Grid Replicated Storage for the Minimum Intrusion Grid

Master Thesis

Computer Science, University of Copenhagen

Supervisor: Brian Vinter

http://www.diku.dk/hjemmesider/studerende/jaws/thesis.tar.bz2

21st of June 2010

Jan Wiberg <jaws@diku.dk>

Abstract

For years now the tendency has been that for data to become ever more vital to our lives and our work. Persistent storage devices have vastly increased in capacity but technologies that ensure data safety and no downtime are not yet as ubiquitous. The Minimum intrusion Grid (MiG) is a free and open source grid platform and it is an important element in the computing infrastructure for many of its users. So far, MiG has been restricted to operate on one host machine, with no data-sharing to other MiG servers or redundancy in case of a complete failure of either hardware, software or network at the single server.

In this paper, I present a design and a working implementation of a distributed storage solution built to fit the needs of MiG, called Grid Replicated Storage (GRSfs). It is able to offer flexible degrees of redundancy and is able to seamlessly fail over to hot-spares if needed. Secondary features include on-demand-fetching from hot-spares and a limited implementation of data branching. I also evaluate the final results from the prototype, independently as well as in the context of the grid server.

Resumé

Der har igennem mange år været en tendens til at data i stigende grad bliver en central del af vores liv og vores arbejde. Fysiske lagringsmedier er drastisk udviklet i samme periode, specielt hvad angår kapacitet, mens teknologier til datareplikering og afværgelse af nedetid er mindre udbredte. Minimum intrusion Grid (MiG) er en åben løsning til brug for grid platforme, og er allerede en vigtigt del af infrastrukturen for mange af dets brugere. Indtil videre har MiG været begrænset til at eksistere på én vært og har ikke tilbudt hverken data-deling til andre MiG servere eller redundans af hverken interne eller bruger data, i det tilfælde at der opstår en større fejl i nogle af serverens komponenter eller tilslutninger.

I denne rapport, præsenterer jeg en arkitektur og en fungerende prototype implementation af et distribueret filsystem, som er tilpasset MiGs behov. Jeg har kaldt dette filsystem for Grid Replicated Storage (GRSfs) og det tilbyder fleksibel ændring af antal kopier af data, og er i stand til at automatisk at inddrage ubrugte ressourser for at tilbyde brugeren en stabil dataløsning uden nedetid. Subsidiære funktioner inkluderer læsning over netværket fra ubrugte ressourser samt en begrænset implementering af data-splitning. Til slut evaluerer jeg resultaterne fra prototypen, både uafhængigt og i kontekst med grid serveren.

Contents

List of Figures

List of Tables

Listings

1 Introduction

This Master's Thesis presents a design and accompanying implementation of a distributed file storage solution with focus on high reliability and feasibility over wide-area networks.

I begin with an overview of motivation, limitations and structure. Later sections will treat design requirements and solutions in depth.

1.1 Motivation

Reliable and distributed storage has for many years been, and is still, a large and highly active field in computer science and IT in general. Many years of evolution have yielded highly advanced local storage technologies, but these are not directly applicable to networked storage which are now commonplace in situations where reliability, scalability and high performance are required. Distributed storage designs originated as comparatively simple client/server solutions, evolved to actual distributed filesystems, and more complex designs such as cluster file systems, wide-area network, SAN and peer-to-peer are starting to become commonplace. Special purpose systems, such as specialist designs for read-heavy workloads in the petabyte size range with relaxed consistency requirements, are increasingly commonplace as well. The usage scenarios vary greatly - a few examples would be small scale networks with access to just a shared drive with relatively low requirements on failover, throughput and latency specifications, large group/server clusters which often run on fiber networks against a SAN or similar storage, or supercomputer clusters with critical requirements for both reliability and speed.

It follows that there is no shortage of possible technologies, many of which are freely available. Many focus on scalability and high performance for HPC workloads, often with reliability-enhancement features (unless they eschew them altogether, in favor of letting the application do its own reliability handling) even if it is rarely the primary feature.

1.1.1 A case for a storage design

"Minimum intrusion Grid (MiG) is an attempt to design a new platform for Grid computing which is driven by a stand-alone approach to Grid, rather than integration with existing systems. The goal of the MiG project is to provide Grid infrastructure where the requirements on users and resources alike is as small as possible (minimum intrusion). MiG strives for minimum intrusion but will seek to provide a feature rich and dependable Grid solution." - www.migrid.org

Giving the large body of existing research and implemented designs, I will present an analysis, design and implementation of a distributed storage solution that is suited for a particular purpose. I have chosen the Minimum intrusion Grid, an open source grid middleware solution, as the baseline that problem analysis and design will be built on.

MiG is a so-called “dual-server” architecture where clients communicate with a MiG host that again communicates with the associated resources. The MiG middleware layer is designed for the presence of multiple MiG servers although it currently does not have a practical mechanism for sharing data between each instance of the core grid server component.

MiG is designed for anonymity: grid users connect to server, server connects to resources and vice versa, all through well-defined interfaces. There is no shared storage between user, server or resource.

The MiG group has previously experimented with alternative methods of synchronizing servers where all synchronization is done by the MiG middleware itself. These efforts have not been wholly successful and they now wish to evaluate a solution where data sharing and reliability handling is abstracted away from the MiG software stack. Both of these objectives are achievable with a distributed storage system and make MiG a good example on which to base further problem analysis.

1.1.2 Desired properties

The MiG group desires a solution with the following properties:

1. The solution should be highly reliable, if the data safety requirement cannot be met, then the system should gracefully disallow modifications at any remaining nodes
2. It should be operable under conditions like those of a non-dedicated wide-area network, which is to say, internet.
3. It should be possible for participants in the system to join and leave dynamically and seamlessly.
4. The redundancy factor - or equivalent - should be at least 3.
5. Files should ideally be accessible as a regular filesystem, and the storage system should follow POSIX file semantics[10]
6. Should support many non-connected data sets
7. While most of the software that comprises the MiG software stack is largely cross-platform, MiG itself is only tested on newer Debian-derived Linux distributions (x86 or AMD64). The host environment can therefore be expected to be almost or entirely homogenous.

Throughput and latency are not described as critical issues but nor can they be disregarded. In short, the performance criteria are somewhat vaguely specified.

The work in this thesis will build on the needs of the Minimum intrusion Grid - though there are no plans to change MiG itself - and explore whether the synchronization problem can be addressed by “out-sourcing” it to the file system layer.

1.1.3 Definition of reliability

Before going any further I will outline what I mean when I talk about distributed storage, reliability, latency and other high-level terms. However, some of the terminology in this paper makes more sense in the context in which the terms are introduced and definitions will appear as needed throughout this paper.

The topic of this thesis is online, fully available, distributed storage. In terms of storage reliability, the contrast is a *backup* - this term refers to copies of data, made at specific intervals, rather than being automatically synchronized when data changes, and can be retrieved in case of loss of individual file loss or failure of the main storage system. Backups should properly be stored off-site in the event of a disaster. Any further discussion here is focused on distributed storage.

A term like reliability means little in itself and needs to be defined closely. For a cluster of nodes that participate in or supply distributed storage to one or more clients there are several possible points where failure can occur:

Node hardware the hardware of the node can fail. That is not necessarily an urgent problem if the hardware itself is redundant (e.g. RAID)

Node software if the software on the node fails, which could be because of an implementation error or oversight

Communication links a node can be transiently or permanently disconnected from its peers/controllers if the network goes down

Data integrity errors in any of the above systems can introduce both detectable and non-detectable errors. Immediately detectable errors are, although not exactly desirable, at least often preferable over non-detectable errors that can corrupt large amounts of data over time

If all of these components were already highly reliable, there would be little reason to build redundancy into the system. As most people who have had any interaction with computers can testify to, computer components can fail at any given time and without warning. "A system is available if it provides service promptly on demand. The only way to make a highly available system out of less available components is to use redundancy, so the system can work even when some of its parts are broken." [45].

Failure types can be, very broadly, categorized into two types, data integrity compromise or failure to provide service. And while both types of error can be very costly in mission-critical systems, the former can have much greater consequences as data may be permanently lost in case of insufficient redundancy or backups. An insidious variant is undetected corruption of data.

Any distributed system cannot be said to be *reliable* unless it is able to safely handle failure, which can occur at any of the above points, of some number of nodes which encompass some fraction of a total system, simultaneously. The fraction of survivable simultaneous failures for a design varies greatly usually based on a deliberate tradeoff: cost per unit increases as redundancy rises, whether this cost is money, performance, or other.

Byzantine failures, or arbitrary failures, are errors where a component essentially acts arbitrarily incorrectly[17].

A system that transparently supports failover is able to reallocate available resources to continue operation during and after a localized failure. This, along with synchronization models, is treated in greater depths later on.

1.2 Objectives and limitations

The objective of this thesis is to analyze and evaluate the requirements for a distributed storage solution that addresses MiGs needs, design, implement, and evaluate the final product. The design will aim to be a solution that fits the needs of the MiG software. The chosen design will be implemented in a prototype solution that will be evaluated by a series of internal and external tests and by critical analysis of the final result.

I aim to address these problems in my thesis:

- Conduct a high-level analysis of the requirements of a distributed storage solution that fits the MiG platform based on the MiG groups' criteria
- Perform an analysis of the reliability and efficiency properties of the chosen design
- Create a prototype implementation
- Evaluate the implementation against the criteria put forth in the analysis sections
- Conclude, and propose extensions and improvements on the work.

1.2.1 Limitations

The focus is on designing and implementing a file system design tailored mainly for the Minimum intrusion Grid. More general-purpose optimizations and features are most often only peripherally considered. To further limit the scope of this thesis I do not consider soft errors or byzantine failures in other components.

A noteworthy limitation is the focus on robustness and correctness of the underlying design rather than performance at this point. There are many situations where we face the choice between two options which accomplish the same end goal but where one may be faster but also significantly more complex and thus generally more error-prone. Often it is possible to take a step back and consider whether the success criterium for a given goal really needs to be achieved in a particular fashion, or if an alternative approach may achieve the same. I have generally favored simplicity and robustness in these situations.

1.3 Summary of Contributions

The main contribution of this thesis work is a flexible distributed storage solution based on a primary-backup model. I present, implement and evaluate a working prototype that demonstrates that it is technically viable to use FUSE and Python together to form a resilient storage solution.

1.4 Thesis Overview

I defined the main wish list from the MiG group in 1.1.2. Section 2 gives a brief history of storage systems and which research currently defines the field, while section 3 describes the grid server architecture of MiG and analyzes operational data from MiG. I build upon these three pillars when I present a design in section refsec:design and the next section further reflects on the choices and implications of the design. Section 6 discusses the actual choices that the model requires in greater detail while 7 explains how to improve performance. Section 8 is an evaluation of a prototype implementation and refers back to previous sections to draw its conclusions. section 9 gives some thoughts on possible future research and improvements to the design and implementation of the prototype. Lastly, section 10 is a conclusion on the theoretical and practical work of the thesis.

2 Related work

Distributed storage and replication has been a focus area for commercial and academic research for four decades, leaving a massive body of research that will only be briefly summarized here.

2.1 Evolution of data Storage

This section is a brief history of filesystems, drawing heavily from work in [71]. As the amount of data has grown over the years and our dependence on safe and reliable storage has grown with it, file systems have evolved in an attempt to meet the challenge. Many lessons can be drawn from the success and failures of these, even if the work in this thesis is more special-purpose than most of the other featured systems.

Local Filesystems

Most modern Unix storage stacks conform either fully or in large part to the POSIX standard[10], which in turn largely derived from early filesystems designs. Local filesystems are most often designed around the physical reality of hard drives: the *sector* - the smallest allocatable unit on a hard drive, and the *seek time* - the latency caused by having to move the disk head to a new position. As a consequence many optimizations revolved around these realities; file meta data was stored close to file data in BSDs Fast File System for example.

Most early system used some variety of an *allocation table* that maps files and folders to individual disk *blocks* - the smallest allocatable unit in a filesystem. For a filesystem that uses 4 KiB blocks, a file of merely 1 B will still take up the full 4096 Bs. This model is rather inefficient for large files - a 1 GiB file needs 262,144 blocks of 4 KiB each. A newer approach is *extents* where the list of blocks is replaced with a list of start/stop pairs for each continuos part of a file.

Aside from efforts to improve utilization, there has been huge advancements for reliability as well. These two technologies are quite common nowadays.

Journalling Changes, to metadata in particular, are kept in a log, spiritually identical to the transaction log that databases often use. The log can be read to verify consistency and replayed if a failure occurs.

Soft updates Updates are written to current unused portions of the storage device, and the allocation system then switches to the new data simply by changing a pointer.

State of the art filesystems, at least non-academically, are probably SUN Microsystems' ZFS[36, 64] or the Oracle-sponsored Btrfs (B-tree file system)[15, 16].

There is little news from Microsoft Corp. in recent years after the, possibly terminal, delay of WinFS[1] in 2006 and it seems that future updates to the Windows local filesystem, NFTS or any successor, is kept under wraps. NTFS lacks some of the more advanced features of the newer Unix offerings, such as volume management, but considering the extremely extensive deployment of this particular product, it must be considered very stable and mature.

Client/Server Models

The earliest distributed filesystems were server-based where a, often single, server hosted the filesystem and allowed clients to access it over a network. There are a great many different systems of this variety. One of most well-known systems, NFS, originated from SUN Microsystems in 1985 and has been updated as late as 2003. NFS was, and in many places still is, ubiquitous in Unix environments and the later updates have greatly contributed to the viability of NFS. SMB, and now its successor CIFS, is the Windows eco-systems equivalent of NFS; it remains in extremely widespread use throughout the world, from small home or business networks and up to large enterprise installations.

Even so, the client/server paradigm is showing strains - SMB/CIFS in particular is an aging protocol that is originally designed to be used on smaller, low-latency, networks. Both of the protocols are also block-based and even with NFSv4 support for multiple data servers, the very large files that are common nowadays are straining the model. The common workaround of storing data on several servers helps alleviate the problems with scaling either storage sizes or performance but it is hard to react fast to changes - if a server grows full and must be split, a lot of clients potentially have to be pointed at the new server for example.

The level of data consistency, reliability and resiliency varies. Most newer client-/server protocols have some level of support for file locking, but redundancy is rarely built into the system and is instead provided by RAID at the server-level and/or offline backups.

Oracle is working on CRFS (Coherent Remote File System). It uses architectural ideas from Btrfs on a distributed level, with the intention of improving greatly over NFS and CIFS.

2.1.1 Distributed, WAN and SAN

Distributed filesystems aims to tackle some of the intrinsic problems of the client/server model. Data are now stored on several servers with various techniques used to abstract data access from the physical storage location. Some designs, particularly early ones, use partitioning schemes that do not achieve full abstraction; AFS for example cannot do atomic operations that span the boundary between these partitions. Some, AFS, Sprite, Coda and others, use a central server for coordination - an obvious single point of failure.

[1]WinFS is/was a departure from the traditional hierarchical structure of filesystems as it is based on a relational database

Variations of the above are wide-area-network (WAN) filesystems that are designed to work over slower high-latency connections. The trade-off that WAN filesystems have to contend with is that of scalability vs consistency: local, or near, caching of data can massively improve scalability but requires either relaxed consistency semantics or frequent communication to update state.

Storage-area-network (SAN) systems are typically purpose-specific networks where a number of storage devices are connected with a number of servers with very fast links such as fiber, thus correspondingly expensive. The core idea is to parallelize data access by allowing storage devices and servers to communicate SxC, i.e. any server can communicate with any storage device. The details vary and there may, or may not be, separate constructs that handle locking, metadata etc.

2.1.2 Object-storage

The object-based storage school of thought[6] is relatively more recent and differentiates from existing models by using and leveraging intelligent storage subsystems. Rather than keeping with the I/O model of blocks, object-storage models separate data from metadata. Data are stored in objects, and are typically given a globally unique identifier[2]. The associated metadata carries the actual filename, location, bookkeeping information, flags, security and the GUID so that the data object can be located.

Treating data as an "object" rather than a set of blocks allows us to abstract the data away from the matter of how it is stored on physical media. This abstraction sits between the file-access layer and the underlying storage, often a block device, and significantly simplifies handling of redundancy, scalability, performance and data management[58]. See also Figure 1. A potential downside, depending on implementation, is that metadata and data modifications are not necessarily atomic.

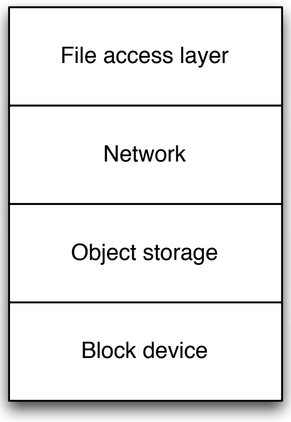


Figure 1: File access layers

The aforementioned "intelligent storage subsystems" is of course a flexible term but often covers a server of PC-level intelligence with some underlying storage. Grouping such together in clusters and using the object abstraction allows the software running on the cluster to move, replicated and otherwise handle data transparently from the client, thus giving a high level of flexibility to scale or otherwise optimize that is not possible with an non-abstracted system as, e.g., NFS or AFS.

[2]Or as close as a randomly generated 128-256 bit identifier gets to be globally unique

2.1.3 Specialized Designs

Most of the designs mentioned generally above are of the general-purpose variety; they are meant to be used with general-purpose applications and follow POSIX semantics, though often with relaxed consistency semantics. There are many other types of storage designs: Google File System[28], Hadoop[63] and others are heavily optimized against extremely large datasets and often only as read/append (i.e. overwriting is either not possible or slow). Others, like Farsite, are able to use non-dedicated resources to form a reliable storage pool.[4]

Googles' GFS utilizes a "master" which coordinates a number of chunk servers. File data are striped or replicated in 64MiB chunks across as many chunk servers as is needed, in a primary-backup scheme.

Dynamo from Amazon.com is a representative of a wholly different kind of system; it is essentially a distributed key/value store which, in the recognition that it is very hard to have both a highly available and highly consistent (ACID) system[24], emphasizes being highly available to avoid any down time for both internal and external users of Amazon.com.[18] A number of services with different performance and reliability requirements run on top of the store - each service can, by specifying different replication requirements etc., tune Dynamo for its specific needs. Dynamo is notable because failures are handled by the overlying application; a node failure in the store can result in multiple versions of the same data. Each of these may be considered correct, such as in the case of a shopping cart where Amazon does not want to lose any customer changes, and thus *reconciliation* between the multiple versions are handled, not by the storage layer, but in the application itself. Dynamo utilizes consistent hashing in conjunction with a ring topology to facilitate dynamic node enrollment and redundancy.

Peer-to-peer, or P2P, filesystems such as Ivy[50] are an interesting evolution in filesystems, but will not be treated in any further detail here.

2.2 Alternatives

During my initial research, a common question was whether an existing system could serve MiGs requirements and given the sheer number of filesystems in existence, the answer is not unlikely "yes". However, while many filesystems are at first glance quite advanced and feature rich, they are not a perfect fit - though they could be possibly be adapted.

RAID 1 over the network DRBD[54][55][56] is basically a RAID 1 setup over a network. DRBD uses a master/slave relationship with active replication to keep two machines in perfect sync. If the master falls over and is later reconnected, DRBD is able to detect and transfer only changed data to lessen the time it takes to resynchronize. It is not a filesystem as it mirrors an entire block device. It is not well-suited for WANs.

High Performance Computing, state of the art Lustre[47] “is an object-based, distributed file system, generally used for large scale cluster computing” [77]. It is primarily geared towards massive cluster installations and runs on several of the worlds largest clusters, grids and supercomputers[3] . Lustre has several reliability and redundancy provisions but does not actively replicate data in RAID 1+ style. Curiously, DRBD is sometimes used to provide active replication for the individual block devices. It is an object-based storage system that separates metadata and data servers.[48]. The metadata servers are again split into two constructs: metadata server (MDS) and metadata target (MDT). The target is basically a separate storage server only for metadata. They are intended to run on modern filesystems and very fast I/O infrastructure such as RAID1+0 solid state disks.

The data server again uses the same split architecture, with an object server (OSS) and object storage (OST). As with any good object storage system, internal storage mechanics and boundaries between storage devices are hidden from the end-user, allowing Lustre to stripe data as needed.

Lustre is gaining some support for WAN deployments though this does not seem finalized, and anecdotally, it is considered somewhat non-trivial to set up and maintain.

Ceph Ceph[72, 71, 69, 68, 70] is a relatively recent addition to the open source filesystem world and is not yet ready for production use. Ceph claims to offer seamless and highly adaptive scaling, as well as a high level of data safety. This is done in part through the use of hashing algorithms to determine file location, rather than explicit allocation tables or similar, and by proactively replicating data across many devices. Ceph is designed to run on a cluster, and like Lustre, appears to have WAN support only in the sense that a “gateway” device can replicate data between remote sites.

Mainly academic The academic world has produced a fairly large variety of filesystems. During the research phase of this thesis, I read papers on SGFS[82] and OBFS[66] amongst others. Either is a potential choice for a LAN-based cluster filesystem.

Another alternative is the IBM-driven GPFS. GPFS will not discussed in detail here but seems to be relatively popular and stable. Some general information can be found in at [76**?**].

2.3 Core features and research areas

End-to-end data integrity Contrary to the expectations of many, most filesystems do not in any way guarantee that data are not corrupted on the path from the system call to the physical disk. Errors in controllers, cables, software

[3]To illustrate the scale, DARPA intends HPC filesystems to handle more than 100PiBs of storage over 10^{12} files, with 30,000 clients, and exceeding 1500 GiB/s of bandwidth[48]

et cetera can all silently corrupt data. As one of the first production-quality filesystems, ZFS combines an end-to-end checksum mechanism with transactional support and replication in an attempt to improve data safety.[81]

Copy-on-write etc Using some variety of journalling, copy-on-write and/or soft updates for modifications has become very common. These systems have a much diminished window of failure as they use unused portions of the storage to write data into. Once the write is fully complete, the system switches to the new data atomically.

Consistent hashing and pseudo-random data placement Many clusters spread data over several servers, but rarely have all data in one place. Since data can be in several locations, it must obviously be possible to find out where they were put. Older techniques generally used a coordination server or some variant hereof, but newer techniques instead use a algorithm to determine placement. Dynamo use a consistent hashing algorithm[38] to place data, while Ceph uses a pseudo-random distribution function called CRUSH[71] to achieve a uniform spread over participating nodes.

HPC trends As Lustre and Ceph showed, the trend within HPC storage is to go beyond any single point of failure: data and metadata are often heavily replicated across multiple physical servers, with advanced software binding clusters together. This requires that the device has a greater degree of "own intelligence" than just a dumb disk. Having multiple redundant copies of any one piece of data also enables these designs to *stripe* data, enabling one client to request one large file from several sources concurrently.

De-duplication Filesystems that compute checksums for the purpose of data integrity typically do so per block of data. Some filesystems, such as ZFS, have expanded the use of the checksum and store it in multi-level lookup tables and if a given checksum is already in the system, it will simply store a pointer to the existing data block instead. There are varying levels of safety built-in, both i.r.t choice of hash algorithm as well as the option to compare directly if the chance of a hash collision is unacceptable. De-duplication is not necessarily a good idea for all workloads: it obviously works best if the degree of duplication within a given filesystem is high.

2.4 Literature review

As previously stated, the distributed systems field is both very wide and very deep. I will present some of the research that I have studied during this thesis, most of which has had a significant influence on my thinking and the end design. Most of my reading on concrete filesystems has been summarized above, and this section will look at the field on a more abstract level.

