PROCEEDINGS OF SCIENCE

# Distributed filesystems (GPFS, CephFS and Lustre-ZFS) deployment on Kubernetes/Docker clusters

**Federico Fornari,**[a,*] **Alessandro Cavalli,**[a] **Daniele Cesini,**[a] **Antonio Falabella,**[a] **Enrico Fattibene,**[a] **Lucia Morganti,**[a] **Andrea Prosperini**[a] **and Vladimir Sapunenko**[a]

[a] *INFN-CNAF,*
*Viale Berti Pichat 6/2, Bologna, Italy*

*E-mail:* federico.fornari@cnaf.infn.it

Nowadays Kubernetes has become a powerful tool to deploy and manage containerized applications. Modern datacenters need distributed filesystems to provide user applications with access to stored data on a large number of nodes. The possibility to mount a distributed filesystem and exploit its native application programming interfaces in a Docker container, combined with the advanced orchestration features provided by Kubernetes, may enhance flexibility in data management, installation, running and monitoring of transfer services. Moreover, it would allow the execution of dedicated services on different nodes, in isolated and automatically replicable environments, so to improve deployment efficiency and fail-safeness. The goal of this work is to demonstrate the feasibility of using Kubernetes and Docker to setup clients capable to access a distributed filesystem from existing clusters and to create clusters based on containerized servers. Despite being just a proof of concept, the effort has shown the possibility of using different types of distributed filesystems (GPFS, CephFS, Lustre-ZFS) with equally positive results.

*International Symposium on Grids & Clouds 2021, ISGC2021 22-26 March 2021*
*Academia Sinica, Taipei, Taiwan (online)*

*Speaker





PoS(ISGC2021)020

## 1. Introduction

The primary goal of this work is to prove the feasibility of using Docker [1] and Kubernetes [2] to setup clients capable of accessing distributed filesystems from existing cluster deployed on virtual machine servers, as well as to create distributed filesystem clusters based on containerized servers. The same kind of deployment has been replicated using three different distributed filesystems:

- IBM Spectrum Scale [3] (formerly GPFS)
- CephFS (the Ceph [4] POSIX-compliant filesystem)
- Lustre [5] with ZFS backend

After verifying the possibility to setup the containerized clusters, some tests have been performed in order to compare non-containerized and containerized servers scenarios, thus assessing eventual behavior differences. Section 2 provides technical details on the clusters deployment and on comparative tests. Section 3 summarizes our conclusions, based on the results of our activities.

## 2. Clusters Deployment and Preliminary Tests

### 2.1 The Openstack Project

An Openstack [6] Project has been created in order to setup the distributed filesystem clusters with Kubernetes 1.20 and Docker 20.10. The Project has been provided with 10 virtual machines, 4 for the Lustre cluster, 3 for the Ceph cluster and 3 for the GPFS cluster, all of them with the same flavor (8 virtual CPUs, 16 GB of RAM, 160 GB of disk space for the root filesystem and CentOS 7.6.1810 operating system, see Figure 1). Each cluster has been created on a separated subnet and provided with two 670 GB volumes (plus two 30 GB volumes for the Lustre metadata server in high availability): so, 4 TB of allocated storage space in total. Figure 2 shows the Project network topology in detail, with each cluster having its instances located into its specific subnet. Each subnet has been connected to the Project router in order for the instances to be accessed from public network through floating IPs. Mutual connection between instances could reach 1 Gbit/s at maximum, which was tested using iperf [7].

#### 2.1.1 GPFS cluster deployment

In Figure 3, on the left, a scheme illustrating the GPFS cluster with non-containerized servers is reported, with two servers deployed on *k8s-gpfs-1* and *k8s-gpfs-2* virtual machines and a containerized client on virtual machine *k8s-gpfs-3*, mounting the filesystem to perform read/write operations. Each GPFS server has a 670 GB disk attached and configured as network shared disk (NSD), with a replica 2 filesystem created on top of the shared disks. IBM Spectrum Scale version used for the setup is 5.0.5-2. At a later stage, the GPFS servers have been put in Docker containers isolated from the host environment and managed by Kubernetes (see Figure 3, on the right). Some read/write tests have been conducted comparing these two kinds of setup, with non-containerized and with containerized servers, in order to understand if server containerization could affect performances.

Kubernetes is mainly based on Pods. A Pod is generally a group of one or more containers, with shared storage, shared network resources, and a specification for how to run the containers.

