# Remote File System Based on Raft Algorithm

## Distributed Algorithms Project Report

*Mengyang Chen, Shuzhi Gong, Jing Wang, Boyang Yue*

## Abstract

The distributed systems have gradually replaced the single-node system since introduced for its efficiency, robustness and computing power. In this project, a distributed server cluster is implemented to avoid single node crash problems and improve the system stability. Since the servers in the cluster handle the same client requests, it's critical to make sure that all servers in the cluster keep the same state machines and the same logs, which can be called as consensus.
To keep the consensus in a server cluster, a consensus algorithm is needed. We chose the Raft algorithm because it's efficient and feasible. Evidence and analysis are also provided in the comparative analysis section. We compared many synchronization and consensus algorithms and concluded that the Raft is the best choice.

The Raft algorithm was implemented on the Remote File System to show its functions and properties. In the Remote File System, three servers run on three virtual private servers (VPS) as a whole cluster to receive and respond to any client requests by HTTP request method. The server cluster uses the Raft algorithm to dispatch the requests from clients and duplicate the logs and state machines. The demo can handle both the normal and special cases and performs well in the demonstration. More implementation detail is contained in Section 5.3.

**Keywords:** Distributed Algorithms, Consensus Algorithm, Raft algorithm, Remote File System, C/S architecture, Flask

# 1. Introduction

One of the main goals in distributed systems is that we want to improve its reliability. There are a lot of potential threats that may influence distributed systems' safety. For example, in a client-server distributed application structure, the number of servers might be more than one. When the servers cooperate with each other, they are easy to have problems agreeing on modifying the same piece of data. This situation generates the agreement problems, which would be needed for the protocols that can continually enable the system as a whole to work. An apparent approach to solving this agreement issue is that we can give the right to processes, which allow them to vote for the divergences and agree to the majority as final [1]. That is the main idea of the consensus algorithms.

Consensus algorithms give the opportunity to the server cluster to cooperate as a group that can assist each other in the case that one or some of its members suffer from the crashed situation. That probably is the main reason why consensus algorithms are becoming more popular and reliable to establish a large-scale distributed system. One of the famous consensus algorithms is called Paxos, which has been considered as the greatest number of implementations of consensus algorithms in the distributed application structure over the last few decades [2, 3]. Even nowadays, Paxos algorithm still is one of the most basic consensus algorithms for students that need to learn if they want to build a distributed structured application. However, Paxos algorithm may not be the best option since the main goal for this project is for educational purposes. Compared to Paxos, the new consensus algorithm that is named as Raft seems could produce a better foundation in a way that it is much easier to understand than Paxos, but the algorithm itself is completely enough to satisfy the needs of solving agreement problems. The detail of reasons why we choose Raft instead of Paxos will be discussed later in section 3 and section 5.

In this project, we firstly explore the different consensus algorithms and compare the differences between them. Next, we demonstrate our design concept of our application and the potential issues that still remain, and how we can solve that in the future work. Then, we illustrate the detailed idea of Raft consensus algorithm and how we implement it in our application. Last but not at least, we discuss the advantages and disadvantages of the Raft consensus algorithm and explain why we implement Raft finally to help our application deal with agreement problems.

# 2. Literature Review

As the computing network comes into being, the distributed system comes into people's sight. A distributed system is one in which components located at networked computers communicate and coordinate their actions only by sending messages [4]. Since the processes in it only communicate by messages, there isn't any global clock which can indicate global states. Therefore, to make a collection of machines in one distributed system that operate as a whole system, protocols are needed to ensure the system's function despite a failure of a limited number of components [5]. These protocols require cooperation and agreements among the processes. For

example, in the remote replicated file system, the server nodes must agree on where the file copies are supposed to reside.

A simple method to achieving agreement is for the processes to vote and agree on the majority value. This leads to the primary consensus algorithm. However, this voting method can only work well in the absence of faults. If some faulty processes send misleading or out-of-data votes, the other processes may get confused and fail to reach an agreement. In paper [6], Davis and Wakerly introduced a multistage voting scheme to solve this problem.

Some problems were kept unsolved until the Paxos was introduced by L Lamport [7] to implement a fault-tolerant distributed system. Paxos is a typical consensus algorithm. So, what is consensus? Consensus is a fundamental element in fault-tolerant distributed systems. It involves multiple servers agreeing on the same values. Once they reach a decision on a value, that decision is final and the whole server cluster should agree on that value.

In Paxos algorithm, since there are no global clocks in distributed systems, if there are some proposed values to run on distributed system nodes, the consensus algorithm must ensure that only a single one can be processed, especially in a large-scale reliable system. Each node has three roles, the proposers, the acceptors and the learners. When there are some proposals which are usually called values (such as a client requests in client/server model), the proposers package the proposals (values) and send them to acceptors. Each acceptor only agrees with one proposal in a term and sends agreement promise back to proposers once it has agreed with one proposal. After receiving a sufficient amount (generally more than half) of promises on one proposal, the proposer will tell all other nodes to learn (replicate) the agreed proposal by a connection between proposers, acceptors and learners.

Although the Paxos algorithm is quite helpful and has influenced most consensus applications in many years, it's quite hard to understand and apply to practice. After struggling with it many researchers had proposed some adjustments on Paxos.

