

2022-03-09

# Title to be decided

David Kleingeld

Email

—

# Contents

<b>1 Background</b>	<b>2</b>
1.1 Distributed Computing .....	2
1.2 Faults and Delays .....	2
1.3 Consensus Algorithms .....	4
1.4 File System .....	15
1.5 Distributed file systems .....	16

# 1 Background

**TODO write tiny intro, do after introduction has been written**

## 1.1 Distributed Computing

When state of the art hardware is no longer fast enough to run a system the option that remains is scaling out. Here then there is a choice, do you use an expansive, reliable high performance supercomputer or commodity servers connected by Ip and ethernet? This is the choice between High Performance (HPC) and Distributed Computing. With HPC faults in the hardware are rare and can be handled by restarting, simplifying software. In a distributed context faults are the norm, restarting the entire system is not an option or you would be down all the time. Resilience against faults comes at an, often significant, cost to performance. Fault tolerance may limit scalability. As the scale of a system increases so does the frequency with which one of the parts fails. Even the most robust part will fail and given enough of them the system will fail frequently. Therefore at very large scales HPC is not even an option.

## 1.2 Faults and Delays

Before we can build a fault resistant system we need to know what we can rely on. While hardware failures are the norm in distributed computing, faults are not the only issue to keep in mind.

It is entirely normal for the clock of a computer to run slightly to fast or to slow. The resulting drift will normally be tens of milliseconds [3] unless

special measures are taken<sup>1</sup>. Even worse a process can be paused and then resumed at any time. Such a pause could be because the process thread is pre-empted, because its virtual machine is paused or because the process was paused and resumed after a while<sup>2</sup>.

In a distributed system the computers (*nodes*) that form the system are connected by IP over ethernet. Ethernet gives no guarantee a packet is delivered on time or at all. A node can be unreachable before seemingly working fine again.

Using a system model we formalize the faults that can occur. For timing there are three models.

1. The Synchronous model allows an algorithm to assume that clocks are synchronized within some bound and network traffic will arrive within a fixed time.
2. The Partially synchronous model is a more realistic model. Most of the time clocks will be correct within a bound and network traffic will arrive within a fixed bound. However sometimes clocks will drift unbounded and some traffic might be delayed forever.
3. The Asynchronous model has no clock, it is very restrictive.

For most distributed systems we assume the Partially Synchronous model. Hardware faults cause a crash from which the node can be recovered later. Either automatically as it restarts or after maintenance.

<sup>1</sup>One could synchronize the time within a datacenter or provide nodes with more accurate clocks

<sup>2</sup>On linux by sending SigStop then SigCont

## 1.3 Consensus Algorithms

In this world where the network can not be trusted, time lies to us and servers will randomly crash and burn how can we get anything done at all? Lets discuss how we can build a system we can trust, a system that behaves *consistently*. To build such a system we need the parts that make up the system to agree with each other, they must have *Consensus*. Here I discuss three well known solutions. Before we get to that lets look at the principle that underlies them all: *The truth is defined by the majority*.

### Quorums

Imagine a node hard at work processing requests from its siblings, suddenly it stops responding. The other nodes notice it is no longer responding and declare it dead, they do not know its threads got paused. A few seconds later the node responds again as if nothing had happened, and unless it checks the system clock, no time has passed from its perspective. Or imagine a network fault partitions the system, each group of servers can reach its members but not others. The nodes in the group will declare those in the other group dead and continue their work. Both these scenarios usually result in data loss, if the work progresses at all.

We can prevent this by voting over each decision. It will be a strange vote, no node cares about the decision itself. In most implementations a node only checks if it regards the sender as trustworthy or alive and then vote yes. To prove liveness the vote proposal could include a number. Voters only vote yes if the number is correct. For example if the number is the highest they have seen. If a majority votes yes the node that requested the vote can be sure

it is, at that instance, not dead or disconnected. This is the idea behind "Quorums," majorities of nodes that vote.

## Paxos

The Paxos algorithm [7] uses a quorum to provide consensus. It enables us to choosing a single value among proposals such that only that value can be read as the accepted value. Usually it is used to build a fault tolerant distributed state machine.

