Seminar Report

Performance Evaluation of BeeGFS Under Node Failure Condition

Surendhar Muthukumar

MatrNr: 28131609

Supervisor: Dr. Freja Nordsiek

Georg-August-Universität Göttingen
Institute of Computer Science

March 31, 2024

Abstract

BeeGFS is a leading softwware-based parallel file system that is mostly developed with concentration on high performance computing. Fundamentally, beegfs combines multiple storage nodes to create a highly scalable file system that can be mounted and used by the clients in a parallel access manner. Striping of file contents, distribution of file system metadata over multiple nodes (typically servers), data mirroring, flexibility over kernel distribution, automatic switching mechanism between RDMA and TCP/IP network connections are some of the key features of BeeGFS that makes it more efficient and fault tolerant. Since BeeGFS was designed in considerations for performance and parallel file access, the following work is concentrated towards testing the performance of a simple Beegfs setup under node failure conditions to analyse the degradation of the performance in this situation. Distinctive to other performance evaluations on file systems under perfect working/ stable condition, this study concentrates on the performance of system under node failure conditions. Artificially induced node failure is used here on the experimental setup to simulate node crash. The performance tests run by open-source benchmarking software elbencho reveal the extent of performance degradation under study scenario. The experimental setup, tuning conditions, benchmarking analysis will be explained in detail in this report.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☐ During brainstorming
- ☑ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☐ For the development of software source texts
- ☑ For optimizing or restructuring software source texts
- ☐ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Abbreviations

PFS	Parallel File System
IO	Input Output
HPC	High-Performance Computing
VM	Virtual Machine
RDMA	Remote Direct Memory Access
POSIX	Portable Operating System Interface
TCP/IP	Transmission Control Protocol / Internet Protocol
GWDG	Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen
RHEL	Red-Hat Enterprise Linux

1 Introduction

Parallel file system enables simultaneous access to the underlying storage nodes by multiple clients at the same time. Clients can store and access data across the network connected storage nodes by using parallel IO paths [6] [12]. High Performance Computing utilizes PFS's parallel data processing property to store and retrieve data for processing. Unlike supercomputers, HPC facilities rely on distributed storage and resources for processing. Latency and uninterrupted data access from the storage to compute resources are crucial. The underlying simultaneous multiple access property of PFS correlates to HPC storage needs and hence PFS are most popular to be used for HPC. Multiple software based PFS are available with enhanced features in addition to above mentioned basic functionality.

BeeGFS is a leading open-source software based parallel cluster file system developed initially for High Performance Computing (HPC) [7]. Due to its benefits of scalability, flexibility and availability it is used in almost many fields.. BeeGFS notably distributes the data across multiple configured servers, enabling capacity and performance aggregation. Data being read and written to the servers are split into stripes to reduce the network load, reducing the possibility of network bottlenecks and bandwidth issues in general. BeeGFS supports in-built mirroring to create buddy groups of the storage target to enable data replication across selected targets of the servers for preventing data loss during node crashes, also enabling load balancing across the servers. User data spread across multiple servers and disks are aggregated into a single namespace for shared access.

BeeGFS can be applied on any commodity off the shelf hardware to create parallel file system as it is POSIX compliant. Generally HPC facilities use high bandwidth RDMA-based interconnects between the servers/nodes for faster inter node communications. RDMA-based and TCP/IP based client server communications are available to be used through BeeGFS. Separation of metadata and the user data enables faster look up of data while enabling storage level abstraction of metadata and the actual chunks of user data.

1.1 Scalability, Availability and Performance

BeeGFS is developed with the main focus on providing highly available, scalable and performance centric PFS. While users generally face bottlenecks on performance and availability due to single server-single interconnect based file access, BeeGFS allows combining multiple storage servers to configure and provide a better file system. In BeeGFS, actual user data and metadata are separately stored in different nodes and managed by different services. Data chunks are stored as stripes of specific size that shall be defined by the system administrator according to the available interconnect capacities or left to be the default 512KB. Striping of data allows for easy distribution of the data throughout the available servers while reducing the network traffic.

While accessing the stored data, BeeGFS clients directly contact the storage servers without any intermediate routing service reducing the network demands. Information on which server to be contacted is stored separately in the metadata servers, once the metadata server provides the client with the information on storage server to be contacted, the communication between the metadata server and client is disabled. Further communications are done between the storage and the client. Performance is enhanced as the clients can jump between the servers containing the data of interest, reducing the network traffic to single storage server, enabling the load balance. Data striping allows for creating data packets of desired size, enabling control over the size of data transmitted over the network which can aid the performance. Additionally, buddy groups of target data, serve as backup servers under node failure conditions and also aid for parallel file access.

1.2 Study Goal

Previously performed studies on the IO performance of the BeeGFS based file systems were concentrated on testing out performance factor of the same for specific application, specific resources and use cases. The studies were highly problem and product oriented [2] [5] [3]. There is a lack of studies that discuss about the performance of the system under node failure condition where promises on high availability are made. Hence this study is primarily designed to analyse the performance of the system under node failure conditions through simulation of node failure on a study setup.

2 Architecture

BeeGFS architecture is composed of services. Tuning of the provided services offers custom benefits during deployment..Understanding the individual services and tuning options is essential to setup an efficient user space file system to satisfy the desired needs. Choice of mapping underlying storage hardwares, resources to BeeGFS services can have varied effect on availability and performance. Tuning and configurational features available for different services provide an additional layer of flexibility on optimization of performance, scalability, availability, latency on top of the preferred mapping.

2.1 BeeGFS services

BeeGFS based parallel file system has 4 main architectural services that help to run, maintain and keep system function, they are:

- Management service -Registry and manager of other running beegfs services
- Metadata service - Stores access permission and striping information
- Storage service - Stores the user file contents
- Client service - Mount the file system to access the data [7]

In addition to the above key components, there is additional beegfs-helperd service and beegfs-utils, which are additional packages that provide command line tools, helpers. Beegfs-helperd service is run on the client nodes for providing logging and DNS lookup functionality. Beegfs-utils provides the command line tools that reports the statistics and perform administrative tasks [1].