2.4.1 Fault-tolerance

When the resilience of a given system against errors is discussed, a few commonly recognized terms typically pop up.

Byzantine A byzantine error can occur if any one or more components in system, either arbitrarily or malicious inject or respond with malformed commands or data.[43] An example of a possible byzantine error source could be a malfunctioning RAM module in a given component. Distributed systems can withstand byzantine errors in some conditions by requiring a majority of participants to agree on the outcome. Effectively, this means that a system will need $2n + 1$ identical copies in order to handle n concurring failures.

Omission An *omission* is when a component transiently or permanently fails to obey its specification.

Crash A crashed system stops completely. Some delay can occur in detecting the crash from other parts of the system.

Fail-stop systems provides a method for other participants and components to detect its failure, and then stop, for a varying definition of "stop" depending on the error condition.[59]

***t-fault* tolerant** is described as "*A system consisting of a set of distinct components is* t-fault-tolerant *if it satisfies its specification provided that no more than* t *of those components become faulty during some interval of interest.*"[59]. A slightly terser explanation is that a system can tolerate t more or less simultaneous failures.

2.4.2 Ordering events

Any two consecutive *read* events are commutative iff we enforce that a read has no side-effects. In the POSIX model, a read operation must include the effect of the last *write* operation.

Whether the datastore is reentrant is irrelevant to the requirement that events are properly ordered. A reentrant datastore is possibly faster, and needs to pay more attention to safety, but unless we require of the overlying software stack that it itself enforces correct ordering, we must assume that any number of processes and threads can start requests and that these requests are unsafely ordered.

Many operations will effectively be weakly causally ordered, i.e. an opendir system call will often be followed in a pattern of $\forall e \in readdir(opendir(path)) : getattr(e)$[4], which can be exploited for optimizations but it will not change the ordering requirement.

[4]Executing ls −l <path> results in such a sequence

There are two major paradigms when designing a distributed storage solution, single master and multi-master. These topics are discussed in many books and articles; I primarily relied on [Distributed Systems (Sape Muller(editor)], Lamport et al., Wiesmann et al., Schneider and various Internet resources.

Ordering for a single master system This is often termed a *primary-backup system*[12] and covers situations with a single node responsible for updates (at least for a given partitioning of data).

Ordering for a multi-master system Multi-master systems are systems in which any member of a system can update data. This requires a distributed lock management scheme to avoid conflicting writes unless it is possible to relax consistency requirements. A replicated state machine is often used to synchronize a distributed system.[59]

The main attraction of a primary-backup system is the relative lack of overhead compared to a highly consistent multi-master scheme.

2.4.3 Commitment protocols

I evaluated a number of atomic commitment protocols. They are typically the simple variant in their class - specialized versions exist for special cases but I have not found reason to include them. Note that most of these are only relevant for the multi-master scheme.

Two-phase commit protocol (2PC) is a very widely used[9] protocol. On a peer-to-peer topology, it is not able to safeguard against deadlocks without a coordinating entity. It takes two message exchanges to commit. It is a safe choice in its own right but the two required exchanges should make it the backstop compared to other solutions (by being "safe" with two exchanges, there should be no need to consider protocols that require three exchanges for the common case)

Batched 2PC is a variant of 2PC which takes advantage of the master/slave nature and allows us to combine the *commit* step of $modification_n$ with the *prepare* step of $modification_{n+1}$. If $modification_{n+1}$ takes longer than acceptable, an explicit *commit* can be forced.

Three-phase commit (3PC) is a variant of 2PC. It requires three message exchanges and is able to remain lively in conditions where 2PC would fail, namely in systems with no central coordinator.

Paxos-family [44] is a whole class of algorithms designed to tolerate failures in a system of n processes. There are multiple variants with different properties, such as speed or resiliency. For some classes of problems, Paxos can ensure safe operation with as few message-exchanges as 2PC although some cave-ats apply.

Logical and vector clocks[42, 23] are used to partially order messages in a distributed system. Briefly, a counter is added to each message and upon successful receive, the receiver synchronizes their own internal clock against the message, keeping members in sync. The vector clock model expands this by adding the member name to the clock field as they pass through, yielding an effective way of keeping track of who has seen what in what order.

Fire and forget eschews any sort of synchronization. This model relies on being able to resynchronize the entire store (between two instances only) based on timestamps.

Application reconciliation This is the Amazon Dynamo approach in which datastore-using applications do their own reconciliation. This model is **not POSIX-compliant as-is.**

2.5 Reliability studies

The specifications for most computer systems and their component parts, at least for "enterprise"-class hardware typically include theoretical numbers that often have next to no real world relevance *Mean Time Between Failure* (MTBF) is probably the most commonly known theoretical failure prediction metric[5]. MTBF statistics are near irrelevant in anything but estimating unit replacements in large populations and is hard to put to any meaningful use in estimating the lifetime of any single complex system. A key point about MTBF does make sense in general use however: a complex system with many components will have a lower overall MTBF. Redundancy lowers MTBF even further, but obviously the entire point of redundancy is to avoid enabling a single-component failure to infect the entire system.

Real-world numbers are relatively harder to come by and detailed information exist mainly for disk drives. Several research groups have established repositories with statistics[1, 2] though most of the information in these seems to be a few years old. Google with their many and very large data center are probably one of the heaviest, if not the heaviest, single consumer of disk drives in the world and have famously published a study[53] detailing their experiences. Other papers have analyzed the reliability of storage devices and statistical life times[61, 20]. A common trend seems to be that new hard drives have only a slightly elevated infant mortality, followed by low and stable failure rates in the first two years of service, after which wear and tear tends to cause strongly elevated rates of failure. Several sources have noted that RAID is no longer an excellent guarantee against data corruption on very large disks.[31, 32]

A combination of hardware and software has often been used in an attempt to guard every step along this way. ZFS' use of cryptographically strong checksums is intended to guard against byzantine failures in any component along the way. This works relatively well against LTE-type errors but less well against memory errors in non-ECC memory.[81] Correspondingly, network hard- and soft-ware

[5]Commonly defined as $MTBF = \frac{\sum(downtime - update)}{number\ of\ failures}$[78]

includes quite a lot of error correction. The ZFS example illustrates that it is a very complicated task to build a system that is error-resilient at all levels.

While hard drive errors are typically one of the least desirable types of errors due to the risk of permanent data loss, it is not the only cause of failure. Schroeder and Gibson mainly dealt with hard drive errors[61] but also included some relevant data on failure rates on other components, and have contributed another study[60] that deals with failure rates of all components in large scale installations.

3 Situation analysis

This section will expand upon the high-level criteria established in Section 1.1.1. In particular I will investigate MiGs' file access pattern, how to map MiGs structure requirements to an underlying storage solution and determine MiGs requirements for resiliency and load-balancing.

I will also look more generally at expected failure rates. Together with the MiG data, this will form a baseline for determining "the common case" that we will optimize for, and the hopefully "rare" case that the system must be resilient against.

3.1 MiG briefly

All human-oriented access to MiG is done through HTTP commands. Most operations involve some degree of behind-the-scene state changes, submission of text files etc. while others involve interaction with the users' own files, or the files of the VGrids whom the user is associated with.

The "human-action" based interface is separate from the main operations of the grid program (grid_script.py), which handles all interaction with resources. It is the access patterns of this that we are primarily going to take a closer look at later on. The human-interface also accesses, reads and writes files but such access is dependent on the types of jobs submitted and is largely predictable (i.e. a user upload of a 1 MB file has no unpredictable or unknown side-effects). It is therefore best to create a realistic and comprehensive test for this case.

Figure 2 diagrams the interaction from both tracks.[37]

All state-keeping in MiG is done through direct file access; there is no external database.

During operation, the grid server will both conduct various bookkeeping tasks, and it will also send and receive job input and output to each active resource unit (see Figure 2). The size and amount of data transferred between the server and associated resources in either direction, and in turn to the users of the grid, is external from the web interface and the grid run loop. It is also nearly exclusively dependent on the types of jobs being run on the server. While the trace was running on the grid server, the load on the grid server was light in both number and size, leaving a relatively close to baseline view of the server I/O traffic.

Varying types of jobs will have correspondingly varying impact on I/O. Some jobs may require a small amount of data as input but yield orders of magnitude larger result output, or any combination hereof. Testing the impact of varying workloads is best done in a controlled environment with a complete implementation.

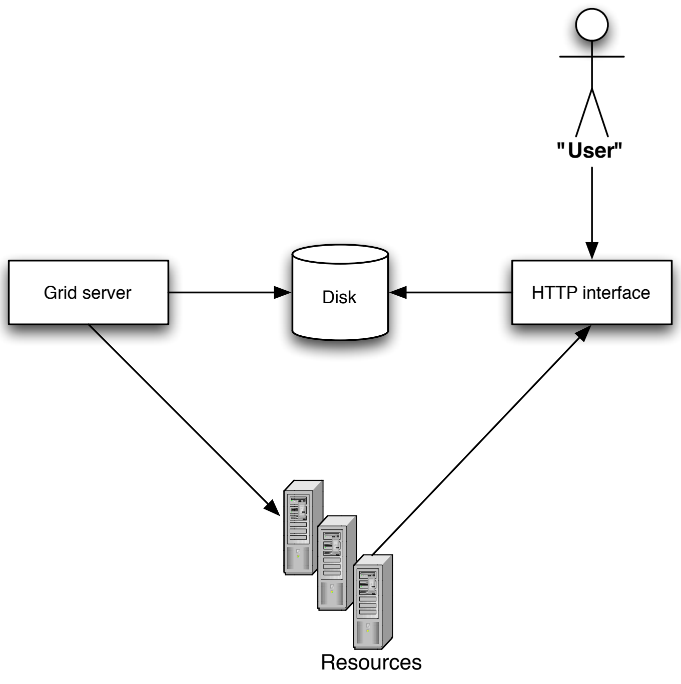


Figure 2: MiG server general job data flow

3.1.1 File locking and concurrent access

The server operations in MiG are designed to be sequential (barring design or code errors). The only place where MiG locks files on the filesystem level is in the interactive web editor by calling Python's implementation of fnctl. Aside from that, MiG falls back on standard POSIX semantics - in the words of the developers "*there is no safeguard against that users can shoot themselves in the foot*". POSIX file locking is by default advisory only[6], and thus it is entirely possible for multiple processes to write to the same file at the same time.

Integrity handling It is common for Unix applications, and MiG, to try to ensure file integrity by using the common *open-write-close-rename* pattern in an attempt to assure atomicity of an update operation. Note that without a pattern more along the style of *open-write-fsync-close-rename*, there is no expectation of integrity - there can be several seconds of delay between flushing of buffers, and until then the contents of the new file can be lost. Moreover, this is complicated further because the actual results vary between filesystems - apparently, *ext4* in default mode requires the *fsync* call to be directed at the directory instead of the file.[65]

[6]Newer versions of Linux support mandatory locking though this is not POSIX standards-based. MiG does not use this feature.

The way the underlying local filesystem is designed is an tangential issue, but it is relevant to discuss for the distributed filesystem later on.

3.1.2 Clustering in MiG

At present, there are no clustering at the grid server level for MiG. The lack of viable synchronization between two or more MiG servers mean that the server is a single point of failure.

3.1.3 Data organization

MiG stores users files in either their home directory or their vgrid directory, if shared. Both are simple hierarchical tree structures and thus isolated from other users/vgrids. These, we term *user data*. Resource states, job scripts etc. are, however, bundled together in one separate, and incidentally quite large, directory (mig/state/...), regardless of which vgrid they are associated with. These are *state-keeping* data.

3.2 I/O patterns

In order to obtain detailed information on MiGs filesystem access, I used the strace utility to monitor I/O operations from the main grid runtime grid_script.py on a normal workday, with about 40 active resources and 60 inactive. The resulting log file was 400 MiB in size, with 2.3 million independent file system accesses. A custom parser and analysis program was developed to aggregate data and statistics at a level of detail where it is still useful. Tracing directly on the main server means there was no need to create a fully functional grid infrastructure, simply for the purposes of creating a realistic environment for load-testing. Tracing only the grid script excludes the unpredictable job input/files; these files are either pushed out by spawning an external process or pulled through the web interface.

A traditional “folder[7]/file” filesystem is the by far most common and also used by MiG. The structure is essentially an directed acyclic graph[8]. The analysis program retains this general organization but simplifies it slightly by not “making the link” when encountering a hard or symbolic link, thus turning it into a pure tree structure. All accesses to a file are then added up into separate counters for each system call. Read and write operations are tracked by the number of independent calls, as well as the total number of bytes read and written.

Bytes read and written include only data passed between the filesystem layer and the application. Filesystem bookkeeping is not included, such as changing of timestamps etc.

[7]Directory can be used interchangeably with folder

[8]When going by the normal definition of a symbolic link as an *alias*, and for a system where directory hard links are not allowed.

MiG startup is two-phased: the software scans its data directories during the initial startup to determine whether there is any work left outstanding which must be picked up; the second phase is the actual operational phase. The first of these phases is extremely heavy on small disk accesses: out of 2.3 million logged accesses during a six hour log run, ≈ 1.6 million, or 70%, took place in the ≈ 10 minutes of start-up. That skews the log information quite heavily and since the start-up is a once-off cost per run, over the lifespan of a MiG server process it will trend towards an increasingly lower part of the full cost of all I/O. All further analysis will therefore deal only with operational data. Since the trace is from a production server, it was by nature not a controlled test: the exact number and footprint of resources and jobs was not fixed. However, using this trace for further analysis is much preferable over pure guesswork.

I mentioned earlier that all state-keeping in MiG is done with raw files; all these files are currently situated in multiple collective folders below the ˜/mig/state folder. All of the attached VGrids share the same state folder. Information under any other ˜/mig/[ˆbranch] folder is accessed only by the grid server, and is generally persistent. Figure 3 shows the major branches where MiG conducts file operations. Our main interest lies primarily in with MiG state while MiG non−state is only of passing interest. Other represents The remaining three branches can be disregarded in this context.

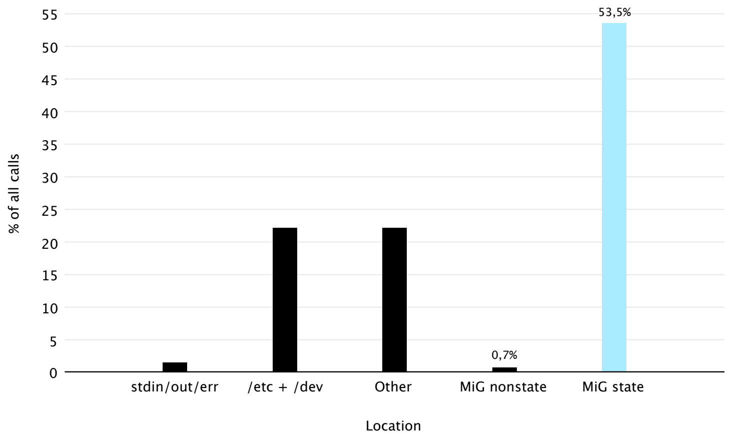


Figure 3: Distribution of system call targets

Figure 4 shows the numeric distribution of individual I/O related system call that were invoked on the state directory and below, as well as the percentage ratio of read-only vs filesystem modifying operations.

Figure 5 shows the ratio of read bytes vs write bytes, averaged across all reads and writes. The somewhat unusual pattern reinforces our expectation that MiG generates more data than it reads.

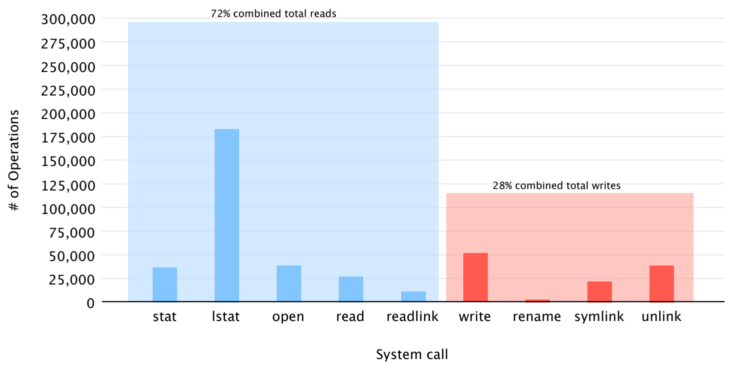


Figure 4: Count of all system calls in phase 2

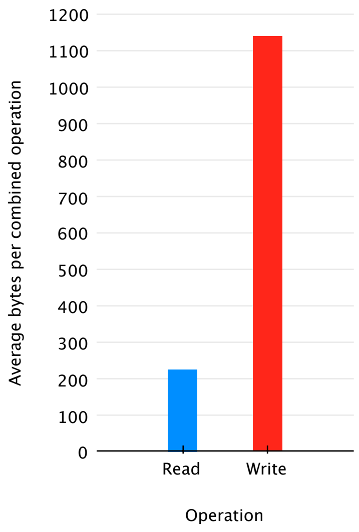


Figure 5: Average bytes transferred during (combined) read and write operation

3.3 MiG requirements

Aside from collecting aggregated usage statistics, we also look at some of the other requirements, such as reliability, disk space and dependability.

3.3.1 Adapting MiG to redundant storage

The main requirement that the MiG group has for the storage subsystem is redundancy and there is wide flexibility regarding how it should be achieved.

Comparing the two main models, primary-backup vs multi-master, the former option wins out in terms of simplicity and minimum overhead while multi-master is more flexible, and offers built-in load balancing. MiG is intended to be able to scale in the amount of servers, so some form for load-balancing is required. There are alternative strategies that will enable a primary-backup paradigm to support load-balancing without incurring the overhead and increased complexity.

Staying within the POSIX model is advantageous because it allows us to immediately move data to the storage subsystem without having to change anything. If a non-POSIX compliant model is used, the storage system would not be useful for general purposes (as an example, a future expansion of letting end-users use this system rather than the original migfs would be difficult) and current MiG filesystem access code would have to be changed to fit the new model.

3.3.2 Reliability and resiliency requirements

MiG has high reliability and consistency requirements, and a core requirement is that jobs are never lost. The user might argue that they would rather not lose files either, but the main difference is that the user is not restricted from damaging their own files in MiG. It must not be possible to corrupt MiGs own state-keeping files in the same manner.

Effectively, this means that consistency requirements cannot be relaxed below those of POSIX that MiG currently relies on.

3.3.3 Disk space requirements

The following is a list of MIGs current disk resource requirements.

```
86G /home/mig/state
4,0K /home/mig/state/.faubackuprc
6,6M /home/mig/state/gridstat_files
2,8M /home/mig/state/mig_system_files
4,1G /home/mig/state/mrsl_files
48K /home/mig/state/mRSL_files
4,0K /home/mig/state/README.txt
48K /home/mig/state/re_files
48K /home/mig/state/RE_files
248K /home/mig/state/re_home
4,0K /home/mig/state/relink.py
48K /home/mig/state/re_pending
48K /home/mig/state/RE_pending
5,7G /home/mig/state/resource_home
1,5M /home/mig/state/resource_pending
588K /home/mig/state/sandbox_home
52K /home/mig/state/server_home
648K /home/mig/state/sessid_to_mrsl_link_home
116M /home/mig/state/sss_home
56K /home/mig/state/.svn
65M /home/mig/state/user_cache
64G /home/mig/state/user_home
128K /home/mig/state/user_pending
6,8G /home/mig/state/vgrid_files_home
3,3M /home/mig/state/vgrid_home
827M /home/mig/state/vgrid_private_base
321M /home/mig/state/vgrid_public_base
```

```
3,5G /home/mig/state/webserver_home
26M /home/mig/state/wiki
150M /home/mig/state/wwwpublic
```

We can see that the user_home folder contains the largest aggregate amount of data, however we know from previously that this folder is further split into individual users' files. These are not shown for anonymity's sake.

3.4 Dependability analysis

It should not be controversial to state that any given component in a system can fail more or less unexpectedly. A distributed system must be able to survive such a loss before it can be classed as reliable and we must expect to pay a cost, both in terms of handling recovering from the incident itself and also i.r.t. what effect it may have on the remainder of the system.

In order to be able to reason about the reliability and the effectiveness of a fault-resistant design, it helps to have some idea of the frequency of critical errors. Unfortunately, because of the lack of a sufficiently large population when it comes to MiG itself, we cannot draw any conclusion from here, as we did earlier with I/O patterns. I reviewed some reliability studies in subsection 2.5 which deal mainly with hard drives in large populations.

There are many other failure points for a entire system, such as memory, power supply, motherboard, processor or network links but these are primarily solid state components and tend to experience fewer failures due to wear and tear. A unique point about disk failures is that they are the most likely cause of data corruption; a disk may fail entirely (fail-stop), or it may experience latent sector errors (LTE) or even transient problems. Stop situations may also be caused by intervention by humans, such as for maintenance.

A physical MiG server is typically situated on a RAID array which adds even more complexity and also means that we care less about the failure rate of a single drive, but rather about the *uncorrectable error rate* of the RAID array, how long it takes to replace a failed part of the array, and of course any component in the entire machine and its network connection, or even the frequency of maintenance.

All of our data are based on large populations which make it chancy at best to apply to a single server. The best that all of this can give us is therefore a rule of thumb as to how often we should expect to need to pack up and do our business elsewhere for a while. Having a high meantime between failure does not necessarily mean that the system cannot fail twice within a short period of time, but merely that in the aggregate it is less likely to do so. Consequently, the reverse is true for a low MTBF and unless it is unreasonably low, just one failing component may hinder performance but should not cause the entire storage system to fail. An option for groups that contain members with a low MTBF expectation is to increase the degree of redundancy.

For now, I define the expected interval between stops of any kind to be one month on average. We do not use this estimate directly but it is useful to have for future planning.

Branching frequency The process of branching is a new feature to MiG and we have no empirical data with which to draw conclusions on the potential frequency. We can make an educated guess however by considering a somewhat similar scenario: if we were to deal with a local system that used a volume manager, or a similar technique that is more flexible than fixed partitions, then the majority of (non-system) volumes on a given system would likely be created in the early life of the system, based on the intended use. New ones are added as required - in MiG this could correspond to whenever a new vgrid is added.

The initial setup is a one-time cost and will be amortized over the lifespan of the system. On the main MiG server, there are currently 91 vgrids at time of writing, where at least 22 appear to be test vgrids judging from their naming. They are presumed to have little constant activity.

This leads to the informal conclusion that migration operations are far from frequent enough to be considered common case, yet sufficiently frequent that their creation should be simple and not block normal operation. For the sake of argument, we will assume that migration on a non-empty set of data happens twice a month.

3.4.1 Summary

During the usage analysis earlier, MiG ran $\approx$ 700,000 filesystem operations over six hours, or $\approx$ 32.4 per second. As a basis for discussion, we will naively assume that a MiG server will run for 1 month per uncorrectable failure. A similarly naive extrapolation then gives that we can expect in the vicinity of at least 8.5×10^7 operations per failure cycle - on average.

3.5 Migration requirements

Migration of a full or partial set of data can happen for two reasons: on failure or when moving a segment to a new set of hosts.

For the migration on failure requirement, the longer it takes, the larger the window for another failure to occur. If we hypothesize that a migration of the full MiG dataset is necessary (86GiB) and that the connection between a lively host and a new host is connected with a bandwidth of 100Mibit/s (and subtract transmission overhead of $\approx$ 15%), it would take 138 minutes - under unrealistically ideal conditions. In practice, this figure is likely to be several times larger. However, it is insignificant compared to the MTTF interval and should therefore very rarely be a problem. In addition to failure migration, it is possible that operators will wish to migrate independently of any failures, such as for hardware upgrades.

