2022-05-16

Title to be decided

David Kleingeld

Email

—

Contents

1 Introduction 2

2 Background 2

2.1 Distributed Computing 2

2.2 Faults and Delays 2

2.3 Consensus Algorithms 4

2.4 File System 15

2.5 Distributed file systems 16

3 Design 23

3.1 API and Capabilities 23

3.2 Architecture 24

3.3 Client requests 26

3.4 Availibility 29

1 Introduction

2 Background

TODO write tiney intro, do after introduction has been written

2.1 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. Resiliance 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 frequancy with which one of the parts fails. Even the most robust part will fail and given enough of them the system will fail frequantly. Therefore at very large scales HPC is not even an option.

2.2 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[1]. Event worse a process can be paused and then resumed at any time. Such a pause could be because the process thread is pre-emted, because its virtual machine is paused or because the process was paused and resumed after a while[2].

In a distributed system the computers (*nodes*)that form the system are connected by IP over ethernet. Ethernet gives no guarentee a packet is deliverd 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 allowes an algorithm to assumes 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 recoverd later. Either automatically as it restarts or after maintenance.

[1] One could synchronize the time within a datacenter or provide nodes with more accurate clocks

[2] On linux by sending SigStop then SigCont

2.3 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 *consistantly*. To build such a system we need the parts that make up the system to agree with eachother, the 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 declear it dead, they do not know its threads got paused. A few seconds later the node responds again as if nothing had happend, and unless it checks the system clock, no time has paused 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 declear 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 descision. It will be a strange vote, no node cares about the descision itself. In most implementations a node only checks if it regards the sender as trustwothy or alive and then vote yes. To prove liveliness 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 concensus. It enables us to chosing 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 fullfil only one or two of these roles. Usually, and for the the rest of this explanation each node fullfils 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 *perpare* 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[3]. 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

[3]This could be nodes incrementing the number by the cluster size having initally 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 recieved in response to its prepare request:

1. It recieved 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. Therefor the content of our accept request will be the value from proposal n_p.
2. It recieved only acknowleding 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 or node wants to propose but with our number n.

Each acceptor accepts the request if it did not yet recieve a prepare request numberd greater then n. On accepting a request an acceptor sends a message to all learners[4]. This way the learners learn a new value as soon as its ready.

Lets 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[5]. Acceptors change their value if they recieve a higher numberd 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 numberd accept requests needs a majority

[4] remember every node is a learner

[5] this is not yet inconsistent, Paxos does not guarentee consistency over wether a value has been chosen

response. A majority response will include a node from $m-1$. That node will incude v_a as the accepted value. The value for the accept request then changes to v_a. No accept request with another value then 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 referd 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 wil 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. Appanding 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 reliable 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 recieve 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.

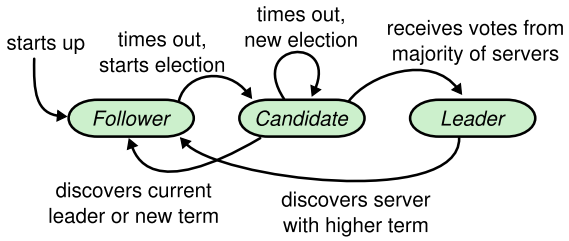

Figure 1: *A Raft node states. Most of the time all nodes exept one are followers. One node is a leader. As failures are detected by time outs the nodes change state. Ajusted from [9].*

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 to long to recieve a majority of the votes a candidate starts a fresh election. When there are multiple candidates requesting votes the vote might split[6], no candidate then reaches a majority. A candidate immidiatly loses the election if it recieves a heartbeat from a *valid* leader. These state changes are illustrated in fig. 1.

In Raft time can be devided 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 reciever knows, it originates from the current leader. A message can only be from the *current* leader if the

[6]Election timeouts are randomised so this does not repeat infinitly.

