Clustered network-attached storage: fault tolerance and performance.

Jonathon A. T. Carter
UP896692

School of Computing
Final Year Research Project (PJE40-R)
May 4, 2022

Supervised by
Dr Rinat Khusainov

Abstract

Clustered network-attached storage heavily relies on the use of distributed file systems. This project looks at key aspects of distributed file systems and singles to measure fault tolerance and performance. It was done due to the gap in research around the measurement of fault tolerance and an interest in performance. By suggesting a way of measuring fault tolerance, this project looks at measuring this on different design types of distributed file systems and thus also testing their performance. The resulting outcome found that the two software were fault-tolerant in their own particular ways, and one was significantly faster performing than the other.

Project repository

All project code and data can be viewed within the Github repository (Carter, 2022).

Consent to share

I consent for this project to be archived by the University Library and potentially used as an example project for future students.

Table of Contents

List of Tables

List of Figures

Acknowledgements

Thank you to all my friends and family who have pushed me with this project. In particular, Mike and Mikey from ILM have helped provide knowledge of real-world experience in clustered NAS. Without my hands-on involvement during placement at ILM, I would not have considered this topic. I owe some gratitude for having that opportunity and a massive appreciation to Yash and the rest of the ILM London technology team.

Many thanks to my supervisor Rinat Khusainov who has provided me with a level headed approach with positive suggestions throughout, and to Linda Yang for taking the time to moderate this project.

Acronyms

The table below is a list of various acronyms that have been used throughout the project.

Acronym	Meaning
NAS	Network-attached storage
DFS	Distributed file system
FOSS	Free and open-source software
I/O	Input / Output
PXP	Personal Extreme Programming
TNF	Time with no node failure
TF	Time after failure of nodes
TFD	Time during failure of nodes
IOPS	Input/output operations per second

Table 1: A number of acronyms used across this report

Chapter 1

Introduction

1.1 The problem

Clustered network-attached storage (NAS) can be used in various situations that require extensive solutions for data storage. These can be stretched across a large region to sync and look after data. There are many solutions for clustered NAS. Deciding the best software, particularly distributed file systems (DFS), for this storage is difficult, especially when there are many solutions in that field. Free and open-source software (FOSS) is a route that can be used by enterprise and home users to achieve their needed solutions. Deciding the best solution for the situation can be a struggle, and having a simple breakdown of what software ranks the best for that situation would be helpful.

1.2 Objectives

This project will focus on the ability to single down aspects of DFS and provide a clearer understanding of where the DFS struggles or excels. It will require measuring a DFS within its specific area to determine this. Therefore a testbed software and subsequent physical cluster will be needed to gather the data required for interpretation. The outcome is a closer insight into the chosen focus aspect of DFS.

Project Objectives:

- Subdivide DFS into key aspects derived from the field of research.
- Provide research questions tailored to the aspects chosen.

- Find points of the DFS that can be measured and interpreted.
- Produce an artefact that can both run and test the candidate file systems.
- Display and conclude the findings inside this report.

1.3 Project constraints

Time is a critical factor in this project, where research can sometimes take years to develop solid conclusions. This project must be completed by the project deadline. Other aspects such as illness from COVID-19 may slow down this project. Getting hold of the needed resources for the project may have caps on the amount available or specification. Data handling needs to be kept in top form to ensure no data is misplaced or corrupted throughout the project.

1.4 Report structure

- Background Research

  Looking at the important aspects of DFS and their software, their specifications and some example hardware that hosts the DFS. Concluded focus aspect to move forward into the project.

- Research Question Specification

  Questions derived from the concluded key concepts of background research.

- Research Design

  Focusing on; how the research questions will be solved, what software will be chosen to be measured and why, the similarities and differences of this chosen software, the experiments to run, the metrics to be measured, how it will be dealt with and the ethics behind this.

- Research Management

  Discusses the way the project is being planned and managed.

- Testbed Requirements

  Describes the target software development method and needed requirements for the testbed.

- Testbed and Cluster Design

  Design discussion of both parts of the artefact following the requirements and needed delivery.

- Testbed and Cluster Implementation

  Describes the development of the whole artefact and how it was created whilst following the artefact design. With discussed issues that were observed and mitigated.

- Results and Discussion

  Graphical representation of the collected results, with bias considered and acted upon. Interpretation of each representation below. Partial reflection included at the end.

- Conclusion

  Project conclusion and reflection with recommendations if repeating and suggestions for further research.

Chapter 2

Background Research

2.1 Key Concepts

What is the most important aspect of a distributed file system?

2.1.1 Fault tolerance

The main goal of clustering is to eliminate every single point of failure possible within a system (Lamb, 2002). It is the ability to continue normal services without the end-user experiencing an impact against the use of the system. It should be considered that if a system does not cover the basic concepts of availability, reliability, safety and maintainability (Ledmi et al., 2018), the resulting overall user satisfaction could decrease when there are impacting moments (see Figure 2.1, p4). Fault tolerance is critical in computing to mitigate failure issues, notably single or multiple points of failure. To decrease outage time, systems can use data redundancy methods. (Shen et al., 2016).

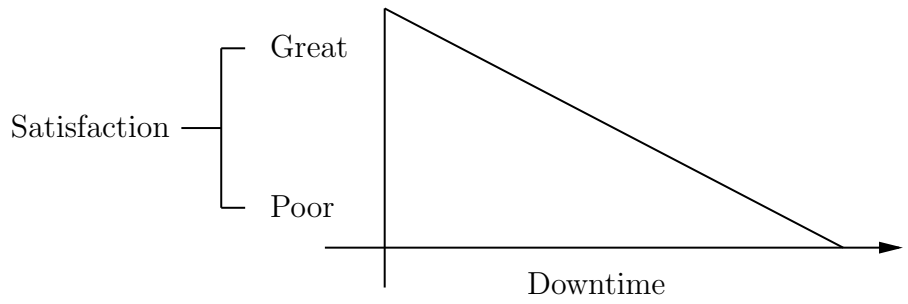


Figure 2.1: User satisfaction decrease with the increase of downtime.

There are a few methods of data redundancy that are available across distributed systems, these being replication and erasure coding (Acronis, 2019). These both work in similar ways where one replicates the file a certain amount of times while the other splits a file up and adds some extra replicas of the splits to allow for a rebuild if parts get lost, and they are distributed. Erasure coding has become popular among many DFS resulting in many adopting this form of data redundancy (Shen et al., 2016).

2.1.2 Hardware reliability

It is clear that as we progress further and further into technology, hardware reliability is ever increasing. This can be defined as the amount of time that a component or system has been operating from the start 0 to failure point t (Verma & Gayen, 2022). Many components make up a clustered NAS from low-level circuitry all the way to high-speed networking. In particular hard drives may be prone to reduced reliability over time, with failure leading to service downtime and data loss (Huang et al., 2019). Ahmed and Wu (2013) argues that DFS has been expressed to be divided into two areas, that being low level and broad level. Where low level relies on physical, meaning hardware. The broad level relies on the server's availability, scalability, communication and connectivity. It could be argued that some areas of low level and broad level could be considered the same with conflicting similarities.

2.1.3 Design and systems

DFS can be built and designed with different typologies (architecture). Each architecture has its advantages and disadvantages. These are the difference in the physical setup of the servers, the roles of each node and query handling. Communication within these designs incorporates many methods adopted by DFS. Many different flavours are available to incorporate into a DFS that allow vast compatibility between different systems.

Different architectures include centralised and decentralised or also known as client-server and peer to peer (Cao et al., 2018; Placek & Buyya, 2007). These significantly define the characteristics of how the DFS would function. Centralised is where a master server is the brain of the operation, distributing information to lower-level servers that handle the cluster's specific tasks. Decentralised is where each server has its responsibility to handle communication, metadata and storage whilst actively updating every other node about its state (see Figure 2.2, p6). Shetty and Manjaiah (2017)

expresses that there are five most important implementations of DFS architectures. With them, it can achieve different requirements set out by users. Some are more performance and scalable than fault-tolerant. Different architectures being designed for a specific application is the crucial point here; however, it remains that the very base of DFS is built upon centralised or decentralised.

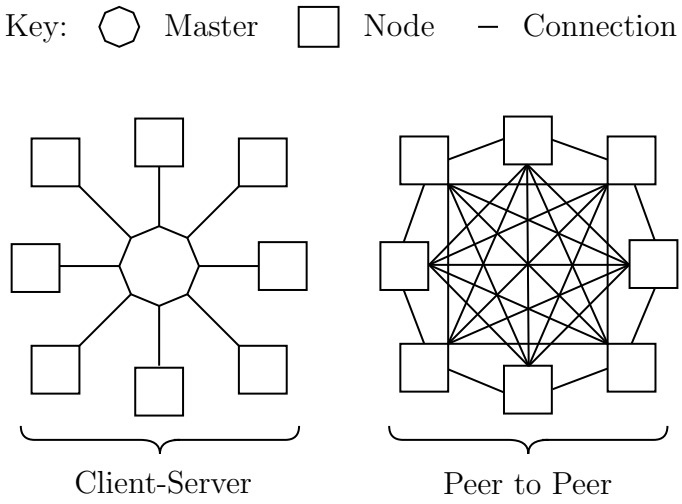


Figure 2.2: Two DFS architecture types (Placek & Buyya, 2007).

Communication methods can be in different forms. The ones between servers and clients (application protocols) or how file systems are composed between cluster components. Application protocols, in most cases, are built upon the established libraries with the incorporation of external libraries for the same system to system connection and external system connections. File systems are built upon established libraries with specific application protocols embedded. Files will need to be universally accessible on all cluster components, so they all use the same file system time. If other external application protocols are incorporated into this, it may decrease the system's performance or client-side operations.

2.1.4 Performance

A DFS will only run well if configured well enough to output its best performance. There are many arguments that specific setups can slow the performance of a system. One is the previous mention of the data redundancy method; erasure coding. It is said to bear a constant load on the CPU, which can cause a slow down (Goldenberg, 2016). It should also be noted that replication enabled on a system would cause operational time in some cases to slow down as the file being replicated 4-5 times would take

longer than 1-2 times. Other factors that affect performance include the caching on the system, amount of clients interacting with the system, scalability of the system and even down to the networking requirements of the system (Gudu et al., 2014; Wang et al., 2019).

Physical networking solutions such as InfiniBand, which has become more widely adopted into high-performance DFS, allow interconnection between data storage with high throughput and low latency. Noronha et al. (2008) was able to see a 188%/150% read/write improvement with the use of Infiniband. Whilst noting that InfiniBand can also reduce link saturation between nodes (Pfister, 2001). The cost of InfiniBand means that primarily enterprise hardware will support this and that for home use will not be affordable.

2.2 Open source software solutions

Currently, there are many options available for users to consider using for their DFS. Each has its advantages and disadvantages with the different design choices and services available. Some example software has been looked into with their considerable differences displayed (see Table 2.1, p8).

Name	**Base architecture**	**Communication protocol type(s)**	**File system type(s)**	**Cluster scalability?**	**OS type**	**Written language**	**Licence usage**
GlusterFS	Trusted Storage Pool (individually managed)	FUSE, NFS, SMB, Swift	FUSE	True	Linux	C	GPLv2 & LGPLv3
Ceph	Reliable Autonomic Distributed Object Store (RADOS) (managed by master)	FUSE, AWS S3, Swift, librados(C, C++, Python, PHP, and Java)	POSIX compliant file system and/or FUSE	True	Linux	C++	LGPLv3
Lustre	Split management, metadata and data services. (managed by master)	POSIX, NFS, SMB	POSIX (ext4)	True	Linux	C	GPLv2
MooseFS	Split, management, metadata and data services. (managed by master)	POSIX, FUSE	POSIX compliant FUSE	True	Linux	C	GPLv2
JuiceFS	Data storage, metadata (individually managed)	POSIX, HDFS, S3, CSI (kubernetes)	POSIX	True	Linux	Go	AGPLv3
SeaweedFS	Split, management, metadata and data services. (managed by master)	HTTP (REST), POSIX, FUSE, S3, HDFS	FUSE	True	Linux	Go, Java	Apache v2

(Ceph, 2022; chrislusf, 2021; Gluster, 2022; JuiceFS, 2022; Lustre, 2021; MooseFS, 2021)

Table 2.1: Some current FOSS DFS solutions.

2.2.1 Breakdown of software

With the different DFS software presented, there is a range of different base architecture types described for each software. When described as management, metadata and data services, this refers to centralised or master architecture. Where a master server routes information to the rest of the cluster, meta-services keep backup records or, in some cases, hold the actual record of movement and data services have their physical storage globally shared across the cluster. In some cases, such as Ceph, there is a different server type which monitors the cluster state. This is why it has a RADOS name since each service has its own dedicated server.

Communication protocols are the API that the systems are built upon and allow interaction. The majority of the systems in the table are built upon FUSE and POSIX. These are filesystems built upon the Unix / Linux kernel, with POSIX following a strict guide of standards to allow it to be followed as a global standard. Other protocols allow communication with third party services such as AWS S3 and CSI. Protocols such as SMB and NFS aim to communicate between servers and clients, mainly Windows-based systems. HDFS is a DFS standard of file systems, meaning it should favour large files over lots of small files. This should be configurable within the DFS software if this is enabled or not.

The formatting of a file system is mainly recommended to be XFS due to the journaling performance over that of ext4. However, Lustre has chosen to stick with ext4 due to its common ground with adoption in a wide range of systems and proven stability. The programming languages chosen for these systems are robust and familiar to the kernel. The software's runtime will be minimal because the languages are similar to that of assembly language (low level, excluding Java). As of writing this report, Rust is on the rise, which mitigates C's dangers, which could very well be seen in newer upcoming DFS. Go is popular since its syntax is much more straightforward than C whilst also maintaining a similar runtime.

The provided licences for each software cover similar ranges of free open source use. GPLv2 allows for commercial use, code modification, distribution of that modified code, use of the modified code privately, and distributors of the original code can place their warranty on its use. GPLv2 must have its licence attached to the distributed code (modified or not), and the original unmodified source owns full copyright. LGPLv3 is similar to GPLv2. However, with more constraints over the distribution of the modified code. Where changes must be stated, it needs to have install instructions attached.

AGPLv3 is very similar to GPLv2, where install instructions must be attached. Finally, Apache v2 is a lesser constricting licence of GPLv2, but the trademark of the source owners must be included in any modified software.

2.3 Enterprise hardware solutions

NAS is very versatile and can be composed of many different makes of hardware that range from commodity to specific enterprise solutions. This section focuses on a range of on-premise enterprise side of the equipment, usually multi-node hardware solutions for companies (see Table 2.2, p10). DFS is installed on these servers to deliver the needed data storage for their specific use case.

Vendor	**Model**	**Storage name type**	**Rack height**	**Storage (chassis)**
Dell	H7000	Hybrid	4U	1.6PB
NetApp	FAS9000	Hybrid Flash	8U	14.7PB
IBM	9500	Flashsystem	4U	4.5PB
HPE	XP8	Hybrid Flash	10U	60PB
Fujitsu	AX2200	Flash	2U	8.8 PB

(Dell, 2022; Enterprise, 2022; Fujitsu, 2022; IBM, 2022; NetApp, 2019)

Table 2.2: Variety of enterprise NAS solutions.

2.3.1 Breakdown of hardware

With a variety of hardware available, it should be noted that specific servers are tailored for specific areas of industry. The naming convention of storage type from this table is mainly hybrid or flash. This is due to HDD or SSD drive arrays installed on the chassis. However, the drive arrays count as one when contributing to the overall shared storage amount when pooling servers together. For example, if the one Dell chassis with 1.6PB (made up of many drives) is grouped with two other Dell chassis, it has a total of 4.8PB of shared storage.

Another factor of enterprise storage is the sheer density of the storage. Where rack height is mentioned, this means the total height on the rack that that server takes (average racks are 42U high, 1.86M). If a server is compact and has lots of storage available, the amount one can fit into a server room increases compared to large servers

with an average amount of storage. Another example is the AX2200 being 2U high, yet the FAS9000 is 8U, which means one could have 4x AX2200s in the same space as that FAS9000, further increasing the amount of available storage in the physical space.

2.4 Conclusion

By dividing this complex topic into these key areas, multiple further discussions can be explored with their respective research questions. DFS has many factors that could well have their own separate project. Some areas that have been looked into do lack particular material, yet seemingly all of the areas are interconnected, making this difficult to break down. The takeaway from this research is that fault tolerance and performance will be looked into for this project.

Chapter 3

Research Question Specification

These have been composed from the earlier background research (see chapter 2, p4) and will be the questions being addressed throughout the project.

3.1 Fault tolerance

Fault tolerance plays an important role. Without it, the system would not be susceptible to faults. The formulated questions cover areas of faults occurring and the ability to investigate the difference in design. These are important as they are focused specifically on certain features but look to answer the aspects mentioned in background research (see subsection 2.1.1, p4).

How functional or tolerant is a system after multiple points of failure?

With pre-planning, a system should be able to continue running after some points have failed. By measuring aspects of the system during the simulated environment, results can be analysed, presented clearly and conducted in a controlled manner.

How do different systems operations compare during a time of failure to other systems?

Many users of a clustered NAS will be interacting with it in many separate ways. In this instance, choosing the correct DFS for a system is very important and can help mitigate problems when the system is under load during a time of failure. This also can be done by measuring aspects of the system during the simulated failure. Choosing different software means that certain architectural types can be compared against each

other.

3.2 Performance

Many factors can influence the performance of a system. This specific question looks to shed light on differences in the design of a system and provides awareness towards the concepts mentioned (see subsection 2.1.4, p6) but with one avenue of measurement.

How do different system designs differ in operational performance?

Different architectures provide the same end goal for a user but in different ways of delivery. Having a way of measuring the speeds that the different architectures perform operations in could provide a clearer picture of the performance of that particular architecture. Straightforwardly measuring the throughput to compare against the different system designs would allow these delivery differences to be seen.

Chapter 4

Research Design

Fault tolerance is a significant aspect of a DFS that cannot be measured in one test. It will need to be done by focusing on an area of the system. System architecture plays a big part in fault tolerance due to the configuration of servers. This area should be a deciding factor in the choice of the DFS. Further factors provided by each software contribute to the system's overall fault tolerance, which will be broken down within the chosen DFS.

4.1 Distributed file system choices

Name	Architecture type
GlusterFS	Peer to Peer typology
Ceph	Master typology
Lustre	Master typology
MooseFS	Master typology
JuiceFS	Peer to Peer typology
SeaweedFS	Master typology

Table 4.1: A number of DFS softwares to select from

From the outline examples in the table (see Table 4.1, p14) I have chosen MooseFS and GlusterFS. After some research into the DFS within the table, these two pieces of software are well documented online for setup, have different main architecture types and do not require enterprise hardware for setup.

4.2 Selected software breakdown

A break down of the selected software that will be used along the course of this report can be seen in appendix (see Appendix C, p78). It has helped for this design section.

4.3 Comparison of architectures

- Master - Has a centralised point of management and connected servers follow the master for instructions and information.
- Peer to Peer - Is an individually managed cluster of servers that work together and distribute instructions and information.

Tests can be tailored to be run similarly on both and test individual aspects of each architecture. The main focus of this will be the time taken for input/output (I/O) operations to complete on the system; this is the primary daily use of the system. With the focus being fault tolerance, the worst-case scenario is a server failure. With a cluster of master architecture, the focus points are the master server and storage servers. A test tailored to master server failure would be a good comparison for other master architecture software for future work. Peer to peer architecture, being individual, would benefit from a comparison of different servers in the cluster going offline in a randomised or ordered form and measuring the time impact.

Other comparison areas could include manipulating hardware usages, such as a high CPU usage on a master server or some peer to peer servers, and disk failures on the storage servers themselves. Network tolerance comparisons could be focused on a saturated vs unsaturated network, which would provide delays in data transmission. These are all viable options for comparing the different architectures.

4.4 Hypotheses

These have been chosen to represent the different architectures being compared and tested. They will be reflected upon after results have been gathered and interpreted.

4.4.1 Master architecture

- Master typology should be fault-tolerant enough to withstand the failure of its master server.

  Assumption - The master server is the brain of a master architecture, without it the cluster would not be able to communicate, therefore the fault tolerance of the master architecture should withstand the failure of it.