Figure.1 depicts a simple file system where the services are running on individual servers. Different services communicate internally to keep the system running and satisfy the client requests. Architectural configuration of a custom BeeGFS file system is highly flexible, usually according to the underlying need. Services shall be configured to run on dedicated nodes or combined to run on same node.

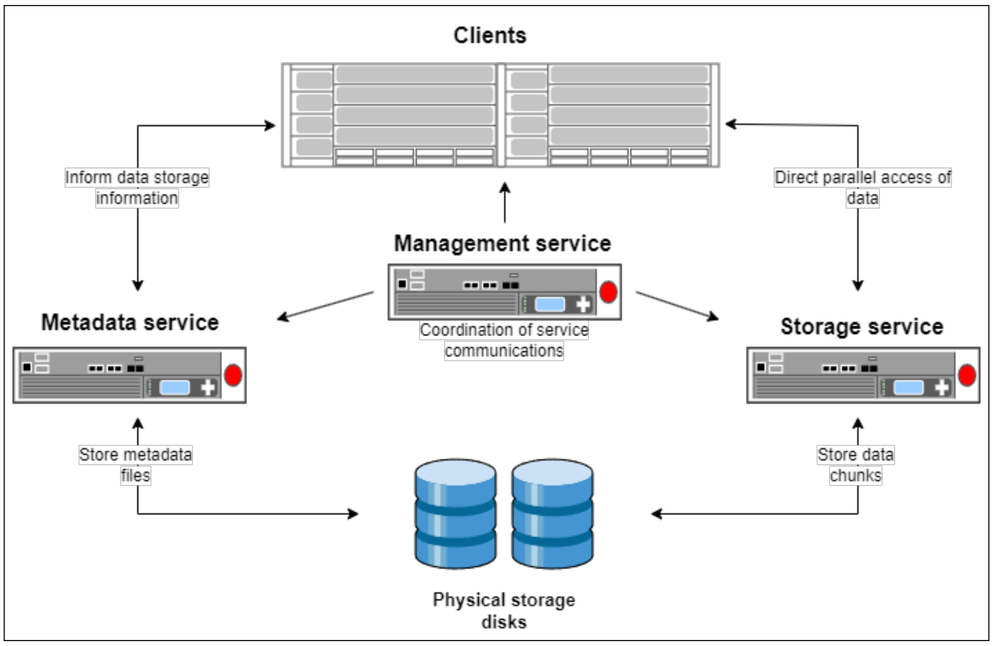


Figure 1: BeeGFS Architecture Overview [10]

2.1.1 Management service

Management service orchestrates the BeeGFS based file system. Management service can communicate with all other running services and check their individual state. In addition to orchestration, management service acts as the registry that stores the configured services along with their state. It is a gateway point where all the services meet. Being light weight management service does not need dedicated hardware to be instantiated, but the choice of dedicated resource is use case and user dependent. Since this service holds track of the other services, in a file system the management service has to be configured first.

2.1.2 Metadata service

Metadata service holds information about the data strips of the files. Important data information includes directory information, file and directory ownership and location of the file in underlying physical storage. Information about the location of a file is provided to the client through the metadata service when the client tries to access any file. After receiving the information, client connects to respective storage target and further communication to metadata service is done only while the client wants to access another file or data. Metadata distribution can be enabled by setting up multiple metadata service nodes.

Each configured service shall hold their exclusive part of the namespace. Metadata services have individually atmost one metadata storage target to store the file information. For efficiency and performance consideration, each user file will have one metadata file of ext4 file format in the target. Unlike a single metadata database, BeeGFS metadata service holds metadata in ext4 file system which is based on RAID1 or RAID10.

2.1.3 Storage service

Actual user file chunks are stored in the storage targets governed by storage services. Storage services are expandable like the metadata services. Unlike metadata service, storage service can have multiple storage targets. Storage node tuning provides options to optimize the traffic on storage targets. BeeGFS storage services are cache enabled and by default BeeGFS uses all the available RAM of the storage node for caching, except those used for active process. Cache also enables aggregation of multiple small IO requests to a bulk process before performing the read/write operation on the disks. Cache enabled storage nodes benefit from better interconnects used for network. Cache benefit can be utilized with a network interface with higher or equal data transfer speeds than RAM architecture.

BeeGFS picks storage targets in a random manner. This method has been observed to perform well in multi-user environment where the file system would handle a mix of large and small files from different clients concurrently. But this can give rise to a problem of depleting particular storage target space soon when by random most of the time a single disk is being chosen. To prevent this, BeeGFS has 3 different labels for the free storage capacity on the targets: normal, low and emergency. The target chooser program running on the metadata service chooses the target with the label normal. The low and emergency labels are assigned for the nodes with very less free space, these free space threshold can also be custom defined in configuration. Target chooser would avoid these critical capacity labelled services until all other services have also entered the critical label state. Additional changes to storage service can be done by modifying the configuration file : **beegfs-storage.conf.**

2.1.4 Client service

BeeGFS client service must be implemented on the client nodes in order to mount and utilize the shared file system directory. Client service must run on all the client nodes that wish to access the shared file system. BeeGFS client service comes with linux kernel module, that provides interface between BeeGFS client software and the client node operating system's native file system. This module handles file system operations and the communication between client and the user space. Additional userspace helper daemon called beegfs-helperd service is utilized by client for DNS lookups and log file maintenance. This helper service is not incorporated into the client service by default, hence this has to be additionally configured in the client nodes.

When the client node is booted and the service is running on the respective node, it mounts the file system directory defined in the mount configuration file **beegfs-mounts.conf**. When client wants to read/write to a file, it communicates with the metadata service first, orchestrated by the management service. After receiving the file specific metadata information, the client directly communicates to the storage service to access the data stored in the disks. Pipe-lining of service through many intermediate systems is resolved in BeeGFS based user spaces. Further changes on behaviour of client service and customization can be done through modifying the configuration file : **beegfs-client.conf**.