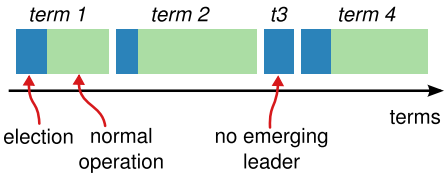

Figure 2: *An example of how time is devided in terms in a Raft cluster. Taken from from [9].*

message *term* is equal or higher then the recieving nodes term. If a node recieves 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 candidates with a *term* equal or higher then that of the majority of the cluster. When recieving a vote request the majority will determine 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 a 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 exampl entry 5 in fig. 3 is committed. The leader includes the index up to which entries are comitted in all its messages. This means entries will become comitted 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 carefull when appending. A follower may have an incomplete log if it did not recieve a

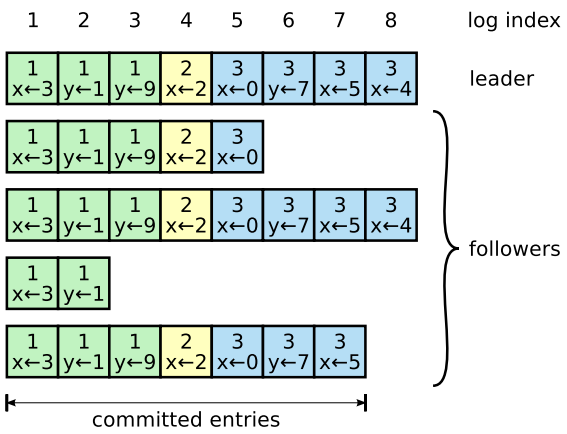

Figure 3: *Logs for multiple nodes. Each row is a different node. The log entries are commands to change shared variables x and y to different values. The shade boxes and the number at their top indictate the term. Taken from [9].*

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[7].

2. When a follower has entries the leader misses these will occupy indices the leader wil 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 wil notice it already has an entry with index k. At that point it simply overwrites what

[7] This is rather inefficient, in the next paragraph we will come back to this

it had at k.

3. A new leader missing commited entries will push a wrong log to followers missing entries. For a 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 then they themself have.

Log compaction Keeping the entire log is rather inefficient. Especially as nodes are added and need to get send the entire log. Usually raft is used to build a state machine, in which case sending over the state is faster then 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 nodes writes the state machine to file together with the current *term* and *index*. It then discards all comitted entries up to the snapshotted *index* from its log.

Followers that lag far behind are 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 build around a log.

Then we need to build the application being carefull 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 themself, *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 guarentees that changes broadcast by the leader are delivered in the order they where send 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. Theses 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:

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

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 consistancy. 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.

2.4 File System

A file system is a tool to organise data, the files, usinga a directory. Data properties, or metadata, such as a files name, identifier, size, etc are tracked using the directory. Typically the directory entry only contains the file name and its unique identifier. The identifier allowes 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 *trunctate* 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 concistancy guarentees. For example allowing a file to be opend 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[8]. Enable such guarentees a file systems can define opening a file in read-only, append-only or read-write mode. On Linux these guarentees are opt in[9]. More fine grained semantics exist, such as opening multiple non overlapping ranges of a file for writing.

[8]The OS can ensure append writes are serialized, this is usefull for writing to a log file where each write call appends an entire log line

[9]see *flock*, *fcntl* or mandatory locking

2.5 Distributed file systems

Here I will discuss the two most widely used destributed fily 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 files 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 envirement it is comman for multiple users to simultaniously access the same files. In NFS this can be problamatic. 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 Googles rapidly growing search index which generated unusually large files [8]. The key to the system is the seperation 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)[10], regulates access to, location of and replication of data. The MDS also manages file properties. Because all descisions 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 recieves which is the primary chunk server. Finally it streams bytes directly to the primary or from the chunk servers. If multiple clients whish 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 result 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 eachoter. 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.

[10]Here I use the term used in Ceph for a similar role. GFS refers to this as the master