In paper [8], BW Lampson agreed with main ideas of L Lamport and introduced another way to construct an efficient distributed system out of replicates. The idea is to run a replicated deterministic state machine and get consensus on each input. Then use one special process to improve efficiency by replacing consensus steps with actions. Both the BW Lampson and L Lamport's methods have no real-time guarantees and ensure the consensus by repeating rounds until there exists the same value. At last, BW Lampson explained how to design and understand a concurrent, fault-tolerant system. The critical step is to write a simple spec as a state machine, find the abstraction function from the implementation to the spec, establish suitable invariants, and show that the implementation simulates the spec. This method works for lots of hard problems.

In 2001, Butler W. Lampson combined the consensus and state machine and introduced a fault-tolerance mechanism [9]. The paper [9] showed that we can replicate state machines based on Paxos consensus algorithm. It also described an abstract version of Lamport's Paxos algorithm for asynchronous consensus. Then Butler W. Lampson derived different versions of Paxos algorithm from that abstract version, such as the Byzantine version, classic version, then discussed

the safety, liveness, and performance of each one, and gave the abstraction functions and invariants for simulation proofs of safety. It's novel but Paxos still can't solve the Byzantine problem well.

In paper [6], L Lamport improved his Paxos algorithm that shortened the time for any process to learn the chosen value from 3 message delays to 2 message delays. In classic Paxos, values are not proposed by the same processes that choose the value. We can explain it in a client/server model. The clients propose the next command, such as uploading a file or downloading a file in our remote file system, then the servers choose one proposed command. It's different nodes to propose and choose. In total, three message delays are required between when a client proposes a command and when a process learns which command has been chosen. In Fast Paxos, L Lamport made an optimization that in normal case learning occurs in two message delays without collision. Although the Fast Paxos has lower latency than Classic Paxos in most cases, the Fast Paxos needs more replicates and has collisions which don't exist in Classic Paxos and lead to higher latency. In paper [10] an experiment showed that when facing large amounts of messages from the client to servers the Classic Paxos performs better and has lower latency than Fast Paxos.

However, the Paxos still had dominated the applications and research of consensus algorithms for more than one decade since published.

In 2014, based on the main idea of Paxos, D Ongaro and J Ousterhout introduced a new and more understandable consensus algorithm, Raft [11]. Raft algorithm is a consensus algorithm ensuring the consensus between nodes by replicating logs. It is based on Paxos algorithm but easier to understand. The Raft algorithm has different structure but same efficiency with Paxos. It separates the key elements of consensus algorithms into three parts: leader selection, log replication and safety, which we will describe in Section 5.

Similar to Paxos, the nodes in Raft algorithm also have three roles, the leader, the candidate and the follower. The difference is that in normal cases the nodes in the raft only need two roles, leader and follower, which simplified the consensus mechanism quite a lot more than Paxos. The Raft has similar performance to other consensus algorithms such as Paxos, however, it is much easier to practice and performs best when an established leader is replicating new log entries. So, raft is implemented in this project to handle a multi-server consensus, and it performs well.

In recent years, the Raft algorithm has also developed a lot. In paper [12], C Copeland, H Zhong combined the classic Raft algorithm and Byzantine Fault Tolerance algorithm and proposed BFTRaft, Byzantine Fault Tolerance Raft algorithm. However, their algorithm gives the client power to start an election so is not very practical. Besides, there is also some research about the Raft and blockchains [13].

## 3. Comparative Analysis

Assume that the consensus problem has a set of processes. Each process can be divided into two parts, proposing its initial value and deciding the value. Consensus algorithm, which ensures that a single one among the proposed values is chosen, refers to a protocol to solve consensus problems. The safety requirements of a consensus algorithm include termination, agreement and integrity. Termination means that each correct process eventually decides a value. Agreement

means that the values decided by all correct processes are the same. Integrity means that if all correct processes propose the same value, the values decided by all correct processes are exactly the proposed value. Common consensus algorithms include Paxos, Raft, ZooKeeper Atomic Broadcast (Zab), Chandra-Toueg consensus algorithm and so on.

## 3.1 Paxos

After the proposal of Paxos protocol, it has become the most popular consensus algorithm. Raft algorithm is the simplified method of Paxos algorithm. Both of them use a leader-based approach to deal with consensus problems. There are three states that any server could become, follower, candidate and leader. A heartbeat failure detection is implemented to detect whether the leader has crashed. If the leader has crashed, there will be a leader selection process. Figure 1 shows two phases in a round of Paxos algorithm. In the first phase, the leader proposes a value in a prepared request to other followers and waits for a majority of responses. If all responses are ACK, the leader updates the estimated decision value to the value in the ACK, otherwise the leader aborts the round. In the second phase, the leader sends the estimated decision value in an accepted request to followers then the followers reply. If all responses are ACK, the leader sends the decision value, otherwise aborts the round.[14]

