Universiteit Leiden

Master Computer Science

[GovFS, a scalable control plane using groups of metadata nodes]

Name:	[David Kleingeld]
Student ID:	[s1432982]
Date:	[25/08/2022]
Specialisation:	[Advanced Computing and Systems]
1st supervisor:	[Dr. Alexandru Uta]
2nd supervisor:	[Dr. Kristian Rietveld]

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden2
The Netherlands

2022-08-25

GovFS, a scalable control plane using groups of metadata nodes

David Kleingeld

Email opensource@davidsk.dev

Abstract With the rise of big data and the move to the cloud the need for highly available and scaling file systems has grown. Distributed file systems address this need. A popular approach is to use a dedicated metadata node. This has two problems: the node becomes a single point of failure, and it limits the metadata capacity of the system. Here we design and implement a system using groups of metadata nodes: *GovFS*. We build it on top of *Group Raft* our extension of Raft that enables scaling partitioned consensus. Experiments demonstrate *GovFS* write performance scales linearly with the number of nodes, showing that the system offers good scaling characteristics. Our initial prototype shows promising performance that can be significantly improved in the future.

Acknowledgement

This endeavor would not have been possible without the frequent advise, guidance and honest feedback of my supervisor: *dr. Alexandru Uta*. I would also like to express my thanks to *dr. Kristian Rietveld* for showing me how to test the limited implementation in a meaningful way.

Contents

1 Introduction

In the last decade we have seen the rise of big data and the shift of everyday work to the cloud. Distributed processing of large datasets has been made accesible thanks to programming models such as MapReduce [7]. Distributed computing frameworks such as Apache Hadoop and Spark [26] allow computations to scale across thousands of machines. In conjunction with this rose a need for a highly available and scaling file system. This need is addressed by distributed file systems, which spread files across multiple machines and keep those replicas consistent.

Files can get corrupted if multiple users write to them at the same time. Systems therefore not only store file data but also regulate access to the files. These tasks are normally done by two separate parts of the system: the data and control plane. The control plane is the bottleneck in file system performance. For this reason many systems allow client to opt out of access control and directly operate on files. This can be useful in High Performance Computing (HPC) where many processes need to operate on the same files and can do so in a consistent manner.

There are currently two approaches to distributed file systems: using a dedicated node to control where each file is placed[1], or a distribution function that decides where a file should be located[2]. The distribution function itself needs only limited shared state however the system still need a separate cluster to regulate file access.

Using a dedicated node has two problems. The node is single point of failure, and it can not scale out. This limits the amount of files a system can store and the maximum workload the system can handle. Recently, implementations have started to address the first issue. *HDFS* for example has added standby nodes [1, 2] which can replace the metadata node. However, ensuring consistency between the standby nodes comes at the cost of decreased performance.

In this thesis we investigate whether we can make the control plane scalable using groups of nodes. We made *Group Raft* an extension on the Raft [17] consistency algorithm which enables highly scalable consensus when using

[1] Introduced by GoogleFs [9, 15] and adopted by the widely used Hadoop file system (HDFS) [20].

[2] Pioneered by *Ceph*[24]

groups of nodes that do not share information. We use *Group Raft* to design and implement GovFS: a file system control plane that uses groups of nodes.

When a hashing function is used to locate files the metadata for these files is stored in the data-plane. It is also located through the hash function. A small cluster regulates access to the files. This solves the scaling problem. There are two reasons why we feel it is important to explore other approaches:

- Using hashing we lose some control over file location, this makes co-locating compute and storage more difficult.
- As the control plane usually forms the bottleneck of the system we might want to optimize it using special hardware [4]. When metadata is stored in the data-plane it is hard to do so.

In the next section we will go over the challenges in distributed systems, how consensus algorithms solve those, what a file system is and finally discuss the most widely used distributed file systems: *HDFS* and *Ceph*. Then in Section 3 we present GovFS's design and explain how it enables scalable distributed storage and take a more detailed look at how we extended Raft. In Section 4 we go over the implementation and discuss how reading and writing is coordinated. Using the GovFS implementation we performed a number of performance tests. We present the test methodology and show the results in Section 5. We then discuss the results in Section 6. Finally, in Section 7 we conclude whether GovFS approach, using groups of metadata nodes, offers better scalability. We finish by listing a number of interesting avenues for further study in Section 8.

2 Background

Here we discuss what distributed computing is and when we use it. How faults and delays make it challenging to build simultaneously consistent and highly available systems. Then we look at consensus algorithms that solve those challenges. After this we discuss what a file system is before we go over the most widely used distributed file systems: HDFS and Ceph. Finally, we look at work related to this thesis.

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 Computing (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 as 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.

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, this is not the only issue to keep in mind. We can not determine whether a system is working and responsive if we can not agree on how much time has passed since it last responded.

It is entirely normal for the clock of a computer to run slightly too fast or too slow. The resulting drift will normally be tens of milliseconds [5] unless special measures are taken[3]. Even worse, a process can be paused and then

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

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[4].

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 a node can be recovered later. Recovery can happen either automatically as the node restarts or manual intervention.

2.3 Consensus Algorithms

In this world where the network can not be relied upon, time lies to us and servers will randomly crash and burn how can we get anything done at all? Let's discuss how we can build a system we can rely on, a system that behaves *consistently*. To build such a system we need the parts that make up the system to agree with each other, the must-have: *Consensus*. Here we 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*.

clocks.

[4] On Linux by sending SIGSTOP then SIGCONT

Quorums

Imagine a node hard at work processing requests from its siblings, suddenly it stops. 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 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 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 reliable 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 [14] uses a quorum to provide consensus. It enables us to choose 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 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.

Let us 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 than the highest number our node has seen up till now. The number is unique to our node[5]. Each acceptor only

[5] This can be done by having each node incrementing their number by the cluster size having

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 response.

In phase two our node checks if it got a response from the majority. Our node is going to send an accept request back to those nodes that responded. 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. Therefor 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 numbered greater than n. On accepting a request an acceptor sends a message to all learners[6]. This way the learners learn a new value as soon as it's ready.

To get a feeling why this works we look 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[7]. 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-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.

initially assigned a number 0 till *cluster size* to each node.

[6] Remember every node is a learner in this example.

[7] This is not inconsistent, Paxos does not guarantee consistency over whether a value has been chosen.

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 read 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 [17] cluster starts without a leader and when it has a leader that leader 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 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.

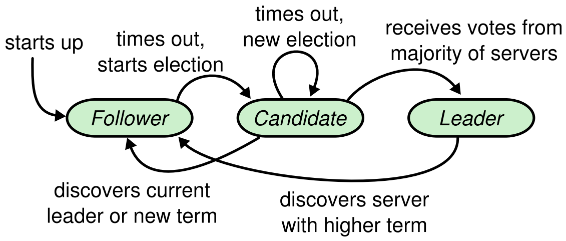

Figure 1: *A Raft node states. Most of the time all nodes except one are followers. One node is a leader. As failures are detected by time-outs the nodes change state. Adjusted from [17].*

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[8], 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 Figure 1.

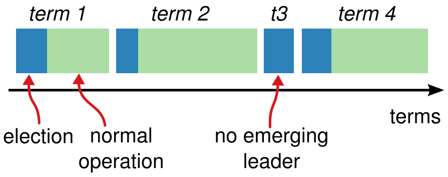

Figure 2: *An example of how time is divided in terms in a Raft cluster. Taken from [17].*

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 Figure 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 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 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-request. For example entry 5 in Figure 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 heartbeat timeout and no entry

[8] Election timeouts are randomized, therefore this does not repeat infinitely.

needs to be added it sends an empty append. There is no need for a special heartbeat message.

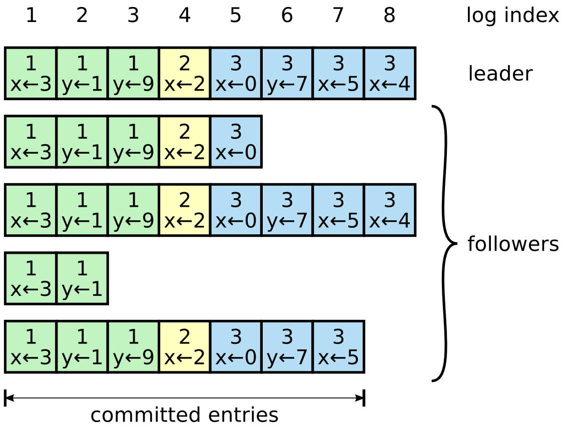
1 2 3 4 5 6 7 8
log index
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0 | 3 y←7 | 3 x←5 | 3 x←4
leader
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0 | 3 y←7 | 3 x←5 | 3 x←4
1 x←3 | 1 y←1
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0 | 3 y←7 | 3 x←5
followers
committed entries

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 [17].*

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 previous append (1), it may have messages the leader does not (2), and finally we must prevent a candidate missing committed entries from becoming the leader (3).

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 every append requests. If a follower's 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[9].
2. When a follower has entries the leader misses these will occupy indices the leader needs to 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 such a follower will notice it already has an entry with index k. At that point it simply overwrites what 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

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

> 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 therefore do not vote for a candidate with a lower log index then they themselves 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 than the log. The state machine 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 snapshotted *index* from its log.

Followers that lag far behind are now sent 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 while being careful to implement the consensus algorithm correctly. A popular alternative is to use a coordination service instead. Here we look at ZooKeeper [12] an example of a wait free coordination service. We first focus on the implementation, drawing parallels to Raft before we look at the application programming interface (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 [13]. 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:

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 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 connected to is lagging behind. Client B can avoid this by calling *sync*, this will make the ZooKeeper node process 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 asking 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 organize data, the files, using a directory. Data properties, or metadata, such as a files name, identifier, size, etc. are tracked using the directory interface. 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 a hard drive or SSD. The file system defines a API to operate on it, providing methods to *create*, *read*, *write*, *seek* and *truncate* files.

A file system can add 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 files concurrently. Depending on the system reading is even safe while appending in parallel from multiple other processes[10]. To 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 opt-in[11]. More fine-grained semantics exist, such as opening multiple non overlapping ranges of a file for writing.

2.5 Distributed file systems

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

[10] 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.

[11] See *flock*, *fcntl* or mandatory locking

Network File System

One way to share files is to expose a file-system 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 as each write is turned into multiple network packets [21, p. 527] writing corrupt data. NFS version 4 improves NFS semantics by respecting UNIX advisory file locks [18]. Most applications do not take advisory locks into account therefore concurrent use still risks data corruption.

Google file system

The Google file system (GFS) [9] was developed in 2003 in a response to Google's rapidly growing search index which generated unusually large files [15]. 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)[12], 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 on which chunk servers the file content are 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 want to mutate the same file concurrently the primary serializes those requests to some undefined order. See the resulting architecture in Figure 4. When clients mutate multiple chunks of

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

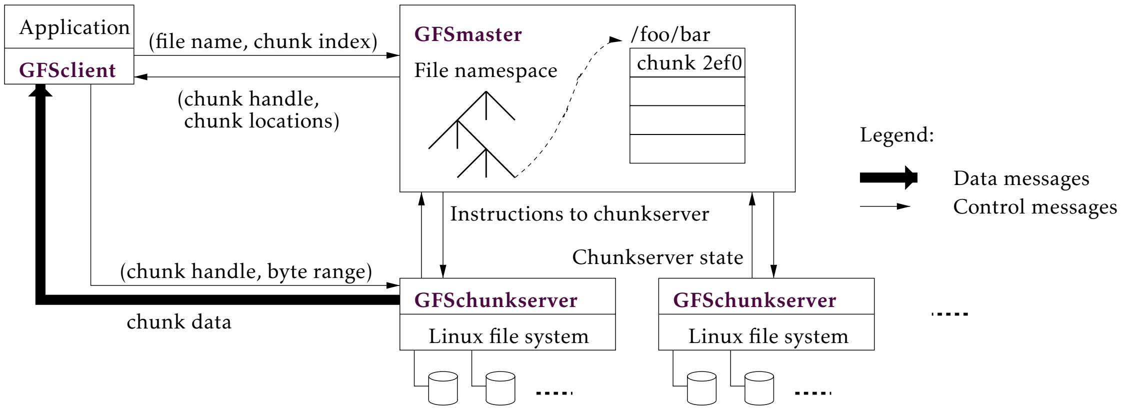

Figure 4: *The GFS architecture with the coordinating server, the GFS master, adopted from [9].*

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 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.

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 GFS was used for services with smaller files. To work around this these services packed smaller files together before submitting the bundle as a single file to GFS [15].