In Paxos there are three roles: proposer, acceptor and learner. It is possible for nodes to fulfill only one or two of these roles. Usually, and for the the rest of this explanation each node fulfills all three. To reach consensus on a new value we go through two phases: prepare and accept. Once the majority of the nodes has accepted a proposal the value included in that proposal has been chosen. Nodes keep track of the highest proposal number  $n$  they have seen.

Lets go through a Paxos iteration from the perspective of a node trying to share something, *a value*. In the first phase a new *value* is proposed by our node. It sends a *prepare* request to a majority of acceptors. The request contains a proposal number  $n$  higher then the highest number our node has seen up till now. The number is unique to our node<sup>3</sup>. Each acceptor only responds if our number  $n$  is the highest it has seen. If an acceptor had already accepted one or more requests it includes the accepted proposal with the highest  $n$  in its respons.

In phase two our node checks if it got a response from the majority. Our node

<sup>3</sup>This could be nodes incrementing the number by the cluster size having initially assigned numbers 0 to *cluster\_size*

is going to send an accept request back to those nodes. The content of the accept request depends on what our node received in response to its prepare request:

1. It received a response with number  $n_p$ . This means an acceptor has already accepted a value. If we continued with our own value the system would have two different accepted values. Therefore the content of our accept request will be the value from proposal  $n_p$ .
2. It received only acknowledging replies and none contained a previously accepted value. The system has not yet decided on a value. The content of our accept request will be the value our node wants to propose but with our number  $n$ .

Each acceptor accepts the request if it did not yet receive a prepare request number greater than  $n$ . On accepting a request an acceptor sends a message to all learners<sup>4</sup>. This way the learners learn a new value as soon as it's ready.

Let's get a feeling why this works by looking at what happens during node failure. Imagine a case where a minimal majority  $m$  accept value  $v_a$ . A single node in  $m$  fails by pausing after the first learners learned of the now chosen value  $v_a$ . After freezing  $m - 1$  of the nodes will reply  $v_a$  as value to learners. The learners will conclude no value has been chosen given  $m - 1$  is not a majority<sup>5</sup>. Acceptors change their value if they receive a higher numbered accept request. If a single node changes its value to  $v_b$  consensus will break since  $v_a$  has already been seen as the chosen value by a learner. A new proposal that can result into higher numbered accept requests needs a majority

<sup>4</sup>remember every node is a learner

<sup>5</sup>this is not yet inconsistent, Paxos does not guarantee consistency over whether a value has been chosen

response. A majority response will include a node from  $m - 1$ . That node will include  $v_a$  as the accepted value. The value for the accept request then changes to  $v_a$ . No accept request with another value than  $v_a$  can thus be issued.

Another value  $v_b$  will therefore never be accepted. The new accept request is issued to a majority adding at least one node to those having accepted  $v_a$ . Now at least  $m + 1$  nodes have  $v_a$  as accepted value.

To build a distributed state machine you run multiple instances of Paxos. This is often referred to as Multi-Paxos. The value for each instance is a command to change the shared state. Multi-Paxos is not specified in literature and has never been verified.

## Raft

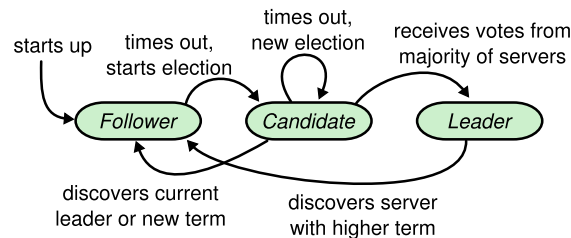
The Paxos algorithm allows us to reach consensus on a single value. The Raft algorithm enables consent on a shared log. We can only append to and reading from the log. The content of the log will always be the same on all nodes. As long as a majority of the nodes still function the log will be readable and appendable.

Raft maintains a single leader. Appending to the log is sequential because only the leader is allowed to append. The leader is decided on by a *quorum*. There are two parts to Raft, *electing leaders* and *log replication*.