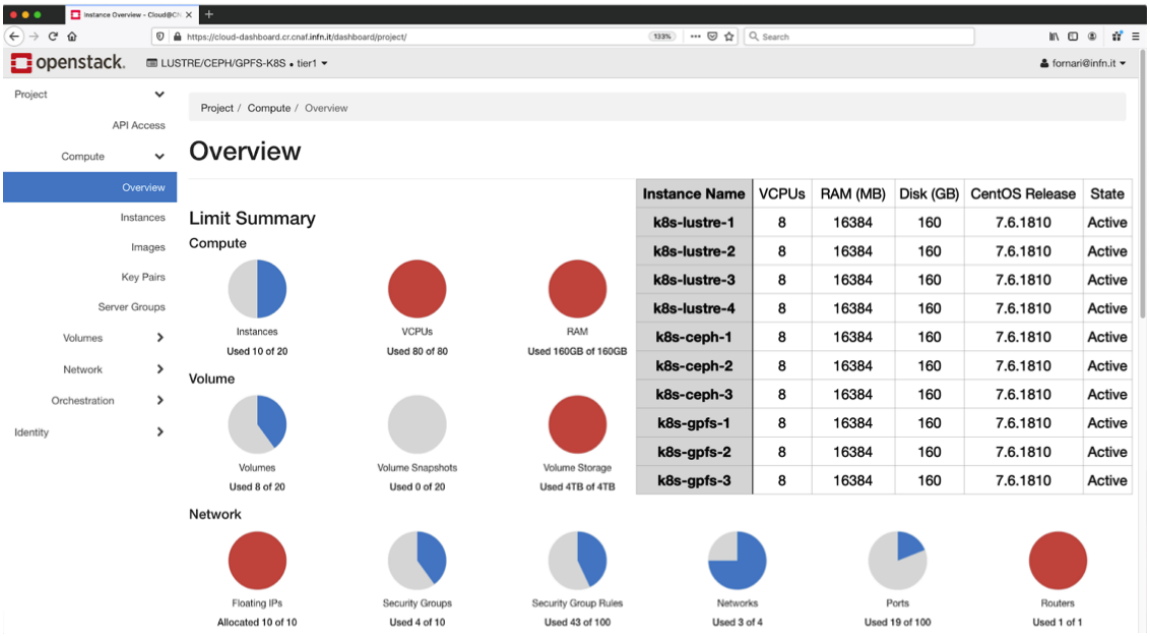


**Figure 1:** An overview of the Openstack Project environment used to setup the distributed filesystem clusters with Kubernetes.

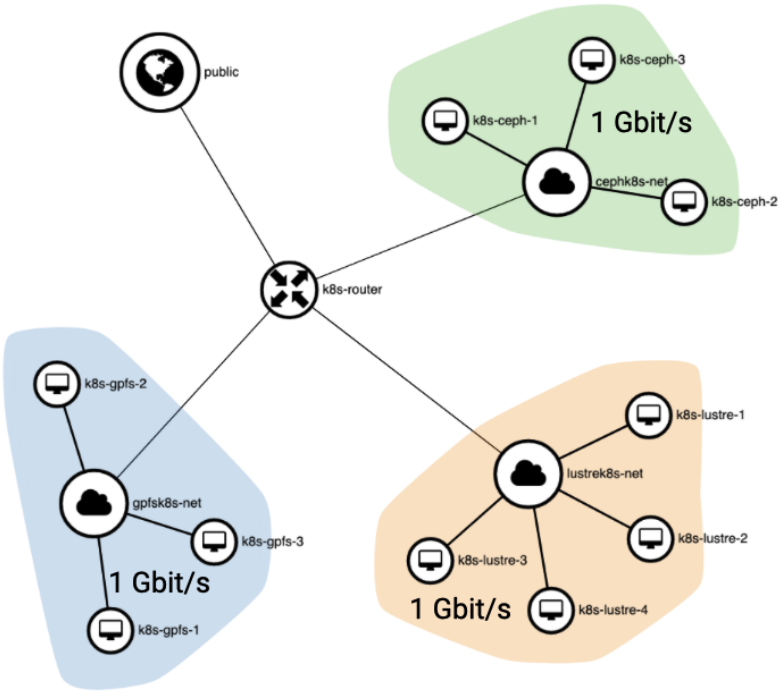


**Figure 2:** The network topology of the Openstack Project in detail. Connection bandwidth between instances: 1 Gbit/s.

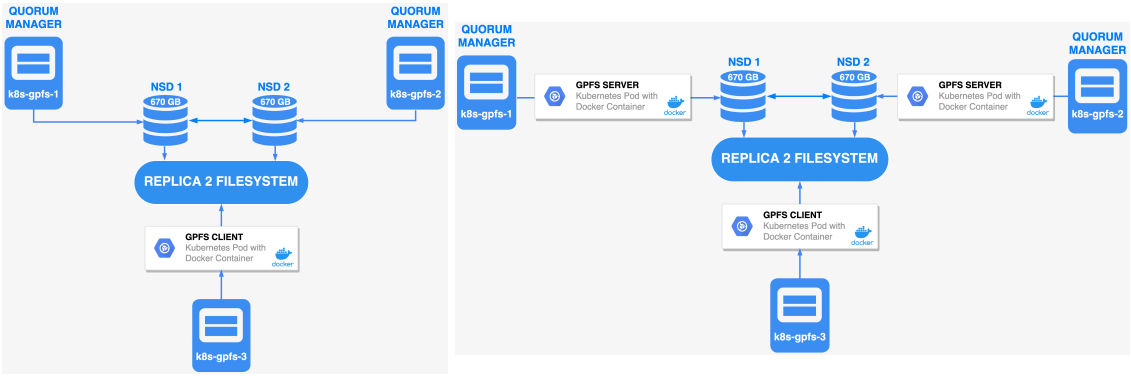


**Figure 3:** GPFS cluster setup with non-containerized servers (left) and with containerized servers (right).

```
[fornari@farm-ops k8s-gpfs]$ cat k8s/server/pod.yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: gpfs-server-1
  namespace: gpfs
  labels:
    app: gpfs-server
spec:
  selector:
    matchLabels:
      app: gpfs-server
  serviceName: gpfs-server-1
  replicas: 1
  template:
    metadata:
      name: gpfs-server-1
      labels:
        app: gpfs-server
    spec:
      hostNetwork: true
      nodeName: k8s-gpfs-1.novalocal
      containers:
      - name: gpfs-server-1
        image: centos/systemd
        command: [ "/bin/bash", "-c", "--" ]
        args: [ "./scripts/start-container.sh" ]
        securityContext:
          privileged: true
        volumeMounts:
        - name: start-container
          mountPath: /scripts/start-container.sh
          subPath: start-container.sh
        - name: init-container
          mountPath: /scripts/init-container.sh
          subPath: init-container.sh
        - name: gendir
          mountPath: /var/mmfs/gen
      volumes:
      - name: start-container
        configMap:
          name: start-container
          defaultMode: 0755
      - name: init-container
        configMap:
          name: init-container
          defaultMode: 0755
      - name: gendir
        hostPath:
          path: /var/mmfs/gen
          type: Directory
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: start-container
data:
  start-container.sh: |
    #!/bin/bash
    # script set in background
    setsid /scripts/init-container.sh &
    # run systemd
    exec /usr/sbin/init
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: init-container
data:
  init-container.sh: |
    #!/bin/bash
    [install kernel, kernel-headers, kernel-devel packages]
    [install openssh server and client packages]
    [install iozone]
    [install ibm-spectrum-scale packages]
    [configure and start openssh server]
```