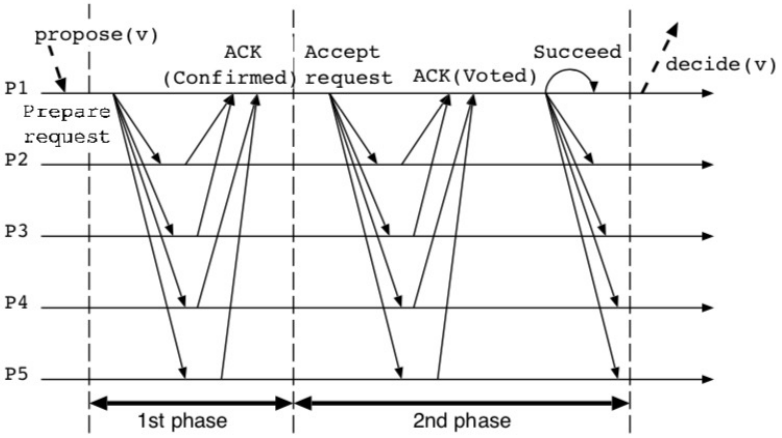


**Figure 1** Phases in a round of the Paxos algorithm.

The differences between Paxos and Raft are listed as below. First of all, in Raft algorithm a follower could become a candidate in any term while in Paxos algorithm a server s can only be a candidate in the term t if t mod n equals s. Secondly, as for ensuring that a new leader has newest log, in Raft algorithm a candidate could vote only when the candidate's log is as up-to-date as the followers' while in Paxos algorithm a candidate updates its log when it receives RequestVote responses from a majority of followers, which contains the follower's log entries. Thirdly, as to how the leader commit log entries from previous terms, in Raft algorithm the leader replicates the log entries to other servers without changing the term while in Paxos the leader adds log entries from previous terms to its log with its term and then replicates the log entries.[15]

Paxos algorithm has some drawbacks. On one hand, Paxos algorithm is quite difficult to understand therefore Raft algorithm was born to simplify the method. On the other hand, Paxos algorithm is not practical because it does not provide a good foundation for implementation, such as no broad agreement on multi-Paxos algorithm.[11]
Overall, the Raft algorithm is more understandable and efficient than the Paxos algorithm.

## 3.2 ZooKeeper Atomic Broadcast (Zab)

Zab is a consensus protocol used in Zookeeper. Both Raft and Zab are based on the state machine, operation log and snapshot mechanism to store data. Figure 2 shows three phases of Zab. In the first phase, the followers send their last promise to the prospective leader and then the leader proposes a new epoch. The followers send ACK messages to the leader. In the second phase, the prospective leader proposes itself as the new leader in this epoch, receives the ACK messages replied by the followers and then commits the new leader proposal. In the third phase, the leader proposes a new transaction, the followers acknowledge leader proposal and then the leader commits proposal.[16]

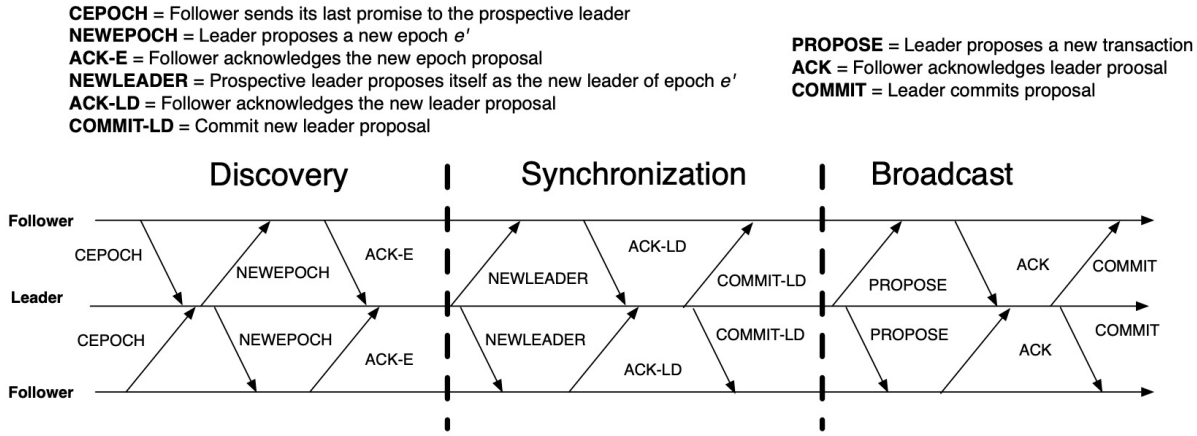


**Figure 2** Zab protocol summary

There are some differences between the two algorithms. To begin with, in terms of the way of processing the requests, the client in Zab establishes a long TCP connection with any node to complete all interactive actions, which may lead to the data out-of-date in read request. Therefore, Zab uses sync() to ensure the data up-to-date. However, in Raft the client randomly obtains a node, finds the leader, and then directly interacts with the leader. Secondly, as for leader selection, in Zab the selected leader really works after synchronizing the log between the leader and the followers while in Raft the leader becomes the leader after it is selected and uses other methods to avoid crash. Thirdly, as for log synchronization, Zab uses two-phase commit, voting phase, where the leader receives more than a half of consent votes from followers, and committing phase, where the data is transferred to the followers and appended to their log, whereas Raft uses heartbeats from the leader to followers, which means the leader sends heartbeat to followers and then receives ACK from followers and replicate log to followers. Some other differences are not listed here.

Zab protocol has some disadvantages. When electing, each server can vote multiple times in a certain election epoch round, update it whenever it encounters a larger vote, and then distribute new votes to everyone. Therefore, the election time is theoretically more expensive than Raft and it may cause live-lock problems.

