ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

**NEUMANN JÁNOS**
**FACULTY OF INFORMATICS**

# THESIS

**OE-NIK** Student's name: **Kali, István**
**2022** Student's registration number: **T/006072/FI12904/N**

Óbudai University
John von Neumann Faculty of Informatics
Institute for Cyber-physical Systems

# THESIS WORK SPECIFICATION

| | |
|---|---|
| Name of the student: | **István Kali** |
| Identification number of the student: | T/006072/FI12904/N |
| Neptun's code: | [redacted] |

Title of the thesis:

**Analyzing machine learning based on distributed file system**
**Gépi tanulás vizsgálata elosztott fájlrendszer alapon**

| | |
|---|---|
| Internal supervisor | Attila Farkas |
| External supervisor: | Róbert Lovas Ph.D. habil. |
| Deadline: | 15 May 2022 |
| Subjects of the final exam: | Computer Architectures<br>Cloud service technologies and IT security specialization |

**The task:**

The rapid development in the field of machine learning created the need for the optimization of all parts in the training process. One of the tasks which will inevitably have to be streamlined, is the process of providing the input data needed for training. Using a distributed file system of multiple nodes instead of one will reduce the I/O overhead and distribute the work in a more efficient way. The task of the student is to select, deploy, configure and test a distributed file system on a cloud computing platform with containerization, using system automation tools – with the purpose of connecting to and also serving a deep learning framework efficiently.

**The thesis must contain**:

- general description of deep learning frameworks,
- analysis of potential distributed file systems based on literature,
- detailed system plan about the distributed file system platform,
- implementation of the system using automation tools,
- testing with regards to performance and efficiency of the training process,
- potential areas for further development.

Place of stamp

ÓBUDAI EGYETEM * Neumann János Informatikai Kar * 1. Budapest

.....
Dr. Rita Fleiner
head of institute

This specification is valid until: **15 May 2024**
(According to OE TVSz 55.§)

I consider the thesis suitable for submission:

.....
external supervisor

.....
internal supervisor

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Faculty of Informatics

# STUDENT'S DECLARATION

I the undersigned student hereby declare that this degree project / thesis is the result of my own work; I have disclosed the references and tools used in an identifiable manner. The results indicated in my degree project / thesis completed may be used by the university and the institution announcing the task for their own purposes free of charge.

Dated: Budapest, 2022.05.15.

.....

student's signature

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Faculty of Informatics

# CONSULTATION LOG

| Student's name: | Neptun code: | Specialty: |
|---|---|---|
| István Kali | [redacted] | Cloud service technologies and IT security |

Phone number: [redacted]

Mailing address (e.g. domicile): [redacted]

Degree project / thesis[1] title in Hungarian:

Gépi tanulás vizsgálata elosztott fájlrendszer alapon

Degree project / thesis[2] title in English:

Analyzing machine learning based on distributed file system

| Institutional supervisor: | External supervisor: |
|---|---|
| Attila Farkas | Róbert Lovas Ph.D. habil. |

Please write the data in printed capital letters!

| Occ. | Date | Content | Signature |
|---|---|---|---|
| 1. | 2021.09.13. | Cooperation arrangement, discussion regarding the field of the thesis. | Farkas Attila |
| 2. | 2021.10.07. | Consultation about potential software tools and thesis work structuring. | Farkas Attila |
| 3. | 2021.11.21. | Consultation about system planning, progress and content review | Farkas Attila |
| 4. | 2021.12.08 | Thesis work and presentation review | Farkas Attila |

The Consultation log is required to be countersigned by either supervisor on a total of 4 occasions of consultation.

The student has complied with the requirements of the subject titled Degree project I / Degree project II (BSc) or Thesis 1 / Thesis 2 / Thesis 3 / Thesis 4[3], and is allowed to proceed for reporting / defense[4].

Dated: Budapest, 2021.12.08

Farkas Attila
.....
institutional supervisor

[1] Underline as appropriate
[2] Underline as appropriate
[3] Underline as appropriate
[4] Underline as appropriate

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Faculty of Informatics

# CONSULTATION LOG

| Student's name: | Neptun code: | Specialty: |
|---|---|---|
| István Kali | [redacted] | Cloud service technologies and IT security |

Phone number: Mailing address (e.g. domicile):

[redacted]

Degree project / thesis[1] title in Hungarian:

Gépi tanulás vizsgálata elosztott fájlrendszer alapon

Degree project / thesis[2] title in English:

Analyzing machine learning based on distributed file system

| Institutional supervisor: | External supervisor: |
|---|---|
| Attila Farkas | Róbert Lovas Ph.D. habil. |

Please write the data in printed capital letters!

| Occ. | Date | Content | Signature |
|---|---|---|---|
| 1. | 2022.02.03. | Definition of tasks and objectives. | Farkas Attila |
| 2. | 2022.03.09. | Review of additions, requesting of computational resources for the implementation. | Farkas Attila |
| 3. | 2022.04.29. | Review of implementation, discussion of documentation | Farkas Attila |
| 4. | 2022.05.12. | Finished thesis and presentation review | Farkas Attila |

The Consultation log is required to be countersigned by either supervisor on a total of 4 occasions of consultation.

The student has complied with the requirements of the subject titled Degree project I / Degree project II (BSc) or Thesis 1 / Thesis 2 / Thesis 3 / Thesis 4[3], and is allowed to proceed for reporting / defense[4].

Dated: Budapest, 2021.05.12

Farkas Attila
.....
institutional supervisor

[1] Underline as appropriate
[2] Underline as appropriate
[3] Underline as appropriate
[4] Underline as appropriate

# Table of Contents

# 1 Introduction

Machine learning has been one of the fastest growing areas in computer science in the last fifteen years and it continues to produce innovation with the training process becoming ever more streamlined. With the rapid development of deep learning tools the demand for training data will only get higher and higher. With central file systems the emergence of a storage bottleneck is inevitable. Transitioning to a distributed file system, from a centralized storage solution, will mean improvements in performance by reducing the I/O overhead, and distributing the work more evenly, while also providing a better return on investment the more our system is scaled up.

The goal of this thesis is the creation and implementation of a distributed file system in cloud environment which can effectively serve as the central storage solution for the distributed deep learning framework Horovod[1]. In addition all configuration steps will be executed using automation tools, which will make both initial and subsequent implementations more streamlined.

After a general description of deep learning frameworks, multiple distributed file systems will be compared against each other, with the aim of finding the most suitable based on it's compatibility, reliability and performance. A short description of Terraform and Ansible will also be included since I am are going to use these tools for building my system.

There are no existing, well known implementations of Horovod based on a distributed file system. However, Spark, a distributed big-data solution which is based on distributed datasets, that store data on multiple machines, similarly to how Horovod distributes worker nodes to multiple physical machines, has been implemented based on HDFS. A good example for this could be the web-based platform, Databricks which was developed by the creators of Spark, and provides automated cluster management and IPython-style notebooks.

# 2 Description of deep learning frameworks

This thesis is created with the goal of improving the performance of a deep learning infrastructure by providing training data faster and more reliably than a single machine could. Since the completed project is intended to be deployed in a deep learning environment, knowledge about machine and deep learning is integral to understanding the whole thesis. Machine learning is a branch of artificial intelligence that is concerned with the emulation of human intelligence, more precisely the ability of learning from the surrounding environment. The goal is the creation of a system that is capable of executing specific tasks, based on statistical data without being explicitly programmed. At the heart of these systems are the machine learning algorithms, which are mathematical algorithms that recognize patterns in data and make predictions, improving their precision automatically through experience based on interaction with input data.

Deep learning is a subcategory of machine learning. While in machine learning the neural network has only a single hidden layer, deep learning algorithms have multiple hidden layers. A simplified graph of an artificial neural network can be seen on Figure 2-1. The two endpoints of a neural network are the input and output layers, the layers in between are the hidden layers each of which consist of an array of non-linear information processing units. The higher the number of layers or the number of units in a single layer, the more complex functions can be represented with the neural network. However, these complex deep learning frameworks require multitudes more training data sets than simpler machine learning ones do [2].

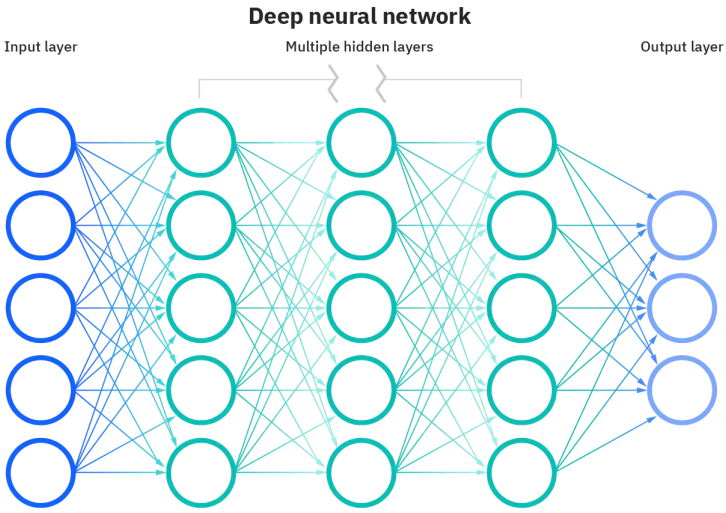



Figure 2-1: Artificial neural network [3]

The distributed file system that will be set up during the implementation of this thesis will work with a distributed deep learning system that uses TensorFlow and Horovod. Tensorflow is a machine and deep learning platform developed by Google. It offers libraries, tools and resources for creating machine learning based applications. Tensorflow has seen vast commercial use from Ebay to SAP and also significant academic use from universities such as Stanford University[4].

TensorFlow has neural networks similar to the one seen on Figure 2-1, however in this platform computational nodes can also be connected in a multidimensional way, which will result in much higher complexity. The resulting structures are called tensors, which is the origin for the platform's name as well.

However, TensorFlow by itself is not enough for creating a distributed deep learning system, for that we also need a specialized framework like Horovod. Horovod was developed by Uber as a distributed deep learning framework that supports TensorFlow, Keras, PyTorch and Apache MXN. As a company with an unusually large amount of data for training, engineers at Uber have encountered a GPU bottleneck due to the physical expandability limitations of their servers.

For this reason they created Horovod which makes the usage of multiple servers for deep learning possible. One of the key points when creating Horovod was the effort required for converting a single GPU program to a distributed one, besides the obvious goal of making the system faster and more easily extendable. Uber definitely achieved its goal with Horovod scaling very efficiently to very high scales as seen in Figure 2-2.