**Leader election** A Raft [9] cluster starts without a leader and when it has a leader it can fail at any time. The cluster therefore must be able to reliably decide on a new leader at any time. Nodes in Raft start as followers,



monitoring the leader by waiting for heartbeats. If a follower does not receive a heartbeat from a *valid* leader on time it will try to become the leader, it becomes a candidate. In a fresh cluster without a leader one or more nodes become candidates.

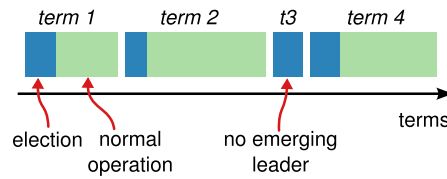


**Figure 1:** A Raft node states. Most of the time all nodes except one are followers. One node is a leader. As failures are detected by time outs the nodes change state. Adjusted from [9].

A candidate tries to get itself elected. For that it needs the votes of a majority of the cluster. It asks all nodes for their vote. Note that servers vote only once and only if the candidate would become a *valid* leader. If a majority of the cluster responds to a candidate with their vote that candidate becomes the leader. If it takes too long to receive a majority of the votes a candidate starts a fresh election. When there are multiple candidates requesting votes the vote might split<sup>6</sup>, no candidate then reaches a majority. A candidate immediately loses the election if it receives a heartbeat from a *valid* leader. These state changes are illustrated in fig. 1.

In Raft time can be divided in terms. A term is a failed election where no node won or the period from the election of a leader to its failure, illustrated in fig. 2. Terms are used to determine the current leader, the leader with the highest terms. A heartbeat is *valid* if, as far as the receiver knows, it originates from the current leader. A message can only be from the *current* leader if the

<sup>6</sup>Election timeouts are randomised so this does not repeat infinitely.



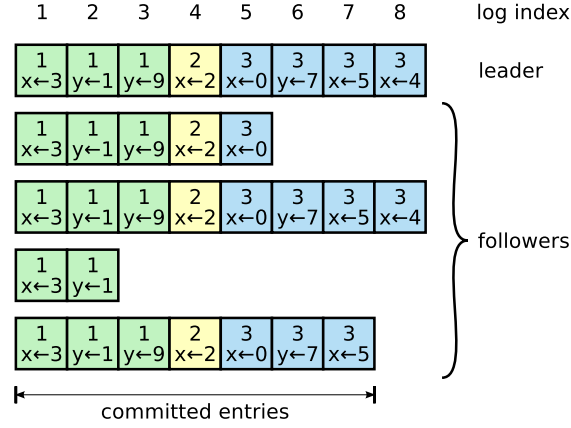
**Figure 2:** An example of how time is divided in terms in a Raft cluster. Taken from [9].

message *term* is equal or higher than the receiving node's term. If a node receives a message with a higher term, it updates its own to be that of the message.

When a node starts, its *term* is zero. If a node becomes a candidate, it increments its *term* by one. Now imagine a candidate with a *term* equal or higher than that of the majority of the cluster. When receiving a vote request, the majority will determine if this candidate could become a *valid* leader. This candidate will get the majority vote in the absence of another candidate and become the leader.

**Log replication** To append an entry to the log, a leader sends an append request to all nodes. Messages from invalid leaders are rejected. The leader knows an entry is committed after a majority of nodes acknowledged the append. For example, entry 5 in fig. 3 is committed. The leader includes the index up to which entries are committed in all its messages. This means entries will become committed on all followers at the latest with the next heartbeat. If the leader approaches the timeout and no entry needs to be added, it sends an empty append. There is no need for a special heartbeat message.

There are a few edge cases that require followers to be careful when appending. A follower may have an incomplete log if it did not receive a



**Figure 3:** Logs for multiple nodes. Each row is a different node. The log entries are commands to change shared variables  $x$  and  $y$  to different values. The shade boxes and the number at their top indicate the term. Taken from [9].

previous append, it may have messages the leader does not and finally we must prevent a candidate missing committed entries from becoming the leader.