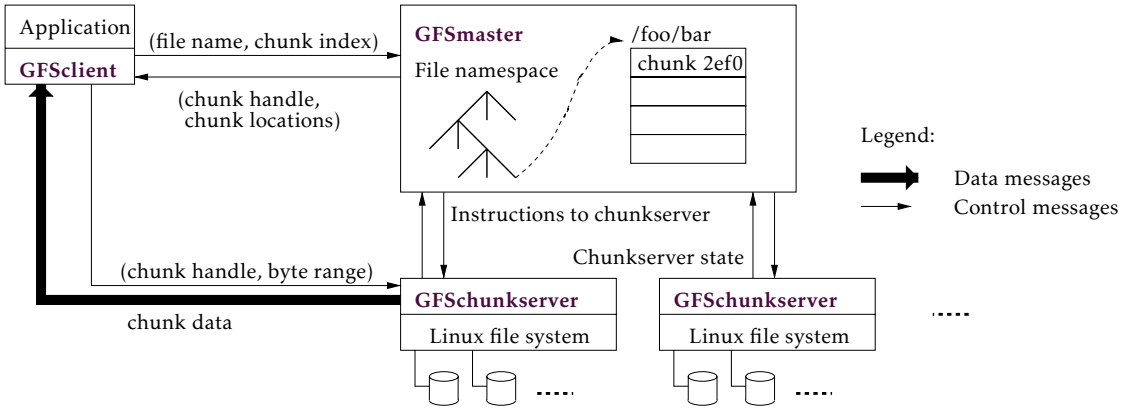

Figure 4: *The GFS architecture with the coordinating server, the GFS master, adopted from [4].*

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 chuncks 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 availibility.

The single MDS[11] 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 reads back what it did, the read request is send to a standby node, it has not yet been updated with the metadata change from the MDS. The standby node awnsers 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 change. The counter is included in the response of the MDS to a write request. Clients performing a *read* include the lastest counter they got. A standby node will hold a read request until the nodes 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 the MDS if it is up to date.

[11] HDFS refers to it as the namenode

Ceph

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

A client resolves a path to an *innode* by retrieving metadata from the MDS cluster. It can scales as needed. Data integrity is achieved without need for central coordination as **obs!**s compare replicas directly.

File Mapping Lets take a closer look at how Ceph uses CRUSH 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 piecesor objects[12] each 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 the OSDs but also defines failure domains, such as servers sharing a network switch. CRUSH uses the map to minimize the chance all replicas are taken down by part of the infrastructure failing.

[12] GFS called these chunks

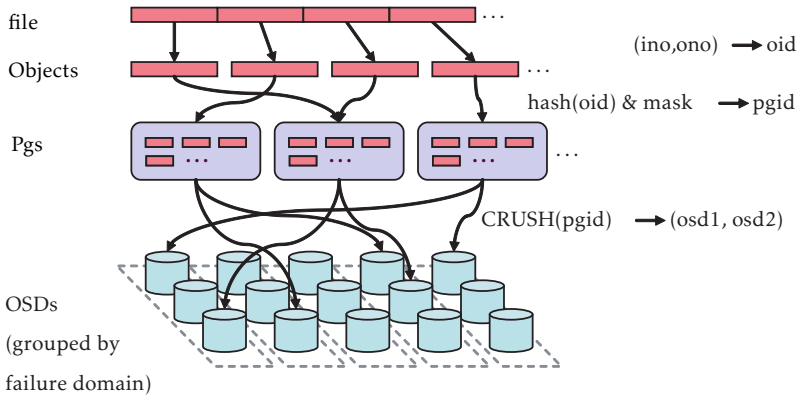

Figure 5: *How Ceph stripes a file to objects and distributes these to different machines.*

The use of CRUSH reduces the amount of work for the MDSs. 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 file content it requests these from a MDSs. There are four capabilities: *read*, *cache reads*, *write* and *buffer writes*. When a client is done it returns the capability together with the new file size. A MDS can revoke capabilities as needed if a client was writing this forces the client to return the new file size if it wrote to the file. Before issuing the *write* capability for a file a MDS needs to revoke all *cache read* capabilities for that file. If it did not a client caching reads would 'read' stale data from its cache not noticing the file has changed. A MDS also revoke capabilities to provide correct metadata for file being written to. This is nessesary as the MDS only learns about the current file upon response of the writer.

Metadata The MDS cluster maintains consistancy while resolving paths to innodes, issuing capabilities and providing access to file metadata. Issuing

write capabilities for an innode or changing its metadata can only be done by a unique MDS, the innodes authoritative MDS. In the next section we will dicuss how innodes are assigned an authoritative MDS. The authoritative MDS additionally maintains cache coherency with other MDSs that cache information for the innode. These other MDSs issue read capabilities and handle metadata reads.