2.2 Tuning and configuration features

Being a high performance file system, BeeGFS provides features for optimization of performance and availability. Multiple tuning and configuration features offer customization over default service definitions, that aid in performance and resource usage optimization. Users generally fine tune the services and configure the services with provided feature enhancements like file striping, mirroring, quota allocation, metadata distribution, storage pool configurations [1]. In this study, custom decisions have been performed on file striping and mirroring to customize performance, availability.

2.2.1 File striping

Parallel file system utilize the file striping functionality to strip the user file into multiple chunks and store it in different servers than putting one file in one server. Distribution of the file chunks enable higher bandwidth, as more users could access the file chunks from different storage servers. Aiding to the bandwidth, the network load sent to a server could also be reduced when a certain file of interest is fetched by multiple users simultaneously, hence the load balancing is partly achieved through striping. In BeeGFS, custom stripe pattern can be set to individual directories and even individual files such that the chunk size can be customized and this might allow to adapt to the network resources available. Stripe pattern combined with buddy mirroring (explained in section 2.2.2) enables enhanced replication of user data with improved availability.

2.2.2 Buddy mirroring

Mirroring is the replication of logical disk volumes into separate configured storage devices for better data availability and reduce loss of storage data under node failure [11]. In BeeGFS mirroring is not equivalent to backup, as the data replication is generally not time stamped snapshots of data. The data is simply replicated between the configured nodes, hence if data on one server node is modified , it is automatically reciprocated into the other. Mirror buddy groups should be defined in BeeGFS to use mirroring, this can be done either manually or automatically. In manual method individual storage target IDs are mentioned to create a group. In the automatic method, BeeGFS assign random targets to a group. Mirror buddy groups are identified through numeric IDs. The buddy group consists of a primary and a secondary target, where primary target is used for data IO operations by default. Data modifications are replicated into the secondary target. When the primary target enters into failure mode, the management service identifies the failure and makes the secondary target as primary until the actual primary target state is restored. After the failed node is restored, re-synchronization of data happens between the nodes.

3 Performance analysis system setup

This study involves setting up a small scale BeeGFS based file system to perform benchmarking under artificially simulated node failure condition. Virtual machines instantiated through OpenStack cloud computing platform through GWDG Academic service is used as nodes for setting up and running the individual beegfs-services.

3.1 System Architecture

The small scale system used for this study involves 1 management service, 2 metadata services (where one of the metadata service run on the same node as management), 4 storage services with 1 target each, 3 client services. One of the client service is run on the management service node to facilitate visualization of the system statistics and connections through the client features. The OpenStack virtual machines are used as hosts for the services, which run CentOS 8. All the VMs are set to have 2 GB of RAM and 20 GB SSD storage. Hence as a whole 8 VMs were used to setup the file system.

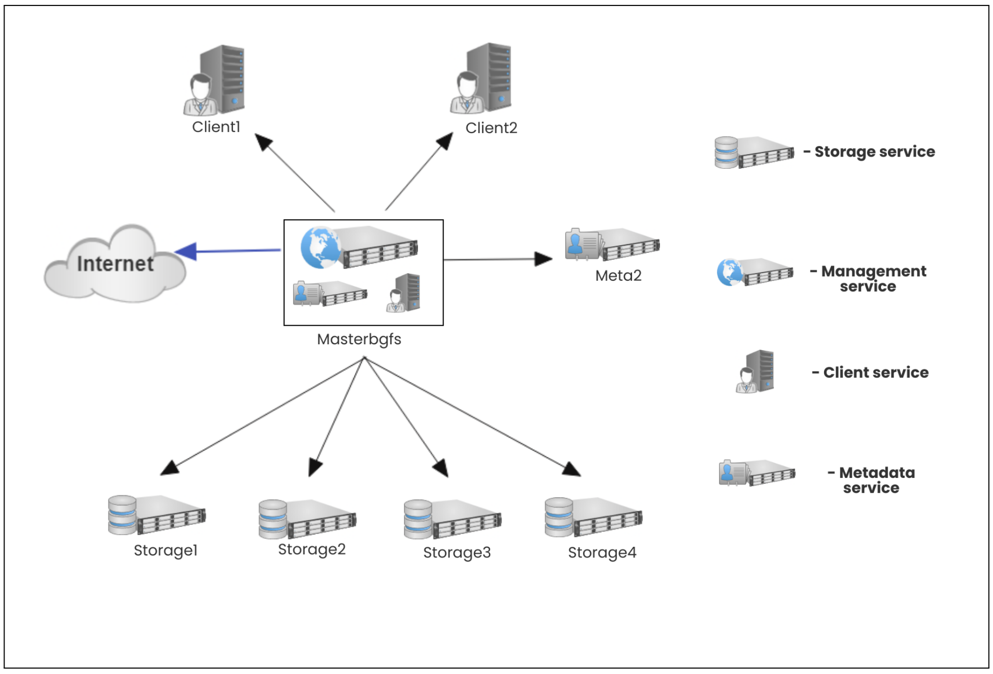


Figure 2: Custom BeeGFS system architecture for Benchmark study

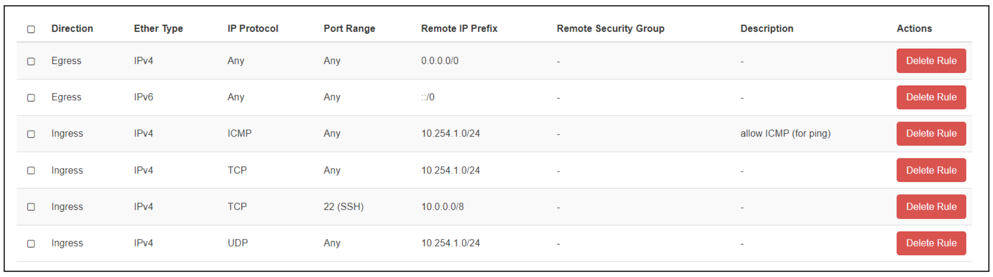
Figure 2 shows the underlying system architecture used for performance analysis. Individual hosts are instantiated as VMs. The VM: Masterbgfs is the central manager host that runs the management service, along with the first metadata service and a client service. A second metadata service is setup in different host named Meta2. Likewise the rest 2 client service and 4 storage services are setup in individual VM. Storage, metadata and client services shall communicate to the management service through underlying ethernet network. Additionally, management service can communicate to the internet, which acts as gateway for all other nodes to access internet.