The cost of branching is harder to estimate since it depends on the amount of data, however, since it is done locally, it is primarily limited by the I/O speed of the node.

3.6 Summary of findings

The preceding sections have detailed a number of important points that bears repeating.

1. A typical application often, as a rule of thumb, has a ratio of read vs writes of 10/1 or larger. Scientific applications generally behave differently; both fluid dynamic simulations and gene sequencing are extremely write-heavy. MiG, before we add jobs, display the same tendency: 28% of state accesses modify the filesystem metadata or data, and data modifications are more than 5 times larger than data reads.
2. Metadata operations are fairly frequent at $\approx 76\%$ of all state accesses.
3. There are really two sets of file access patterns: MiGs internal state keeping is composed mainly of many small file and meta data changes in a flat namespace, which contrasts against the expected access pattern of MiG users of fewer but larger changes in a hierarchical structured namespace. The former could be served by a non-POSIX store. However, it would be complicated, and likely not efficient, to overlay a POSIX compatible layer over such a store.
4. Unlike user/group files, there is no organization of grid state data, which could be relevant when considering load-balancing. This is fairly easy to change however, and the MiG group has indicated willingness to do so.
5. According to the MiG group, MiGs I/O is effectively highly centered around VGrids. This could prove useful when exploring options for load-balancing and failover.
6. Reliability and consistency requirements require POSIX, or better, guarantee of data integrity.

So far, we have assumed that *read* does not have side-effects that might result in a *write* operation. POSIX requires that access to a file is also tracked (atime). There are, particular in Linux, variations of atime that are less I/O heavy such as relatime, nodiratime and noatime

4 Design basics

This section will present a system design with the primary objective to obtain a highly reliable storage solution, and secondarily to enable load-balancing for the applications that use this storage.

We draw on the original requirements from the MiG group(See 1.1.2) and the subsequent situation analysis(See 3) to arrive at a final requirements list. The related work chapter(See 2) has contributed both inspiration in general and work by Schneider and Lamport et al. in particular is the theoretical underpinnings beneath our choices of replication model and group communication.

To start off with, I will mention all of the planned key features. Each entry is then discussed in more detail later in this section.

1. A primary-backup model.
2. User-selectable degree of redundancy with three as default.
3. Closed group.
4. Dynamic and seamless entry and exit of group members.
5. File system tree can be branched and made active on another node-set.
6. POSIX compliant and mountable as a file system
7. Replaceable storage backends

A new distributed filesystem is a reasonably complicated piece of design and software. In an attempt to explain the design with sufficient clarity, I have opted to split the presentation into two parts, an introductory-level overview (this section) and a separate section where the nitty-gritty details are explained (next section). The overall architecture is illustrated in Figure 6.

Since all things should have a name, and because "migfs" is already taken for the tool used by MiG users to mount their home directory, I will call the design for Grid Replicated Storage (GRSfs).

Terminology for this design For the remainder of this document, I will use a specific set of terms to avoid confusion. For starters, a *machine* is an independent computing entity (physical or virtual, doesn't matter). It is connected to other *machines* with a *network* - a transport mechanism with which each *machine* can exchange *data*. Each machine hosts a number of individual storage buckets, *filesystems* that can be only linked to one another with *symbolic links*[9] but not with hard links[10]. Each filesystem is thus independent from one another - most

[9]A symbolic link, or symlink, is a special file which contains a reference to another file. For the filesystem itself there is no special significance to a symlink file, but the abstraction layers above the filesystem can handle and follow the link, without requiring the applications developer to take any special action.

[10]a hard link is a direct connection to the underlying data of a file. Accessing a hard link will directly access the underlying data. Hard links between directories are not allowed on modern systems to avoid creating cycles in the file system structure graph, nor are hard links between different filesystems allowed as they do not share the same namespace or physical space

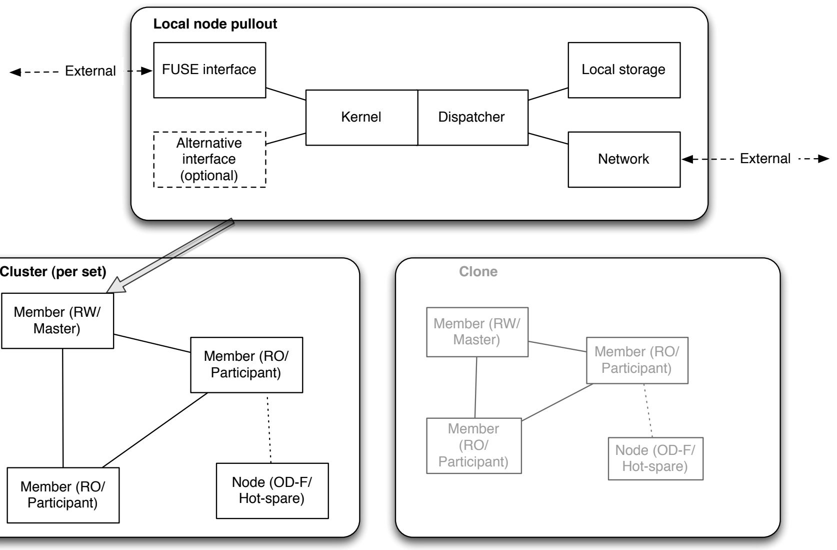


Figure 6: GRSfs design

filesystems are typically *local* and exist only on the machine in question. Each filesystem again hosts any number of directories and files, collectively termed *content*. Content consist of metadata: name, stats[11], and data: the contents of files.

In our case, we require a filesystem that is network-enabled. This network can be a local-area network, or a wide-area network, and there are other machines on it that wish to collaborate to make a failure-resilient system. When GRSfs is mounted, it will become an *member*.

There are two types of *members*; active members actively participate in replication. The counterpart to these is not passive but rather standby - a *hot-spare* that can take over if needed.

Within group theory, we normally talk about open and closed groups. Very briefly, the distinction between the two is closed groups communicate only between themselves while open groups may communicate externally. Closed groups are generally much simpler to deal with, particular as the number of participants are known and every communicating party is known to one another. Active members are connected together in a graph with edges connecting the vertices (members) as needed. Each such graph is also a closed *group*. I will typically refer to these as groups, or $group_x$ for a specific group. Each member is not bound to exist on a specific machine; indeed a machine can host an entire group by itself (although that is probably only useful for testing) but it must exist in just one place.

The term *modification* is used to describe any sort of write to the filesystem, whether it be metadata or actual data. If the filesystem was to track file access

[11]access time, last modified time, creation time, size and so forth

times, then each and every operation is a modifying call.

A group is *latent* if it is unable to offer full service at a given time.

4.1 Software stack integration

Several choices that are detailed later on are closely related to how GRSfs is situated in the software stack between userland applications and the permanent storage hardware, and this was in turn heavily influenced by the design of FUSE[12].

On a local only MiG server, all data-handling is done by standard system calls (through a layer of Python) that act on a local filesystem. By inserting a new layer before the local filesystem that obeys the same rules (POSIX), we can gain transparent replication (Figure 7). This layer builds on the FUSE package and its extension, fuse-python. The FUSE architecture can be seen in Figure 8.

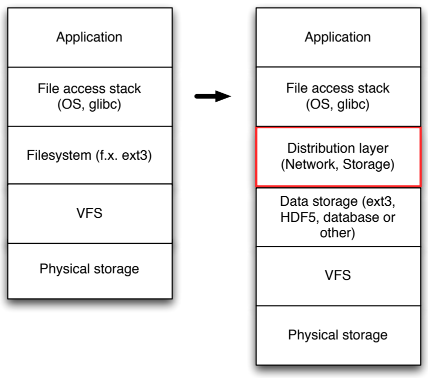

Application
File access stack (OS, glibc)
Filesystem (f.x. ext3)
VFS
Physical storage
Application
File access stack (OS, glibc)
Distribution layer (Network, Storage)
Data storage (ext3, HDF5, database or other)
VFS
Physical storage

Figure 7: From a local-only stack to a distributed

This layer (GRSfs) is, from the perspective of all overlying software, a file system in its own right. I have chosen not to implement physical level storage in this layer, as there are many existing solutions that focus, and focus well, on this particular task. Instead, GRSfs will handle replication and interface to a local physical storage solution.

Because it is a filesystem, all interaction happens through standardized channels[13]. GRSfs is not aware of the overlying software stack - this stems from the belief that any component should stand on its own and building in, say, MiG specific functionality both decreases the flexibility and simplicity of the storage component and increases the likelihood of breakage when MiG itself is changed.

MiG does need to know some details about the underlying filesystem that is

[12]Filesystem in Userspace

[13]This is not the entire truth as the FUSE front-end can be bypassed$^{(\text{See } 4.3)}$ but it is the intention that all normal operation will take place through this

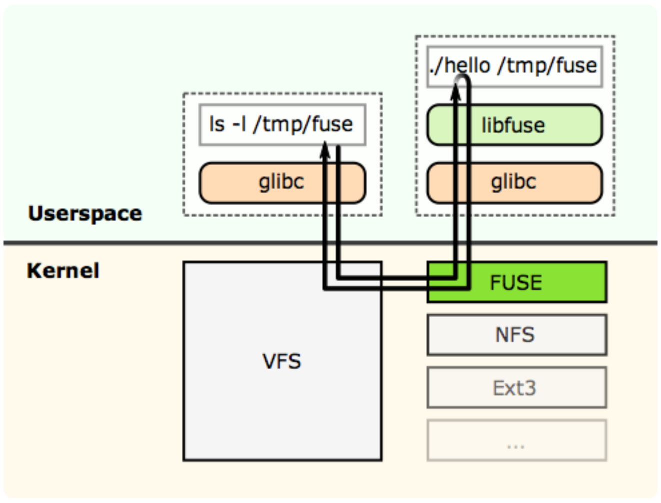


Figure 8: FUSE architecture (w/o fuse-python) ©Sven/Wikimedia Commons

not immediately available. More on this later.

4.2 Distributed interactions

If we go all the way back to 1.1.2 (p. 2), recall that MiG required a WAN capable storage solution with at least triple redundancy. The redundancy requirement is based on "a write must be persisted at at least two independent nodes". Choosing a replication factor of three allows the system to remain operational during periods of *one concurrent failure*. The implication is that the system can remain working if a node goes down until the failure condition has been rectified (pending definition on what is an acceptable operational state). An additional property is in the event that any single member (A) is disconnected from the remainder of the group B, C, then the recovery options are well-defined for each of the now disjoint groups ($group_{new} = Aandgroup_{old} = B, C$).

The logical inference of the WAN requirement is that latency conditions of greater than 10 m sec, lower overall bandwidth and less stability in the network. Given the above redundancy requirement, we can conclude that under the active replication scheme, any modification will take at least the cost of latency+transfer speed to the fastest peer.

subsection 2.4 and 3.3.1 originally discussed primary-backup (single master) paradigms against a multi-master approach. Considering MiGs high ratio of writes vs. reads a multi-master scheme would be expected to incur a significant performance penalty. Choosing primary-backup simplifies the communication scheme which in turn simplifies the implementation of the design. A master/slave does not need a distributed lock manager, or an advanced commit protocol. Many WAN filesystem designs have chosen to relax consistency in favor of performance, which is not acceptable in this particular case. Load-balancing will be addressed very shortly.

We term the write-enabled node *master* and its backups as $replica_1$ and $replica_2$. These states are not fixed; if a master goes down, one of the replicas takes over in an *election* process.

The group G is a closed group. Modifications done at a *master* are *pushed* to the replicas. Each full member of the group thus has a complete copy of all data. Data are not committed to the masters' datastore before they have been accepted at at least one replica.

I defined hot-spares earlier. Hot-spares are parts of the group but do not offer full service. They offer a partial read capability through *on-demand fetching*, i.e. by *pulling* the data. In the event that a full member of the group is no longer able to offer service, a spare can be upgraded to a full member by performing a *failover*. During the upgrade cycle, the system may continue to offer service as long as at least two full members of the group is active, though performance is allowed to be degraded.

Different error situations cause different reactions, all intended to keep the system either running seamlessly, or stop gracefully[14] if safety margins are violated.

1. If the group falls below 2 full members even though spares are available, it cannot honor the minimum replication criteria and must go into a fail-safe condition. I.e. for a three member group, we can survive the loss of one member at any given time.
2. If 2 full members remain but no spares are available to upgrade, the system will also go into fail-safe. The member must either return or the operator intervene to add a hot-spare.

This makes the system 1-fault-tolerant for the purposes of offering write services and 2-fault-tolerant for the purposes of offering read-services.

Leader election is done when required. The election process takes current progress (based on a logical clock) and "load" on each member machine into account.

Coarse-grained load-balancing is possible by letting the owner *migrate* any branch of the file system tree, and moving it to a different host. This can be done to reclaim storage space or lighten the load on any given master. Figure 9 illustrates how each branch may have a different master, or even reside on a completely different set. If a branch is moved to an entirely different set, there are two options for having it remain available, though not writeable, in the old branch; the owner can either mount it as a on-demand-fetching instance, or query the file system for information on where the branch has moved to, in a manner similar to tuple-spaces.

Figure 10 illustrates how hosts A through H are all involved in hosting instances of the filesystem. The graph of each set only has edges between involved hosts, however. Let us take the filesystem *red* (and let us say that *red* contains the contents of the *DIKU* folder, as in Figure 9). *Red* is being actively replicated from **A** to D and E. We say that $group_{red} = \{A, D, E\}$ and that B is, in this

[14]Gracefully defined as "stop but do not lose data"

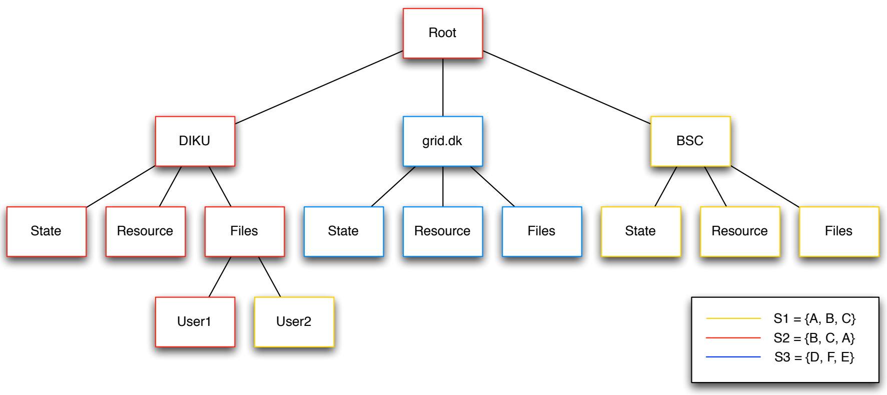


Figure 9: Different masters throughout a file system tree

case only, hot-spare for $group_{red}$. The groups $red, green, blue, yellow, black$ are disjunct - they are not related and are not aware of each other's existence. Any host can harbor any number of members or complete sets if so desired.[15]

4.3 Internal design

Each member as it reside on a machine is deliberately isolated. This means less interdependence on other components, such as a central control authority or name server, but also means it is less simple to coordinate between several group members residing on the same machine. This can be relevant when *electing* a new master, for instance.

An important consequence of the primary/backup decision along with the deliberate isolation is a fracturing of the namespace which, one way or another, ends up being user-visible. The most significant example of this is that overlying applications are expected to handle transitions to another group after a branch manually. The underlying reason for this is two-fold: first off, since we only have one write-enabled node in each group, having seamless transition between any two groups means that at any time a given directory might be read-only. This is a solvable problem since we can redirect writes to the master - at great cost in performance. This in turn leads to a more philosophical discussion regarding when and how software should strive to meet the expectations of users. To frame it in the context of our specific problem: if a given sub-root in a tree might be "owned" by a separate group, should we hide it for the user who will only be able to detect this by a sudden drop in performance relative to other folders in the same tree (and no other discernable difference), or should this be user-visible (by virtue of the folder being read-only and otherwise detectable by calls such as os.ismountpoint(...)). The latter approach is chosen for GRSfs.

[15]Operating system limitations may apply

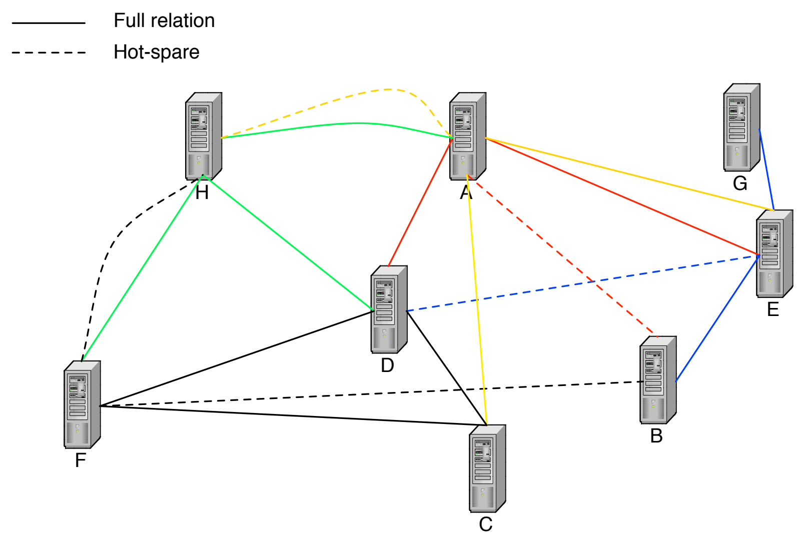


Figure 10: Multiple systems hosting multiple filesystems by color differentiation

Internal separation of metadata and data Regarding internal data handling mechanisms, the expectation is, given MiGs workload as addressed in subsection 3.2, that the best opportunities to scale are given if metadata are handled separately from associated data. This leads us to discussing how.

Given that this design is not intended to be massively scaleable, the benefits of an object-storage methodology are not clear. One notion that is worth taking along from object-storage is the separation of a piece of data and associated metadata (a file) and its physical storage properties. The current iteration of the design merely takes care not to entangle data and metadata and leaves it to future expansions to fully separate the two types of information.

Again, a major difference between this design and object based systems is that other systems heavily rely on separate meta- and data-servers whereas we combine both.

Resource usage GRSfs is intended to share the machine that hosts it with both server applications (MiG) and other instances of itself. For that reason, it needs to be well-behaved and not assume that it can monopolize all the resources of the host. This is normal modus operandi for a local filesystem of course, and mostly highlights another difference between this and a dedicated storage server.

Flexible storage-backends Internally, files are not considered in terms of blocks. This continuation of the abstraction from block devices allows for fine tuning in the network layer, as well as the use of nearly any underlying storage that supports the same storage semantics as the FUSE layer, either built-in or through adaptation. Example storage backends could be Python shelves, HDF5 or simply passing requests to another filesystem. This structure is highly flexible and can be combined with specific handling in the GRSfs layer for specific applications. For example, a vgrid might benefit from a storage backend optimized for smaller objects while others might prefer a passthrough backend. The passthrough backend is very generic but also allows easy access to data without using GRSfs which could be relevant in cases of error recovery.

Flexible frontends The FUSE layer is a relatively thin front-end to the kernel/dispatch layers (figure 6). This makes it possible for an alternative interface or even an application to effectively mount an instance in-process, bypassing the file system layers. This is out of scope for this paper however.

4.4 System local and global states

Figure 11 shows which states each member can have and the transitions between each (this is not a state diagram in the traditional sense as modeling the entire state of each member in one diagram is too complex).

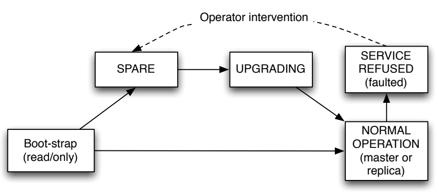
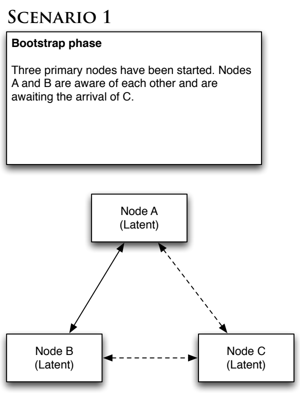

Operator intervention
SPARE
UPGRADING
SERVICE REFUSED (faulted)
Boot-strap (read/only)
NORMAL OPERATION (master or replica)

Figure 11: Local states

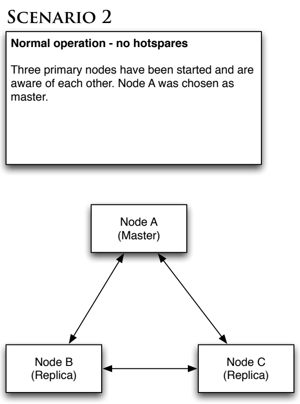
Going a step above algorithms and protocols, each group can find itself in a number of different *scenarios*. With just one originator of writes in the system, the entire model is greatly simplified. Consider Figure 12 as well as 12(a) to 12(g) which illustrates (generally) the spectrum of possible scenarios.

4.5 Summary

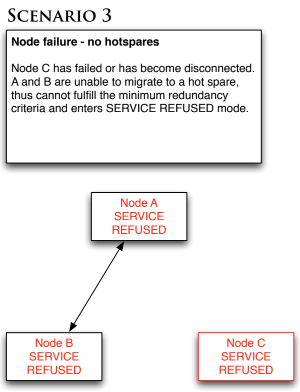
This concludes our brief introduction of the design. Many questions are deliberately left open as the next section will go into greater depth and detail from a theoretical angle. The implementation chapter deals with the practical implementation details.


SCENARIO 1
Bootstrap phase
Three primary nodes have been started. Nodes A and B are aware of each other and are awaiting the arrival of C.
Node A (Latent)
Node B (Latent)
Node C (Latent)

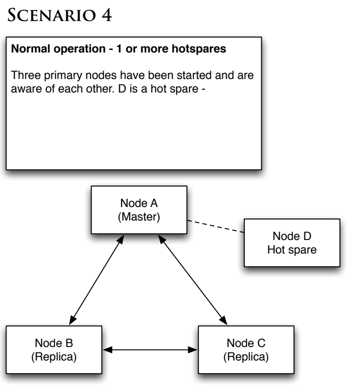
(a) Node state scenario 1


SCENARIO 2
Normal operation - no hotspares
Three primary nodes have been started and are aware of each other. Node A was chosen as master.
Node A (Master)
Node B (Replica)
Node C (Replica)

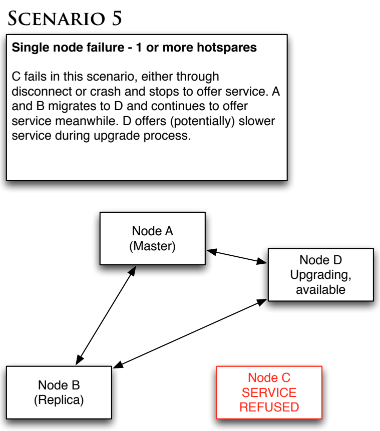
(b) Scenario 2


SCENARIO 3
Node failure - no hotspares
Node C has failed or has become disconnected. A and B are unable to migrate to a hot spare, thus cannot fulfill the minimum redundancy criteria and enters SERVICE REFUSED mode.
Node A SERVICE REFUSED
Node B SERVICE REFUSED
Node C SERVICE REFUSED

(c) Scenario 3


SCENARIO 4
Normal operation - 1 or more hotspares
Three primary nodes have been started and are aware of each other. D is a hot spare -
Node A (Master)
Node D Hot spare
Node B (Replica)
Node C (Replica)

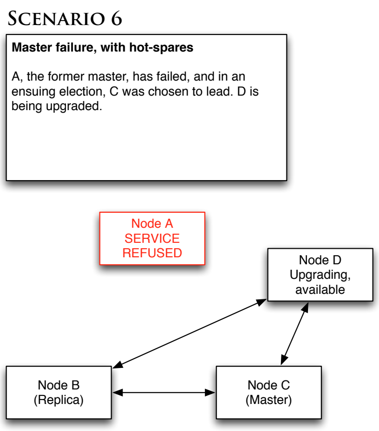
(d) Scenario 4


SCENARIO 5

Single node failure - 1 or more hotspares

C fails in this scenario, either through disconnect or crash and stops to offer service. A and B migrates to D and continues to offer service meanwhile. D offers (potentially) slower service during upgrade process.