**Figure 4:** The content of a `yaml` configuration file for a GPFS server/client Kubernetes Pod.

In this work, a Pod always includes a single Docker container. The Kubernetes server/client Pods have been configured as *StatefulSet* in order to preserve volume data and network IP/hostname across possible restarts or failures, as shown in Figure 4. Directory `/var/mmfs/gen`, containing all necessary files for GPFS to work, is mounted from the host in order to guarantee persistence of configuration files and to easily recover failed Pods, adding them back into the GPFS cluster after restart. Through the *start-container* Kubernetes configmap, `systemd` is started in the Docker containers, and then the `init-container.sh` script from *init-container* configmap is executed in background, in order to start GPFS and *ssh* services inside the containers.

The monitoring of the GPFS cluster has been accomplished with the use of IBM Spectrum Scale GUI [8], configured to show performance data collected from the NSD servers. GPFS GUI provides predefined monitoring dashboards which are very useful in order to perform real-time checking on cluster activity and health status (see Figure 5, on the left). However, data manipulation from this GUI is not so immediate, so that in order to get access to collected data in `.csv` format, a Grafana

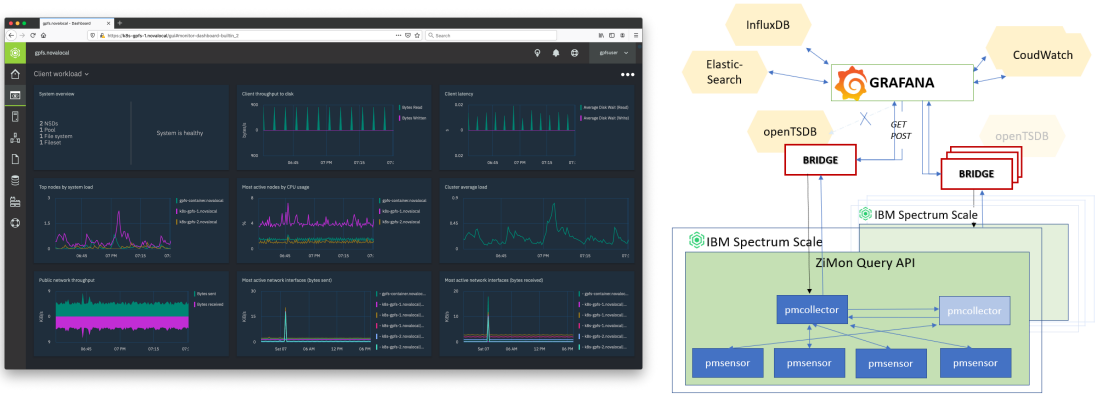


**Figure 5:** GPFS cluster monitoring has been accomplished with GPFS GUI (left) and IBM Spectrum Scale Grafana Bridge Service (right).

bridge service [9], introduced with Spectrum Scale 5.0.5-2, has been configured to send monitoring data to Grafana [10] dashboards, from which this data could be easily downloaded and processed, in order to extract useful statistics (see Figure 5, on the right).

### 2.1.2 Ceph cluster deployment

Left side of Figure 6 shows Ceph cluster setup with non-containerized servers: each Openstack node hosts Ceph monitor and manager services, while the CephFS metadata server has been instantiated on *k8s-ceph-1* virtual machine. Ceph Nautilus 14.2.10 installation on the cluster has been accomplished with Ansible 2.8.19 [11]. *k8s-ceph-1* and *k8s-ceph-2* servers have been provided with Ceph Object Storage Devices (OSD) configured on 670 GB disks, and a CephFS replica 2 filesystem has been created over the OSDs. Finally, a client Pod mounting the filesystem to perform read/write tests has been started on *k8s-ceph-3* virtual machine.
A similar setup could be achieved, this time containerizing all Ceph services, with the use of specific Kubernetes Helm [12] Charts (see Figure 6 on the right). Helm is an application manager for Kubernetes and uses a packaging format called Charts. A Chart is a collection of files that describe a related set of Kubernetes resources, and can be used to easily deploy complex applications. A Github Project [13] has been created in order to collect these Helm Charts and allow the possibility to publicly download them through Github Pages. The Charts support Ceph Nautilus version, BlueStore object backend, CentOS 7 operating system and advanced RBD image features (layering, striping, journaling, etc.). Users just need to edit `values.yaml` and `ceph-overrides.yaml` configuration files in order to set services instantiation, with eventual replicas, on desired Kubernetes cluster nodes.
The `yaml` configuration file for a Ceph client Pod is reported in Figure 7: the *init-container* configmap manages the installation of all required packages, while the CephFS filesystem is mounted on the Docker container via native Kubernetes CephFS features, simply providing Ceph monitor address, username and corresponding secret, obtained with specific `ceph auth get-key` command and stored into a file on the host. Grafana provides predefined dashboards to get a deep insight in

