Project Report

Multi-Cloud Distributed File System

Nevedha Ravi || Shreyas Badiger || Siddharth Narasimhan
Team 10

CS 237 - Distributed Systems Middleware
Spring 2019, University of California, Irvine

I . Key Objective :

In a day and age of large scale distributed systems, where most applications on the web, or on a mobile platform, are being used simultaneously by thousands of people all across the globe, it is of paramount importance that the data stored in the servers is consistent, and the services are highly available. This means that if a person using an application in the USA changes a piece of data at a particular instance, then this change must immediately (or with minimum latency) be reflected to all the other users who are simultaneously accessing this piece of data, be it from any corner of the world. This is exactly where the concepts of large scale distributed systems play a vital role.

A distributed file system is a classic example of a large scale distributed system, where the files are being accessed by multiple users all across the world, simultaneously. The file system would support all the features that a user could possibly do with a file on their local computer, which includes creating a file, reading a file and writing a file, and deleting a file. The design of a good distributed file system presents certain degrees of transparency to the user and the system.

1. ***Access Transparency*** - Clients are unaware that files are distributed and can access them in the same way as local files are accessed.
2. ***Failure Transparency*** - The client and client programs will operate correctly after a server failure.
3. ***Tolerance for Network Partitioning*** - The entire network or certain segments of it may be unavailable to a client during certain periods. The file system will be tolerant of this.
4. ***Replication Transparency*** - To support scalability, we may replicate files across multiple servers. Clients will be unaware of this.

II . Proposed Solution :

In this project, we have designed and developed a distributed file system in which the nodes and files are distributed across multiple public clouds, which means they could be geographically separated. Users working in remote places would be able to create, read, write and delete files as they wish, concurrently. Throughout the workflow, where multiple users are accessing the same file at the same time, data consistency is always maintained on a global perspective. We have also implemented a mutual exclusion infrastructure using a central coordinator design, where the Mutex server will handle and process all the client requests for a particular file. At any instance, only one client will be allowed to write into a particular file, across all the public clouds. This state will be known to all the clients across all the public clouds, and any new request for writing into a file which is currently being written into, will be rejected by the Mutex server. As in the case of any central coordinator design, it becomes a bottleneck for the entire system. In order to overcome this obstacle, we have incorporated the concept of fault tolerance, where in case the primary Mutex server encounters a failure, the backup Mutex server immediately assumes the role of the primary Mutex server, and the system will be notified of such a failure. We have designed the architecture of the system in such a way that each public cloud is considered as a cluster of nodes. Each cluster has a Master node which is the point of contact for all the

nodes belonging in the cluster. All the requests coming from nodes in the cluster will be handled by their respective Master node. However, the Master node is just a mediator. It does not act as the storage server. The Master node communicates with the storage servers, for which we have incorporated HDFS (Hadoop Distributed File System). The HDFS also ensures that the files being written are replicated with a replication factor which can be set by the administrator. The Master node is also responsible for broadcasting the changes to a file to all the other Master nodes. This ensures that the edited file is updated in the HDFS across all the public clouds. The system design and architecture has been explained diagrammatically in the following figure.