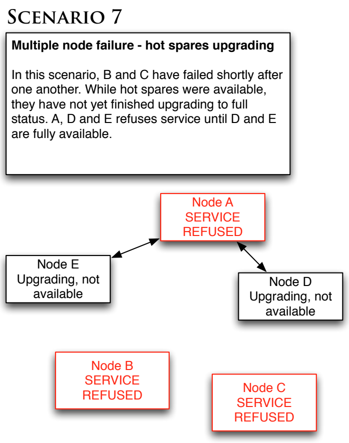
(e) Scenario 5


SCENARIO 6

Master failure, with hot-spares

A, the former master, has failed, and in an ensuing election, C was chosen to lead. D is being upgraded.

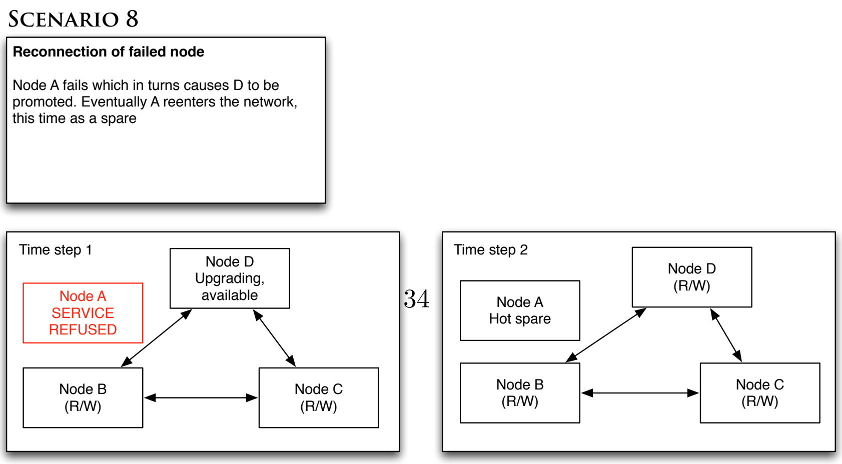
(f) Scenario 6


SCENARIO 7

Multiple node failure - hot spares upgrading

In this scenario, B and C have failed shortly after one another. While hot spares were available, they have not yet finished upgrading to full status. A, D and E refuses service until D and E are fully available.

(g) Scenario 7


SCENARIO 8

Reconnection of failed node

Node A fails which in turns causes D to be promoted. Eventually A reenters the network, this time as a spare

5 Details and implications

The remaining questions from the previous sections can be divided into three main sections, *Group management*, *Communication*, *Local choices* and *Expected efficiency*. Lastly, I will briefly detail some *Thoughts on security*.

As has been stated several times already, data consistency is extremely important and efficiency is a secondary concern. With a bit of careful forethought however, we can make intelligent choices and avoid sacrificing more efficiency than strictly needed.

The system has differing performance requirements depending on the type of operation. Table 1 shows the relative frequency of events out of 100M operations.

Distribution	**Operation**
$7.2 \times 10^{1}\%$	Reads[3.2]
$2.8 \times 10^{1}\%$	Writes
$2.34 \times 10^{-7}\%$	Branching
$1.17 \times 10^{-7}\%$	Elections/migrations[3.4.1]

Table 1: Making the common case fast

Throughout the planning and implementation phases of this project, I often found it useful to divide operations and components into self-contained boxes. The term *operation* in this context covers actions that manipulate or are manipulated by the other members of the group. Some operations may themselves be divided by one or more smaller operations. The other term, *components*, is used to define functionally separate blocks that serve a specific purpose.

Based on the requirements and previously shown scenario diagrams[12(a) to 12(g)], Table 2 shows when each distributed operation primitive and components are involved.

Use cases	DISTRIBUTED OPS	Components at work
Master read	–	FUSE, dispatch, datastore
Master write	REPLICATE	FUSE, dispatch, atomic network + datastore
Replica read	–	FUSE, dispatch, datastore
Spare read	FETCH	FUSE, dispatch, network, (datastore)
Master fail	UPGRADE, ELECT	Group mngt., network
Replica fail	ELECT	Group mngt., network
Spare fail	–	Group mngt., network
Member join	JOIN, ELECT	Group mngt., network
Member leave	LEAVE, ELECT	Group mngt., network
Migration to hot-spare	UPGRADE	datastore
Branching	–	datastore, snapshot

Table 2: Usecases overview

5.1 Group management

Group management, replication and election strategies are heavily influenced by how the group of participating processes are structured. Again, we favor consistency and simplicity. By making three simple choices, we can simplify the remainder of the design to a significant degree.

- For the purposes of internal coordination and replication, active members are organized in a *closed group.*[17]
- Each member has a *unique process ID*, the *tuple* <logical clock step, host preferability, random long>.
- The *group size* is known and limited - limited to three in this particular case.

Note that the group does not have a persistent ID though a transient id would be the set of all tuples above. We need to support the following high-level operations: GROUP_JOIN, for when a new member wishes to join the group, and GROUP_LEAVE when a member leaves.

5.1.1 Elections

Election events are very infrequent and there is no need for an optimized protocol for the sake of performance alone. However, picking a simple and straightforward approach is always good - it can make later choices easier and make it much easier to reason about the system in itself. We set up the following criterion to begin with:

1. Once elected, a master stays master unless an event causes a re-election.
2. There is no master in a latent group.
3. The algorithm must be guaranteed to terminate, be deterministic and influenceable.
4. Using network conditions as a criteria is not generally acceptable, because it can tend to cluster masters in one physical location.[16]
5. While not strictly a requirement, it is beneficial if an election results in an ordered and globally known list ([0: Master, 1: Replica1, 2: Replica2, 3+: Spares 1..n] and so forth).

I considered two main options:

Deterministic selection can be used if all members have exactly the same set of information and a deterministic algorithm to choose between them. It is

[16]An example is a group situated in SE, DK and NL. Most traffic between SE→NL passes through the Danish internet exchange, which will mean that DK has the lowest latency and in some cases possibly also the only link. That is ok for one group but for a given set of servers that might want to offer a dozen or more groups will end up with the DK server being master of all.

necessary for this hypothetical algorithm that it cannot collide (or that there is a method to resolve a stalemate).

Bully is a well-known algorithm[26] for leader election. In our case, we use the unique identifier earlier as process ID. The bully algorithm suits our requirements quite well. By using the logical clock as primary key we can select a new master if required, i.e. at startup one node has the logical clock set directly to non-zero positive value and will thus win the election. The two remaining pieces of information in the tuple means that in the case of a stale-mate, we can select the best host based on some set of objective criteria.

Host preferability If the logical clock is in-step, we fall back to finding the best host available. As hinted above, we prefer to distribute masters relatively evenly. There is no global coordinator coordinating all the participants; it is of course possible to poll other active members on a machine and select the least loaded (which should help with an even distribution). However, a machine that hosts few groups is not necessarily a good choice as master. Other options include pseudo-randomly electing the next master. In the aggregated average case that will give an even distribution. In the worst case, obviously, it will not and again, a randomly selected machine is not necessarily the best choice.

As an alternative, we will use Equation 1 to get an evaluation balanced between available disk space vs I/O performance. Disk is the amount of free space on the target storage location (in GiB) and IO is an indication of I/O performance on the same location. The constants are arbitrarily chosen in this example.

$$\frac{max((min(disk, 200) - 50), 1.5)}{150} \times IO \tag{1}$$

One idea that could be explored further is whether the preferability algorithm should be biased towards the existing master. If two hosts are at the same clock step and nearly the same level of performance, it is probably preferable to not reconfigure the entire group.

5.1.2 Resuming

The logical clock step is persistent. If the group is taken down, we can restart it and it will resume from where it was. This functions as an *integrity check* as, rather than validate every piece of information in the data set against an authoritative copy (which can be a majority of the group or operator-chosen), we assume that no changes have been made to the set if the logical clock is in step.

5.1.3 Spares

Spares are not bound by the tri-member-group model: there can be any number of spares (though this mechanism is not intended to support a vast number of actively communicating spares). Spares are organized simply as a prioritized list where the logical clock step is disregarded, i.e. the best spare is next in in line to take over from a missing member.

5.2 Communication

Having established how groups are laid out, we look at how to communicate between members.

5.2.1 Replication strategies

Before we talk too much about replication strategies, let us define a few abstract primitives. COPY is a primitive for replicating one item to one replica. REPLICATE is a full replication operation, start-to-finish for both replicas.

Keeping the application stack informed For a given application above the file system layer, it is necessary for it to know whether a write succeeded or not. The definition that an application can use to determine this is:

1. If an operation that should cause a replication returns, then the replication was successful.
2. If such an operation does not return, the replication may or may not have succeeded. The application is responsible for any further handling at this point.

Because we cannot expect the operating system (and Python) on each member to flush buffers simultaneously, failing to flush the buffers on all three members after each replication can lead to inconsistencies.

Sequential active replication The simplest and "safest" replication strategy is to actively replicate all modifications as they come in, in-order, and consider a write successful based on the criteria just above. Thus the cost of REPLICATE is

$$copy(replica_1) + copy(replica_2) + commit()$$

The cost of each individual COPY (and by extension REPLICATE) depends on the choice of commit protocol. Read operations are delayed until the write has been locally committed. This is the most conservative option.

Delayed writes On the other extreme end of the spectrum, replication can be almost entirely decoupled from the modification-causing system call. Modifications can be batched until a *high water mark* is reached or a buffer flush (fsync) is forced. This model is significantly more complicated as not only must local reads also check the batch buffer, but our success criteria are now less clearly defined - only during a buffer flush can errors be reported back. In this model, in the best case, a modification is simply added to a local buffer and costs $o(1)$

while an fsync or worst-case modification results in $n \times replicate$ for a cost $O(n)$ (where n is the length of the buffer queue) though overhead will be less when replicating several writes at once. A problematic aspect is that we risk paying the entire replication cost at an unknown time - for a fully performant solution, it is probably beneficial to let high water mark flushing run in parallel.

Conclusion The conservative option is actually significantly more stable than the semantics of a local filesystem - unless the application user forces a disk buffer flush with *fsync*, the operating system only flushes buffers rarely.[17]. It can be noted that the simple option can be a good starting point, for practical reasons, and that the batch model can be identical to the sequential in the case of a high water mark of just 1 write operation.

First I will describe how to implement the conservative option. The advanced decoupled solution is discussed in section 7.

5.2.2 Replication algorithm

To chose between a number of options when it comes to picking a replication algorithm, it must first be clear which properties are relevant. We assume that all members are able to communicate with one another, that we are operating on a reliable (ordered) transport mechanism, that we have reliable storage at each instance and that members can disappear without warning.

- must be able to survive loss of coordinator (master) without loss of committed data
- must be deterministic
- must not lead to dead-, live-lock or race conditions, obviously
- all communication must be ordered to satisfy the ordering requirement[10]$^{\text{(See 2.4.2)}}$
- should minimize required cross-traffic given the above constraints

12(d) to 12(g) visualized the three possible failure scenarios that the replication algorithm must be able to handle:

1. One or more replicas fail but the master member remains active and no data is lost. Spares will be upgraded if available.
2. Master fails simultaneously with a replica - only one replica is still active. In this case, the remaining instance becomes the master$^{\text{(See 5.1.1)}}$ and by our definition the authoritative copy.
3. The master fails and the replicas remain online. They must reach consensus on the status of any received data (and elect a new master) and it is the topic for the remainder of this section.

[17]Buffer flush delay varies greatly between filesystems and operating systems and can in some cases be as long as five minutes. As an additional complication, data and metadata do not necessarily flush with the same interval

Logical clocks We discussed commit protocols earlier(See 2.4.3). In our model with one coordinator, a logical clock is sufficient to order messages.

1. A system call involving a write is invoked on the master (other nodes are read/only). The master increments the logical clock and adds this counter to the message.
2. Master starts a write to replica$_1$ and replica$_2$ simultaneously.
3. Each receiver updates their own internal clock to the received value.

This process, along with the result of a master failing is visualized in Figure 13. **A** starts off as the master member and a logical clock of 0. As the writes come in, **A** increments its logical clock and replicates the writes to **B** and **C**. After two writes, **A** fails and **C** takes over after an election(See 5.1.1). The diagram is simplified by choice and does not include:

1. The acknowledgement from each replica is not included.
2. Updating the masters' data store is delayed until after the replicas are updated, in case of catastrophic failure
3. If the master fails after transmitting to replica$_1$ but before transmitting to replica$_2$, the replicas are out of sync. This will be covered in a moment.
4. The spare-upgrade process for **D** is not shown.

An *election* functions as a barrier, ensuring that a write operation can only originate from one member.

Figure 12 is a simplified break-down of the steps that a modification passes through on an instance.

5.2.3 Failure recovery

Failure handling can be sliced into three distinct phases

1. Detecting failure
2. Ensuring system remains in a consistent state
3. Taking appropriate steps for continued operation or controlled stop

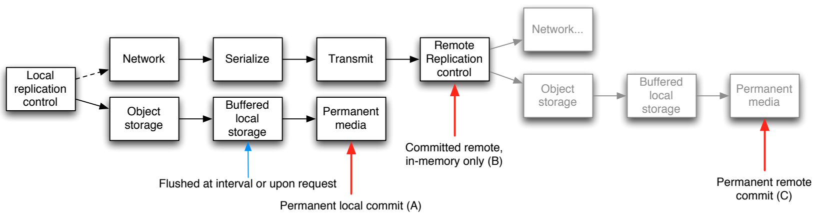

Local replication control
Network
Serialize
Transmit
Remote Replication control
Network...
Object storage
Buffered local storage
Permanent media
Object storage
Buffered local storage
Permanent media
Committed remote, in-memory only (B)
Flushed at interval or upon request
Permanent local commit (A)
Permanent remote commit (C)

Figure 12: Break-down of the steps involved in storing and replication (1)

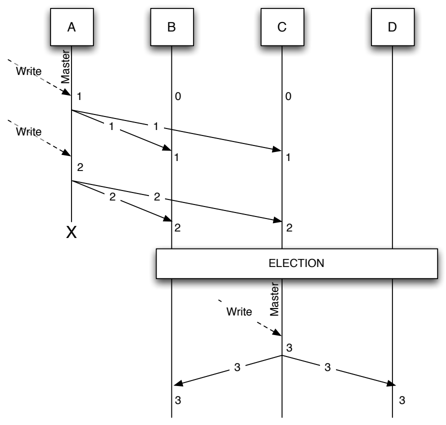


Figure 13: Progression of the logical clock on each member and failover

Phase I A member is *lively* if it responds with an OK before the connection times out. The *timeout* is defined within the replaceable network backend. A member that is not lively, is immediately replaced.

To be able to detect whether a node has failed, a ring is established that ensures that each member is responsible for checking the health of one other member. For example, the master can watch replica 2, replica 2 watches replica 1, replica 1 watches the master. This system is designed to ensure that a master cannot fail silently and leave the system down for longer periods of time, since the replicas rarely initiate communication to the master on their own. It is acceptable to have some delay before failure is detected since the system is designed to allow for a 1-fault failure condition. Aside from the heartbeat mechanism, any initiated communication between two members is also an opportunity to detect a failure. If the master detects that too many replicas are unresponsive it can fail gracefully.

Spares are not proactively watched for liveliness, nor do they watch any members.

Phase II If a single member fails then the remaining two instances must recover and handle two separate issues: election and synchronization.

Synchronization is necessary if the master and each replica was not all at the same clock-step, such as if the master fails after having updated one but not the other. The member with the highest clock is the one with the newest data and needs to replicate this last change onwards.

Replay logs A log of the last n writes received. For an actively replicated and synchronous setup, the replay log only needs to be one element long.

Post-recovery consistency check Requires a full traversal of the entire filesystem at re-election time.

While the latter option technically suits the *common case fast* criteria, synchronizing many thousand of files and gigabytes of data over a WAN connection is never going to be particularly fast.

GRSfs uses a replay log of length 1 to store the last received modification.

Phase III The next step is to upgrade a spare. Because we consider one active replica to be sufficient for a time-limited period, it is equally acceptable to continue to offer read/write services for the duration of the upgrade.

The requirements for the migration operation is based on SNAPSHOT$^{\text{(See 5.2.7)}}$.

Failure to continue In case it is not possible to upgrade a spare, or two members out of three are down concurrently, the question presents itself what to do with the remaining member(s). We could continue to offer service while waiting return of a member or spare, but this breaks the minimum replication requirement. The alternatives are to go into either a limited or full fail-stop state. In both cases, the system must gracefully ensure that nothing is "hanging" - in the limited fail-stop state, the system can still offer read/only access while the full fail-stop will shut down entirely. I find the limited fail-stop preferable, both because it corresponds to the *latent* state that the system is in at start-up and because it allows easy access to data, which might otherwise be problematic (depending heavily on which type of backend is used). The down-side is that there is no way to signal a partial error condition through the system calls interface. Without this option, we must use another option to signal problems, see subsubsection 5.3.2. The full fail-stop option is a somewhat heavy-handed solution to this problem, but obviously applications cannot fail to notice if the filesystem stops operating.

5.2.4 Clock synchronization

The physical clocks on different physical machines cannot be assumed to automatically be synchronized (the opposite is true more often than not). This will lead to situations, if uncorrected, where the timestamp of replicated updates (ctime, mtime) is entirely unrelated to the physical clock of the recipient member. Clocks can be synchronized by a highly reliable external source, implementing internal synchronization in the filesystem itself, or by extension of the latter, correcting for drift when updating timestamps.

This is not an issue as long as the group is stable but it can be problematic for applications in a sequence of events where control is moved from one machine to another with significant drift. In such a situation, the timestamp of files can no longer be used as an ordering indicator.

However, the problem is not unique for distributed application. Local-only applications face the same problem, as the machine clock may be changed at any given time and applications should not rely on the timestamp of files for critical work, for that same reason. External clock synchronization is the recommended solution to the problem, mainly to avoid confusing human users.

5.2.5 Data transfers

Transferring small files is best done in one portion while larger files can be split into smaller parts. The user-land applications will normally issue requests in limited chunks, often a few kilobytes at a time. FUSE has no concept of streaming data but the various submodules are free to employ potentially performance-enhancing strategies such as pre-fetching. For example, a candidate for pre-fetching could be the ls −l example invoked earlier[2.4.2].

5.2.6 On-demand fetching

Spares can establish a connection to a member of the group and fetch metadata and data directly. In the initial version, this is a read-only link. Data fetched from the connection can be cached on the spare to speed up ensuing operations and prepare for an upgrade to active member.

5.2.7 Snapshot

A *snapshot* is the process in which a consistent copy of some set of data is made. The SNAPSHOT operation is defined as follows:

1. A node in the file structure tree on source α is specified as the root τ.
2. τ and all descendants (τ^+) are copied from α to a destination δ.
3. At the time of finish, $\frac{\tau^+}{\delta} = \frac{\tau^+}{\alpha}$. The implication is that either no changes can be made to either $\frac{\tau^+}{\delta}$ or $\frac{\tau^+}{\alpha}$ during the process, or that the changes made synchronized before the process exits.

Snapshotting can come into play in several places, either locally or distributed; this section merely describes which criteria the snapshot operation must fulfill. A global snapshot algorithm, such as [13] by Chandy and Lamport would be an obvious choice for the distributed snapshot cases.

5.2.8 Relaxation of POSIX compliance

While we have otherwise considered it important to have a high level of compliance with POSIX, some practical considerations occasionally interfere. One example is the topic of the mandatory atime - strict adherence will effectively turn any read into a write. Additionally, only the master is even able to update atime leading to inconsistencies in behavior between group members[18]. Because of the performance implications, and because MiG does not require it, atime will not be tracked. Instead, it will be updated to the value of ctime/mtime when these are updated.

[18]Beyond the obvious read/write vs read/only, some applications have historically depended on atime. For our, more limited, scope we will ignore this particular class of applications

5.3 Implementation architecture

5.3.1 Module interfaces

Both the network and the storage module are designed to be replaceable. As such they must be generic and follow a strict interface. Any implementation specific logic must be encapsulated within the module; an example here is whether to allow concurrent write access to the datastore.

5.3.2 Control

The user-land applications (MiG) must be able to get information about the current state of the system out to handle the branching notion. Additional information may also be interesting, such as performance statistics (though this is out of scope for this paper)

There are two primary models for the first part: push or pull. In the push version, there would be an interface where applications could subscribe for new information. In the pull versions, applications must periodically poll.

Push is advantageous in cases where immediate notification is required, but pull is less intrusive and less complicated to design and build. For now, GRSfs uses a pull model which is detailed in the implementation chapter.

Concurrent writes Some data-stores may support concurrent writes, or not, but the kernel is currently sequential for writes. The short explanation is that it is simpler to test and argue for correctness when all writes are essentially sequential and reads cannot interfere with writes.

The longer explanation involves thinking about when a read or write operation may conflict with another write. Some data-stores may in themselves be fully equipped to handle concurrent writes but recall that we do not commit to the local data store until we are certain that at least one replica has accepted the modification. For data-stores that support concurrent writes, it must be able to decide if two operations can affect one another negatively and if so, handle the situation appropriately. Consider the following scenario which breaks POSIX requirements:

Step	Thread 1	Thread 2
1	Write alters /some/path, replicates	
2		Read to /some/path
3	Replication returns, change committed	

This leaves us with the choice between roughly the following options:

1. Fully sequential operation.
2. Global reader/writer lock
3. Interference detection lock

4. Shadow-copy of changes

The first is unacceptable as it would essentially sequentialize all filesystem access. The second option is much better and carries little overhead, but may block the critical path much too often. The third option only locks if it can detect interference between two options - this check must sit on the critical path of all operations but will allow a greater degree of concurrency for many workloads.

The last option is an advanced development on the interference detection mechanism and out of scope for this paper. Briefly described, it is an option where changes that are not yet fully committed at written to a staging area until the replication returns, only after which the filesystem will return the data to any read operation, for instance.

5.3.3 Branching

Branching is the process in which we take a *snapshot*, starting from some root τ. The branching process is local and is intended for use in situation where we want a new group to handle parts of our file structure tree. The SNAPSHOT operation is extended to MIGRATE:

1. Perform SNAPSHOT to a local destination
2. Leave behind a pointer to where data might be found.
3. (The operator may elect to instead mount the new group at τ, which would allow applications to continue seamlessly)

5.4 Expected efficiency

The design so far has been focused on robustness and resiliency and generally not on performance. This section takes a brief look at the expected overhead of the simple model. Potential performance improvements are discussed in section 7.

I will discuss local read performance and triple replicated write performance. The system also offers on-demand fetching but the expected performance here can be largely inferred from the discussion.