Hadoop FS When Hadoop, a framework for distributed data processing, needed a file system Apache developed the Hadoop file system (HDFS) [20]. It is based on the GFS architecture, open source and (as of writing) actively worked on. While it kept the file limit issue of GFS it offers improved availability.

The single MDS[13] 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 in the cluster 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 [19] 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 maintained by explicitly requesting up-to-date metadata. The standby node then checks with 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 server (MDS). Ceph [24] minimizes central coordination enabling it to scale near infinitely. Metadata is stored on multiple MDS instead

[13] HDFS refers to it as the namenode.

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 controllable hash algorithm. Given an *inode* number and map of the object store devices (OSDs) Ceph uses *CRUSH* to locate where a files data is or should be stored.

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

File Mapping We take a closer look at how Ceph uses CRUSH to map a file to object locations on different servers. The process is illustrated in Figure 5. Similar to GFS files are first split into fixed size pieces or objects[14] each is assigned an ID based on the files *inode* number. These object IDs are hashed into placement groups (PGs). CRUSH outputs a list of n 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.

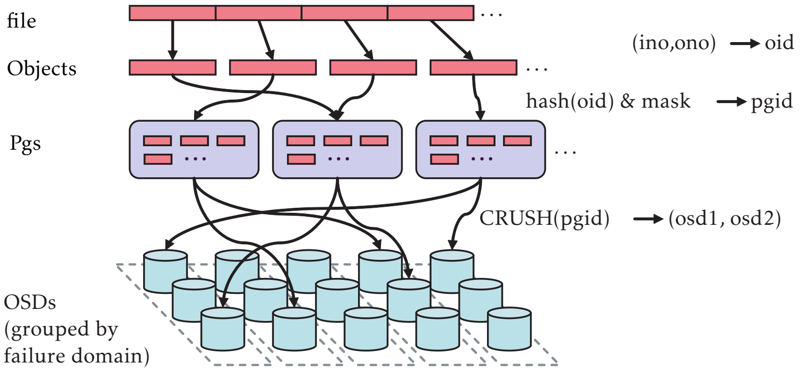

Figure 5: *How Ceph stripes a file to objects and distributes these to different machines. Adapted from [24].*

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.

[14] GFS called these chunks.

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. Before issuing *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. MDSs also revoke capabilities to provide correct metadata for file being written to. This is necessary as the MDS only learns about the current file upon response of the writer.

Metadata The MDS cluster maintains 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 unique MDS, 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 MDSs that cache information for the inode. 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). Journaling, appending changes to a log, is faster than 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 inodes shared the same authoritative MDS changing metadata and issuing write capabilities would quickly bottleneck Ceph. Instead, inodes are grouped based on their position in the file system hierarchy. 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 inodes. The counters decay exponentially with time. A counter is increased whenever the corresponding inode or one of its decedents is used. Periodically all subtrees are compared to decide which to merge and which to split.

Since servers can crash at any time migrating inodes 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.

2.6 Related work

One of the key contributions of this thesis is *Group Raft*. We use it together with subtree partitioning to scale up the control plane without the overhead that would come with scaling up a normal *Raft* cluster. To the best of our knowledge there is no work using Raft to scale up the control plane of a distributed FS. There is literature on using multiple raft groups and there is work on using Raft within a distributed FS.

Ceph is a distributed file system with excellent scalability. It uses a cluster to provide metadata access to clients. Ceph uses another cluster, the monitors, to manage the metadata cluster. Based on the source code[15] and discussions on the Ceph mailing list[16] the monitors seem to coordinate using a modified version of *Paxos*. Each node in the metadata cluster uses a journal to store its state. The journal is replicated in the data plane and used to recover the file system state when a metadata node crashes. In GovFS we store the metadata in multiple dedicated groups instead of a journal in the data plane.

TiDB is a Raft based Hybrid Transactional and Analytical Processing database [11]. It uses multiple independent Raft groups to maintain consistency among replicas. To speed up reading a learner role is introduced. The learner does not use log replication nor does it vote in elections. The Raft groups leader pushes updates to the learner. When reading strong consistency is enforced between the leader and learner. In our *Group Raft* we speed up reading by adding a Raft layer on top that takes care of heartbeats and elections.

PolarFS provides a distributed file layer for cloud databases [4]. It uses Raft to replicate file system chunks to three replicas. The replicas are only for redundancy and are normally not read from. Since modifications to different file system chunks do not affect each other they do not need to be serialized.

[15] https://github.com/ceph/ceph/blob/main/src/mon/MDSMonitor.cc

[16] https://ceph-devel.vger.kernel.narkive.com/hkhSNebO/paxos-vs-raft

PolarFS introduces *ParallelRaft*[17] it allows holes in the log making out-of-order replication possible. It makes it possible for followers to handle new log entries in parallel as they no longer need to ensure the previous entry is committed. The leader ensures only those log entries that modify different FS chunks are appended in parallel.

Clusterd Raft is presented and proven safe in *A hierarchical model for fast distributed consensus in dynamic networks* [6]. It is designed to enhance throughput in Geo-distributed context. Each location runs a local Raft cluster, the leader of a location then replicates in batches to the other clusters. In contradiction to GovFSs *Group Raft* their system still provides a global shared Raft Log. They report a 5x performance improvement in a globally distributed system when compared to Raft. We suspect keeping the log global comes at the cost of significant latency compared to *Group Raft*.

[17] ParallelRaft has been formally verified [10].

3 Design

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

3.1 API and Capabilities

The file system 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 file system and our system's API is based on it. If we expand beyond POSIX by adding methods that allow for greater performance we exclude existing applications from these gains. This is a trade-off made by Ceph (Section 2.5) by implementing part of the POSIX High Performance Computing (HPC) IO extensions [25]. Ceph also trades in all consistency for files using the HPC API.

We also expand the file system API beyond POSIX. While this makes our system harder to use it does not come at a cost of reducing consistency. Specifically GovFS 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 GovFS clients gain capabilities when they open files. These capabilities determine what actions a client may perform on a file. There are four capabilities:

- read
- read region
- write
- write region

A client with write capabilities may also read data. Similar to Ceph capabilities are only given for a limited time, this means a client is given a lease to certain capabilities. The lease needs to be renewed if the client is not done with its file operations in time. The lease can also be revoked early by the system.

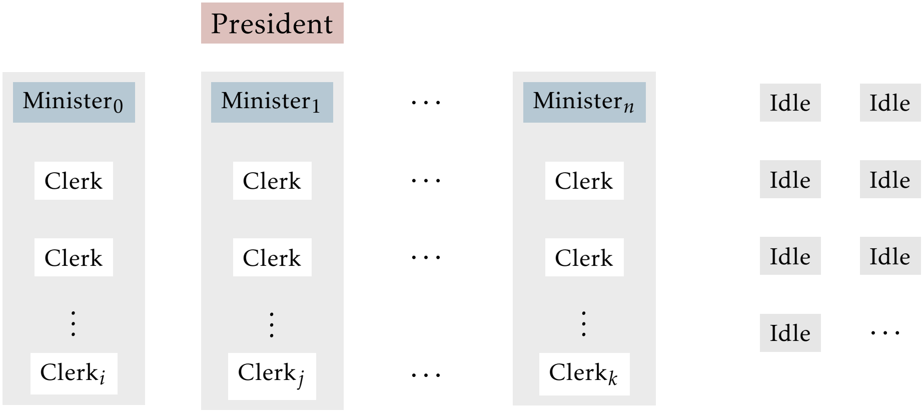

Figure 6: *An overview of the architecture. There is one president, n ministers ■, each ministry can have a different number of clerks □. Not all servers in a cluster are always actively used as represented by the idle nodes ■*

GovFS tracks the capabilities for 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

The overhead of maintaining consensus within a group increases linearly with the number of members. GovFS uses hierarchical leadership to decouple the overhead from the clusters size for interactions that only need consensus with a subset of the cluster. In GovFS there is one president elected by all servers. The president in turn appoints multiple ministers then assigns each a group of clerks. A minister contacts its group, and promotes each member from idle to clerk, forming a ministry.

The president coordinates the cluster: monitoring the population, assigning new ministers on failures, adjusting groups given failures and load balances between all groups. Load balancing is done in two ways: increasing the size of a group and increasing the number of groups. To enable the president to make load balancing decisions each minister periodically sends file popularity.

Metadata changes are coordinated by ministers, they serialize metadata modifying requests and ensures the changes proliferate to the ministry's clerks. Changes are only completed when they are written to half of the clerks. Each minister also collects file popularity by querying its clerks periodically. Finally, write capabilities can only be issued by ministers.

A ministry's clerks handle metadata queries: issuing read capabilities and providing information about the file system tree. Additionally, each clerk tracks file popularity to provide to the minister.

It is not a good idea to assign as many clerks to ministries as possible. Each extra clerk is one more machine the minister needs to contact for each change. The cluster might therefore keep some nodes idle. We will get back to this when discussing load balancing in Section 3.4.

Consensus

Consistent communication has a performance penalty and complicates system design. Not all communication is critical to system or data integrity. We 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 nodes.

Communication from the minister to its ministries clerks uses log replication similar to Raft. When a minister is appointed it gets a Raft term. Changes to the metadata are shared with clerks using log replication. A minister can fail after the change was committed but before the client was informed of the success. The log index is shared with the client before it is committed, on minister failure the client can check with a clerk to see if its log entry exists[18]. When the minister fails the president selects the clerk with the highest *commit index* as the new minister. We call this extension of Raft *Group Raft*.

Load reports to the president are sent over TCP, which will almost always ensure the reports arrive in order. A minister that froze, was replaced and starts working again can still send outdated reports. By including the term of the sending minister the president detects outdated reports and discards them.

Group Raft

In *Group Raft* a third party instructs a node to become the leader. Followers are not informed directly but rather accept the new leader as it has a higher

[18] Without the log index the client can not distinguish between failure and a successful change followed by a new change overriding the clients.

term. When receiving a message the normal Raft rules apply, therefore messages from the old leader[19] are rejected as their term is too low.

In GovFS nodes must be able to move between groups with assigned leaders. This present two problems, the first issue becomes apparent looking at the following series of events:

- Leader B receives message that follower x is assigned to it
- Leader B appends to its log and sends an append request to its followers now including x
- Follower x *accepts* the request as x has a *lower* term than the request
- Follower x *increases* its term to match B
- Leader A receives a message that assigns x back to it
- Leader A appends to its log and sends an append request to its followers which now again includes x
- Follower x *rejects* the request as x now has a *higher* term then the request

Leader A must initially have a lower term then B and than a higher term than B for the re-assignment to work. The second problem: we need a follower to re-write its log after every move to match that of its new group.

To solve the first problem we make the third party change the term of the leader when assigning it a node. Every re-assignment will succeed if it guarantees the new term is the highest of all the groups. It is trivial to ensure that by using a single third party. This third party can atomically increment the term every assignment and re-assignment.

This coincidentally also solves our second problem. Since the highest number is always unique, the correct log for each group now has an increasing sequence of unique terms. If a node receives an append request and its previous log entries, term and index do not match that of the leader it rejects the request. The leader then starts sending older log entries[20]. A successful append can only happen on correct (partial) group log given terms are now unique to groups.

[19] The third party replacing the leader usually indicates there is a problem with the current leader, making it doubly important that messages from old leaders are ignored.

[20] This is optimized in GovFS by clearing the entire log if a clerk came from another ministry.

3.3 Client requests

In this section we go over all the operations of the system, discussing four of them in greater detail. We also explain how most operations are simpler 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 an official (a minister or clerk) to talk to. If the request modifies the namespace, such as write, create and delete, the client needs to contact the minister. In Figure 7 we see how this works. Since load balancing changes and minister appointments are communicated through Raft each cluster member knows which ministry owns a given subtree. A client can not judge whether the node it got directions from has up to date and correct information. In the rare case the directions are incorrect. This means an extra jump through an irrelevant ministry before getting correct information.

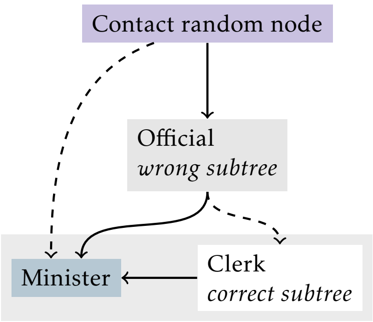

Figure 7: *A new client* ■ *finding the responsible minister* ■ *for a file. Its route can go through the wrong subtree* ■ *or via the correct ministry's clerk* □. *Whenever there is a choice the dotted line indicate the less likely path.*

Capabilities

A client needing write-capability on a file contacts the minister. It in turn checks if the lease can be given out and asks its clerks to revoke outstanding read leases that conflict. A read lease conflicts when its region overlaps with the region needed for the write capability. If a clerk 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.