1. To detect missing log entries, the entries are indexed incrementally. The leader includes the index of the previous entry and the term when it was appended in append requests. If a followers last log entry does not match the included index and term the follower responds with an error. The leader will send its complete log for the follower to duplicate<sup>7</sup>.
2. When a follower has entries the leader misses these will occupy indices the leader will use in the future. This happens when a previous leader crashed having pushed logs to some but not majority of followers. When the leader pushes a new entry with index  $k$  the follower will notice it already has an entry with index  $k$ . At that point it simply overwrites what

<sup>7</sup>This is rather inefficient, in the next paragraph we will come back to this

it had at  $k$ .

3. A new leader missing committed entries will push a wrong log to followers missing entries. For an entry to be committed it must be stored on the majority of the cluster. To win the election a node has to have the votes from the majority. Thus restricting followers to only vote for candidates that are as up-to-date as they are is enough. Followers thus do not vote for a candidate with a lower index than they themselves have.

**Log compaction** Keeping the entire log is rather inefficient. Especially as nodes are added and need to get sent the entire log. Usually raft is used to build a state machine, in which case sending over the state is faster than the log. The state machine send is a snapshot of the system, only valid for the moment in time it was taken. All nodes take snapshots independently of the leader.

To take a snapshot a node writes the state machine to file together with the current *term* and *index*. It then discards all committed entries up to the snapshot *index* from its log.

Followers that lag far behind now send the snapshot and all logs that follow. The follower wipes its log before accepting the snapshot.

## Consensus as a service

So the problem of consensus has been solved, but the solutions are non trivial to implement. We need to shape our application to fit the models the solutions use. If we are using Raft that means our systems must be built around a log.

Then we need to build the application being careful we implement the consensus algorithm correctly. A popular alternative is to use a coordination service. Here we look at ZooKeeper [5] an example of a wait free coordination service. I first focus on the implementation, drawing parallels to Raft before we look at the *Api* ZooKeeper exposes.

A ZooKeeper cluster has a designated leader that replicates a database on all nodes. Only the leader can modify the database, therefore only the leader handles *write* requests. The nodes handle *read* requests themselves, *write* requests are forwarded to the leader. To handle a *write* requests ZooKeeper relies on a consensus algorithm called Zab [6]. It is an atomic broadcast capable of crash recovery. It uses a strong leader quite similar to Raft and guarantees that changes broadcast by the leader are delivered in the order they were sent and after changes from previous (crashed) leaders. Zab can deliver messages twice during recovery. To account for this ZooKeeper turns change requests into *idempotent* transactions. These can be applied multiple times with the same result.

ZooKeeper exposes its strongly consistent database as a hierarchical name space (a tree) of *znodes*. Each *znode* contains a small amount of data and is identified by its *path*. Using the *Api* clients can operate on the *znodes*. They can:

- |                                      |   |
|--------------------------------------|---|
|                                      | 4. Sync with the leader                         |
| 1. Create new <i>znodes</i>          | 5. Query if a <i>znode</i> exists               |
| 2. Change the data in a <i>znode</i> | 6. Read the data in a <i>znode</i>              |
| 3. Delete existing <i>znodes</i>     | 7. Get list of the children of the <i>znode</i> |



The last three operations support watching: the client getting a single notification when the result of the operation changed. This notification does not contain any information regarding the change. The value is no longer watched after the notification. Clients use watching to keep locally cached data is up-to-date.

Clients communicating outside of ZooKeeper might need special measures to ensure consistency. For example: client A updates a *znode* from value  $v_1$  to value  $v_2$  then communicates to client B, through another medium (lets say over *tcp/ip*). In this example client A and B are connected to different ZooKeeper nodes. If the communication from A causes B to read the *znode* it will get the previous value  $v_1$  from ZooKeeper if the node it is connect to is lagging behind. Client B can avoid this by calling call *sync*, this will make the zookeeper node processes all outstanding requests from the leader before returning.

Raft has the same race condition like issue. We might think we can just ensure a heartbeat has passed, by then all (functioning) node will be updated. This is not enough however, a faulty node could be suspended and not notice it is outdated. A solution is to include the last log index client A saw in its communication to client B.