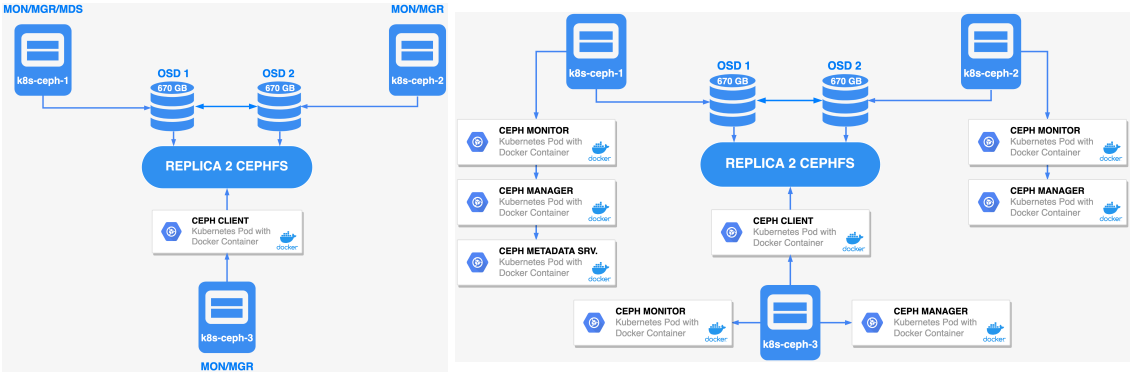


**Figure 6:** Ceph cluster setup with non-containerized servers (left) and with containerized servers (right).

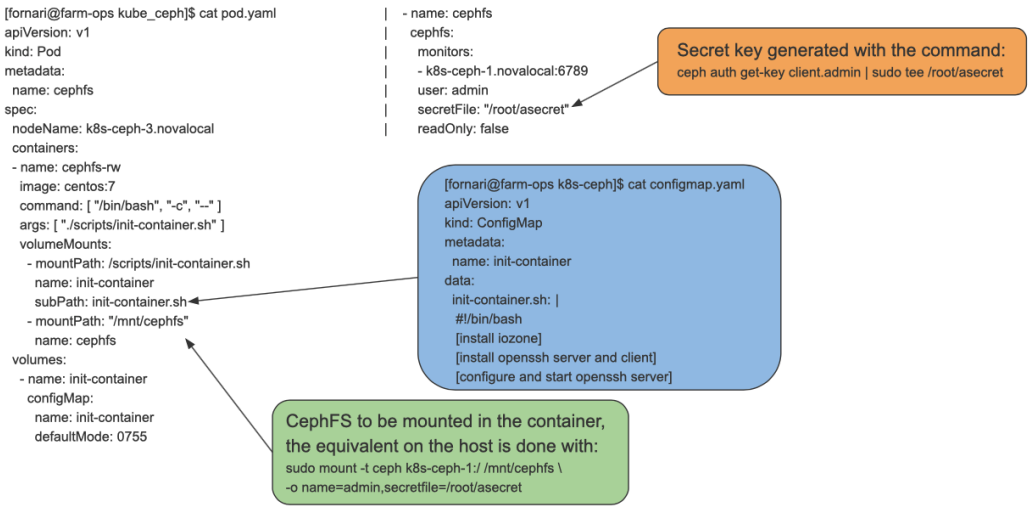


**Figure 7:** The content of a `yaml` configuration file for a Ceph client Kubernetes Pod.

Ceph cluster behavior, so this tool has been chosen to plot Ceph test results and extract relevant information.

### 2.1.3 Lustre cluster deployment

The Lustre Kubernetes cluster has been setup implementing replica 2 data redundancy using Distributed Replicated Block Device [14] (DRBD). As shown in Figure 8, an Object Storage Server (OSS) Pod has been configured on *k8s-lustre-1* virtual machine in order to use a 670 GB disk replicated on *k8s-lustre-2* via DRBD, and the same procedure has been adopted to setup a Metadata Server Pod on *k8s-lustre-3* using two 30 GB DRBD-coupled disks. A client Pod has been started on each of the 4 cluster virtual machines, but only one client Pod out of 4 has been involved in preliminary performance tests to resemble the configuration of the tests for the other filesystems. The Lustre version selected for the setup is 2.10.8, following compatibility matrix with CentOS 7.6 kernel version.

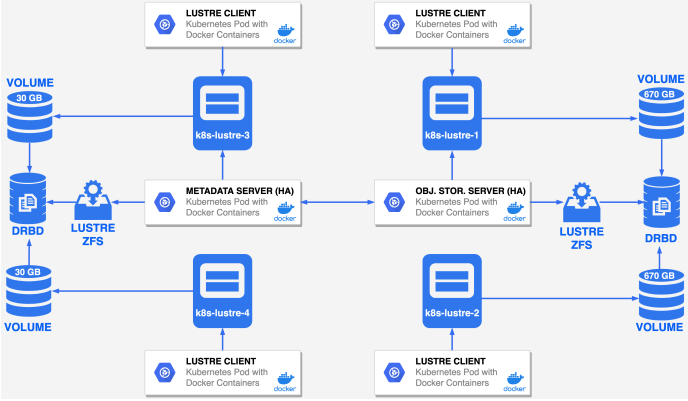


**Figure 8:** An overview of the Lustre cluster setup with High Availability servers configured using DRBD.

A GitLab project [15] has been created on official INFN GitLab to collect Lustre `yaml` configuration files. Users should just edit Kubernetes `yaml` configuration files for services, specifically `kube-lustre-configurator.yaml` to select which services to run among servers and clients (debug mode on installation process is also configurable), and `kube-lustre-config.yaml` to adjust servers and clients inner configurations and eventually enable High Availability for the servers. Dockerfiles for building of Docker images required by Kubernetes to deploy Pods are also available: *kube-lustre-configurator* container acquires the cluster configuration and provides the deployment of all the required services, then the remaining Docker images provide the installation of Lustre, ZFS and DRBD packages, starting the services in the end. In the Gitlab Project, multiple development branches for specific Lustre versions have been created. For each branch there are two paths, containing configuration files and scripts respectively for non-containerized and containerized server setups. Server setup is managed by Kubernetes in both cases, the only difference being the fact that in containerized situation, Lustre software and services are installed and run within containers, while in non-containerized scenario, Lustre packages are installed on the host and the services are executed after changerooting to a host directory.
Pod configuration in case of containerized servers is similar to GPFS case, where container initialization runs in background after `systemd` start: after the installation of necessary packages, required kernel modules are loaded, network for communication between servers is configured with specific Lustre utilities (`lnetctl`) and the servers are finally started.
Lustre monitoring has been implemented through Lustre Monitoring Tool [16], developed by Lawrence Livermore National Laboratory (LLNL), available on Github and easily installable with LLNL wiki instructions. Moreover, `lstat` and `ltop` information, continuously printed to file as data series, has been parsed and converted to plots using Python dataframes.