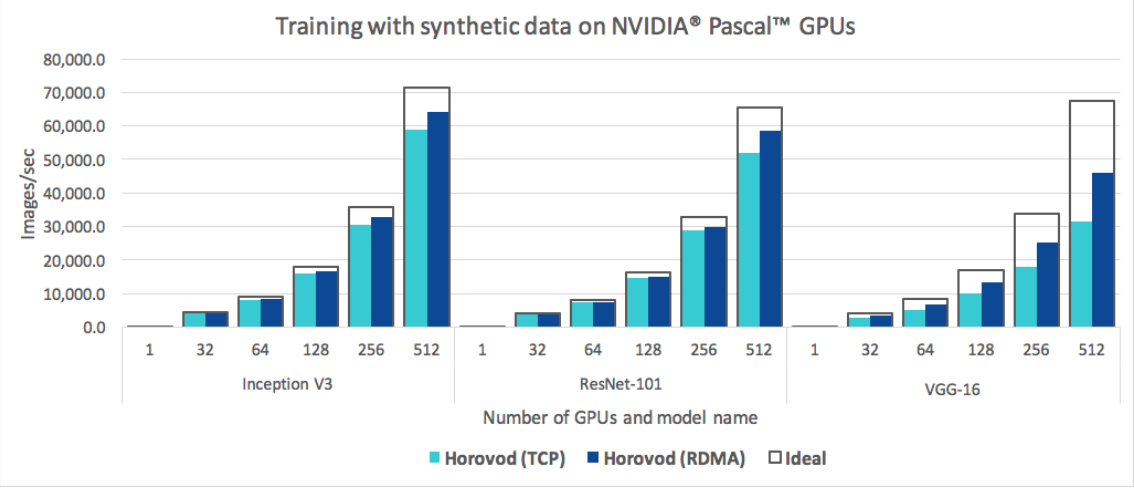



Figure 2-2: Scaling efficiency [5]

# 3 Comparison and analysis of potential software tools

The goal of this thesis is to optimize the I/O overhead, general speed and volume of data that can be supplied for a distributed deep learning framework. This will be achieved through building a distributed file system that is capable of outperforming centralized storage solutions.
To ensure that future builds of this file system could be reliably created the whole process will be automated. The prevention of a bottleneck in the flow of training data is not only dependant on the execution of these tasks but also the software chosen. Since the Horovod cluster to which my system will provide the storage aspect for was created using Terraform and Ansible[6] I will be using the same software in my implementation for the sake of coherency. For these reasons, in this section a short analysis of Ansible and Terraform will be given, as well as the comparison of multiple distributed file systems.

## 3.1 Analysis of automation tools

Automation tools are a central application type in the emerging field of DevOps. System automation can help system administrators by using code as a directive rather than human input. These tools can keep servers and other systems in the desired state and execute operations by following automation logic code [7]. This code can be changed on the administrators workstation and then be automatically updated on machines configured using that code. For this project these tools are needed to set up virtual machines in a cloud environment as well as configure my distributed file system on these virtual machines.

### 3.1.1 Terraform

The first automation tool I will take a look at is Terraform. This software was created by HashiCorp and released in 2014 as an open-source tool that is used for the automation of virtual infrastructure. Terraform builds, changes and destroys infrastructure by managing external resources, such as cloud infrastructures or network appliances.
The management of these resources are all done through code. More precisely with HashiCorp Configuration Language (HCL) in special files with the .tf extension. In these files are multiple blocks all responsible for different parts of the configuration to be executed as seen on Figure 3-1. Lastly the Terraform files can be run using the Terraform CLI.

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "3.68.0"
    }
  }
}

provider "aws" {
    profile = "default"
    region = "eu-central-1"
}

resource "aws_instance" "virtual_machine" {
    ami = "ami-830c94e3"
    instance_type = "t2.micro"
}
```

Figure 3-1: Basic AWS instance creation

Terraform has a registry somewhat similar to Docker Hub in the sense that it provides the user with exact commands for reaching the desired resource from a provider. Unlike Docker Hub however, in the registry are also examples for calling the given resources as well as a documentation of all arguments that can be used with it. This makes it more similar to a programming framework's documentation, with an overall directory of hundreds of services.
In this thesis, Terraform will be used for setting up a virtual network and an array of virtual machines acting as hosts in the distributed file system's storage cluster.

### 3.1.2 Ansible

Ansible is another open-source automation software, however unlike Terraform which is mostly used in the creation of virtual infrastructure, Ansible is typically used for the configuration of already existing virtual devices. These virtual devices must have connectivity with the machine from which the automation is deployed. This is usually achieved through SSH.
Similarly to Terraform's .tf files Ansible uses YAML files called playbooks. These playbooks can contain multiple plays which are a set of tasks to be run on the target hosts, and a task is a single command in a form that is usable for the target CLI.