3.2 Host SSH setup

Individual nodes must be able to communicate with other nodes that run the services to start booting and perform tasks. Hence an internal network must to configured. This internal network setup and the communication is orchestrated by the management service in BeeGFS service level, but the node level communication should be handled manually so that management host can reach other hosts without any interruption and vice versa. Individual service nodes can be reached from the management node through ssh. For facilitating ssh connection, a **ssh-config** file is created in the management host. The config file with IP addresses of the other service node with their respective key is created, such that when ssh-ed into the other host of interest, this config file is read and the system knows which key shall be used for the mentioned host [8]. Additionally, each service host is assigned a name by modifying the **\etc\hosts** file to skip the use of entering the IP address during every ssh attempt.

Management host in our study system is connected to the internet for external communication purposes. By default openstack instances have port security enabled, hence incoming communication requests from external network is blocked for security reasons.

This port security blocks all the communication requests from other service hosts to the management. Hence the port security in the management node must be configured in a way that the system only communicates to specified external IP addresses (in our case the IP addresses of service hosts). Since the service hosts belong to a subnet, the subnet shall be mentioned as trusted external hosts to be allowed by the management host's port security for incoming communications. Defined rules for the management host can be seen in figure 3, where the incoming requests are allowed only from a trusted subnet with address belonging to 10.254.1.0/24. Since the other service nodes are not connected to internet, the port security is disabled and by default only management host can communicate.



☐	Direction	Ether Type	IP Protocol	Port Range	Remote IP Prefix	Remote Security Group	Description	Actions
☐	Egress	IPv4	Any	Any	0.0.0.0/0	-	-	Delete Rule
☐	Egress	IPv6	Any	Any	::/0	-	-	Delete Rule
☐	Ingress	IPv4	ICMP	Any	10.254.1.0/24	-	allow ICMP (for ping)	Delete Rule
☐	Ingress	IPv4	TCP	Any	10.254.1.0/24	-	-	Delete Rule
☐	Ingress	IPv4	TCP	22 (SSH)	10.0.0.0/8	-	-	Delete Rule
☐	Ingress	IPv4	UDP	Any	10.254.1.0/24	-	-	Delete Rule

Figure 3: Openstack port security rules for Instances

3.3 Node setup

As the first step after the physical host instance setup, BeeGFS services are installed in every individual host to start respective services. The hosts use RHEL 8 based distribution. Individual nodes are accessed through SSH. BeeGFS package for the respective linux distribution is downloaded from the official BeeGFS package repository. For RHEL based system below command is used in the command line interface to download the package.

```
\$ wget -O /etc/yum.repos.d/beegfs_rhel8.repo
    https://www.beegfs.io/release/beegfs_7.4.2/dists/beegfs-rhel8.repo
```

3.3.1 Management node setup

Management service is installed in the management node using the standard yum installation command **yum install beegfs-mgmtd**. The management service obtains the local host name **masterbgfs** and the Ip_Address of the management host is added to the \etc\hosts list of all other nodes. Since our management node also hosts one metadata and a client service, these services are also installed in the same way as before **yum install beegfs-meta beegfs-client**. beegfs-helperd and beegfs-utils service are installed for assistance on viewing system statistics. When individual services are started using the command **systemctl start**, the system reads the respective **.conf** file to perform service specific configuration tasks that are predefined by BeeGFS based on the parameter values in the configuration file. Customization of service startup is done through configuration file modification. Before staring the management service, the storage location for node information should be defined using below command:

```
\$ /opt/beegfs/sbin/beegfs-setup-mgmtd -p /data/beegfs/beegfs_mgmtd
```

Management service needs to be setup and started first as it is the primary registry service. Once the service is successfully started, a connection authentication file is generated through the following command.

```
\$ dd if=/dev/random of=/etc/beegfs/connauthfile bs=128 count=1
```

This connection authentication file: **connauthfile** is used like a shared secret between the nodes/hosts that run the services to authenticate itself to other nodes in the internal BeeGFS network. Connection authentication file can be shared between the nodes using **scp**, secure copying command. All other nodes are instantiated and configured in a way that they are accessible through ssh from the management node for easy maintenance and data transfer between the nodes. The metadata and client service in the management host is setup as explained in the sessions below.

3.3.2 Metadata service setup

Metadata service is installed in the node. Storage target must be defined to store metadata in this node. After service installation, storage target is setup using below command:

```
\$ /opt/beegfs/sbin/beegfs-setup-meta -p /data/beegfs_meta -s 2 -m
   masterbgfs
```

Once the target is successfully created and the connAuthFile in the .conf file has been updated same as the management node, the service shall be started through **systemctl start beegfs-meta**.