## 2.2 Comparative Tests

Some read/write tests have been performed in order to compare non-containerized and containerized server setups. The tests have been carried out using `iozone` [17], a well-known filesystem benchmarking tool, and consisted in sequentially writing and reading binary files of increasing size from 1 to 16 GB with a number of parallel threads equally increasing from 1 to 16.

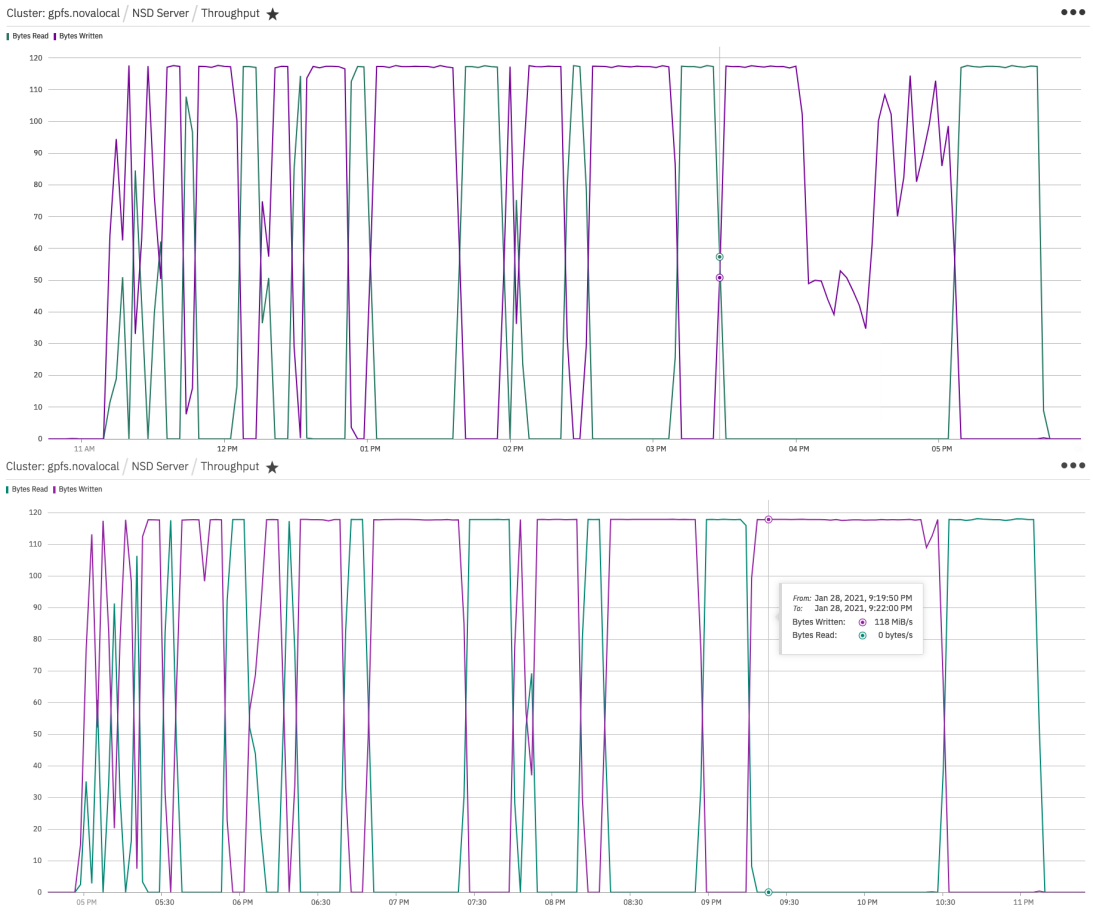


**Figure 9:** GPFS non-containerized (top) and containerized (bottom) NSD servers read/write throughput.

### 2.2.1 GPFS results

Starting with GPFS, the top plot in Figure 9 shows read/write throughput for the NSD servers as a function of time in case of non-containerized server setup. Bandwidth saturation is evident at 118 MiB/s, corresponding to about 123 MB/s, which is in fact 1 Gbit/s. The square wave pattern of the throughput is due to consecutive test sessions of exclusive sequential writing and reading, systematically reaching bandwidth saturation. Interestingly, there are no significant differences when servers are containerized, as shown at the bottom of Figure 9: read/write throughput saturates again at 1 Gbit/s, and it also seems more stable in the late writing phase of the test with respect to non-containerized servers scenario. In Figure 10 some surface plots showing throughput seen by iozone on client side are reported; different throughput ranges are represented with different colors. In case of non-containerized servers, excluding caching effects for low number of threads and small file sizes, a stable read throughput at 120 MB/s and a write throughput at 60 MB/s are seen. Physical bandwidth limit at 1 Gbit/s is highlighted. Again, in case of containerized servers, no relevant differences are detectable, as evident from Figure 11: same rates at 120 MB/s for reading and 60 MB/s for writing, with evident caching peaks for small number of threads and small file sizes.