- Operations will take longer to complete on a system with an overloaded master server.

  Assumption - Due to the master server being the brain of the master architecture, if the brain is overwhelmed with computations it would take longer for that queue to be completed.

- Time taken of operations during failure will be longer than that of time taken after failure.

  Assumption - Operations being interrupted should take longer to correct due to the possibility of node communication going down and needing to be rerouted.

4.4.2 Peer to peer architecture

- Peer to peer topology will be less resilient to failure than that of master typology.

  Assumption - Due to the amount of servers needing to update metadata about cluster failure compared to one master, more failure may result in this metadata not being communicated. This may cause more faults and time delays.

- There will be no effect on the time taken of operations if different servers fail as they all should be equally distributed.

  Assumption - Peer to peer works by having all servers being equal to each other. Therefore all loads should be equally distributed and not favour one particular server over another when distributing files.

- Time taken of operations during failure will be longer than that of time taken after failure.

  Assumption - This will be the same for peer to peer architecture as master architecture (see subsection 4.4.1, p16).

4.5 Experiments

The following tests have been picked to compare the time taken for operations to complete and if operations are still viable after the failure within a system. A test for hardware stress has also been chosen for the master architecture. It will provide helpful insight into master architectures. Some further tests have been provided to give an insight into the system performance of hardware usage and operations completed under different workloads. Data redundancy (replication) has been accounted for to help provide a similar system tolerance setup. Caching has been disabled to provide a better average for the total time taken for the operation to complete. If a file is written, deleted and re-written, it will not have that file in the cache to complete the operation faster than the first time it was written to the cluster.

Test ID	**Test Detail**	**Moose (M*)**	**Gluster (P2P†)**
1	Measure the time taken for operations to complete with no failure with a replication of 4 and caching disabled.	✓	✓
2	Time taken for operations after failure with 4 replication and caching disabled on the storage.	✓	✓
3	Time taken for operations during failure with 4 replication and caching disabled on the storage	✓	✓
4	To what amount of node failure does the cluster operations stop working with 4 replication on the storage (completed or not)	✓	✓
5	Time taken for operations under failure of master and 1 storage node with 4 replication on storage, this will test the tolerance of master typology	✓	
6	Time taken for operations of the cluster to complete with a master server that has extra CPU stress.	✓	
7	Time taken for operations of 1, n node failure and 1, m where n and m are different nodes in each test.		✓
8	Overall cluster load during failure of storage nodes	✓	✓
9	Measurement of throughput of the different architectures with the same I/O operations	✓	✓

* Master architecture, † Peer to peer architecture.

Table 4.2: Tests to be conducted against each system type. Test details (see subsection 4.6.1, p18)

Each operation will be conducted three times and have a 60 second cooldown to let the cluster "catch up" (time for the system to get back to "low usage"). It provides accuracy within the final results and a maximum, average and minimum time. Each system will have 12 storage nodes set up in the cluster providing a fair comparison between architectures when conducting tests. The replication has been set to 4 because it provides a repeatable test, gives a total cluster redundancy goal of ¼ and due no other redundancy methods available on MooseFS FOSS version (erasure coding locked) (see Appendix C, p78).

4.6 Method and analysis breakdown

4.6.1 Test details

1. The workload I/O is run against the cluster mount point as it would in a normal healthy operation with no node failures. The time taken is measured here.
2. This will involve the cluster having workload I/O run against the mount point. A node or nodes will be turned off for each test before the I/O is run, continuing until the tests fail. The time taken for each I/O run to finish is measured.
3. This is similar to test 2, where workload I/O is run against the mount point. However, the node(s) will be turned off mid-test. This time of failure is determined by the run time of each I/O from test 1, where operation average runtime divided by 4 is the time until that a node(s) will turn off.
4. This involves turning off a number of nodes until the cluster becomes unresponsive to see how far the cluster can handle failure with the replication level still set at 4. It can be gathered during tests 1 and 2. Operations completed are being measured here.
5. Focused on the master architecture, which involves turning off the master server and one storage node. Measuring the time it takes to complete the operations. It will involve turning off the nodes before, waiting 60 seconds and proceeding with the operations. Time taken measured.
6. Again focused on the master server. The test will have CPU stress run against the master server. The normal operational time of workload will be compared against the operational time of the CPU stress run set at 50%, 80% and 100%

CPU usage. The CPU stress length will be longer than the normal operational time for total consistency in the results. Time taken is measured here.

7. Focused on the peer to peer architecture, this involves turning off a selected node a and turning off a random node b, it is then repeated again, meaning a will stay the same and b will be different in each trial. The nodes will be turned off before, wait 60 seconds and proceed with the operations. Time taken is being measured here.
8. This involves capturing the hardware usage of the individual clusters during the failure of nodes which will need to be captured in the background during test 3. Hardware usage will be measured.
9. This looks at measuring the throughput performance of the cluster with a varying amount of operations. Either a tool will be used for this or a singular file with various user use cases of file I/O.

4.6.2 Result analysis

- Time taken will be captured and put into a raw table of values for the test. Charts can be drawn up and plotted, which will be used for analysis.
- Completion of operations will be captured and put into a raw table in the form of yes or no. This will be used for analysis.
- Hardware usage of the cluster will be measured and most likely captured in a live dashboard style. This can be used for analysis.

4.7 Variables

There are many different variables in the tests that will be run, below is an overview list of independent, controlled and dependent variables gathered from the test details (see subsection 4.6.1, p18).

4.7.1 Independent

- The number of nodes failing before or during operations
- The selected nodes to fail during tests 5 & 7

- The amount of CPU load staged during test 6

4.7.2 Controlled

- The system architecture (master or peer to peer)
- The cluster design (same amount of storage servers and networking used)
- The operations run against the systems (random and sequential I/O)
- The workloads of the operations (large job or many jobs)
- The method for gathering the throughput of the system

4.7.3 Dependant

- Time taken for operations to complete
- Completion of operations on the system
- Hardware usage of the cluster
- Throughput of operations on the cluster

4.8 Input/output operations load

Workloads of different types are run against different distributed file systems. In the real world, it is the case between an environment of multiple accountants requesting small files or an environment of engineers rarely requesting singular large files (Leung et al., 2008). To capture the most diverse range of operations, both random and sequential, reads, writes and readwrites should be run with two different loads. These two loads should be as follows:

- Load 1 - A singular 100MB file accessed in a read, write and readwrite form.
- Load 2 - A singular 10MB file that has 10 users access it in a read, write and readwrite form.

Tools for this kind of simulation will be discussed further in the report.

4.9 Data collection and analysis

Time completion of the operations needs to be collected, and system metrics for hardware usage and latency between client and typology can be collected for further analysis. The results will be compared for completion time against its own architecture. This data will be presented in a clear bar chart format, tables of values and validation of completion for certain tests. The different workloads and hypotheses will be reflected upon with the collected data.

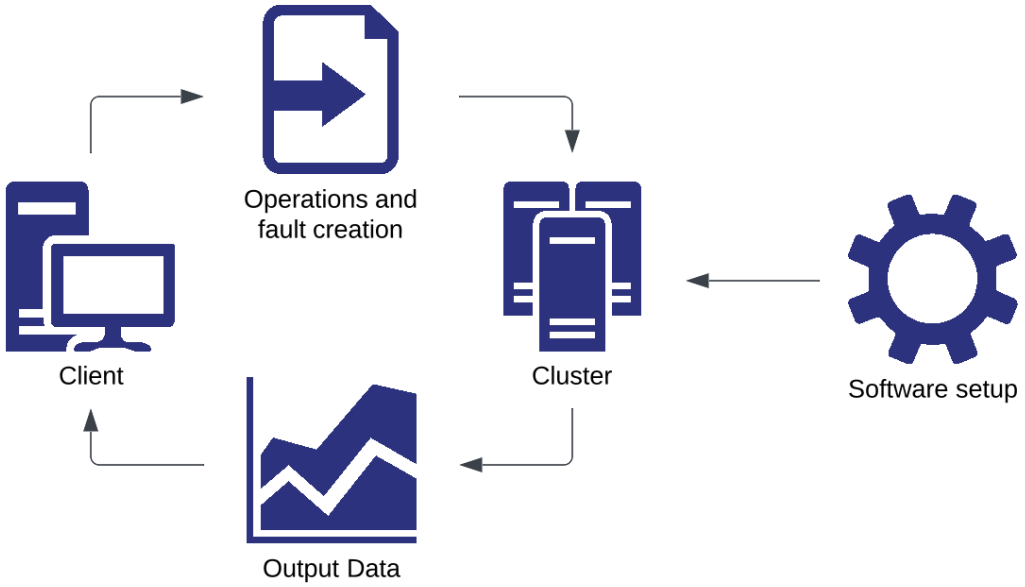


Figure 4.1: Concept method of testing and data collection

4.10 Ethics

In order to make sure the project is as ethical as possible, the research design choices, to the best of this project's research, are ethically sound. The chosen software is being obtained legally, with the data collected being disposable and will not affect the integrity of the software or developers. Any bugs found will be not exploited but flagged appropriately. There are no known risks throughout this project, and the benefits of this research are solely for the improvement of the research area. These points have been concluded from the ethical codes mentioned in Saltz et al. (2018) table 2 where ethical codes from the literature (professional conduct, duty to industry, privacy, data misuse, personal and group harm) apply to this research being undertaken.

Chapter 5

Research Management

5.1 Data collection management

Data will be collected and stored in two locations to avoid any data loss. It will consist of the cloud and physical storage, keeping it safe during the project. Data will be named according to the data collection date and named after the tested software. It will be done to help with analysis later in the project, and further researchers can access the data when this research and testbed have been released.

5.2 Project planning

Maintaining an outlined plan throughout a project is vital in ensuring it does not lose its original intended outcome. The original PID gantt chart will be adhered to as it proposes a sensible path for the project (see Appendix D, p82). The project parts and milestones can be set through kanban tools such as Trello, Jira or Asana. This project will be using Github's kanban board to keep track of the multiple requirements and sections in development. Along with this, any week by week progress is noted in an external spreadsheet with the supervisor, discussed, and the week ahead targets concluded.

5.3 Testbed code management

Keeping track of modifications to the repository through source code management tools is valuable for this project. In this instance, Github is an excellent tool for managing

the code committed to a repository with the ability to double-check changes or roll back to an earlier version if required. This way, the code pushed to the repository can be measured for progress, and then further weekly targets can be created from this.

Chapter 6

Testbed Requirements

6.1 Development methodology

Due to the nature of this project being individual and requiring a lot of set-up, a personal extreme programming (PXP) (Dzhurov et al., 2009) approach would be well suited. It would involve requirement gathering, then iteration of the design, implementation, testing, and review by a single developer for a client, which is also the developer. Further allowing for improvement throughout the project, enabling it to surpass what was initially intended.

6.2 Gathered requirements

The following requirements have been made to meet the research design variables, along with some additional administrative requirements for the user conducting the tests. These will help build the testbed to a usable standard where all of the collected data is reliable for result analysis and conclusion.

6.2.1 Must have

- Measure the time take for operations to complete - read / write / readwrite
- Measure a systems latency - between node to client
- Measure a systems hardware usage - each node
- Measure a clusters amount of nodes

- Sent a shutdown signal to a node and measure if that has occurred
- Bring nodes back online after they have been shut down
- Output results in a human-readable medium

6.2.2 Should have

- A command line interface for user to run simulations from
- Adjustment for variation of system use case
- Indication of test length or remaining time.
- Accurate measurements for all data types
- Average measurements from across the cluster for output

6.2.3 Could have

- GUI based interface for user to interact with
- Provide simple documentation on how to operate

6.2.4 Won't have

- Security implications will not be covered
- Non-English translation

Chapter 7

Testbed and Cluster Design

7.1 Testbed design discussion

7.1.1 Tools

Language for the testbed must be compatible with operating systems needed for MooseFS and GlusterFS. Python would be a good choice for this as it is diverse. It has many packages that can aid in the development and will assist in delivering the requirements effectively.

Many Linux based tools do specific tasks. Due to the number of operations needed to be requested, a tool to remotely run commands is essential. Pssh, known as parallel-ssh, will allow for commands to be piped into hosts directly from the client. This would allow for turning off nodes, measuring the amount online, turning nodes back online and any further administration commands that may be needed. Specific tools for latency include ping, netcat and nmap. Nmap will be used due to the simplicity and versatility it can provide with a quick response.

Commands to support I/O workload generation include FIO, IOR and DD. FIO will be used due to the amount of customisation allowed with the tool. It also provides a very clear transcript for the conducted test, which outputs the time taken in milliseconds. This can be used to generate the workloads and extract the time for results analysis.

Further performance testing of the clusters can be done with an FIO-bench library created by Louwrentius (2022). It comes with another tool called FIO-plot, which

is used for directly displaying the output of FIO-bench into readable graphs. A tool to generate CPU workload is as simple as a stress tool. The tool called stress-ng is precisely needed due to its easy setup and compatibility. Hardware usage will be captured via a Linux daemon. More specifics are mentioned in the cluster software (see subsection 7.2.2, p30).

7.1.2 FIO workloads

Previously mentioned in the research design I/O load (see section 4.8, p20), two different operational load types will be needed. Due to FIO's compatibility, one can simulate the desired I/O workload that is desired. For this project and to fit the research design, the commands will be built as so:

- Load 1
  - 1 User
  - 100MB file size
  - Operation type - random or sequential; read, write and readwrite
- Load 2
  - 10 Users
  - 10 MB file size each
  - Operation type - random or sequential; read, write and readwrite

Both loads will be using 128KB block size segments with an iodepth of 1, meaning 128KB chunks of the file will be sent or requested each request (this will aid results collected due to the hardware limitations to be mentioned further (see subsection 7.2.1, p30). Further options such as direct being set to 1 (true) will help reduce inflated results as it will bypass any RAM caching that may occur with the I/O. The randrepeat option is set true to make sure every FIO random operation is completed in the same way, randomised but repeatable. This will strengthen the results collected. The ioengine is set as libaio, which is the Linux asynchronous I/O library and common across all Linux distributions. Other options such as test name, location of workload and operation type will be configurable within the testbed. Information used to build these commands have been from various sources (Axboe, 2022; LTS_Tom, 2021; Nutanix, 2021).

7.1.3 Stress-ng load

In order to provide varying levels of CPU load during the test against the master architecture, stress-ng has many configuration options available. The total number of master cores need to be specified to simulate the total usage of the CPU. Otherwise, it will focus on a singular core. Usage percentage can be directly put into the command out of 100.

7.1.4 Backend and interface

Having a modular approach to the testbed will allow for expansion if needed. The chosen operating system is Linux based, and it will use a mixture of Python and Unix commands to interact with the host and external systems. A simple text interface should be used to give the user a clear choice of what to run. When the option is chosen, the command will run. This will indicate the progress of the test and may give an indication of time left till the finish.

7.1.5 Automation and error handling

In order to have result collection be as seamless as possible, the testbed will include a complete automated process of running the primary result collection by turning off nodes and bringing them back online. With the use of the Virtualbox, this will be possible. More on this (see subsection 7.2.2, p30) section. In order to make sure the automation can be left unattended, error handling will need to be implemented into the testbed. This will catch errors but not stop the run from finishing. It will log/display where the error was thrown and roughly what type of error occurred.

7.1.6 Use case

A user use case diagram shows how the system is going to interact with a tester. In this figure (see Figure 7.1, p29) the tester will interact with a menu from where commands can be executed. With a main menu and submenu for the FIO commands to be run. Subsequently, all of the commands will be handled by the Python program. The tester will only need to manage the resulting output and the different task configurations within the program.

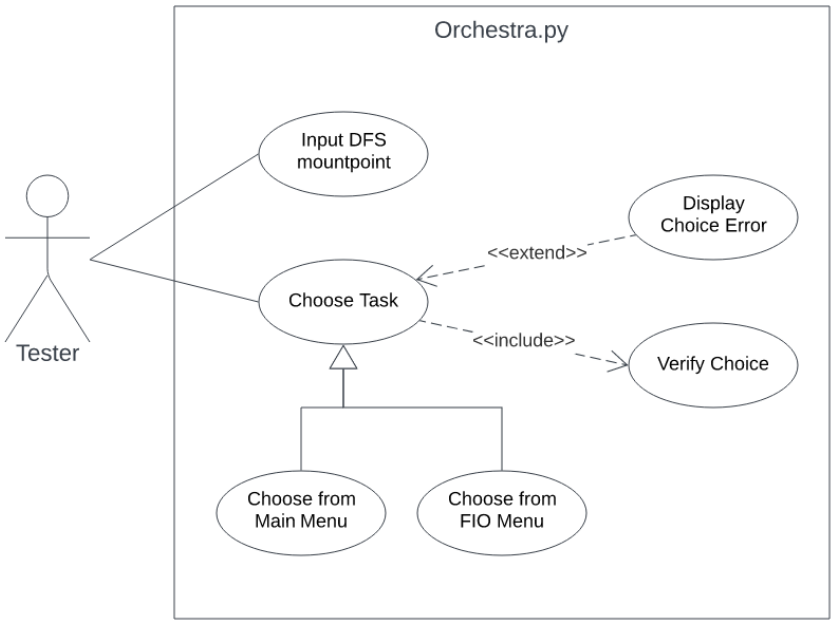


Figure 7.1: Use case diagram of testbed script.

7.1.7 Output

Results need to be clear, easy to understand and well managed. By having the script divide up the tests by a variable name (the type of test), name of the software (Moose or Gluster) and give the time that the script started, it can be well managed and handled when it comes to turning the raw data into readable results. FIO already has a clear output that can be analysed effectively for the time taken. FIO-bench outputs the results into a directory of multiple reports where FIO-plot can generate the differences in IOPS and latency per candidate (see Figure 7.2, p30). Other measurements will be collected with the use of Telegraf, InfluxDB, and Grafana to be mentioned in cluster software (see subsection 7.2.2, p30).

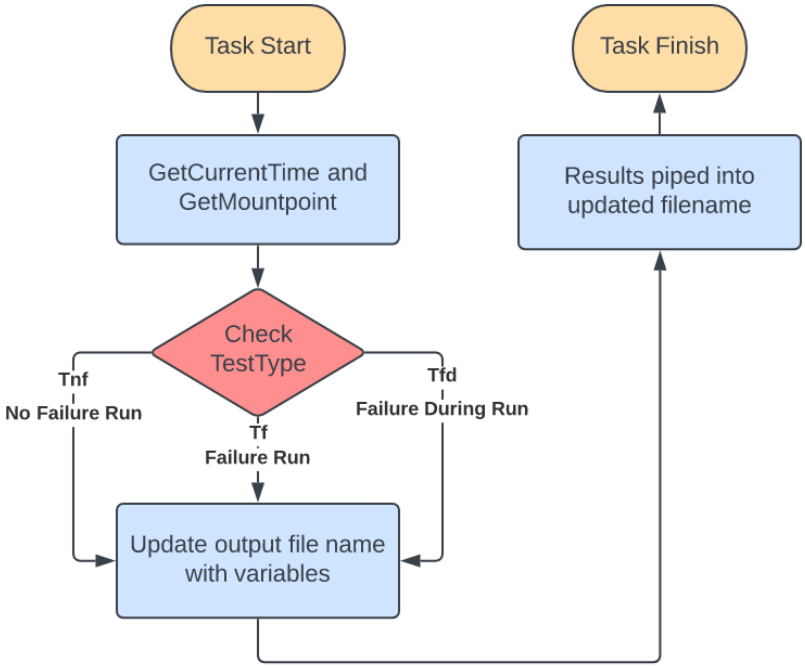


Figure 7.2: File handling and result piping output.

7.2 Cluster design discussion

7.2.1 Hardware

Hardware provided by the university is 8 HP Z230 workstations with two dedicated disks per machine, which are hosted on a private subnet. The network switch is a 100Mbps Netgear FS524. This would mean that overall network speeds are not the best but workable and manageable. Recommended network requirements by Gluster and Moose suggest 1Gbps however, due to the workload test sizes (scaled down for this particular reason), it should be manageable. A lesser spec HP Compaq 8000 SFF machine is being used as a client, where the testbed will be conducted. This is all being hosted inside of the university.