```
---
- name: Configure DHCP server on R1
  hosts: R1

  tasks:
    - name: enable dhcp
      ios_config:
        lines:
          - ip dhcp excluded-address 192.168.10.1 192.168.10.3
          - ip dhcp excluded-address 192.168.10.86 192.168.10.255
          - ip dhcp pool R1G1
          - network 192.168.10.0 255.255.255.0
          - default-router 192.168.10.1
          - dns-server 192.168.10.1
          - domain-name ccna-labs.com
```

Figure 3-2: DHCP server configuration

At the very beginning of the playbook the name for the play is given as well as the name of the host or host group. Hosts and their corresponding IP addresses are defined in separate files, so that the naming scheme can be consistent throughout the playbooks. Under the tasks part of the playbook another name can be given for the specific task. Finally, after entering configuration mode on the router our commands can be entered as seen on Figure 3-2.

Ansible is not only useful for configuration but is also a convenient tool for monitoring devices. We can even schedule the execution of our playbooks, which can be used for making regular health checks. Using Ansible I will configure the previously created hosts which make up the infrastructure of the chosen distributed file system.

## 3.2 Comparison of distributed file systems

Distributed file systems store data in an unstructured way, by organizing data into hierarchical namespaces. To the user it might seem like and ordinary storage solution, however most if not all of the files are stored remotely on servers [8]. There are many file systems specialized for numerous tasks and environments. The ones compared in this thesis will have to meet the criteria of being compatible with Horovod's mode of access, able to provide low latency when working with small files and reasonably good scalability for usage in supplying more complex deep learning algorithms.

### 3.2.1 Ceph

Ceph was developed at University of California by Sage Weil in 2003. It has become one of the most used and recognized distributed file system over the last decade. It is an open source, very reliable and scalable storage solution. Ceph can be scaled to the exabyte-scale without significant compromise when it comes to reliability [9].

Ceph answers one of the big problems with cloud infrastructures, which is the need for storage that can be cheaply scaled up and out and is also easy to integrate with other components. Thanks to its scalability Ceph is the file system of choice for many cloud architectures such as CloudStack[10], OpenStack[11] and OpenNebula[12]. In OpenStack for example Ceph can substitute both Swift and Cinder fulfilling their roles as both object and block storage solution [13].

In Ceph the distributed file system, the underlying operating system and the physical hardware make up a node. Combining these nodes makes up a Ceph cluster as seen on Figure 3-3. This cluster can be scaled without hard upper limits. Data is distributed between these nodes and retrieved by Ceph's algorithm called CRUSH, which helps evade single points of failure [14].

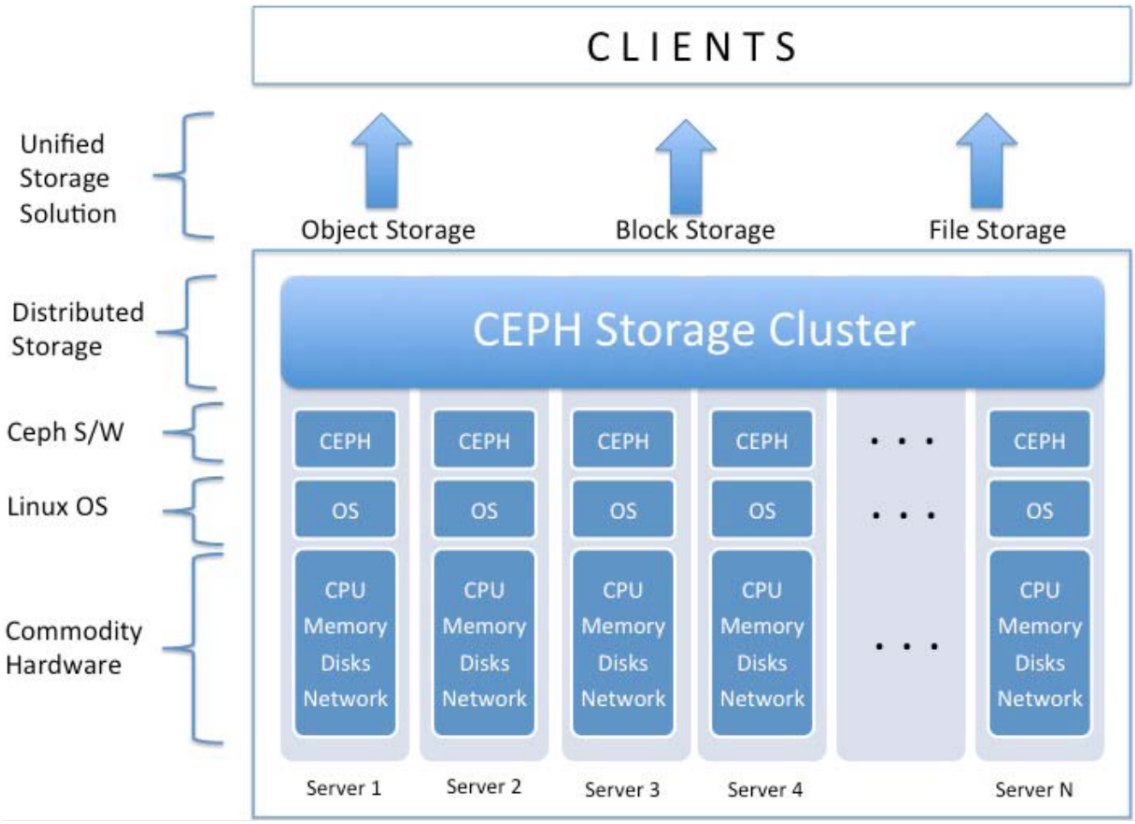



Figure 3-3: Ceph cluster [15]

However there are downsides to this file system that make it less than optimal for the purposes of this project. The first drawback is that Ceph does not offer POSIX API. This makes multiple write operations to different files with Horovod impossible. Although Ceph started out as a file system to be installed on top of an already existing local file system, its most used versions today are run directly on raw storage, this is at least in part thanks to Ceph not being the easiest file system to set up, lot of expertise is required for installing and configuring vanilla Ceph. Bare metal implementation has made great improvements with regards to its I/O overhead [16], however since the chosen file system will be running on virtual machines with a base operating system I cannot use these improved distributions.

### 3.2.2 HDFS

Hadoop Distributed File System (HDFS) was released in 2006 as a part of Apache Hadoop, a collection of open-source software that utilizes a network of computers for big data and computational use. Hadoop was a project inspired by Google's big data architecture which similarly has a cluster-based distributed file system called the Google File System. HDFS is Hadoop's storage part and is know for being a scalable, and portable file system written in Java.

It provides high throughput with larger files, with files used in a HDFS environment usually being several gigabytes to terabytes large [17]. High throughput could be ideal for applications with large data sets, which could make this file system ideal in supplying a distributed deep learning system that usually requires extraordinary amount of training data.

In the HDFS architecture, which can be seen on Figure 3-4, DataNodes are responsible for reading, writing and storing data. The NameNode regulates access to clients, stores metadata and executes file system namespace operations such as opening, closing and renaming files and directories. Metadata in the NameNode is stored in two distinct files. One of them scales automatically the other however does not, because of this when the server is restarted the more files stored in the cluster the longer it takes to reboot. Secondary NameNodes can be added to the cluster, to speed up the boot time.

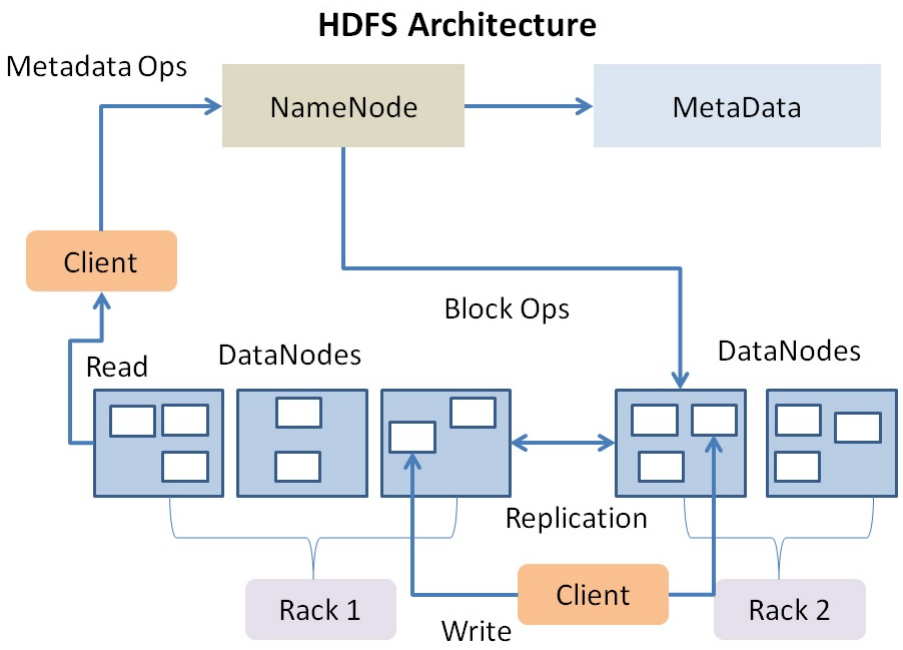



Figure 3-4: HDFS Cluster [18]

However HDFS notoriously struggles with small file transmission [19]. This is a significant complication since deep learning training data usually consists of smaller files. There are drop-in replacements for HDFS such as HopsFS that correct the slow small file operations, however these improved versions, just as the original are optimized to be used together with other members of the Apache Hadoop software family. Thus configuration would be cumbersome and the performance unpredictable. Unfortunately HDFS is also not POSIX compliant which is a must if it was to be used with Horovod.

### 3.2.3 GlusterFS

GlusterFS is an open source, Linux based distributed file system that was developed by Gluster in 2010 before being acquired by Red Hat in 2011. It was designed to be highly scalable however its scalability is very hardware dependent [20]. This file system is fully POSIX compliant and also supports FUSE which makes containerization and cloud storage more achievable since there is no need for implementing a kernel module. However Gluster has no central metadata storage, metadata is distributed which can cause data loss and also make removal cumbersome [21].

In Gluster shares on the kernel space file-system under GlusterFS are called volumes, these volumes are directly exposed to the clients as seen on Figure 3-5. Different servers can host bricks which are subvolumes that are assigned to volumes by different translators.

Translators can also make operations on the bricks besides connecting them to volumes.

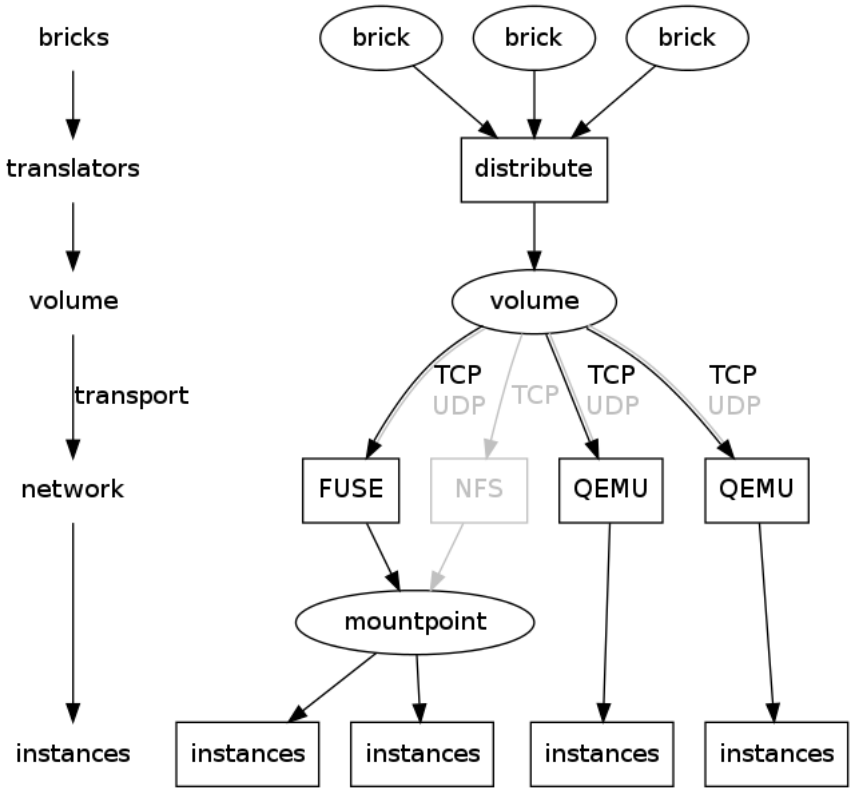



Figure 3-5: GlusterFS components [22]

When tested GlusterFS outperformed both Ceph and HDFS in read operations while narrowly following HDFS in write operations [21] as seen on Figure 3-6. However for my purposes only the performance of read operations are essential.

| | Ceph CF | GLUSTER 3.3 | HDFS |
|---|---|---|---|
| read | 126.91 | 427.3 | 220.05 |
| write | 64.71 | 268.57 | 275.27 |

Figure 3-6: Comparison of read and write speeds [21]

### 3.2.4 OpenAFS

OpenAFS is the earliest file system discussed in this thesis, its first iteration was released in 2000 by IBM. OpenAFS is the open-source implementation of the Andrew distributed file system (AFS) that was developed at Carnegie Mellon University. It offers a client-server architecture for distributing data, with support for UNIX, Linux, MacOS X and Windows.

It's key features include support for Kerberos based authentication, ACL protected directories, read-only file replication, file server location transparency, and client-side file caching. Since OpenAFS has been in development for over two decades it has a wide array of features and supporting systems like a dedicated Ansible collection and numerous third party versions.
AFS was closely developed with Kerberos 4 which is a network authentication protocol that is considered out of date and unsafe. Although the bad security can be fixed-up [23], it is not as convenient as using a system that has good security out of the box.
In most aspects OpenAFS is rather a competitor of the Network File System (NFS) than of other distributed file systems demonstrated on Figure 3-7. It has notable advantages over NFS but falls short behind other, more modern distributed file systems.

| | AFS | NFS/CIFS |
|---|---|---|
| Mount Points | One per client | One per filesystem |
| Hierarchy | Single global hierarchy | Local, requires auto-mounter / AMD |
| Caching | Consistent client-side caching | Minimal, usually implemented by 3rd-party products |
| Management | Online data migration | Offline data migration |
| Scalability | Highly scalable | Not scalable |
| Load Balancing | Automatic | None |
| Client fail-over (server failure) | Automatic | None |
| Performance | Multi-threaded server/client | Multi-threaded server/client |

Figure 3-7: Comparison of AFS and NFS/CIFS [24]

However in multiple studies even when compared to network file storage, for example NFS and Samba, OpenAFS had worse read and write speeds [25] [26]. The lack of performance makes OpenAFS especially unappealing for use as the storage structure for a high demand deep learning system.

### 3.2.5 MooseFS

MooseFS was introduced in 2008 by Core Technology. It is a fault-tolerant, open-source distributed file system that has high availability, performance and scaling. This File system is fully POSIX compliant which means that users can interact with files through a hierarchy of folders just as with centralized storage solutions, although the data itself is distributed throughout machines which make up the complete file system.
In the MooseFS infrastructure, which can be seen on Figure 3-8, data is stored on

chunkservers. These chunkservers are the physical machines that store the data saved to the storage cluster. The master server is responsible for storing metadata that can be used for the retrieval of files. In most cases the storage cluster has multiple master servers, from which the primary one is called the leader and the others are called the followers and serve as backups to the leader. Metaloggers create and store changelogs that can also be used in case the primary master server or the chunkservers fail. Additionally, the MooseFS storage cluster can have a CGI server which will provide a graphical interface for monitoring the cluster.

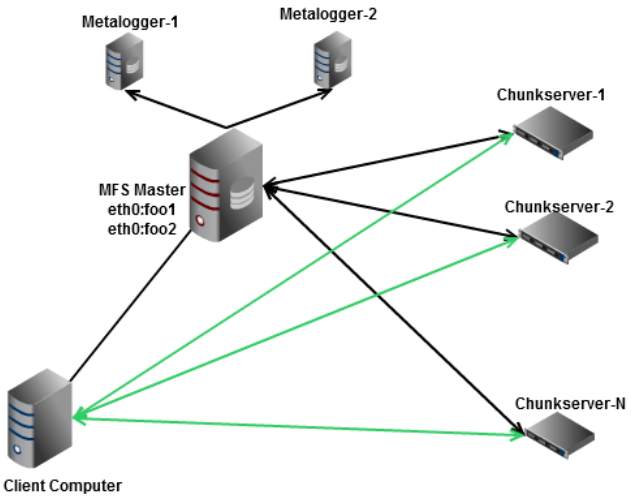



Figure 3-8: MooseFS Storage Cluster [27]

MooseFS stores all data with redundancy and in case of a failure the automatic failover mechanism will keep operations running. Even the hardware components can be changed underneath without having to interrupt file access. MooseFS has a working FUSE implementation for all supported operating systems. Thanks to its implementation of FUSE all file operations work the same way as they would on any other centralized file system [28]. As for the performance of MooseFS, in an article comparing its read and write speeds with HDFS and Lustre, MooseFS tied Lustre in write speeds however was outdone by HDFS when it came to read performance [29] as seen on Figure 3-9. This supposes that in a direct comparison between MooseFS and GlusterFS, the read speeds of MooseFS would be less than it would be for Gluster since GlusterFS executes read operations faster than HDFS [21].

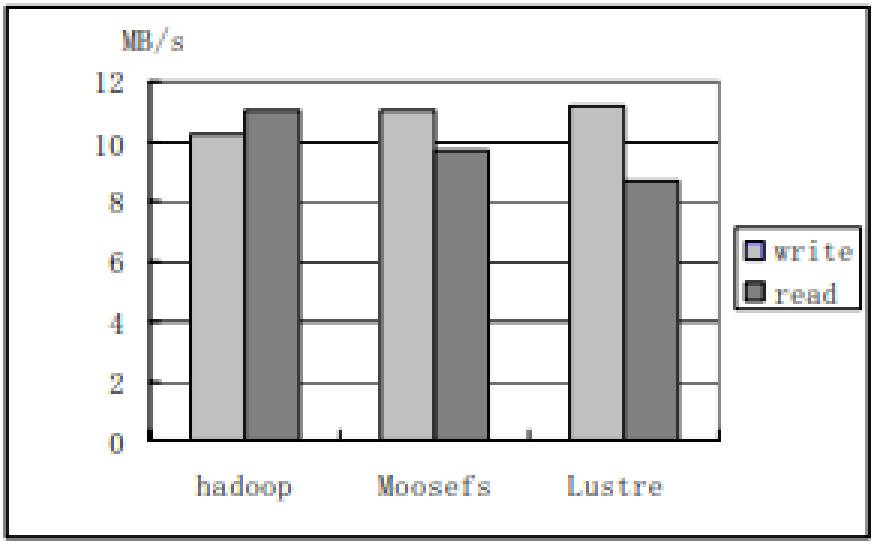



Figure 3-9: Comparison of HDFS, MooseFS and Lustre [29]

### 3.2.6 Selected distributed file system

In this section I will reiterate the most notable features of the file systems discussed above. Comparing them and choosing the one to be used in an implementation as a storage solution for a distributed deep learning system.

I first looked at Ceph which is one of the most popular distributed file systems. It provides a cheap way to scale up and out. However it barely misses full POSIX compliance, and it's most used distributions are designed to run on bare metal, meaning they are not suitable for running on a virtual machine with a pre-installed base operating system.

The Hadoop Distributed File System is Apache Hadoop's solution for a scalable, and portable file system. It performs exceptionally well with high throughput of large files making it a great choice for applications with large datasets. However, although deep learning requires high throughput of data, individual file size is usually under the gigabyte range. Which is a big problem for HDFS since it is known for being slow with small file transmission.

GlusterFS checks all compliance requirements for Horovod. It has a major difference from other distributed file systems in that it has no central metadata storage, which can lead to data loss. Gluster however more than makes up for the below average fault tolerance with its performance. When it comes to the read performance, GlusterFS has significantly outperformed both Ceph and HDFS.

OpenAFS is a distributed file system well past its prime. It was released more than twenty years ago which shows when compared with other, more modern file systems. It's security

and performance are two biggest drawback about OpenAFS, and although it provides a valid competitor for NFS and CIFS, it provide no real reason for choosing it in this day and age.

Lastly, MooseFS just like Gluster provides both POSIX and FUSE compatibility out of the box. This file system concentrates more on security than GlusterFS does, by using a central metadata server. However, it is beaten by Gluster when it comes to performance.

| Name | Release year | Open source | POSIX compliant | FUSE compliant |
|---|---|---|---|---|
| Ceph | 2010 | YES | NO | YES |
| MooseFS | 2008 | YES | YES | YES |
| HDFS | 2006 | YES | NO | YES |
| GlusterFS | 2005 | YES | YES | YES |
| OpenAFS | 2000 | YES | NO | EXPERIMENTAL |

Figure 3-10: Comparison of file systems

After researching five distributed file systems only two has satisfied every requirement needed for compatibility with the Horovod deep learning training framework, as seen on Figure 3-10. MooseFS and GlusterFS are both great choices as storage solutions for deep learning. While GlusterFS possibly provides better performance than MooseFS, in this thesis I will use MooseFS since it has better supporting material and more overall utilization than Gluster.

# 4 Cloud computing and containerization

Besides the automation tools and the chosen distributed file system, my implementation will be done with the application of cloud computing and containerization. These technologies will ensure that the finished implementation can be replicated and expanded with ease.

## 4.1 Cloud computing platform OpenStack

As mentioned before the finished implementation will be utilizing the flexibility, easy managability and other perks of modern clouds platforms. I chose to use OpenStack because it provides more flexibility than other services, and also this is the cloud provider that I am most familiar with and thus able to utilize it's functionalities fully.
OpenStack is an Infrastructure-as-a-Service provider. This means that it gives the highest amount of freedom to developers by letting them directly manage resources unlike Platform-as-a-Service and Software-as-a-Service providers which are more user friendly but provide much less in terms of customization and meaningful configuration options.
Infrastructure-as-a-Service type cloud providers pool computing resources, which then can be distributed by creating virtual machines that will be accessible through the OpenStack dashboard or if configured, by a remote machine through the internet.