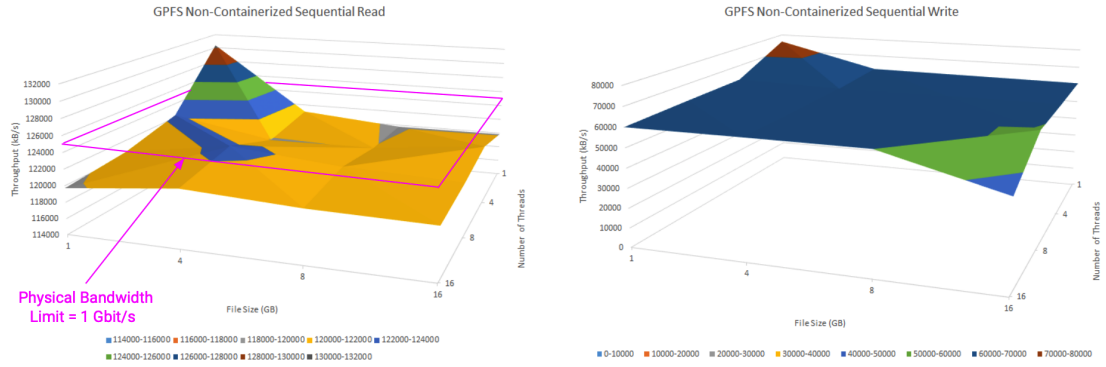


**Figure 10:** GPFS read/write throughput measured by `iozone` in case of non-containerized NSD servers.

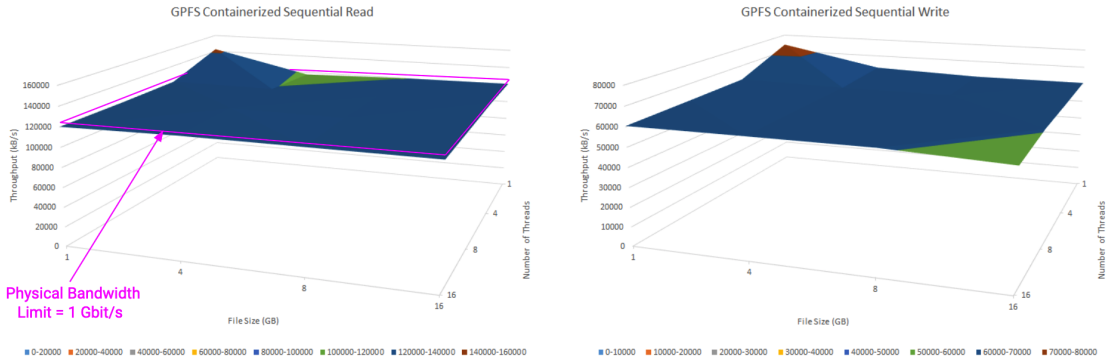


**Figure 11:** GPFS read/write throughput measured by `iozone` in case of containerized NSD servers.

### 2.2.2 Ceph results

Concerning Ceph, the plot in Figure 12, on the top, shows read/write throughput for one of the two OSD servers in non-containerized scenario as a function of time; the behavior is identical for the other OSD server. It is evident that 1 Gbit/s bandwidth saturation is reached only during writing phases and for limited time intervals, probably due to non-optimal default CephFS configuration. Read throughput does not exceed 90 MB/s. Server containerization does not yield significant changes in throughput behavior also for Ceph (bottom of Figure 12). Saturation at 1 Gbit/s is reached only along limited time intervals during writing operations, and read throughput does not exceed 75 MB/s. Caching effects for small files are strongly evident for Ceph, with peaks of almost 10 GB/s on client side, as shown in the iozone surface plots reported in Figure 13. Far from small files vs. few threads region, read/write throughput gets comparable to GPFS one, reaching about 100 MB/s for reading and less than 100 MB/s for writing. Again, a similar behavior can be noticed between non-containerized and containerized server setups, as shown in Figure 14: no relevant pattern differences between iozone surface plots with respect to Figure 13 are detectable.