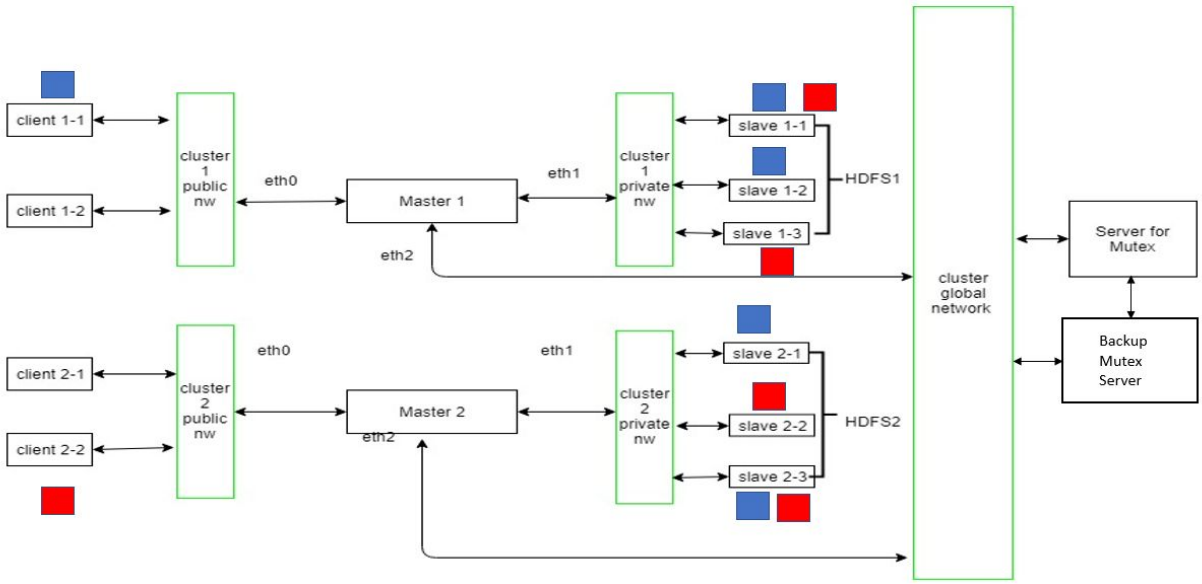


Fig 1.1 : Design of the system showing two clusters, and their connection with their respective Master nodes, and HDFS.

III . Related Efforts:

- **Data storage in a distributed file system**

A large-scale distributed system could have several data sources which produce data that has to be shared across to other nodes in the system. [1] identifies that the conventional methods for efficient sharing of data files in a large-scale distributed system fail when the files change often (everyday, more than once a day etc.). A file server model in which files are stored in some dedicated server which caters to requests for the same from multiple clients has this server and the interconnection network of the system as the bottleneck. In a client-based caching scheme where the clients request for a file from the server and caches it for future use, and reuses it later after validating it with the server. This overhead traffic between the client and the server to check if the file has been updated every time it is opened in the client burdens the network, affects scalability. In a call-back scheme, the server is made responsible to notify the clients who have an outdated version of the file thus reducing the communication

overhead between the client and server significantly. However, all these conventional schemes fail in large-scale deployments with files changing often, being geographically distributed and large.
A cluster-based file replication scheme called **Frolic** is proposed in [1]. A cluster is a group of workstations and one or more file servers on a LAN. Initially a file is only held by the owner server. When another node in some other cluster needs this file, it is dynamically replicated and made available in the second cluster's file server. This effectively reduces the burden on the owner server as it no longer needs to serve requests from the second cluster which has a copy in its file server. By doing this, the distance between the server and the client is also reduced which reduces the burden on the interconnection network and reduces the latency to access the file.
In the proposed system, a file could be modified only by the cluster which owns the file. All the nodes in the cluster could modify it using some technique to maintain consistency of the file. But others must acquire ownership of the file before they could modify. The owner keeps track of the information about where the replicas are. Synching the replicas after every write operation is costly, so synchronization is done only when the file is closed after modification. Frolic approach significantly reduces file access delays and reduces the burden on the server and backbone network in a large-scale distributed system. The disk space requirements are low, since there is only a single replica per cluster. No particular model of data management within the cluster is imposed, hence Frolic could be implemented on top of an existing distributed system.

It is important to make the best design decisions when it comes to data placement in a distributed system. The quality of service is majorly based on these design decisions which is critical when the data is geographically distributed or when the amount of data the system is expected to manage is large. Even though the prototype we are building is not going to be tested with trillions of bytes data, we have been keen to design a system that could scale well with respect to size of data. For us to make the best design decisions we have studied the following systems.

- **Cassandra**

Cassandra is a distributed database that can spawn across multiple cheap commodity hardware and yet promises to provide a high read and write throughput. One of the key services offered by Cassandra is the high availability with no single point of failure. (more investigation and details about this in the following sections) Unlike traditional databases, Cassandra is a NoSQL database. It instead provides its own data model that is simple and easy to use for any type of data.
Cassandra is essentially a database which is distributed across several different machines. The data stored in Cassandra is in the form of a multimap. While a hash map doesn't support multiple copies of a <key,value> pair, a multi-dimensional map does. Each table of Cassandra is an entry of the multimap. These key value pairs are stored in hierarchy of columns. The authors don't dive into the intricate details of the data model.
While there are several desirables for a good database, the paper rightfully discusses only the key features addressed by Cassandra.
1) Scaling: Cassandra partitions the data items (data load) dynamically whilst supporting read/write. Cassandra uses consistent hashing to assign the data items to nodes. Nodes are arranged in a logical

