Northeastern University

CS6650 Scalable Distributed Systems

Final Project

Distributed File System

Manan Patel

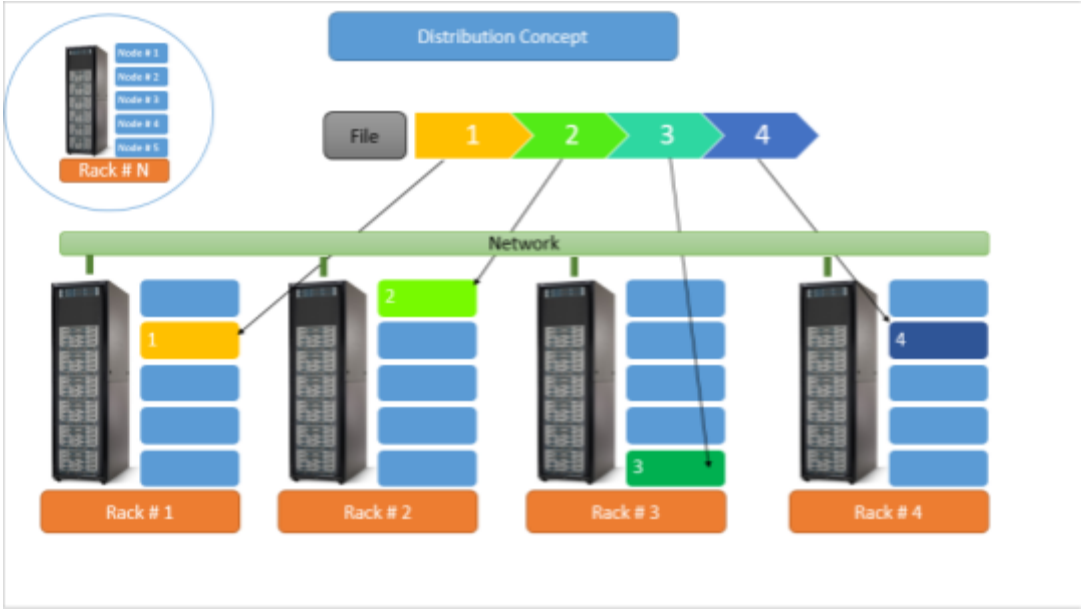
Arvindkumar Thiagarajan

Bishwarup Neogy

Table of Contents

1. Abstract

The first storage mechanism used by computers to store data was punch cards. Each group of related punch cards (Punch cards related to the same program) used to be stored into a file; and files were stored in file cabinets. The computer systems evolved; but the concept remains the same. Instead of storing information on punch cards; we can now store information / data in a digital format on digital storage devices such as hard disk, flash drive, etc. Related data are still categorized as files; related groups of files are stored in folders. Each file has a name, extension and icon. The file name gives an indication about the content it has while file extension indicates the type of information stored in that file. For example; EXE extension refers to executable files, TXT refers to text files. etc.