## 3.3 Chandra-Toueg consensus algorithm

Chandra-Toueg algorithm uses the rotating coordinator paradigm and the failure detector. Figure 3 shows a round in the Chandra-Toueg algorithm. In the first phase, each server proposes its estimation of the decision value to the leader. In the second phase, the leader receives the estimations from both itself and the followers, selects the newest estimation of decision value and sends it to the followers. In the third phase, the followers receive the message from the leader and then send ACK messages to the leader. If the failure detector of a follower suspects the leader, it sends a NACK message. In the fourth phase, the leader receives the responses from the followers. If the responses are all ACK, the leader decides the value and broadcasts it, otherwise aborts the round.[14]

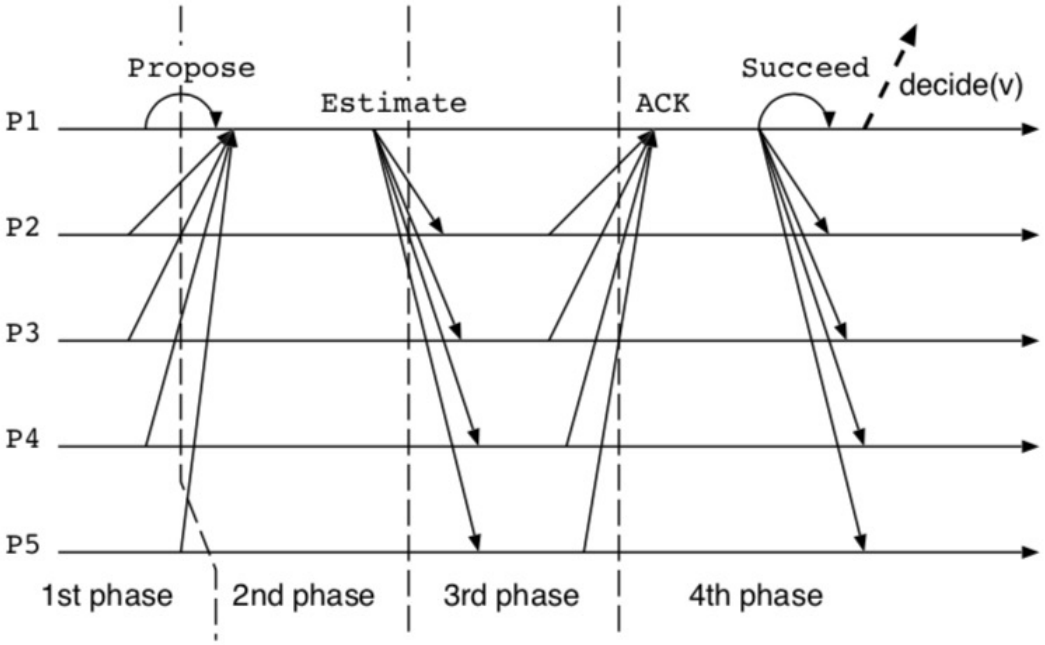


**Figure 3** Phases in a round of the Chandra-Toueg algorithm.

It is shown that Chandra-Toueg algorithm has worse latency of consensus, and therefore is less efficient than Paxos when the leader process immediately crashes after starting the round of the consensus algorithm while the latencies of two algorithms are almost the same in other cases.[14] In our work it is necessary to stop the leader manually or it is possible that the leader could crash randomly. Therefore, Chandra-Toueg algorithm is not suitable for our work.

# 4. Discussions, Future Direction and Application Domains

## 4.1 System Architecture

To build the application, a cloud system including four instances has been constructed with the help of Amazon Web Services (AWS). Among the four instances, Apache CouchDB is deployed on one of them. The other three are used for the deployment of RESTful Web Services based on the lightweight WSGI web application framework - Flask and the synchronization system based on the Raft Consensus Algorithm. Figure 4 depicts the system architecture of our application.

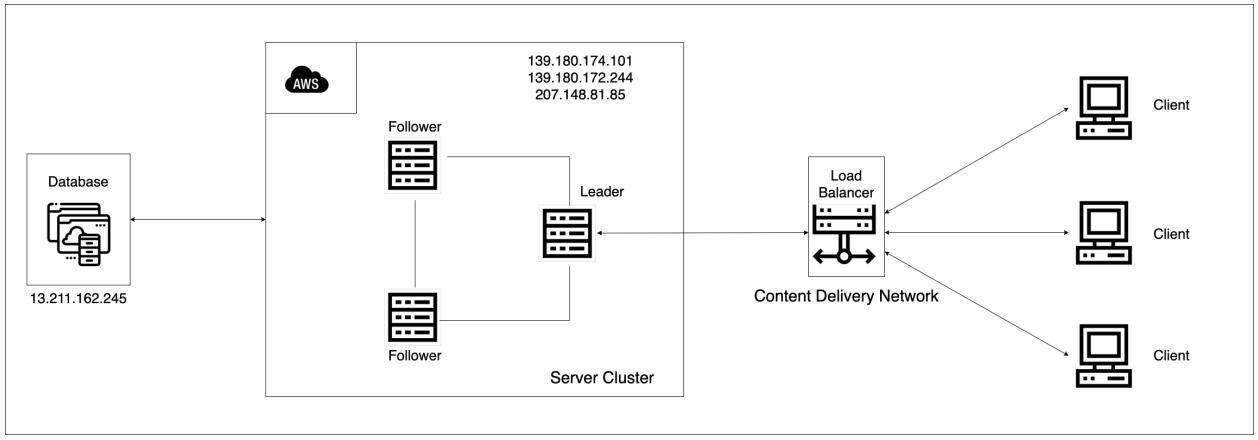


**Figure 4** System architecture diagram.

From the next section, we are going to introduce the key technologies used in the application.

### 4.1.1 Representational State Transfer

REST is an architecture of good functional, performance, and social properties. It is defined in Roy Fielding's 2000 Ph.D. dissertation at UC Irvine[17]. Six following guiding constraints define a RESTful system:

- Client-server architecture:
  Split the data storage concerns and user interface concerns to improve the portability.
- Statelessness:
  Server would not store the information of clients.
- Cache ability:
  Clients can cache responses in order to improve performance.
- Layered system:
  A client does not know whether it is connected to the end server or to an intermediary along the way. A load balancer or proxy would not affect the communications between server and client.