3.3.3 Storage service setup

Like metadata node storage service needs storage target to store user file chunks. More than one storage target shall be configured in a storage host. Here only one such storage target is used. After service installation, storage target is setup using below command:

```
\$ /opt/beegfs/sbin/beegfs-setup-storage -p /mnt/myraid1/beegfs_storage
   -s Node_Id -i Storage_target_Id -m masterbgfs
```

Node_Id and Storage_target_Id can be any unique numerical value for later reference. After target creation, connAuthFile is defined as same as management node and the service is started through **systemctl start beegfs-storage**

3.3.4 Client service setup

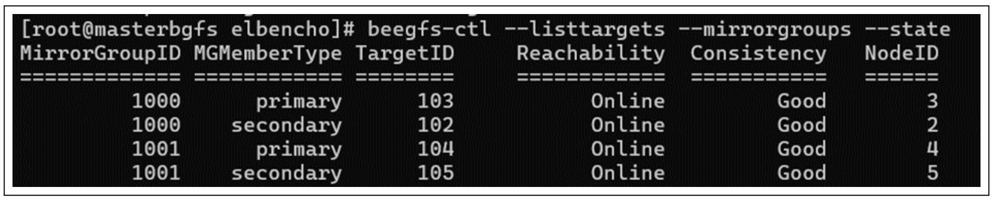
In addition to client service, client hosts also need helperd service, hence both are installed in the host. Since BeeGFS-client is implemented as linux kernel module, linux-headers and linux development tools are installed in the client host based on RHEL8. After installing the dependencies, client service is informed about the management service host through below command:

```
$ /opt/beegfs/sbin/beegfs-setup-client -m masterbgfs
```

The connAuthFile in the beegfs-client.conf file is updated. Client mount directory is defined in separate configuration file for mounts **beegfs-mounts.conf**. This mount configuration file will be used by client during service startup. By default, the mount directory is /mnt/beegfs and it remains the same in our study. After updating the configuration files, helperd and client services are started using **systemctl start beegfs-helperd** and **systemctl start beegfs-client**.

3.3.5 Configuring buddy mirrorgroups

General use case of BeeGFS is HPC and other large scale data processing engines. Hence there is a need for uninterrupted data access with less risk of loosing data due to hardware and network failures. Risk on loss of user data chunks is reduced already through the underlying RAID6 based file system architecture. Now, risk on accessibility issues of data due to any failures can also be reduced further using mirroring. Storage and metadata buddy mirror groups are configured such that when one of the targets belonging to the buddy groups enters into a node failure, the other target is used for data input/output operations.



```
[root@masterbgfs elbencho]# beegfs-ctl --listtargets --mirrorgroups --state
MirrorGroupID MGMemberType TargetID     Reachability  Consistency   NodeID
============= ============ ========     ============  ===========   ======
         1000      primary      103           Online         Good        3
         1000    secondary      102           Online         Good        2
         1001      primary      104           Online         Good        4
         1001    secondary      105           Online         Good        5
```

Figure 4: Storage targets buddy group details

Buddy groups are configured manually in our study using the node IDs. Data IO operations are performed on primary target in normal situations. Figure 4 shows that there are 2 buddy groups with buddy groupIDs 1000 and 1001 configured with 2 storage targets each. Reachability field value being online states that the target nodes are reachable and are in active state. Setting up buddy mirroring is import for multi server data access with data availability at node failure conditions for high performance.

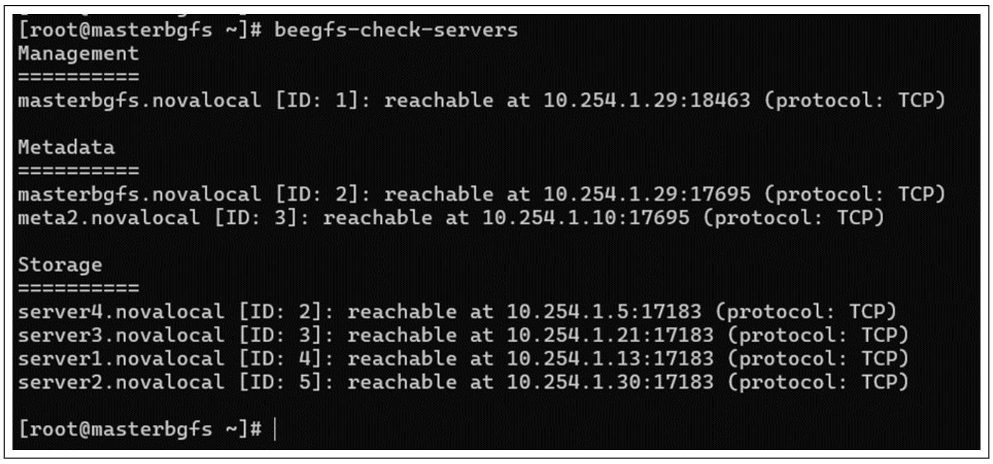
```
\$ beegfs-ctl --addmirrorgroup --nodetype=storage --primary=1
   --secondary=2 --groupid=100
```

3.3.6 Directory stripe pattern configuration

Striping configures the individual file chunk size stored in the storage targets. To utilize the buddy mirroring feature in specific directory, striping pattern of the directory must be set to **buddymirror**. The default stripe pattern is RAID0, where the data written to the server is not replicated to the secondary target. When the buddymirror pattern is set, the data and operations are replicated between the buddy targets.