For both reads and writes there is a number of system-imposed steps (please see Figure 8) which will not be analyzed in depth for two reasons: they are unavoidable as long as we use FUSE, and the overhead is expected to be comparatively negligible.

Read operations are piped directly into the storage backend and are thus dependent on the properties of this backend - an enterprise class SSD will perform much better than an 4200 RPM laptop drive. FUSE will cost some overhead but this will vary greatly depending on workload as most operations will be limited by I/O speed (FUSE requires more context switches than an in-kernel filesystem, but this is dwarfed by the cost of actually hitting the disk).

A workload with many micro-operations may move the bottleneck to the CPU. Two additional aspects will affect the read performance:

- By default, FUSE caches some information from read requests which in some specific cases can enable a FUSE filesystem to perform better than a in-kernel filesystem. Running in the Python interpreter is most likely a greater cost than FUSE.
- A mixture of read and write requests can cause the read requests to block while waiting for the write requests to complete.

Write operations will be affected by the same basic properties of FUSE and the underlying storage but their latency are vastly more network-dependent. In order to replicate, we need to perform these steps:

1. Serialize the data into a network transmittable format.
2. Wait for write lock
3. Persist write on first replica.
4. Persist write on second replica.
5. If successful, persist locally

Step 1 is negligibly expensive, step 2 is workload dependent, and step 5 is again dependent on the local I/O speed. Step 3 and 4 are functionally identical and the most expensive (unless the group is situated on a network where network access is faster than local disk access). Latency to a remote server will of course vary greatly based on distance and conditions on the path between the two servers. Let us for a moment assume that our main server is situated at the Institute of Computer Science of Copenhagen University, the first replica is at the Polytechnical University of Catalonia (at a distance of around 2100 km on ground) and the other is at the University of Karlsruhe in Germany (950 km). Based on observations by Cheshire in [14], we should expect a roundtrip latency of roughly half the speed of light in a physical medium such as fiber, relative to distance. This article is from 1997 but latency does not seem to have improved since then. About the fastest it can ever get, based on speed of light in fiber[19], round trip from DIKU to UPC should be $\frac{2100km}{200\times10^6 m/s}\times 2 = 21ms$, but casual testing seems to indicate that practical round-trip-time is in the vicinity of 85 ms to UPC (and 40 ms to Karlsruhe).

The average number of data bytes transferred per write was slightly less than 1200 bytes. Metadata operations will typically be smaller in size. Encoding a binary write of 1200 bytes in Base64 will roughly add about 37% to the original size. To this, we must add the XMLRPC headers which varies but seems to require at least ≈320 bytes.[5] and protocol information per write which we will set to 100 bytes for the sake of argument$^{(\text{See } 6.4.1)}$.

Assuming an effective network speed at 10 Mibit s and we assume about 15% network overhead, then the transfer itself will add $\frac{(1200\times1.37)+320+100}{10mbit/sec} = 1.94$ ms. Unfortunately, Python 2.6s XMLRPC does not support keep-alive on sockets which again means that every RPC invocation must create a new socket.

[19] 66%± of speed of light in vacuum which is around 300×10^6

Our total write cost, again for the actively replicated model, is therefore $85 + 40 + 1.94 = 126.94$ ms, excluding TCP/IP connection overhead.

5.5 Thoughts on Security

Security in a given filesystem encompasses several different fields.

Access rights are mainly handled by the overlying operating system. The file system must simply store whatever data are required to handle this problem. Problems arise, however, when the OS interprets these data because what might make sense for **A** does not necessarily work for **B**. Generally, since GRSfs is intended to run on identical Linux installations, constant-type values (such as access right bit-masks) should pose no problem. User and group identifiers can be problematic unless there is a method in place to synchronize the ID stored in the system user/group databases. Several such methods are available externally.

Authentication is the topic on how to ensure that only authorized entities can access the system as a whole.

Data security covers how to protect data from unauthorized access as they are stored on disk. Typically, some form of on-disk encryption is used to handle this.

Transmission security covers securing communication while in transmit between parties.

Side-channel attacks is the term used for attacks against a system by observing information leaks. This is an advanced topic and will not be covered in any more detail here. The reader is advised to use resources such as http://www.hbarel.com/Misc/side_channel_attacks.html or http://en.wikipedia.org/wiki/Side_channel_attack as starting points.

Practical security Security-related decisions are heavily based on what kind of environment we operate in. For example, while laptop users might often use on-disk encryption, it is generally speaking an excessive cost to pay for servers that reside in a data center, at least for anything but the most critical data. We assume here that the security of the MiG servers are not our concern - only security related directly to GRSfs is in scope here.

For data security, we note that GRSfs is essentially a layer between user-land and the physical storage. It is entirely possible to encrypt data passing through but the question remains whether it makes sense. Cryptography is both expensive i.r.t. performance and requires great care to do correctly. There are already many excellent solutions available that encrypt data on disk if this is required, such as ZFS.

Transmission security and authentication can be related, as it is obvious to use PKI for the authentication step and during the process of negotiating a secure connection. This mechanic will be outsourced to the replaceable network layer which handles connection setup and transmissions. The existing XMLRPC implementation has an easy option here, as it can be combined with standard SSL solving both authentication and transmission security in one go. Another alternative here is to set up a Virtual Private Network which has the advantage of being able to work with any network backend. Both of these solutions have the advantage of not being homegrown - and can reasonably be expected to have been tested extensively by experts in the field.

No synchronization of user/group IDs is done at this point. It is not required for a network with synchronized user databases and it should be carefully considered if this belongs in the filesystem in the first place.

6 Implementation

So far the discussion about GRSfs has been held mostly at an abstract level but there are many ground-level decisions that need to be taken when implementing a filesystem, and many of these have direct impact both on understanding how and why things happen as and when they do, but also simply in terms of expectations when interacting with the system.

6.0.1 Why FUSE and Python

The two main components that this paper uses to implement the prototype is FUSE (Filesystem in Userspace) for Linux and Python. To bind Python and FUSE together, we use the python-fuse module that is freely available for Python.

It was an initial requirement that this project was implemented as a filesystem and there are two major paths to creating a filesystem in Linux: a kernel module or FUSE. A kernel module is potentially significantly faster but it is also much harder to create a fully functional implementation. FUSE incurs some additional overhead, and also has the drawback that it hides some interesting information that is available in a kernel module, such as *why* a system call was made. This information can be used for some possible optimizations - an example is the getattr system call that fetches some metadata about a given file. This call might be invoked for several reasons, and not all callers require the full set of metadata. This is, however, rather advanced functionality and out of scope for this thesis.

The choice of Python as implementation language is perhaps more controversial. Python is significantly slower than C or C++, both more common languages for filesystems. However, Python is an excellent prototyping language and given that the rest of MiG is also written in Python, it is trivial to both maintain and also allows comparison against earlier attempts to handle the same problem.

6.1 Internal structure

Figure 14 shows the internal data-flow inside the kernel and dispatchers. Note that the diagram covers only masters and replicas.

There are two paths that a request can take: a read path which is always served locally and correspondingly a write path. A replica can serve a write but only when it is received from a master instance. In the diagram, master-only steps are dashed, while external implementations are lightly grayed out. The master-only REPLICATE operation is shown in separate pull-out. Of particular note is the open file cache (mainly an important optimization, discussed in 6.4.1) and the step (logical clock) handling. The latter ensures that events are always ordered by a single source, the master, and that the logical clock step is not persisted until replicas have committed a modification successfully. Replicas in term also update their own logical clock so that is in synchronization with the master, thus allowing a replica to take over if a master fails and detect if the other replica is at the same clock-step.

Hot-spares which can only do on-demand fetching have no write path and do not serve reads locally, but invoke a remote read over RPC.

The read and write paths constitutes the critical path for all common-case operations. Operations that are not strictly related to these two tasks should be relegated elsewhere if possible.

6.1.1 Thread structure

The threading model was largely given as FUSE offers just two choices for concurrency:

1. All requests run in same thread - cannot service concurrent requests at all.
2. Each request gets its own thread. Non-FUSE threads must be started during the FUSE initialization phase.

Obviously the first choice is unpalatable. There are two obvious strategies to follow: either transfer all requests into a queue, let the kernel handle them, then return the result to the waiting FUSE thread *or* let the FUSE threads “own” the request all the way down and up through the kernel, storage and network layers.

The first option can cut down on the interdependence between components and can reduce overhead if the communication channels are single ported, but also makes it more complex to service requests concurrently, even reads.

GRSfs uses the latter alternative; serving concurrent requests is important and read requests are significantly more frequent than writes (more on write locking later) and this is exacerbated because we take requests from more than one source - any full member can also serve one or more hot-spares.

In addition to the FUSE-request threads, the network backend runs in a thread(-pool) of its own, allowing it to service more than one request at a time, similarly to FUSE.

Lastly, there is the maintenance thread. The maintenance thread is responsible for the following periodical tasks:

1. Heartbeat / pro-active failure detection
2. Pruning dead members
3. Finalizing connections to any new members that may have joined
4. Upgrade hot-spares if necessary

6.1.2 Locking

The lock mechanism for the passthrough storage engine is a global reader/writer lock. An “unlimited” number of read locks can be held simultaneously, but only one thread can enter the write lock at a given time. One thread holding a read

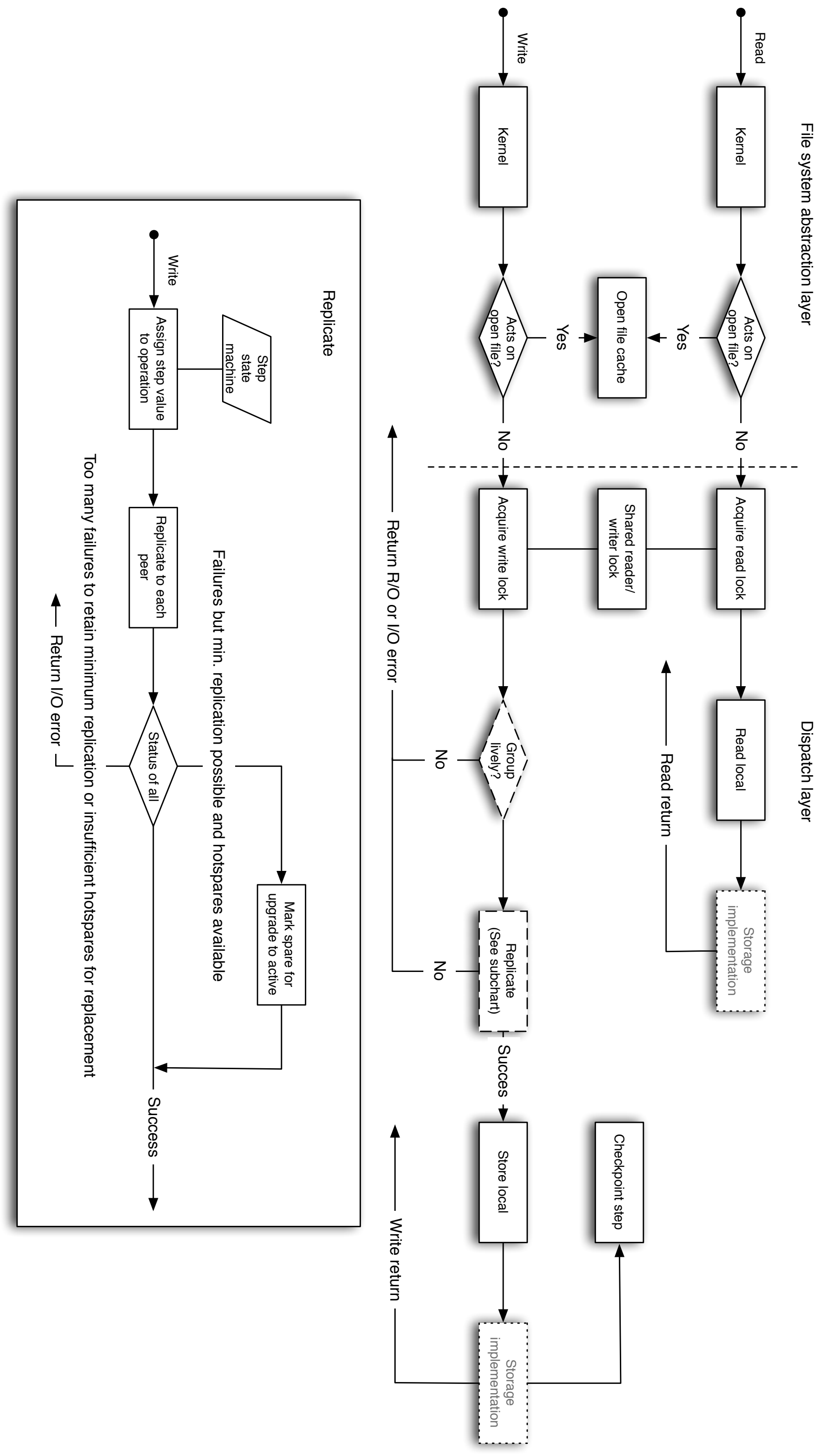


Figure 14: Read and write data paths

lock can upgrade to writer, but must exit before another reader can upgrade. Write locks are prioritized ahead of readers as well.

6.1.3 Transient statekeeping

Filesystems generally do a lot of transient state-keeping, where open files and position of read/write seeks are most immediately obvious. Other information can be kept around for performance reasons (caching, prefetching). For FUSE filesystems, this can be handled by the VFS kernel module. Taken to an extreme, a FUSE filesystem can be fully stateless.

It might not be the best choice, however. While other layers in the stack might provide for caching, a fully stateless design would not be able to utilize prefetching and it would have to incur the overhead of some operations repeatedly, such as opening a file on every read (provided that such an operation does carry overhead, of course - it can depend on the storage backend being used).

There is currently no caching or prefetching as such built into GRSfs. As Figure 14 hints, the open system call is stored in a local data structure - primarily to avoid having to reopen the file repeatedly but also to enable future caching and prefetching extensions in the FUSE layer.

6.1.4 Persistent statekeeping

The clock step checkpoint mechanism is designed to persist the stepping that each member is at, so that when it starts up again after shutdown, it can resume where it picked off (in terms of who is master).

This mechanism could potentially be expanded later on to carry more recovery data.

6.1.5 Sub-roots in a filesystem structure

Recall earlier that it was desirable to let different subtrees run on separate groups. This has a number of advantages, such as load-balancing and the ability to segregate data amongst qualified peers only, i.e. in the example shown in Figure 10 the servers that carry data for the grid.dk sub-root has no access to data for any of the other sub-roots.

This can be implemented relatively transparently, and entirely outside the context of GRSfs, by simply mounting the requisite group at the necessary location in the filesystem. For some purposes, this is not the optimal solution because the overlying user-land application would rather handle the situation it To continue with the grid.dk example, by mounting the $group_{blue}$ of $\{D, F, E\}$ in the grid.dk subdirectory from “root” we get the transparent switch to another set of servers. This carries two implications:

- The master in the sub-tree might not necessarily be the same. The user-land application must be able to handle it.
- This solution requires that the node remains connected to the set, although it may merely be a hot-spare.

Some applications, MiG included, may prefer to redirect requests onwards so that requests to subtree hosted by another group is also served by a different MiG server. To do so, it is necessary for the MiG server to know where data are moved to after the set is split off. Obviously we can carry such information inside the filesystem but unless we want to break with standards - thus potentially requiring major changes to many different components - the information must be query-able within the confines of the standard system calls.

Again, this can be handled in many different ways, such as implementing an RPC function that MiG could query or by embedding the information into the filesystem itself, in a non-standards violating fashion. GRSfs chooses the latter option by leaving behind a .migrated file with information on where the sub-root has migrated to.

6.2 Communication details

Some aspects of the communication between nodes is explained in greater detail in this section.

6.2.1 Entry and exit of nodes

Figure 15 details the process when a node contacts an existing member, by calling kernel.node_register(...) on that member and asks to join. The reverse process is also possible but it is less complicated - it first ensures that it is not in the middle of something and then simply announces that it is leaving by calling kernel.node_unregister().

6.2.2 Election details

We already discussed the bully algorithm and there is relatively little to add to this discussion. The procedure is detailed in Figure 16. It is designed to handle cases where two or more nodes simultaneously call for an election by using two flags: hold_election and election_in_progress. If a node attempts to hold an election but discovers that one is in progress, it backs off (but does not clear the election flag) and awaits a conclusion. Once an election is concluded, both flags are cleared.

After an election, the *dispatcher* may have changed. There are four types of dispatchers:

- Standalone which is intended mostly for debugging. Does no replication.

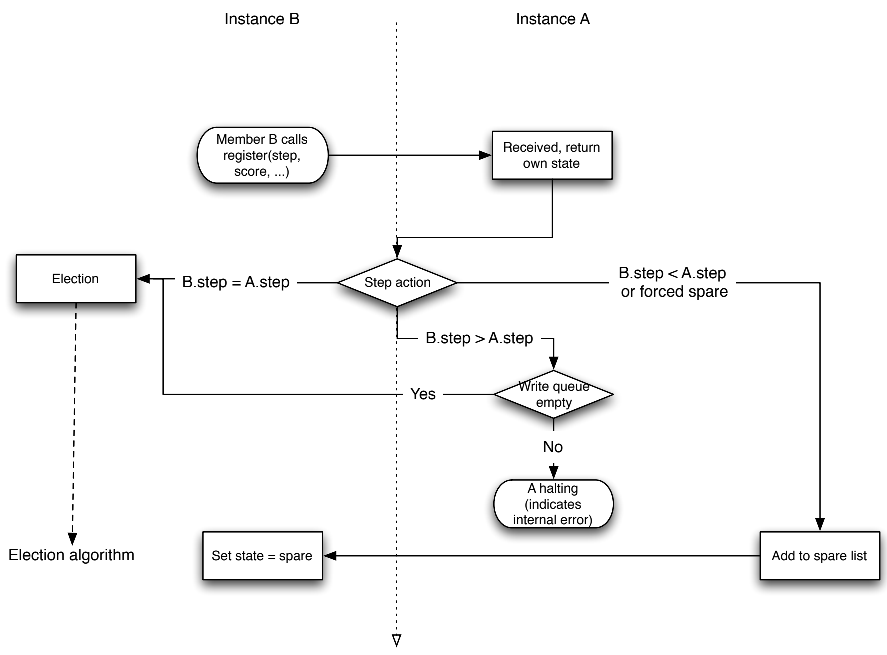


Figure 15: New node entry flowchart

- Master
- Replica
- Spare/On-demand fetcher

The purpose of each of these is hopefully self-explanatory.

6.3 Error handling

There are three main classes, or origins, of errors in GRSfs:

1. Transient errors, caused by an action somewhere in the above software stack. “File not found” is an obvious example.
2. Communication and replica integrity errors, such as losing the link with a replica or the disk running full in one or more nodes.
3. Illegal operations, such as attempting a write on a replica or hot-spare.
4. Internal errors within the filesystem.

For the first class of errors, it should be simply be propagated upwards where relevant. For the second class of errors, it generally means that the member should go into fail-stop condition. The last class is the most insidious - internal errors can lead to spurious errors in the first and second class, and is also realistically the most likely cause of data corruption (at least until an implementation has been through extended use and thus been “burned-in” so to speak).

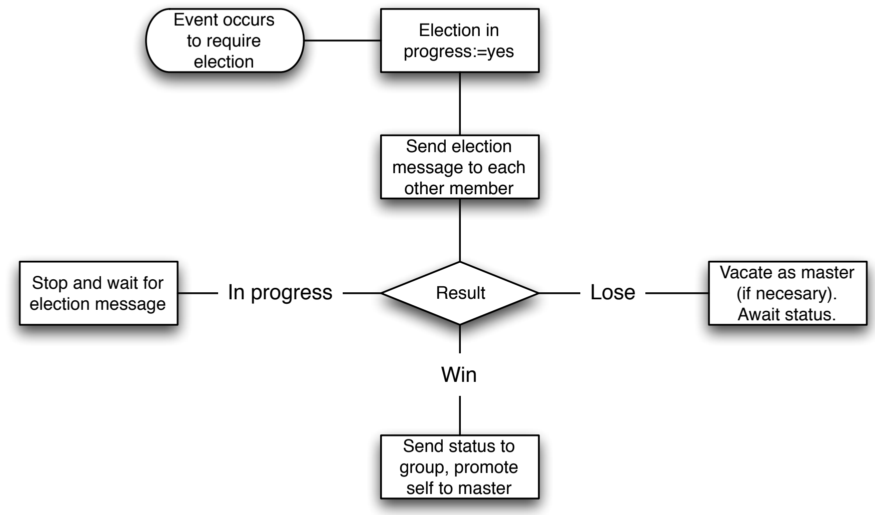

Event occurs to require election
Election in progress:=yes
Send election message to each other member
Result
In progress
Stop and wait for election message
Lose
Vacate as master (if necesary). Await status.
Win
Send status to group, promote self to master

Figure 16: Election process flowchart

There are some differences as to how errors should be handled, depending on which state a given node is in.

Transient errors

In the general case, reads on nodes in any state, and writes on the master, all transient errors are simply propagated upwards. It is noteworthy that the entire group is not guaranteed to agree on transient errors. A transient error that happens on a write replicated to a replica but not on the master is cause for concern. For now, this is considered an integrity failure.

Communication and integrity errors

Error conditions that are not transient are considered permanent and will lead to fail-stop for the node in question. This can cause the entire system to enter a state where a previously allowed operation is no longer allowed (as an example, if we have the group $group_x = \{A, B, C\}$ with no hot-spares. A fail-stop for any one of A,B or C will cause the group to reject writes).

Illegal operations

Illegal operations, in the context of just the distributed storage, is most often simply attempting to write to replicas or spares. For a group that has lost enough members to not satisfy the replication criteria (or because an election event occurred that switched control of the group), it means that a node that used to be a master may now no longer be writable.

It is possible to put any system error code through FUSE but it helps to use codes that are expected in a given situation. Returning errno.EDOM: Math argument out of domain of func is not useful as no tool or user will expect such a reply. GRSfs will use the following return codes to cover a variety of underlying problem causes.

Code	Description	Cause
EROFS	Read/only FS	Returned for writes on non-writable nodes
EIO	I/O error	On master: write operation failed to replicate. The write was not persisted on master in this case. If the error originated in the local storage backend, the node fail-stops.
–	–	On spare: unable to communicate with peer for on-demand fetching
EACCESS	Access denied	Fallback error

6.4 Protocols and network

Communication between two (or more) entities obviously requires them to agree on the language. For GRSfs, we have already stated that we operate on a homogenous platform(See 1.1.2), with no difference of opinion on low-level architectural issues such as endianness. Each host or the network equipment in place may have slightly differing opinions on optimal settings (such as for MTU or similar), but the focus is still on correctness, rather than performance.

There are multiple layers of network protocol that need to be in place to communicate. To keep the discussion on track, we will not discuss protocol layers any lower than TCP/UDP. While both of these build on the IP protocol, UDP has no guarantee of delivery quality. TCP is explicitly designed to handle such quality concerns as resending of lost packets, ordering packets correctly and will generally detect many types of errors in communications. It is not a perfect abstraction; errors can still happen and may very rarely even be missed by the built-in TCP checksum mechanism.

Table 3 lists a few common options that are, more or less, directly usable in stock Python. Pyro is probably a very good option on an LAN, but it is not designed to be firewall friendly. The remaining two options are vastly different: Pythons XMLRPC implementation is very simple and easy to get started with but it is not particularly fast. A custom protocol, built directly on the socket API, could expand on firewall-friendly protocols (such as HTTP) and only implement features that we expressly need though this may be slower than an optimized binary protocol. The main drawback is that a custom protocol/network implementation could easily consume an order of magnitude or more of resources to create and verify to the same quality standard as Pythons well-tested XMLRPC implementation, particular if it was to offer full abstraction, firewall friendliness and RPCs. It is also worth noting that it is not entirely trivial to write a truly fast network layer in Python - partially due to the many layers of abstraction.