fashion. The partitioning algorithm is load-aware hence allocates the data-items judiciously and ensures no node is overloaded. Cassandra also uses Zookeeper for the node election.
The bootstrapping algorithm governs the way the nodes join/leave the cluster. A new node joins at the random location in the logical ring and is eventually placed in its rightful position. The bootstrapping algorithm learns the nodes positions by gossiping information shared periodically. Cassandra asserts a node failure must not lead to rebalancing the node as the failure might be temporary. Hence the node addition/removal is marked as the responsibility of an administrator.
2) Reliability: Cassandra follows the replication mechanism similar to Amazon's dynamo. The replication factor is programmable. The system also supports features that chooses/not chooses the rack and data center awareness. This helps to achieve the data locality.
3) Fault detection and Fault tolerance: Cassandra nodes learn that a node is down by using a modified version of Accural Failure Detector method. Instead of emitting a Boolean value to identify the status of a node, Cassandra's AFD gives a numerical value that is associated with node's reliability in the cluster.
Since Cassandra uses consistent hashing mechanism to distribute the load amongst nodes. A node's removal or addition is bound to affect none beyond it's neighbouring nodes. **T**his is how Cassandra is able to achieve a high fault tolerance.
4) Efficiency: In a database, a larger part of CPU cycles is devoted towards transferring the data from disk to memory(vice-versa). Cassandra ensures to do sequential writes instead of burst writes to minimize the IO overhead. This is a key reason behind Cassandra's high write/read throughput**.** In addition to that, Cassandra uses Bloom filters, to fetch data from the respective node. Bloom filters uses hashing mechanism to store data and is capable to handle an extremely large set of data. The data stored in Cassandra tables undergoes frequent merge like Google's Bigtable.

- **OceanStore**

OceanStore is a utility to store, access persistent piece of information stored across several untrusted servers. OceanStore is an Internet-scale, persistent data store designed for incremental scalability, secure sharing, and long-term durability. OceanStore follows several distributed techniques like data protection through redundancy, data security through cryptographic techniques, data redundancy through cache-anywhere technique, periodic monitoring to avoid security attacks like DoS. When the paper was written the OceanStore was still in its inception phase (working on the prototype).
OceanStore is envisioned as a cooperative utility model for persistent storage. The users will be charged with a nominal fee for the services provided by OceanStore. The set of servers to host the OceanStore could belong to anyone from a café to a restaurant etc. The hosts are responsible to the data integrity and services offered.
OceanStore addresses to popular issues that could be faced in the above mentioned situations

1. **Untrusted infrastructure** -> All the data stored in the systems are **c**ryptographically hashed and secure. The dynamic addition and deletion of the participating nodes handles the scalability aspect gracefully.
2. **Nomadic Data** -> The data stored at one place could be queried from another server which is geographically far apart. In such cases data locality aspect becomes key. OceanStore addresses this issue by using a mechanism called as promiscuous caching**.** It refers to cache anywhere and

anytime as per the need. This feature aims to untie the data from its actual origin and allows the data to freely flow across the system.

Objects are stored and replicated on several machines in the OceanStore cluster. A term called is floating replicas is introduced here. Replicas in the OceanStore aren't bound to one system in the OceanStore. They also follow the free-flowing-of-data model and are never tied to any server at any given point of time.
These replicas are detected through a two approach model. First the system attempts to locate the data using a probabilistic algorithm to locate the server nearest to the requesting client, if this fails, then the slow deterministic algorithm takes over to locate the data.
In-order to handle updates, OceanStore adopts a strategy to append a new version of the data and invalidates the old version (As the in-place updates are usually expensive).
Every data entry in the OceanStore qualifies to be an object. These objects exist in two different forms. An active form of an object and an archival form of the object. The active form is the one which is the most recent one. The archival form is the permanent read-only version of the object.
In the paper, the author explains the following components of OceanStore as the system components:

- **Naming** -> how the objects are created, named and stored across the OceanStore cluster.
- **Access control** -> OceanStore provides two kinds of permissions read/write. These permissions are essential to maintain data integrity and data security.
- **Data location and routing** -> The probabilistic algorithm uses attenuated bloom filters to locate the data, whereas the deterministic algorithm uses "Wide-Scale Distributed Data Location" algorithm to locate the data.