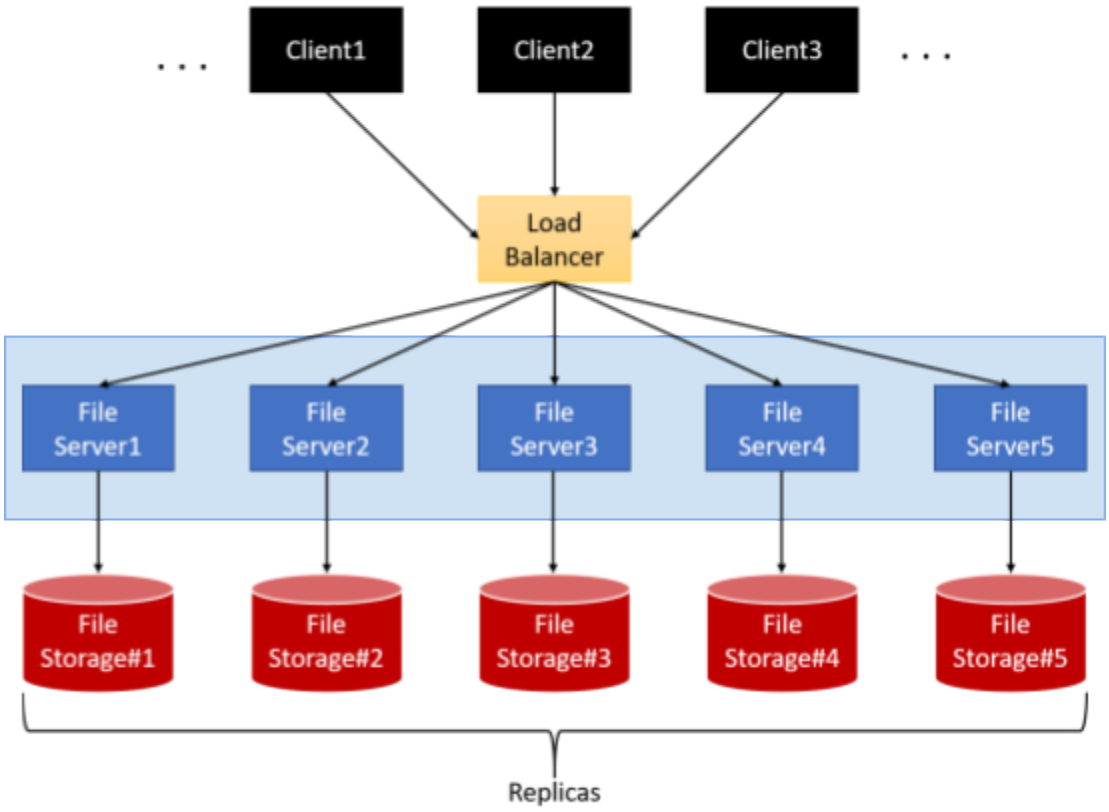
A distributed system, also known as distributed computing, is a system with multiple components located on different machines that communicate and coordinate actions in order to appear as a single coherent system to the end-user. The machines that are a part of a distributed system may be computers, physical servers, virtual machines, containers, or any other node that can connect to the network, have local memory, and communicate by passing messages. A *distributed file system* is a client/server-based application that allows clients to access and process data stored on the server as if it were on their own computer.

As part of this project we have implemented a Distributed File system based on a client server architecture where we allow multiple clients to access servers. The system provides the client with operations such as file upload, file download, deleting a file as well renaming an existing file on the server. The object was to build a robust, fault tolerant, highly available distributed file system.

2. Design Overview

The Distributed File System consists of multiple file servers which serves the storage needs of various clients/users. The load balancer handles the client requests and ensures that a single server is not overloaded with requests in a round robin fashion. In order to make the system tolerant, each server maintains its own copy of the files, in other words the data is replicated across all servers. Thus, if any server goes down or its storage gets corrupted, the data can be recovered from other servers and the system would continue to work.

2.1 System Architecture



2.2 Operations

The system supports the following features:

- Upload: When a user fires a request to download a file, the load balancer would connect the user to an available server and the server will then fetch the file from the corresponding replica where it was saved.

- Download: When a client requests to upload a file, the load balancer connects the client to the next available server, and the file is then written to all the replicas. Therefore, this system follows the read-any-write-all protocol.
- List files: The client can also request the server for a list of files that are currently stored in the server. This operation is very similar to download, instead of returning a file, the server returns a list of file names.
- Rename: The client sends the server the file to be renamed, the new name of the file as well as a duration*. The server then looks for the file in its replica and renames it as per the client's request. However, the client locks the file for the given duration. In this period, no other client can make changes to the file.
- Delete: A client can also request the server to delete a file from its replica. The server takes the filename to be deleted as an input from the client. It then locates the file in its storage and deletes it.

* The duration has been added to demonstrate the mutual exclusion property. Please refer to the implementation details for more info.

It is important to note that this also promotes replication transparency. The client is never aware of the replication happening at the server side and is also unaware of the distributed nature of the server side. Additionally, the system ensures that the data is consistent across all replicas at all times. How this was achieved has been discussed in the following sections of the report.