- Code on demand:
  The server can temporarily extend or customize functions by sending executable code to the client.
- Uniform interface:
  The architecture is simplified by following the uniform interface constraint. In RESTful Web services, each resource has an identification. The resources could be called for by using standard HTTP methods including GET, POST, PUT, DELETE and PATCH.

We use Flask to build RESTful APIs. Flask is a popular web framework within the Python community. It is flexible, lightweight and easy to use. The interfaces of our application are as Table 1:

| Interface | Method | Usage |
|---|---|---|
| /users/<name>/<password> | GET, POST | To allow a user to register or login |
| /uploads/<token> | GET, POST | To upload a file or get the list of existing files |
| /uploads/<filename>/<token> | GET, DELETE | To download or delete a file from the server |
| /uploads/<oldfilename>/<newfilename>/<token> | PUT | To change a file's name |
| /sharedfiles/<token> | GET | To obtain the list of shared files which are shared by other users |
| /sharedfiles/<targetusername>/<filename>/<token> | GET, POST, DELETE | To accept, decline or create a sharing request. |
| /leader | GET | To obtain the address of current leader nodes |

**Table 1** interfaces of application in framework

The first six interfaces are essential for the business logic, and the last gives the client developer a convenient way to interact with the server cluster without a load balancer.

### 4.1.2 Persistence

We use Apache CouchDB to achieve persistence. CouchDB is a database supported by the Internet. In CouchDB, a number of databases can be created, and the data saved in databases as JSON documents. There are many approaches to process the CouchDB, such as JavaScript or using API in python. The most important is that CouchDB could be configured as a single node database

and a clustered database. It gives developers freedom to build the application. Besides, its unique replication protocol also offers lessons for implementing synchronization algorithms.

### 4.1.3 Load balancing

In recent years, the rapid growth of cloud computing has caused a massive increase in the number of service requests to cloud servers. As the demand for cloud computing increases, the distributed systems become more and more complicated. Hence, effective load balancing techniques become more and more important.

In computing, load balancing refers to balancing and spreading the load (work task) to multiple operating units to perform in order to improve the efficiency of the whole system. Load balancing techniques could optimize the response time for each task and avoid overwhelming particular nodes. A typical load balancer balances the load by using four steps [18]:

- Receive service requests from clients.
- Calculate the size of the incoming load request from clients and create a request queue.
- Check the current load status of the server cluster periodically by using a server monitor daemon.
- Uses an appropriate load balancing strategy to select appropriate servers from the pool.

## 4.2 Challenges and Solutions

### 4.2.1 Single Node Deployment

The first challenge is to coordinate the Web service and the synchronization system. Although it is lightweight and easy to use, Flask's built-in server is not the best for production as it does not scale well. We choose uWSGI as a deployment option on the server. uWSGI is both a protocol and an application server which coule serve uWSGI, FastCGI, and HTTP protocols. Nginx is a web server which could be used as a reverse proxy, mail proxy and HTTP cache. It can also serve as a software load balancer.

### 4.2.2 Multi-Node Deployment

Multi-node deployment is a must in a system which is designed to demonstrate Raft algorithm. In that case, batch deployment that enables you to deploy multiple servers at once is particularly useful, and that is why we use Docker. Docker is an open platform which could develop, ship, and run applications, separating the application from the infrastructure, to simplify the delivery process. Compared to the Virtual Machine deployment method, Docker has its unique advantages, containers can run with very minimal CPU and memory requirements, which can reduce resource utilization. CouchDB, and Nginx Server are run in instances as a docker container. Applications running in instances are used as docker containers.

#### 4.2.3 Optimization

To shorten the response time, a simple but effective method is to trigger the synchronization only when it is necessary to do so. For instance, the synchronization system should not deal with a request which would not change the data on the server (e.g., a request to purely obtain the list of existing files). Besides, a shared database no matter a single node one or a cluster could also help. Undoubtedly, to set the load balancer in a content delivery network would also be a good idea.

## 5.Raft algorithm

## 5.1 Raft Basic

Usually, a Raft cluster should exist more than a single server, in our project, we use three servers. In our case, our distributed system can tolerate one server crash unexpectedly. All of the servers are either one of the following three statuses: leader, follower, or candidate [11]. Normally, the leader server is the single and should be the only one server from the server cluster. All of the other servers that are not the leader should be the followers. Leader server should handle all the requests that are sent out by the clients and if in the case that client sends the request to the follower servers, that request should be redirected to the leader server. In the other words, followers don't actively operate. They receive the requests from the leader server and candidate server, and then react according to the requests that they have received only. The flowchart for how the follower, candidate, and leader cooperate with each other is showing as figure 5.