- **Google File System**:

The design and architecture of our file system is inspired a lot from the Google File System. Ours is a file system that would be distributed across a cluster of nodes on a public cloud. The cluster will follow a master-slave architecture, inspired by Google File System. The master, which is a file server, will contain the metadata of all the files stored on all different edge nodes in the cluster. Hence this system resembles the master-slave design of the Google File System. There might be multiple clusters spread across multiple cloud environments. When a client node, which is outside the cluster, requests files from this server, it caches them locally, so that future requests can be served without making a request to the master server. These local copies will maintain a version number which is associated with every file. Whenever the master copy is updated, the edge node will detect this change, by comparing the version number associated with the file, and thus request for the latest version of the file. In the Google File System, the files are divided into smaller chunks of 64MB each, and spread across the edge nodes. In our system, we will not be following this initially (may incorporate later), because the sizes of files that our file system intends to serve are much smaller.
Just like the Google File System, we intend to incorporate the concept of distributed Mutex, mainly in order to maintain data integrity.

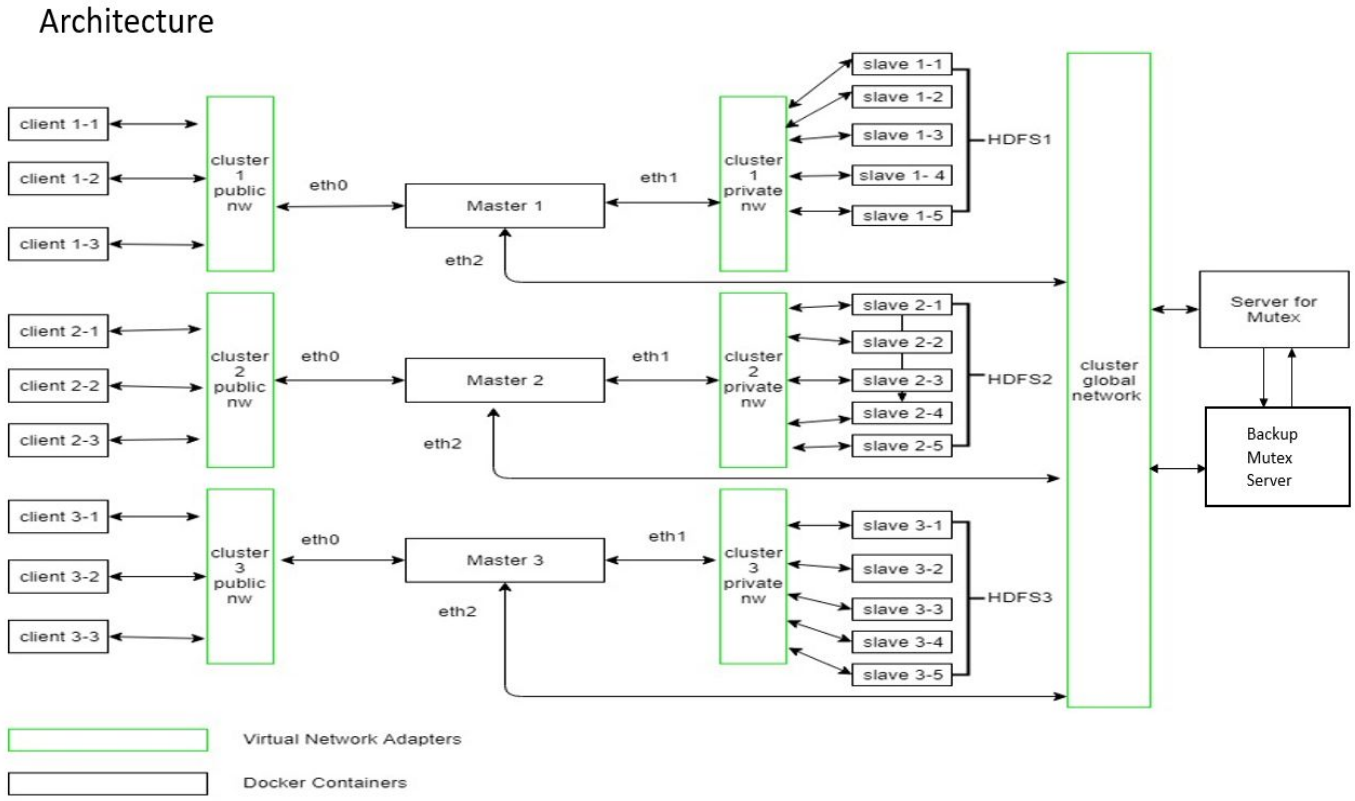
In the Google File System, the master acquires read-write locks on the source, logs the operation to disk, and duplicates the metadata for the source. The created snapshot files points to the same chunks as the source. This approach is highly effective to ensure consistency in the distributed file system.
We intend to achieve parts of this design in our File System too.
The Google File System has four very vital advantages. We intend to achieve at least three out of these four in our project.