The MDS cluster must be able to recover from crashes. Changes to metadata are therefore journaled to Ceph's object store devices (OSDs). Journalling, appending changes to a log, is faster then updating an on disk state of the system. When a MDS crashes the MDS cluster reads through the journal applying each change to recover the state. Since OSDs are replicated metadata can not realistically be lost.

Subtree partitioning If all innodes shared the same authoritative MDS changing metadata and issuing write capabilities would quickly bottleneck Ceph. Instead innodes are grouped based on their position in the file system hierachy. These groups, each a subtree of the file system, are all assigned their own authoritative MDS. The members of a group, representing a subtree, dynamically adjust to balance load. The most popular subtrees are split and those hardly taking any load are merged.

To determine the popularity of their subtree each authoritative MDS keeps a counter for each of their innodes. The counters decayes exponantially with time. It is are increased whenever the corrosponding innode or one of its decendents is used. Periodically all subtrees are compared to decide which to merge and which to split.

Since servers can crash at any time migrating innodes for splitting and merging needs to be performed carefully. First the journal on the new MDS is

appended, 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.

3 Design

Here I will present my system's design and explain how it enables scalable consistent distributed file storage. First I will discuss the API exposed by my system then I will present the architecture, finally I will detail some of the system behaviour using state machine diagrams.

3.1 API and Capabilities

The filesystem is set up hierarchically: data is stored in files which are kept in folders. Folders can contain other folders. The hierarchy looks like a tree where each level may have 1 to n nodes.

The portable operation system interface (POSIX) has enabled applications to be developed once for all complying filesystem and my system's API is based on it. If we expand beyond POSIX by adding methods that allow for greater performance we excludes existing applications from these gains. This is a tradeoff made by Ceph

section 2.5

by implementing part of the POSIX high performance computing (HPC) IO extensions [15]. Ceph also trades in all consistency for files using the HPC API.

Key to my approach is also expanding the file system API beyond posix. While

this makes my system harder to use it does not come at a cost of reducing consistency. Specifically my system expands upon POSIX with the addition of *open region*. A client that *opens* a file *region* can access only a range of data in the file. This API enables consistent *limited parallel concurrent* writes on, or combinations of reading and writing in the same file.

Like Ceph in my system clients gain capabilities when they open files. These capabilities determine what actions a client may perform on a file. There are four capablities:

- read region
- write region
- read
- write

A client with write capabilities may also read data.

My system tracks the capabilities for a each of its files. It will only give out capabilities that uphold the following:

- Multiple clients can have capabilities on the same file as long as write capabilities have no overlap with any region.

3.2 Architecture

My system uses hierarchical leadership, there is one president elected by all servers. The president in turn appoints multiple authoratative MDSs (aMDSs) then assigns each a group of uninitialized MDSs (uMDSs). An aMDS contacts its group, and promotes each member from unassigned to caching.

The president coordinates the cluster: monitoring the population, assigning new aMDSs on failures, adjusting group given failures and load balances between all groups. Load balancing is done in two ways: increasing the size of

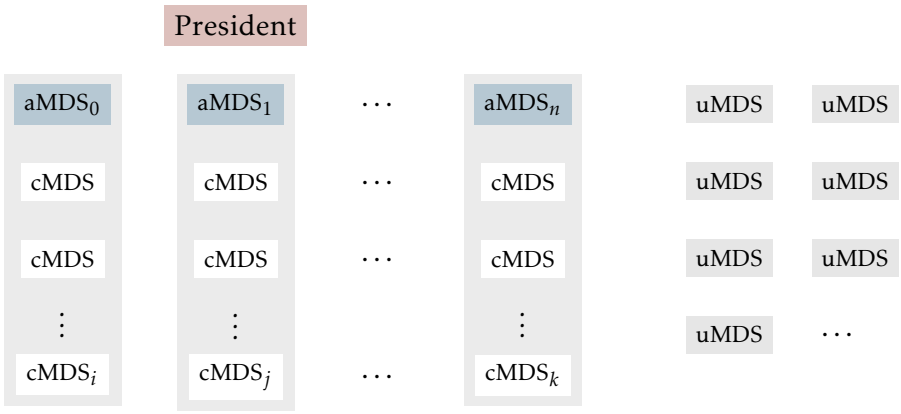

Figure 6: *An overview of the architecture. There is one president, n ministers ■, each ministry can have a different number of clerks □. Not all servers in a cluster are always actively used as represented by the idle nodes ■*

a group and increasing the number of groups using dynamic subtree partitioning [14] (see: section 2.5). To enable the president to make load balancing decisions each aMDS periodically sends file innode popularity.