This service has a modular architecture, meaning that it consists of various components as seen on Figure 4-1. Each component is responsible for a different functionality.

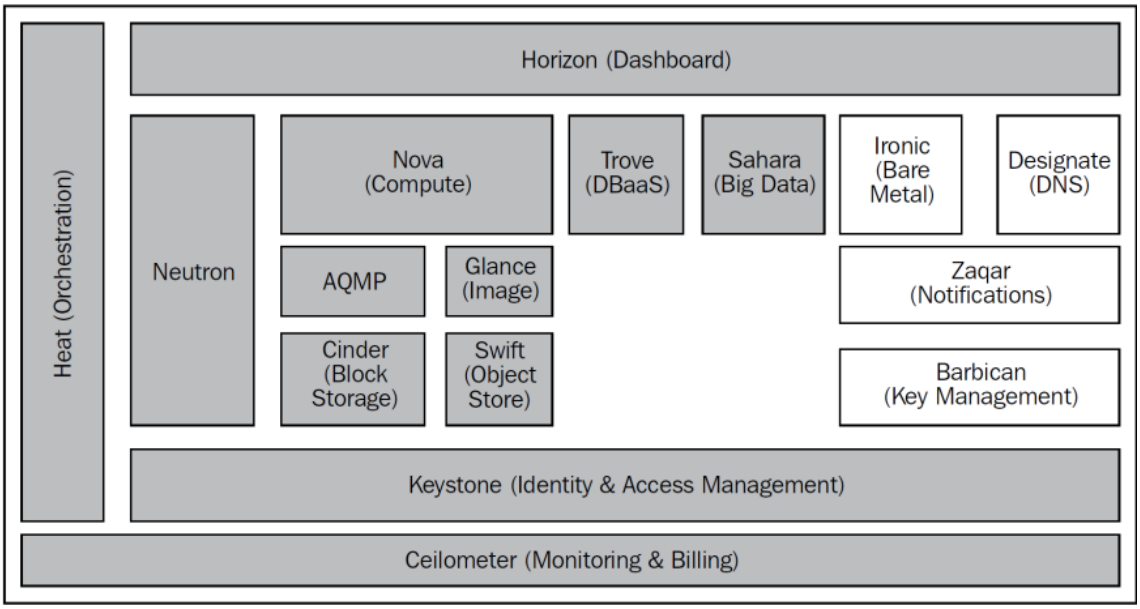



Figure 4-1: OpenStack components

Thanks to Nova I will be able to create virtual machines remotely from my local computer with the automation tool Terraform. Terraform interacts with the API of OpenStack which is managed by the Heat component. With this component infrastructure can be easily created and destroyed via API calls which is even safer than manual configuration, since everything can be undone, no lasting configuration errors can be made. Another component used when connecting through the API is Keystone. Thanks to Keystone my Terraform client will be able to authenticate to my OpenStack project.

When creating virtual machines OpenStack allows for the definitions of images that are managed by the Glance component. Images can be used to create virtual machines with a preconfigured operating system image. This will be very helpful during implementation since MooseFS has some software requirements that can be conveniently added this way, and will be installed whenever a new VM is started.

Lastly, my created virtual machines will also need to communicate with each other. This need is solved by Neutron which provides networking functionalities. Besides allowing virtual machines to communicate with each other through virtual networks that can be created in OpenStack, Neutron also provides so called floating IPs. Floating IPs are public IP addresses that can be assigned to a virtual machine thus providing it with the capability to communicate over the internet. This will allow me to configure the created virtual machines with an automated configuration tool which need to be able to reach the target machine through SSH.

## 4.2 Containerization with Docker

Besides using cloud infrastructure, my project will also use containerization technology to ensure replicability and consistency with each application.
To this end, I will use Docker which is the most well known and utilized containerization engine. With Docker it is possible to create isolated execution environments, called containers. Containers work similarly to virtual machines as far as the creation of an environment which is independent from the underlying software of the host machine and a way to replicate these conditions and environments without regards to the underlying hardware and operating system.
However, the main difference between a virtual machine and a Docker container is that while a virtual machine will require the installation of an operating system and the replication of a whole running, functional host, a container will only run the containerized application. This means that if we want to have a single application in an easily portable and replicable package than we are much better off with a Docker image, since a virtual machine image will contain a whole bunch of unnecessary software and data that we do not need, increasing the size of the image to be much larger and harder to manage.
Another advantage to using Docker is the Docker hub. The Docker hub is an online database of images that can be downloaded and ran by our local Docker client application. Docker hub contains most of the applications one could have a need for, but if something is not available there, other repositories are also usable with Docker. Besides downloading images Docker also provides an easy way to creating and uploading our own, locally built images as well. These images then can be used in other projects as well as shared with the Docker community.
I my thesis I will be using Docker when configuring the different MooseFS roles inside the storage cluster. On each created virtual machine in the cloud a single Docker container will be started inside which MooseFS will run the role that is filled by that machine. Instead of using basic Docker, my implementation will use Docker-compose which is a more sophisticated version of the regular Docker engine that allows for running multiple containers at once or as it will be used in my case, it can be used for making extra configurations steps before and after running the Docker image.

# 5 Detailed system plan

I have successfully compared potential software tools to be used in the implementation. Now based on these components the system plan for the whole infrastructure can be designed. The system plan will describe the interactions, and connections between the deep learning framework (Horovod), the distributed file system (MooseFS) and the automation tools (Ansible, Terraform).

## 5.1 Configuring the file system

The MooseFS implementation will be installed on an array of virtual machines located in the cloud. The cloud I will be using is the ELKH Cloud[30], which is an OpenStack cloud service used for different types of research projects. It allows for the creation of e-infrastructure that is tailored to the needs of the given project. By creating quotas and allocating resources differently projects can specialize for different applications of cloud environment such as Spark clusters, Kubernetes clusters or deep learning support environments. All projects and their reference architectures are publicly available to inspect on the official website, with some of the most advanced and groundbreaking research taking place in this cloud jointly operated by the Eötvös Loránd Research Network, the Institute for Computer Science and Control (SZTAKI) and the Wigner Physics Research Institute. With access to the ELKH Cloud I will have the necessary resources to create my implementation.

Virtual machines in the cloud will be set up using Terraform. As an open-source infrastructure automation tool, it will make configuration of the initial virtual machines as well as later additions easier, while also ensuring uniformity between them. The image that will be installed using Terraform will also contain a base operating system, of Ubuntu 20.04 since it has low hardware requirements and is fully compatible with MooseFS.

After having set up the desired number and type of virtual machines the installation of MooseFS can begin. For the automation of this process I will use Ansible which makes the remote execution of configuration steps possible. Ansible works with playbooks, which are predefined configuration steps. For setting up a full implementation of MooseFS I will need to provide Ansible with a playbook for every role that machines can take inside the storage cluster, which consists of a master server, metalogger, chunkservers, CGI server, CLI and client.

These roles will be implemented on the created virtual machines inside Docker containers. By using containers I can further ensure that the implementation will be replicable and

can be reliably recreated in the future.

As for the different MooseFS roles, each of them have different responsibilities. For example, chunkservers are responsible for storing data with redundancy. The number of chunkservers can be scaled in theory infinitely, new machines can also be added even when the file system is in use. As for the size and specification, chunkservers do not need to be of the same size or specification.

The master server stores the metadata needed for retrieval of files from the chunkservers. When a client wants to interact with data on the cluster it has to request the corresponding metadata from the master server to be able to find the information it is looking for. The master server however does not handle the actual data, the retrieval happens between the client and the chunkservers. The master server also manages access to the file system by controlling which clients can mount the storage. This authorization happens based on IP address. Being a critical component the master servers should have a backup. In the MooseFS architecture additional master servers, called followers, can be configured which can take the place of the primary master server, called the leader. However the open-source version has no option for having more than one master server. Instead, a metalogger can be added instead.

The metalogger or metadata backup server is connected to the master server. Its task is to log changes in the metadata so that in case of the master server failing the changelogs can help in the restoration of lost information. In a cluster where multiple master servers are configured metaloggers are optional, however without a follower a metalogger is strongly recommended. By configuring a virtual machine for metalogger which is at least as powerful as the master server, it can be easily configured to take the place of the leader.

The CGI server will provide the client with a monitoring interface that has extensive documentation of machines inside the cluster as well as information about the exchange rate of data, free space, disk utilization and so on.

The MooseFS CLI is also a helpful tool in monitoring the status of MooseFS. It provides a command line interface which provides access to the same information that can be observed in the CGI, however CLI can be used with scripts, allowing the creation of automatic health checks with Ansible.

Lastly the client will provide access to the storage. MooseFS will be mounted to the client just like any ordinary storage. Thanks to the POSIX compliance of the file system data will be accessible exactly the same way as it would be on a centralized file system, through a hierarchy of folders. This client will provide the access point to the storage for

the deep learning framework, Horovod.

## 5.2 Automation of the build process

By using automation setting up all components of the distributed file system, adding new components in the future, and replicating the whole configuration will be come much less burdensome while also guaranteeing the considered result every time.

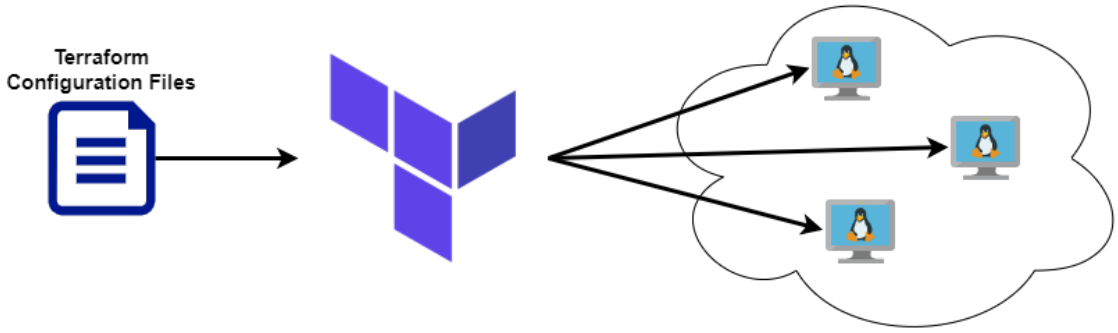



Figure 5-1: Terraform automation

For setting up the virtual machines that will fill different roles in the storage cluster I will use Terraform as seen on Figure 5-1. For cloud virtual machine installation I will use predefined image that also contain the operating system. This way by using this tool I can setup an array of identical virtual machines running Ubuntu 20.04 in the cloud. The creation of the virtual network for my virtual machines will also be done using Terraform. The virtual network will provide connectivity between the VMs

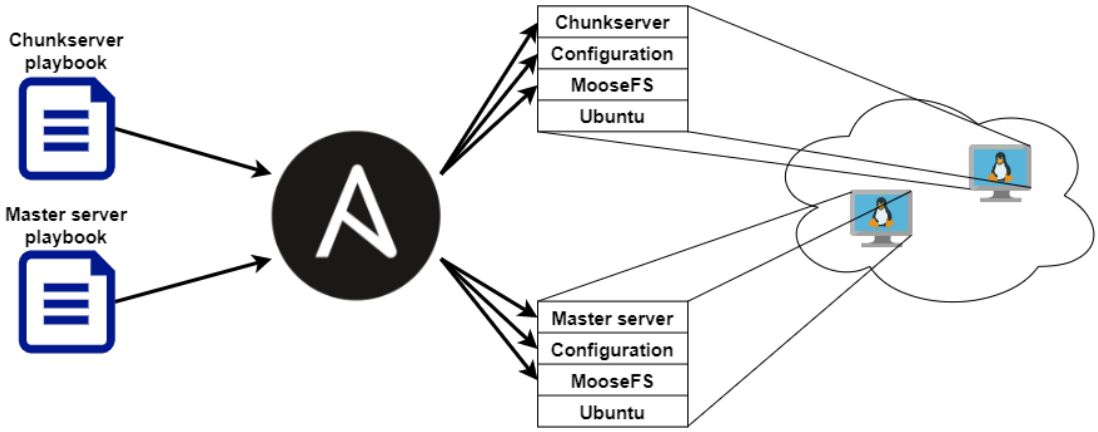



Figure 5-2: Ansible automation

After having successfully set up the required number virtual machines I can start to configure them as different parts of MooseFS. For this I will use Ansible. If there is a process that can be executed in a command line on a machine that can be accessed remotely, there is a good chance that it can be automated with Ansible. For this project I will need to configure the created virtual machines according to the role the given VM will fill. This means that playbooks will have to be created for the chunkservers, master server, metalogger, CGI server, CLI, and client. Thanks to the virtual network created using Terraform the virtual machines can be configured remotely, from a single machine.

## 5.3 Connecting to the deep learning infrastructure

After having set up and made sure that my MooseFS implementation is running correctly I have to connect it to Horovod. To do this I will configure the virtual machine that Horovod runs on as a MooseFS client. Then give access for this client to freely mount the file system, which can be done through the master server of MooseFS. By mounting the file system, the training data's location can be specified using a simple file path. This is thanks to MooseFS presenting the data stored in the cluster through a hierarchy of folders just like a centralized file system would.

## 5.4 System architecture

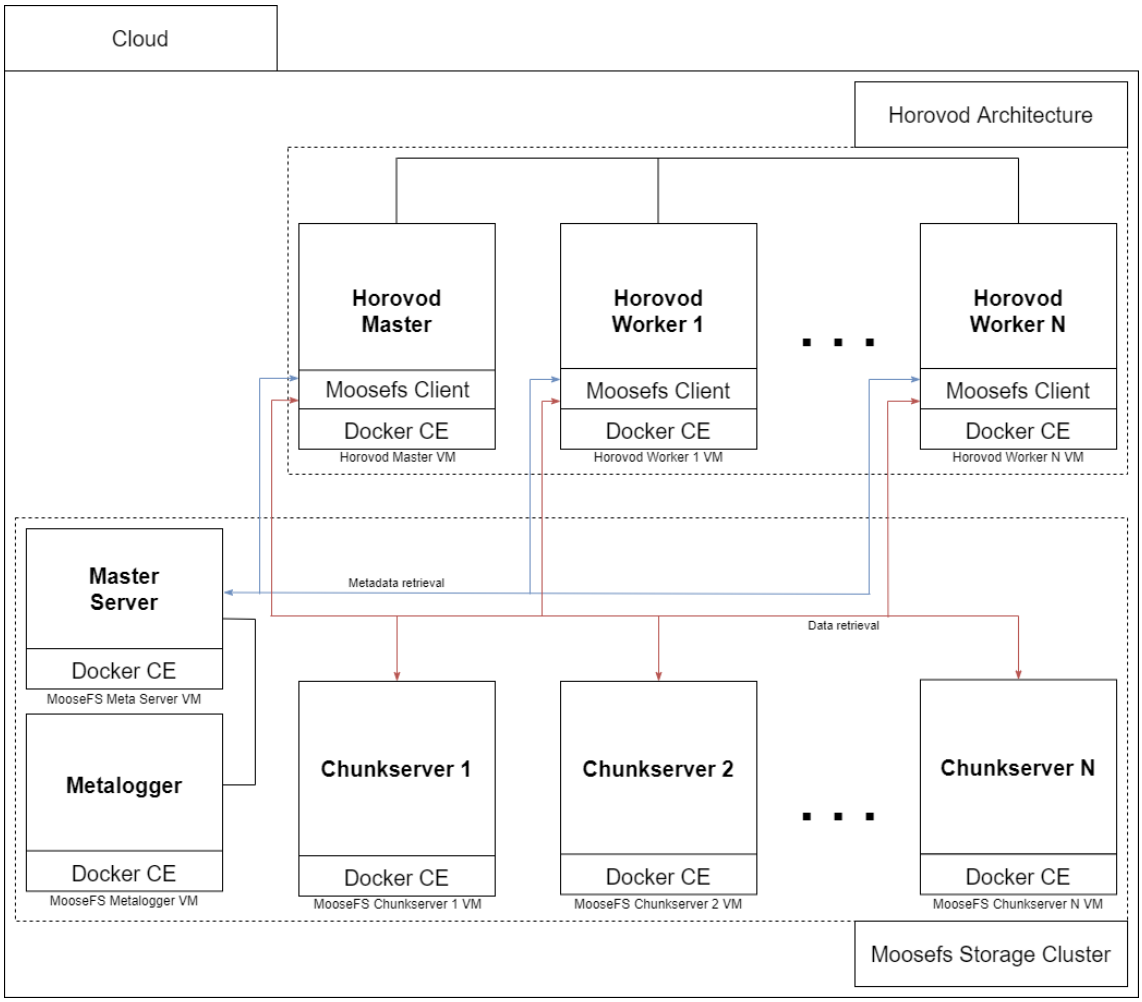



Figure 5-3: Complex architecture

In the finished implementation, as seen on Figure 5-3, the whole system will be running in the cloud on separate virtual machines. These virtual machines will run the Docker images of different MooseFS roles. VMs inside Horovod will all have the MooseFS client role in order to be able to mount the file system. The clients will request the metadata from the master server with which they can retrieve the actual data from the chunkservers. The master server will also have a metalogger attached which will monitor changes in the metadata and make recovery possible in case the master server's data was lost.

# 6 Implementation

My finished implementation of a MooseFS cluster featured as the storage for a Horovod distributed deep learning infrastructure has been implemented so that it could be rebuilt and replicated with the least manual configuration. This is achieved with the automation tools Terraform and Ansible and by using Docker containerization. The cluster itself is built inside OpenStack which provides Infrastructure-as-a-Service in the form of private and public clouds.

## 6.1 Infrastructure creation with Terraform

The first step in the implementation is the creation of the virtual machines and networking infrastructure required for a working MooseFS storage cluster. To achieve this I created a Terraform file in which I defined OpenStack as a provider. By specifying my and the project's credentials, Terraform is able to connect to the project and create the infrastructure. Although Terraform creates its own SSH keys, it is not possible to use them outside of Terraform, because of this I locally created an SSH key which I then used for the creation of a key pair in OpenStack.
The first virtual machine created will play the role of master server in MooseFS and also acts as a gateway through which all the other virtual machines get configured. This, and all the other virtual machines are configured with the m2.large flavor containing 4 CPU cores and 8 gigabytes of ram, which satisfies the minimum requirements for both the operating system and MooseFS. Additionally 25 gigabytes of storage and a Ubuntu 20.04 base image is configured, that includes a pre-installed version of Docker, Docker-compose and GIT. I also attach the previously created key pair to the new machines and configure two security groups, one of which is responsible for allowing SSH connections and another which allows network traffic between machines, required by MooseFS and Horovod. After creating the first virtual machine I attach a public IP address to it, referred to as a floating IP in OpenStack. By associating the VM with this IP and allowing remote connections through a security group, it is possible to reach, and configure the virtual machine from across the internet with my local host.
I then define the parameters for the eight other virtual machines, namely a CGI server, a meta logger, four chunkservers, and two clients. All virtual machines are identical and configured with the resources mentioned above. Besides having attached a floating IP to my master server, I also attach one to one of the MooseFS clients which will serve as the Horovod master, this is needed so that I could use the Jupyter notebook capabilities of

this Horovod build.

After the successful creation of my hosts I create an Ansible inventory file in the same Terraform file by creating a string that references the private IP addresses of the created VMs as seen on Figure 6-1. Since the IP addresses are randomly assigned from the network's addressing range the IPs of individual virtual machines change with every build. With my solution this will never cause a problem, all created VMs will be referenced correctly. The created file will later be transferred to the master server by Ansible, where it will be used during the configuration of individual hosts.