1. ***High availability.*** Even if couple of nodes fail, data is still available. This is achieved through data replication. Google has designed this system assuming that component failures are inevitable.
2. ***High throughput.*** Multiple nodes working in parallel, in order to perform the read and write operations, also achieving fault tolerance and consistency.
3. ***Reliable storage.*** Corrupted data can easily be detected and re-replicated. This is achieved mainly with the help of efficient data structures used to build the file system.
4. ***Atomic append operations*** ensures no synchronization is needed at client end.

There are certain disadvantages that the Google File System faces. Some of which may be applicable to our File System as well. However, the numerous advantages that such a design offers outnumbers the disadvantages, and hence makes the project a very valuable one.
One of the bigger disadvantages of such a system, that even the Google File System faces is:
Master can be bottleneck. This is a big drawback, considering that the master is a single point of failure. If the master fails, the system ensures that another node assumes the role of the master. However, depending on the circumstances, this may result in some downtime.
However, the high focus on data replication which helps in achieving high availability, creates an environment to counter this disadvantage.
Another disadvantage that the Google File System has, but is not applicable to our File System is that it is not efficient for smaller sized files. Since the chunk size in GFS is set to 64MB, it does not work very well for files whose size is lesser than that. However, in our project, the files that our File System intends to serve are much smaller in size. Hence our design does not break the files into smaller chunks.

IV . Detailed Design of the System:

A cluster (master node along with HDFS) is a file server which is designed in a master-slave architecture, inspired by Google File System [5]. The master is a multithreaded server which has the metadata of the files stored on other slave nodes of the cluster which run HDFS on them. The system is made up of number of such clusters. A client node, which is outside the cluster can add files to this system, can request files in read-only or read-write mode from this system and caches them locally for reuse.



Overview of Components used in the system:

- **Mutex Server**

A multithreaded server to maintain mutual exclusion with respect to updates on files.

- **Backup Mutex Server**

A backup of the mutex server which takes over when the primary server fails.

- **Multiple Master Servers**

Each cluster has one master node which is responsible for communicating with the clients and the slave nodes running HDFS.

- **Multiple Slave nodes running HDFS**

These are the nodes that hold the files.

- **Client nodes**

Client nodes talk to a master of the cluster to add/update files in the system.

Features of the system:

- **Adding a file to the system**: Any client node attached to any of the clusters can add the file to the system through the master node to which it is attached. This master node takes care in replicating the same file in 3 of the slave HDFS nodes and also sends it across to other clusters. It broadcasts the added file to all the other masters in the system, which takes care to add the files to the attached HDFS slave nodes. A client requesting the file again will get an updated copy of the same.

- **Reading a file from the system**: A file in the system could be requested by a client in read-only or read-write mode. The client requests the file to the master it is attached to, the master fetches it from the slave HDFS nodes to send it across to the client.

- **Updating a file in the system**: Any client can edit any file available in the system. However only one client can edit a file at any point in time. This is ensured by using a central server which holds the metadata about the file and locks the file that are currently being written. This helps in achieving mutual exclusion by denying concurrent file update requests. Once the file is edited by the client, it is sent back to the master it is attached to and the master takes care to broadcast the updated file across to the other masters.

- **Blocking concurrent writes to files:**

  If two clients in the same cluster try to write a file concurrently, the master node takes care to block the client which requested second. To achieve this, it stores the current list of files that are being written by the client nodes talking to it. For achieving mutual exclusion across the clusters, a centralized server is used. The central server is a multithreaded server to achieve mutual exclusion (referred as Mutex Server) blocks any concurrent updates to a file by blocking it while a client is writing. When it is done, the file is unblocked and available for others to write. This primary server has a backup server which takes over when the primary mutex server goes down. The primary server keeps synchronizing its metadata with the backup server once every minute. For quick recovery, the primary server keeps sending a heartbeat message to the backup server every 30 seconds. When the backup server doesn't receive the heartbeat message, it times out and takes charge as the primary server. We thus achieve fault tolerance with respect to Mutex Server.