3.3.7 System connectivity check

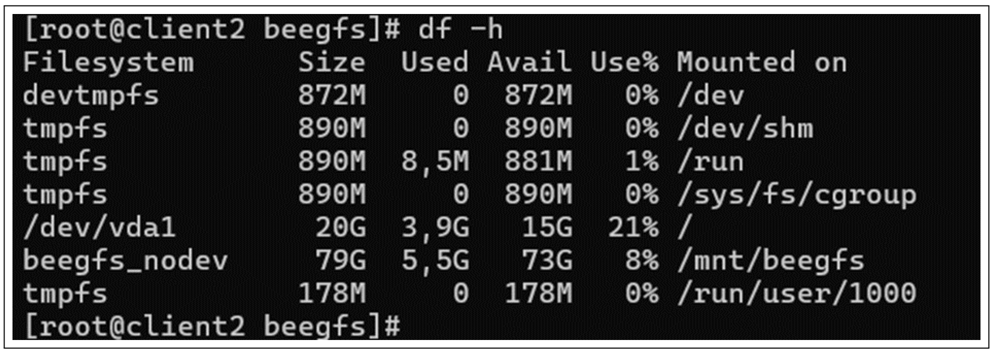
BeeGFS services configured on different physical servers/nodes use network interconnects to communicate with each other. This network can either be general ethernet hardware connection or RDMA based network hardwares (like Infiniband, Omnipath, RoCE) for faster inter server communication. Servers in industrial level file systems are connected using RDMA-capable network hardwares. It is essential to verify if the servers that are connected using these network hardwares use respective transfer protocol to ensure the internal connectivity between servers. However in this study, no RDMA based transfer is used, the network between the service hosts is established through ethernet. But the setup connection check can provide important information on the routes to the servers, node reachability and service configuration stats. Possible connectivity to the servers involved in the file system can be verified using the command : **beegfs-check-servers**.



```
[root@masterbgfs ~]# beegfs-check-servers
Management
==========
masterbgfs.novalocal [ID: 1]: reachable at 10.254.1.29:18463 (protocol: TCP)

Metadata
==========
masterbgfs.novalocal [ID: 2]: reachable at 10.254.1.29:17695 (protocol: TCP)
meta2.novalocal [ID: 3]: reachable at 10.254.1.10:17695 (protocol: TCP)

Storage
==========
server4.novalocal [ID: 2]: reachable at 10.254.1.5:17183 (protocol: TCP)
server3.novalocal [ID: 3]: reachable at 10.254.1.21:17183 (protocol: TCP)
server1.novalocal [ID: 4]: reachable at 10.254.1.13:17183 (protocol: TCP)
server2.novalocal [ID: 5]: reachable at 10.254.1.30:17183 (protocol: TCP)

[root@masterbgfs ~]# |
```

Figure 5: Server connectivity status of the BeeGFS file system used for performance study

Figure 5 states the connectivity details of the servers involved in the system used for the study. The storage, metadata and management hosts are connected through ethernet and hence they follow TCP protocol, which was the desired for our network connection. Hence this states that the system communication if stable and usable.



```
[root@client2 beegfs]# df -h
Filesystem      Size  Used Avail Use% Mounted on
devtmpfs        872M     0  872M   0% /dev
tmpfs           890M     0  890M   0% /dev/shm
tmpfs           890M  8,5M  881M   1% /run
tmpfs           890M     0  890M   0% /sys/fs/cgroup
/dev/vda1        20G  3,9G   15G  21% /
beegfs_nodev     79G  5,5G   73G   8% /mnt/beegfs
tmpfs           178M     0  178M   0% /run/user/1000
[root@client2 beegfs]#
```

Figure 6: Storage targets and Mount directory statistics

Following the network connection check, the beegfs shared mount directory details can be obtained through the **beegfs-df** command. Figure 6 shows the aggregated storage target and metadata target statistics. Since 4 storage targets of 20GB each are used for our system, the aggregated shared directory has storage space of around 79GB, the difference disk space is used for service related scripts and log function. In the figure it can be seen obvious that the command is ran on the client host instead of the management node. These statistics check commands can be run on any configured client host belonging to the system. The filesystem named beegfs_nodev represents the shared file system mounted on the mount directory : **/mnt/beegfs**.

4 Benchmarking the Filesystem

As the main goal of the study was to analyse the performance of BeeGFS based user space file system under node failure conditions, benchmarking is done to obtain the statistics on system performance and availability through IO benchmarking. An open source benchmarking software : **elbencho** is used here. Elbencho is designed to benchmark PFS and AI engines. The custom created filesystem is artificially induced with a node failure and followed by benchmarking to achieve the study goal.

4.1 Node failure simulation

Node failure can be simulated in multiple ways, through turning off the node physically or degrading the server through dumping huge volumes of data than the node could handle or by software induced failure. In our study, the node failure is induced by physically shutting down the node/service. The host instance can be shutdown using the openstack instance management dashboard, which is equivalent to turning off a server. Or the service running in the host shall be stopped, so that the management/client/metadata service cannot communicate to this storage service and hence marks this storage target as degraded.

4.2 Elbencho benchmarking

Elbencho is a open source, filesystem agnostic benchmarking tool for distributed storage in PFS and AI engines. This tool is used in the study because of its filesystem agnostic nature with ability to perform benchmarking on directory level [4]. Directory level benchmarking is an advantage for our study as 2 directories with and without mirroring enabled is used to distinguish the performance of parallel access under non-mirrored conditions as well.

Elbencho needs C++ compatible local compiler to compile and execute the tool in the local machine without any need for external communication to internet. The dependencies can be installed in RHEL based system using the below command mentioned in the README section of elbencho git repository.

```
sudo yum install boost-devel gcc-c++ git libaio-devel make ncurses-devel
    numactl-devel rpm-build
```

With the dependencies installed, the tool could directly be cloned into the host system from the respective git repository **git clone https://github.com/breuner/elbencho.git**. Program files shall be copied into the ~/**elbencho** directory, and compiled using the make command, that compiles and executes the tool installation. Benchmarking can be now performed on desired directory using the **bin/elbencho** command, which indicates the system that the benchmarking should be performed with required parameters passed.

4.3 Benchmarking parameters

Elbencho can perform different benchmarking like IO throughput, network benchmarking, multi host/client involved distributed system benchmarking. For this study, multi client IO benchmarking is used for analysing the distributed storage access. All available benchmarking test parameters can be obtained by analysing the output of the elbencho help command **bin/elbencho –help-all**. The parameters for the test can be passed to the underlying benchmarking program in 2 different ways. Parameters can either be passed as command line arguments or through custom configuration file. 2 configuration files, **writetest.elbencho** and **readtest.elbencho** is used in this study. The important test parameters are:

- **threads** - number of parallel threads used for read/write operation
- **hosts** - comma separated list of hosts/clients in multi-host file systems
- **dirs** - number of directories to be used
- **files** - number of files per directories
- **read** - perform data read operation
- **write** - perform data write operation
- **rand** - write/read at random offsets [4] [9]

4.4 Performance evaluation

File system performance is measure under 3 different system behaviour as follows:

- IO performance difference between mirrored and non-mirrored directory
- IO performance under simulated node failure condition
- IO performance during node re-synchronization