If the client needs read capabilities it sends its requests to a clerk. The clerk 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 the lease or quickly refresh it.

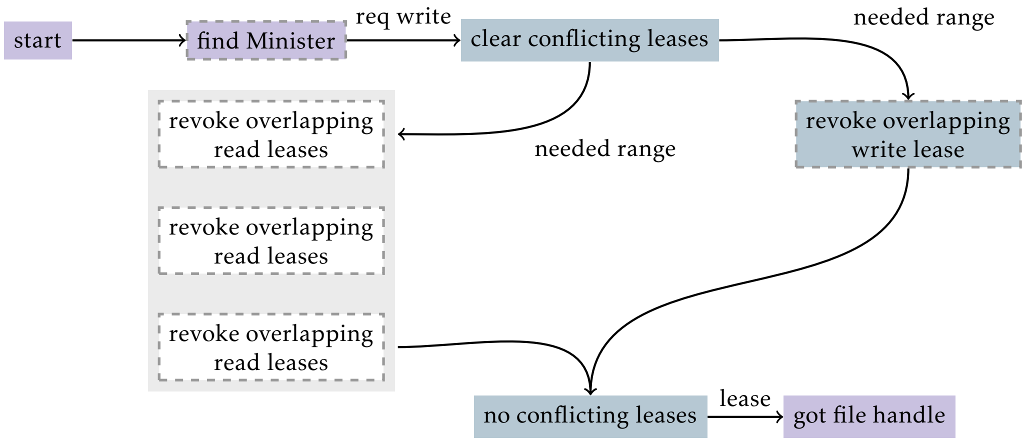
start
find Minister
req write
clear conflicting leases
needed range
revoke overlapping read leases
needed range
revoke overlapping write lease
revoke overlapping read leases
revoke overlapping read leases
no conflicting leases
lease
got file handle

Figure 8: *A client ■ requesting ranged write capabilities. It finds and contacts the responsible minister ■. The minister then contacts the ministry's clerks □ to clear conflicting read capabilities. Meanwhile, it revokes any conflicting write leases it gave out.*

Namespace Changes

Most changes to the namespace need simple edits within ministries metadata table. The client sends its request to the minister. The change is performed by adding a command to the *Group Raft* log (see: Section 3.2). Before committing the change the client gets the log index for the change. If the minister goes down before acknowledging success the client verifies if the change happened using the log index.

Removing a directory spanning one or more load balanced subtrees needs a little more care. One or more ministers will have to delete their entire subtree. This requires coordination across the entire cluster. The client's remove request is forwarded by the minster to the *president*. It in turn appends *Subtree Delete* to the cluster wide log. The client receives the log index for the command to verify success even if the *president* goes down. The steps the minister takes are shown in Figure 9.

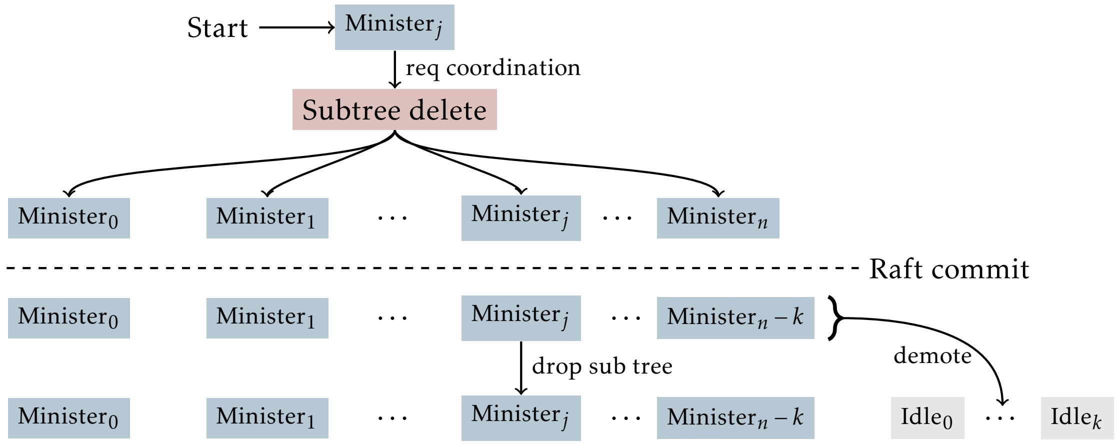

Figure 9: *A minister ■, here Minister$_j$, removes a directory (tree) that is load balanced between multiple ministries. The president ■ coordinates the removal by appending a command to the log. Once it is committed the ministers hosting subtrees of the directory demote themselves to idle and Ministry$_j$ drops the directory from its database*

3.4 Availability

Ensuring file system availability is the responsibility of the *president*. This includes replacing nodes that fail and load balancing. To detect nodes that go down we use the TCP ACK of the nodes 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.

A failing minister

When the president notices a minister has failed it will try to replace it. It queries the ministries clerks to find the ideal candidate for promotion. If it gets a response from less than half the ministry it can not safely replace the minister. At that point it marks the file subtree as down and may retry in the future. A successful replacement is illustrated in Figure 10.

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

- There is at least one idle node. The president assigns the idle node to the failing nodes group.

- There are no idle nodes. The president, through raft, commands a clerk in the group with the lowest load to demote itself. Then the president drops the matter. The clerk wait till its read leases expire and unassigns.

When the groups' minister appends to the *Group Raft* log it will notice the clerk misses entries and update it (see Section 2.3).

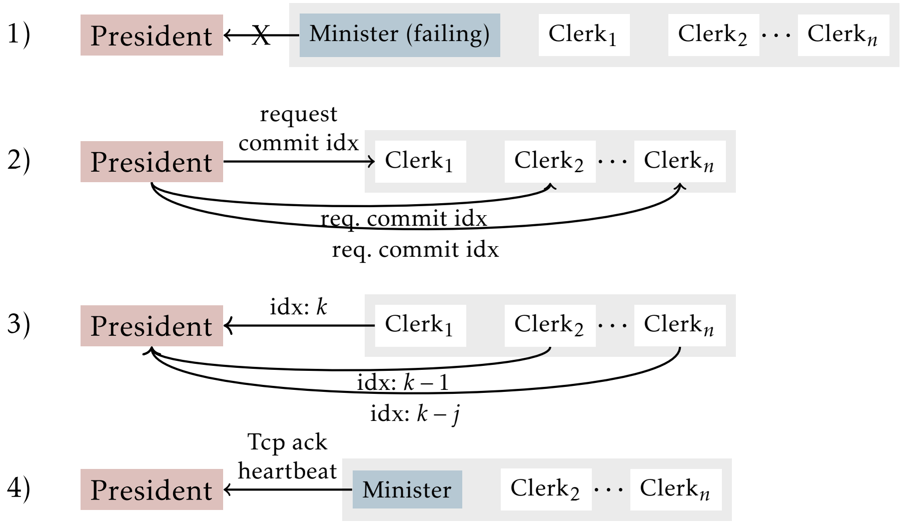

Figure 10: *A minister ■ fails and does not send a heartbeat on time (1). The president ■ requests the latest commit index (2). Node Clerk$_1$ □ has commit index k which is the highest (3). The president has promoted Clerk$_1$ to minister, it has started sending heartbeats (4). Note the heartbeats sent by the clerks are not shown.*

Load balancing

From the point of availability 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 ministries, add and remove them and expand the read capacity of some. 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. The President checks the balance for each report it receives.

Trade off There is a trade-off here: a larger ministry means more clerks for the minister to communicate changes to, slowing down writes. On the other hand as the file system is split into more subtrees the chance increases a client will need to contact not one but multiple ministries. Furthermore, to create another ministry we might have to shrink existing ministries. Growing a ministry can even involve removing another to free up nodes. We can model the read and write capacity of a single group as:

$$
\begin{aligned}
r &= n_{\mathrm{c}} && (1)\\
w &= 1 - \sigma * n_{\mathrm{c}} && (2)
\end{aligned}
$$

Here n_{c} is the number of clerks in the group, and σ the communication overhead added by each clerk.

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

$$
\begin{aligned}
\delta &= w - W && (3)\\
\delta &= r - R && (4)
\end{aligned}
$$

Now solve for n_{c}, the number of clerks.

$$
\begin{aligned}
r - R &= w - W && (5)\\
n_c - R &= (1 - \sigma * n_c) - W && (6)\\
n_c &= \frac{R - W + 1}{1 + \sigma} && (7)
\end{aligned}
$$

From this we draw two conclusions. Any spare capacity for writing is at a cost of spare reading capacity. The number of clerks roughly equals the read load.

Read balancing A ministry experiencing low read load relative to their capacity is shrunk. A ministries read load is low if it could lose a clerk without average clerk CPU utilization rising above 85%. A ministry has multiple members not only for performance. The members form a *Group Raft* cluster and ensure metadata is stored redundantly. This only works if a ministry has at least three members. Therefore, groups can not be made smaller than that. To shrink a ministry the President issues a Raft message unassigning a clerk.

A group under high relative read load has average clerk CPU utilization passing 95%. The president then issues a Raft message assigning an idle node to the group if one is available.

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 work the president will try to create a new group. This is illustrated in Figure 11.

Constraints If the system is experiencing a high read and write load, and can no longer add nodes the greedy approach described above will no longer work. At this point we will need to carefully balance the number of ministries and their sizes using the available nodes. Using the equations derived above we can define an objective function as function of the load reports. Then we can use a constrained optimization method to find the optimal configuration for GovFS.

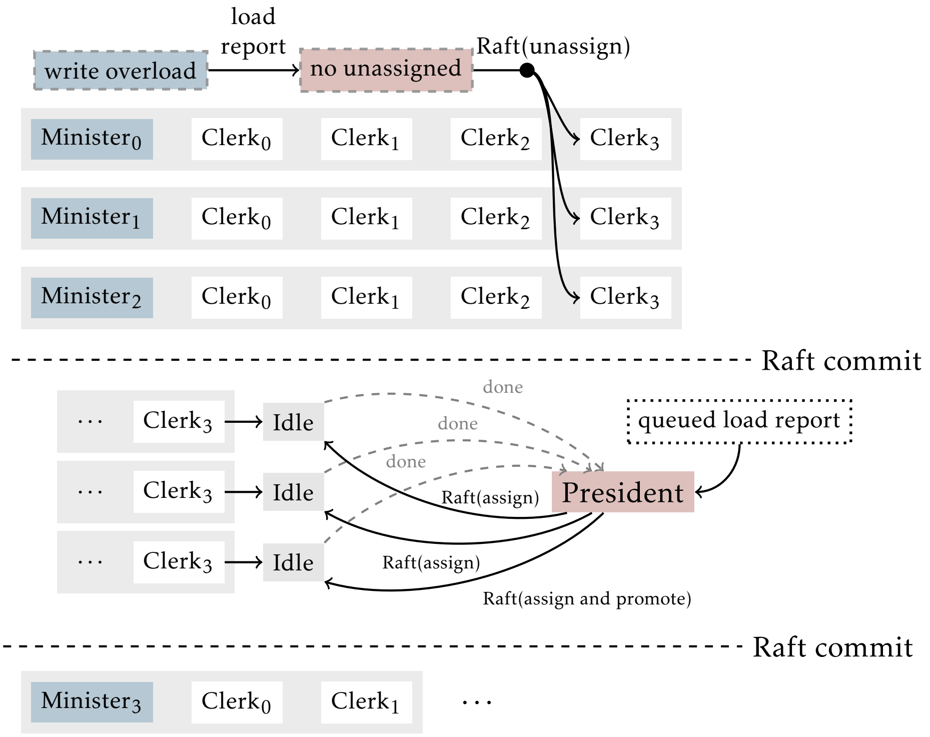

Figure 11: *A minister ■ under too high a load, higher than all other, sends a load report to the president ■. It can not create a new ministry as there are no or not enough idle nodes ■. The president removes three clerks □ from the ministries under the lightest read load and queues the load report. After the clerk removal is committed the load report is enqueued.*

4 Implementation

Here we go over the implementation of the design, which is written in *Rust*. We begin by motivating the choice for *Rust*. Following that we go over the concurrency model. Then using small extracts of source code we discuss the structure. Next we take a more detailed look at our extension of Raft (see: Section 2.3) and discuss why we could not build on existing libraries. Finally, we see how the file leases are implemented.

4.1 Language

Distributed systems are notoriously hard to build with many opportunities for subtle bugs to slip in. Therefore, it is important to choose a language with features that aid our work and make it harder to introduce bugs. Let's discuss one of the key features that can help us and one that could become problematic.