Paxos does not *need* to suffer from this problem but it can. In Paxos reading means asks a *learner* for the value, the *learner* can then ask the majority of the system if they have accepted a value and, if so, what it is. Usually Paxos is optimized by making acceptors inform learners of a change. In this case a leader that missed a message, that there is a value, from an acceptor will incorrectly return to client B there is no value.

## 1.4 File System

A file system is a tool to organise data, the files, using a directory. Data properties, or metadata, such as a file's name, identifier, size, etc are tracked using the directory. Typically the directory entry only contains the file name and its unique identifier. The identifier allows the system to fetch the other metadata. The content of the data is split into blocks which are stored on stable storage such as an hard drive or ssd. The file system defines an application programming interface (API) to operate on it providing methods for *create*, *read*, *write*, *seek* and *truncate* files.

Usually a file system adds a distinction between open and closed files. The APIs: *read*, *write* and *seek* can then be restricted to open files. This enables the system to provide some consistency guarantees. For example allowing a file to be opened only if it was not already open. This can prevent a user from corrupting data by writing concurrently to overlapping ranges in a file. There is no risk to reading from concurrently. Depending on the system reading is even safe while appending concurrently from multiple other processes<sup>8</sup>. Enable such guarantees a file system can define opening a file in read-only, append-only or read-write mode. On Linux these guarantees are optional<sup>9</sup>. More fine grained semantics exist, such as opening multiple non overlapping ranges of a file for writing.

<sup>8</sup>The OS can ensure append writes are serialized, this is useful for writing to a log file where each write call appends an entire log line

<sup>9</sup>see *flock*, *fcntl* or mandatory locking



## 1.5 Distributed file systems

Here I will discuss the two most widely used distributed file systems. We will look at how they work and the implementation. Before I get to that I will use a very basic file sharing system, network file system (NFS), to illustrate why these distributed systems need their complexity.

### Network File System

One way to share files is to expose a filesystems via the network. For this you can use a *shared file system*. These integrate in the file system interface of the client. A widely supported example is network file system (NFS). In NFS a part of a local directory is exported/shared by a local NFS-server. Other machines can connect and overlay part of their directory with the exported one. The NFS protocol forwards file operations from the client to the host. When an operation is applied on the host the result is traced back to the client. To increase performance the client (almost always) caches file blocks and metadata.

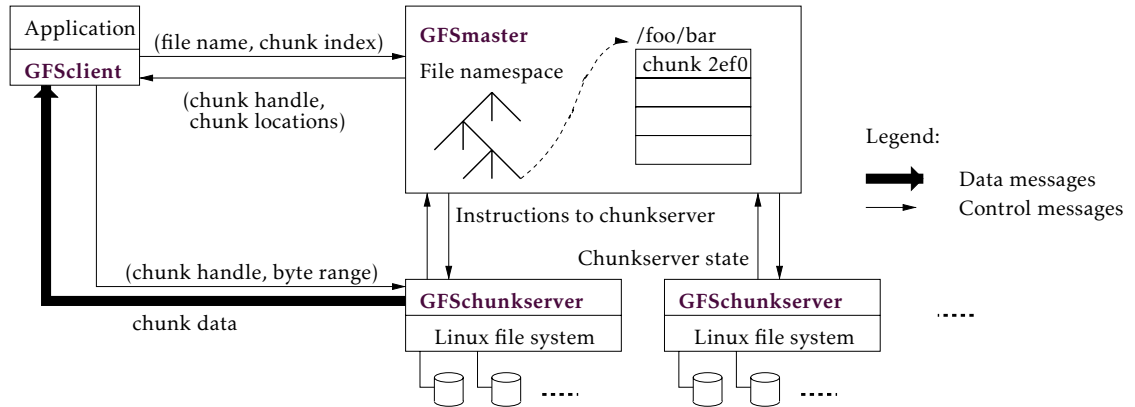
In a shared environment it is common for multiple users to simultaneously access the same files. In NFS this can be problematic. Metadata caching can result in new files appearing up to 30 seconds after they have been created. Furthermore simultaneous writes can become interleaved, writing corrupt data, as each write is turned into multiple network packets [13, p. 527]. NFS version 4 improves the semantics respecting unix advisory file locks [10]. Most applications do not take advisory locks into account concurrent users therefore still risk data corruption.