4.4.1 Performance difference on mirroring

/mnt/beegfs is the distributed file system directory mount by clients to access the PFS user space. As mentioned in section 2.2.2, introducing mirroring enhances the availability of the data which gives performance benefits under node failure conditions. In our experimental setup two sub-directories to this mount directory are created to understand and analyse the performance difference under mirroring and non mirroring condition. The key point to note is mirroring combined with file pattern set to mirroring is the key aspect that keeps the system available under node failure condition, hence for our main study on failure condition performance, configuring mirroring is essential. **/mnt/beegfs/mirrordir** and **/mnt/beegfs/nomirrordir** are the 2 sub-directories created. The former directory is configured to use mirroring through stripe pattern configuration. Creating buddy mirror groups of storage targets alone doesn't trigger mirroring, it should be enabled through stripe pattern. Mirroring can be enabled for a directory using the below beegfs-ctl command.

```
beegfs-ctl --setpattern --numtargets=2 --chunksize=1M
   --pattern=buddymirror /mnt/beegfs/mirrordir
```

The above command sets the stripe pattern for the mirroring enabled directory to buddy mirror, which informs the management service that this directory is mirroring enabled and hence are the files and sub-directories that shall be created inside it. For any operations on this directory, 2 storage targets among the available 4 shall be used with file chunks being stored into the buddy targets instead of being stored into a single target (this doesn't mean all the chunks of a single file will be stored in same target, the chunks are distributed but there won't be any replica of the chunks created in another directory). For the **nomirrordir** the –pattern is set to **RAID0**, so that mirroring is not enabled. To analyse the I/O performance, write and read benchmarks are executed on the 2 defined directories through benchmark configuration files mentioned in Section 4.3

Mirroring status	Operation Type	File size (MiB)	Throughput (MiB/s)	Avg. Time (seconds)	Avg. Latency (seconds)
Mirroring enabled	Write	3720	22	160.58	4.94
Mirroring disabled	Write	3720	33	106.64	3.32
Mirroring enabled	Read	3000	43	70.69	2.49
Mirroring disabled	Read	3000	44	68.8	2.45

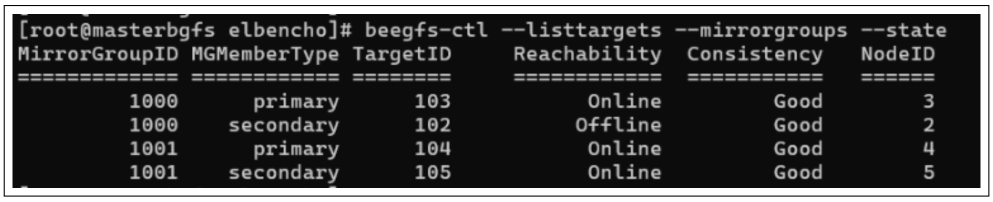
Table 1: Benchmarking statistics on mirroring

The benchmarking statistics of the mirroring comparison is provided in table 1. Read and write benchmarking has been performed on the 2 defined directories : mirrordir and nomirrordir. The underlying test involves 3 parallel clients with 10 threads to simulate real world parallel access. Multiple clients can be involved by starting elbencho in respective client using **elbencho --service**. Performance difference between mirrored and non-mirrored directory can be obtained by comparing the average time taken and average latency statistics. Mirroring disabled directory has better write performance than mirroring enabled directory. This behaviour is expected due to data replication at the buddy mirrored targets. Comparatively read performance does not get affected by huge margins between the directories, but are slightly better in non-mirrored directory for the same reason. For the upcoming performance comparisons, the statistics from table 1 are considered as control to evaluate the performance difference.

4.4.2 Performance difference during node failure

To study the performance difference under node failure/ node crash conditions, one of the storage node is simulated to behave as a failed node as explained in section 4.1. IO performance benchmarking is performed during this node failure condition of the file system and compared with the control benchmarking test performed in the previous section. Node failure state can be confirmed through the **--listtarget** command as shown in Figure 7.

Figure 7 shows that storage target with NodeID 2 is offline, which in-turn means that the specific storage target is not reachable for any operations. Consistency of the system is still good, because after node failure, there hasn't been any read or write operations done, hence the consistency state. Benchmarking under node failure can be performed only under mirrored condition as the IO operation fails due to node un-reachability in



```
[root@masterbgfs elbencho]# beegfs-ctl --listtargets --mirrorgroups --state
MirrorGroupID MGMemberType TargetID     Reachability  Consistency   NodeID
============= ============ ========     ============  ===========   ======
         1000      primary      103           Online         Good        3
         1000    secondary      102          Offline         Good        2
         1001      primary      104           Online         Good        4
         1001    secondary      105           Online         Good        5
```

Figure 7: State and availability of storage targets

non-mirrored condition. From table 2, write performance under node failure condition is maintained same or slightly better than normal stable condition, because the secondary node replication is not performed. The read performance of the system is highly degraded by a factor of 81.9% in terms of time taken and the latency is increased twice as normal condition.

System status	**Operation Type**	**File size (MiB)**	**Throughput (MiB/s)**	**Avg. Time (s)**	**Avg. Latency (s)**
Normal	Write	3720	22	160.58	4.94
Node failure	Write	3720	23	156.25	4.71
Normal	Read	3000	23	70.69	2.49
Node failure	Read	3000	22	128.58	4.74

Table 2: Benchmarking statistics on mirroring

4.4.3 Performance during node re-synchronization

During the above test, the system was fresh and had very less data written to the underlying physical storage targets. This system condition is not always similar to real world scenarios, where the file systems have huge volumes of data already residing in the targets. To replicate this general use case condition, huge volume of data is written to the system under node failure condition and the failed node is restarted which initiates node re-synchronization. Benchmarking test is performed under this new state to study the extent of further performance degradation.

System status	**Operation Type**	**File size (MiB)**	**Throughput (MiB/s)**	**Avg. Time (s)**	**Avg. Latency (s)**
Node resynchronization	Write	3720	7	474.83	7.44
Node resynchronization	Read	3000	15	197.92	7.30

Table 3: Benchmarking statistics on node re-synchronization

Analysing the results from table 3, the write and read performance of the system has been reduced by a factor of 195.5% and 179.5% respectively. Node re-synchronization has affected the IO performance of the system by huge factor than usual.