A strongly typed language with algebraic data types enables us to express properties of the design in the type system. An example: *Clerks* are listening for messages from the *President* or their *Minister*, we keep these separate by listening on different ports. Normally a port is expressed as an integer. If we make the type of the President's port different from the Ministers the compiler will prevent us from switching these around. This practise is known as Type Driven Development (TDD).

Timing is critical for the design, if the president does not send heartbeats in time elections might start. Languages using Garbage Collection (GC) pause program execution once every while to clean up memory. This can cause timing problems, also known as the *stop the world problem*. It is possible but hard to mitigate this by carefully tweaking the GC to keep its pauses short. If possible we should use a language without GC.

Only the *Rust* language has such a type system without using GC. Furthermore, the language guarantees an absence of data races which makes a concurrent implementation far easier.

4.2 Concurrency

When sending and receiving data over the network most time is spent waiting. Blocking while waiting is not at all efficient. We can use this valuable time to start and or finish sending and receiving other data concurrently. Usually this is solved by spawning a thread for each connection. Another way of doing this is using *non-blocking IO*, however organizing a single thread of execution to use non-blocking-IO for a diverse set of concurrent operations becomes highly complex. Maintaining file leases (see: Section 3.3) requires us to hold many concurrent connections. On the other hand one thread for each connection could limit the number of connections as we run out of threads. To get around the problematic complexity of non-blocking-IO we use: *Async/await*[21]. It is a language feature which allows us to construct and combine non-blocking functions as if they were normal functions. *Rust* has native support for the needed syntax but requires a third party framework to provide the actual IO implementation, here we use the *Tokio* project [23].

There is a trend in distributed systems to take scalability as the holy grail of performance [16]. While the design of the system focuses on scalability our implementation tries to use the underlying hardware optimally. *Moor'se Law* still holds its ever-increasing transistor count however no longer results in significantly increased single core performance. Instead, the increased transistor budget goes towards horizontally scaling [8]. In recent years we see this scaling in the form of increasing core counts[22]. The implementation should proof the design is future-proof by taking full advantage of available task parallelism. Fortunately the above-mentioned framework *Tokio* provides tasks which combine organized non-blocking-IO with parallel execution. These tasks are divided into groups where each group runs concurrently on a single OS-thread. Creating and destroying tasks is fast compared to OS threads.

Sharing state concurrently is with few exceptions achieved by passing messages between tasks. Where needed these include a method to signal back completion. Some shared state is used to keep track of the Raft lock, it is contained to the *raft* module. By mostly using message passing less time is spent waiting on locks and deadlocking bugs are contained to sections using shared state[23]

[21] See Appendix A for an introduction to Async/await.

[22] Enabled by CPU chiplets: multiple smaller dies that are combined into a single multicore CPU.

[23] Most of the message passing does not block, instead has a small buffer and returns an error if the buffer is full.

Cancelling tasks

In GovFS's design we frequently need to abort a concurrently running task. Clerks for example handle client requests in a concurrently running task. When a clerk becomes president it needs to stop handling those requests. If we were using threads we would do this by changing a shared variable. The task would be written such that it frequently checks if the variable is changed and when it is the task returns.

Whenever an *async* function has to await IO it returns control to the scheduler. When IO is ready the scheduler can choose to continue the function. We can ask it not to, this effectively cancels the task. Since Rust enforces Resource Acquisition Is Initialization (RAII) [22, p. 389] [24] the framework drops all the objects in the scope of canceled tasks.

Task handles instruct the framework to cancel their task when they are dropped. A group of tasks can be canceled by dropping the data structure that contains their task handles. We organize concurrent tasks as a tree, cancelling and cleaning up an entire branch is as easy as dropping the task handle for the root of that branch. Concretely if we abort the *president* task we automatically end any tasks it created.

4.3 Structure

Nodes in GovFS switch between the role of *president*, *minister*, *clerk* and *idle*. The roles are separate functions. When a node switches role it returns and enters the function corresponding with its new role. The switching is implemented in the state machine seen in Listing 1. In Rust expressions return a value, the **match** statement in line 2 returns the role for the next iteration. The different work functions set up the async tasks needed, then they start waiting for an exit condition.

[24] A programming idiom where acquiring a resource is done when creating an object. When the object is destroyed code runs that release or cleans up the object.

Listing 1: *The state machine switching between a nodes different roles*

```
let mut role = Role::Idle;
loop {
  role = match role {
    Role::Idle => idle::work(&mut state).await.unwrap(),
    Role::Clerk { subtree } => {
      clerk::work(&mut state, subtree).await.unwrap()
    }
    Role::Minister {
      subtree,
      clerks,
      term,
    } => minister::work(&mut state, subtree, clerks, term)
      .await
      .unwrap(),
    Role::President { term } => {
      president::work(&mut state, &mut chart, term).await
    }
  }
}
```

Before nodes enter the state machine they set up two Raft logs. The president log handles messages, timing out on inactivity and holding elections in a background task. The minister log handles only receiving messages. In both cases newly committed log entries are made available through a queue to a Raft Log object. Election losses and wins are also communicated through this queue.

Let us take a look at the president work function in Listing 2. We enter it if we are elected president. One of the arguments this function receives is the presidential Raft Log. It borrows the logs parts: the queue, and the Raft state. The state is wrapped in a LogWriter which allows appending to the Raft log and waiting till the entry is committed. Finally, a LoadBalancer instance is set up. The created objects are passed to the async functions or task, which are:

- *load_balancing*: issues orders assigning nodes and file subtrees to ministries using the LogWriter, re-assigns based on events such as: nodes going down, coming back online, new being added and ministry load.
- *instruct_subjects*: performs the leader part of the Raft algorithm. Shares log entries with all other nodes and tracks which can be committed.
- *handle_incoming*: handle requests, redirecting clients to ministries.
- *receive_own_order*: apply committed orders from the Raft Log queue to the programs state.

These tasks are selected on, making them run concurrently until one of them finishes. Here this means they run until *recieve_own_order* returns. This happens when the Raft background task inserts a ResignPres order indicating a higher termed president was noticed. After the select call finishes the president work function ends and returns the next role: Idle.

The other work functions similarly select on multiple async tasks. These tasks themselves create yet other tasks. This way the program builds up a tree of concurrently running functions. The tree is illustrated in Figure 12. Work that scales with system load is divided over a variable amount concurrently running tasks. Each connection to a client for example is run in parallel on a separate task.

Listing 2: *The president work function, it performs all the tasks of the president. In this code snippet brackets and parenthesis containing whitespace mean the structs and functions there have their arguments hidden for brevity*

```
pub(super) async fn work( ) -> crate::Role {
  let Log { orders, state, .. } = pres_orders;
  let (broadcast, _) = broadcast::channel(16);
  let (tx, notify_rx) = mpsc::channel(16);

  let log_writer = LogWriter { };

  let (load_balancer, load_notifier) = LoadBalancer::new( );
  let instruct_subjects = subjects::instruct( );
  let load_balancing = load_balancer.run( );

  tokio::select! {
    () = load_balancing => unreachable!(),
    () = instruct_subjects => unreachable!(),
    () = msgs::handle_incoming(client_lstnr, log_writer) => {
      unreachable!(),
    }
    res = receive_own_order(orders, load_notifier) => {
      Role::Idle
    }
  }
}
```

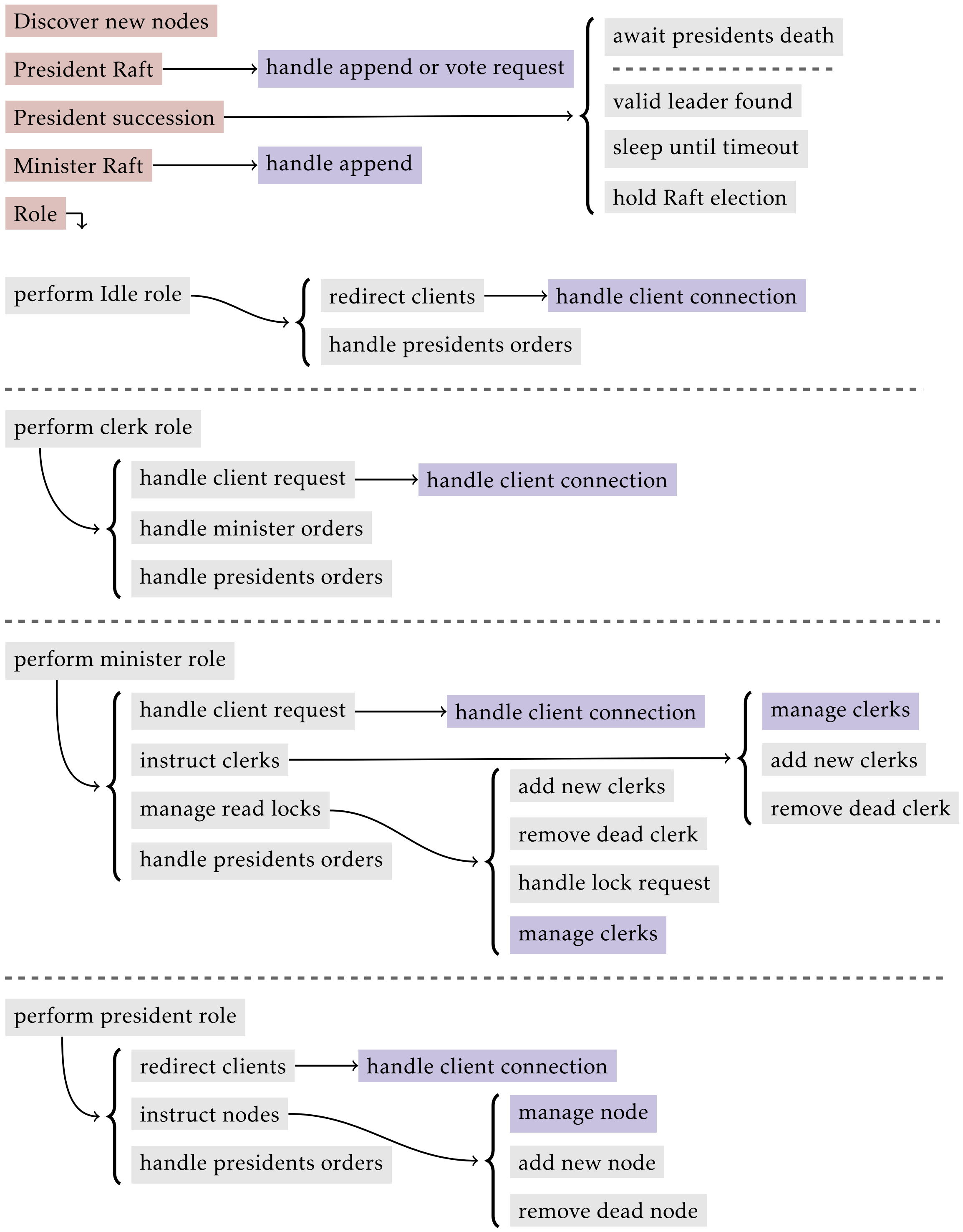

Figure 12: *Diagram of all concurrently running functions in a node. A dashed line between items means only one of those items can be running at the time. For example a node in the Idle role can not concurrently be a Minister. Functions in red* 🟥 *are single tasks while purple* 🟪 *indicates there are between zero and n instances of the function running. Functions in gray* ⬜ *are futures: they are running concurrently, however share a thread with any parent and or child futures.*

4.4 Raft

There are a lot of reliable Raft implementations. Developing our own took a significant amount of time. A new implementation will be less stable as it misses years of testing, this may impact our systems' stability. Building on existing work however was not an option as GovFS has two unique requirements:

- GovFS uses the Raft heartbeat to maintain file system consensus (see: Section 3.2). Newly assigned clerks for example use the heartbeat duration to know determine if their state is up-to-date[25] and can begin serving clients.
- GovFS needs a special version of Raft, one where elections are rigged and leaders (minsters) are assigned by a third party (the president). Multiple of these instances (or ministries) must be able to exist simultaneously. The log must stay consistent and clients should see no entries of an old leader after being assigned to another leader.

To demonstrate that GovFS scales and can be optimized in future work the custom implementation must also scale and be optimizable. If it does not then the design of GovFS could be relying on an implementation detail that fundamentally limits its performance.

Perishable log entries

When a Raft message arrives it can cause entries in the log to become committed. At that point they are made available to the system. These could be old entries, long ago committed by other nodes. The message contains the index of the last committed entry or entries which we use to recognize if an entry is old. Newly committed messages can still become outdated if they are applied too slowly. This can happen if the server slows down due to bugs in GovFS or hardware issues. The system notes the time a newly committed entry arrived. The time is combined with the entry into a perishable entry. It is what is made available to GovFS and can be asked whether it is fresh.