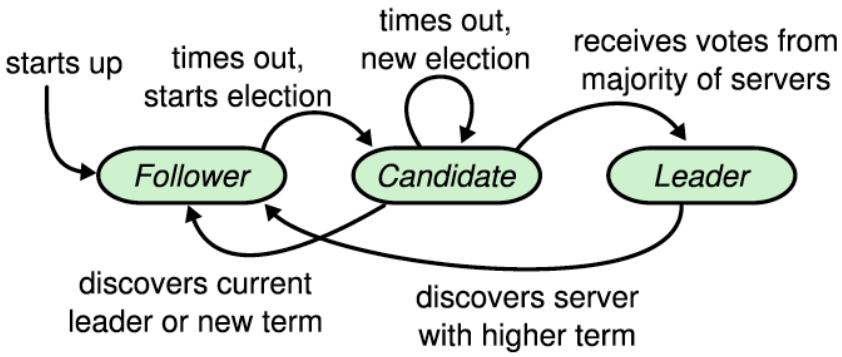


**Figure 5** How can a follower successfully be selected as a leader [11]

The terms of a leader depend on the constant length that we give. Once the server cluster monitors that there is no more successful heartbeat message, they imply that the current leader term is ended, and followers are now becoming the candidates that try to elect the new leader. As the figure 5 shown at the below, Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term.

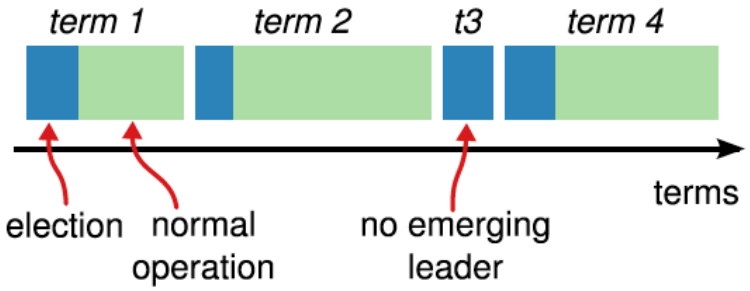


**Figure 6** leader election regarding on the time [11]

### 5.1.1 Leader election

The way that Raft uses to monitor the moment for election is based on the heartbeat message. When the server cluster is activated, all of their initial states are the followers or the leader. Followers would continuously receive the Remote produce call after every certain length of time from the leader. The message that is sent out by the leader is the heartbeat message. If followers are unable to receive that message over a period of time, followers will assume that the leader thread is ended unexpectedly and change their states to the candidate to elect the new leader.

Before the followers really proceed the election, they need to increase one term to the current term. Then, followers change their status to the candidate and require other followers or candidates to vote for them. They will remain their status unless one of the two following conditions are satisfied.

- One of the candidates wins the election and become the new leader:

The requirement that a candidate is needed to win the election is that it needs to obtain the vote from the majority of servers. For example, if we have N servers in our server cluster, candidates will be needed (N+1)/2 votes to win the election.

- Candidate cannot elect a new leader after a certain length of time:

This situation might be caused by a lot of reasons. For example, two of the candidates receive the exact same number of votes, but only a single leader is allowed in Raft. Or, there are too many candidates requiring the votes from others, so there is no candidate who can acquire the majority number of supports from others.

If there exists a candidate that wins the election, it becomes to the new leader now. This is the beginning of a new term, and the new leader would restart to send out the Remote produce call to other servers. Once servers receive the heartbeat message that was sent out by the new leader, they realize that the new leader is elected out and they will change back their states to the follower.

If no candidate wins the election, it is called the election time out. If this situation happens, all of the candidates will increase their term number by one and start a new election period. However, if we have a large scale of server cluster, this situation probably won't change even if we reelect again and servers might fall into a deadlock to vote for the new leader forever. That is the basic reason why Raft election time out length is a random number between an interval. This

solution prevents servers from frequently splitting out the vote so that no candidate can acquire the majority. It is because that for the majority of times, by using this mechanism only a single server would face the issue of time out. Since the limit of election time is random, there is always one of the servers that would reach the limit. Therefore, before other severs fall into the problem of election time out, it can firstly win the election and send out the heartbeat message.

By comparing to other leader-based consensus algorithms, we may notice that the design of the leader election process for Raft is the most complete and understandable than others. Other consensus algorithms may suffer from the election timeout issue if they don't have such randomized election period mechanisms [11].

### 5.1.2 Log replication

After the leader election, the server cluster can start to receive and serve client requests. In the Raft algorithm and the remote file system in this report, the request from the client includes a command to be executed by all servers. So, it's necessary to run this command by the replicated state machine. When receiving a request, the leader appends the command (entry) in that request into its log, then calls the function 'AppendEntries' and tells its followers to replicate the command (entry). After safe replication, the leader will add the command (entry) to its own state machine and return the executed result to the client as a response. If followers crash or network packets are lost, the leader will repeat the 'AppendEntries' function until every follower keeps the same log as the leader. Above is a normal log replication process as Figure 7, but it's quite different if there exist some faults.

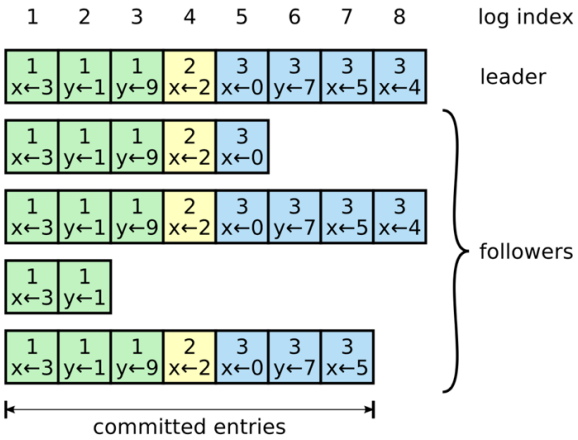


**Figure 7** Logs consist of entries. Each entry consists of the term when It was created and the command to be executed on state machines. [11]

When an entry's log is replicated by a sufficient large majority of followers, the leader will receive its followers' responses and regard that entry as committed entry. A committed entry represents the log replication for that entry is successful and safe, and the command in it can be executed.