**Figure 12:** Ceph non-containerized (top) and containerized (bottom) OSD servers read/write throughput.

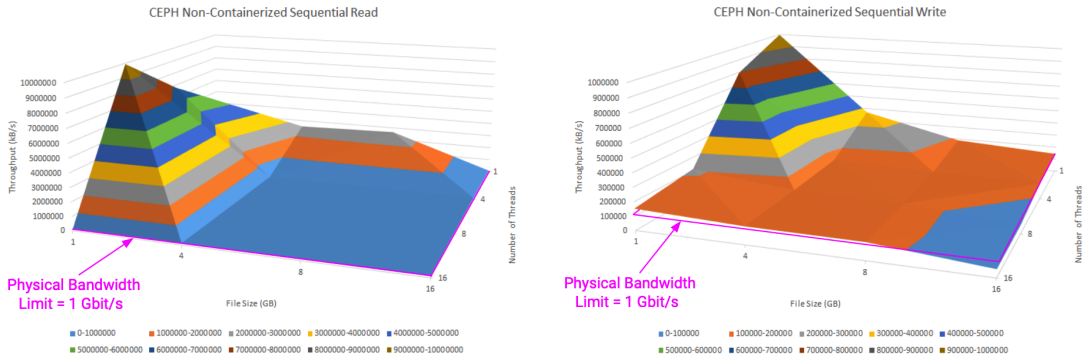


**Figure 13:** Ceph read/write throughput measured by `iozone` in case of non-containerized OSD servers.

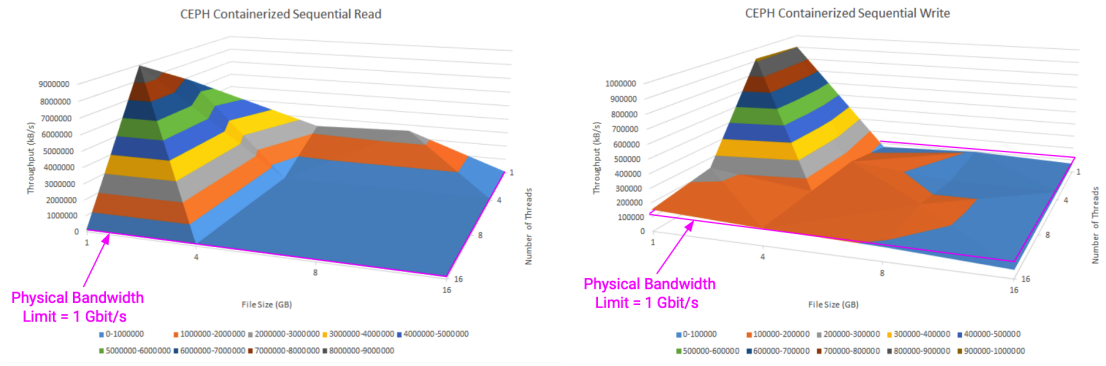


**Figure 14:** Ceph read/write throughput measured by `iozone` in case of containerized OSD servers.

### 2.2.3 Lustre results

Lustre revealed much more problems in an early phase with respect to Ceph and GPFS. The read/write throughput plot for the non-containerized OSS at the top of Figure 15 shows that bandwidth saturation is only reached during some very short time intervals, and then immediately lost, with very low average throughputs overall (under 50 MB/s for writing and under 30 MB/s for reading). Moreover, the non-containerized server test took more than double time to complete with respect to Ceph and GPFS, and the same results showed up with containerized servers: 14 hours to complete the test and very low average throughputs, as reported at the bottom of Figure 15. On client side, iozone surface plots for non-containerized OSS are very similar to Ceph ones, as seen in Figure 16: huge cache effects for read operations occur with small files and few threads, but, far from that region, read throughput is around 100 MB/s and write throughput is around 60 MB/s. Looking at Figure 17, no appreciable variations between non-containerized and containerized server setups on client side can be observed, except for slightly higher cache effects in case of non-containerized server setup.

By default, Lustre is optimized for large files reading and writing. So, some adjustments of the configuration were required in order to resemble the performances of the other filesystems. A considerable number of ZFS and other kernel module parameters have been tweaked in order to improve metadata and cache access and optimize network communications between servers. The updates to the configuration of the modified kernel modules are resumed in Figure 18.

After this fine tuning, the improvement has been noticeable. Read/write throughput for a containerized OSS after the modifications is shown in Figure 20, with writing peaks close to 100 MB/s and average reading throughput around 70 MB/s. Test duration got reduced to about 6 hours, approximately the same time registered for Ceph and GPFS. Cache effects are still evident on client side after Lustre fine tuning, but, for example, write throughput for higher number of threads and bigger files increases from 60 MB/s to about 80 MB/s, as shown in Figure 20.