7.2.2 Software

Virtualbox is being used to host two virtual machines on one workstation. It is being done to allow for maximum remote management without being physically in person at the cluster. It further allows for easy image replication and deployment across the

cluster. Each virtual machine will have its dedicated disk on the workstation, which will stop competitiveness for two virtual machines on one disk, mitigating the resulting discrepancy of the I/O workloads. The use of virtual machines also increases the number of available nodes due to the amount of hardware provided.

Hardware usage can be challenging to capture manually. Tools such as Telegraf, Glances and Prometheus can capture many different component areas and pass them to one source in a dashboard-style configuration. For this, Telegraf will be installed on each node, a central InfluxDB server setup on a dedicated HP Z230 rather than the client to reduce the amount of traffic interfering with the I/O workloads. Grafana will be used to display the data on this workstation due to its compatibility with InfluxDB. This will be where the hardware usage can be viewed.

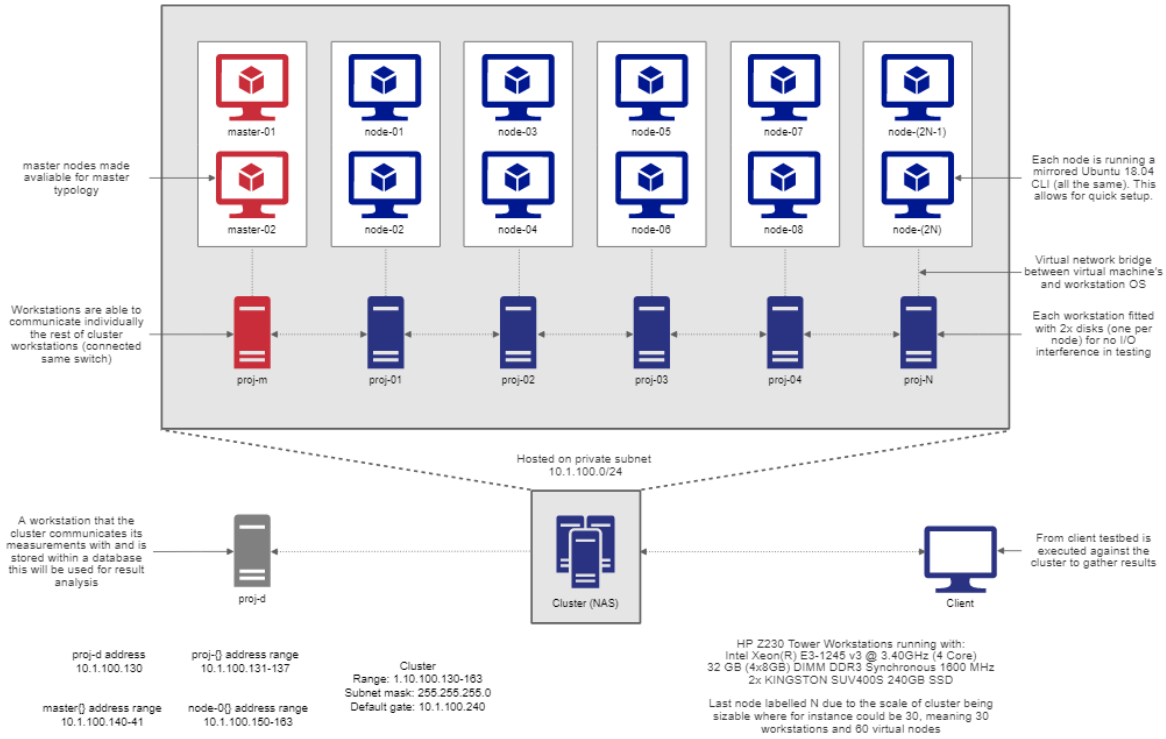


Figure 7.3: Overview of the cluster design with all the machines and virtual machines. Larger view available in appendix (see Appendix E, p84)

7.2.3 Networking

In order to maintain a good record of what machines are doing, a network table has been designed. It is solely to help implementation and troubleshooting of the cluster. Each node will have the cluster addresses and hostnames built into the hosts file, which

will reduce the amount of IPv4 use.

Subnet	10.1.100.XXX/24	
Mask	255.255.255.0	
Default Gateway	10.1.100.240	
TYPE	**HOSTNAME**	**STATIC IPv4**
data station	proj-data	10.1.100.130
management	proj-man	10.1.100.131
workstation	proj-01	10.1.100.132
workstation	proj-02	10.1.100.133
workstation	proj-03	10.1.100.134
workstation	proj-04	10.1.100.135
workstation	proj-05	10.1.100.136
workstation	proj-06	10.1.100.137
...	...	...
virtual management node	master-01	10.1.100.140
virtual management node	master-02	10.1.100.141
...	...	...
client	client	10.1.100.145
...	...	...
virtual node	node-01	10.1.100.150
virtual node	node-02	10.1.100.151
virtual node	node-03	10.1.100.152
virtual node	node-04	10.1.100.153
virtual node	node-05	10.1.100.154
virtual node	node-06	10.1.100.155
virtual node	node-07	10.1.100.156
virtual node	node-08	10.1.100.157
virtual node	node-09	10.1.100.158
virtual node	node-10	10.1.100.159
virtual node	node-11	10.1.100.160
virtual node	node-12	10.1.100.161

Figure 7.4: Assigned static IPv4 addresses to each individual server in the cluster and of the whole project.

Chapter 8

Testbed and Cluster Implementation

8.1 Testbed

Complete testbed code can be seen at this repository (Carter, 2022).

8.1.1 Administration

The first part of the testbed was to get the underlying tools to check the cluster's health and enable the user to turn services on and off. Using the tools mentioned in the design, this was possible by laying out set variables of commands that could be called and run inside of a Python subprocess. Each of these are set to run in shell mode and the output kept in that subprocess (in the background) (see Figure 8.1, p33).

```
1 subprocess.Popen("command", shell=True, stdout=subprocess.PIPE)
```

Figure 8.1: Example subprocess function used to run Linux commands

In order to see what has happened in that subprocess, it can be checked with a `communicate()` function, this allows for pattern matching with regular expressions to determine what next step to take or output. To wait until that process has finished `wait()` is used (without needing to check output), this allows for commands to finish before moving on inside of the script. By using a list of the command variables, a simple number interface can run these and have the result piped out quickly to the

user. This is an example of the interface (see Figure 8.2, p34) and some Python pseudocode of how this occurred (see Figure 8.3, p34). For this project, the current code is specifically tailored to run the set commands for the experiments. However, these commands could be put inside a configuration file if needed in future work.

```
    [Menu]

    1 = Shut off nodes (shutoff-hosts.txt is used here)
    2 = Start telegraf across the cluster
    3 = Stop telegraf across the cluster
    4 = Check the amount of nodes online across the cluster
    5 = Check the amount of nodes offline across the cluster
    6 = Measure average latency of client to cluster
    7 = Start offline nodes (all-nodes.txt is used here)
    8 = FIO command submenu


    S = Print this menu
    Q = Exit program


[Enter] an option value:
```

Figure 8.2: Testbed menu interface.

```
1  item1 = "linux command 1"
2  ...
3  commands = [item1, item2, item3]
4  set = ["1", "2", "3"]
5  While True:
6      option = str(input("Enter item number")
7          if option in set:
8              # subprocess function
9              runSp(commands[(int(option) - 1)])
10         else:
11             print("Invalid input")
```

Figure 8.3: Python pseudo of selecting and running commands available.

By using regular expression matching on the stdout, text output to the user can be quickly analysed and piped automatically into files for the user to gather. As stdout is in byte form, to provide readable, clear text to the user, this must be decoded. It was done with `decode("utf-8")`. This Python pseudocode figure describes an example of how results were piped out for both ways in the script (see Figure 8.4, p35).

Using these principles of subprocess and outputting the administration implementation into the testbed was straightforward. However, this is not the best method for implementation and could be improved. Further use of subprocess for FIO and Stress-ng has been used for these areas of the testbed.

```
# stdout is standard output
# stderr is standard error output
run = subprocess.Popen("command", shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
out = run.communicate().decode("utf-8")
match = re.findall("expression", str(out))
if len(match) != 0:
    print("output to terminal")
    # write to a file function
    outputFile(out)
```

Figure 8.4: Python pseudo handling the output of the subprocess and sending to terminal or file.

8.1.2 FIO and Stress-ng

FIO was set up with the use of subprocess. Again variables were set to the individual commands for read, write and readwrite of random or sequential operations. It allowed them to be iterated in sequence, which further allowed them to be repeatable. In order to make sure the DFS mount point was being targeted, a global mount point variable was used. With the use of the `format()` function, this would allow it to be changeable very quickly for all the commands. Here is an example FIO command inside of a Python pseudocode (see Figure 8.5, p35).

```
mountpoint = "/mnt/dir"
fioCmd = "fio --randrepeat=1 --ioengine=libaio --direct=1 --name=setTest-{0} --filename={0}/test --bs
    =128k --size=100M --readwrite=randread --ramp_time=4".format(mountpoint)
run = subprocess.Popen(fioCmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
out = run.communicate()
```

Figure 8.5: Output of all FIO results being output to one readable block and to a csv file.

With this, multiple variables were set up inside the script to iterate through and run commands against the mount point. The output of those commands is directly piped into text files using some Linux side commands. When sending the FIO output to the file, the `tee` Linux common is appended to the end of the FIO command. This uses the operating system to pipe the results rather than the Python program itself. When the

iteration of commands is complete, an overview of the run is generated. It grabs all of the generated files from the run, opens and matches for the time taken (milliseconds), calculates the minimum, average and maximum, then pipes it into a printable string and outputs it to a file. This is done explicitly in CSV format for easy upload to spreadsheets for analysis and data plotting (see Figure 8.6, p36). Any errors during the run are caught but do not stop the program from running. It will let the user know where an error occurred and on what command/iteration. FIO-bench has been implemented similarly, where execution and output are done from the program.

```
output = "‘‘‘‘"
output+= "\nMountpoint = {0}, Mode = {1}, Failures = {2}, Time = Msec, Output Mode = csv".format(mp, file, num_lines)
output+= "\ntestType,min,avg,max"
# opens all the files that has been generated and puts them into an output string
for i in range(len(commands)):
    results = open("fio-{0}-test-{1}.txt".format(name, commands[i][0]), 'r')
    match = re.findall("-(\d*)msec", results.read(), re.DOTALL)
    if len(match) != 0:
        match = list(map(float, match))
        # formatting the found -(\d*)msec with the FIO run type
        output += "\n{0},{1},{2:.1f},{3}".format(commands[i][0], min(match), mean(match), max(match))
        print("{0}:\tmin/avg/max {1}msec/{2:.1f}msec/{3}msec".format(commands[i][0], min(match), mean(match), max(match)))
# caught errors are added with their iteration and error type
output += "\nCaught Errors: {}".format(caughtErrors)
output += "‘‘‘‘"
out = sp.Popen("echo '{0}' > fio-{1}-test-overview-{2}.csv".format(output, name, file), shell=True, stdout=sp.PIPE)
out.wait()
```

Figure 8.6: Output of all FIO results being output to one readable block and to a csv file. Discord output (see Figure 8.11, p40)

A package called alive-progress was included to add a progress bar to the interface. This was to remove the need to wait around at the terminal. It greatly improves the user experience of the script.

Stress-ng was implemented by using a pssh command that pipes in stress-ng to the remote host. Due to stress-ng not being able to be spawned in a background process on Linux itself, the Python testbed needed to “spawn” the stress-ng on the remote system and then run another check to see if stress-ng had been deployed. Using a command called pgrep would provide the process id associated with the process name searched. This allowed it to match if there was or was not an output (see Figure 8.7, p37).

```
print("[Alert] Running stress-ng")
# argument variables grabbed from config.ini
stressNg = "stress-ng -c {2} -l {0} -t {1}".format(loadLevel, loadTime, processors)
sp.Popen(sshPass(psshCommand(stressNg, server)), shell=True, stdout=sp.PIPE, stderr=sp.PIPE)
print("[Sleep][---] 10s to spawn cpu hogs")
time.sleep(10)
print("[Alert] Checking process is running")
pgrepCommand = "pgrep stress-ng"
run = sp.Popen(sshPass(psshCommand(pgrepCommand, server)), shell=True, stdout=sp.PIPE, stderr=sp.PIPE)
out = run.communicate()[0].decode("utf-8")
# checking if there is an output of pid's or not
match = re.findall('\[SUCCESS\]', str(out))
if len(match) != 0:
    print("-> Stress-ng executed on remote system, running for {} time. Ready for FIO testing!".format
     (loadTime))
else:
    print("-> Stress-ng did not execute")
```

Figure 8.7: Stress-ng launching and checking if the process is running on the remote host.

Variables for the stress-ng command were used globally within the side module and could be edited within a configuration file. From here, the user is now able to conduct the testing they need to do to gather results against the DFS software.

8.1.3 Backend

Lots of pssh commands were used to run remote commands on the needed machines. To keep password usage to a minimum sshpass was incorporated into the list of Linux commands. This stores a password in a file on the host, which allows for the pssh command when asking for the password to be passed by sshpass (see Figure 8.8, p37). This further aided the process in the automation subsection.

```
subprocess.Popen("sshpass -f pass_file  pssh {args}", shell=True, stdout=subprocess.PIPE, stderr=
    subprocess.PIPE)
```

Figure 8.8: Python subprocess example of sshpass being used before pssh command for passwordless running.

With a list of global variables needed to let the testbed run effectively, they initially were stored in variables on the script and had the user input them on launch. It did not work well when needing to work with multiple files, as it would reset when launching into the second Python file. With the use of a package called configparser a file could

be used to store and update the variables to the script. The testbed was set up to write changes to this file. This further allowed the change of FIO command type to be passed onto the secondary script where FIO workloads were run.

```
[default]
mountpoint = /mnt/mfs
tfd = False
tnf =True
tf = False
tnfList=[11925.3,6486,10031,11602,6449.3,9009.7,9571.3,5085.7, 3836.7,5171.3,9569,4330]
loadLevel=50
loadTime=3h
stressCores=3
```

Figure 8.9: Configuration file with changeable variables from inside the testbed and manually by the user themselves.

8.1.4 Automation

Due to the tests measuring the time taken for operations to complete and with the number of tests being run, the collection process would take an hour or two. Ultimately down to the delay added to allow the cluster time to catch up after nodes are turned off and back on (not the case during simulating failure during operations). To combat the time wasted waiting, some steps were taken to automate the program.

With the use of the configuration file, the gathered time taken under no failure was put into a list and could be parsed by the testbed to turn off nodes after the allotted time. When FIO has finished its run, the testbed turns on the offline nodes, ready for the next FIO iteration. It allowed for complete automation of tests run with only the output files needing to be moved after each complete test (see Figure 8.10, p39).

To be notified about the program finishing while away from the terminal, a Python package called knockknock was used to send a notification to Discord via webhook. This greatly improved the experience of the collection due to being able to work on other areas while tests were running in the background. Error handling was added to output via Discord and to create an additional error file. This tells the user exactly where errors were caught during the FIO runs. It does not stop the testbed from continuing but logs and moves on (see Figure 8.11, p40). It will further help during result analysis when interpolating the gathered results.

```
# Checking the test type, will be eiter one or the other or none
if Tfd == True:
    print("-> Using shutoff-hosts.txt to turn off host(s) mid task.")
    print("-> Nodes will be shut off Tnf/4 time each run and brought back online automatically.")
if Tf == True:
    print("-> Using shutoff-hosts.txt to turn off host(s) for FIO commands")
    command = "pssh -i -h shutoff-hosts.txt -A -l root -x '-tt -q -o StrictHostKeyChecking=no -o
     GSSAPIAuthentication=no' 'hostname; sleep 1; init 0'"
    sp.Popen("{0}{1}".format(sshPass,command), shell=True, stdout=sp.PIPE)
    print("-> Shutting off node(s)")
    print("[Sleep][---] 60s to allow cluster catchup...")
    time.sleep(60)
# alive-progress loop where the FIO tests are iterated within
with ap.alive_bar(3, title="-> FIO {0} tests\t".format(commands[index][0]), bar='classic2', spinner='
     classic', enrich_print=False, elapsed=False) as tar:
    # run each test 3 times
    for j in range(3):
        if Tfd == True:
            # Turn off node mid task, tnfList used here from config
            # (avg Tnf / 4) / 1000 for seconds
            specTime = ((tnfTimeList[index] / 4) / 1000)
            command = "pssh -i -h shutoff-hosts.txt -A -l root -x '-tt -q -o StrictHostKeyChecking=no -
     o GSSAPIAuthentication=no' 'hostname; sleep {}; init 0'".format(specTime)
            sp.Popen("{0}{1}".format(sshPass,command), shell=True, stdout=sp.PIPE)
            print("-> Shutting off node(s)")
        # Executing the fio command specific to the iteration / run
        fioRun = sp.Popen("{0}".format(commands[index][1]), shell=True, stdout=sp.PIPE, stderr=sp.PIPE)
```

Figure 8.10: Testbed automation snippet, different modes both automated to be left to run.

Figure 8.11: Testbed output with notification on Discord, notifiying when the test is complete.

8.2 Cluster

8.2.1 Hardware

With the help of the university system administrators, the cluster was set up inside of Anglesea building. Remote access was set up not to have to be at the machines in person, also allowing testing to be conducted at any hours of the day. Administrator access over the system was granted to allow complete freedom of set up and to solve any troubleshooting problems.

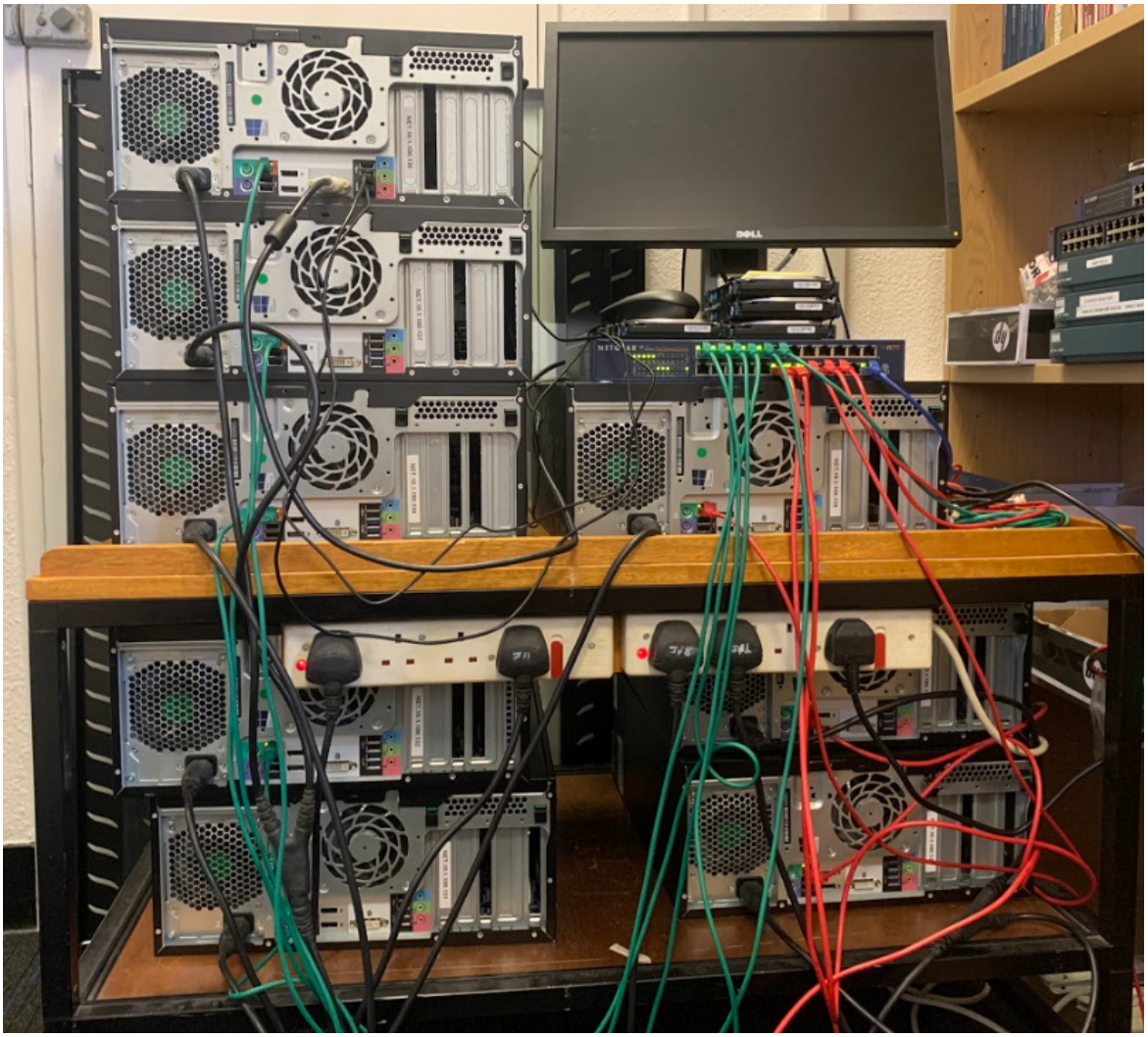


Figure 8.12: Physical cluster machines inside of Anglesea building.