```
resource "local_file" "hosts" {
  content  = "[mfscgi]\n ${openstack_compute_instance_v2.mfscgi.access_ip_v4} ansible_user=ubuntu
  \n[mfsmetalogger]\n ${openstack_compute_instance_v2.mfsmetalogger.access_ip_v4} ansible_user=ubuntu
  \n[mfschunkserver1]\n ${openstack_compute_instance_v2.mfschunkserver1.access_ip_v4} ansible_user=ubuntu
  \n[mfschunkserver2]\n ${openstack_compute_instance_v2.mfschunkserver2.access_ip_v4} ansible_user=ubuntu
  \n[mfschunkserver3]\n ${openstack_compute_instance_v2.mfschunkserver3.access_ip_v4} ansible_user=ubuntu
  \n[mfschunkserver4]\n ${openstack_compute_instance_v2.mfschunkserver4.access_ip_v4} ansible_user=ubuntu
  \n[mfsclient]\n ${openstack_compute_instance_v2.mfsclient.access_ip_v4} ansible_user=ubuntu
  \n[mfsclient2]\n ${openstack_compute_instance_v2.mfsclient2.access_ip_v4} ansible_user=ubuntu"
  filename = "../../ansible/jumpServerAnsible/hosts"
}
```

Figure 6-1: Creation of Ansible inventory

Lastly, in my Terraform file I locally execute the Ansible playbook responsible for configuration of all the hosts. Since executing Ansible through Terraform returns with the same result and even prints out the output of the running playbook, there is no downside to running one automation tool inside another. It can even be more efficient since Terraform logs the time an operation takes, logging it to console every ten seconds. Since Ubuntu is prone to stopping the outputting of console logs, only reacting when an input is given, with Terraform it is obvious when the logging gets stuck.

## 6.2 Configuration with Ansible

As the last step of the Terraform configuration an Ansible playbook is ran, this playbook configures all the hosts, sets up both a MooseFS storage cluster as well as a Horovod cluster. Besides a playbook, Ansible requires an inventory file, which contains the hosts we want to configure. Since I will be either configuring the master server or using the master server as an intermediary configuration machine, I only need to define this one VM's public IP address in my local machine's inventory.

The playbook starts off by copying the local host's private key to the master server. This same key is used for the connection between the local machine and the master server, and

will subsequently be used for connections between the master server and all the other VMs which are to be configured. The other machines possess the public counterpart to this key as an OpenStack key pair, which was configured at their creation by Terraform. Since access to a copied private key will be too lenient for SSH, I had to change its permissions after copying, so that only the owner would be able to read the file which contains the key. To use the master server as a configuration machine I also need to have Ansible installed on it, so in the next play I am installing Ansible through pip, I chose to use pip since it installs a much newer version of Ansible than apt, and thus provides access to modules the earlier version does not. This includes the collection which is used for running Docker-compose via Ansible.

Running Ansible on the master server also requires an inventory file, this was created by Terraform before launching the Ansible playbook, the inventory or hosts file contains the private addresses and a group name for all other virtual machines. The group name can be used inside the Ansible playbooks to define which host should be configured. It is possible to have multiple machines' IP addresses under one group, in this case if a playbook is called for that group all hosts in the group will be configured with the same playbook separately. I however, kept the hosts separate in my implementation, defining one host per group as seen on Figure 6-2.

