, it can either get the data or a redirect message
- ?? The client follows the redirect message and gets data from that server
- ?? Servers never redirect a redirected client

Server

Servers are multi-threaded, using the pthread API. This approach ensures that servers offer good response to requests, at the same time consuming the minimum system resources. The code is also much simpler than using select() or poll() or multiplex IO. All servers need to know the names of all other servers. Servers are added dynamically, for this, a simple mechanism is used. When a new server is started, the user gives names of all other running servers. The maximum server is defined in MAX_SERVERS in the server.h file. Servers always communicate directly to the client, if the requested file is not found or there is a better server to get the file from, then the server redirects the client to it.

Features:

- ?? Dynamic addition of servers
- ?? Fully multi-threaded
- ?? Load distribution through client redirection
- ?? Dynamic replication of busy files depending on the MAX_THRESH_REPL.
- ?? Write coherence of files are guaranteed. Whenever a client closes a file that was opened in write mode, the file is replicated to all servers that originally had that file. File locking can be used in conjunction to prohibit more than one client from accessing the file at a time.

Client

The client is assumed to be a single threaded process. All accesses to the server is required to use the API provided. APIs are provided to connect to a server, open, read, write, seek and close files.

The library

All client functions are in the form of a library that both the server and client must statically link to. This avoids replication of code, since the server acts like a client when it communicates with other servers.

General packet format

This is the general structure of any header packet that needs to be transmitted before passing any data associated with the corresponding action.

int Operation_Code	int parameter1	int parameter2	int parameter3

OPEN file packet

This packet is sent to the server when the client wishes to open a file for reading.

OP_OPEN	<Open mode>	Length of path name	null

The client then sends the path name as a separate packet.

Path name of file

Acknowledgement for the OPEN file packet

On receipt of the Open file packet, the server opens a file and sends back the local file descriptor back to the client.

OP_OPEN or ERR_OPEN	<FD> or <error code>	null	null

READ file packet

This packet is sent when the client wished to read a file it has already opened on the server.

OP_READ	<FD>	<Size of local buffer>	null

Acknowledgement for the READ file packet

The server sends this packet describing how much data is being sent.

OP_READ or ERR_READ	<Size of data> or <error code>	null	null

A stream of data will follow this, whose length is specified in the previous packet.

Data

WRITE file packet

This packet is sent when the client wants to write to an opened file.

OP_WRITE	<FD>	<Size of data>	null

This will be followed by a stream of data whose size is as mentioned.

Data

Acknowledgement for the WRITE packet

This packet acknowledges the data write and gives the actual number of bytes written to disk.

OP_WRITE or ERR _WRITE	<Size written> or <error code>	null	null

SEEK file packet

This packet is sent when an open file needs to be traversed.

OP_SEEK	<FD>	<offset>	<whence>

Acknowledgement for the SEEK packet

This packet acknowledges the seek packet sent by the client.

OP_SEEK or ERR_SEEK	<Position> or <error code>	null	null

CLOSE file packet

When the client is finished with the file, it must close it. This packet signals the same to the server.

OP_CLOSE	<FD>	null	null

Acknowledgement for the CLOSE packet

This packet acknowledges the received packet, server returns what close() returns.

OP_CLOSE or ERR_CLOSE	<FD>	null	null

SCANDIR packet

This packet is sent when the client wants to read the remote directory's contents.

OP_SCANDIR	null	null	null

Acknowledgement for the SCANDIR packet

The server returns the number of files in the root directory, followed by the names of all files.

OP_SCANDIR or ERR_SCANDIR	<Total files>	null	null

This is the packet the server returns. The client must read this into a char[MAX_PATH][MAX_FILES] array.

List of files

GET_FT packet

The server asks for the remote server's file table

OP_GET_FT or ERR_GET_FT	null	null	null

Acknowledgement for the GET_FT packet

The remote server acknowledges the request and sends the size of the structure.

OP_GET_FT	total_files	null	null

The second packet is the data in the structure.

List of files

SET_FT packet

The server request the remote server to update its tables, server sends the size of the next packet.

OP_SET_FT or ERR_SET_FT	total_files	null	null

The second packet is the data in the structure.

List of files

Acknowledgement for the SET_FT packet

This packet is sent acknowledging the successful update of the table.

OP_SET_FT	null	null	null

GET_ST packet

The server asks for the remote server's server table

OP_GET_ST or ERR_GET_ST	null	null	null

Acknowledgement for the GET_ST packet

OP_GET_FT	total_servers	null	null

The second packet is the data in the structure.

List of servers

SET_ST packet

The servers requests the remote server to update its local tables.

OP_SET_ST or ERR_SET_ST	total_servers	null	null

The second packet is the data in the structure.

List of servers

Acknowledgement for the SET_ST packet

This packet is sent on successfully updating the tables.

OP_SET_ST	null	null	null

REDIRECT packet

This packet is sent by the server when it determines that the client can be serviced better by some other server. The first packet contains the length of the next packet.

OP_REDIRECT	Length	null	null

The next packet has the name of the server to contact.

Server name

END packet

When the client is done with all the processing, this must be sent. The server cleans up and then closes the socket, there is no acknowledgement packet sent.

OP_END	null	null	null

Testing procedure:

Various clients are started and all of them make simultaneous requests to the server. The server either services the client or redirects it.
Replication is fully implemented, to test it run two servers (one of the servers does not have the file) and then start the client program. When client request to read a file for the 2^{nd} time (MAX_REPL_THRESH) the file will be replicated to the second server.
Write coherence of files are also guaranteed. Whenever a client closes a file that was opened in write mode, the file is replicated to all servers that originally had that file. File locking can be used in conjunction to prohibit more than one client from accessing the file at a time.

Performance results:

I started 1 server and 3 clients. Then a total of 20 read and write were performed by all the clients on a file of size 20Kb. In the second round of tests, 3 servers were spawned and the above test was repeated.
The results are compared in this table:

Operation	**Time in sec**			
	C1	C2	C3	Avg
Read with 1 server	6	6	7	6.33
Read with 3 servers and replication	3	4	3	3.33
Write 1 server	8	7	7	7.33
Write with 3 servers and replication	4	3	4	3.67

There is not much difference when a server is added, this can be attributed to the fact that the file sizes are very small and the overhead for a file open is high compared to the time it takes to transmit the file. In other words, whenever a file is opened, the global structures are being replicated to all servers, if this overhead is more than transferring the file, then our approach is inefficient.

Conclusion

It can be clearly seen this type of approach is useful only when the size of the file being transferred is large. The server scales well, since each server directly talks to the client transferring data stored directly.