8.2.2 Workstation setup

Each workstation was pre-installed with Centos 7, VirtualBox and Xrdp to allow remote access from the Windows RDP app. With the two dedicated disks on the machine, the extra disk was formatted to XFS, then fstab setup with the mounting point. Another folder was created in `/mnt/` to keep the location of the virtual machines relative for ease of access. Hostnames and addresses of the cluster were added to `/etc/hosts` to remove any need for a DNS server or service. This was bundled into a bash script that assisted in the speed of setup across the six machines `proj-01` to `proj-06` (see Figure 8.13, p42). A further external drive was added to `proj-data` to store collected results before being uploaded to the cloud; this was to mitigate any data loss.

```
echo "What is the new hostname for this system? (be careful upon input)"
OLDHOST=$(hostname)
read NEWHOST
# update hosts and hostname with new hostname
sed -i "s/127.0.1.1 ${OLDHOST}/127.0.1.1 ${NEWHOST}/" /etc/hosts
sed -i "s/${OLDHOST}/${NEWHOST}/" /etc/hostname
# install packages and check if they are running
yum -y install VirtualBox-6.1
yum -y install xrdp
systemctl enable xrdp
systemctl start xrdp
systemctl status xrdp

# run fdisk to manually wipe then format, mount
# and make directories for the virtual machines
fdisk /dev/sdb
umount /dev/sdb1
mkfs.xfs /dev/sdb1 -f
mkdir /mnt/500GB; mount /dev/sdb1 /mnt/500GB; mount | grep /dev/sdb1;
chmod ugo+rwx /mnt/500GB; chmod ugo+rwx /mnt/main;

# the final tree look being
# /mnt/
# ├── 500GB
# │   └── node-07
# │       └── Logs
# └── main
#     └── node-08
#         └── Logs
reboot
```

Figure 8.13: Testbed automation snippet, different modes both automated to be left to run.

8.2.3 Virtual image setup

A base image for the virtual machines was created in VirtualBox, 3x cores, 8GB of RAM and two virtual disks were added. Ubuntu 18.04 server (CLI only) was installed along with any packages needed to run the system at minimal specs. Telegraf was installed to collect hardware metrics along with configuration to point to InfluxDB. More on this (see subsection 8.2.4, p43). Again `/etc/hosts` was edited to include the cluster addresses as well as assigning the base image an unused static IP. Placed on the base image was a setup bash script that could be used to quickly generate new node hardware ids and assign the hostname to its static set address (see Figure 8.14, p43). These additions would aid setup and deployment time.

```
# this script must be run as 'sudo ./setup.sh'
echo "This is a shell setup script"
echo "Regenerating machine ID"
# this stops virtual bridging issues where the same ip is assigned to the clone of the original
# the netplan file also has 'dchp-identifier: mac' added inorder to mitigate the issue straight away after deployed
# this is solely to get it working with multiple cloned virtualboxes however could be of use for virtualisation softwares
rm -f /etc/machine-id
dbus-uuidgen --ensure=/etc/machine-id
rm /var/lib/dbus/machine-id
dbus-uuidgen --ensure
echo "Regen done!"
echo "What is the new hostname for this system? (be careful upon input)"
OLDHOST=$(hostname)
read NEWHOST
sed -i "s/127.0.1.1 ${OLDHOST}/127.0.1.1 ${NEWHOST}/" /etc/hosts
sed -i "s/${OLDHOST}/${NEWHOST}/" /etc/hostname

# this assigns a static ip address for the node which is predetmined according to the design of the network
echo "What is the static ip address needed for this server?"
echo "Refer to the network design for the specific ip address for this node"
echo "10.1.100.XXX - Where the entered XXX is the assigned ending"
read STATIC
sed -i "s/- 10.1.100.149\/24/- 10.1.100.${STATIC}\/24/" /etc/netplan/00-installer-config.yaml

reboot
```

Figure 8.14: Base image with included setup script for quick deployment

8.2.4 InfluxDB and Grafana

Both InfluxDB and Grafana were set up locally on the `proj-data` machine. InfluxDB was set up with an infinite data retention period to mitigate any data loss throughout the project. InfluxDB was then linked with Grafana, where the data could be queried to form readable graphs, a baseline template for InfluxDB was used and altered to filter the specific data. This being set up allowed for each node that had the Telegraf agent installed to send data to the dashboard. Telegraf config can be seen at this repository (Carter, 2022).

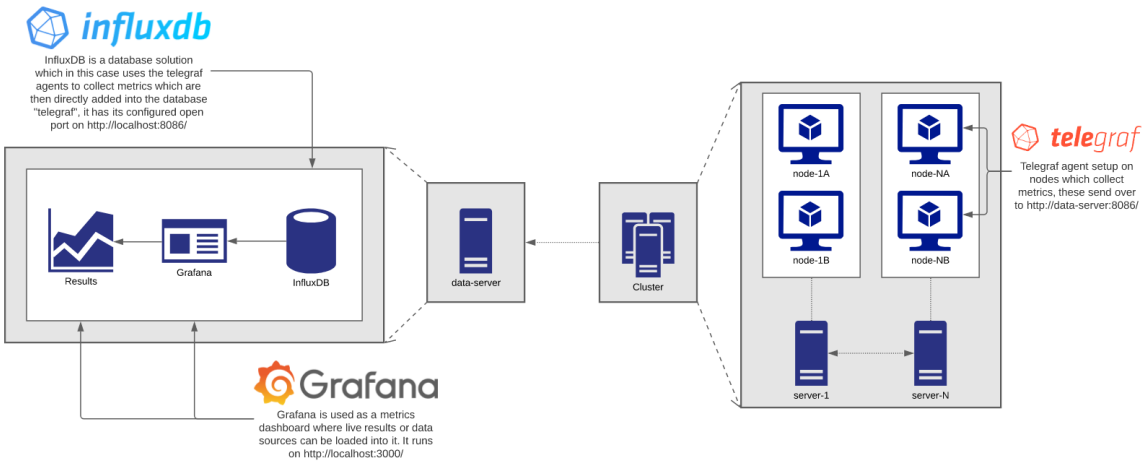


Figure 8.15: Telegraf, InfluxDB and Grafana flow.

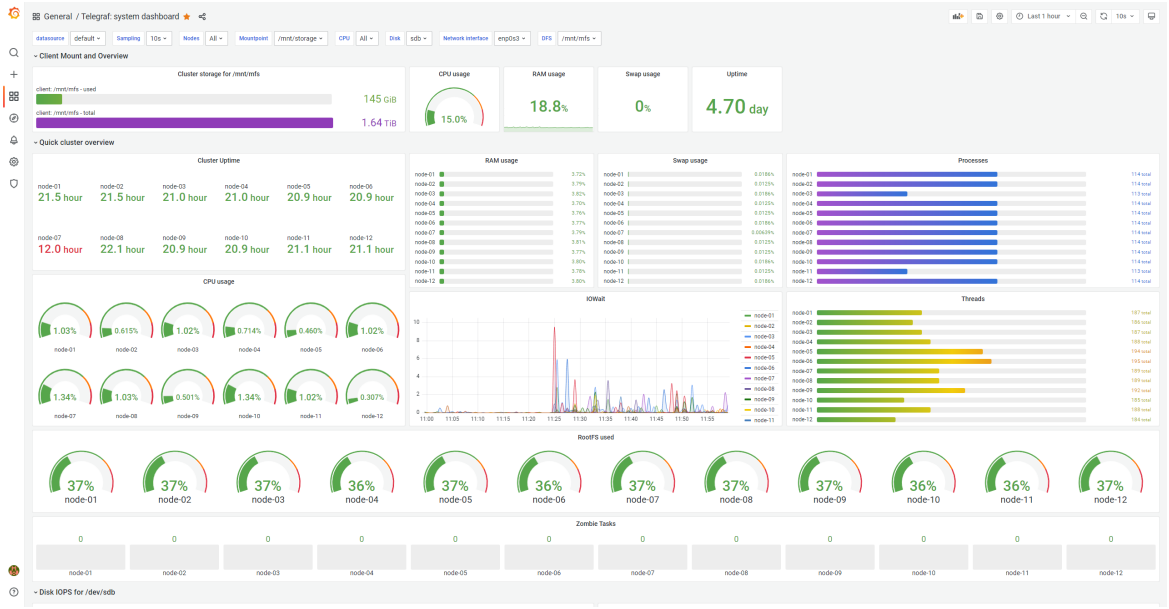


Figure 8.16: Set up Grafana dashboard where hardware metrics of the cluster will be captured.

8.2.5 MooseFS

(MooseFS, 2021)

Moose was set up across three virtual base images, one for master, one for meta logger and one for chunk server. Each image had the necessary packages installed, configuration files left as default (except to point to the master server) and saved as the image that needed to be deployed. Chunk servers specifically had their second

virtual disk formatted to XFS and mounted to `/mnt/storage`. This was to be each individual chunk server's shared storage to the cluster. The files were distributed out to the machines necessary and then, one by one, deployed with the help of the setup script that was built into the original base image. Due to having a fixed IP for the base image, one terminal could be used to continuously run the script and relaunch to the next base image when that was brought online. When online with all the chunk servers connected to the master, `mfscli` could be used to check the status of those connected servers (see Figure 8.17, p45). Nothing volume-related would be changed as Moose requires the client to specify the file goal that is needed, Moose Pro (paid) has erasure coding features that are locked on normal MooseFS.

```
droid@master-01:~$ mfscli -S -CS
```

Chunk Servers

ip/host	port	id	labels	version	load	maintenance	'regular' hdd space				'marked for removal' hdd space				
							chunks	used	total	% used	status	chunks	used	total	% used
node-01	9422	2	-	3.0.116	1	off	427	13 GiB	140 GiB	8.95%	-	0	0 B	0 B	-
node-02	9422	1	-	3.0.116	0	off	437	13 GiB	140 GiB	8.97%	-	0	0 B	0 B	-
node-03	9422	3	-	3.0.116	0	off	411	13 GiB	140 GiB	8.97%	-	0	0 B	0 B	-
node-04	9422	4	-	3.0.116	1	off	422	13 GiB	140 GiB	8.98%	-	0	0 B	0 B	-
node-05	9422	6	-	3.0.116	0	off	438	13 GiB	140 GiB	8.94%	-	0	0 B	0 B	-
node-06	9422	5	-	3.0.116	0	off	426	13 GiB	140 GiB	8.97%	-	0	0 B	0 B	-
node-07	9422	8	-	3.0.116	1	off	434	12 GiB	140 GiB	8.93%	-	0	0 B	0 B	-
node-08	9422	7	-	3.0.116	1	off	454	13 GiB	140 GiB	8.93%	-	0	0 B	0 B	-
node-09	9422	10	-	3.0.116	0	off	446	13 GiB	140 GiB	8.94%	-	0	0 B	0 B	-
node-10	9422	9	-	3.0.116	0	off	425	12 GiB	140 GiB	8.81%	-	0	0 B	0 B	-
node-11	9422	11	-	3.0.116	0	off	439	13 GiB	140 GiB	8.97%	-	0	0 B	0 B	-
node-12	9422	12	-	3.0.116	0	off	422	13 GiB	140 GiB	8.99%	-	0	0 B	0 B	-

Figure 8.17: MooseFS cluster set up, mfscli command used to view chunk servers.

Once the cluster side of Moose was set up, a client could be connected via a mount point. Specifically, for the duration of the tests, the mount point `/mnt/mfs` would be used. Permissions for the mount directory are controlled by the master server, meaning the client would not have to change the folder's permissions to access it once mounted. When mounted via `mfsmount` and the file goal set, the client can send file I/O to the cluster. Any options for caching and replication goals are set via the client, not the master (see Figure 8.18, p45). Complete configuration and process steps can be found in this repository (Carter, 2022).

```
# set nocache when mounting
sudo mfsmount /mnt/mfs -H master-01 -o mfscachemode=NONE
# set replication goal 4
# df -h
# Filesystem                      Size  Used Avail Use% Mounted on
# master-01:9421                  1.7T  9.0G  1.7T   1% /mnt/mfs
sudo mfssetgoal -r 4 /mnt/mfs
```

Figure 8.18: MooseFS client mounting and replication goal.

8.2.6 GlusterFS

(Drake, 2020; Gluster, 2022)

Similarly, the base image was installed with Gluster server software. The second virtual disk was formatted to XFS and then set to `/mnt/storage`. Once the image was ready, it was distributed and deployed to the necessary machines. Once all machines were online and individually named, a Gluster server was chosen to run the commands. The cluster was then formed by peer probing each node running the Gluster server software. Due to security not being a factor in this project, this was very straightforward. In a real-life setting, each node would have had firewall rules set up to allow this to occur (see Figure 8.19, p46).

```
sudo gluster peer probe node-{}
```

Figure 8.19: Gluster probe command to add node to the pool

```
root@node-01:~# gluster pool list
UUID                                    Hostname        State
52fb27ae-0a8f-4a5e-91d8-6a2ef3b69d04    node-02         Connected
82591bc7-ef85-415e-be13-0999f99e768f    node-03         Connected
91ad4453-2dee-439e-88e2-a71a97dcb28f    node-04         Connected
2d130d24-ae46-47cd-b30f-77a719c2c464    node-05         Connected
852e7b6a-8011-4934-bf85-4daa07996e7a    node-06         Connected
53cc8054-adfa-4f04-879b-7a7f8cfffd3f    node-07         Connected
5536c376-1822-437c-93e5-95ec5a28f076    node-08         Connected
2743d474-f4f0-4fe1-9088-3d471f14e577    node-09         Connected
a411221e-01ff-41a5-83d9-116b220fd585    node-10         Connected
f90aee4c-9c55-495e-9d9f-f71661d4ab25    node-11         Connected
8ac6c3bf-de64-4d72-bf1b-0dd422426bb4    node-12         Connected
e8003d2c-02a0-4b39-b2f7-364bcf28e784    localhost       Connected
```

Figure 8.20: Gluster pool list of connected nodes.

After all Gluster servers have peered together, the volume type is set up. During this, the nodes and shared storage points are specified along with the replication amount (that being 4). When generated, the volume is started, along with the caching settings disabled (see Figure 8.21, p47). The volume is ready to have clients connect, which requires Gluster client software installed on the client machine. For the mount point, a similar mounting name `/mnt/gfs` is used. After mounting, the client can send file I/O. All cluster options are kept away from the client, which is configurable on any number of the peered Gluster servers. Complete configuration and process steps can be found in this repository (Carter, 2022).

```
root@node-01:~# gluster volume info gfs

Volume Name: gfs
Type: Distributed-Replicate
Volume ID: db9f3408-7704-411d-a611-de26664139a8
Status: Started
Snapshot Count: 0
Number of Bricks: 3 x 4 = 12
Transport-type: tcp
Bricks:
Brick1: node-01:/mnt/storage
Brick2: node-02:/mnt/storage
Brick3: node-03:/mnt/storage
Brick4: node-04:/mnt/storage
Brick5: node-05:/mnt/storage
Brick6: node-06:/mnt/storage
Brick7: node-07:/mnt/storage
Brick8: node-08:/mnt/storage
Brick9: node-09:/mnt/storage
Brick10: node-10:/mnt/storage
Brick11: node-11:/mnt/storage
Brick12: node-12:/mnt/storage
Options Reconfigured:
performance.flush-behind: off
performance.write-behind: off
performance.io-cache: off
diagnostics.count-fop-hits: on
diagnostics.latency-measurement: on
transport.address-family: inet
storage.fips-mode-rchecksum: on
nfs.disable: on
performance.client-io-threads: off
```

Figure 8.21: Gluster volume information.

8.3 Issues observed

Throughout the journey of setting up both testbed and cluster for this project, multiple hiccups have occurred. Most have been resolved or mitigated. For example, when testing that the testbed could measure FIO runs accurately with failure introduced, it was essential to know what FIO errors occur and if such errors are produced during the result collection process. It required some iterative testing by looking into the FIO errors and the system errors that can occur. Doing this allowed for regular expression matching of the correct error messages. These needed to be logged with the results so that discrepancies are noted clearly.

Due to the project's length and additions to the code carried out over a few months, the original part of the testbed was poorly optimised and set out. When automation was implemented, a new module script was added to the original testbed, allowing the code to be further optimised and improved upon without scrapping the original.

Aspects of MooseFS CPU stressing had to be troubleshot due to the command launched requiring an active terminal where it was not possible to send it to a background screen without crashing. This led to the use of subprocess without a `wait()` function. This

would allow for a pssh connection to be held in the background. It was important to alter the pssh command further to include a non-timeout on itself as, by default will halt after 60 seconds of being left without any output. Thus the stress-ng command was possible in the end.

Further issues with cluster setup included the time taken for the sourcing of hardware and making sure that all the workstations sat equally in hardware power as each other, `proj-2` was noted to be missing 16GB of RAM when first set up of MooseFS was occurring. This was done by double-checking the hardware components via the lshw command after unsimulated nodes were crashing.

DNS originally was not considered. When it came to wanting the hostnames of each server being "pingable" from every cluster node, `/etc/hosts` came into the scope and solved the issue quickly. However, there would be issues if there were a significant network change to the addresses. The nodes being statically bound on each image during the setup script would require either another full-scale deployment or a newer script to edit the current static address to a new IP.

The storage size of the cluster looks to be smaller for Gluster than Moose. Gluster takes into account the shrinkage of the overall volume. Moose is different due to the ability to set multiple directories with varying replication goal sizes. It is harder to grasp quickly how much storage space is being allocated for Moose rather than Gluster. It could cause confusion during the comparison of volume size between different software.

It should be noted that after some I/O speed testing that MooseFS reads are slower than writes. Investigation into this revealed that there is an implemented unconfigurable write-back cache enabled (Wieliczko, 2015). This should not impact the results due to the I/O time completing client-side, yet it is not known if the file is completely written to the chunk servers during the time of failure. Due to these tests focusing on the time taken to complete I/O rather than file integrity, this is left alone. Although for further research into file completion, fsync option should be considered within the FIO command.