3. Implementation

3.1 Replicated Data Management

Copying the same data over multiple nodes in distributed systems is critical to keep the database up and keep on serving queries even during faults. Following are other reason behind data replication in a distributed system :

- Higher Availability: To ensure the availability of the distributed system (System keeps on working even if one or fewer nodes fail)
- Reduced Latency: Replication assists in reducing the latency of data queries (By keeping data geographically closer to a user. For example, CDN(Content Delivery Networks) keeps a copy of replicated data closer to the user. Ever thought how Netflix streams videos with such short latencies!)
- Read Scalability: Read queries can be served from replicated copies of the same data (this increase overall throughput of queries)
- Network Interruption: System works even under network faults

In our distributed file system, we maintain upto 5 copies/replicas on 5 different servers of files being uploaded to the system. This allows clients to fetch data from any of the servers, thus, improving read scalability. Even if a few servers go down, the other live servers can still process the request without the client knowing about any failure.

3.2 Distributed Transaction

A distributed transaction is a type of transaction with two or more engaged network hosts. Generally, hosts provide resources, and a transaction manager is responsible for developing and handling the transaction. Like any other transaction, a distributed transaction should include all four ACID properties (atomicity, consistency, isolation, durability)

In our application, when a user makes a request to upload, download, delete or rename a file, the request goes to the server to which the client is connected. If this transaction was not distributed, it would be sufficient for the server to directly go ahead with the request and make changes to the database it is connected to. However, since the data has been replicated in multiple databases handled by multiple servers, the server that received the request needs to communicate this change to the other replicas to guarantee data consistency. This is what makes the transaction distributed in nature and the entire task of communicating with other servers is carried out using the Paxos algorithm as described in the section below.

3.3 Paxos

Paxos is a family of protocols for solving consensus in a network of unreliable processors (that is, processors that may fail). Consensus is the process of agreeing on one result among a group of participants. This problem becomes difficult when the participants or their communication medium may experience failures. Consensus protocols are the basis for the state machine replication approach to distributed computing, as suggested by Leslie Lamport and surveyed by Fred Schneider. State machine replication is a technique for converting an algorithm into a fault-tolerant, distributed implementation. Ad-hoc techniques may leave important cases of failures unresolved. The principled approach proposed by Lamport et al. ensures all cases are handled safely.

In our application, the client can either upload/download/delete or rename a file. When a client chooses from either upload/delete or rename, paxos is invoked. For example when a user selects to upload a file, the system(proposer) sends a *prepare* message to all the servers alive(acceptors). The server(acceptor) that receives this message, checks its current paxos id against the id that it received in *prepare*. If the server's(acceptor) id is greater than the id in *prepare* then the server(acceptor) does not send a promise otherwise it sends a promise to the proposer. Now the proposer checks how many promises it received for the sent our *prepare* requests. If the proposer gets a consensus which means more than 50% of the acceptors promised than it goes ahead and

sends the data to all the promised acceptors by calling *acceptRequest.* In case it does not reach a consensus, the proposer increases its *proposal id* and makes a request to other acceptors.