[25] That is, the clerk has applied all log committed entries, and the last was committed within a Raft heartbeat of it being committed.

4.5 File leases

As discussed in Section 3.2 read and write access is coordinated by a ministry. Before issuing write access a minister must ensure outstanding read leases are revoked. Similarly, clerks must ensure they do not offer read-leases to files that can be written to. The minister *locks* the needed file on all the ministries clerks before issuing a write-lease.

Managing these read locks is the responsibility of the *lock manager* which runs concurrent to the ministers other tasks (see Figure 12). When the client connection handler ■ receives a write request it enters a write_lease function. This checks if it has already given out a write-lease, returning an error if it has. Then the *lock manager* is requested to lock the file. A lease-guard is constructed once the file has been locked on the clerks. The guard unlocks the file if the handler leaves the write_lease function. This guarantees the file is unlocked even if the function is aborted due to an error. Then the client is returned the lease together with a time before which it needs to be renewed. For as long as the client keeps sending RefreshLease on time the handler stays in the write_lease function.

Leases are not flushed to stable storage (hard drives) and as such they are volatile. When a clerk goes down all leases issued by it are lost and clients need to reacquire them. A minister going down means loss of all the write-leases, the clerks, however, can keep issuing leases. The new minister unlocks all files when it comes online. These rules allow GovFS to use simple TCP messaging instead of relying on Raft for everything. Assuming files access is more common than file creation and removal optimizing lease management will speed up GovFS significantly.

Locking Rules

The *lock manager* times its lock requests to ensure consistency and correctness. It is easiest to explain this at the hand of an example. Here a clerk gets partitioned off from the rest of the cluster at the worst possible time:

- A minister receives a write request for file F
- At time *T* clerk A receives its last heartbeat from the President
- Clerk A loses connection to the rest of the cluster but stays reachable for clients.
- The lock manager fans out a lock request for F, it can not reach clerk A and starts retrying.

- Just before time $T + H$ clerk A issues a read lease to a client, it is valid until just before $T + 2H$
- At time $T + H$ clerk A misses the next heartbeat and stops handling client requests
- Just before time $T + 2H$ the client fails to refresh its lease and stops reading
- After $2H$ the lock manager gives up sending lock requests to clerk A. It is guaranteed that any outstanding read-lease issued by clerk A has now expired
- The minister issues the write-lease for F

We see that $2H$ after the lock manager started trying to lock the file it can assume the file locked. A clerk going offline increases file write access by $2H$. If the manager keeps trying to reach it we keep this $2H$ overhead. Instead, the manager removes the clerk before handling another request. Without any failure file write access time should be dominated by the latency of the TCP roundtrips.

Performance

The lock manager has been written to handle many simultaneous requests. It is therefore lockless and keeps an open TCP connection to its clerks. Keeping the connection open eliminates the overhead of opening one for each lock request. The minister communicates with the manager through message passing. When clerk gets assigned by the president the lock manager receives a message. It then opens a connection in a new concurrent task dedicated to this new clerk.

The decisions the lock manager makes directly impact the rest of the cluster. Each lock placed on a clerk potentially blocks read-leases which potentially slows down read performance. Therefore, it is important to unlock as soon as possible. The lock manager thus prioritizes unlock above lock requests.

Known problems

The current implementation has four known problems. Three of these have simple solutions however fourth requires changes to the design. First, an imposter or failing node can still send unlock requests. Including the current minister term in the request and checking if its valid would solve this[26].

[26] Similar to Group Raft, see Section 3.2.

Second, a newly assigned clerk can serve clients before it has processed all the existing locks. Clerks get their ministerial Raft log up-to-date before they start serving requests. The same should be done for lock requests.

Third, a network fault could make it impossible for *only* a minister, and thus lock manager, to reach one of its clerks. Traffic from the president and clients would still reach the clerk. In this case the lock manager assumes a file locked after $2H$ while the clerk does not miss a heartbeat from the president and stays up. This clerk could now enable reading to a file that is being written to. We can prevent this by making the minister inform the president of the clerk's failure. The president would then exclude the clerk from heartbeats triggering its shutdown on time.

Finally, a lock request can fail when the file has not yet been created on a clerk. In Raft a log entry becomes committed after the majority has accepted it. In the current implementation file creation is done as soon as the corresponding log entry is committed. A clerk that is behind in processing log entries can receive and start processing lock requests. Unfortunately this reveals a design problem: *there is no mechanism to handle a clerk lagging behind*. In section Section 8 we discuss how the design can change to address this.

5 Results

We now describe a number of experiments to investigate whether a file system using groups of metadata nodes can offer better performance and scalability then using a single node.

Our prototype implementation has a number of limitations. The construction of a full system based on this technology is left as future work. The limitations are:

- The Raft implementation sends only a single log entry at the time and log entries are sent each heartbeat period instead of whenever data becomes available. Effectively this imposes a rate limit on changing the systems metadata.
- Load balancing only replaces nodes that go down, it does not perform subtree partitioning at runtime (see: Section 2.5).
- Sometimes nodes become unresponsive when making many requests that change metadata. They then miss heartbeats which triggers re-elections.

In Section 5.1 we look at the *ministry architecture* tracking how performance changes when we change the number of ministries. Then we look at *ranged based file locking* in section 5.2, we compare write performance with and without locking. In each section we will detail how the experiments work around the limitations, ensuring their outcome would match an implementation without restrictions. Every benchmark was performed five times and all the results are presented. The nodes were monitored during each run and if an error occurred we repeated the entire run. When using multiple clients to create a larger load the nodes were given time to initialize and the test start was synchronized. All raw data is available at github.com/dvdsk/Thesis.

The benchmarks have been performed using the fifth generation distributed ASCI Supercomputer [3]. Each node has dual eight-core Intel Xeon E5-2630 v3 CPUs and 128G of ram. As networking delays are a key characteristic of the systems' performance we used Ethernet for the communication between nodes even though the hardware is equipped with InfiniBand. The Raft heartbeat duration was set to 75 milliseconds for all tests.

5.1 Ministry Architecture

Here we test the impact of varying the number of ministries on performance. We expect to see an almost linear improvement with more ministries given optimal size and shape of the load.

We use static subtree partitioning to vary the number of ministries. The load balancer starts by initializing a single ministry responsible for the root directory. Additional static ministries can be passed through command line parameters. For these test we let it initialize n-1 extra ministries at paths */n*. Including the root ministry (it must always be present) there are now n ministries available for testing.

List directory

The first workload we try is list directory. We perform 60 thousand list requests spread across all directories in the root directory. We do this from 30 clients concurrently. Each client sends the requests one after another as fast as possible. To keep the overhead from sending back the directory content small each directory contains only 10 files. The client processes are spread between 3 physical nodes.

The 2000 requests each client sends can be in two different orders: *batch* and *stride*. In *batch* a client performs all the requests for a single directory before sending those for the next. When using the *stride* pattern a client sends a request to the first directory and then sends the next request to another directory. There were a few extreme outliers, these were further out than 4 standard deviations. We consider these outliers therefore they are not shown in the plots.

In Figure 14 we see a violin plot comparing the two request orders for clusters with various number of ministries. On the Y-axis we see the time it took a single request to complete. A wider distribution means more requests were completed within the same time. Note the multimodal distribution, the increasing duration for the slowest requests as the number of ministries increases and the difference in the fastest times between batch and stride. From this we learn that batch mode is always faster than stride and that using more nodes increases tail latencies.

Figure 13 shows cumulative density functions (CDFs) of clients performing 60 thousand list directory requests in *batch* mode for varying amount of ministries. On the Y-axis are the proportion of requests that complete within the time on the X-axis. Note using *one* ministry is always the fastest followed initially by a configuration using *five* ministries until 80% of requests complete. Past 80% the second place is taken by the configuration with *two* ministries. This shows us that when using more than 2 ministries more ministries will speed up most list directory requests a small fraction at the cost of significantly longer tail latencies.

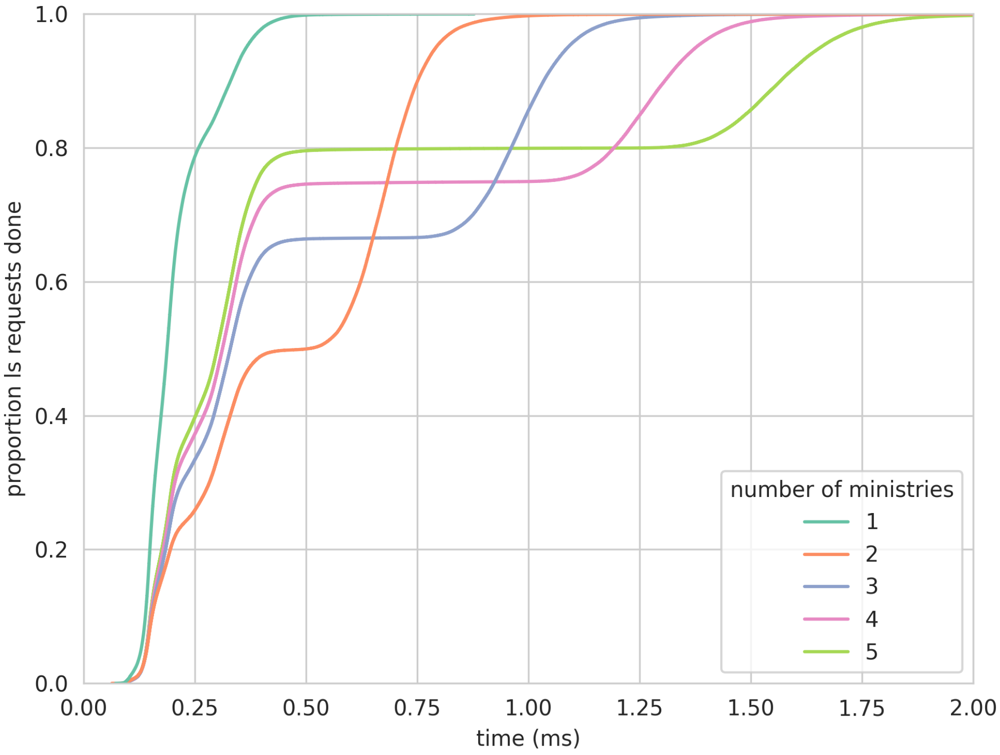

Figure 13: *CDFs of clients performing 60 thousand list directory requests for varying amount of ministries. On the Y-axis the proportion of requests completed, at 1.0 all 60 thousand requests have been answered. The X-axis shows the time until the proportion was reached. The clients batch ordered their requests: they first perform all requests for one ministry and then for the next. The chart goes up to two milliseconds, a tiny fraction of requests take longer then that.*

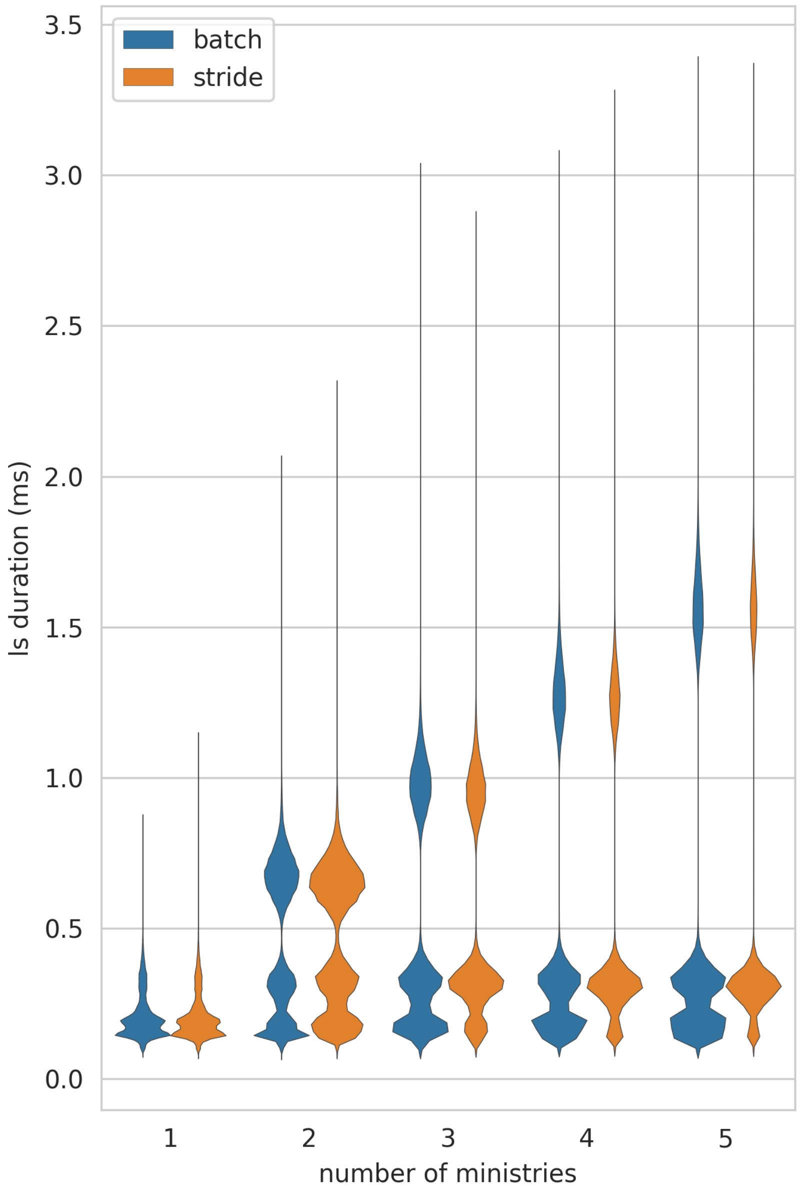

Figure 14: *Distribution of list directory duration vs number of ministries for two different request orders. On the Y-axis the time it takes a single request to complete. The X-axis shows the number of ministries the file system is using. The orange distributions are results from runs where 30 clients ordered their requests such that the ministries were accessed one after the other, a stride pattern. The blue distributions show results from runs where 30 clients used a batch pattern: they first perform all requests for one ministry and then for the next. Outliers further out than 4σ are not shown.*

Create file

The second workload we try is file creation. To create a file a minister appends a single message to the log making creating files rate limited. Without the rate limit, load could increase until the communication with the clerks or the hardware of the nodes becomes the bottleneck.