Chapter 9

Results and Discussion

All results were gathered well on time. Each time taken operation was run three times with the average taken for the graphical representation. Due to the FIO raw output being in milliseconds, it has been converted to seconds for ease of interpretation. Some results have had values kept out of the graphs, which would skew the analysis with clear outliers; others had tests rerun to check if anomalies were part of the actual result. For example "time during failure" tests had clear timeouts where it was not representative to include on the graphs. The scale of failure in "after failure" and "during failure" results has been kept the same. It allows for a clear interpretation of trends that can be seen. See the following pages for results gathered from MooseFS and GlusterFS. Previously mentioned (see section 8.3, p48), writes whilst "time during failure" has been left included as this is representative for I/O against the mount point. However, data integrity is possibly unknown. Raw collected data can be seen at this repository (Carter, 2022)

9.1 Fault tolerance of MooseFS

9.1.1 Normal operational time

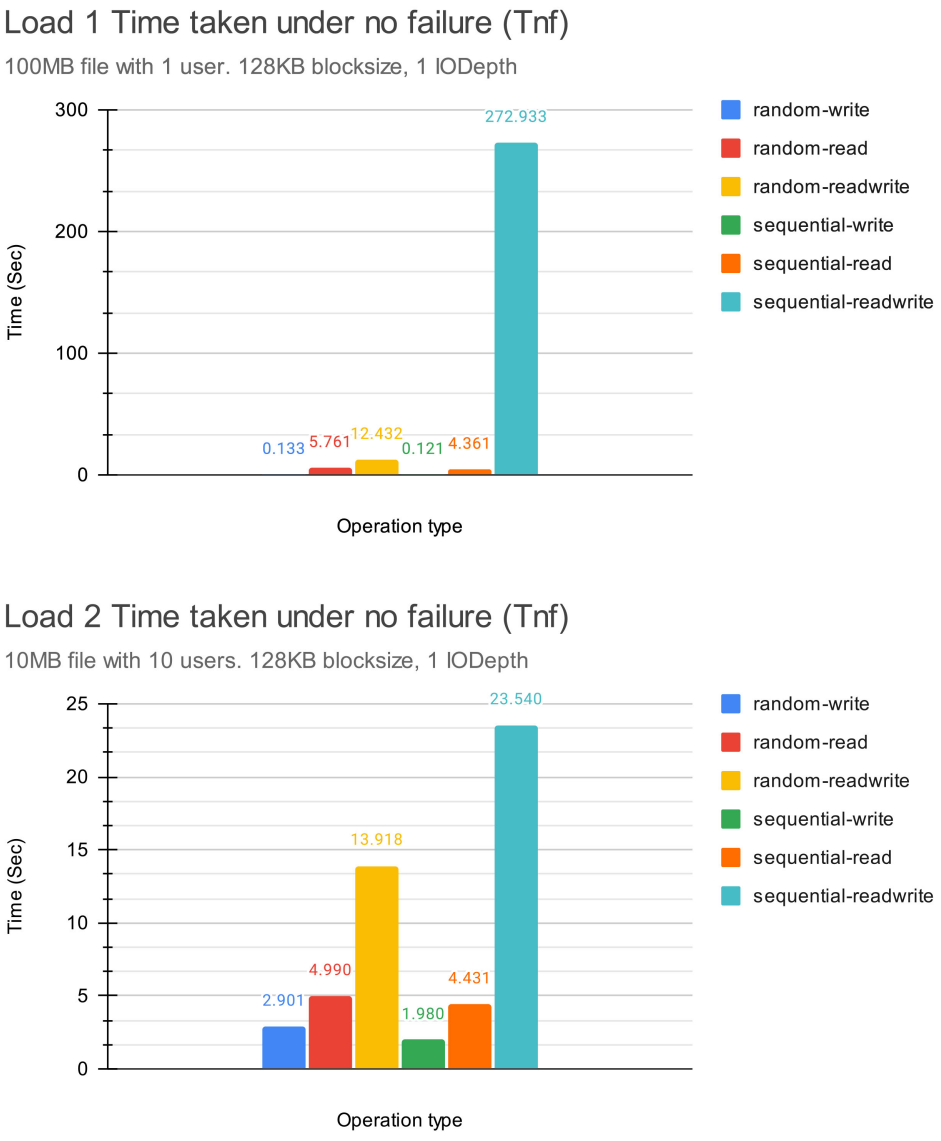


Figure 9.1: Load 1 & 2 average time taken for operations to complete with no failure (Tnf) - MooseFS.

The presented results operational "time under normal" circumstances of each load is swift, albeit the sequential readwrite of load 1. However it does show that MooseFS can handle file I/O effectively and efficiently. This represents **Test 1** (see subsection 4.6.1, p18).

9.1.2 Operational time after failure

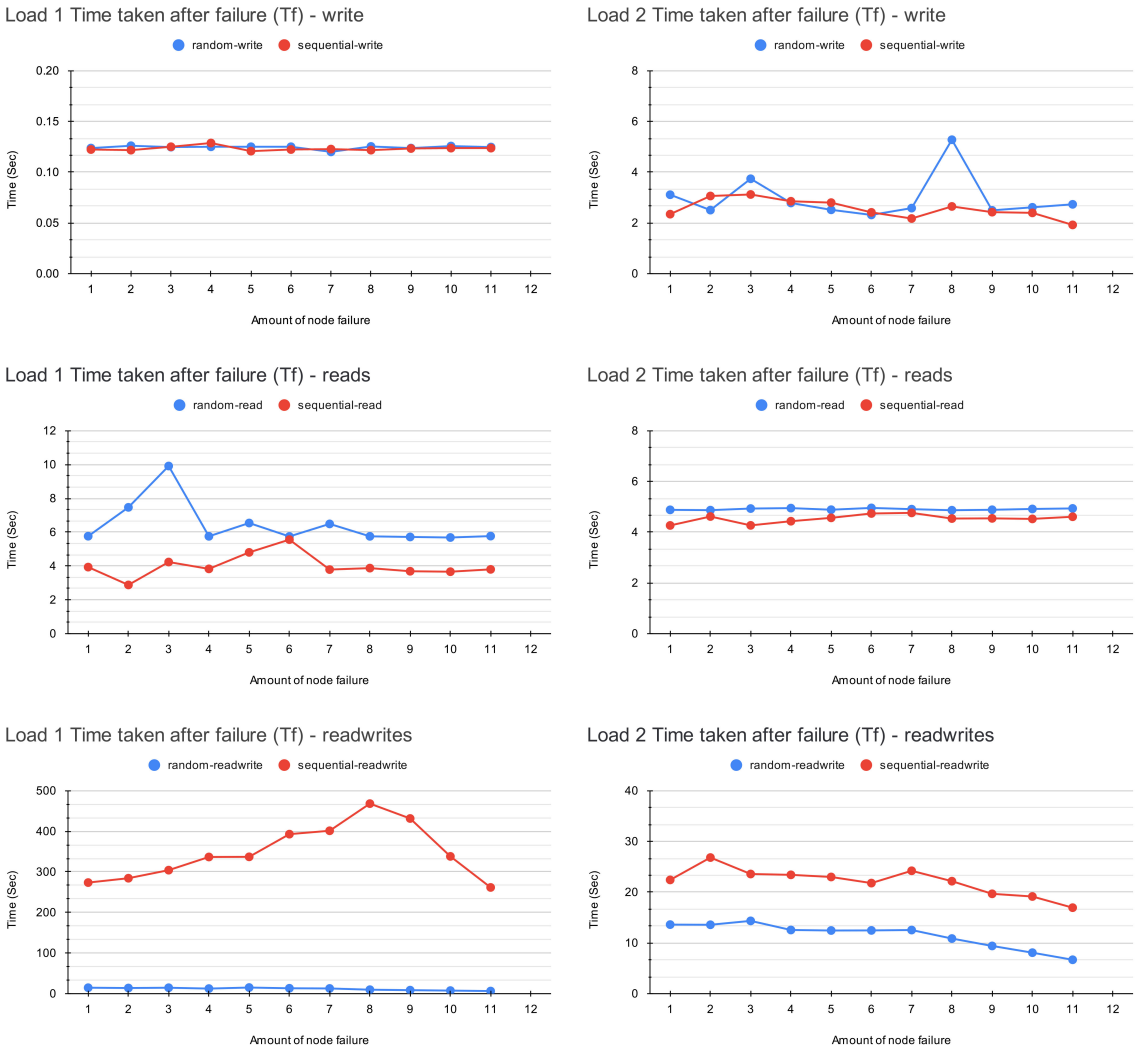


Figure 9.2: Load 1 & 2 average time taken for operations to complete after failure (Tf) - MooseFS.

Looking at the "time after failure", it is clear that once the cluster has had some time to readjust to the loss of nodes, it can still deliver the speeds close to or faster than the standard time. This is due to the reduced amount of nodes needing to communicate within the cluster. Again sequential readwrite seems to be a struggle for load 1 on MooseFS and there are some discrepancies where data is taking a little longer than expected to finish with the operation. It is unknown specifically as to why but may be down to inter cluster operations being affected by latency due to previous node failure. This represents **Test 2** (see subsection 4.6.1, p18).

9.1.3 Operational time during failure

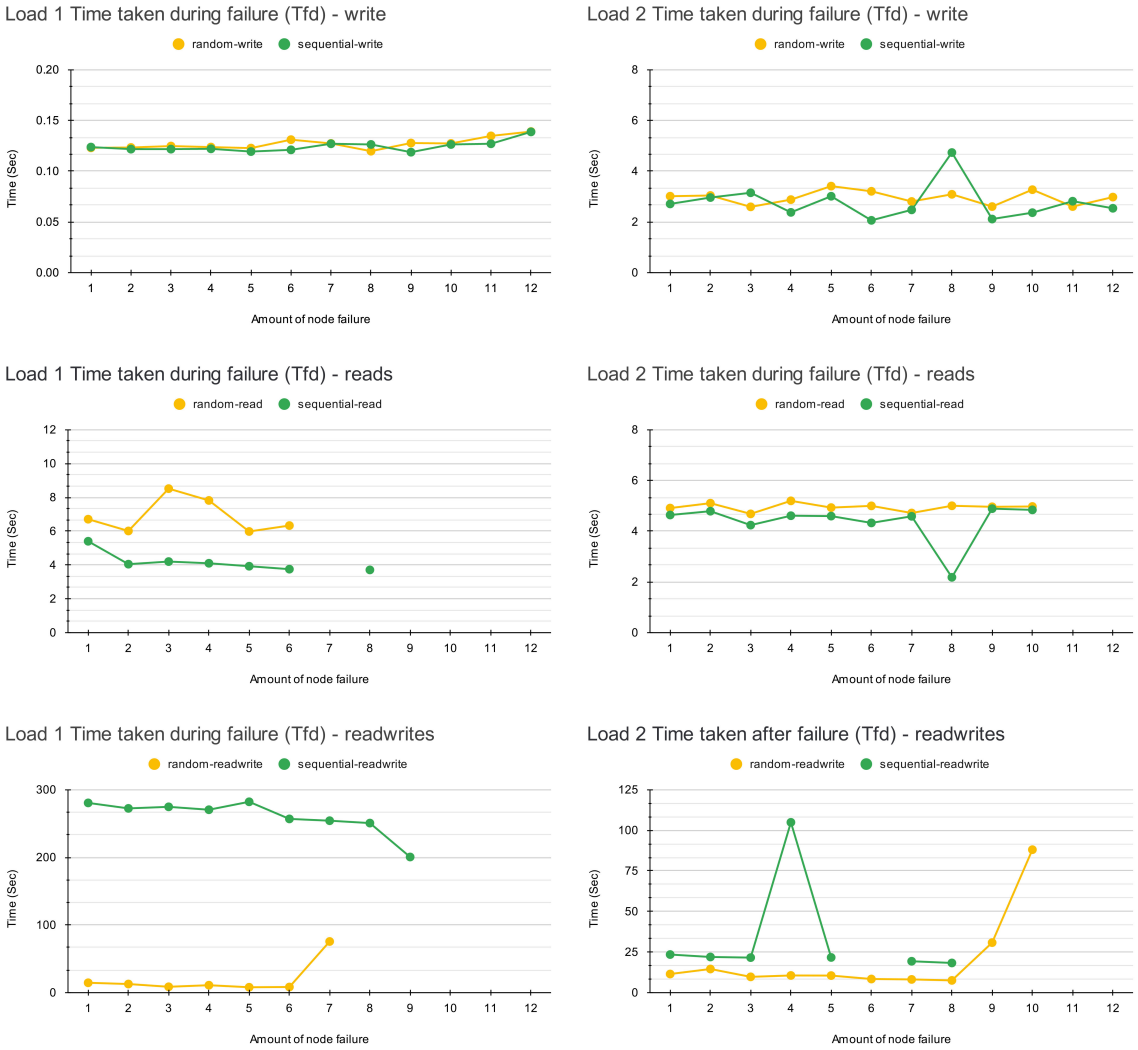


Figure 9.3: Load 1 & 2 average time taken for operations to complete during failure (Tfd) - MooseFS.

Within "time during failure" the operational trend is similar to that of "time after failure" (see subsection 9.1.2, p51). However it is clear that some operations such as reads and readwrites with both loads struggle to complete in all tests. Load 2 reads has an unexpected discrepancy that may point to data being lost, the same with load 2 random readwrites, where it may be delayed due to the specific node in use failing during that request. It can be seen that the results are a little more sporadic or null with more failure, this is due to timeouts or operations being interrupted by the failure. The null values in this case is due to operations timing out. This represents **Test 3** (see subsection 4.6.1, p18).

9.1.4 Timeout errors caught during operation testing

Operation type	Node failure											
	1	2	3	4	5	6	7	8	9	10	11	12
random-write	0	0	0	0	0	0	0	0	0	0	0	0
random-read	0	0	0	0	0	0	2	3	2	3	3	3
random-readwrite	0	0	0	0	0	0	1	3	3	3	3	3
sequential-write	0	0	0	0	0	0	0	0	0	0	0	0
sequential-read	0	0	0	0	1	0	2	0	3	3	3	3
sequential-readwrite	0	0	0	0	0	1	0	0	1	2	2	3

Table 9.1: Load 1 timeouts caught during failure: 0/3 best case, 3/3 worst case - MooseFS.

Operation type	Node failure											
	1	2	3	4	5	6	7	8	9	10	11	12
random-write	0	0	0	0	0	0	0	1	0	1	0	0
random-read	0	0	0	1	0	0	0	0	1	1	2	3
random-readwrite	0	0	0	0	0	0	0	0	1	1	2	3
sequential-write	0	0	0	0	0	0	0	0	0	1	1	0
sequential-read	0	0	0	0	0	0	0	0	1	1	2	3
sequential-readwrite	0	0	0	1	0	2	0	1	2	2	2	3

Table 9.2: Load 2 timeouts caught during failure: 0/3 best case, 3/3 worst case - MooseFS.

Operational timeout errors show that nodes in load 1 around ½ cluster failure start to fail due to the file not being completely read, it is unknown if the write has been complete on the cluster side, yet for the client, the I/O has finished. Load 2 is less impacted until ¾ cluster failure, yet has the same questions around writes data integrity at high failure. This represents **Test 4** (see subsection 4.6.1, p18).

9.1.5 Operational time after master failure

Unfortunately during the test against master failure no results could be recorded. This was due to the cluster being unable to function without the master server. Even with the metalogger server still functioning, the cluster was unable to handle I/O requests, and thus the mount point on the client failed. This represents **Test 5** (see subsection 4.6.1, p18).

9.1.6 Operational time with master under stress

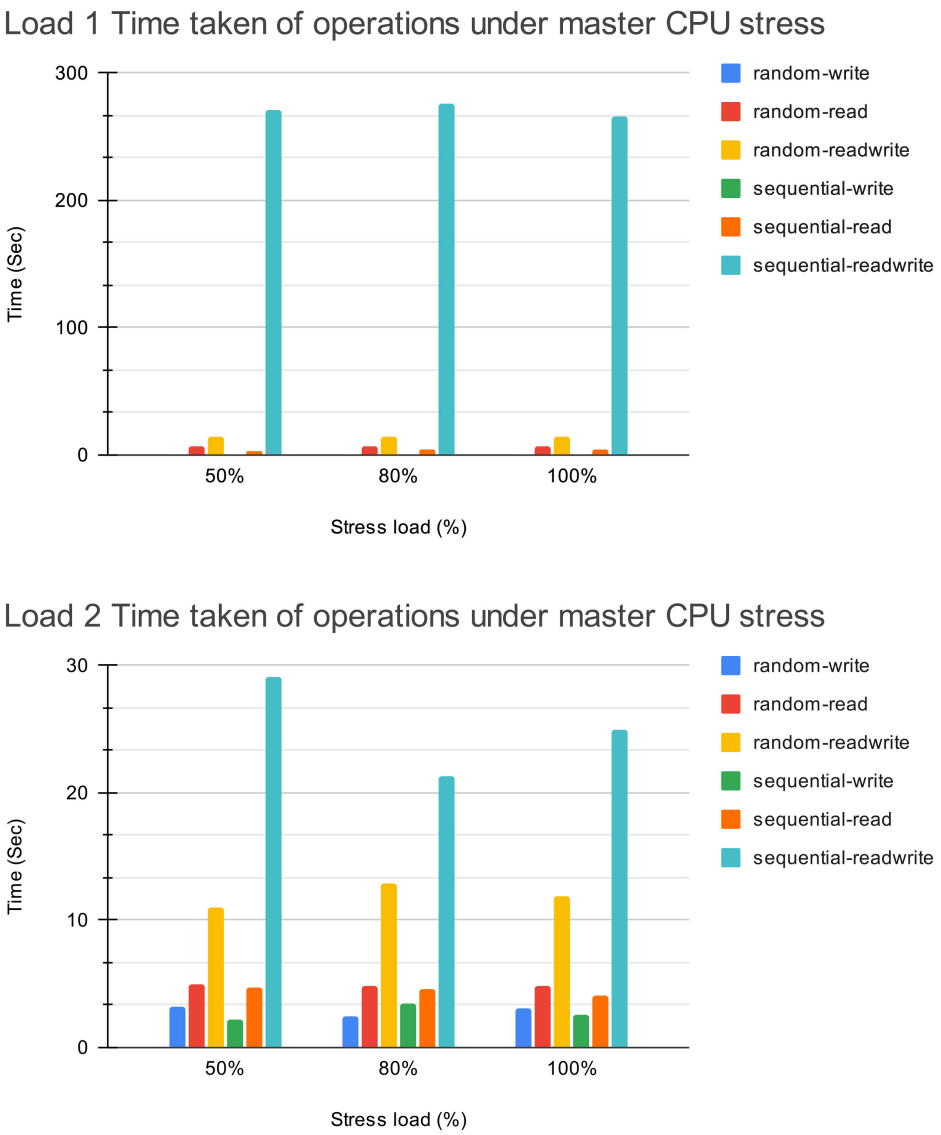


Figure 9.4: Load 1 & 2 average time taken for operations to complete with master server under different levels of CPU stress - MooseFS.

Interestingly, operational time under master stress was not adversely affected when delivering different levels of CPU stress to the master server for loads 1 and 2. There is a slight variance in time for load 2, yet inconclusive enough to show a correlation between increased stress level causing increased operational time. It may be due to MooseFS daemon running at a nice level of -19, whereas stress-ng runs at lowest 0 (-19 overrules 0 in this case). This represents **Test 6** (see subsection 4.6.1, p18).

9.2 Fault tolerance of GlusterFS

9.2.1 Normal operational speeds

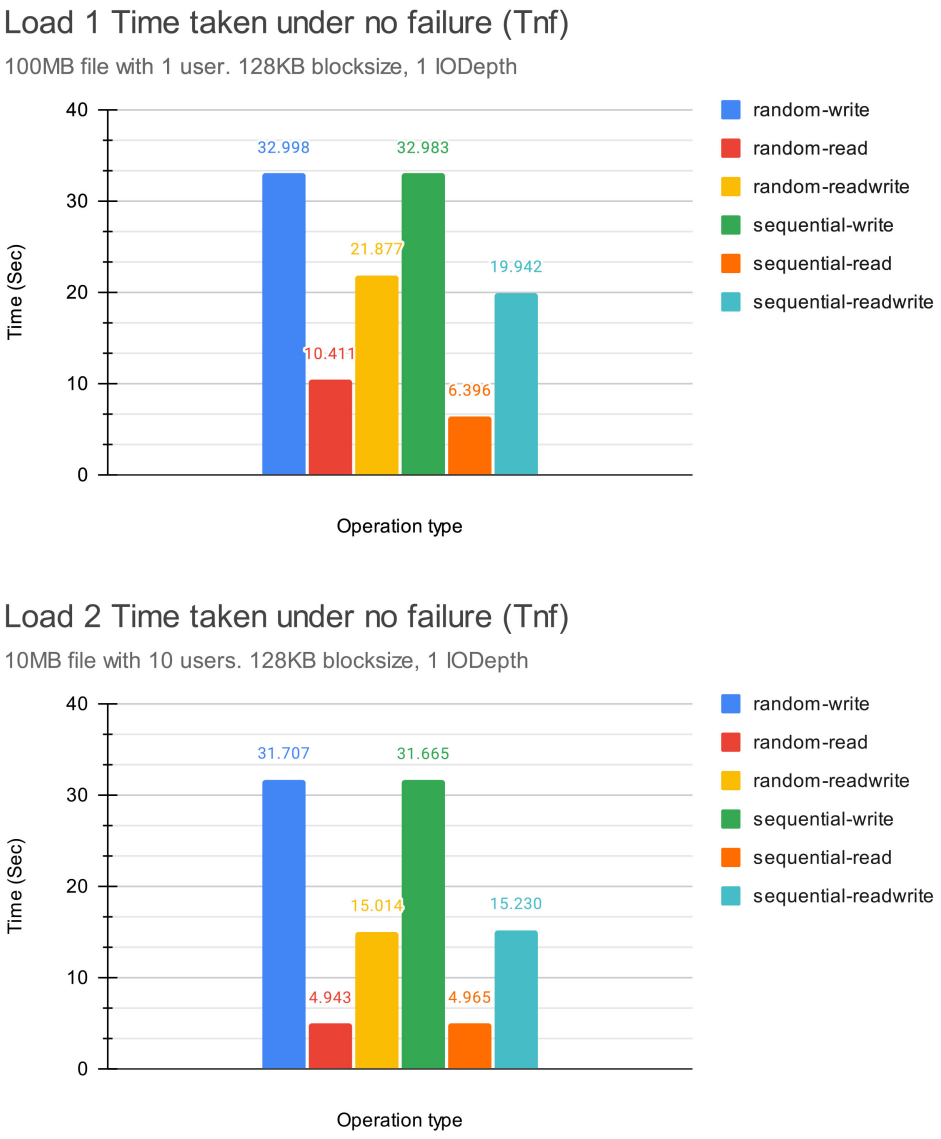


Figure 9.5: Load 1 & 2 average time taken for operations to complete with no failure (Tnf) - GlusterFS.