3.4 Fault Tolerance

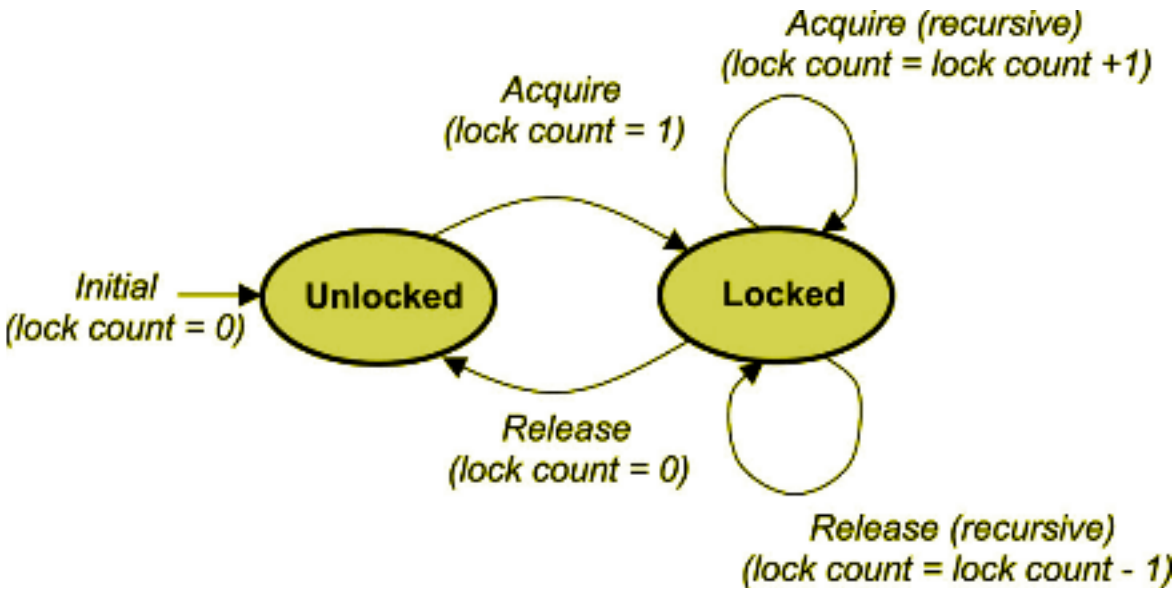
Unlike a single system, distributed systems have partial failures. Overall failure of a single system tends to make the whole system down. On the other hand, in a partial failure, the system can continue to operate while recovering from a partial failure without seriously affecting the overall performance. The ability to endure service even if failure occurs In addition, a system with fault tolerance is sometimes called a high dependability system.

In the current system, even if a few servers are down, the system will detect that and not use them for consensus. This way consensus becomes independent of servers which don't contribute. Now when the server comes up and becomes an acceptor or proposer. It will realize that it is not up to date with other servers. Hence, it will make requests to other servers and retrieve the most updated data. If a server did not agree to the consensus because of some internal error, the request will still go through if the majority of the acceptors reply with a promise. For example out of 5 acceptors only if 3 return back a promise then we can continue and commit the change we want to. This makes the system fault tolerant and does not rely on 100% functioning of servers.

3.5 Mutual Exclusion

Mutual exclusion is a property by which concurrent transactions are handled in a fault effective manner. This is done by preventing access to a shared resource by a client/request while that resource is being accessed or operated on by another client/request. This can be achieved with the help of locks in Java.

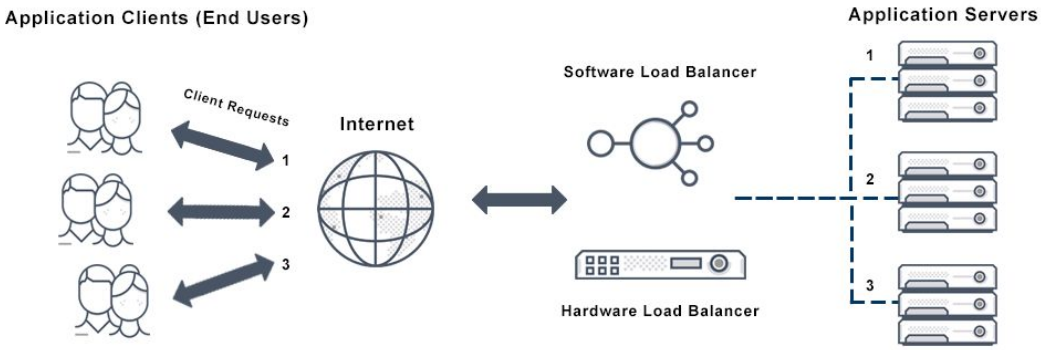
The concept of locks is that when a client requests to access a shared resource which is not being accessed by any other client at that point in time, then a lock is placed on that file. This lock would prevent other clients from accessing said resource. Once the client's request has been processed, the lock is released from that resource, allowing another client to access it. This process is illustrated in the following diagram.