Metadata changes are coordinated by an aMDS it serializes metadata modifying requests and ensures the changes proliferate to the group's caching MDSs (cMDSs). Changes are only completed when they are written to half of the cMDSs. Additionally the aMDS keeps track of file innode popularity querying its cMDSs periodically.

A groups cMDSs handle metadata queries: issueing read capabilities and providing information about the filesystem tree. Additionally each cMDS tracks file innode popularity to provide to the aMDS.

It is not a good idea to assign as much cMDSs to groups as possible. Each extra cMDS is one more machine the aMDS needs to contact for each change. The

cluster might therefore keep some uMDSs unassigned. I will get back to this when discussing how to add a new subtree in Section 3.4.

Consensus

Consensus communication has a performance penalty and complicates system design. Not all communication is critical to data integrity or system. I use simpler protocols where possible.

The president is elected using Raft. Its coordination is communicated using Raft's log replication. On failure a new president is elected by all metadata servers (MDSs) using Raft.

Communication from the aMDS to its group members uses log replication similar to Raft. When an aMDS is appointed it gets a Raft term. Changes to the metadata are shared with cMDSs using log replication. An aMDS can fail after the change was comitted but before the client was informed of the success. The log index is shared with the client before it is committed, on aMDS failure the client can check with an cMDS to see if its log entry exists[13]. When the aMDS fails the president selects the cMDS with the highest *commit index* as the new aMDS. I call this Presidential Raft (pRaft).

Load reports to the president are send over normal TCP ensuring the arrive in order. An aMDS that froze, was replaced and starts working again however can still send outdated reports. By including the term of the sending aMDS the president detects outdated reports and discards them.

[13]Without the log index the client can not distinquish between failure to apply the change and a new change overriding its own.

3.3 Client requests

Here we go over all the operations of the system while discussing four in greater detail. I also discuss how other operations are simplere forms of these four. This section is split in two parts: client requests and coordination by the president. For all requests a client needs to find a MDS to talk to. If the request modifies the namespace, such as write, create and delete, the client needs to contact the aMDS. In Figure 7 we see how this works. Since load balancing changes and aMDS appointments are communicated through Raft each cluster member knows which MDS group and aMDS owns a given subtree. Because client does not know the current Raft *term* nor *commit* they can not judge if the information is up to date. At worst this means a few extra jumps through irralivent MDSs before they get up to date information.

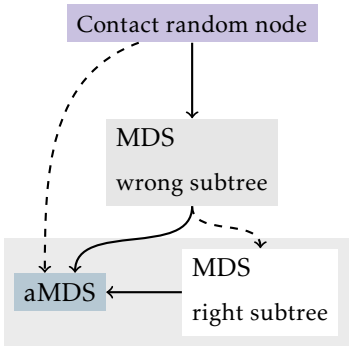

Figure 7: *A new client ■ finding an aMDS ■ for a file. Its route can go through the wrong subtree ■ or via a group member □. Whenever there is a choice dotted lines indicate the less likely path.*

Capabilities

A client needing write capability on a file contacts the aMDS. It checks if the lease can be given out and asks its cMDSs to revoke outstanding read leases

that conflict. A read lease conflict when its region overlaps with the region needed for the write capability. If a cMDS lost contact with a client it can not revoke the lease and has to wait till the lease expires. The process is illustrated in Figure 8.

For read capabilities the requests is send to a cMDS. It checks against the Raft log if the leases would conflict with an outstanding write lease. If no conflict is found the lease is issued and the connection to the client kept active. Keeping an active connection makes it possible to revoke or quickly refresh the lease.