```
[mfscgi]
 192.168.0.44 ansible_user=ubuntu
[mfsmetalogger]
 192.168.0.213 ansible_user=ubuntu
[mfschunkserver1]
 192.168.0.101 ansible_user=ubuntu
[mfschunkserver2]
 192.168.0.19 ansible_user=ubuntu
[mfschunkserver3]
 192.168.0.75 ansible_user=ubuntu
[mfschunkserver4]
 192.168.0.158 ansible_user=ubuntu
[mfsclient]
 192.168.0.193 ansible_user=ubuntu
[mfsclient2]
 192.168.0.53 ansible_user=ubuntu
```

Figure 6-2: Ansible inventory

Although the chunkserver and client configurations could be put in collective groups, I opted for keeping them separate. However, this is something that might later be changed and I talk about it in more detail in the Future Development section.

At this point all that is required for a configuration machine, except for the playbooks, have been established on the master server. Before configuring all the other hosts however,

the playbook configures the master server itself. This is done by cloning a git project created and maintained by the creators of MooseFS. This is a Docker-compose project that is created for a simple demonstration of MooseFS and by starting it, a complete storage cluster will be started with each role on the cluster running in a different Docker container. If the cloned directory is built and ran with Docker-compose without any modifications a complete storage cluster would be set up. However since I want to run the different roles on their own separate virtual machines, for each role I created a Docker-compose file that will only run one of the Docker containers. So as the next play, I clone this git project to the master server, copy the modified Docker-compose file from my local machine, and run Docker-compose creating the MooseFS master server.
The modified Docker-compose files' content differ for each MooseFS role. As an example a chunkserver's Docker-compose file can be seen on Figure 6-3. A Docker-compose file can have multiple, individual images defined, from which the containers are built. In this case there is only the location of the chunkserver image defined. By defining the network mode as host, the container will use the host's network, without it containers on the different VMs would be unable to communicate. Next the container name and a label is defined. Labels are used for grouping chunkservers when they are inspected on the CGI server. In the volumes section the directories of the virtual machine are mapped to the container's directories. Lastly under networks by defining an external network to be used by the container it will use the host's network to reach the other MooseFS containers on different cirtual machines.

```
version: '3'
services:
  mfschunkserver3:
    build: ./moosefs-chunkserver
    network_mode: "host"
    container_name: "mfschunkserver3"
    environment:
      - LABELS=M
      #- SIZE=100
    volumes:
            - ./data/cs1/hdd0:/mnt/hdd0
            - ./data/cs1/meta:/var/lib/mfs

networks:
  default:
    external: true
```

Figure 6-3: Chunkserver docker-compose file

With the configuration of the master server finished, all the other VMs can be configured as well. For the configuration of these machines, I have created playbooks and modified

Docker-compose files. The latter is a modified version of the original Docker-compose file, which just like on the master server will only start the container corresponding to the role of the target machines. The playbook on the other hand will clone the git project, move the modified Docker-compose file into the cloned project, and then build and run the MooseFS project. The simplified process of host configuration can be seen on Figure 6-4.

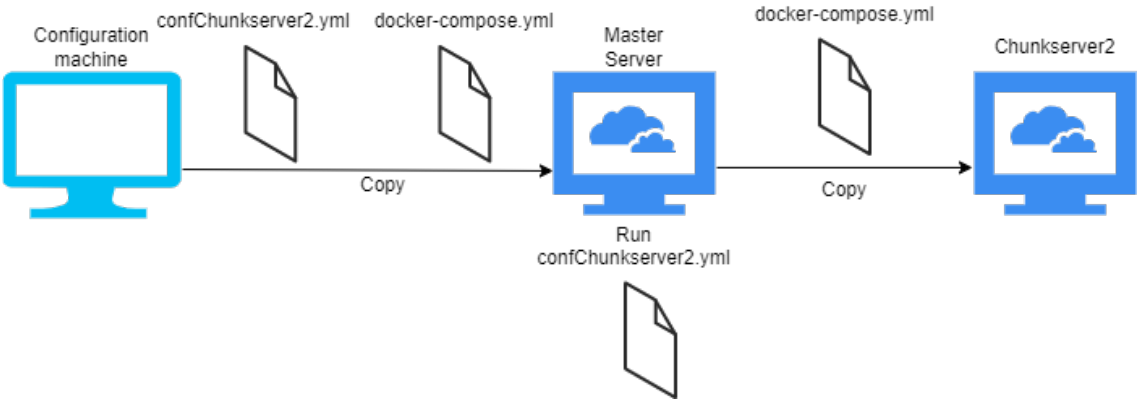



Figure 6-4: Chunkserver configuration

The only exceptions from this method of configuration are the MooseFS client machines, since these are configured using regular Docker instead of Docker-compose. As such, the playbooks with which the clients are configured are configured do not clone the aforementioned repository, nor do they need a custom Docker-compose file. These playbooks are copied to the master server where they are ran to configure the two clients.

## 6.3 Horovod integration

The initial plan for integrating my configuration with Horovod was that on the two MooseFS clients I would install a Horovod master to one and a Horovod worker to the other. This way the Horovod implementation would not be disturbed, no modifications would be made to it's configuration. I would then create Docker volumes of the /mnt/-moosefs folder, which is folder that can be used for storing data on the storage cluster. Docker volumes are used for persisting data generated by the containers as well as for communicating data between containers. This means that by creating a Docker volume, directories can be shared between containers. Although I was successful in the creation and sharing of this volume being able to see the shared directories on the Horovod container, the directory's content was not updated afterwards. This meant that even though

the folder is shared, Horovod had no access to it's content. Due to these complications I ended up combining the Horovod and MooseFS client images in a single Docker image. Luckily the Horovod implementation is based on a newer Horovod image that uses Ubuntu 20.04. This is important since older variants use Ubuntu 18.04 as their base image. This older operating system will not be able to host MooseFS since it requires version 3 of fuse and the older Ubuntu version is only compatible with version 2.

I started the creation of a new image by cloning the repository containing the files used for building the Horovod Docker image that I used. The Horovod image is made up of a Docker file and an entrypoint file that is ran when the Horovod image is successfully created, and is used for basic configuration. Meanwhile the Docker file, which is based on the official Horovod image downloads Jupyter lab, which is later used as an interface to Horovod, and downloads and mounts an NFS file system. In this Horovod implementation NFS provides the means for sharing files between the master and worker nodes. Although the aim of my project is to replace NFS with MooseFS, I did not remove it, since in future development the capabilities and performance of the two solutions can be compared and measured by running them side by side.

As for merging my MooseFS implementation with the Horovod image files, I had to extract the content of the Docker file that is ran by Docker compose for starting a MooseFS client. I then added the content of this file to the before mentioned Horovod Docker file resulting in code seen on Figure 6-5. The lines added are responsible for installing dependencies required by MooseFS, as well as the downloading and installation of the file system's client software.

MooseFS installation

```
FROM horovod/horovod-cpu:sha-464c82e

RUN cp -r /horovod/examples /

RUN pip install jupyterlab==3.4.0
RUN apt-get update && apt-get install -y \
    nfs-common \
    && rm -rf /var/lib/apt/lists/*

ENV NFS_MOUNT_SSH=/root/.ssh
ENV NFS_MOUNT_HOROVOD=/horovod

RUN apt-get update && apt-get install -y wget fuse3 libfuse3-3 gnupg2 python3

RUN wget -O - http://ppa.moosefs.com/moosefs.key 2>/dev/null | apt-key add - 2>/dev/null

RUN echo "deb http://ppa.moosefs.com/3.0.115/apt/debian/buster buster main" > /etc/apt/sources.list.d/moosefs.list

RUN apt-get update && apt-get install -y moosefs-client moosefs-cli

COPY docker-entrypoint.sh /
WORKDIR /horovod

ENV JUPYTER_PASSWORD=

ENTRYPOINT ["/docker-entrypoint.sh"]
CMD ["--ip=0.0.0.0", "--no-browser", "--config=/var/lib/jupyter/jupyter_lab_config.py", "--allow-root", "--notebook-dir=/horovod"]
```

Figure 6-5: Docker file content

The other part of the Docker image, the entrypoint file also had to be modified. In the

original Horovod implementation this file configures the NFS file system to a working state, by sharing information with NFS such as the location of the folder in which the files shared through NFS should be located, as well as generating an SSH key, which it then shares with NFS to ensure secure connectivity. Lastly, in the entrypoint file Jupyter lab is started, which can then be used as an interface for the Horovod master. In the case of worker nodes Jupyter will not be started, instead the entrypoint file concludes in an infinite sleep cycle, ending further operations.
In order for MooseFS to work properly I had to extend this file with the steps needed for mounting the distributed file system. These steps consist of creating the directory that is shared across the storage cluster and then mounting it to MooseFS. For the workers, I could do this without any complications, by replacing the infinite sleep cycle with the aforementioned steps as seen on Figure 6-6, with the now removed code added as a comment. For the master node however, which will conclude operations by starting Jupyter, this approach would not work. For reasons which I discuss in detail in the Furher Develoepement section. For now the initial configuration steps done on the worker nodes when the entrypoint file is ran have to be manually executed on the master node.