In normal cases the leader's log and the follower's logs can stay consistent perfectly, and the leader's log is the authority for all other nodes to follow.

However, if the leader crashes and another node win the next election, the logs will be inconsistent, as shown in Figure 8. To solve this problem, the raft algorithm requires all nodes to replicate the leader's log. That means all nodes' logs can only be sub-lists of the leader's

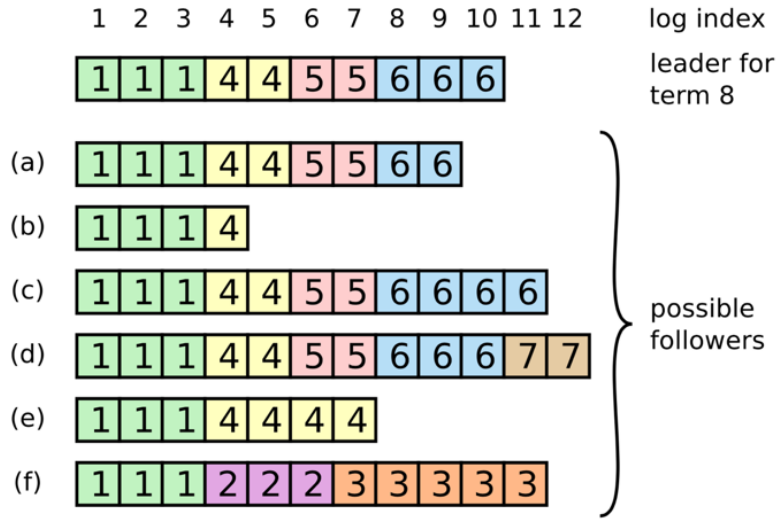


**Figure 8** When a leader just comes into power, there are many possible situations of its followers' logs. A follower may miss or have extra uncommitted logs.[11]

To force a follower's log consistent with its own, the leader needs to find the longest entry list the two logs agree and get the latest entry index in that list. All entries in follower's log after the latest entry will be removed and the leader will send all entries after the latest entry to the follower, force the follower to replicate until their logs are all the same.

Thanks to the above mechanism, the leader needn't take any special steps to maintain the authority log. Every elected leader just needs to follow the steps above to ensure the log consistency of all node's log.

### 5.1.3 Safety

We have introduced the several similar consensus algorithms above at section 3, and now it is the time that we decide which one is the most suitable for our dropbox application and how we can implement them in reality. There are a couple of advantages and disadvantages of the Raft algorithm that need to be considered before we choose Raft to implement in our application.

In the last two subsections, the leader election and log replication processes were described. In this subsection, more mechanisms used to ensure the safety of a consensus algorithm will be introduced.

- Election restriction: Election restriction means that the leader must store all of the committed log entries eventually. The raft ensures that by forcing log entries can only flow

from leaders to followers while log replication. Besides, in the leader election process, a candidate can't win an election if it doesn't contain all committed entries.

- Committing entries from previous: The term is the proof of the validity of the one message. Raft treats the out-of-valid messages by only admitting log entries with current leader's term.
- Leader Completeness Property: If one log entry with a certain term is committed in the log replication process, then that entry will exist in all the higher-numbered terms leaders' logs. The Leader Completeness Property can ensure the state machine safety property and was proved in paper [11] with many arguments.

## 5.2 Advantages and Limitation

### 5.2.1 Advantages

- **More suitable for the educational purposes:**

  Since our goal for this project is for an educational purpose, Paxos family's algorithms are obviously too complicated to be the best option for us. Our main goal is not only to implement the raft algorithm in our application, but also understand why it works efficiently or underestimated. If we compare the Paxos family's algorithms with Raft, we can observe that Raft has a clearer flow path for the leader election and log replication.

  Raft is really similar to other consensus algorithms, but there are several unique characteristics of Raft that make it more reliable than other consensus algorithms.
- **Stronger leader:**

  The title of the leader in Raft consensus algorithms is stronger compared to other consensus algorithms. In Raft consensus algorithms, the log replicated processes and appending entries' messages are all conducted by the leader server. Once one of the servers is granted the vote from more than (N+1)/2 servers, this server also obtains the highest level of commanding rights. Since the leader server is the only server that needs to manage the log replication processes, it makes Raft more understandable to learners like us.
- **Member changes:**

  The approach that Raft uses to change the set of servers in the cluster is a new joint consensus method, which means it ensures that the majorities of two different configurations overlap during transitions. This unique characteristic gives Raft the ability to achieve the goal that cluster servers can continually operate and ignore the influence that was generated by configuration changes.
- **Leader election:**

  In our application, Raft sets a constant number as the time limit to monitor if their leader in the server cluster is still working properly. Once the server cluster observes that the leader server crashed, they will start a new leader term election. This unique characteristic of what it's called as heartbeats ensure Raft can solve the conflicts more efficiently.

### 5.2.2 Limitations

- **Strict to single leader sever:**

  There are also some of the disadvantages for the Raft consensus algorithm. The first one is that Raft is strict with only one leader server, so the clients communicate with the leader server only. Just like the situation that we have discussed above, if we have too much traffic on the leader server, for example, we have a huge amount of nodes, the system might run out of resources. In the worst case, the safety of the entire distributed system might be threatened.