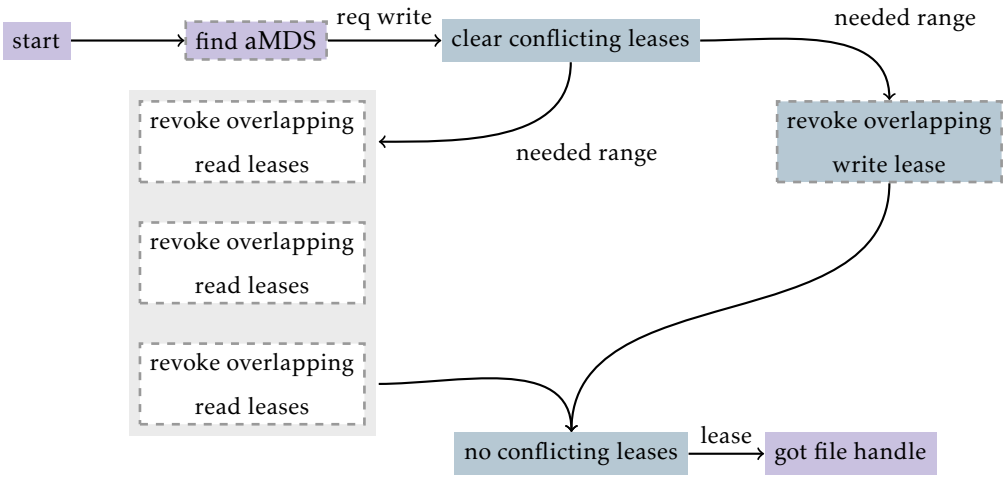

Figure 8: *A client ■ requesting ranged write capabilities. It finds and contacts the aMDS ■. The aMDS then contacts the cMDSs □ to clear conflicting read capabilities. Meanwhile it waits for any conflicting write leases it gave out to expire.*

Namespace Changes

Most changes to the namespace are simpel edits to a MDS groups metadata table. The client sends its request to the aMDS. The change is performed by adding a commands to the pRaft log (see: Section 3.2). Before committing the change the client gets the log index for the change. If the aMDS goes down before acknowleding success the client verifies if the change happend using the log index.

Removing a directory spanning one or more load balanced subtrees needs a little more care. One or more aMDSs will have to delete their entire subtree. This requires coordination across the entire cluster. The clients remove requests is forwarded by the aMDS to the *president*. It in turn appends *Subtree Delete* to the cluster wide log. The client recieves the log index for the command to verify success even if the *president* fails. The steps the aMDS takes are shown in Figure 9.

3.4 Availibility

Ensuring file system avalibility is the responsibility of the *president*. This includes replacing nodes that fail and load balancing. All aMDSs and cMDSs periodically send heartbeats to the president. In this design I improve scalability by moving tasks from a central authority (the president) to groups of nodes. Following this pattern we could let the aMDSs monitor their cMDSs. This is more complex then letting them report directly to the *president*. Even reporting directly to the president is not needed, instead I use the TCP ack to the *president*'s Raft heartbeat. When the *president* fails to send a Raft heartbeat to a node it decides the node must be failing.