Finally, we note both that the XMLRPC implementation is entirely sufficient to demonstrate correctness of the design and that it might be worth not looking too hard on the overhead of a single method invocation over XML but instead investigate some possible optimizations.

For these reasons, XMLRPC was chosen as the transport protocol. The major drawback (ignoring overhead) is expected to be the imperfect abstraction that XMLRPC offers on Python; the lack of ability to marshal local resources is to be expected (and not required in any case) but it also does not transparently marshal some built-in types and classes, such as tuples and exceptions. The stock `xmlrpclib` and `SimpleXMLRPCServer` was therefore expanded/configured with the following features:

- RPC Server
  - Threading enabled through the `ThreadingMixIn` to serve more than one request simultaneously.
  - The None type is declared marshalable.
  - Support for stopping the server was added, which is required for an orderly dismount.
- Client
  - XMLRPC `fault` errors are intercepted and replaced with the original exception, provided this is FUSE-compatible. This touches on the relatively involved problem of detecting the error class, see 6.3 and below.

Remarshalling the original exception happens only to Pythons `IOError` and `OSError` exception classes. For on-demand fetching, it solves one of the major abstraction leaks in the XMLRPC model since generally errors must be propagated to the user-land application which initiated the failing request the first place. For active members, the situation is more complex: some errors indicate a failure in the node itself while others can be indicative of a problem in the overlying software stack.

6.4.1 Application layer protocol

Given the choice of XMLRPC as transport protocol, many of the choices in relation to how types should be (un)marshaled are given for us. XMLRPC is based, obviously, on the Remote Procedure Call model, i.e. we can invoke functions and methods relatively transparently. This is a major simplification but it is still important to define a stable and useful interface. Recall that earlier we wanted the storage backend to be replaceable. Given Pythons high flexibility when it comes to types and function handling, we might as well aim for the same property in the network backend.

[20]Python 2.7 seems to have this feature

	XMLRPC	Pyro	Sockets
Abstraction level	Medium	High	n/a
Firewall friendly	✓	-	n/a
Ready out-of-the-box	✓	✓	-
RPC model	✓	✓	n/a
Extensible	✓	-	n/a
Persistent connections	-[20]	✓	✓
Expected overhead	High	Medium	Varies

Table 3: Comparison of transport protocols

1. Should the protocol be stateless or stateful?
2. Handling parameters that XMLRPC cannot transparently marshal?
3. Streaming data

Recall earlier our discussion of state-keeping in the filesystem which concluded that keeping state was beneficial in many regards. However, the same does not necessarily hold true to the same degree at the network protocol level.

Let us compare two possible (internal) ways to handle the typical $open \leftarrow read$ sequence as called over the network.

Stateful
$identifier \leftarrow open(path, flags, mode)$
$data \leftarrow read(identifier, length, offset)$

Stateless
$data \leftarrow read((path, flags, mode), length, offset)$

The additional overhead for the protocol is negligible in the second example. It also greatly simplifies the design. Consider the following sequence of events on two nodes. For this example, file_id is a number:

Informal ordering	Master Events	Replica Events
1	local `open for read yields file_id = 1`	
1	open `for write yields file_id = 2`	
1	replicate `write(2, ...)`	Receives `write(2, ...)`

With a stateful protocol, both master and replica must agree on what "2" signifies. Simply replicating the open for write call will not work as-is, as this requires that each node comes up with the same identifier for each replicated write. Choosing that all nodes should exchange open-file information is wasteful and runs counter to the read-only aspect of the non-masters. An improvement would be to replicate the open for write calls and replicate a common identifier along with it, whether this be master-mandated or through a mechanism that ensures agreement across all nodes. This added complexity, as well as having to replicate an open operation, can be bypassed with a stateless protocol that always carries the required information to open a file. Caching the result of the initial open call can negate any possible penalty involved in having to repeatedly reopen the file, leaving just the cost of serializing and transferring the extra bytes over the network. Comparatively speaking, this cost can be significant in terms of extra bytes per call but parts of it can in turn be minimized at the possible cost of some overhead with some optimizations.

Summarizing this discussion, GRSfs is using a stateless protocol.

To address the remaining two questions: no abstraction is perfect and XMLRPC is quite far from perfect. The lack of support for tuples, for instance, requires the overlying implementation to take care, especially because the Python/FUSE bindings and the Python file operation functions both take tuples as parameters in some instances. Fortunately, Python is very easy to work with in this regard but the issue of leaky abstraction is worth noting in case there are any future aspirations to replace the network backend.

Neither TCP nor XMLRPC supports streaming which could be relevant when transferring very large blocks over the network[21] This is not a problem for MiG when handling **state** data. However, user data can vary greatly in size and it would not be unexpected to encounter user data files in the gigabyte size range. FUSE expects that the result is returned in full, but it could still be beneficial to start a transfer, return parts of the result and still continue. Doing so would not be terribly helpful when replicating (as the filesystem cannot signal OK until it is entirely complete) but it could be a possible improvement for on-demand fetching. In practice, it would probably be implemented as a form of prefetching.

6.5 Storage backend

There are an immense amount of difference between possible storage backends and Table 4 will list only the most immediately relevant criteria. The "concurrent read" criteria refers to whether the datastore is capable of correctly serving read requests concurrently with one write request. "Independently mountable" refers to whether the datastore can be mounted as any other filesystem, i.e. does not require any special tools to access.

A few comments about the remaining listed options:

- the ACID database is mainly included as a comparison; it is certainly possible to implement a filesystem using a database as store, but the service and reliability model of the database is not a particularly good match to POSIX.
- Shelves is a persistent object serialization model, offered by Python and implemented by an underlying database library (typically a dbm or derivative, with no ACID properties)
- HDF5 and its associated Python binding, PyTables, is highly regarded as an efficient implementation of a hierarchical database for numeric values.

[21] Which can vary somewhat depending on the network. Extremely broadly speaking, handling smaller blocks at a time increases overhead while larger blocks can introduce latency

Newer versions of HDF5 offer highly flexible handling of data, and is no longer primarily fit for numeric values. It is also extremely fast on lookups and offers built-in compression.

.

6.5.1 Concurrent writes

Concurrent writes can offer some degree of speed-up but filesystem writes are rarely embarrassingly parallel. The Unix filesystem model and POSIX has several features in place to handle this. We will not get into this in detail here, but will briefly mention the requirement that all reads must reflect any previous writes and the use of a reference counter which facilitates hard links, and which from the user perspective allows deletion of files while they are open.

Other backends are generally not that flexible - in fact, most or all of Pythons single file storage containers are not thread-safe.

6.5.2 Cross-references

Cross-references in this context means symbolic links or hard links. Symbolic links are handled above the FUSE layer while hard links is part of the filesystem domain. For the passthrough storage backend, these are simply passed along and we let FUSE and Python do the validation as necessary.

6.5.3 Passthrough as default option

The MiG group expressed strong preference for the passthrough[22] option during the initial planning phase. Given that passthrough is quite a good choice in its own right, it was adopted as the storage backend for the initial implementation.

Amongst the benefits of the passthrough option are

- Simplicity
- Supports concurrency - any number of threads can write simultaneously without data corruption, although the overlying software must of course order operations correctly.
- Easier recovery in case of failure
- Highly generalized

It does however also have a number of drawbacks

- No transactional support

[22] "Passthrough" refers to a backend where all operations are passed along to a local mountpoint

	Passthrough	Shelves	Pytables/HDF5	ACID database
Concurrent reads	(✓)	-	✓	✓
Concurrent writes	(✓)	-	-	✓
Independently mountable	✓	-	-	-
Transactions	(-)	-	-	✓
Built-in cross-references	(-)	-	-	✓

Table 4: Comparison of storage backends

- Highly generalized - at the same time both an asset and a liability as it might not be suited for all workloads, although some of this can be mitigated by choosing an appropriate underlying filesystem.
- Difficult to separate data and metadata
- Cannot reorganize data without breaking the ability to easily recover
- Lack of integration with Python

To summarize briefly, the passthrough option inherits the qualities, good and bad, from the underlying filesystem. This makes it both extremely simple to implement - everything from metadata storage, over hard links to concurrent access is already implemented - and at the same time it is very well tested.

One drawback that may not be immediately obvious is that since we just pass requests onwards, we have little integration and knowledge of the underlying system. It is of course possible to build a shadow model of the directory graph, to point out one example, but seamless integration would probably simplify such a feature.

6.6 Branching

The theoretical description of the snapshot process was described in subsubsection 5.2.7. The process of branching a subtree to a different group does not necessarily require involving the storage design; a simple solution is to just rsync/move a tree to a different place. As we talked about earlier in 6.1.5 and 5.3.3, MiG prefers leaving behind information on where the branched data migrated to.

A special RPC function, branch_tree(path, local destination, forwarding information) implements this functionality. Each storage backend must implement its own functionality as the process may, at least from an efficiency angle, vary for each. The passthrough backend gets around this the easy way by calling rsync.

6.7 Migration to hot-spare

A hot-spare that is upgraded to a full member may previously have been a full member, and thus have some or most of the data that is presently in the authoritative copy. Rather than attempting to detect the set of differences between them, we simply copy the entire authoritative set of data to them and join them in the group. This in turn causes an election cycle.

6.8 Other implementation details

This section contains miscellaneous details.

6.8.1 Invoking

GRSfs is a FUSE "executable" and resembles other filesystems in the way it is started. This example mounts the filesystem on the mount folder and tells it of another peer it can begin to form a group with. −o parameters are GRSfs specific, while options such as −d or −f are handled directly by FUSE. Use −−help for a list.

```
python grsfs.py −o backingstore=/tmp/scratch,serverport=8000,backingstorestate=/tmp,
        contact=n0:8000 −f mount
```

The most common parameters are explained in this table while a terse list of all options is available below in Listing 1.

Parameter	Req.	Description
backingstore	✓	File or folder where the backing store data can be found
backingstorestate	✓	Folder where state and scratch data can be saved
serverport	✓	Listen port
contact	(✓)	Semicolon separated list of other peers in the host:port format.
logverbosity	-	Int {0,1,2,3} specifying verbosity of debug output. Defaults to 0.
mincopies	-	Minimum degree of redundancy (including self). 0 is a special case for standalone. Defaults to 2
mincopies	-	Maximum degree of redundancy (including self). More nodes than this becomes spares. Defaults to 3

Listing 1: Configurable parameter attributes

```
spare = False
heartbeat = 5
backingtype = "passthrough"
network = "xmlrpc"
backingstore = None
backingstorestate = None # directory where to store all scratch data
initial_connect_list = []
contact = None # the string version of the initial connect list
serveraddress = socket.gethostname() # autodetect the hostname
```

```
serverport = -1
mincopies = 2
maxcopies = 3
benchmark_fast_start = True # randomly finds a score to speed startup.
benchmark_min_gb_free = 50
benchmark_max_gb_free = 200 # above this the score doesn't increase
benchmark_min_free_score = 1.5
freespace_target = "." # Should be same partition as backingstore
iobench_target = "/tmp/random2mbfile" # req. for I/O benchmarking.
network_timeout = 5 # set to None for default
logdir = "./logs" #"/var/log/grsfs"
logquiet = False # whether to print to stdout
logverbosity = 0
neverparticipate = False # turns it into pure ODF
```

6.8.2 File locks

All file locks are local. It is meaningless to propagate a write lock across to a read/only member, and in case of failover, the userland applications no longer hold the same file handles. The actual implementation is copied from xmp.py and is based on Python fnctl/flock.

7 Performance and delayed writes

The active replicated model is expected to have some severe limitations when it comes to performance. This section details some alternatives that may improve on the situation.

Latency hiding Currently each replica is contacted in sequence rather than concurrently. It is possible to parallelize this but there are two issues to plan around. First off, FUSE only allows threads to be created at initialization time which makes it impossible to dynamically adjust the redundancy factor unless we create more threads for this than we expect to use. Secondly, it adds some complexity to the logical clock handling in case any member fails halfway through. This change can yield a potential improvement per replication from $copy(replica_1) + copy(replica_2) + commit()$ to $max(copy(replica_1), copy(replica_2)) + commit()$

Interference detection As mentioned earlier, it is not necessary to keep a read (or write if the storage backend supports it) to /home/my/path blocked if a write is busy only in /home/his/path. This can be tested through by employing graph coloring on the directory tree structure (or equivalent), i.e. a write that changes / or /home should block for the read, but a write that touches only the subgraph under /home/his can be allowed to proceed. Such a component would be generally useful, not just for active replication. Some system calls change the filesystem in two places, such as moves and hard links.

Negative acknowledgement protocol The current protocol is based on *positive ack* as each replica will return status during conclusion of the COPY procedure. This could be changed to fire-and-forget where the modification status for each replica is piggybacked into the response for the next incoming write.

Delayed writes This is the strategy briefly discussed in 5.2.1 and also discussed below.

The first of these two options can be implemented transparently to the user while the last option has by far the largest potential, but it is also user-visible as it changes the semantics of our filesystem from “safer than posix” to “just as posix”.

7.1 Delayed writes design

Most of the performance penalty for a write comes from having to start the entire overhead of a transfer for each write. It is possible to put writes into a buffer that is periodically flushed such as when a high watermark is reached, the user requests a manual flush or an interference is detected.

A design with a high degree of concurrency could organize the buffer to consist of separate groups: writes go into a specific group, in order. The group is then *closed* and replicated. Any following writes go into the next group and the process repeats.

Improvements from this are dependent on how many writes we can buffer up before we must flush. Flushing happens for the following reasons:

- We reach a high watermark of pending writes. Having a limit to the buffer guards against using too many resources for the buffer, taking an extreme amount of time to flush when it eventually happens, and avoid losing too much data if the system does not manually force a flush.
- When the user requests a flush by using fsync. Note the difference between flush and fsync on the filesystem level. Flushing, at least in Python, empties Pythons buffers but does not ensure that the OS syncs changes to disk. Fsync takes care of this on the other hand but is technically designed to flush just one file descriptor. When we use the term “flush” in connection with the delayed writes, we are referring to the process of copying the buffer to replicas where it is persisted.
- Due to interference. It is possible to buffer up as many file data writes to a single file as we like, but it is less trivial to handle operations that require previous data to be committed before they can act on them. An obvious example is attempting to read from a file that was just written to, and trying to move a file that was just created. In both of these cases, data must be committed, either to permanent storage or to a shadow-copy, before the operation can be completed. This last problem is likely to be the largest hindrance to performance improvements, because the following scenario does trigger interference and it is very common.

```
f = open('file.tmp')
f.write(data)
f.close()
os.move('file.tmp', file)
```

  We will discuss whether we can relax the success criteria and an alternative to flushing on interference at the end of this section.

A delayed buffer implementation should give us performance improvements in aggregate for the following reasons:

- Reduced network protocol overhead. Rather than paying in the order of 320 B in overhead per message (with XMLRPC), we can pay that just once per group replication
- Mostly bypasses the keep-alive problem of XMLRPC on Python 2.6 as the connection needs only be opened once per group replication.
- We can compress the group contents. MiG state date and internal protocol information is largely text-based and should compress very well. This trades increased CPU utilization for decreased network I/O.

For a single, non-flushing, write operation, the cost should drop from some double-digit number in milliseconds to a few microseconds. For an operation that causes a flush, it depends on the size of the group. Let us assume, optimistically perhaps, that our group contains 10 writes of the same type as we used to calculate efficiency earlier$^{(\text{See } 5.4)}$.

Ten actively replicated writes will cost at least $10 \times 126.94\,\text{ms} = 1.26\,\text{s}$, again excluding TCP overhead.

A group of 10 buffered writes takes slightly less than 10 times the space in bytes, and thus in bandwidth latency:

$$\frac{(12000 \times 1.37) + 320 + 1000}{10 mbit/sec} = 16.7\,\text{ms}$$

But we only have to pay the latency cost to replicas once, so in total it is $85 + 40 + 16.7\,\text{ms} = 141\,\text{ms}$ which is nearly 9 times faster - still without TCP overhead. Even if buffer/group management was to take 1 ms per write in pure overhead (which would be unexpectedly high), the improvement is quite extreme.

Adding compression to the mix is also possible. In this case we would probably choose to bypass XMLRPCs multicall functionality in favor of serializing the entire group data structure to a string, which could then be compressed. I will not analyze the numbers in detail but assuming that the overhead in bytes before compression is maybe 15% for this composition of data, that the compression step took an additional 2 ms, and lastly that we can compress to one third of original size (for text!). This could cut the 16.7 ms figure down to perhaps half.

7.1.1 Problems

There are two immediately obvious solutions to the interference problem but neither are an immediate fix.

- Relax POSIX requirements and only report errors when flushing the group. This is an option only if the application is aware of this requirement, which many are not.
- Use a shadow-copy where we can roll out changes, and roll them back if replication fails.

The latter option is the least intrusive but also the most complicated. There is a number of ways to achieve this functionality, such as using a transactionable storage-backend or use an in-process copy of data and metadata. The latter can be quite complicated to implement correctly as it would need to be able to support all possible modifications, and then be able to persist them to disk.

A partial solution would be to use a write-through model for metadata, but use write-back for data. Unlike metadata, data are somewhat simpler to cache and then persist.

8 Evaluation

The earlier parts of this thesis described the MiG groups' own requirements for a distributed storage solution for MiG (1.1.1) and then combined research into existing filesystems (2) with a close look at MiGs actual usage patterns (3) to best determine the course going forward. The first part of this evaluation compares the goals against the capabilities of the design and the prototype. The second part describes how *correctness* is evaluated.

Lastly, for this system to be real-world viable we cannot ignore performance. The third part of the evaluation describes and performs several benchmarks on the prototype.

Together, the three parts of this evaluation assess the suitability of GRSfs, while taking into consideration that filesystems is a mission-critical piece of infrastructure.

8.1 Requirements evaluation

If we go all the way back to our initial requirements list, it is clear that not everything turned out exactly as originally envisioned.

It became apparent after an analysis of MiGs usage patterns that a fully active replicated/sequential design would most likely experience performance problems and significant thought has gone into possible solutions to this problem, detailed in section 7. Other suggested features, while they have been implemented, could be improved upon and this has been discussed in section 9.

8.2 Correctness

The testing methodology was to combine automatic and manual tests.

- Automatic tests are a series of separately executable "unit tests". Each test is either very narrow, or builds upon previous tests. A test is successful if it does not crash, does not hang and produces the expected result. Performance is irrelevant here.
- Manual tests are done through the command line for tests that are not easily scripted.
- By using a third-party filesystem test tool.

That all tests yield the expected result is one of two success criteria when determining correctness. The last criterion is an error free performance evaluation run, as described later.

Table 5: Test case categories

Test	Result	Description
System calls	✓	Automatic test by invoking function_test.py which tests each system call through Python and some mode combinations individually.
Long-running	✓	The system was observed to be stable after more than 2.5 million replicated modifications.
Resume	✓	Could the system resume after being shut down (with the same configuration)
Branching	(✓)	Branch to different folder. Error(s) encountered: if mountpoint present in subtree
Failover	✓	Killing an instance through the command line. Attempting to join with a node that was out of sync. Test of hot-spare re-imaging
Static analysis	✓	All files were analyzed with Pylint. While Pylint is not exactly the state-of-the-art of static analysis tools, it can provide useful hints.

In addition to these tests, the two third-party utilities known as PostMark and Bonnie++ were used for both correctness and performance testing.

PostMark is a small filesystem benchmark that is designed to simulate a mail server. It has many of the same characteristica in its use of the filesystem as MiG, such as a high rate of file metadata operations and a heavier than normal bias towards write-heavy operations (though PostMark is even more write-based). It is single-threaded and sequential, like MiG.

Bonnie++ is also a file-based benchmark; but at a lower level than PostMark as it is a synthetic benchmark, as opposed to attempting to simulate a real-world workload. As an example, the very first Bonnie++ test consists of writing very large files, 1 byte at a time. This is very nearly the worst possible scenario for the active replication model as every 1-byte write must be replicated twice before it returns. For that reason, it was unrealistic to complete a full Bonnie++ run with active replication and the test was stopped after approximately 1,000,000 writes[23]. Bonnie bases its test set on the properties of the machine on which it is run, by default, and on a host with 8 GiB of RAM, the full test set would consist of tens of thousands of files requiring at least 16 GiB or more of disk space. Please see the Bonnie man page for more information.

Both of these third-party tools ran without any errors reported. Observing the system at the time of the Bonnie++ run showed that resource use was low and entirely stable ($\approx$ 60 MiB of active memory use and acceptably low CPU utilization) throughout the run.

While we can have some level of confidence in the primary feature, the replicated filesystem, it is clear that less frequent features such as failover and branching could benefit from more tests.

[23] A full run of Bonnie++ with default settings would have required at least 1×10^{10} operations

8.3 Performance

Performance is measured in three different ways:

1. Synthetic benchmark that tests only read and write operations. This primarily tests data throughput by repeating reading, respectively writing, 1,000 times with data size varying from 1 B to 256 KiB. The underlying data file for the read operations was a pseudo-randomly generated file of 300 MiB.

   This is repeated 3 times and the lowest (best) result of the three is used.
2. A benchmark that approximates MiG grid_script behavior (but with greater interference). The workload consists mostly very short operations (meta-data, small reads and writes) and the tests is therefore mostly latency-limited. The workload is repeated 1,000 times, and again repeated three times as above.
3. PostMark, described above
4. Bonnie++, also described above. The test size was reduced to 1 MiB as the full test is extremely long.
5. Alternating reads/writes which is designed to stress the concurrency lock.

To establish a baseline, each of the above benchmarks was first invoked on a instance where replication was artificially disabled (termed "1 Copy"). This generally tells us how much overhead the FUSE/Python/GRSfs combination costs before replication. The benchmarks were then repeated with degree 2 and 3 on up to three separate physical machines (on a LAN, as WAN-networked machines with open firewalls were not available). In order to have an idea of the cost of replication over native disk access, each benchmark also includes data from native (local disk) and a different type of network filesystem, NFS. This is to some degree an apples-to-oranges kind of comparison, which is worth bearing in mind when considering the charts below.

System specifications All benchmarks were done on a 100 Gibit s LAN on a cluster with identical physical machines: Intel Core 2 Quad Q9400 @ 2.66 2.66 GHz. Each machine has 8 GiB of RAM and share storage via NFS.

The physical data store for each GRSfs instance was placed on the local drive rather than on the NFS share.