- **Not Byzantine failures tolerant:**

  When we implement the raft consensus algorithm, we have to assume that the Byzantine failures won't appear, and this assumption greatly reduces the applicability that implement this algorithm in reality. Just imagine a candidate of followers that votes twice in a given term, or votes for another candidate that has a log which is not up to date like its own and that candidate becomes leader. Such behavior could cause the controversy of a real leader because those two nodes believe that they should be the leader at the same time. This situation would also cause the inconsistencies in the log.

  Many other scenarios like sending fake but valid heartbeat messages are also examples that show Raft is not byzantine fault tolerant.

## 5.3 Implementation Detail

In this project, we implemented a fault-tolerant remote file system based on the classic raft algorithm. The clients can connect to the server cluster by Http requests. The server cluster in this demo consists of three servers deployed on three VPS (Virtual Private Server). The three servers connect with each other based on TCP and UDP connections and are implemented with the Raft algorithm. After three servers successfully launch, they will elect one leader and two followers based on the Raft. The online client can sense which one is the leader server and always prefer to connect and send requests to the leader. When the leader Server receives any requests, it will process a log replication process in raft to force its followers to copy the log entry. When the process succeeds, the leader will run the command in that request, respond to clients and replicate the state machine for its followers. Above is the normal case for this demo.

When the leader crashes, the other followers can't receive the leader's heartbeat and then become candidates after a random time (a maximum message delay time). Then a new leader election will start. The leader election process detail is described in Section 5. We have to explain for a special case. During the leader election, the online clients may can't connect to the leader server because the leader has crashed, so it will try to connect to other alive servers. The alive servers can still solve some simple requests that don't change the state machine, such as checking the files, downloading an existing file, but can't process any requests that need a raft process that changed the state machines. This design promises that the servers in a cluster always keep the same state machine and simplifies the log replication. After a successful leader election, online clients can still connect to new leaders and enable all functions.

To handle more problems, such as a large cluster containing 1,000 server nodes, shorten the leader election time, some discussions and future directions are in Section 4.

# 6. Conclusion

In this project, we did the research and have a clear understanding based on the consensus algorithms. There are definitely some advantages and disadvantages of those algorithms and based on our comparative analysis on section 2 and section 3, we found that the Raft algorithm is the most suitable to implement in our C/S based application. We used Amazon Web Services (AWS) (Apache CouchDB to be specific) cloud system to achieve the establishment of our server cluster. The API that we have chosen to use is Flask and that is illustrated at section 4. We also briefly discuss the processing flow about how the leader can be elected and how the leader can command the followers with the log replication mechanism in section 5. We listed several advantages why we chose the Raft algorithm. However, there are still some limitations for the Raft consensus algorithm and we also faced some issues when we tried to implement this algorithm on our application. All in all, we believe that for an educational purposes project, we implement the raft algorithm reasonably and successfully and we also believe that those challenges that were mentioned in section 4 can be solved in the future work.

## References


1. Fischer, Michael J. "The Consensus Problem in Unreliable Distributed Systems (A Brief Survey)" (PDF). Retrieved 21 April 2014.
2. LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.
3. LAMPORT, L. Paxos made simple. *ACM SIGACT News 32*, 4 (Dec. 2001), 18–25.
4. Coulouris, G. F., Dollimore, J., & Kindberg, T. (2005). *Distributed systems: concepts and design*. pearson education.
5. Fischer, M. J. (1983, August). The consensus problem in unreliable distributed systems (a brief survey). In *International conference on fundamentals of computation theory* (pp. 127-140). Springer, Berlin, Heidelberg.
6. Lamport, L. (2006). Fast paxos. *Distributed Computing*, *19*(2), 79-103.
7. Lamport, L. (2001). Paxos made simple. *ACM Sigact News*, *32*(4), 18-25.
8. Lampson, B. W. (1996, October). How to build a highly available system using consensus. In *International Workshop on Distributed Algorithms* (pp. 1-17). Springer, Berlin, Heidelberg
9. Lampson, B. (2001, August). The ABCD's of Paxos. In *PODC* (Vol. 1, p. 13).
10. Junqueira, F., Mao, Y., & Marzullo, K. (2007). Classic paxos vs. fast paxos: caveat emptor. *Proceedings of USENIX Hot Topics in System Dependability (HotDep)*.
11. Ongaro, D., & Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)* (pp. 305-319).
12. Copeland, C., & Zhong, H. (2016). Tangaroa: a byzantine fault tolerant raft.
13. Huang, D., Ma, X., & Zhang, S. (2019). Performance analysis of the raft consensus algorithm for private blockchains. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*.
14. Hayashibara, N., Urbán, P., Schiper, A., & Katayama, T. (2002). Performance comparison between the Paxos and Chandra-Toueg consensus algorithms. In Proc. Int'l Arab Conference on Information Technology (ACIT 2002) (No. CONF, pp. 526-533).
15. Howard, H., & Mortier, R. (2020, April). Paxos vs Raft: Have we reached consensus on distributed consensus?. In Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data (pp. 1-9).
16. Junqueira, F. P., Reed, B. C., & Serafini, M. (2011, June). Zab: High-performance broadcast for primary-backup systems. In 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN) (pp. 245-256). IEEE.
17. Fielding, R. T., & Taylor, R. N. (2000). Architectural styles and the design of network-based software architectures (Vol. 7). Irvine: University of California, Irvine.
18. Rahman, M., Iqbal, S., & Gao, J. (2014, April). Load balancer as a service in cloud computing. In 2014 IEEE 8th International Symposium on Service Oriented System Engineering (pp. 204-211). IEEE.