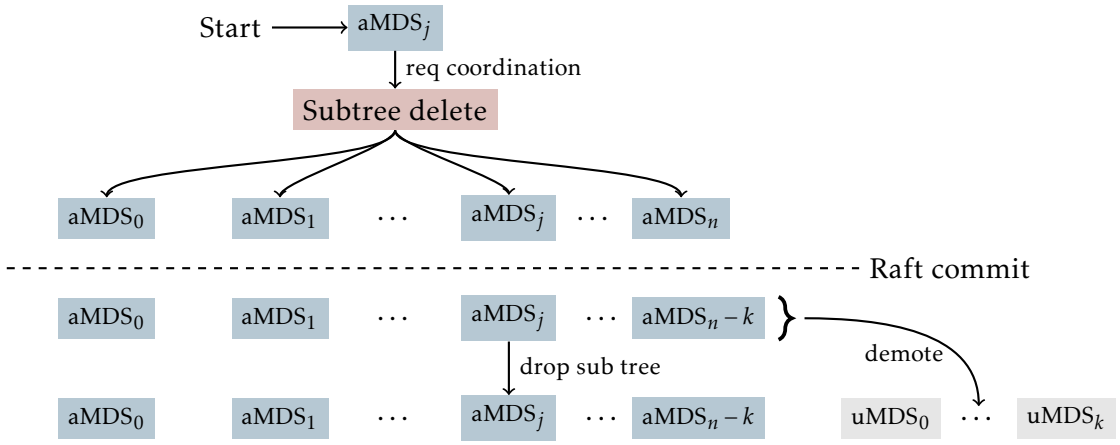

Figure 9: *An aMDS ■, here aMDS$_j$, removes a directory (tree) that is load balanced between multiple groups. The president ■ coordinates the removal by appending a command to the log. Once it is committed the aMDSs hosting subtrees of the directory demote themself to uMDS and MDS group j drops the directory from its db*

A failing MDS

When the president notices an aMDS has failed it will try to replace it. It queries the aMDS's group to find the ideal candidate for promotion. It it gets a response from less then half the group it can not safely replace the aMDS. At that point it marks the file subtree as down and may retry in the future. A succesful replace is illustrated in Figure 10.

A cMDS going down is handled by the president in one of two ways:

- There is at least one uMDS. The president assigns the uMDS to the failing nodes group.
- There are no free uMDS. The president, through raft, commands a cMDS in the group with the lowest load to demote itself. Then the president

drops the matter The cMDS wait till its read leases expire and unassigns.

When the groups aMDS appends to the groups pRaft log it will notice the replacement cMDS is outdated and update its log (see section 2.3).

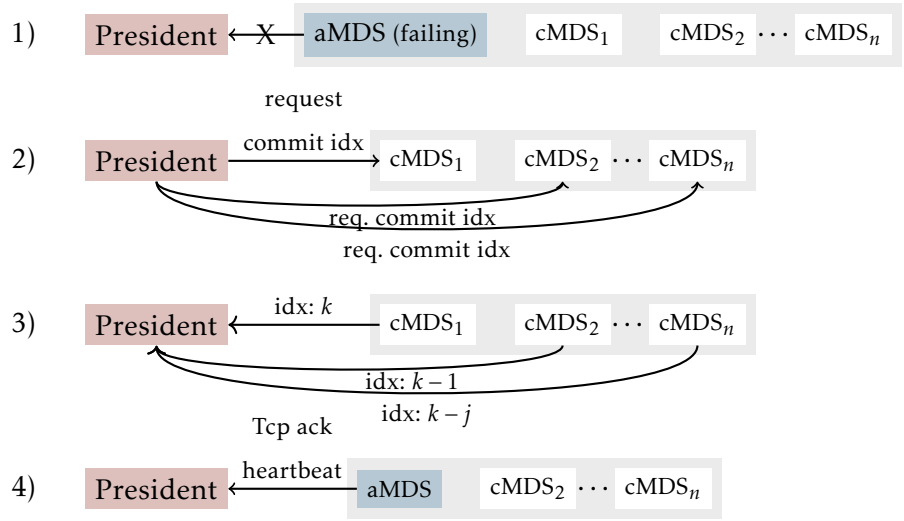

Figure 10: *An aMDS ■ fails and does not send a heartbeat on time (1). The president ■ requests the latest commit index (2). Node $cMDS_1$ □ has index k and it is the highest (3). The president has promoted $cMDS_1$ to aMDS, it has started sending heartbeats (4). Note the heartbeats send by the cMDSs are not shown.*

Load balancing

From the point of avalibility a system that is up but drowning in requests might as well be down. To prevent nodes from getting overloaded we actively balance the load between MDS groups, add and remove groups and expand the read capacity of groups. A load report contains cpu utilization for each node in the group and the popularity of each bit of metadata split into read

and write popularity. The President checks the load balance for each report it recieves.

Tradeoff There is a tradeoff here: a larger group means more nodes for the aMDS to communicate changes to, slowing down writes. As the file system is split into more subtrees the chance increases a client makeing a set of changes will need to contact not one but multiple groups. Further more to create another group we ususally have to shrink existing groups. In reverse growing a group can involve removing one to free MDSs. We can model the read and write capacity of a single group as:

$$r = n_c \tag{1}$$

$$w = 1 - \sigma * n_c \tag{2}$$

Here n_c is the number of cMDS in the group, and σ the communication overhead added by an extra cMDS.