Across both loads during "normal" operational time, the opposing functions of random and sequential match up at similar speeds. This shows GlusterFS has a moderate and reliable file I/O time. This represents **Test 1** (see subsection 4.6.1, p18).

9.2.2 Operational speeds after failure

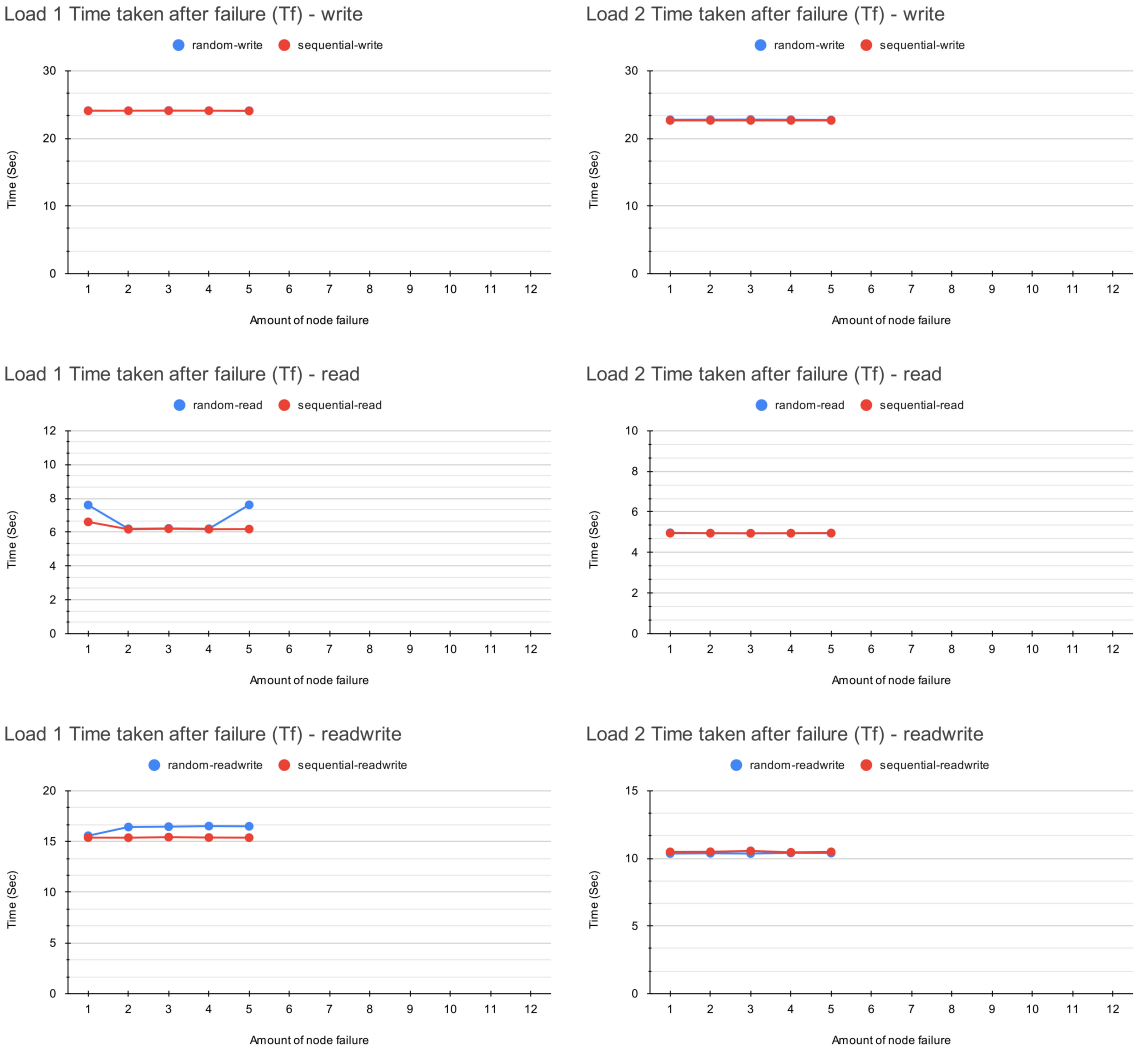


Figure 9.6: Load 1 & 2 average time taken for operations to complete after failure (Tf) - GlusterFS.

Within "time after failure", it is clear the cluster can match a faster speed to that of the "time under normal" operations, which shows that the cluster can readjust to failure and perform better. It performs roughly similar across all tests except load 1 random reads and readwrites. This is expected as random should take longer than sequential. However if operation speed has improved then it is unclear why it stays at similar speeds with increased node failure. Results have stopped after 5 failures (see subsection 9.2.4, p58). This represents **Test 2** (see subsection 4.6.1, p18).

9.2.3 Operational speeds during failure

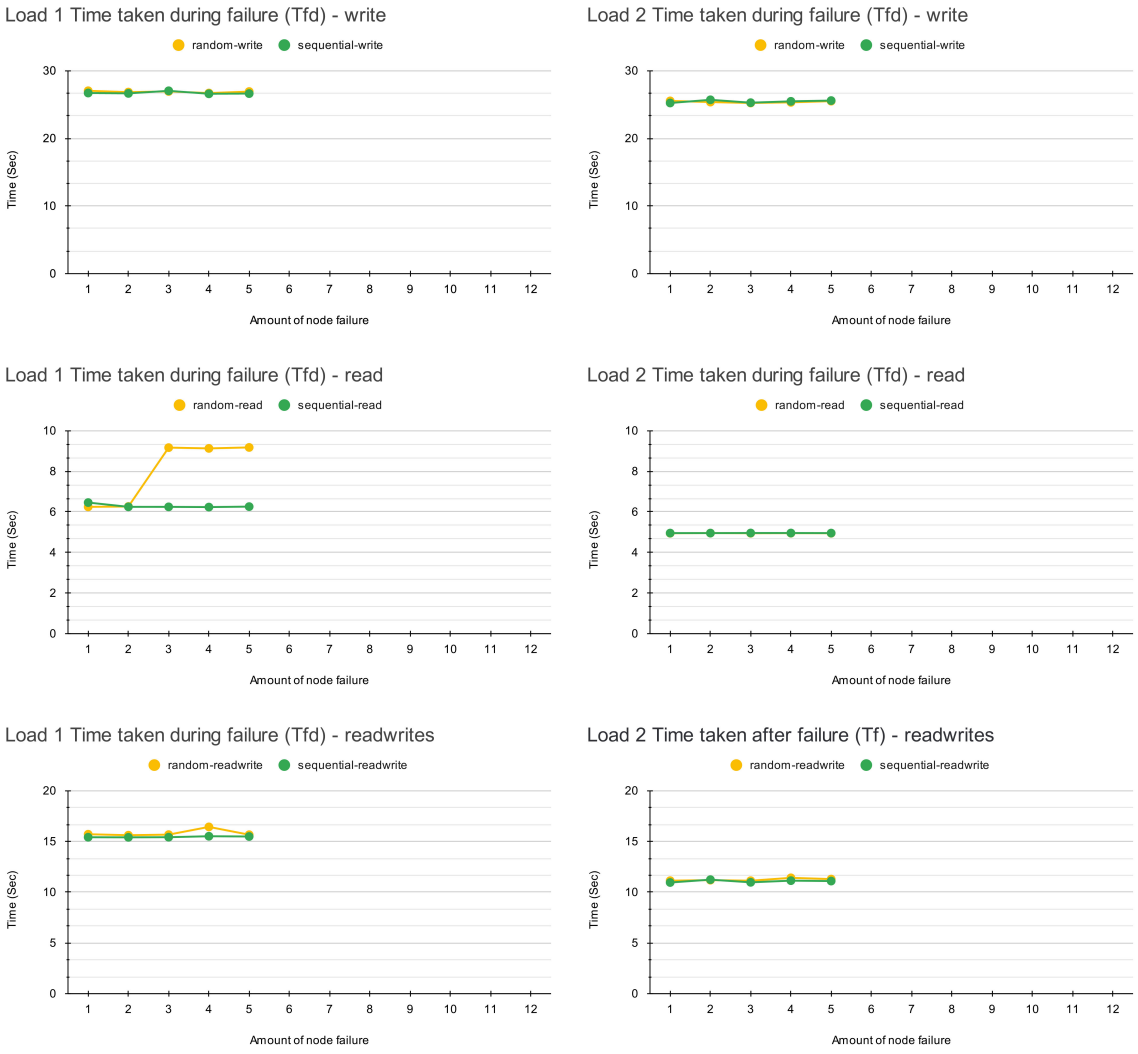


Figure 9.7: Load 1 & 2 average time taken for operations to complete during failure (Tfd) - GlusterFS.

Across all the results in "time during failure" except load 1 random read, perform a little slower yet just as repetitive as "time after failure" (see subsection 9.2.2, p56). Load 1 random read show an upward trend after 2 failures that runs at a similar time with more failure. This is possible due to the operation becoming interrupted during its request, however not obviously clear as to why there is not a decreasing time trend with more failures. This represents **Test 3** (see subsection 4.6.1, p18).

9.2.4 Timeout errors caught during operation testing

Operation type	Node failure											
	1	2	3	4	5	6	7	8	9	10	11	12
random-write	0	0	0	0	0	3	3	3	3	3	3	3
random-read	0	0	0	0	0	3	3	3	3	3	3	3
random-readwrite	0	0	0	0	0	3	3	3	3	3	3	3
sequential-write	0	0	0	0	0	3	3	3	3	3	3	3
sequential-read	0	0	0	0	0	3	3	3	3	3	3	3
sequential-readwrite	0	0	0	0	0	3	3	3	3	3	3	3

Table 9.3: Load 1 timeouts caught during failure: 0/3 best case, 3/3 worst case - GlusterFS.

Operation type	Node failure											
	1	2	3	4	5	6	7	8	9	10	11	12
random-write	0	0	0	0	0	3	3	3	3	3	3	3
random-read	0	0	0	0	0	3	3	3	3	3	3	3
random-readwrite	0	0	0	0	0	3	3	3	3	3	3	3
sequential-write	0	0	0	0	0	3	3	3	3	3	3	3
sequential-read	0	0	0	0	0	3	3	3	3	3	3	3
sequential-readwrite	0	0	0	0	0	3	3	3	3	3	3	3

Table 9.4: Load 2 timeouts caught during failure: 0/3 best case, 3/3 worst case - GlusterFS.

GlusterFS mount point becomes unresponsive after ½ of the cluster failing. This means that it cannot provide a service to the user after this amount of failure (for this particular case of 12 peer nodes). This affects both loads in the same way. This represents **Test 4** (see subsection 4.6.1, p18).

9.2.5 Operational time after selective failure

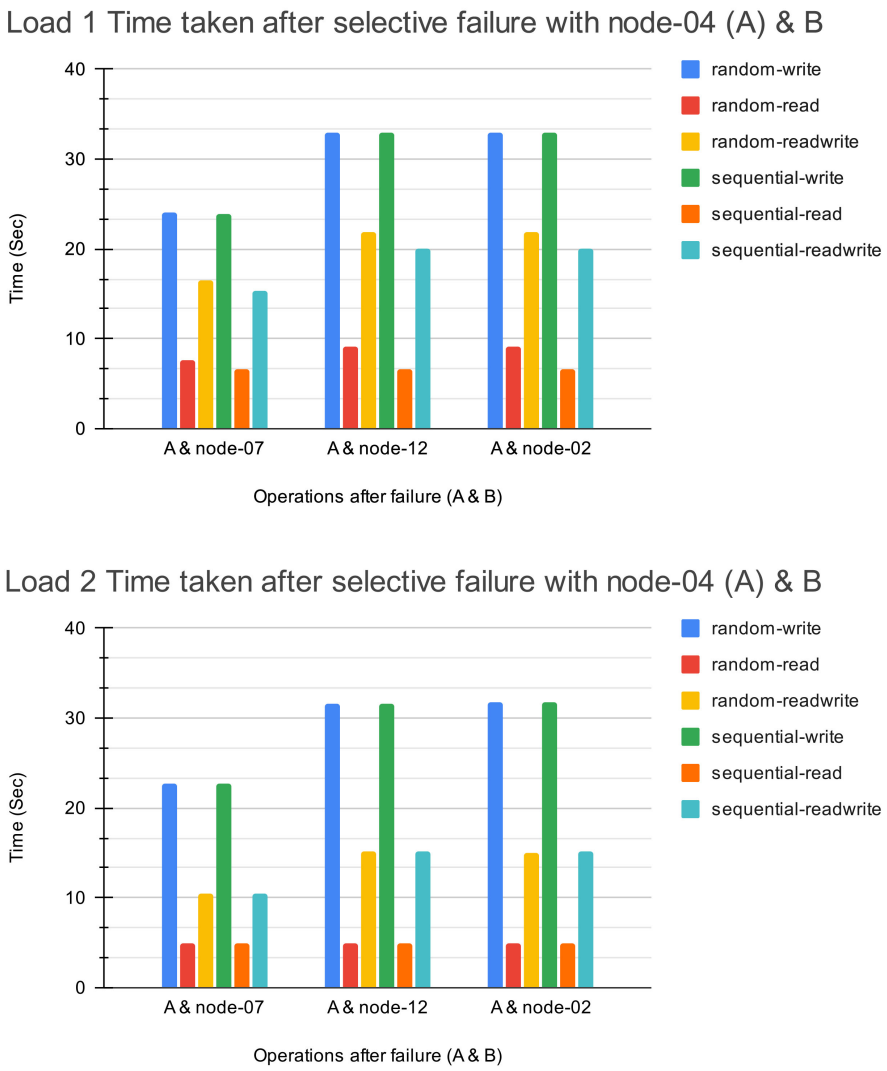


Figure 9.8: Load 1 & 2 average time taken for operations to complete after selective failure of cluster nodes - GlusterFS.

During selective failure across GlusterFS, some possible variation is seen in both loads regarding the server that fails alongside the chosen node. Operations with writes are higher in both sets of results of node-12 and node-02. That could be down to GlusterFS having a laid out pattern of jobs for distribution that gets readjusted if one of those "stepping stones" is unavailable. Interestingly, GlusterFS should be equal across all peer to peer nodes, but the results for both loads are similar in pattern, suggesting a possible hierarchy within its design. This represents **Test 7** (see subsection 4.6.1, p18).

9.3 Hardware measurements

Unfortunately, the data gathered by Telegraf for both systems was difficult to interpret via the live dashboard. It has failed to provide any conclusive data and would not be worth including in the report. This would have represented **Test 8** (see subsection 4.6.1, p18). It could have been improved if there was a dedicated method of measuring specific hardware points and tying them into the duration of each operation test.

9.4 Performance of MooseFS and GlusterFS

Throughput of the cluster has been shown with random, sequential reads and writes. This represents **Test 9** (see subsection 4.6.1, p18).

From the results (see subsection 9.4.1, p61), it is clear that MooseFS I/O latency increases with the number of jobs occurring. The IOPS against the cluster stays reasonable throughout the tests, and in some instances, a trend of IOPS decreases as latency increases with more jobs occurring. However, the sequential write graph has been generated with microseconds for the latency measure, making it challenging to compare. It can be seen that there is an average cap of ∼2.8k top IOPS for reads, yet random is more impacted than that of writes.

The high peak of 35k IOPS for sequential writes is exceptionally high, which shows MooseFS is a high performer when sequentially writing with jobs of 1-4 occurring to the cluster. The 17kμs (17ms) latency for 64 jobs matches close but lower than that of random writes, which is expected with an increased amount of cluster load.

Throughput of GlusterFS (see subsection 9.4.2, p62) shows that random reads do not perform very well when used by 1 job, yet when 8 to 32 jobs are randomly requesting reads on a file, it performs a higher throughput and lower latency ratio than that of 1 user. This may be due to the cluster having to find the requested file multiple times for 1 job. However, when an increased number of jobs request this file, the location is known in advance until the cluster becomes saturated around 32-64 jobs increasing latency.

It can be seen that there is an average cap of ∼2.7k top IOPS for reads and ∼660 top IOPS for writes, regardless of the method of operation. Across results, the reads and writes perform well with an upward curve in the IOPS and latency relationship. This shows that GlusterFS may not be the fastest performer yet is relatively stable.