```
if [[ "${JUPYTER_LAB}" = "true" || "${JUPYTER_LAB}" = "True" || "${JUPYTER_LAB}" = "TRUE" ]]; then

    CONFIG_DIR=/var/lib/jupyter/
    if [[ ! -f "$CONFIG_DIR/jupyter_lab_config.py" ]]; then
        mkdir -p $CONFIG_DIR
        export JUPYTER_PASSWORD_HASH=$(python -c "from notebook.auth import passwd; print(passwd('$JUPYTER_PASSWORD', 'sha1'))")
        echo "c.ServerApp.password = '$JUPYTER_PASSWORD_HASH'" >> $CONFIG_DIR/jupyter_lab_config.py
        echo "c.ServerApp.terminado_settings = {'shell_command': ['/bin/bash']}" >> $CONFIG_DIR/jupyter_lab_config.py
    fi

    set -- jupyter lab "$@"
    exec "$@"
else
    # sleep infinity
    mkdir /mnt/moosefs
    mfsmount /mnt/moosefs -f
fi
```

Figure 6-6: MooseFS mounting integrated into entrypoint file

Before building the new Docker image I also had to change the permissions for the entrypoint file, leaving them as is will not thrown an error during building the image however once the image is started it will not execute properly due an inaccessible entrypoint file. Finally, I created the new Docker image and uploaded it to Docker Hub. I then replaced the references in the playbooks to feature my new image and also removed the plays in my Ansible playbook which were building the MooseFS clients, since from now on Horovod and MooseFS are using the same image. Lastly I had to extend the playbook to satisfy MooseFS's dependencies.

## 6.4 Cluster inspection with CGI Server

The finished MooseFS cluster contains a CGI Server which is used for monitoring the storage of each chunkserver and the cluster as a whole as well as providing information about the clients mounted and much more.

The CGI server provides a dashboard through a web page hosted on the virtual machine's 9425 port. This means that besides being accessible on the VM itself, the dashboard can also possible be used from a remote host if the host has access to the above mentioned port. First I created a new security group in Terraform that allows outgoing traffic to any IPv4 and IPv6 address and allows incoming traffic to the above mentioned port. Then by allocating a public IP address to the CGI server virtual machine and adding the new security group I was able to access the interface from my local machine.

On the dashboard's homepage information about the master server such as CPU and RAM usage as well as information about the cluster's storage can be inspected. The amount of total and free storage space as well as the number of files and directories in the storage are also logged here as see on Figure 6-7. Under the servers tab we can see all the chunkservers with information about their individual storage. For an even more detailed look at storage we can navigate to the disks section, under which the individual disks inside the chunkservers are listed. Since my implementation uses virtual machines, MooseFS detects storage as if every chunkserver would have a single disk. But of course without cloud usage or by mounting multiple block storage to one or more of the virtual machines we would be able to inspect them here separately.

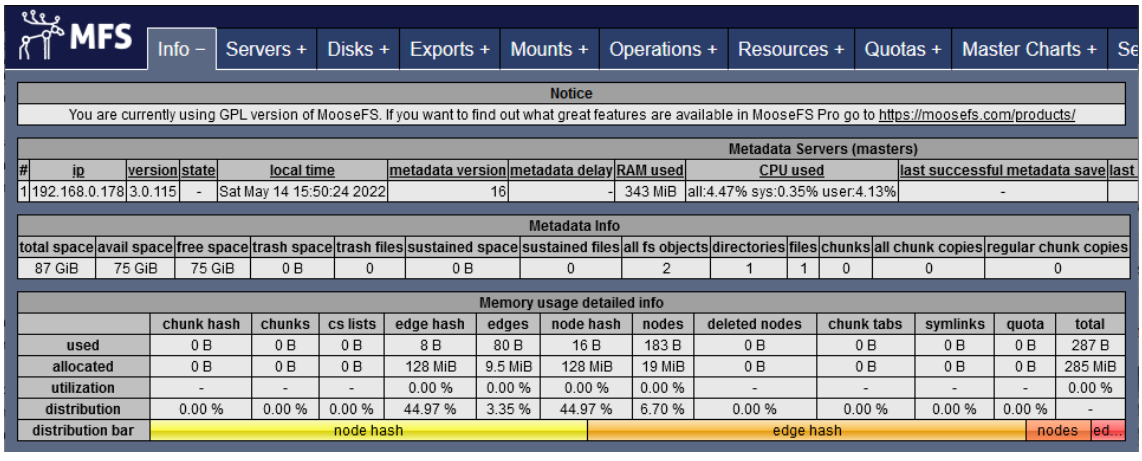



Figure 6-7: CGI Server dashboard

Under the mounts tab we can see information about every mounted host, including their IP address, the directory they have mounted, their permissions, the version of MooseFS they

are running and much more. Lastly, under master charts we can get the same information that is displayed on the main page about the master server's resources, however here we can see how utilization changes overtime.

# 7 Testing

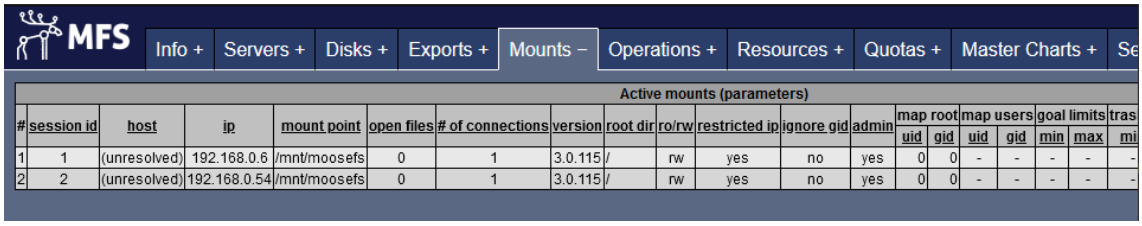
After managing to set up the MooseFS storage cluster with Horovod integration, I tested whether the file system and Horovod both work as expected. I had to make sure that files added to the folder which corresponds to the combined storage of the file system, does in fact store files in the cluster and I am able to reach and modify the content of this directory from any MooseFS client. The other and arguably just as important part of testing was making sure that Horovod teaching works correctly. Since I modified the Docker image that runs Horovod there was the possibility that in the process of merging the MooseFS and Horovod images together I broke Horovod and it would no longer function as intended. This could include the Horovod worker and master nodes not seeing each other, not being able to communicate due to some kind of networking configuration error, the teaching process refusing to run, or Jupyter notebook not functioning properly. Lastly, after validating both software individually I had to test whether they work as intended when their functionalities are combined. Namely, is Horovod able to reach the storage of MooseFS, is it possible to share files needed for the teaching process through the storage cluster, and is Horovod able to start the teaching process using the shared file?

Firstly, after setting up the MooseFS storage cluster, but before integrating it with Horovod I wanted to test whether the file systems components are connected correctly. I done this by creating two MooseFS clients and adding files to the /mnt/moosefs directory, which is the directory mounted to the file system. Through this process I could confirm that both clients will see the same files, staying completely in sync. Configuring new clients and adding them to the same cluster is also possible, with their directories getting updated as expected. Even deleting the mounted directory and then recreating it would not break the file systems operation, since as soon as the folder is recreated it's content will be updated to match that of the others.

As for Horovod, after creating a combined Docker image of it and MooseFS, I wanted to make sure that Horovod works properly and can execute the teaching process as before. For this I used the one of the teaching files that are installed with the version of Horovod I used. At this time, the file sharing between the Horovod nodes was not managed by MooseFS but rather by running NFS which was the original way this implementation of Horovod shared files between nodes. By defining the IP addresses of the two nodes and pinpointing the file I wanted to use for the teaching I was able to successfully execute distributed teaching, although without any distributed file system usage at this time.

To make sure that the newly configured clients are mounted correctly and to also test the

CGI server, I opened the dashboard of the CGI server through a web browser on my local computer. There I was able to confirm that both clients are mounted as seen on Figure 7-2. As well as I was able to make sure that all the chunkservers are connected to the cluster. According to the CGI server the cluster had four connected chunkservers and 22 GiB per chunkserver which is the amount I configured them with.




| # | session id | host | ip | mount point | open files | # of connections | version | root dir | ro/rw | restricted ip | ignore gid | admin | map root | | map users | | goal limits | | tras |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | uid | gid | uid | gid | min | max | mi |
| 1 | 1 | (unresolved) | 192.168.0.6 | /mnt/moosefs | 0 | 1 | 3.0.115 | / | rw | yes | no | yes | 0 | 0 | - | - | - | - | - |
| 2 | 2 | (unresolved) | 192.168.0.54 | /mnt/moosefs | 0 | 1 | 3.0.115 | / | rw | yes | no | yes | 0 | 0 | - | - | - | - | - |

Figure 7-1: Mounted clients inside CGI Server

Lastly, I tested the finished implementation, by running first seeing whether the file system still works correctly. I did this by copying the Python file I used for teaching to the shared folder on one of the clients and see whether it is present on the other client. As it can be seen on Figure 7-2 the file appeared on the other client correctly.

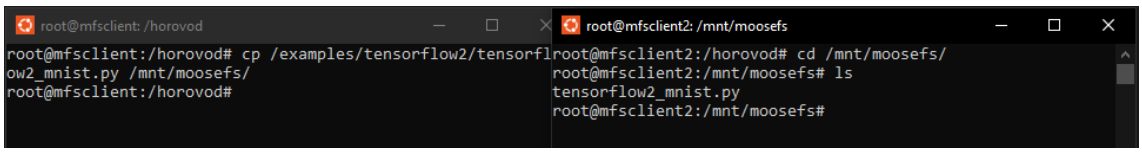



Figure 7-2: File shared between clients via MooseFS

Then I ran the same teaching process as before however this time through a Jupyter notebook terminal with the required file being shared through the MooseFS storage cluster. As previously mentioned, on the worker node I copied the aforementioned file to the shared MooseFS folder. I then opened Jupyter on the Horovod master on my local computer. Since a public IP address has been assigned to the master, the Jupyter notebook exposed on the node's 8888 port will be reachable through the internet. In Jupyter I opened a terminal and created the folder which I then mounted to the file system. After mounting it, the directory is updated and contained the file I placed there on the worker node. Then I started the teaching process with that file, and was able to confirm that the teaching is working as intended as seen on Figure 7-3.