## Google file system

The Google file system (GFS) [4] was developed in 2003 in a response to Google's rapidly growing search index which generated unusually large files [8]. The key to the system is the separation of the control plane from the data plane. This means that the file data is stored on many *chunk servers* while a single server, the metadata server (MDS)<sup>10</sup>, regulates access to, location of and replication of data. The MDS also manages file properties. Because all decisions are made on a single machine GFS needs no consensus algorithm. A chunk server need not check requests as the MDS has already done so.

When a GFS client wants to operate on a file it contacts the MDS for metadata. The metadata includes information on which chunk servers the file content is located. If the client requests to change the data it also receives which is the primary chunk server. Finally it streams bytes directly to the primary or from the chunk servers. If multiple clients wish to mutate the same file concurrently the primary serializes those requests to some undefined order. See the resulting architecture in fig. 4. When clients mutate multiple chunks of data concurrently and the mutations share one or more chunks the results are undefined. Because the primary chunkserver serializes operations on the chunk level mutations of multiple clients will be interspersed. For example if the concurrent writes of client *A* and *B* translate to mutating chunks  $1_a 2_a 3_a$  for client *A* and  $2_b 3_b$  for client *B*. The primary could pick serialization:  $1_a 2_a 2_b 3_b 3_a$ . The writes of *A* and *B* have now been interspersed with each other. This is a problem when using GFS to collect logs. As a solution GFS offers atomic appends, here the primary picks the offset at which the data is written. By tracking the length of each append the primary assures none of them overlap. The client is returned the offset the primary picked.

<sup>10</sup>Here I use the term used in Ceph for a similar role. GFS refers to this as the master



**Figure 4:** The GFS architecture with the coordinating server, the GFS master, adopted from [4]

To ensure data will not get corrupted by hardware failure the data is checksummed and replicated over multiple servers. The replicas are carefully spread around to cluster to prevent a network switch or power supply failure taking all replicas offline and to ensure equal utilization resources. The MDS re-creates lost chunks as needed. The cluster periodically rebalances chunks between machines filling up newly added servers.

A single machine can efficiently handle all file metadata requests, as long as files are large. If the cluster grows sufficiently large while the files stay small the metadata will no longer fit in the coordinating servers memory. Effectively GFS has a limit on the number of files. This limit became a problem as it was used for services with smaller files. To work around this applications packed smaller files together before submitting the bundle as a single file to GFS [8].

**Hadoop FS** When Hadoop, a framework for distributed data processing, needed a filesystem Apache developed the Hadoop file system (HDFS) [12]. It is based on the GFS architecture, open source and (as of writing) actively worked on. While it kept the file limit it offers improved availability.

The single MDS<sup>11</sup> is a single point of failure in the original GFS design. If it fails the file system will be down and worse if its drive fails all data is lost. To solve this HDFS adds standby nodes that can take the place of the MDS. These share the MDS's data using either shared storage [1] (which only moves the point of failure) or using a cluster of *journal nodes* [2] which use a quorum to maintain internal consensus under faults.

Around 90% of metadata requests are reads [11] in HDFS these are sped up by managing reads from the standby nodes. The MDS shares metadata changes with the journal cluster. The standby nodes update via the journal nodes. They can lag behind the MDS, which breaks consistency. Most notably *read after write*: a client that wrote data tries to read back what it did, the read request is sent to a standby node, it has not yet been updated with the metadata change from the MDS. The standby node answers with the wrong metadata, possibly denying the file exists at all.

HDFS solves this using *coordinated reads*. The MDS increments a counter on every metadata change. The counter is included in the response of the MDS to a write request. Clients performing a *read* include the latest counter they got. A standby node will hold a read request until the node's metadata is up to date with the counter included in the request. In the scenario where two clients communicate via a third channel consistency can be achieved by explicitly requesting up to date metadata. The standby node then checks with

<sup>11</sup>HDFS refers to it as the namenode

the MDS if it is up to date.