9.4.1 Throughput of MooseFS

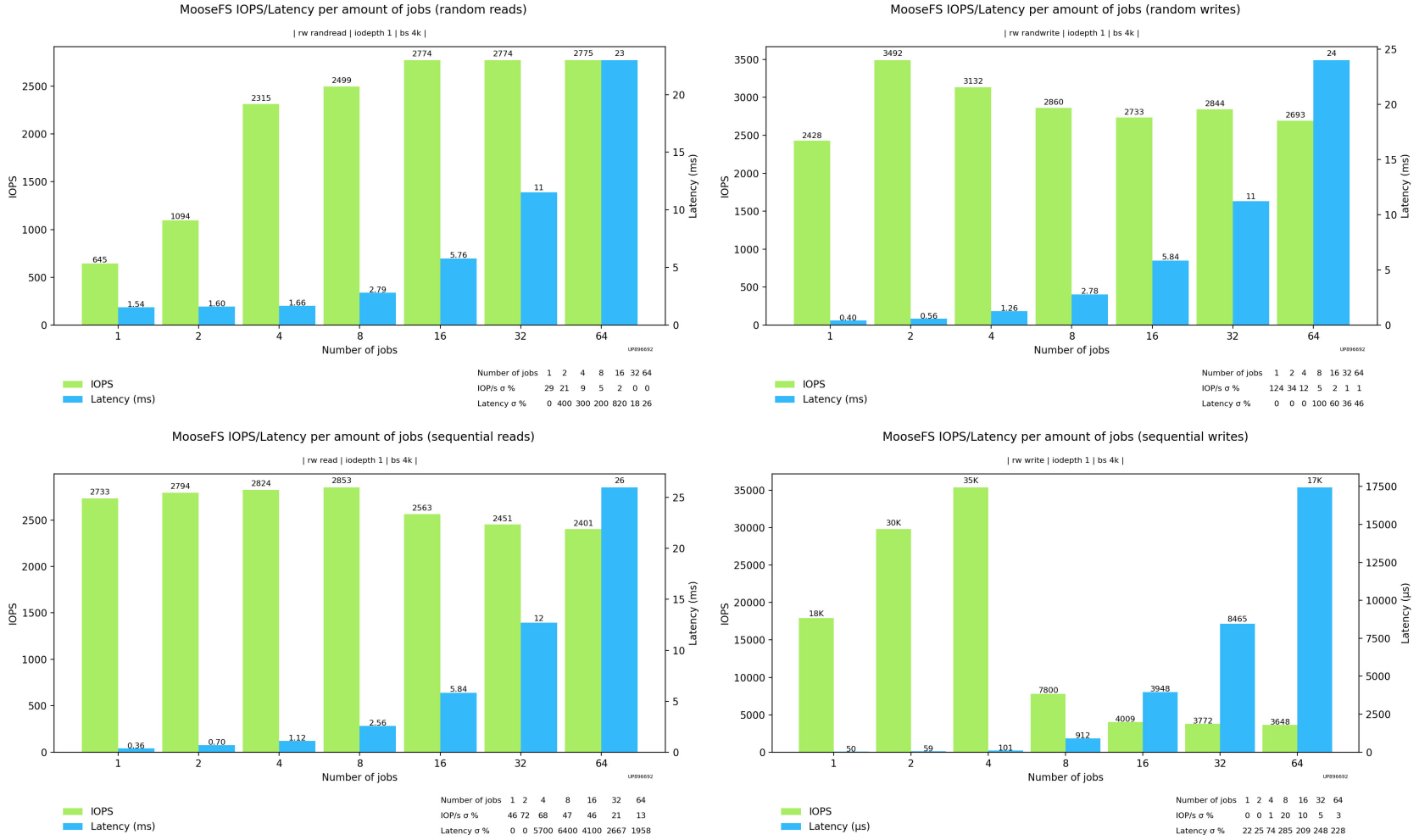


Figure 9.9: Operational IOPS/Latency per amount of jobs (users) reads and writes - MooseFS

9.4.2 Throughput of GlusterFS

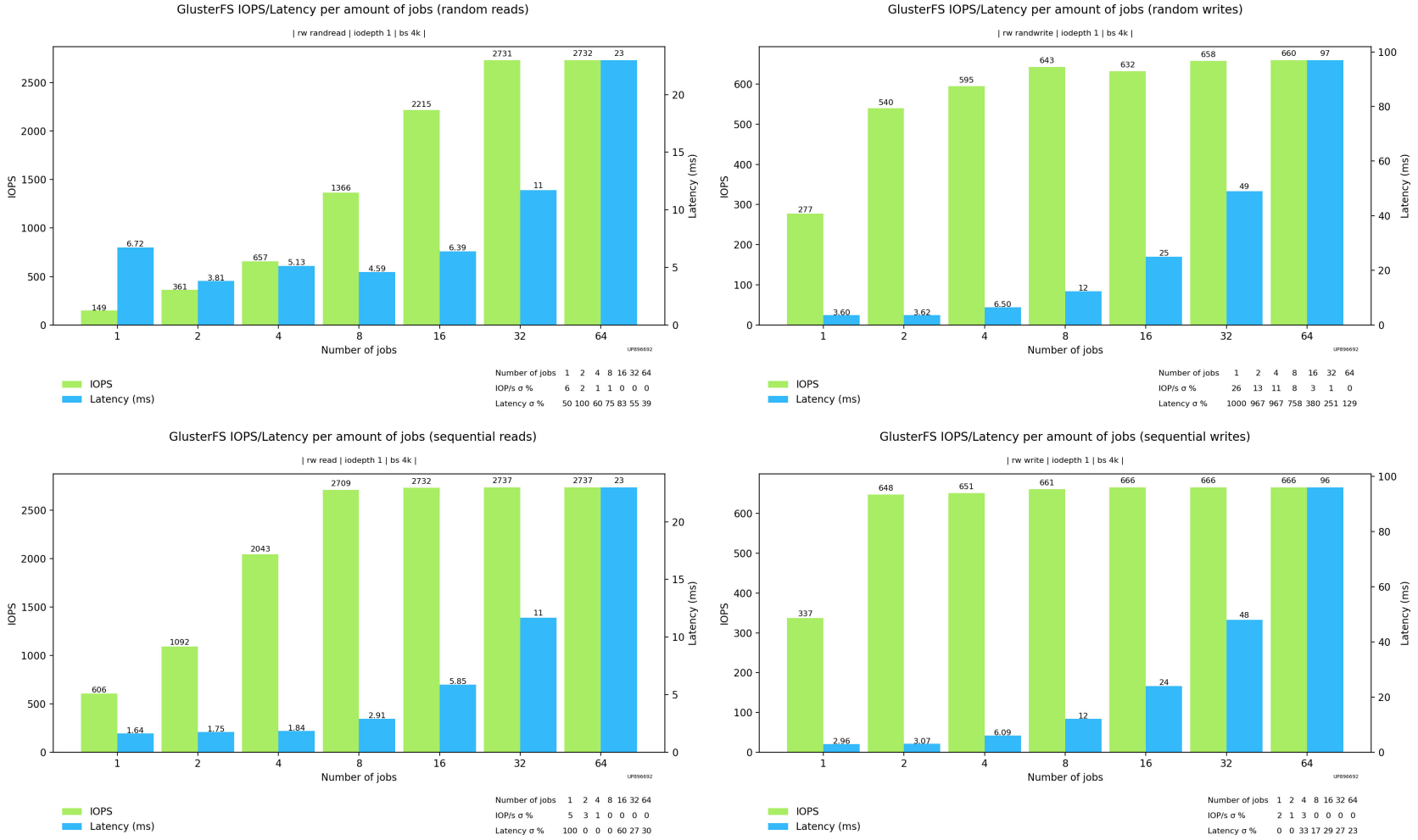


Figure 9.10: Operational IOPS/Latency per amount of jobs (users) reads and writes - GlusterFS

9.5 Requirements and variables reflection

The testbed has produced the needed data to provide an insightful look into the different software and provide some new data in this area of research. Both systems were controlled, data managed, and operations handled well. Each requirement bar one has been fulfilled with some thought into further work. Hardware measurements will needs some further thought (see section 10.2, p66)

9.6 Hypotheses reflection

9.6.1 Master architecture

Master typology should be fault-tolerant enough to withstand the failure of its master server.

MooseFS is one of many master architecture type DFS. Results have shown that MooseFS has not been fault-tolerant enough to withstand the failure of its master server. This claim is valid from this report yet may not represent other DFS.

Time taken of operations during failure will be longer than that of time taken after failure.

The observed results and discussion show that the time taken for operations in both situations remains averagely similar. However, when reaching a high number of node failures, it can be correct when operation timeout is occurring during failure.

Operations will take longer to complete on a system with an overloaded master server.

When loaded with high CPU usage on the master, MooseFS continued operating at normal capacity and operational speed. As discussed, the nice level of the MooseFS daemon was unbeatable by the stress tool, which means there is insufficient evidence to satisfy this claim. Future work may focus on other aspects that cause an overloaded master.

9.6.2 Peer to peer architecture

Peer to peer topology will be less resilient to failure than that of master typology.

The gathered results show that this claim is valid because GlusterFS becomes completely inoperable after 5 failures. It could be argued that if this hypothesis were steered towards data integrity tolerance, then GlusterFS would prevail over MooseFS since the write-back cache mentioned (see section 8.3, p48) creates unknowns about written data assurance.

There will be no effect on the time taken of operations if different servers fail as they all should be equally distributed.

Results show that GlusterFS may be affected if specific servers fail. Nonetheless, the amount of data gathered may not entirely represent the peer to peer architecture. More data should be gathered to verify this for GlusterFS and other peer to peer architectures.

Time taken of operations during failure will be longer than that of time taken after failure.

Again as seen in the results and discussion, the time taken for operations during and after failure remain averagely similar. There is some with an increase in time. With others, they are the same throughout.

9.7 Research questions reflection

The data gathered and interpreted has provided a look into the set questions (see chapter 3, p12). Fault tolerance has been captured using the operational time before, during and after server failure. This has allowed a view into how each cluster may react during each situation. Splitting the operations down one by one to compare it has helped identify if particular software struggles with specific operations and allows for an overall portrait of I/O rather than a generalised view. Performance was touched upon and measured in a controlled way that provided clear graphs for interpretation which could be then repeated against other DFS in the same controlled environment. The testbed has maintained integrity for the collection and repeatability of these results.

Chapter 10

Conclusion

10.1 Project reflection

The research has shown that fault tolerance within the DFS field is challenging to measure with one particular aspect of the system. What has been shown by measuring I/O with FIO against MooseFS and GlusterFS is that they are both fault-tolerant in their own ways, MooseFS providing maximum uptime concerning server failure, GlusterFS providing possibly better data integrity which is another aspect of fault tolerance. It has provided an insight into the different architectural design types of DFS. Performance I/O has been looked into using FIO-bench, providing a repeatable, controlled test that multiple DFS could check against (albeit the test environment is the same). Although the FIO-plot tool that comes with it may need some changes to have results show with the same unit values.

The project management has gone well. However, there is room for improvement due to tasks becoming scattered across multiple environments. Thus the kanban board was not being thoroughly utilised. The use of regular meetings with the supervisor has greatly helped provide weekly goals and feedback along the journey (seen here) (Carter, 2022). It should be noted that the original project time plan did overrun (see Appendix D, p82). The PXP method for implementation worked well for this type of project. Issues were noted, corrected and expressed within the project implementation. Mid project, the research showcase event allowed for scrutineering to occur, which led to an improved research design and result collection. Without this, the report outcome and results would have been very different. The testbed and cluster have provided the

necessary data, and the research questions have been overviewed well.

10.2 Recommendations

If conducting this specific research again, certain aspects such as the fsync close with FIO, dummy system workload (stress) and better hardware measurement per test should be considered. It could provide a different view or understanding of the current project reflection and further understanding of this tricky topic. MooseFS was, in hindsight, a negative due to the FOSS side only having the ability for 1 master server. Picking a software more representative of all master architecture DFS would have represented it better. Python was an excellent choice for this project due to its compatibility with many Unix and Windows systems. Its extensive libraries that can be easily added allow for many aspects to be obtained quickly, and this is how the Discord notification was implemented.

The hardware obtained for the cluster was adequate and allowed for the total completion of result collection. A faster network speed may have sped up the time it took to collect all the necessary data, also providing a different interpretation of results at a higher networking capacity. Virtualbox worked well due to the ability to start and stop nodes remotely. Virtualisation does add another layer to the overall stack; it provides quick deployment and image replication without needing a high level of skill.

Projects similar to this would benefit from setting push goals and having reasonable goals that can be met. This allows for completing the work with enough overhead time to improve upon that work. Without pushing to get enough work done for the research showcase, this project would not have been improved upon to deliver these results and conclusions.

10.3 Future work

It would be interesting to see other DFS software run the same way that MooseFS and GlusterFS were conducted. This will give a larger pool of data to conclude if one architecture design is better than another for some instances. Further aspects of fault tolerance that touch different system areas such as network, hardware, and software fault injection may provide a further overview of the software tolerance to these faults. An indepth look at sequential readwrite may also be considered when working with

MooseFS again to determine what may be the cause for the prolonged operational times.

Another area that could be measured is the impact of these fault-tolerance aspects on the system's performance. By enabling and disabling certain features, what would the system performance then be? Performance could further be analysed when working with larger clusters. This could provide a picture of the scalability to performance ratio.

Finally, with the ever-increasing size of cloud computing and cloud storage, this work may help design a new architectural type when considering fault tolerance and performance.

References

Acronis. (2019). *2.4. understanding data redundancy*. https://dl.acronis.com/u/storage2/html/AcronisStorage_2_installation_guide_en-US/planning-acronis-storage-infrastructure/understanding-data-redundancy.html

Ahmed, W., & Wu, Y. W. (2013). A survey on reliability in distributed systems. *Journal of Computer and System Sciences*, *79*(8), 1243–1255.

Axboe, J. (2022). *Flexible I/O Tester*. https://github.com/axboe/fio

Cao, W., Liu, Z., Wang, P., Chen, S., Zhu, C., Zheng, S., Wang, Y., & Ma, G. (2018). Polarfs: An ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment*, *11*(12), 1849–1862.

Carter, J. (2022). *Clustered network-attached storage: fault tolerance and performance.* https://github.com/jnoc/PJE40-R

Ceph. (2022). *Architecture — ceph documentation*. https://docs.ceph.com/en/latest/architecture/

chrislusf. (2021). *Home · chrislusf/seaweedfs wiki*. https://github.com/chrislusf/seaweedfs/wiki

Dell. (2022). *Specification sheet dell powerscale hybrid family spec sheet dell powerscale hybrid family*. https://www.delltechnologies.com/asset/en-gb/products/storage/technical-support/h16071-ss-powerscale-hybrid-nodes.pdf

Depardon, B., Le Mahec, G., & Séguin, C. (2013). Analysis of six distributed file systems.

Drake, M. (2020). How to create a redundant storage pool using glusterfs on ubuntu 18.04. https://www.digitalocean.com/community/tutorials/how-to-create-a-redundant-storage-pool-using-glusterfs-on-ubuntu-18-04

Dzhurov, Y., Krasteva, I., & Ilieva, S. (2009). Personal extreme programming–an agile process for autonomous developers.

Enterprise, H. P. (2022). *Hpe xp8 storage data sheet (psn1012138134uken.pdf)*. https://www.hpe.com/psnow/doc/PSN1012138134UKEN.pdf

Fujitsu. (2022). *Fujitsu storage eternus ax2200*. https://sp.ts.fujitsu.com/dmsp/Publications/public/ds-eternus-ax2200-ap-en.pdf

Gluster. (2022). *Architecture - gluster docs*. https://docs.gluster.org/en/latest/Quick-Start-Guide/Architecture/

Goldenberg, D. (2016). *Optimize storage efficiency & performance with erasure coding hardware offload — snia*. https://www.snia.org/educational-library/optimize-storage-efficiency-performance-erasure-coding-hardware-offload-2016

Gudu, D., Hardt, M., & Streit, A. (2014). Evaluating the performance and scalability of the ceph distributed storage system. *2014 IEEE International Conference on Big Data (Big Data)*, 177–182.

Huang, S., Liang, S., Fu, S., Shi, W., Tiwari, D., & Chen, H.-b. (2019). Characterizing disk health degradation and proactively protecting against disk failures for reliable storage systems. *2019 IEEE International Conference on Autonomic Computing (ICAC)*, 157–166.

IBM. (2022). *Ibm flashsystem 9500 — data sheet*. https://www.ibm.com/downloads/cas/EMQRM37L

JuiceFS. (2022). *Architecture — juicefs document center*. https://juicefs.com/docs/community/architecture/

Lamb, J. M. (2002). *Clustering for fault tolerance — what is a microsoft cluster? — informit*. https://www.informit.com/articles/article.aspx?p=25748%5C&seqNum=3

Ledmi, A., Bendjenna, H., & Hemam, S. M. (2018). Fault tolerance in distributed systems: A survey. *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)*, 1–5.

Leung, A. W., Pasupathy, S., Goodson, G., & Miller, E. L. (2008). Measurement and analysis of {large-scale} network file system workloads. *2008 USENIX Annual Technical Conference (USENIX ATC 08)*.

Louwrentius. (2022). *fio-plot*. https://github.com/louwrentius/fio-plot

LTS_Tom. (2021). *Linux benchmarking with fio*. https://forums.lawrencesystems.com/t/linux-benchmarking-with-fio/11122

Lustre. (2021). *Documentation — lustre*. https://www.lustre.org/documentation/

MooseFS. (2021). *Moosefs*. https://moosefs.com/support/#documentation

NetApp. (2019). *Fas9000 modular hybrid flash system — netapp*. https://www.netapp.com/pdf.html?item=/media/8939-ds-3810.pdf

Noronha, R., Ouyang, X., & Panda, D. K. (2008). Designing a high-performance clustered nas: A case study with pnfs over rdma on infiniband. *International Conference on High-Performance Computing*, 465–477.

Nutanix. (2021). *Nutanix support & insights*. https://portal.nutanix.com/page/documents/kbs/details?targetId=kA07V000000LX7xSAG

Pfister, G. F. (2001). An introduction to the infiniband architecture. *High performance mass storage and parallel I/O*, *42*(617-632), 10.

Placek, M., & Buyya, R. (2007). *A taxonomy of distributed storage systems*. http://www.cloudbus.org/reports/DistributedStorageTaxonomy.pdf

Saltz, J. S., Dewar, N. I., & Heckman, R. (2018). Key concepts for a data science ethics curriculum. *Proceedings of the 49th ACM technical symposium on computer science education*, 952–957.

Shen, Z., Shu, J., & Lee, P. P. (2016). Reconsidering single failure recovery in clustered file systems. *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 323–334.

Shetty, M. M., & Manjaiah, D. (2017). Challenges of distributed storage systems in internet of things. *Internet of things: Novel advances and envisioned applications* (pp. 193–204). Springer.

Verma, A., & Gayen, T. (2022). Reliability assessment of combined hardware–software non-repairable time-critical systems. *The Computer Journal*.

Wang, F., Xue, X., Liu, B., Yan, F., Zhang, L., Zhang, Q., Xin, X., & Calabretta, N. (2019). Opsquare datacenter networking architecture optimization based on flow-control fast optical switching. *2019 Asia Communications and Photonics Conference (ACP)*, 1–3.

Wieliczko, A. (2015). Moosefs / thread: [moosefs-users] question about mfs reading speed slower than writing speed??? https://web.archive.org/web/20220501165633/https://sourceforge.net/p/moosefs/mailman/moosefs-users/thread/CAKQMey-oodzcTWXH8axWzWLa1DL93N0btyuE8ruzbbO46b1E1g%5C%40mail.gmail.com/#msg33792353

Appendix A

Project Initiation Document

See next page for PID document.

1. Basic details

Student name:	Jonathon Carter
Draft project title:	Open source clustered network attached storage solutions
Course:	BSc (Hons) Computer Science
Project supervisor:	Rinat Khusainov
Client organisation:	N/A
Client contact name:	N/A

2. Degree suitability

Clustered network attached storage (NAS) is an important technology within the information technology sector, through development and testing this can innovate and improve for future use. This technology primarily focuses on reducing single point of failure, large scalable storage, concurrency control and deliverable performance. Within BSc (Hons) Computer Science a student looks into a wide berth of areas where clustered NAS is using principles from.

3. Outline of the project environment and problem to be solved

There are many open source clustered NAS softwares out in the wild, however when it comes to choosing what is the best solution for your specific task it can be overwhelming. Therefore the project will look at investigating open source clustered NAS softwares for users to read upon and decide what is the best solution for them. It is important to do this as groups who can not afford the expensive business solutions will be looking for the best software for themselves to set up. The audience for this research could consist from higher academia to the average engineer, therefore this needs to be concise and to the point with clear results.

The significance of this project is directly correlated to the importance of clustered systems and architectures, without them large scale operations risk large impact if a standalone system goes offline. Clustered NAS can be used in a variety of situations that require extensive solutions for data storage, these can be stretched across a large region to sync and look after this data. The project research questions will focus on answering known issues and concerns that have been expressed within the field, comparing data and analytically producing a hypothesis with the results.

4. Project aim and objectives

The aim of the project is to have a clear report that outlines what clustered NAS solutions have been explored with suggestions for tuning and/or additional features; these explored routes will be presented with comparisons and evaluations of their features. The objectives for this include:

- Outlining what clustered NAS is compared to other technologies.
- Exploring the advantages and disadvantages of clustered NAS.
- Exploring the protocols that can be used to communicate with a clustered NAS.
- Practical demonstration would be to look into comparing the various available softwares that are open source.
    - Investigating performance, reliability, ease of use, integration ect of each software.
- Look into either tuning or adding features to the explored software(s).
- Present this research project concisely and clearly.

5. Project deliverables

This research project will look at delivering the following points:

- Clear overview of what clustered NAS is.
- Outlined advantages and disadvantages of clustered NAS and protocols.
- A testbed developed to gather results and run simulations with.
- Demonstrated clustered NAS setup on one or multiple available softwares.
- Either of the points below:
    - A strong comparison of the selected softwares with their advantages and disadvantages.
    - A clear outlined tuned selected software.
- Suggestions for software improvements.

6. Project constraints

Time is a big constraint for this project, it must not overrun its deadline 6th May 2022 and meet the expected outputs as mentioned. The other factor of time is managing my other modules this academic year which will require well managed and balanced timing to achieve this. Given that this project is research it must be well presented, it needs to be clear and to the point from the start. By not doing this it could skew the purpose of this project away from being a formal investigation into the problem.