As adding more ministries could alleviate future bottlenecks it is interesting to see how file create performance scales with the number of ministries even given with the rate limits.

Because of this limit imposed by the implementation we send only 90 create requests. They are sent from 9 clients concurrently. These numbers were empirically determined to maximize the load while keeping the cluster stable enough to complete all the tests.

In Figure 15 we see CDFs for the time it takes to create a file on configurations with a various number of ministries. On the Y-axis the proportion of write requests completed, and the X-axis shows the time in milliseconds. Note the first jump upwards is about 75 milliseconds and all the other jumps are around multiples of 75. The more ministries we use the faster most requests are done. Finally, note the proportion of requests completed jumps up in discrete steps. Write requests duration is dominated by the time it takes the *Group Raft* log to commit a change.

Figure 16 offers another look at the same data. Here we see the time needed for each create as a function of when the request started. Darker tones are requests from tests with more ministries. On the Y-axis we see the time needed to complete the request in seconds while on the (logarithmic) X-axis we see the time a create request was sent. The vertical jumps in the CDF (Figure 15) show up as horizontal bands here. For example the lowest horizontal band are requests that took 75 ms which matches the first vertical jump. Note the gap just after the start of the test. When the system has more ministries more Raft logs can be appended to in parallel. Creating files therefore scales linearly with the number of ministries.

5.2 Range based file locking

Now we evaluate the contribution of ranged writes compared to writing the entire file. A good use case of ranged file access is writing out one or more

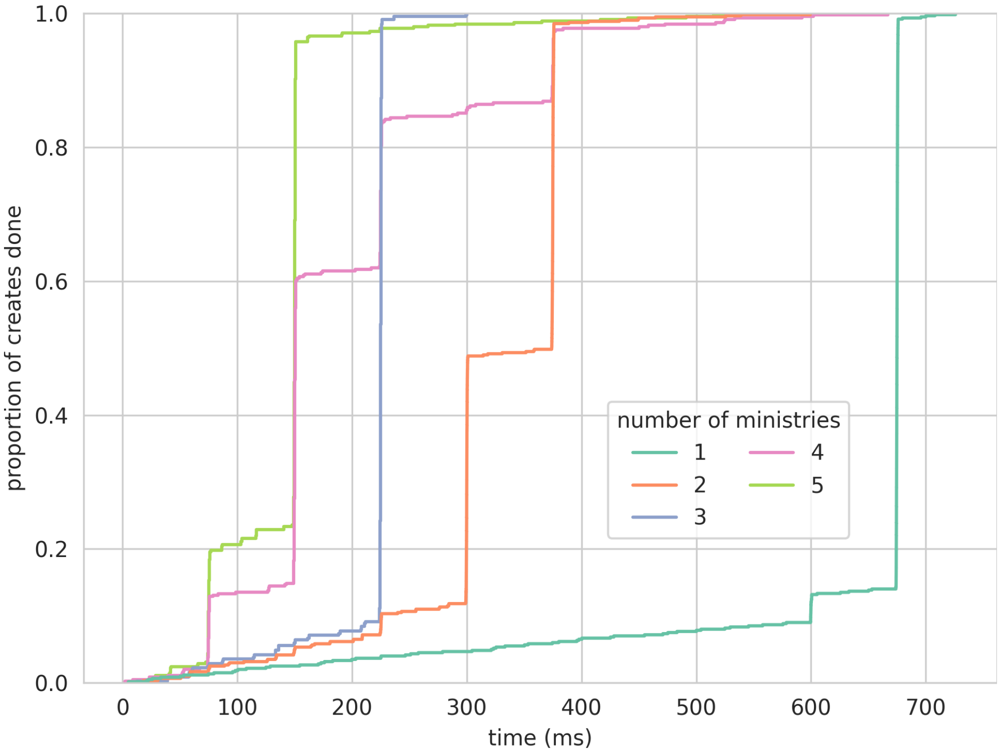

Figure 15: *CDFs of clients creating 90 files on clusters with various amount of ministries. On the Y-axis the proportion of files created, at* 1.0 *all 90 files have been created. On the X-axis the time in milliseconds until that proportion was reached.*

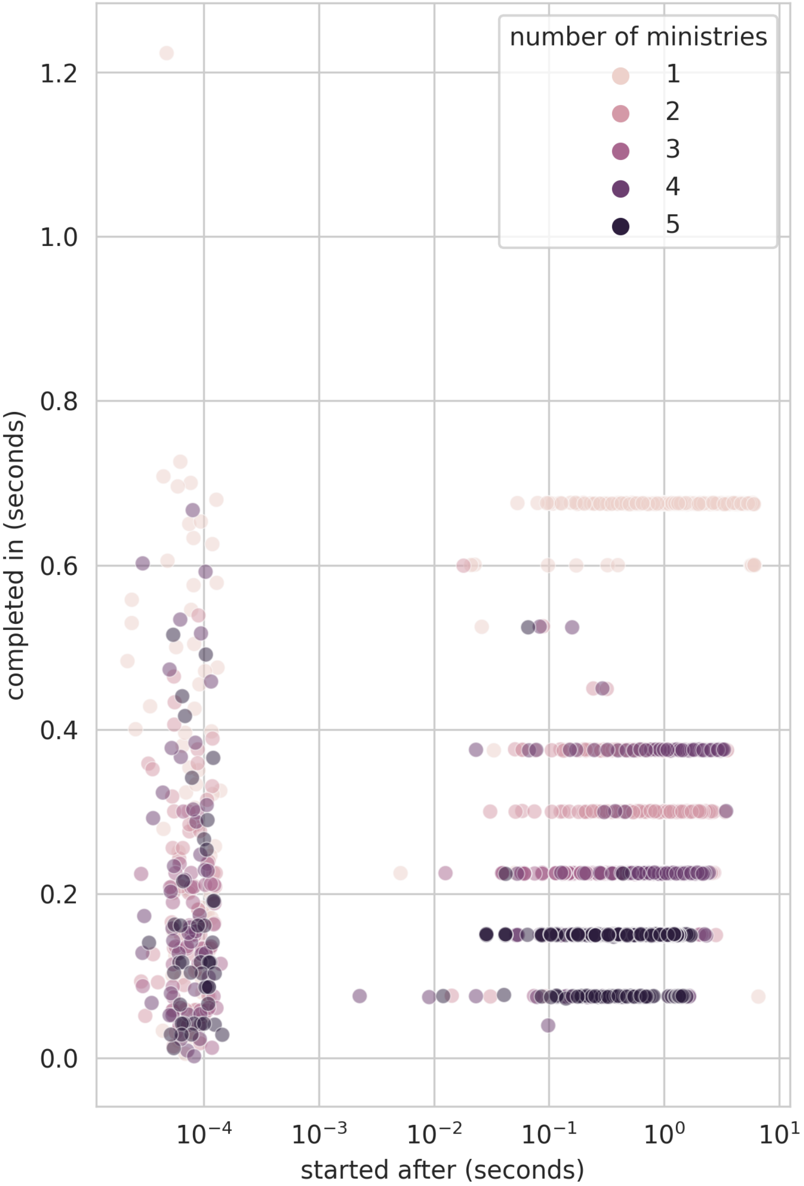

Figure 16: *A logarithmic plot of file create request time vs when the request was started. The same data as in Figure 15. On the Y-axis the time it took a request to complete in seconds while the X-axis shows the point at which the request was started. Darker points are from runs where the system had more ministries.*

rows in a file. In other systems we request exclusive access to the entire file and writing out all the needed rows. Here we can request access to and write out one row at the time. We expect this second method to be faster when contending with more clients for the same file and with larger files.

Here we compare these methods using two different experiments. In the first experiment we write a single row of a file with 10 rows from multiple clients simultaneously. Each client picks a row at random. In the second experiment each client writes all rows. Each client uses a random order when writing by row.

We run both experiments for various row size and number of concurrent writers. When varying the row size the number of concurrent writers was fixed at six. While changing the number of writers the row size was kept at 10 megabyte (MB).

Since GovFS has no data plane implementation writing is simulated by sleeping on the client side. We simulate writing at a speed of 200 MB per second[27].

In the table below we see the time spent on simulating IO and the average duration for writing one or 10 rows either by locking the whole file or locking by row:

	Row Size (MB)					
	0.1	1	10	20	40	80
Writing a single row (ms)						
IO simulation	0.5	5	50	100	200	400
Lock by row	411	370	366	444	636	971
Lock entire file	62	114	298	618	975	1644
Writing 10 rows (s)						
IO simulation	0.005	0.05	0.50	1	2	4
Lock by row	2.83	3.00	3.52	4.58	7.29	12.12
Lock entire file	0.57	0.82	1.51	2.65	4.90	9.60

In the next two subsections we will look at these results in greater depth.

[27] This corresponds to a slow hard drive which is a good bandwidth target for Hadoop like file systems targeting HDDs. It is also the best case scenario for writing by row since the low speed increases the ratio of time writing versus connecting and locking.

Writing a single row

Figure 17 shows the time it takes to write a single row for different row sizes. The Y-axis shows the duration a single write request takes in seconds. Each dot is a single measurement. We see that locking the entire file is slower than locking only the needed row. Increasing the row length shrinks the performance gap between these methods. Note furthermore that there are many outliers when locking by row. Locking by row is most efficient for small files.

In Figure 18 we see that with more simultaneous writers locking only a single row becomes faster. The Y-axis shows the duration a single write request takes in milliseconds. Note how locking the entire file is faster up to and including 4 writers. This is strange as both methods lock the file once. The experiment was therefore run thrice and the results carefully checked.