Now we can derive the number of cMDSs needed give a load. Since we want to have some unused capacity δ. I set δ equal to the capacity with the current load subtracted. This gets us the following system of equations:

$$\delta = w - W \tag{3}$$

$$\delta = r - R \tag{4}$$

Now solve for n_c, the number of cMDSs.

$$r - R = w - W \tag{5}$$

$$n_c - R = (1 - \sigma * n_c) - W \tag{6}$$

$$n_c = \frac{R - W + 1}{1 + \sigma} \tag{7}$$

From this I draw two conclusions, any spare capacity for writing is at a cost of spare reading capacity. The number of cMDS roughly the read load.

Read balancing A group expeciencing low read load relative to their capacity are shrunk. A group is under low read load if it could lose a cMDS without average cMDS cpu utilization under 85%. A group has multiple members because not only for performance. The members form a pRaft cluster and ensure metadata is stored redundantly. This only works if a group has at least three members. To shrink a group the President issues a Raft message unassigning a group member.

A group under high relative read load has average cMDS cpu utilization passing 95%. The president then issues a Raft message assigning a uMDS to the group if one is availible.

Write balancing When the president decides a group can no longer handle the write load it will try to split off some work to another group. If no existing group can handle the write work the president will try to create a new group. If that is not possible then the

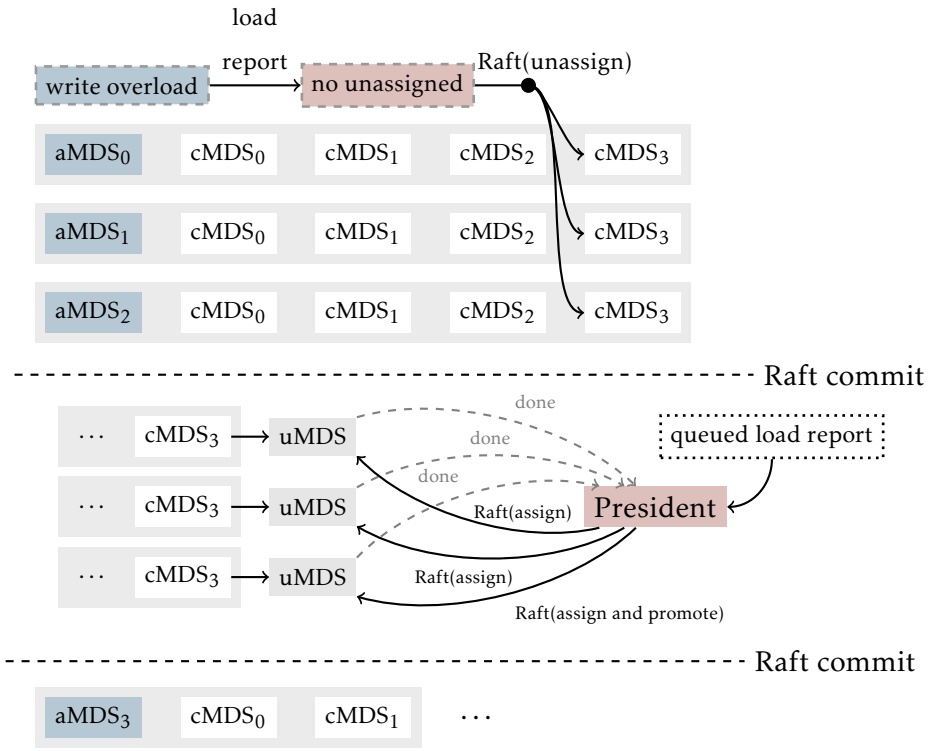

Figure 11: *An aMDS ■ under too high a load, higher then all other, sends a load report to the president ■. It can not create a new MDS group as there are no or not enough uMDSs ■. The president removes three cMDSs □ from the groups under the lightest read load and queues the load report. After the cMDS removal is comitted the load report is unqueued.*

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 Caporaloni 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 conceps*. 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.

[15] Brent B. Welch. “POSIX IO extensions for HPC”. In: 2005.