In our implementation, we have enforced mutual exclusion with the help of the above-mentioned locks. In a scenario where a client requests to rename a file on the server, a FileChannel object is initialized on this file. Then, the trylock() method is invoked on it in order to attempt acquiring an exclusive lock on the file. If the method returns null or if an OverlappingFileLockException is thrown, then that means a lock has already been placed on this file prior to this request and the file cannot be accessed for renaming. Otherwise, the file is available for access and a new lock is placed on it for the duration specified by the client. For this duration, no other similar request will be processed by the server. Once the rename operation goes through and the duration specified expires, the lock will be released, opening the file to future requests.

3.6 Load Balancer

Load balancing is the process of distributing network traffic across multiple servers. This ensures no single server bears too much demand. By spreading the work evenly, load balancing improves application responsiveness. It also increases availability of applications and websites for users.



In our project, we have used a Round Robin Load balancer. Round robin load balancing is a simple way to distribute client requests across a group of servers. A client request is forwarded to each server in turn. The algorithm instructs the load balancer to go back to the top of the list and repeats again. This method rotates servers by directing traffic to the first available server and then moves that server to the bottom of the queue. Most useful when servers are of equal specification and there are not many persistent connections.

In a nutshell, round robin network load balancing rotates connection requests among web servers in the order that requests are received. For a simplified example, assume that an enterprise has a cluster of three servers: Server 1, Server 2, and Server 3.

- The first request is sent to Server 1.
- The second request is sent to Server 2.
- The third request is sent to Server 3.

The load balancer continues passing requests to servers based on this order. This ensures that the server load is distributed evenly to handle high traffic.

5. Future Scope

Currently the scope of the project was to provide all basic functionalities of a standard distributed file system. However, there are some areas that can be improved with different and more sophisticated approaches and the project can also be extended to include many interesting and extremely useful features. We have listed some of the potential features and rooms for improvement.

Security: We can improve the security of the application by using a service to register users and provide them a password protected session to interact with the file system. We can also improve security by storing only the encrypted version of the files in the system, which can then be decrypted using the private key of the user who wants to access it.

Notifications: In future releases of this application, we can introduce a notification service to notify users whenever a file they uploaded is accessed/modified by another client.

Private files: We can also incorporate privacy by allowing users to store private files that cannot be accessed by any other client. This feature can be further extended to allow sharing of files with specific individuals, similar to Dropbox or Google Drive.

Improved Load balancing: The biggest drawback of using the round robin algorithm in load balancing is that the algorithm assumes that servers are similar enough to handle equivalent loads. If certain servers have more CPU, RAM, or other specifications, the algorithm has no way

to distribute more requests to these servers. As a result, servers with less capacity may overload and fail more quickly while capacity on other servers lie idle. The weighted round robin load balancing algorithm allows site administrators to assign weights to each server based on criteria like traffic-handling capacity. Servers with higher weights receive a higher proportion of client requests.

6. Conclusion

We built a fault tolerant distributed file system that primarily facilitates multiple clients to interact with multiple servers to perform various operations on files stored on the servers. A user would be able to upload files to be stored securely on servers. These files can be downloaded by users locally, renamed, or deleted from the servers at any desired time.

The files would be replicated across numerous servers to ensure that files are secure and be accessed when a server is down or terminated. This is done with the help of a client request to replicate which is handled by a round-robin load balancer, whose job is to distribute the traffic among all servers to prevent server overloading. This enforces replicated data management and fault tolerance in our system. In order to maintain consistency of these replicated files across the servers, the system employs the Paxos algorithm. This algorithm helps achieve consensus among all the servers if any transaction can go through or not.

The distributed file system also enforces mutual exclusion to manage concurrent transactions. When a user attempts to rename or delete a file at the same time a server is processing another rename transaction, the user is denied access to the file at that point of time. Only when the server completes the transaction and releases the lock placed temporarily on the file will the user be able to access that file.