5 Conclusion

BeeGFS is an efficient high performance PFS that supports high availability and scalability. Performance and availability enhancements in BeeGFS are achieved through feature tuning. Multi-server parallel access for I/O operations enhance performance in general PFS. BeeGFS additionally enables tuning of file striping and mirroring functionality to enhance performance and availability according to underlying physical hardware resource. Through mirroring enabled storage, availability of the system can be increased.

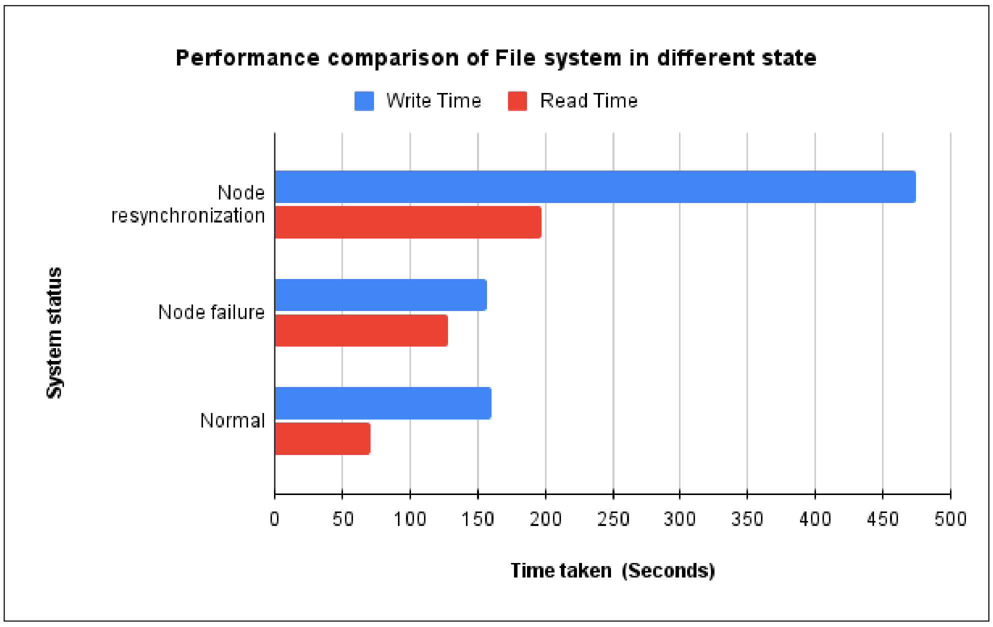


Figure 8: Performance comparison chart of the file system under different system state

Our study conducted on the performance through benchmarking reveal that there is a degradation of performance under node failure condition. Though this behaviour is expected as reduced servers shall be involved in I/O operations, the scale of performance degradation is huge that it shall hinder the efficiency of application operations that use this file system. Further reduction of performance is observed when IO operations are performed under node re-synchronization condition, which is likely to happen in real world use cases. Although there is severe performance drops while maintaining high availability, this does not make the system unusable. Without this mirroring, the system shall be completely unreachable leading to service outages. So, BeeGFS can still be better software based PFS option depending upon use case. It can be used for high performance engines or time dependent data streaming applications with prior measures to tackle the performance reduction under node crashes.

References

[1] BeeGFS. Beegfs- documentation. https://doc.beegfs.io/latest/index.html.

[2] Francieli Boito, Guillaume Pallez, and Luan Teylo. The role of storage target allocation in applications' i/o performance with beegfs. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 267–277. IEEE, 2022.

[3] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn Mohror, and Weikuan Yu. I/o characterization and performance evaluation of beegfs for deep learning. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[4] elbencho. Elbencho - documentation. https://github.com/breuner/elbencho.

[5] Yuri Goncharuk, Yuri Grishichkin, Alexander Semenov, Vladimir Stegailov, and Vasiliy Umrihin. Evaluation of the angara interconnect prototype tcp/ip software stack: Implementation, basic tests and beegfs benchmarks. In *Russian Supercomputing Days*, pages 423–435. Springer, 2022.

[6] Google. Parallel file systems for hpc workloads. https://cloud.google.com/architecture/parallel-file-systems-for-hpc#:~:text=In%20a%20parallel%20file%20system%2C%20several%20clients%20store%20and%20access,workloads%20that%20use%20SAS%20applications.

[7] Frank Herold and Sven Breuner. An introduction to beegfs. 2, 2018.

[8] Tecmint Linux Blog Aaron Kili. How to configure custom ssh connections to simplify remote access. https://www.tecmint.com/configure-custom-ssh-connection-in-linux/.

[9] Glenn Lockwood. Getting started with elbencho. https://www.glennklockwood.com/benchmarks/elbencho.html.

[10] NetApp. Beegfs on netapp with e-series storage - architecture overview. https://docs.netapp.com/us-en/beegfs/beegfs-architecture-overview.html#building-block-architecture.

[11] Wikipedia. Disk mirroring. https://en.wikipedia.org/wiki/Disk_mirroring.

[12] Wikipedia. Parallel virtual file system. https://en.wikipedia.org/wiki/Parallel_Virtual_File_System.

A Elbencho scripts

```
threads=10
iodepth=4
block=1M
hosts=masterbgfs,client1,client2
direct=1
size=124M
files=1
dirs=1
deldirs=0
delfiles=0
blockvaralgo=fast
mkdirs=1
write=1
rand=1
lat=1
```

Listing 1: Writetest.elbencho - Benchamrking script file

```
threads=10
iodepth=4
block=1M
hosts=masterbgfs,client1,client2
direct=1
size=100M
files=1
dirs=1
deldirs=0
delfiles=0
blockvaralgo=fast
mkdirs=1
read=1
rand=1
lat=1
```

Listing 2: readtest.elbencho - Benchamrking script file