8.3.1 Result charts

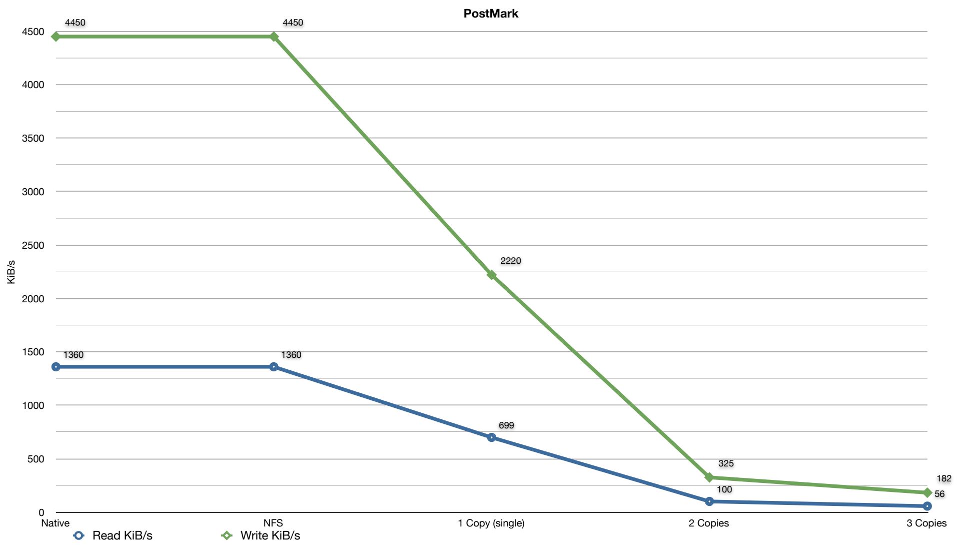


Figure 17: PostMark

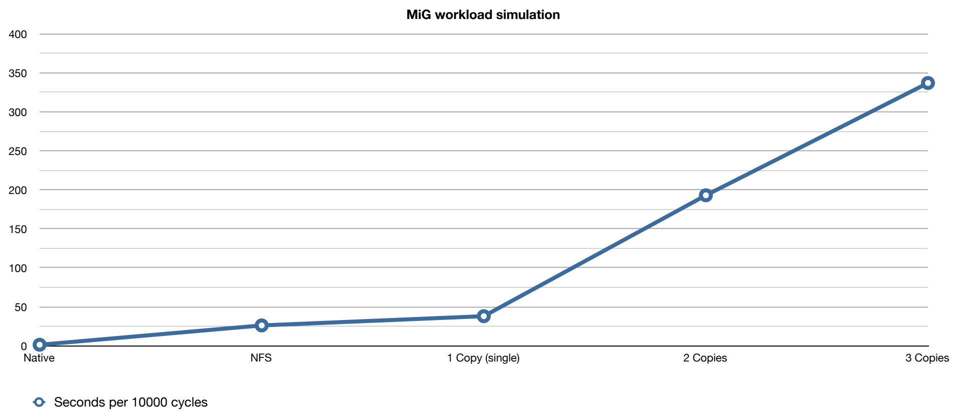


Figure 18: MiG workload simulation

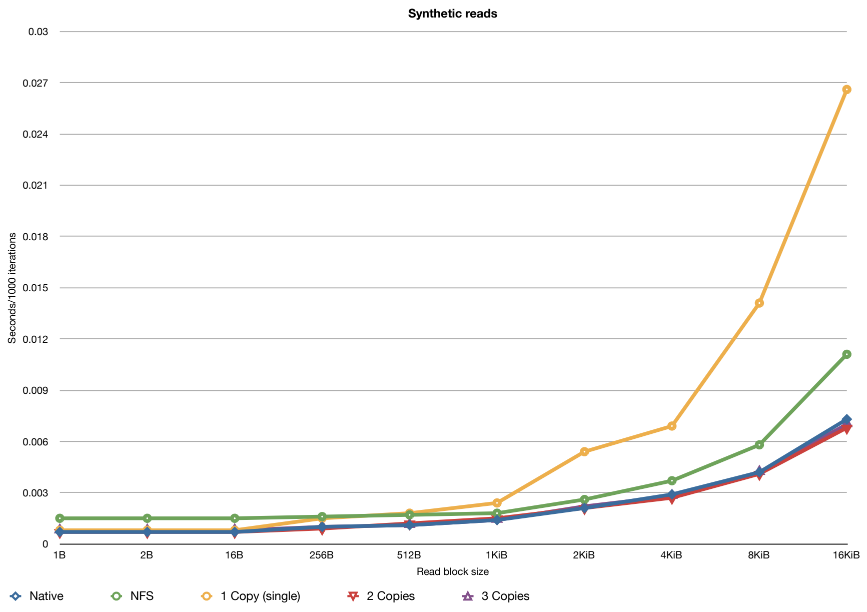


Figure 19: Synthetic benchmark - reads

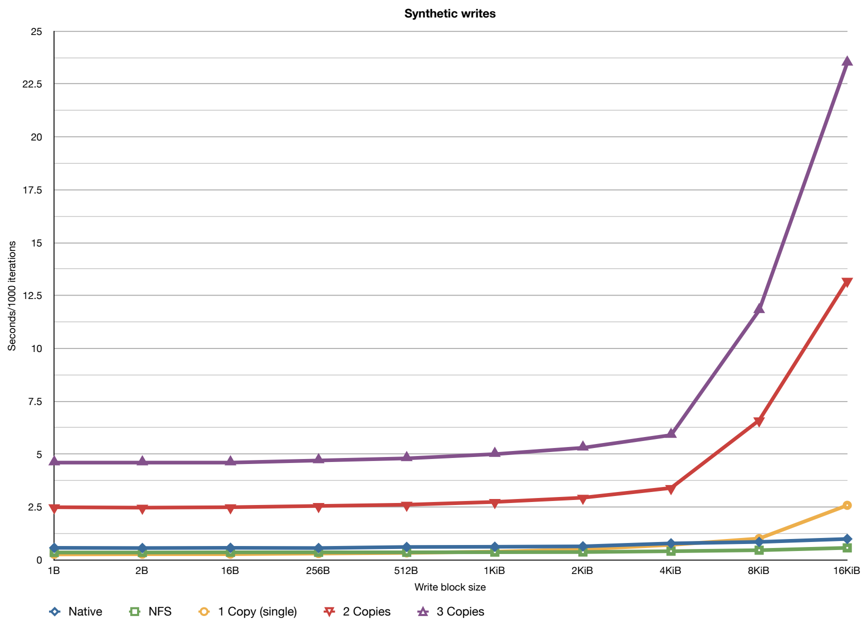


Figure 20: Synthetic benchmark - writes

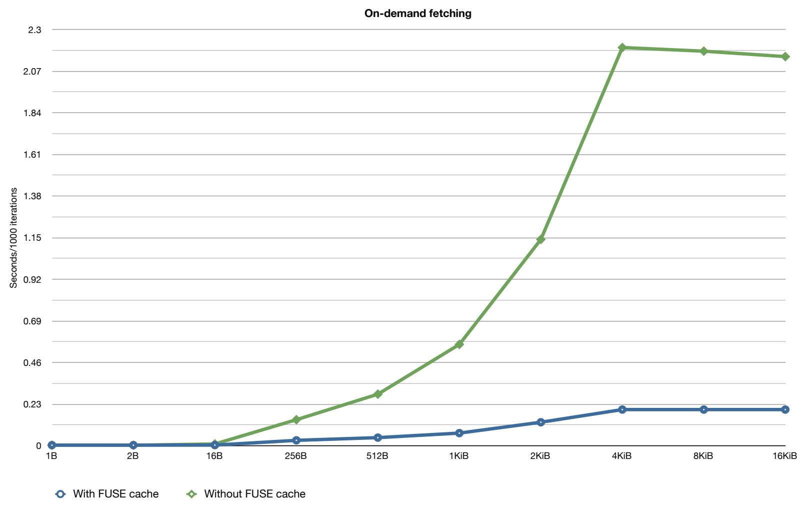


Figure 21: On-demand fetching

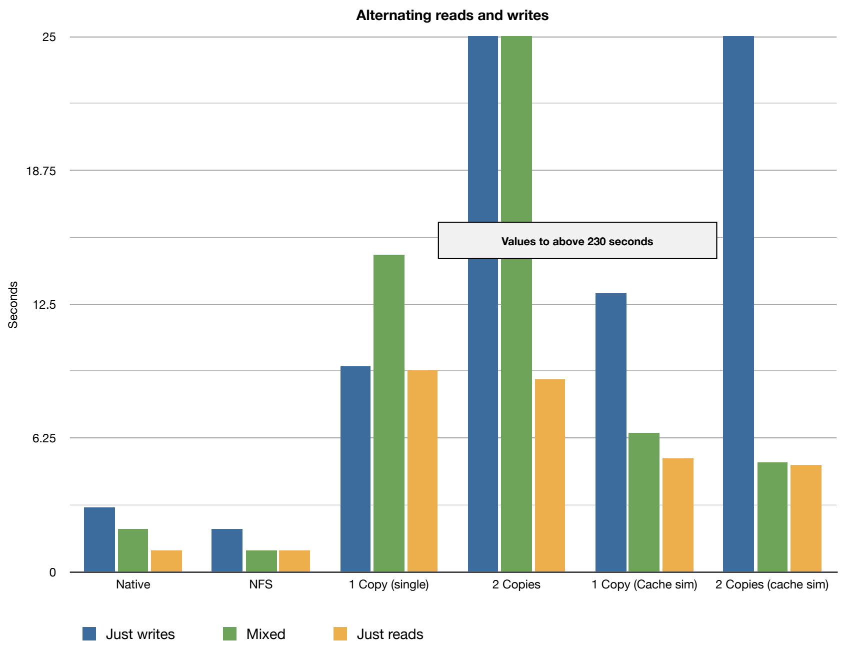


Figure 22: Alternating reads/writes

8.3.2 Remarks and reflection

The preceding charts generally indicate the same trend: the filesystem itself is quite fast but replication is very expensive. Starting with PostMark we can see performance degrade as the redundancy degree gets higher. It is noteworthy that read performance drops dramatically as well since PostMark is fully sequential, and lock contention should not be an issue. A possible reason is if slow write speeds affects read speeds as well within PostMark. Aside from data read/write performance, PostMark also tracks some metadata operations. These data can be see in Listing 3.

The MiG workload simulation (Figure 18) indicates that any type of network drive is vastly slower than locally, for this type of workload. This result conflicts somewhat with the indications from PostMark. We can also see that GRSfs performs reasonably well with no replication and that replication costs rise linearly with degree of redundancy.

The synthetic read benchmark (Figure 19) show generally very favorable results with little overhead for read requests. The stand-alone instance is consistently slower than all other runs - the reasons for this is not entirely clear but may be related to the special-case "stand-alone" dispatcher, which was intended for debug purposes only. The remainder acts as expected: above 1 KiB in block size, we start to see an increase in timing, roughly linear to the number of bytes read per iteration. Block sizes above 16 KiB have not been charted in order to retain some detail at the short block sizes, but results are available in text format in Listing 2.

Synthetic writes (Figure 20) display the tendency that can be expected; replication is expensive and latency rises with block size. Again, raw data and block sizes above 16 KiB are available in Listing 2 and corroborates the linear-with-block-size increase we saw for reads.

Bonnie++ data have not been included as charts, mainly because each run gives quite a lot of information that is not easily distilled into just one or two charts. The reader can find the raw data at Listing 2.

As expected, there was no slowdown in relation to file sizes, only to block sizes.

Figure 21 shows read performance over the network, with and without FUSE cache. The results are generally as expected at the higher block sizes, while the lower block size reads are improbably fast when the cache is disabled. It may indicate that we are still only touching internal buffers.

There are a number of interesting areas that warrant closer attention. First off, we note that the latency with double redundancy (1 copy over the network) for a 1 B write is around $\frac{2.493\,\text{s}}{1000\text{ iterations}} = 24\,\text{ms}$ which is quite high for a local area network. There are a number of possible factors: general overhead, serialization, RPC, remote overhead, flushing and so forth. By setting up an XMLRPC server with an empty function, and a corresponding XMLRPC client that calls the function 1,000 times, we get an indication of the cost of serialization/RPC/deserialization (results in Listing 6), and this turns out to be around 12 ms per call for a test that is somewhat biased in XMLRPCs favor since it skips the transfer of meta-information that the filesystem needs.

The other point is how much the current type of lock hurts. The expectation would be that for some workloads where reads alternated with writes, it would be quite costly and Figure 22 bears this out. The chart details the latency for 3 different workloads in six situations (note that the chart goes massively off the scale for 2 copies). The test starts one write thread (with short block size writes) and four threads that only read (with a larger block size) and no overlap between the write file and the read file. For the last two scenarios, termed "cache sim" we have simulated the existence of a cache and bypassed the global lock. In an ideal world, we would expect the mixed case results to be much closer to the read case than the write case and we can see this holds for our hypothetical non-blocking/cached implementation. It should be mentioned that the test may be affected by being written in Python; the Python GIL is supposedly able to release at blocking I/O[3], meaning that threads should be sufficient for the purposes of this test[24] but it is worth considering redoing this test in a language such as C.

8.4 Real-world usage ready?

It was mentioned earlier that filesystems are a critical infrastructure component. Typically, filesystems are tested on a vast number of machines, with a vast number of failure scenarios thrown at it. It is not possible for the author, or even the MiG group, to create enough boundary or stress tests to entirely satisfy this metric. However, our use case is much narrower as well.

Consider these two aspects:

- Is it possible to recover and bypass the system if a critical error occurs?
- Is it sufficiently fast?

The first point is somewhat helped by using the passthrough backend: it is both based on highly tested storage and is accessible without GRSfs.

The second point is harder to answer. Obviously performance with active replication is very much slower than native disk access but we can look at the data from the workload simulation and make some very general conclusions. This test, with 3 repetitions, runs at about 210,000 replicated writes and takes 337 s to do so (obviously the server has nothing else to do here which may skew results!). $\frac{210000/3}{337} \approx 207$ operations per second. 207 is significantly more than the 32.4 that MiG required during the analysis phase$^{(\text{See } 3.4.1)}$. Thus we may have an acceptable starting point that we can continue to work on, even before any of the suggested optimizations. Certainly, the MiG software startup will be very arduous if it was running on GRSfs and performance for user-driven tasks, which were not included in the analysis, is likely quite low.

[24]The Python GIL is the subject of much controversy[8] and it is possible that it is in fact interfering negatively with the test

It is worth stressing that it is not recommended that GRSfs is adopted for general use at this point - at the very least it should enter a period of prolonged test on non-critical data. A mirror copy of MiGs' state data and grid_script would be a good idea for further testing.

9 Future work

9.1 Improved analysis

The initial analysis of MiGs I/O patterns$^{(\text{See } 3.2)}$ was greatly helpful in the planning phase of this project. Many of the suggestions in this section revolve around improving the caching or prefetching of information, and this would benefit from having more detailed data about MiG.

The experience with the initial data-gathering process and tools showed the limitations of using standard desktop tools to handle this task. Even after significant post-processing and aggregation, the final data file contained more than 70,000 data points, each with a set of aggregated statistics. This should not be an unreasonable amount of data for enterprise-level tools, which spreadsheet applications of any flavor proved not to be.

The conclusion is that it is a simple process to gather even more information in the data trace itself, such as logging more system calls and the timestamp for each event. It is a somewhat larger task to store all the data for easier analysis and to create "reports" that can aggregate the information to a useful level. An alternative that might require less potent tools would be to run the analysis over a much shorter timeframe and assume that the data gathered remained representative of the whole. It does depend on the user-supplied workload on the server and this would have to be planned ahead.

On the topic of having a standardized workload for MiG, it is clear that it would be beneficial to have to be able to measure how changes in the filesystem affected MiG. The drawback is of course that it takes some time and care to compile a comprehensive set of data.

9.2 Smarter hot-spares

Hot-spares can be previous active member nodes, and they also double as on-demand fetching instances. By extension they will in some cases already carry some of the data that otherwise must be transferred as part of an upgrade. While fail-over is not a particularly frequent operation, we can save time required for failover proportional with the amount of non-changed data that remains on the node (minus some cost for ascertaining exactly what is and what is not changed).

9.3 End-to-end data integrity

ZFS is a primary mover when it comes to end-to-end data integrity. It does this by exploiting a combination of storing data in reasonably sized chunks (blocks) which are checksummed with an appropriate hash-algorithm and using Merkle trees to form a validated tree.[81]

There is nothing to hinder for a distributed filesystem such as GRSfs to do the same, although it is difficult to retain the 1-1 relationship in the current storage backend while doing so. In other words, this feature is likely best combined with a new storage backend.

9.4 Alternative network backend

The limitations of XMLRPC has been discussed several times in this paper. A sufficiently good implementation of delayed writes will likely alleviate many performance problems with this backend. Nevertheless, it is not a particular great choice for a production type system. Amongst other problems, delayed writes help very little when acting as an on-demand fetching node.

9.5 Alternative storage backend

Using passthrough is a significant tradeoff. On one hand, it is simple, well-tested, and fast. On the other hand, many improvements are difficult to do without tighter integration to the data store. The data-integrity method mentioned above is one such example[25]

While features such as copy-on-write, where we can copy datasets in the background without blocking and simply redirect the new data to a temporary location, would be useful for both branching, fail-over and perhaps even to parallelize this portion of the REPLICATE step, it is not entirely simple to do so with passthrough (for data only, perhaps). A custom storage backend, or reuse of one with support for such features, would help greatly.

9.6 Transfer optimizations

Some general transfer optimizations are easily achievable. In general, they are expected to have correspondingly little effect unless coupled with a less strictly-replicated model.

Compression of in-flight data This was discussed in the section on delayed writes. There is nothing to hinder this being applied to the actively replicated model as well. In practice, it would probably make sense to set a low watermark such as 4 KiB before compression was applied

Binary wrapping The actively replicated model currently wraps every block of data in a binary-compatible wrapper (BASE64). This costs an overhead of about 37% and is not necessary for ASCII transfers.

[25]though such tricks as using hidden state files is useful for storing metadata that the system is otherwise not capable of. Another option would be extended meta-attributes but this is not universally supposed. For example, Python does not

9.7 Dynamic adaptation for unstable groups

The current design is static when it comes to the degree of redundancy: though it can be changed on the fly, it must be done manually.

It is trivial to implement a system which kept track of node failures, and if node failures were more frequent than expected, it would increase the redundancy degree - or vice versa obviously.

Another option that could be relevant is to keep a much longer log of operations which could be used to quickly bring a node up to speed after a transient fault. Some analysis would have to be performed on the distribution and length of such transient faults, if any. If faults were frequent and very short, then keeping a log of a few minutes of operations is likely to require only very reasonable resource usage.

9.8 Migrated writes

Migrated writes covers sending a write request to the master, instead of rejecting it. Some care will have to be taken to avoid overloading the master, of course. Speed is also not likely to be impressive.

9.9 Improved automatic tests

The currently automatic test framework is primarily focused on file operations, while meta-operations such as failover have been tested manually so far. A fully automated test framework would assist a great deal when making changes in the future so the human operator is not forced to repeat a potentially long series of steps (that may not even cover all situations) at every change.

9.10 Known issues

Background operation Threading within the XMLRPC server seems to have caused some interference with FUSEs own threading model. It is possible to start the process in the foreground (and then background it if desired), but FUSEs own in-background switch is currently broken.

Logging switch There is a problem with the logverbosity command line option. If not specified, logging is silent. If specified, then logging is forced to DEBUG level.

Nested mountpoints during snapshot are not seamlessly supported for spare-upgrades, and branch-and-delete will fail if it encounters one such.

10 Conclusion

In this paper I have collected high-level requirements for a new distributed and replicated storage solution for the Minimum intrusion Grid. Following this, I have examined the current state of the art within both local and distributed storage solutions. The current usage patterns of the MiG grid server software was collected and analyzed through a custom parsing and analysis tool that was able to distill out highly relevant information about MiG.

This preparatory work was used to form a basis for discussion on several possible approaches that would satisfy the criterion and the individual details that form the basis of a distributed storage solution were each discussed, and eventually used to choose a system design. This design was in turn implemented into a working prototype.

The final prototype features a flexible degree of redundancy and is sufficiently resilient to keep operating with full functionality, down to a user-definable minimum safety level. Even falling below this level, the prototype remains able to offer reduced service. Further, the prototype offers limited, but safe, branching of its datastore.

The features of the prototype has been validated against the initial requirements while correctness was tested by a series of specific correctness tests, as well as a series of demanding storage benchmarks, two of which were designed and written for general-purpose filesystem. Both are well regarded and in active use in many sectors, both for commercial, academic and hobbyist use. The final correctness and performance is also compared against MiGs current storage backend and while there is room for improvement, we can see that the prototype is able to offer new and useful features for MiG while remaining sufficiently performant to meet the requirements of the main grid server software. Several features are proposed that are expected to offer a significant performance increase though they are not sufficiently well-developed at time of writing.

While the project has been largely successful, there remains many small and large issues that must be handled before the prototype can mature. On the practical side, many lessons were learned that can be used to simplify the internal API. A better analysis of the MiG software and improved handling of failover and branching are good candidates for more theoretical work.

On a closing note; writing a distributed filesystem is a very large and complicated task. It is largely a testament to the power of free and open source such as Python and FUSE that it has been possible to create a fully functional distributed filesystem, with hot fail-over in the course of a master thesis.

References

[1] Computer failure data repository. URL http://cfdr.usenix.org/.

[2] Hpc resilience consortium, failure data repository. URL http://resilience.latech.edu/mediawiki/index.php/Data:Repository.

[3] 2010. URL http://wiki.python.org/moin/GlobalInterpreterLock.

[4] A Adya, W Bolosky, M Castro, and G Cermak. Farsite: Federated, available, and reliable storage for an incompletely trusted .... *ACM SIGOPS ...*, Jan 2002. URL http://portal.acm.org/citation.cfm?id=844130.

[5] M Allman. An evaluation of xml-rpc. *ACM SIGMETRICS Performance Evaluation Review*, 30(4):11, 2003.

[6] A Azagury, V Dreizin, M Factor, E Henis, and D Naor .... Towards an object store. *Proceedings of the 20th ...*, Jan 2003. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.8121&rep=rep1&type=pdf.

[7] MG Baker, JH Hartman, MD Kupfer, KW Shirriff, and JK Ousterhout. Measurements of a distributed file system. *ACM SIGOPS Operating Systems Review*, 25 (5):212, 1991.

[8] David Beazley. Understanding the python gil. *PyCon 2010*, pages 1–62, Feb 2010.

[9] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987. ISBN 0201107155. URL http://www.amazon.com/Concurrency-Control-Recovery-Database-Systems/dp/0201107155%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0201107155.

[10] IEEE Standard Board. Information technology—portable operating system interface (posix) part 1: System application program interface (api) [c language], April 1990.

[11] L Bruck and M Code. Deterministic voting in distributed systems using error-correcting codes. *Citeseer*. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.29.9536&rep=rep1&type=pdf.

[12] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The primary-backup approach. Dec 2001. URL http://citeseer.ist.psu.edu/549054.

[13] K Chandy and L Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (...*, Jan 1985. URL http://portal.acm.org/citation.cfm?id=214451.214456.

[14] Stuart Cheshire. 1997. URL http://www.stuartcheshire.org/rants/Latency.html.

[15] et al. Chris Mason. 2010. URL https://btrfs.wiki.kernel.org/.

[16] Jonathan Corbet. 2009. URL http://lwn.net/Articles/358940/.

[17] G Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design (3rd Edition)*. Addison Wesley, 2000. ISBN 0201619180. URL http://www.amazon.com/Distributed-Systems-Concepts-Design-3rd/dp/0201619180%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0201619180.

[18] G DeCandia, D Hastorun, M Jampani, G Kakulapati, A Lakshman, A Pilchin, S Sivasubramanian, P Vosshall, and W Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):220, 2007.

[19] Sape Muller (editor). *Distributed Systems*. 1993. ISBN 0-201-62427-1.

[20] J Elerath. Hard-disk drives: the good, the bad, and the ugly. *portal.acm.org*, Jan 2009. URL http://portal.acm.org/citation.cfm?id=1516059.