7. Project approach

This project will require some background research to start, the practical issues and problems faced by using clustered NAS will need to be explored and looked into. This will require papers, contacting current engineers in the field and other information gathering methods to complete my background research and conclude some research questions. For my artefact I will base the requirements on what research questions I am looking to conduct testing on. The skills that will be required to carry out this project include time management which is to keep track of not overrunning the project and its individual sections. Analytical skills will be used and learnt throughout the process, this will be down to research and practice via open online or physical material.

8. Literature review plan

The literature review will look at multiple papers from a variety of online sources, some will be generic to the clustered NAS area as others will correlate to the research questions. If and when needed other resources will be used from any area or form that they are in. These materials will be gathered, read and referenced within the literature review if relative to the topic, otherwise will be left aside. Some examples of these papers are:

- A Scalable Architecture for Clustered Network Attached Storage
- Designing a High-Performance Clustered NAS: A Case Study with pNFS over RDMA on InfiniBand
- Data ONTAP GX: A Scalable Storage Cluster

9. Facilities and resources

I will require lab computers to run the open source software, as I will be running different softwares on the machines, they will be required to keep the same specifications for each testbed I run. The possibility of running this software and remotely controlling it to run diagnostics on will be helpful however is not a requirement. The constraints on getting lab computers would be the setup time and availability during the university working hours.

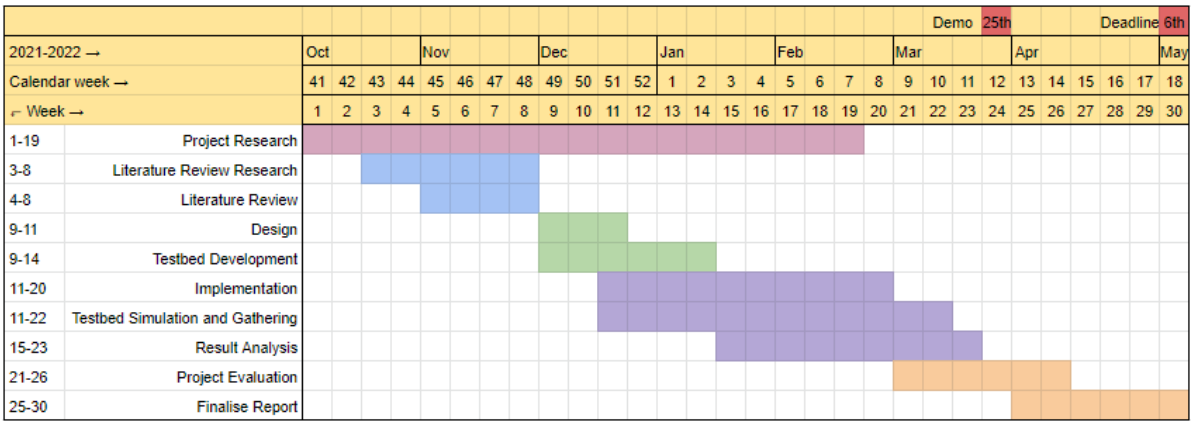
10. Log of risks

Description	Impact	Likelihood	Mitigation	First indicator
COVID-19 outbreak means I cannot get into a lab for usability testing or illness	*Severe*	*Likely*	*Get in while I can, prioritise lab tasks in time.* *Make an alternate test plan that does not need the lab.*	*University informs that lab closure is likely*

Loss of diagnostics that is being collected	Medium	Unlikely	To keep an eye on diagnostics after they have been set off. This means that by actively observing the testing it can be mitigated earlier.	Software is not responding properly.
Loss of data	High	Unlikely	Data and results should have a backup made.	Files that seem to be missing or have corrupted.
Unable to get access to lab computers	Medium	Unlikely	Make sure to book the lab room and turn up for the allotted time.	Uncertainty over the room usage.

11. Project plan

The project has been broken down into these initial stages to get an understanding of how it will look to spend time on each task. The generic terms of ‘design’ and ‘implementation’ are currently there as allocated time for those areas however the individual tasks within them have not been broken down currently as this is an initial gantt chart. This chart will be updated accordingly over the course of the project and the targets to be kept to.



12. Legal, ethical, professional, social issues (mandatory)

This project does not impact ethical, professional or social issues and therefore will not constraint the project's progression. Given the project is using open source software it will need to be vetted to make sure the licence agreements are not being conflicted. Regarding security, the project is focusing on the comparison of the software(s) rather than handling sensitive data, meaning the data I use and generate will not have any impact on this.

Appendix B

Ethics Review

See next page for Ethics Review document.

Certificate of Ethics Review

Project title: Open source clustered network attached storage solutions

Name:	Jonathon Carter	**User ID**:	UP896692	**Application date**:	12/10/2021 14:29:49	**ER Number**:	TETHIC-2021-101294

You must download your referral certificate, print a copy and keep it as a record of this review.

The FEC representative(s) for the **School of Computing** is/are **Philip Scott, Matthew Dennis**

It is your responsibility to follow the University Code of Practice on Ethical Standards and any Department/School or professional guidelines in the conduct of your study including relevant guidelines regarding health and safety of researchers including the following:

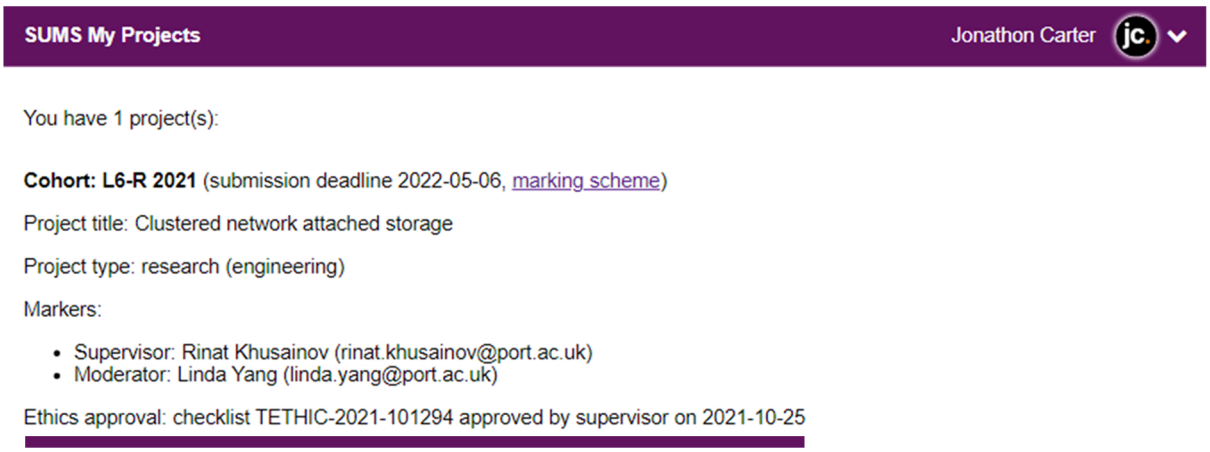
- University Policy
- Safety on Geological Fieldwork

It is also your responsibility to follow University guidance on Data Protection Policy:

- General guidance for all data protection issues
- University Data Protection Policy

Which school/department do you belong to?: **School of Computing**
What is your primary role at the University?: **Undergraduate Student**
What is the name of the member of staff who is responsible for supervising your project?: **Rinat Khusainov**
Is the study likely to involve human subjects (observation) or participants?: No
Will financial inducements (other than reasonable expenses and compensation for time) be offered to participants?: No
Are there risks of significant damage to physical and/or ecological environmental features?: No
Are there risks of significant damage to features of historical or cultural heritage (e.g. impacts of study techniques, taking of samples)?: No
Does the project involve animals in any way?: No
Could the research outputs potentially be harmful to third parties?: No
Could your research/artefact be adapted and be misused?: No
Does your project or project deliverable have any security implications?: No

I confirm that I have considered the implications for data collection and use, taking into consideration legal requirements (UK GDPR, Data Protection Act 2018 etc)
I confirm that I have considered the impact of this work and and taken any reasonable action to mitigate potential misuse of the project outputs
I confirm that I will act ethically and honestly throughout this project



Appendix C

Chosen Distributed File Systems

C.1 MooseFS

(Depardon et al., 2013; MooseFS, 2021)

C.1.1 Architecture

MooseFS is a centralised master configuration DFS. It has a master that manages the cluster's metadata and attached chunk servers that store and replicate data blocks. This architecture comes with a metalogger server which captures the metadata from the master to be used as the fall over in case of a master failure.

C.1.2 Communication protocols

Clients connecting to the MooseFS cluster use a FUSE mounting client mfsmount. It is directly connected via the file system of POSIX compliant systems. Third-party alternatives are available to connect from OSX and Windows yet are not directly offered by MooseFS. This mfsmount communicates directly with the master server, which allows the client to access the rest of the cluster.

C.1.3 Data redundancy

MooseFS offers file replication where copies are created to be preserved across multiple chunk servers. A client writes to the mount point where the master will send it data about the location of the chunk servers. Then the client writes this to the first chunk

server with the received location data. From here, this file is then replicated across the cluster. MooseFS Pro (paid) offers erasure coding as another way of data redundancy. Rather than replicating the file, which takes up valuable storage space, it gets split into "erasure codes" of 8+n chunks. This uses less space while keeping the data redundant to failure. It is distributed across the cluster in the same way as replication.

C.1.4 Data caching

MooseFS master automatically handles data caching. Automatically it is enabled. However, an administrator enables or disables this. There is no caching done from the client-side.

C.1.5 Loads, scalability and faults

If a server in the cluster becomes unavailable, the system will store replicas on other chunk servers. On the other hand, if the cluster has too many replicas of a file of the specified MooseFS replica goal, it will remove the extra copies. Data is managed with a version number so that when connecting a data server with an older copy, it will not cause the files to become muddled. This allows for effortless scalability with new chunk servers. Client failures do not have any influence on the cluster. If a chunk server were to fail, it would be logged as offline, and no data would be processed or sent to it.

C.1.6 Advantages

- Large amount of storage supported
- Can be built on commodity hardware
- High availability with no single point of failure
- Data is divided into chunks and redundantly spread across storage servers
- High scalability available
- Many clients can access many files concurrently

C.1.7 Disadvantages

- Does have a locked erasure coding data redundancy in a paid version
- Only one master server available in the unpaid FOSS version

- Lots of configuration requirements
- No direct windows compliant mounting protocol

C.2 GlusterFS

(Depardon et al., 2013; Gluster, 2022)

C.2.1 Architecture

GlusterFS is a peer-to-peer based configuration DFS. Each server manages its storage and metadata about the cluster. When a group of GlusterFS servers are connected together, this is called a trusted pooled storage on here volumes of bricks (storage) can be set up across the multiple servers.

C.2.2 Communication protocols

GlusterFS offers FUSE as a way of mounting via the Gluster native client with the glusterfs command. This is the recommended method for high concurrency and high write performance. NFS v3 and CIFS are also offered for access to the GlusterFS volumes, which allows for a larger audience of devices. Although, it may not provide the best results for performance.

C.2.3 Data redundancy

GlusterFS provides multiple different options of volume types, a volume being the logical collection of bricks. The cluster can be set up without any redundancy where the client writes to the cluster, and there are no replicas of that file made. Distributed replication, where copies of the files are made across the cluster, may take up an increasing amount of storage space depending on the number of copies required. Distributed dispersion is the erasure coding of data across the GlusterFS cluster. This uses less space than that of distributed replication.

C.2.4 Data caching

GlusterFS has an automatic caching setup on the cluster side. Although not completely clear how much, it does have tuning options available to increase or decrease the limit

of files stored and the duration of the storage time. GlusterFS, by default, does not have client-side caching.

C.2.5 Loads, scalability and faults

When a server within the cluster fails, failover is automatically already set up, meaning that the most up to date data will start being served by another server within the cluster. The volume size will be re-updated and balanced when the administrator is ready to put the fixed server back in the trusted storage pool. The scalability of GlusterFS is straightforward due to the availability of using "mix and match" hardware which can be added straight into a trusted storage pool. As long as the volume UUID does not match another in the cluster, it can be added straight away and balanced into the volume size.

C.2.6 Advantages

- Low hardware specifications
- High availability with no single point of failure
- Multiple options of data redundancy available
- Limited configuration needed
- Large amount of storage supported
- Many clients can access many files concurrently

C.2.7 Disadvantages

- Performance limited outside of FUSE
- No centralised management server
- Not so performance focused

Appendix D

Project Plan and Progress

See below figures for the original project plan and the progress over the course of this project.

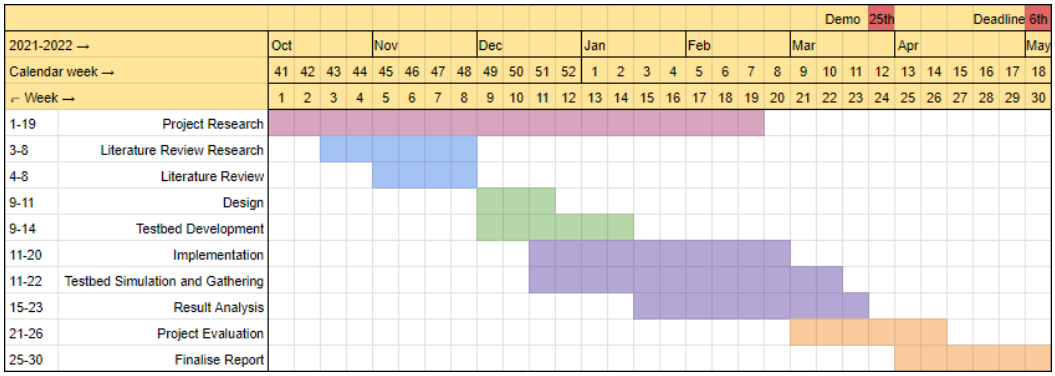


Figure D.1: Original plan as seen from PID.

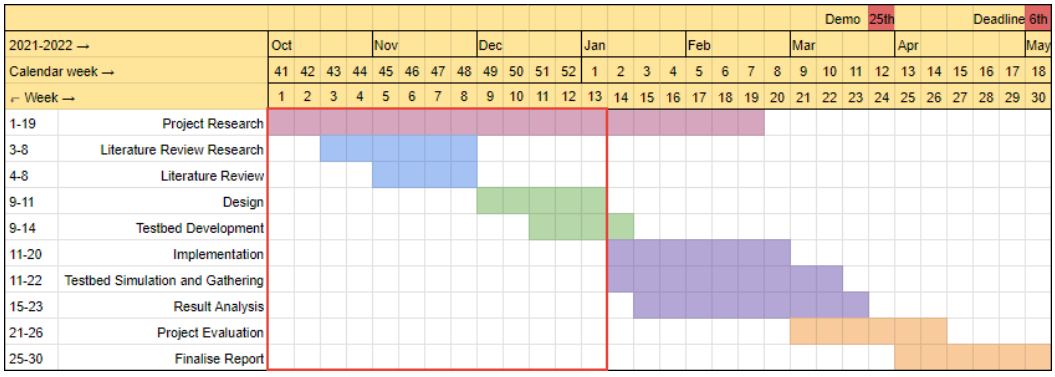


Figure D.2: Project progress near to half way through.

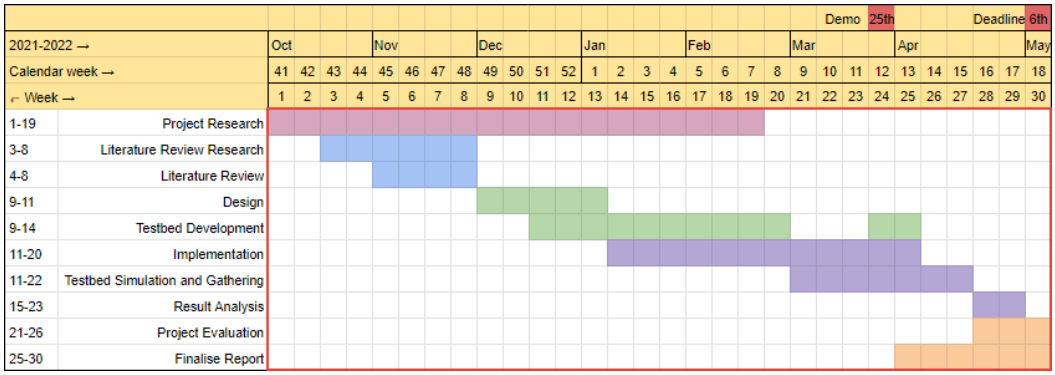


Figure D.3: Project progress as of finish.

Appendix E

Cluster Design

See next page for Cluster design image.

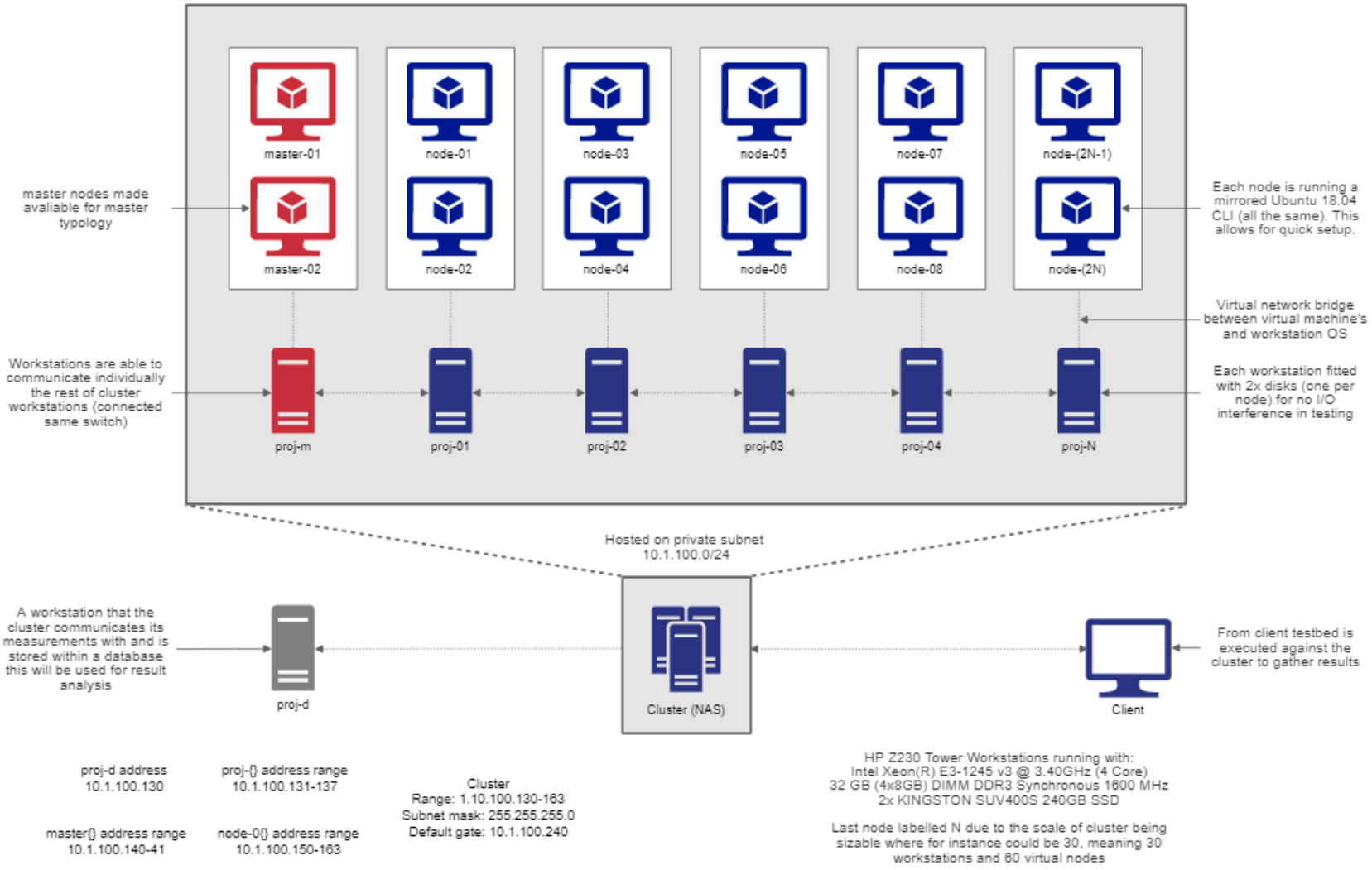


Figure E.1: Cluster design for the project