```
[1,0]<stdout>:Step #0   Loss: 2.309310
[1,1]<stdout>:Step #0   Loss: 2.305134
[1,0]<stdout>:Step #10  Loss: 0.868170
[1,1]<stdout>:Step #10  Loss: 0.787745
[1,0]<stdout>:Step #20  Loss: 0.535032
[1,1]<stdout>:Step #20  Loss: 0.504290
[1,0]<stdout>:Step #30  Loss: 0.327684
[1,1]<stdout>:Step #30  Loss: 0.432973
[1,0]<stdout>:Step #40  Loss: 0.300651
[1,1]<stdout>:Step #40  Loss: 0.302894
[1,0]<stdout>:Step #50  Loss: 0.448973
[1,1]<stdout>:Step #50  Loss: 0.281442
[1,0]<stdout>:Step #60  Loss: 0.257108
[1,1]<stdout>:Step #60  Loss: 0.334507
[1,0]<stdout>:Step #70  Loss: 0.327279
[1,1]<stdout>:Step #70  Loss: 0.318004
[1,0]<stdout>:Step #80  Loss: 0.169920
[1,1]<stdout>:Step #80  Loss: 0.272620
[1,0]<stdout>:Step #90  Loss: 0.139547
[1,1]<stdout>:Step #90  Loss: 0.140876
[1,0]<stdout>:Step #100 Loss: 0.127557
[1,1]<stdout>:Step #100 Loss: 0.239982
[1,1]<stdout>:Step #110 Loss: 0.091342
[1,0]<stdout>:Step #110 Loss: 0.200691
[1,0]<stdout>:Step #120 Loss: 0.066247
[1,1]<stdout>:Step #120 Loss: 0.118757
[1,0]<stdout>:Step #130 Loss: 0.177164
[1,1]<stdout>:Step #130 Loss: 0.121045
```

Figure 7-3: Distributed teaching ran through Jupyter

Also based on the output captured on Figure 7-3 it can be seen that both the Horovod master and worker are running, this we can see based on the numbers preceding each output line. [1,0] and [1,1] will denote the two different nodes, one being the Horovod master and the other being the worker node. Of course by adding more workers the teaching process can be even further distributed.

# 8 Further development

There are several areas which can provide further ways of improvement regarding the implementation making the project faster and more streamlined.
The first thing that should be changed is a transition to using an SSH jump server to configure the virtual machines without a public IP, instead of the current solution of running Ansible on the master server by starting a playbook on the local configuration machine. Through a jump server the local machine would be able to directly configure remote nodes, by using the master server as a gateway to them. It would be a better solution, since by configuring through a jump server significant amounts of time would be saved by not having to install Ansible on the master server as well as not having to copy playbooks to it either.
The most significant improvement that could be made however, is a transition from using Docker-compose to creating custom Docker images for each role based on the images used by the Docker-compose project. This would mean that instead of downloading the entire project, and with it every role's data, to each virtual machine, only data needed for setting up the given VM would be downloaded. This would significantly decrease the time it takes to set up the MooseFS storage cluster. This will most probably require the creation of custom Docker images that combine the MooseFS base image with the configurations that are made when the Docker-compose sets up the cluster. Although initially I wasn't sure whether this was possible, after doing these exact changes with the MooseFS client when integrating with Horovod, I can confidently say that it is without a doubt doable, although time consuming.
A now insignificant but overtime important change to make would be grouping multiple hosts under the same group in the master server's inventory file. In my implementation each host has it's own group inside the inventory file, which I explained in the Configuration with Ansible section. At the current scale of the project this is completely fine, since the configured storage cluster is very minimal in size, with only four chunkservers and two clients, however if we scale up the cluster and use tens, or even hundreds of virtual machines the inventory file will become needlessly cluttered, and difficult to read. For this reason in later iterations chunkservers should be put into the same group, and new groups should be created for MooseFS client nodes, one for Horovod workers and one for the Horovod master node.
Another area of improvement would be making the Terraform and Ansible configuration files more dynamic, since currently there are numerous things that are hard coded into

these file. For example, when the public IP addresses are assigned concrete addresses are defined in the Terraform file. This should be modified so that Terraform picks one of the free floating IPs instead of relying on the availability of that single address. If the defined IP is assigned elsewhere there will be no error thrown during the Terraform configuration, however Ansible will be unable to connect. Or in the case of the Horovod master, no public IP will be assigned to the VM making Jupyter notebook connections impossible.
Lastly, on the Horovod master it isn't possible to mount the MooseFS folder automatically like it is the worker. The cause of this can be seen in the entrypoint file which is part of the Docker image used for building this VM. For the worker the execution of this file originally ended with an infinite sleep cycle, technically "ending" operations this way. I replaced this with the mounting of MooseFS which similarly stops the further execution of the entrypoint file. However, the master takes a different route during execution, entering a Jupyter notebook process instead of stalling like the worker did. This is a problem since if MooseFS is mounted before the execution gets to start Jupyter then Jupyter will never be launched, conversely if Jupyter is started first, then MooseFS will never be mounted. The current workaround is that MooseFS is mounted manually in either a Jupyter notebook terminal, or by entering the container. This is not critical since these are two short commands which do not have to be modified, however for the sake of automation it should be fixed nevertheless.

# 9 Conclusion

In this thesis multiple distributed file systems have been considered to be used as the storage for a distributed deep learning system implemented with Horovod. After comparing five of the most well known distributed file systems MooseFS has been chosen as the most suitable one. Although MooseFS is not the most well known distributed file system it stands with its data security and availability. Its easy to set up nature also provides the flexibility of re-configuring machines to fill other roles in the infrastructure of the storage cluster, for example substituting the master server with the metalogger in case of a hardware failure.

Then a detailed system plan is presented describing how the chosen software will interact with each other and form a storage solution that deep learning needs to ensure the evasion of a storage bottleneck. The implementation of this system plan will provide these deep learning frameworks with a storage solution that is scalable, cost-effective, easy to set up and manage.

Using the chosen software and based on the system plan I then created an implementation on the OpenStack cloud platform using the resource of the ELKH Cloud platform. The finished implementation utilizes Terraform and Ansible to provide an easy to set up and manage solution for storage. It is also very scalable and limitlessly expandable thanks to it's integration of automation tools and usage of a distributed storage solution. Despite the initial setbacks regarding the MooseFS clusters integration with Horovod is completely functional, working as pictured in the system plan. Which I have demonstrated, by thoroughly testing the operational correctness of the file system and Horovod. Lastly, I have provided ways in which the implementation can be further improved both by extending and modifying it.

# 10 Összefoglalás

Szakdolgozatomban összehasonlításra került az öt legismertebb elosztott fájl rendszer. Ezek közül a választás a MooseFS-re esett mint a legalkalmasabb fájl rendszer egy Horovod elosztott mélytanulási rendszer tárhelyeként. Bár a MooseFS nem a legelterjedtebb elosztott fájl rendszer, adat biztonsága és elérhetősége alkalmasabbá teszi társainál. Konfigurációjának egyszerűsége és a már valamilyen szerepkört betöltő gépek újrakonfigurálásának lehetősége szintén e mellett a fájl rendszer mellett szól.
A megfelelő elosztott fájl rendszer kiválasztása után egy részletes rendszerterv került elkészítésre. Ez a terv leírja, hogy a választott szoftver eszközök miként fognak egymással együttműködni egy olyan tárhely megoldás elkészítésében, ami képes lesz áthágni a mély tanulási rendszerek tárhely akadályait. A rendszer terv gyakorlati kivitelezése egy bővíthető, költséghatékony és könnyen felállítható rendszerrel fogja támogatni a mély tanulást.
Ezt követően a gyakorlati megvalósítást az OpenStack felhő platformon készítettem el az ELKH Cloud erőforrásait használva. A kész implementáció Terraformot és Ansiblet használva készült el, egy könnyen kezelhető és felállítható tárhely képében. Ezek mellett a rendszer jól skálázható és a limit nélkül bővíthető, automatizálási szoftverek és elosztott fájl rendszer használatának köszönhetően. A MooseFS és Horovod integráció kezdeti nehézségei ellenére a kész implementáció teljesen működőképes és teljeskörűen megfelel a rendszertervben leírtaknak. Mind a fájl rendszer, mind a Horovod megfelelő működését részletes teszteléssel demonstráltam. Ezenkívűl összegyújtőttem olyan területeket, ahol az implementáció további fejlesztéssel még hatékonyabbá tehető.

# Table of Figures

# References


[1] "Horovod documentation. https://user-images.githubusercontent.com/16640218/38965607-bf5c46ca-4332-11e8-895a-b9c137e86013.png.." Last visited: 2021.12.10.

[2] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[3] "Neural network. https://1.cms.s81c.com/sites/default/files/2021-01-06/ICLH_Diagram_Batch_01_03-DeepNeuralNetwork-WHITEBG.png.." Last visited: 2021.12.10.

[4] "Tensorflow documentation. https://searchdatamanagement.techtarget.com/definition/TensorFlow.." Last visited: 2021.12.10.

[5] "Horovod. https://horovod.ai/.." Last visited: 2021.12.10.

[6] "Horovod cluster documentation. https://git.sztaki.hu/science-cloud/reference-architectures/horovod/-/blob/master/README.md.." Last visited: 2021.12.10.

[7] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam, "Automated testing of chef automation scripts," in *Proceedings Demo amp; Poster Track of ACM/IFIP/USENIX International Middleware Conference*, MiddlewareDPT '13, (New York, NY, USA), Association for Computing Machinery, 2013.

[8] J. Blomer, "A survey on distributed file system technology," vol. 608, p. 012039, may 2015.

[9] D. Anthony, B. Vaibhav, and S. Karan, *Learning Ceph: Second Edition*. Livery Place, 35 Livery Street, Birmingham, B3 2PB, UK: Packt Publishing Ltd, 2017.

[10] "Apache Cloudstack. https://cloudstack.apache.org/.." Last visited: 2021.12.10.

[11] "OpenStack. https://www.openstack.org/.." Last visited: 2021.12.10.

[12] "OpenNebula. https://opennebula.io/.." Last visited: 2021.12.10.

[13] A. D'atri, V. Bhembre, and K. Singh, *Learning Ceph: Unifed, scalable, and reliable open source storage solution*. Packt Publishing Ltd, 2017.

[14] "Ceph Documentation. https://docs.ceph.com/en/pacific/.." Last visited: 2021.12.10.

[15] "Ceph cluster. https://miro.medium.com/max/2400/1*9q3aVfK8fvqAWa-iZK3jtg.png.." Last visited: 2021.12.10.

[16] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis, "File systems unfit as distributed storage backends: Lessons from 10 years of ceph evolution," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, (New York, NY, USA), p. 353–369, Association for Computing Machinery, 2019.

[17] D. Borthakur *et al.*, "Hdfs architecture guide," *Hadoop apache project*, vol. 53, no. 1-13, p. 2, 2008.

[18] "HDFS cluster. https://www.whizlabs.com/blog/wp-content/uploads/2018/08/hdfs-architecture.jpg.." Last visited: 2021.12.10.

[19] S. Niazi, "Size matters: Improving the performance of small files in hadoop," 2018.

[20] E. B. Boyer, M. C. Broomfield, and T. A. Perrotti, "Glusterfs one storage server to rule them all," 7 2012.

[21] G. Donvito, G. Marzulli, and D. Diacono, "Testing of several distributed file-systems (HDFS, ceph and GlusterFS) for supporting the HEP experiments analysis," vol. 513, p. 042014, jun 2014.

[22] "GlusterFS cluster. https://docs.ganeti.org/docs/ganeti/2.12/html/_images/graphviz-afe6b4e16c8604e4a80cc25f8a95dd0a20d6246b.png." Last visited: 2021.12.10.

[23] C. Alexander, "Improving the openafs security model without client-side changes," 2014.

[24] "OpenAFS documentation. http://workshop.openafs.org/afsbpw08/talks/wed_1/OpenAFS_and_the_Dawn_of_a_New_Era.pdf.." Last visited: 2021.12.10.

[25] S. Hagen and C. Eriksen, “Comparison of nfs, samba and afs,” 2005.

[26] B. Boris, “Comparison and evaluation of nfsv3, nfsv4, and afs distributed filesystems,” 2001.

[27] “MooseFS cluster. `http://contrib.meharwal.com/home/moosefs`.” Last visited: 2021.12.10.

[28] R. K. Piotr and K.-Z. Agata, “Moosefs 3.0 user’s manual,” 2017.

[29] S. Bai and H. Wu, “The performance study on several distributed file systems,” in *2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pp. 226–229, 2011.

[30] “ELKH Cloud. `https://science-cloud.hu/en/elkh-cloud`.” Last visited: 2021.05.15.