[21] S Elnikety, F Pedone, and W Zwaenepoel. Database replication using generalized snapshot isolation. *... DISTRIBUTED SYSTEMS*, Jan 2005. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.4364&rep=rep1&type=pdf.

[22] Andreas Ermedahl, Hans Hansson, and Marina Papatriantafilou. Wait-free snapshots in real-time systems: Algorithms and performance. Mar 2001. URL http://citeseer.ist.psu.edu/428641.

[23] C Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer ...*, Jan 1988. URL http://sky.scitech.qut.edu.au/~fidgec/Publications/fidge88a.pdf.

[24] Armando Fox, Steven D Gribble, Yatin Chawathe, Eric A Brewer, and Paul Gauthier. Cluster-based scalable network services. Oct 2003. URL http://citeseer.ist.psu.edu/700751.

[25] Haim Gaifman and Michael J Maher. Replay, recovery, replication, and snapshots of nondeterministic concurrent programs. Nov 2000. URL http://citeseer.ist.psu.edu/409150.

[26] H Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, 31:48–59, 1982. doi: http://doi.ieeecomputersociety.org/10.1109/TC.1982.1675885.

[27] R Garg, V Garg, and Y Sabharwal. Scalable algorithms for global snapshots in distributed systems. *Proceedings of the 20th annual ...*, Jan 2006. URL http://portal.acm.org/citation.cfm?id=1183439.

[28] S Ghemawat, H Gobioff, and S Leung. The google file system. *ACM SIGOPS Operating ...*, Jan 2003. URL http://labs.google.com/papers/gfs-sosp2003.pdf.

[29] B Hardekopf, K Kwiat, and S Upadhyaya. Secure and fault-tolerant voting in distributed systems. *Proceedings of the IEEE Aerospace ...*. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.26.9544&rep=rep1&type=pdf.

[30] B Hardekopf, K Kwiat, and S Upadhyaya. A decentralized voting algorithm for increasing dependability in distributed .... *5th World Multi-Conference on ...*, Jan 2001. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.15.5788&rep=rep1&type=pdf.

[31] Robin Harris. 2007. URL http://storagemojo.com/2008/02/18/latent-sector-errors-in-disk-drives/.

[32] Robin Harris. 2007. URL http://storagemojo.com/2007/02/20/everything-you-know-about-disks-is-wrong/.

[33] D Heger. Workload dependent performance evaluation of the btrfs and zfs .... *dhtusa.com*. URL http://www.dhtusa.com/media/IOPerf_CMG09DHT.pdf.

[34] RJ Honicky and EL Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)*, 1, 2004.

[35] A Hospodor and EL Miller. Interconnection architectures for petabyte-scale high-performance storage systems. *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 273–281, 2004.

[36] Bill Moore Jeff Bonwick. Zfs, the last word in file systems. 2004. URL http://www.opensolaris.org/os/community/zfs/docs/zfs_last.pdf.

[37] MiG Group Jonas Bardino. Complete job flow overview. URL http://code.google.com/p/migrid/wiki/MiGJobFlow.

[38] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. Jan 2001. URL http://citeseer.ist.psu.edu/477370.

[39] J Katcher. Postmark: a new file system benchmark. 1997.

[40] K Kwiat and B Hardekopf. Distributed voting for security and fault tolerance. *Storming Media*, Jan 2001. URL http://www.stormingmedia.us/61/6190/A619093.pdf.

[41] K Kwiat, K Ravindran, C Liu, and A Sabbir. Performance and correctness issues in secure voting for distributed sensor .... *scs.org*. URL http://www.scs.org/getDoc.cfm?id=1859.

[42] L Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, Jan 1978. URL http://portal.acm.org/citation.cfm?id=359563.

[43] L Lamport, R Shostak, and M Pease. The byzantine generals problem. *ACM Transactions on ...*, Jan 1982. URL http://portal.acm.org/citation.cfm?id=357172.357176.

[44] Leslie Lamport. The part-time parliament. Aug 1998. URL http://citeseer.ist.psu.edu/299559.

[45] B Lampson. How to build a highly available system using consensus. *LECTURE NOTES IN COMPUTER SCIENCE*, Jan 1996. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.61.8330&rep=rep1&type=pdf.

[46] AW Leung and EL Miller. Scalable security for large, high performance storage systems. *Proceedings of the second ACM workshop on Storage security and survivability*, page 40, 2006.

[47] SUN Microsystems. Lustre file system. pages 1–20, Nov 2008.

[48] SUN Microsystems. Lustre hpcs design overview. pages 1–31, Aug 2009.

[49] C Moh. Snapshots in a distributed persistent object storage system. *Citeseer*, Jan 2003. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.7985&rep=rep1&type=pdf.

[50] A Muthitacharoen, R Morris, and T Gil. Ivy: A read/write peer-to-peer file system. *OPERATING SYSTEMS ...*, Jan 2002. URL https://www.usenix.org/events/osdi02/tech/full_papers/muthitacharoen/muthitacharoen_html/.

[51] C Olson and EL Miller. Secure capabilities for a petabyte-scale object-based distributed file system. *Proceedings of the 2005 ACM workshop on Storage security and survivability*, page 73, 2005.

[52] JF Pâris and DDE Long. Efficient dynamic voting algorithms. *13th International Conference on Very Large Data Bases (VLDB)*, pages 268–275, 1988.

[53] E Pinheiro, W Weber, and L Barroso. Failure trends in a large disk drive population. *usenix.org*, 2007. URL https://www.usenix.org/events/fast07/tech/full_papers/pinheiro/pinheiro_old.pdf.

[54] P Reisner and L Ellenberg. Replicated storage with shared disk semantics. *drbd.org*. URL http://www.drbd.org/fileadmin/drbd/publications/drbd8.pdf.

[55] Philipp Reisner. Drbd distributed replicated block device. *drbd.org*, pages 1–9, Mar 2002. URL http://www.drbd.org/fileadmin/drbd/publications/drbd_lk9.pdf.

[56] Goldwyn Rodrigues. Drbd: a distributed block device. 2009. URL http://lwn.net/Articles/329543/.

[57] D Roselli, J Lorch, and T Anderson. A comparison of file system workloads. *usenix.org*. URL https://www.usenix.org/events/usenix2000/general/full_papers/roselli/roselli_html/.

[58] T Ruwart. Osd: a tutorial on object storage devices. *NASA CONFERENCE PUBLICATION*, Jan 2002. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.2916&rep=rep1&type=pdf.

[59] F Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, Jan 1990. URL http://portal.acm.org/citation.cfm?id=98163.98167.

[60] B Schroeder and G Gibson. A large-scale study of failures in high-performance-computing systems (.... *Parallel Data Laboratory*, Jan 2005. URL http://repository.cmu.edu/cgi/viewcontent.cgi?article=1045&context=pdl.

[61] B Schroeder and GA Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you. *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–16, 2007.

[62] Livio B Soares, Orran Y Krieger, and Dilma Da Silva. Meta-data snapshotting: A simple mechanism for file system consistency. Sep 2003. URL http://citeseer.ist.psu.edu/652654.

[63] Steve Staso. Hadoop: a primer. 2008. URL http://wikis.sun.com/download/attachments/38208497/Hadoop-Primer.pdf.

[64] R Strobl and O Evangelist. Zfs: Revolution in file systems. *Sun Tech Days*, Jan 2008. URL http://www.opensolaris.cz/system/files/czosug_tuke_20091007_zfs_1.pdf.

[65] Theodore Ts'o. Don't fear the fsync!, March 2009. URL http://thunk.org/tytso/blog/2009/03/15/dont-fear-the-fsync/.

[66] F Wang, SA Brandt, EL Miller, and DDE Long. Obfs: A file system for object-based storage devices. *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 283–300.

[67] F Wang, Q Xin, B Hong, SA Brandt, EL Miller, DDE Long, and TT McLarty. File system workload analysis for large scale scientific computing applications. *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 139–152, 2004.

[68] SA Weil, KT Pollack, SA Brandt, and EL Miller. Dynamic metadata management for petabyte-scale file systems. *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC'04)*, 2004.

[69] SA Weil, SA Brandt, EL Miller, DDE Long, and C Maltzahn. Ceph: A scalable, high-performance distributed file system. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[70] SA Weil, SA Brandt, EL Miller, and C Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 122, 2006.

[71] Sage A Weil. Ceph: Reliable, scalable, and high-performance distributed storage. *PHD Thesis*, pages 1–239, Dec 2007.

[72] Sage A Weil. Rados: A scalable, reliable storage service for petabyte-scale storage clusters. pages 1–24, Nov 2007.

[73] M Wiesmann and A Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Transactions on . . .*, Jan 2005. URL http://doi.ieeecomputersociety.org/10.1109/TKDE.2005.54.

[74] M Wiesmann, F Pedone, and A Schiper . . . . Understanding replication in databases and distributed systems. *. . . on Distributed . . .*, Jan 2000. URL http://doi.ieeecomputersociety.org/10.1109/ICDCS.2000.840959.

[75] M Wiesmann, A Schiper, and F Pedone . . . . Database replication techniques: A three parameter classification. *. . . DISTRIBUTED . . .*, Jan 2000. URL http://doi.ieeecomputersociety.org/10.110910.1109/RELDI.2000.885408.

[76] Wikipedia. 2010. URL http://en.wikipedia.org/wiki/IBM_General_Parallel_File_System.

[77] Wikipedia. Lustre. 2010. URL http://en.wikipedia.org/wiki/Lustre_(file_system).

[78] Wikipedia.org. Mean time between failures. URL http://en.wikipedia.org/wiki/Mean_time_between_failures.

[79] JC Wu and SA Brandt. The design and implementation of aqua: an adaptive quality of service aware object-based storage device. *Proceedings of the 23rd IEEE/14th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 209–218.

[80] Q Xin, EL Miller, and DDE Long. Impact of failure on interconnection networks for large storage systems. *Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 189–196, 2005.

[81] Y Zhang, A Rajimwale, A Arpaci-Dusseau, and R Arpaci . . . . End-to-end data integrity for file systems: A zfs case study. *usenix.org*. URL http://www.usenix.org/events/fast10/tech/full_papers/zhang.pdf.

[82] Ming Zhao and R J Figueiredo. A user-level secure grid file system. pages 1–11, Aug 2007.

A Benchmark raw data

We previously looked at performance in subsection 8.3 where some data were omitted for brevity's sake. These included the raw data for the synthetic benchmarks that exceeded a block size of 16 KiB. These are all included here.

All data here are also included in the benchmarks folder in the distribution package.

Listing 2: Synthetic benchmark dataset

```
# 3 COPIES
[(1, 0.00078797340393066406),
 (2, 0.00078797340393066406),
 (16, 0.00079298019409179688),
 (256, 0.0010569095611572266),
 (512, 0.0011699199676513672),
 (1024, 0.0014748573303222656),
 (2048, 0.0022699832916259766),
 (4096, 0.0028889179229736328),
 (8192, 0.0042281150817871094),
 (16384, 0.0070888996124267578),
 (32768, 0.012328863143920898),
 (65536, 0.022382020950317383),
 (262144, 0.063840150833129883)]
[(1, 4.6492829322814941),
 (2, 4.6589751243591309),
 (16, 4.6640608310699463),
 (256, 4.7612500190734863),
 (512, 4.8591070175170898),
 (1024, 5.0036420822143555),
 (2048, 5.3208351135253906),
 (4096, 5.9856860637664795),
 (8192, 11.808433055877686),
 (16384, 23.561254978179932),
 (32768, 46.890957117080688),
 (65536, 93.660502195358276),
 (262144, 376.11737203598022)]

2 COPIES:
[(1, 0.00073885917663574219),
 (2, 0.00074005126953125),
 (16, 0.00075578689575195312),
 (256, 0.00099611282348632812),
 (512, 0.0012750625610351562),
 (1024, 0.001522064208984375),
 (2048, 0.0021200180053710938),
 (4096, 0.0027909278869628906),
 (8192, 0.0041370391845703125),
 (16384, 0.0068759918212890625),
 (32768, 0.011848926544189453),
 (65536, 0.021671056747436523),
 (262144, 0.081042051315307617)]
[(1, 2.4932858943939209),
 (2, 2.4785711765289307),
 (16, 2.4962089061737061),
 (256, 2.5591549873352051),
 (512, 2.6214668750762939),
 (1024, 2.7413198947906494),
 (2048, 2.9467809200286865),
 (4096, 3.406714916229248),
 (8192, 6.6847288608551025),
 (16384, 13.266027212142944),
 (32768, 26.420931100845337),
 (65536, 52.961051940917969),
 (262144, 210.18758893013)]


1 COPY:
[(1, 0.00087404251098632812),
 (2, 0.00087499618530273438),
 (16, 0.00089597702026367188),
 (256, 0.0015401840209960938),
 (512, 0.0018589496612548828),
 (1024, 0.0024809837341308594),
 (2048, 0.0054440498352050781),
 (4096, 0.0069410800933837891),
 (8192, 0.014101982116699219),
 (16384, 0.026638984680175781),
 (32768, 0.051359176635742188),
 (65536, 0.10213208198547363),
 (262144, 0.4118649959564209)]
[(1, 0.26797890663146973),
 (2, 0.2695620059967041),
 (16, 0.27400588989257812),
 (256, 0.30984711647033691),
 (512, 0.33446693420410156),
 (1024, 0.39197182655334473),
 (2048, 0.50861907005310059),
 (4096, 0.71786999702453613),
 (8192, 1.0129048824310303),
 (16384, 2.5895280838012695),
 (32768, 4.9613039493560791),
 (65536, 9.8733329772949219),
 (262144, 29.05021595954895)]


# NFS
[(1, 0.0015201568603515625),
 (2, 0.0014951229095458984),
 (16, 0.0015070438385009766),
 (256, 0.0016829967498779297),
 (512, 0.0017201900482177734),
 (1024, 0.0018708705902099609),
 (2048, 0.0026030540466308594),
 (4096, 0.0036079883575439453),
 (8192, 0.0056998729705810547),
 (16384, 0.011099100112915039),
 (32768, 0.019392967224121094),
 (65536, 0.03705596923828125),
 (262144, 0.14974689483642578)]
[(1, 0.35287308692932129),
 (2, 0.34989714622497559),
 (16, 0.35769820213317871),
 (256, 0.35921788215637207),
 (512, 0.35864090919494629),
 (1024, 0.36520981788635254),
 (2048, 0.3718109130859375),
 (4096, 0.4101409912109375),
 (8192, 0.45876002311706543),
 (16384, 0.56862306594848633),
 (32768, 0.98230910301208496),
 (65536, 1.2944378852844238),
 (262144, 3.9185469150543213)]

# NATIVE
[(1, 0.00078105926513671875),
 (2, 0.00075316429138183594),
 (16, 0.00079011917114257812),
 (256, 0.0010199546813964844),
 (512, 0.0011348724365234375),
 (1024, 0.0014150142669677734),
 (2048, 0.0021810531616210938),
 (4096, 0.0029091835021972656),
 (8192, 0.0042560100555419922),
 (16384, 0.0073068141937255859),
 (32768, 0.012576103210449219),
 (65536, 0.019133090972900391),
 (262144, 0.062674999237060547)]
[(1, 0.57127904891967773),
 (2, 0.56081104278564453),
 (16, 0.57546687126159668),
 (256, 0.56190299987792969),
 (512, 0.61463308334350586),
 (1024, 0.62937402725219727),
 (2048, 0.64177107810974121),
 (4096, 0.78691697120666504),
 (8192, 0.85102510452270508),
 (16384, 0.99324178695678711),
 (32768, 1.206082820892334),
 (65536, 1.5944290161132812),
 (262144, 3.5738201141357422)]
```

Listing 3: PostMark full dataset

```
# 3 copies:
PostMark v1.51 : 8/14/01
pm>run
Creating files...Done
Performing transactions..........Done
Deleting files...Done
Time:
        25 seconds total
        12 seconds of transactions (41 per second)

Files:
        764 created (30 per second)
                Creation alone: 500 files (50 per
                    second)
                Mixed with transactions: 264 files (22
                    per second)
        243 read (20 per second)
        257 appended (21 per second)
        764 deleted (30 per second)
                Deletion alone: 528 files (176 per second)
                Mixed with transactions: 236 files (19 per second)

Data:
        1.36 megabytes read (55.89 kilobytes per second)
        4.45 megabytes written (182.11 kilobytes per second)


# 2 copies
PostMark v1.51 : 8/14/01
pm>run
Creating files...Done
Performing transactions..........Done
Deleting files...Done
Time:
        14 seconds total
        7 seconds of transactions (71 per second)

Files:
        764 created (54 per second)
                Creation alone: 500 files (100 per second)
                Mixed with transactions: 264 files (37 per second)
        243 read (34 per second)
        257 appended (36 per second)
        764 deleted (54 per second)
                Deletion alone: 528 files (264 per second)
                Mixed with transactions: 236 files (33 per second)

Data:
        1.36 megabytes read (99.80 kilobytes per second)
        4.45 megabytes written (325.20 kilobytes per second)


# 1 copies
PostMark v1.51 : 8/14/01
pm>run
Creating files...Done
Performing transactions..........Done
Deleting files...Done
Time:
        2 seconds total
        1 seconds of transactions (500 per second)

Files:
        764 created (382 per second)
                Creation alone: 500 files (500 per second)
                Mixed with transactions: 264 files (264 per second)
        243 read (243 per second)
        257 appended (257 per second)
        764 deleted (382 per second)
                Deletion alone: 528 files (528 per second)
                Mixed with transactions: 236 files (236 per second)

Data:
        1.36 megabytes read (698.62 kilobytes per second)
        4.45 megabytes written (2.22 megabytes per second)

# NFS
PostMark v1.51 : 8/14/01
pm>run
Creating files...Done
Performing transactions..........Done
Deleting files...Done
Time:
        1 seconds total
        1 seconds of transactions (500 per second)

Files:
        764 created (764 per second)
                Creation alone: 500 files (500 per second)
                Mixed with transactions: 264 files (264 per second)
        243 read (243 per second)
        257 appended (257 per second)
        764 deleted (764 per second)
                Deletion alone: 528 files (528 per second)
                Mixed with transactions: 236 files (236 per second)

Data:
        1.36 megabytes read (1.36 megabytes per second)
        4.45 megabytes written (4.45 megabytes per second)

# native
PostMark v1.51 : 8/14/01
pm>run
Creating files...Done
Performing transactions..........Done
Deleting files...Done
Time:
        1 seconds total
        1 seconds of transactions (500 per second)

Files:
        764 created (764 per second)
                Creation alone: 500 files (500 per second)
                Mixed with transactions: 264 files (264 per second)
        243 read (243 per second)
        257 appended (257 per second)
        764 deleted (764 per second)
                Deletion alone: 528 files (528 per second)
                Mixed with transactions: 236 files (236 per second)

Data:
        1.36 megabytes read (1.36 megabytes per second)
        4.45 megabytes written (4.45 megabytes per second)
```

Listing 4: Bonnie++ dataset for native, NFS and 1-3 degrees of redundancy

```
# 3 COPIES
Version 1.03c ------Sequential Output------ --Sequential Input- --Random-
                    -Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
Machine Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP /sec %CP
n2 1M 694 0 635 0 642 0 +++++ +++ +++++ +++ 776.8 0
                    ------Sequential Create------ --------Random Create--------
                    -Create-- --Read--- -Delete-- -Create-- --Read--- -Delete--
              files /sec %CP /sec %CP /sec %CP /sec %CP /sec %CP /sec %CP
                 16 98 0 6587 5 216 0 97 0 7009 6 219 0
n2,1M,694,0,635,0,642,0,+++++,+++,+++++,+++,776.8,0,16,98,0,6587,5,216,0,97,0,7009,6,219,0


# 2 COPIES
Version 1.03c ------Sequential Output------ --Sequential Input- --Random-
                    -Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
Machine Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP /sec %CP
n0 1M 1284 3 1104 0 1112 0 +++++ +++ +++++ +++ 1374 2
                    ------Sequential Create------ --------Random Create--------
                    -Create-- --Read--- -Delete-- -Create-- --Read--- -Delete--
              files /sec %CP /sec %CP /sec %CP /sec %CP /sec %CP /sec %CP
                 16 183 0 9185 8 420 0 182 0 6606 2 422 0
n0,1M,1284,3,1104,0,1112,0,+++++,+++,+++++,+++,1374.0,2,16,183,0,9185,8,420,0,182,0,660

# 1 COPY
```

```
Version 1.03c ------Sequential Output------ --Sequential Input- --Random-
                    -Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
Machine Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP /sec %CP
n3 1M +++++ +++ +++++ +++ +++++ +++ +++++ +++ +++++ +++ 5309 1
                    ------Sequential Create------ --------Random Create--------
                    -Create-- --Read--- -Delete-- -Create-- --Read--- -Delete--
              files /sec %CP /sec %CP /sec %CP /sec %CP /sec %CP /sec %CP
                 16 1257 2 6427 4 3419 4 1230 3 6712 6 3534 6
n3,1M,+++++,+++,+++++,+++,+++++,+++,+++++,+++,+++++,+++,5309,1,16,1257,2,6427,4,3419,4,1230,3,6712,6,3534,6


# NFS
Version 1.03c ------Sequential Output------ --Sequential Input- --Random-
                    -Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
Machine Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP /sec %CP
n0 1M +++++ +++ +++++ +++ +++++ +++ +++++ +++ +++++ +++ +++++ +++
                    ------Sequential Create------ --------Random Create--------
                    -Create-- --Read--- -Delete-- -Create-- --Read--- -Delete--
              files /sec %CP /sec %CP /sec %CP /sec %CP /sec %CP /sec %CP
                 16 3727 11 13838 8 3848 6 3642 12 19696 16 3778 4
n0,1M,+++++,+++,+++++,+++,+++++,+++,+++++,+++,+++++,+++,+++++,+++,1


# FULLY NATIVE
Version 1.03c ------Sequential Output------ --Sequential Input- --Random-
                    -Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
Machine Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP /sec %CP
n0 1M +++++ +++ +++++ +++ +++++ +++ +++++ +++ +++++ +++ 5308 6
                    ------Sequential Create------ --------Random Create--------
                    -Create-- --Read--- -Delete-- -Create-- --Read--- -Delete--
              files /sec %CP /sec %CP /sec %CP /sec %CP /sec %CP /sec %CP
                 16 +++++ +++ +++++ +++ +++++ +++ +++++ +++ +++++ +++ +++++ +++
n0,1M,+++++,+++,+++++,+++,+++++,+++,+++++,+++,+++++,+++,+++++,+++,1
```

Listing 5: On-demand raw data

```
# FUSE Cache
[(1, 0.0035960674285888672),
 (2, 0.0036158561706542969),
 (16, 0.0037360191345214844),
 (256, 0.030879974365234375),
 (512, 0.045548915863037109),
 (1024, 0.070745944976806641),
 (2048, 0.13031387329101562),
 (4096, 0.20740914344787598),
 (8192, 0.20886898040771484),
 (16384, 0.20739889144897461)]

# Non FUSE cache
jaws@n1:~/src/fs/mount$ ../../benchmarks/benchmark_ro.py
[(1, 0.0030140876770019531),
 (2, 0.0029840469360351562),
 (16, 0.0099229812622070312),
 (256, 0.14429712295532227),
 (512, 0.28574609756469727),
 (1024, 0.56893515586853027),
 (2048, 1.1414408683776855),
 (4096, 2.2052149772644043),
 (8192, 2.1837830543518066),
 (16384, 2.1590509414672852)]
```

Listing 6: XMLRPC standalone test

```
jaws@n1:~$ python rpcclient.py
[1.1946079730987549, 1.1831672191619873, 1.1822700500488281]
```