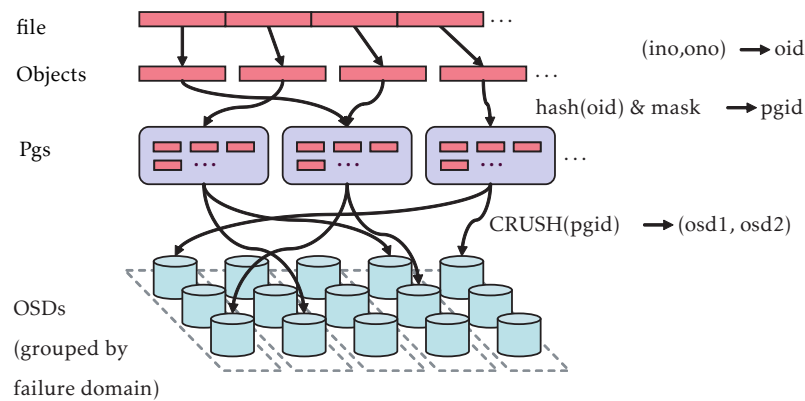
## Ceph

Building a distributed system that scales, that is performance stays the same as capacity increases, is quite the challenge. The GFS architecture is limited by the metadata serving metadata server (MDS). Ceph [14] minimizes central coordination enabling it to scale infinitely. Metadata is stored on multiple MDS instead of a single machine and needs not track data distribution. Instead objects are located using Ceph's defining feature: *controlled, scalable, decentralized placement of replicated data (CRUSH)*, a controllable hash algorithm. Given an *innode number* and map of the object stores Ceph uses the *CRUSH* algorithm to locate where a files data is stored.

To get an *innode number* from a path metadata is retrieved from the *metadata cluster*. This cluster can scales as needed not limiting a Ceph size. Data integrity is also achieved without need for central coordination. Finally Ceph charts the entire system with a growable *monitor cluster*. This is enabled by Ceph's defining feature: the *CRUSH* algorithm which given an innode number outputs a list of nodes on which the data is or should be stored.

**File Mapping** Lets take a closer look at how Ceph uses hashing to map a file to object locations on different servers. The process is illustrated in fig. 5. Similar to GFS files are first split into fixed size objects, each object is assigned an id based on the files innode number. These object ids are hashed into placement groups (PGs) . CRUSH outputs a list of  $n$  object store devices (OSDs) on which an object should be placed given a placement group, cluster map

and replication factor  $n$ . The cluster map not only lists all the OSDs but defines failure domains, such as servers sharing a network switch. CRUSH uses the map to minimize the chance all replicas are taken down by one failure.



**Figure 5:** How Ceph stripes a file to objects and distributes these to different machines

The use of CRUSH reduces the amount of work for the metadata servers. They only need to manage the namespace and need not bother regulating where objects are stored and replicated.

**Capabilities** File consistency is enforced using capabilities. Before a client will do anything with a file's content it requests these from the MDSs (metadata servers). There are four capabilities: *read*, *cache reads*, *write* and *buffer writes*. When a client is done writing it returns the capability together with the new file size. The MDS can revoke capabilities as needed, in the case of write forcing the client to return the new size. For example before issuing the *write* capability for a file the MDS needs to revoke all *cache reads* capabilities for that file. If it did not a client caching reads would 'read' stale

data from its cache while not noticing the file has changed. The MDS also revoke capabilities to provide correct metadata about a file being written to such as when its file size is requested.

***Metadata*** Clients need an inode number and capabilities before they can find and access file data. This is handled by the metadata server cluster. These maintain consistency while resolving paths to inodes, issuing capabilities and providing access to file metadata. Issuing write capabilities for an inode or changing its metadata can only be done by a single node, the inodes authoritative MDS. In the next section we will discuss how inodes are assigned an authoritative MDS. The authoritative MDS additionally maintains cache coherency with other MDS that cache information for the inode. These other MDS are used to issue read capabilities and read the inodes metadata.