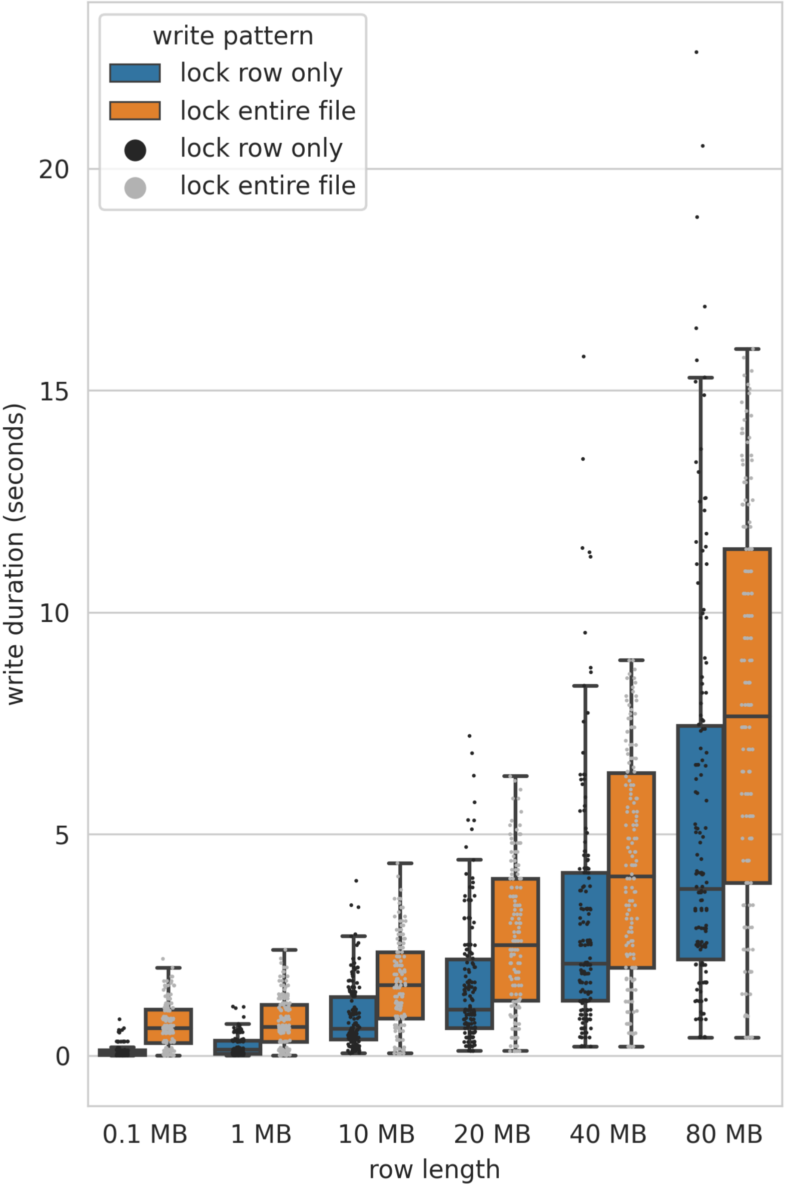

Figure 17: *The time it takes to write a single row given different row sizes. Every dot is a single measurement. On the Y-axis the duration a single write request takes in seconds. On the X-axis the row size in MB. In blue the time needed when only locking the needed row while in orange we see the time needed when locking the entire file.*

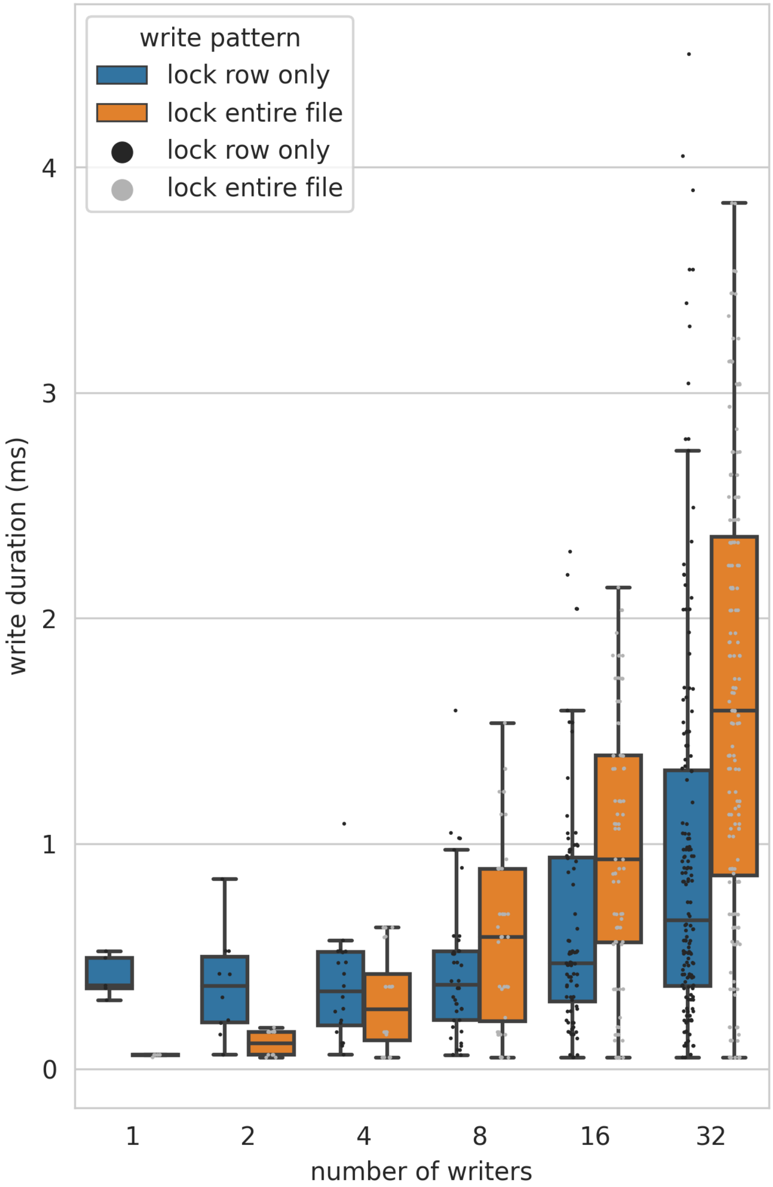

Figure 18: *The time it takes to write a single 10 MB row given different numbers of writers from 1 to 32. On the Y-axis the duration in milliseconds. The X-axis shows the number of writers, doubling every time. In blue the time needed when only locking the needed row while in orange we see the time needed when locking the entire file.*

Writing the entire file

In Figure 19 we look at every individual duration measurement for writing all 10 rows given various row lengths. Again we compare locking by row versus locking the entire file. The logarithmic Y-axis shows the duration in seconds. The X-axis shows various sizes for the rows. As expected larger rows result in longer write durations. Furthermore, we see discrete levels in write duration when writing the entire file and not when writing by row. Writing a file by writing each row separately is significantly slower than writing out the entire file.

Figure 20 again shows the duration it takes to write all the rows this time for varying number of writers, comparing locking by row versus locking the entire file. Each row is ten MB. The Y-axis shows the time it takes to write all 10 rows. Note how writing by row takes significantly longer with the gap closing when the number of writers increases. Also note the uniform distribution between the fastest and slowest time when locking the entire file (orange). The overhead from locking each row individually is not worth the gains from writing in parallel for normal file sizes.

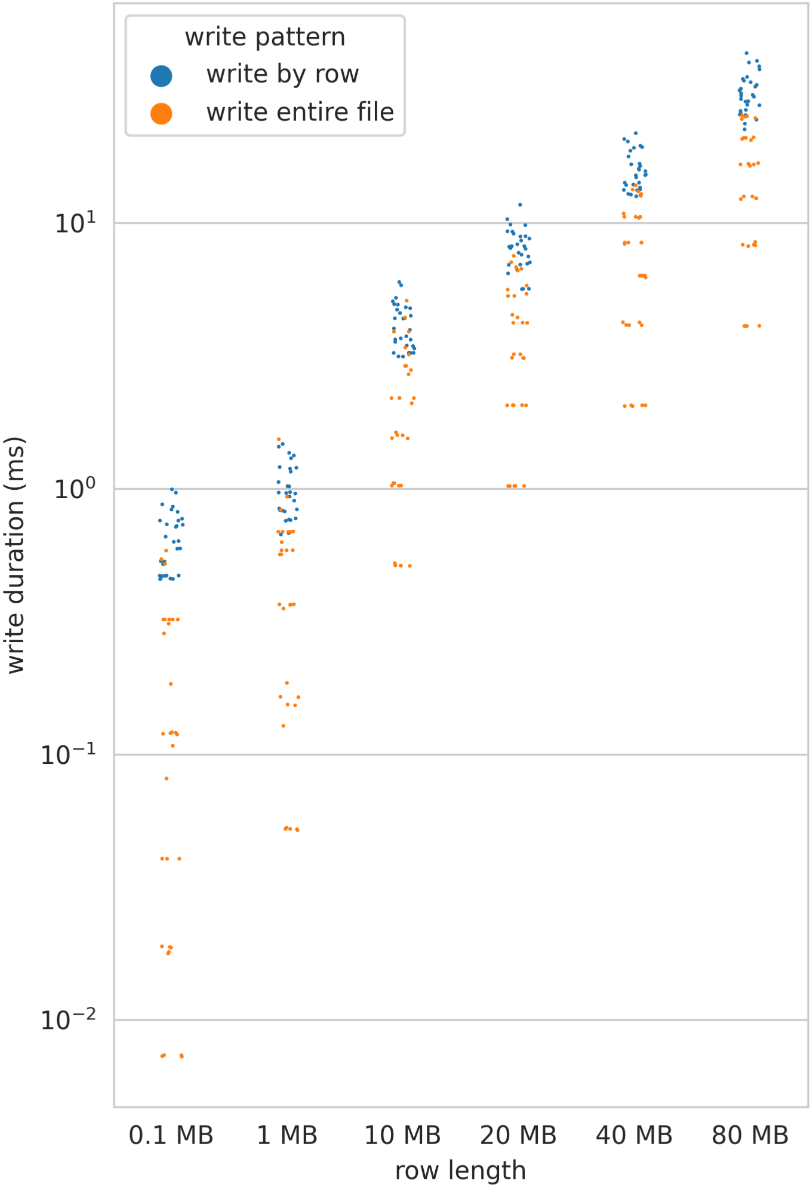

Figure 19: *The time it takes to write 10 rows for varying row sizes. The blue dots are time measurements when only locking the needed row, locking in total ten times to write all rows. In orange the duration when locking the entire file and writing all rows at once.*

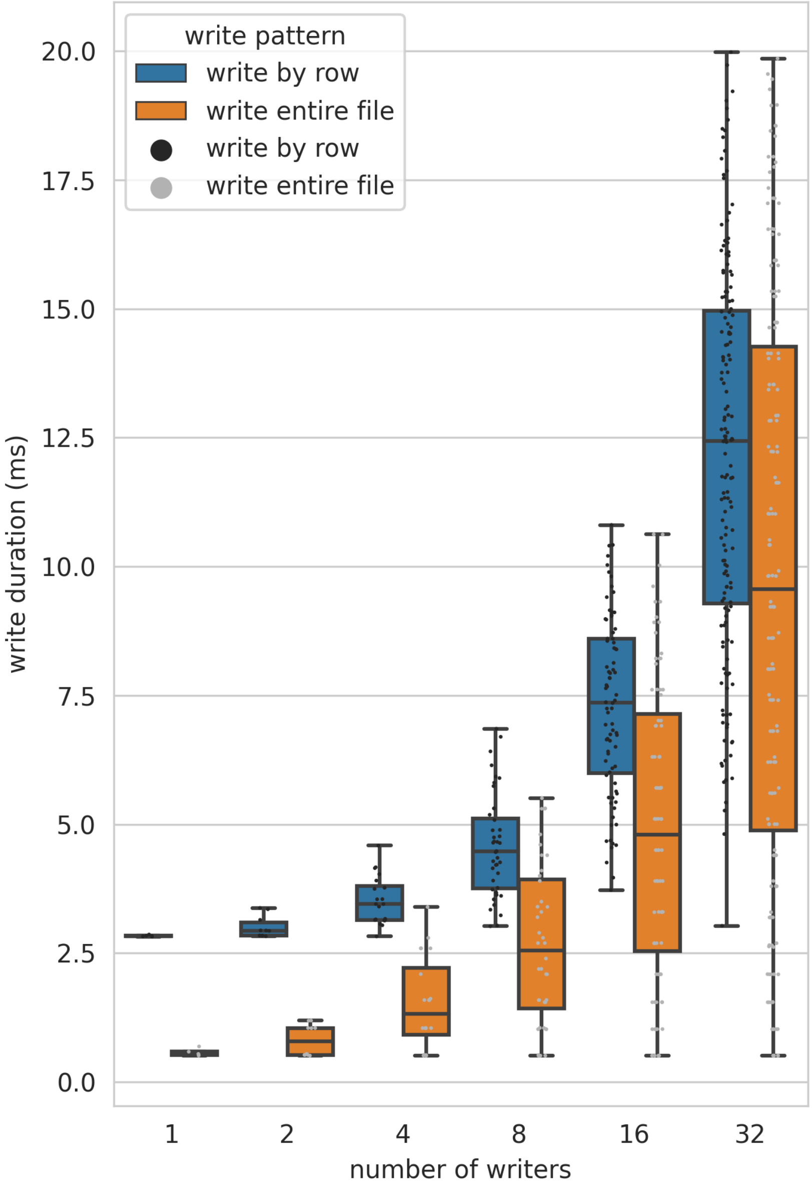

Figure 20: *The time it takes to write ten rows each ten MB with between one and 32 concurrent writers. On the logarithmic Y-axis the duration in milliseconds. In blue the duration when only locking the needed row, locking in total ten times to write all rows. In orange the result when locking the entire file and writing all rows at once.*

6 Discussion

In the previous section we presented the performance of GovFS. We focussed on its two distinguishing features: the *ministry architecture* and its *ranged based file locking*. Another important characteristic of any system is its complexity. The main contribution of the Raft consensus algorithm is a more understandable solution that is therefore easier to implement. We begin by discussing the complexity of GovFS then we discuss the performance implications of the *ministry architecture* before we finish discussing *ranged based file locking*.