- **File Replication and Fault Tolerance:**

  Each file is replicated in 3 nodes within the cluster. This way, if a node of the cluster goes down, we have other nodes with same files which could serve requests, thus achieving fault tolerance

in the system. Also, multiple clients requesting the same file could be served concurrently. Every file is also replicated across in other clusters in 3 nodes. This way we keep the distance between a data source and the consumer as close as possible which prevents hogging of the network.
The Mutex server sends heartbeat messages to a backup server every 10 seconds to communicate its liveness. It also synchronizes its data with the backup server every 30 seconds, so that in case the primary server crashes, the backup could detect it fast and take charge.

V . Frameworks Used:

The system we have designed and implemented makes use of Docker containers, wherein each component such as a client, master node and Mutex server are containerized into Docker containers.
The reason we are using Docker containers is to simulate a large scale distributed system where each cluster is deployed in a geographically separated public cloud.
Docker containers were used as the nodes of the system. Without using an existing container orchestration tool like kubernetes, we have developed our own container orchestration script which deploys the container and the virtual network adapters forming the deployment network.
The slave nodes have Hadoop Distributed File System running on them. The HDFS helps to replicate the files depending on a replication factor that the administrator specifies. For our testing purposes, we have used a replication factor of 3, which means there will be 3 replicas of each of the files stored in the HDFS.
Making use of HDFS on the slave nodes, makes the system more efficient because the master nodes do not store any of the files that the clients are pushing.

VI . Evaluation:

In order to evaluate the performance of the system, the following workflows were tested on the system. These workflows ensure that the system is being tested for most corner cases, such as large files, multiple concurrent write requests and more.

Workflow 1:
A file created by one client in cluster1 was accessible to a client in cluster2. When the file was created,

Workflow 2:
One client in each of the clusters create a file. All these files were accessible by all other clients in all other clusters.

Workflow 3:
On a scaled setup with 12 clusters, we tested creation and updation of 1300 files. The system did not go on a deadlock. We were able to successfully create, read and update files.

Workflow 4:
Two clients from two different clusters try to write a file at the same time, only one was allowed to write at any point of time. The other was blocked due to the action taken by the Mutex server.

Workflow 5:
When the Mutex server is running, and encounters a failure, the backup Mutex server assumes the role of the primary Mutex server.
This was tested by monitoring the periodic heartbeat messages sent by the primary Mutex server to the backup Mutex server.

Workflow 6:
Heterogenous files. The system was tested with heterogeneous files (of different extensions). We have designed the system and coded it in such a way that the files are read and written byte wise. Hence different file types are supported.

Workflow 7:
The system was tested with large file sizes. For such a case, we have designed the system in such a way that the maximum page size is set to 1024 bytes. Which means that all the write operations take place in sizes of 1024 bytes or lower. If the file exceeds the page size, then the write operation is broken into parts.

VII . Conclusion and Future Work:

Using the concepts studied in the course, we have created a prototype that could be used as a distributed file system in an enterprise level workload. The file system uses replication and backup servers to achieve fault tolerance. It also uses heartbeat messages to detect liveness of a server. The file system provides capability for any client to access any file, anywhere. We have identified open problems like dynamic load balancing, a distributed server for achieving distributed mutex instead of a centralized server which we consider as opportunities for future work on this.

VIII . References :

[1] S. Sandhu and S. Zhou, Cluster-based file replication in large-scale distributed systems, ACM SIGMETRICS, 1992
[2] C.l Fidge, Timestamps in message-passing systems that preserve the partial ordering, Australian Computer Sci. Comm. 10 (I) (February 1988) 56-66.
[3] Avinash Lakshman and Prashant Malik, Cassandra - A Decentralized Structured Storage System.
[4] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao, OceanStore : An Architecture for Global-Scale Persistent Storage, ACM ASPLOS 2000

[5] Ghemawat, S., Gobioff, H., and Leung, S.-T. 2003. The Google File System. In 19th Symposium on Operating Systems Principles. LakeGeorge, NY. 29-43
[6] Gray, P. Helland, P. Neil and D. Shasha ,The dangers of replication and a solution, ACM SIGMOD, 1996
[7] K. Aguilera, W. Chen, and S. Toueg, Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication, Cornel, 1997
[8] Ricart and A. Agrawala, Ricart and A. Agrawala, An optimal algorithm for mutual exclusion in computer networks, Communications of the ACM, 1981, Communications of the ACM, 1981