To recover from crashes, changes to metadata are journaled to Ceph's object store devices (OSDs). Journalling, appending changes to a log, is faster than updating an on disk state of the system. When a node crashes the MDS cluster reads through the journal to recover the state. Since OSDs are replicated metadata can not easily be lost.

***Subtree partitioning*** If all inodes would share the same authoritative MDS metadata changes and write capability requests would bottleneck Ceph. This would be similar to HDFS with standby nodes. Instead we group inodes based on their position in the file system hierarchy. Each of these groups, each a subtree of the file system, are assigned their own authoritative MDS balancing

the load. The subtrees adapt to the load on the system. The most popular subtrees are split and those responsible for low load are merged.

To determine the popularity each authoritative MDS track the popularity of their subtree by tracking a counter for each innode. The counters decay exponentially with time. They are increased whenever the corresponding innode or one of its descendants is used. Periodically all subtrees are compared migrating part the most popular subtrees to their own authoritative MDS while merging less popular subtrees.

Since servers can crash at any time migration needs to be performed carefully. First the journal on the new MDS is appended to noting a migration is in progress. The metadata to be migrated is now appended to the new MDS's journal. When the transfer is done an extra entry in both the migrated to and migrated from server marks completion and the transfer of authority.



## References

- [1] Apache. *HDFS High Availability*.  
<https://hadoop.apache.org/docs/r3.3.1/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html>. 2021.
- [2] Apache. *HDFS High Availability Using the Quorum Journal Manager*.  
<https://hadoop.apache.org/docs/r3.3.1/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html>. 2021.
- [3] M Caporloni and R Ambrosini. “How closely can a personal computer clock track the UTC timescale via the internet?” In: *European Journal of Physics* 23.4 (June 2002), pp. L17–L21. doi: 10.1088/0143-0807/23/4/103. url: <https://doi.org/10.1088/0143-0807/23/4/103>.
- [4] Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung. “The Google File System”. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP ’03. Bolton Landing, NY, USA: Association for Computing Machinery, 2003, pp. 29–43. isbn: 1581137575. doi: 10.1145/945445.945450. url: <https://doi.org/10.1145/945445.945450>.
- [5] Patrick Hunt et al. “{ZooKeeper}: Wait-free Coordination for Internet-scale Systems”. In: *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. 2010.
- [6] Flavio P Junqueira, Benjamin C Reed and Marco Serafini. “Zab: High-performance broadcast for primary-backup systems”. In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE. 2011, pp. 245–256.

- [7] Leslie Lamport. “Paxos Made Simple”. In: *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001) (Dec. 2001), pp. 51–58. URL: <https://www.microsoft.com/en-us/research/publication/paxos-made-simple/>.
- [8] Marshall Kirk McKusick and Sean Quinlan. “GFS: Evolution on Fast-Forward: A Discussion between Kirk McKusick and Sean Quinlan about the Origin and Evolution of the Google File System”. In: *Queue* 7.7 (Aug. 2009), pp. 10–20. ISSN: 1542-7730. DOI: 10.1145/1594204.1594206. URL: <https://doi.org/10.1145/1594204.1594206>.
- [9] Diego Ongaro and Ousterhout John. *In Search of an Understandable Consensus Algorithm (Extended Version)*. <https://raft.github.io/>. accessed 15-Feb-2022. 2014.
- [10] S Shepler et al. *Network File System (NFS) version 4 Protocol*. RFC 3530. IEFT, Apr. 2003. URL: <https://www.ietf.org/rfc/rfc3530.txt>.
- [11] Konstantin Shvachko et al. *Consistent Reads from Standby Node*. <https://issues.apache.org/jira/browse/HDFS-12943>. Accessed: 03-03-2022. 2018.
- [12] Konstantin Shvachko et al. “The Hadoop Distributed File System”. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 2010, pp. 1–10. DOI: 10.1109/MSST.2010.5496972.
- [13] Abraham Silberschatz, Peter Baer Galvin and Greg Gagne. *Operating system concepts*. John Wiley & Sons, 2014.
- [14] Sage A Weil et al. “Ceph: A scalable, high-performance distributed file system”. In: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pp. 307–320.