6.1 Complexity

Raft is an understandable consensus algorithm. Here we have extended it to get *Group Raft* which provides greater scalability. This extended algorithm had two problems. These were both solved by re-using the Raft concept of terms. No new variables were added to the algorithm nor did we add new routines. It is still quite simple.

GovFS clerks need to know if their log is running behind. For this we added Perishable Log entries to *Group Raft*. These use log entry arrival time, the groups leaders commit index and the heartbeat duration to determine if a log entry is applied on time. Tracking arrival time is a trivial addition. All other variables where already available making perishable a simple addition.

As leases did not need to survive system crashes and reboots we did not use Raft but designed our own algorithm for non-persistent consistency. This enables GovFS to reach higher read and write performance. Even non-persistent consistency is still a hard problem, therefore this algorithm makes GovFS's implementation significantly more complex. In Section 4.5 we saw there are four known problems with this algorithm. While most have solutions solving these will further increase the now already high degree of complexity.

6.2 Ministry architecture

Design Faults During testing, we realized there is a fault in the current design. The president is responsible for detecting nodes that go down.

However, if the president can reach a clerk and its minister that does not guarantee that the clerk and minister can reach each other. We can address this by nodes informing the president when they can not reach one of their assigned nodes.

List Directory We saw the fastest response from a GovFS configuration using only a single ministry. As soon as we start using multiple latency increases, as we increase the number of ministries average performance recovers. Clients start without knowledge of the configuration, they are designed to initially connect to a random node. It is therefore logical that we see an increase in latency with more ministries: given twice as many ministries the chance to connect to the right node on the first try halves.

As expected the request order *batch* order is faster than *stride*. We would however expect there to be a larger difference. We did not expect the difference to get more pronounced as the number of ministries increases. It could be that it takes the client longer to re-establish its connection when the connection has been out longer. This requires study of the clients' performance and implementation.

The tail latencies become longer as we increase the number of ministries. This is unexpected. Requests should arrive at the right ministry after at most one redirect. The number of requests that need a redirect should increase as the number of ministries increases however those redirects should take the same amount of time. An explanation could be that the higher number of redirects overloads the redirect system leading to longer response times for those queries.

Create file Mean file creation time decreases almost linearly with the number of ministries. This should hold true even after the Raft implementation has been improved. Most file creations complete in a multiple of 75ms. This corresponds to the heartbeat duration used for Raft which imposes a rate limit on log modifications in this implementation.

Up to 5% of writes are completed in less time. At least a heartbeat period must pass before a new log entry can be appended. Only after appending the log entry does the minister inform the client the creation is done. Further study is required to explain these results.

6.3 Ranged based file locking

Writing a single row Locking only the needed data when writing to a file gives a dramatic increase in performance. This is in line with our expectations as the chance of lock contention decreases given less overlap in lock regions. Given larger files the difference in performance between the methods decreases as the time spent on (simulated) IO starts to dominate the time spent on locking. There are some far outliers that only appear when using row locking at the same time the variance of row locking is lower. We have no explanation for this, and it should be studied further starting with the behavior of the client.

In this experiment both methods lock the file once and both locks should succeed in the same time given the files start unlocked. The only difference is the range of the lock that is acquired. Locking the entire file is however much faster. This even holds with a single writer in which case there can be no contention. With only a single writer both methods should therefore do exactly the same and have the same results. Only when using more than four simultaneous writers do we see locking the smaller range becoming faster. Given these results, especially the results when using a single writer, there is a fault in either the test or the lease system.

Writing the entire file The lower lock contention when locking by row does not make up for the overhead of extra lock operations when locking ten times. Even when using very large files with rows of 80 MB locking the entire file is significantly faster. The gap does seem to close as row size increases. There is a uniform distribution of results when locking the entire file. This distribution highlights how each write can only be started after the previous completes. The width of the distribution comes from the five separate runs that are presented.

7 Conclusion

We have investigated whether GovFS, a file system using groups of metadata nodes can offer better scalability than using a single metadata node. The architecture of GovFS and *Group Raft* are relatively simple. The algorithm that coordinates file locking however is highly complex. The complexity kept us from building a highly performant implementation. We were able to test the impact of the architecture and ranged locking even with these limitations.

The speed at which we can create files scales almost linearly with the number of ministries. With multiple ministries' directory metadata queries take longer then when using only a single ministry. Mean query performance recovers slightly however as we add more than two ministries. This comes at the cost of increasing tail latencies.

With many clients writing to non overlapping chunks 10% of a file ranged locking offered substantial performance improvements. Using ranged locking when writing an entire file concurrently from many clients offered no advantage actually slowing down performance compared to locking the entire file.

We found a few surprising results that should not be possible given the design. These are probably caused by the limitations of the current implementation. With more implementation effort these results should be eliminated which would exclude faults in the design.

To enable the ministries' architecture we came up with *Group Raft* which provides scalable partitioned consensus. It has proven to be a simple extension with promising performance characteristics. Ranged file locking enables performance gains in limited circumstances but is easy to implement. Both of these should be studied further.

We conclude that a distributed file system using multiple small Raft clusters offers great scaling characteristics. More research and implementation effort is needed to see if such a system can be made as performant as the current state of the art.

8 Future work

Although we did not succeed in building a performant system we found a number of interesting avenues for improvement. These fall into three categories: dynamic scaling, more efficient Group Raft and ranged leases.

8.1 Dynamic scaling

The GovFSdesign includes load balancing through shaping the systems' configuration to the current load. Using subtree partitioning to spread directories across more or less ministries. In Section 3.4 we discussed the trade-off between read or write performance. This trade-off could be transformed into constrained optimization problem. It would be interesting to have GovFS keep an optimal configuration by continuously solving this problem. To allow users to specify to what degree each file should be replicated the replication factor can be added as additional constraint.

8.2 Consensus

The *Group Raft* implementation is very inefficient because it sends only one log entry at the time at fixed intervals. An efficient implementation will make it possible to directly compare GovFS with existing distributed FS. We expect a dramatic speedup by batching Raft log messages and sending them on demand. Since modifications to different directories can never impact each other we can use a combination of *ParallelRaft* [4] and *Group Raft*.

Currently, any node will start an election if it misses a heartbeat. Ministers can, and sometimes are elected as president. The loss of the minister causes its ex-ministry to pause metadata changes and file writes while a clerk is promoted and a replacement clerk assigned. Banning ministers for taking part in elections and preferring idle nodes to clerks will prevent such pauses.

One reason for re-elections is newly elected presidents taking too long to establishing connections and start emitting heartbeats. A better implementation would re-use the connections a candidate established asking for votes when it becomes president.

As groups only communicate with their members, clients and the president. Communication with the president does not require low latency nor high throughput. This could make *Group Raft* really useful for geo-distributed systems, that is systems consisting of clusters in different locations. As long as the members of each group are kept to the same location normal performance should not decrease.

8.3 Leases

Every write triggers a request to all the clerks to stop giving out read leases. This is inefficient for files experiencing mostly writes or for consecutive small writes. Ministers should keep files locked when the files are experiencing high write load. Ranged writes would then become suitable for concurrently writing a file from multiple clients using small non overlapping chunks.

Ranged locking should enable better performance in workloads with lots of reading and writing. It would be interesting to have this difference quantified.

The locking implementation does not batch requests. We suspect batching will significantly increase throughput under load.

A Introduction to Async

Async is a syntactic language feature that allows for easy construction of asynchronous non-blocking functions. *Asynchronous* programming lets us write concurrent, not parallel, tasks while looking awfully similar to normal blocking programming. It is a good alternative to *event-driven* programming which tends to be verbose and hard to follow. All Async systems are build around special function that do not return a value but rather a *promise* of a *future* value. When we need the value we tell the program not to continue until the promise is fulfilled. Let's look at the example of downloading 2 files:

```
1  async fn get_two_sites_async() {
2    // Create two different "futures" which, when run to
3    // completion, will asynchronously download the web pages.
4    let future_one = download_async("https://www.foo.com");
5    let future_two = download_async("https://www.bar.com");
6
7    // Run both futures to completion at the same time.
8    let futures_joined = join!(future_one, future_two);
9    // Run them to completion returning their return values
10   let (foo, bar) = futures_joined.await;
11   some_function_using(foo,bar);
12 }
```

Notice the `async` keyword in front of the function definition, it means the function will return a promise to complete in the future. The `join!` statement on line 8 combines the two promises for a future answer to a single promise for two answers. In line 10 we await or 'block' the program until `futures_joined` turns into two value. Those can then be used in normal and async functions.

The caller of our `async` *get_two_sites_async* function will need to be another async function that can await *get_two_sites_async*, or it can be an executor. An executor allows a normal function to await async functions.

Let's go through our example again explaining how this mechanism could work. The syntax and workings of async differ a lot here we will look at the language *Rust*. In rust these promises for a future value are called futures. Until the program reaches line 10 no work on downloading the example sites is done. This is not a problem as the results, *foo* and *bar*, are not used before line 11. The runtime will start out working on downloading `www.foo.com`, probably by sending out a DNS request. As soon as the DNS request has been sent we need to wait for the answer, we need it to know to which IP to connect to download the site. At this point the runtime will instead of waiting start work on downloading bar where it will run into the same problem. If by now we have received an answer on our DNS request for *www.foo.com* the runtime will continue its work on downloading foo. If not the runtime might continue on some other future available to it that can do work at this point.

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] Henri Bal et al. "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term". In: *Computer* 49.5 (2016), pp. 54–63. DOI: 10.1109/MC.2016.127.

[4] Wei Cao et al. "PolarFS: An Ultra-Low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database". In: *Proc. VLDB Endow.* 11.12 (Aug. 2018), pp. 1849–1862. ISSN: 2150-8097. DOI: 10.14778/3229863.3229872. URL: https://doi.org/10.14778/3229863.3229872.

[5] 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.

[6] Timothy Castiglia, Colin Goldberg and Stacy Patterson. "A Hierarchical Model for Fast Distributed Consensus in Dynamic Networks". In: *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. 2020, pp. 1189–1190. DOI: 10.1109/ICDCS47774.2020.00137.

[7] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[8] Yinxiao Feng and Kaisheng Ma. *Chiplet Actuary: A Quantitative Cost Model and Multi-Chiplet Architecture Exploration*. 2022. DOI: 10.48550/ARXIV.2203.12268. URL: https://arxiv.org/abs/2203.12268.

[9] 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.

[10] Xiaosong Gu et al. “Compositional Model Checking of Consensus Protocols Specified in TLA+ via Interaction-Preserving Abstraction”. In: *arXiv preprint arXiv:2202.11385* (2022).

[11] Dongxu Huang et al. “TiDB: a Raft-based HTAP database”. In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 3072–3084.

[12] Patrick Hunt et al. “{ZooKeeper}: Wait-free Coordination for Internet-scale Systems”. In: *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. 2010.

[13] 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.

[14] 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/.

[15] 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.

[16] Frank McSherry, Michael Isard and Derek G Murray. “Scalability! but at what {COST}?” In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. 2015.

[17] Diego Ongaro and Ousterhout John. *In Search of an Understandable Consensus Algorithm (Extended Version)*. https://raft.github.io/. accessed 15-Feb-2022. 2014.

[18] S Shepler et al. *Network File System (NFS) version 4 Protocol*. RFC 3530. IEFT, Apr. 2003. URL: https://www.ietf.org/rfc/rfc3530.txt.

[19] Konstantin Shvachko et al. *Consistent Reads from Standby Node*. https://issues.apache.org/jira/browse/HDFS-12943. Accessed: 03-03-2022. 2018.

[20] 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.

[21] Abraham Silberschatz, Peter Baer Galvin and Greg Gagne. *Operating system conceps*. John Wiley & Sons, 2014.

[22] B. Stroustrup. *The Design and Evolution of C++*. Programming languages/C+. Addison-Wesley, 1994. ISBN: 9780201543308. URL: https://books.google.nl/books?id=GvivU9kGInoC.

[23] Tokio Contributors. *Tokio*. Version 1.19.2. 27th June 2022. URL: https://tokio.rs.

[24] 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.

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

[26] Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’10. Boston, MA: USENIX Association, 2010, p. 10.