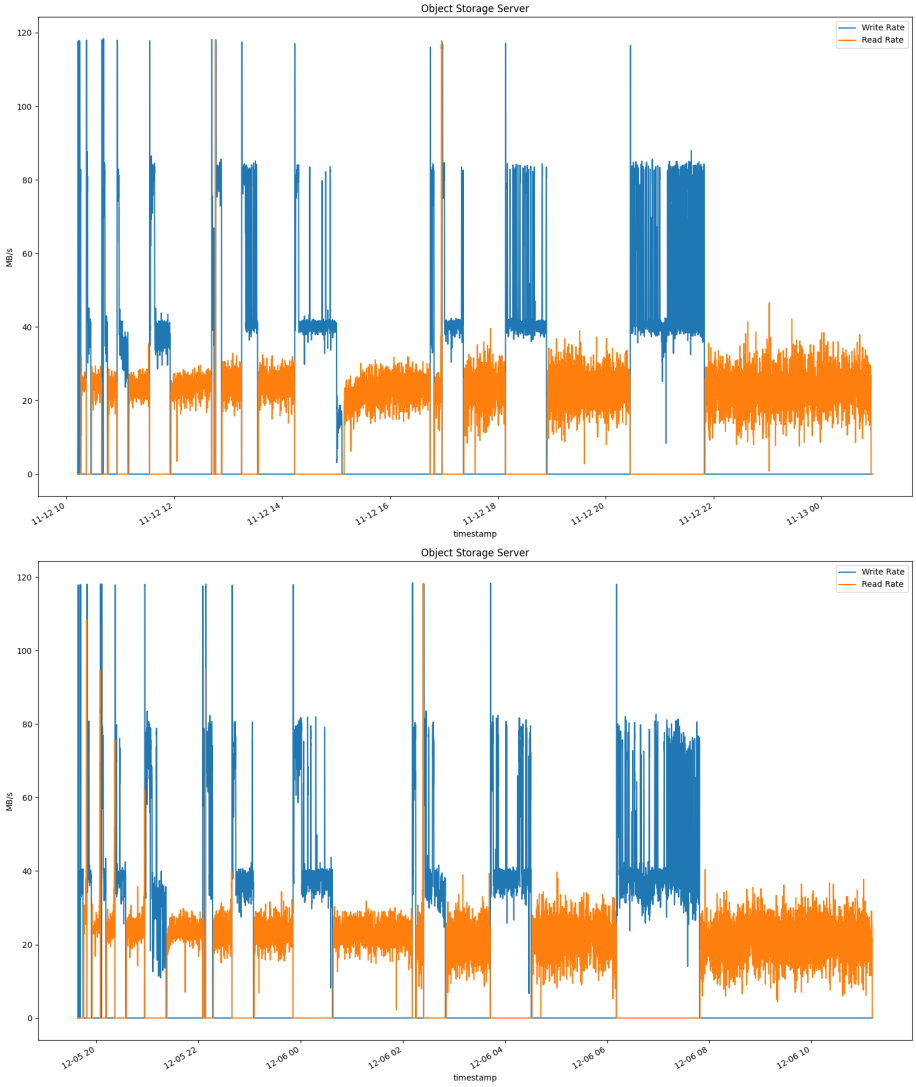


**Figure 15:** Lustre non-containerized (top) and containerized (bottom) OSS read/write throughput.

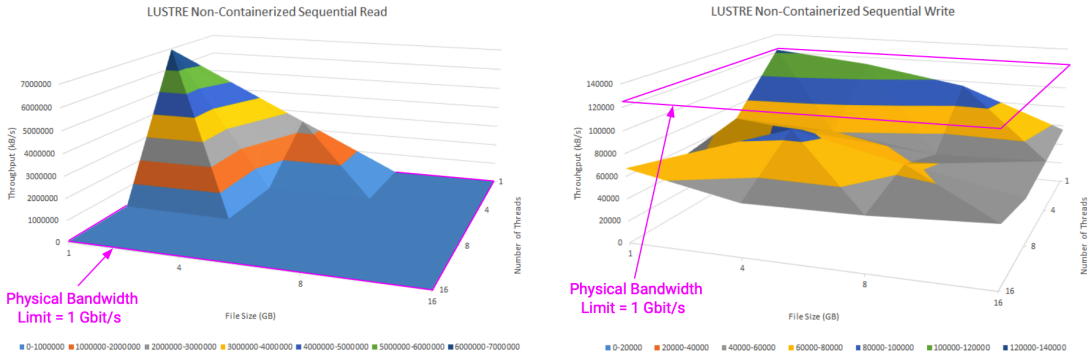


**Figure 16:** Lustre read/write throughput measured by `iozone` in case of non-containerized OSS.

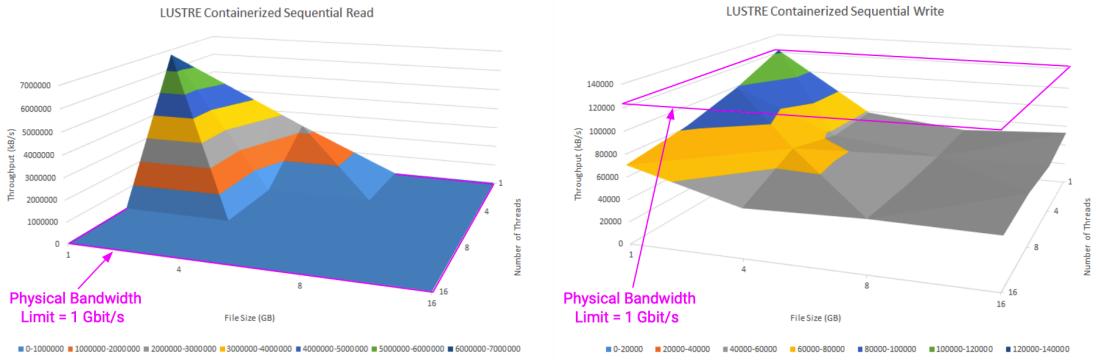


**Figure 17:** Lustre read/write throughput measured by `iozone` in case of containerized OSS.

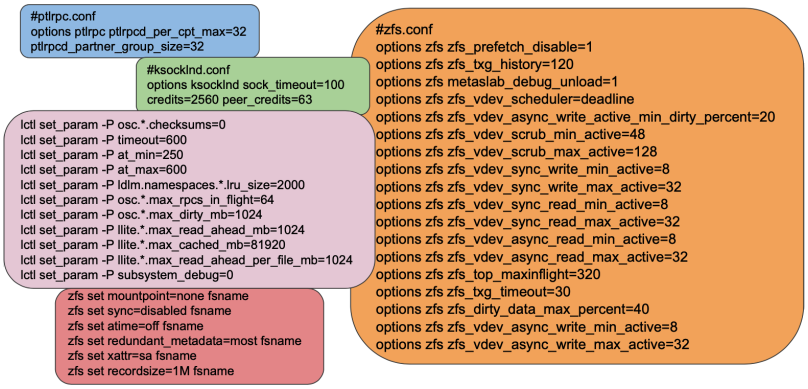


**Figure 18:** The updates to the configuration of the kernel modules adjusted to optimize Lustre performances.

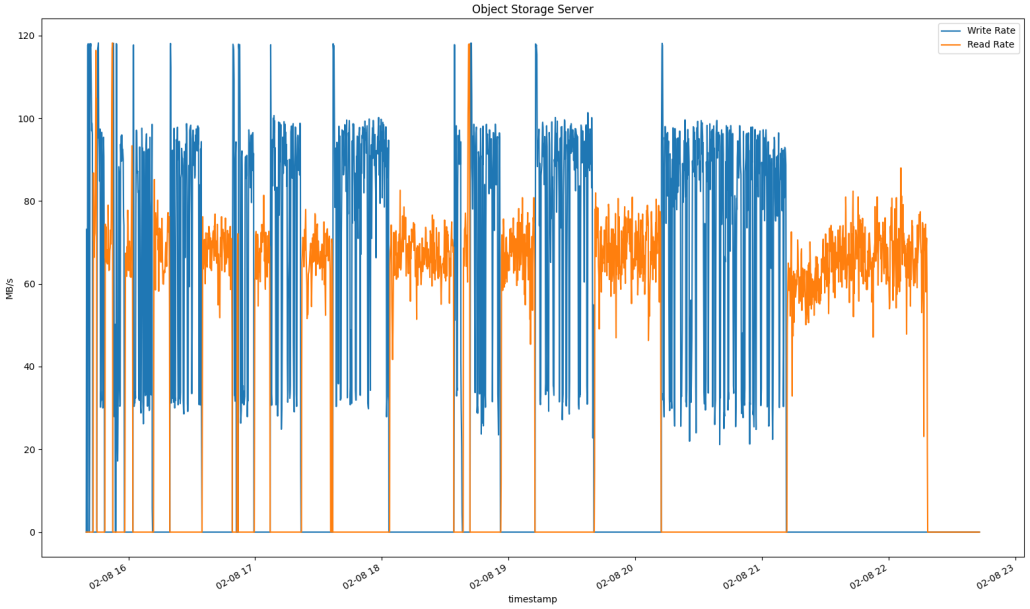


**Figure 19:** Lustre containerized OSS read/write throughput after tuning on configuration of kernel modules.

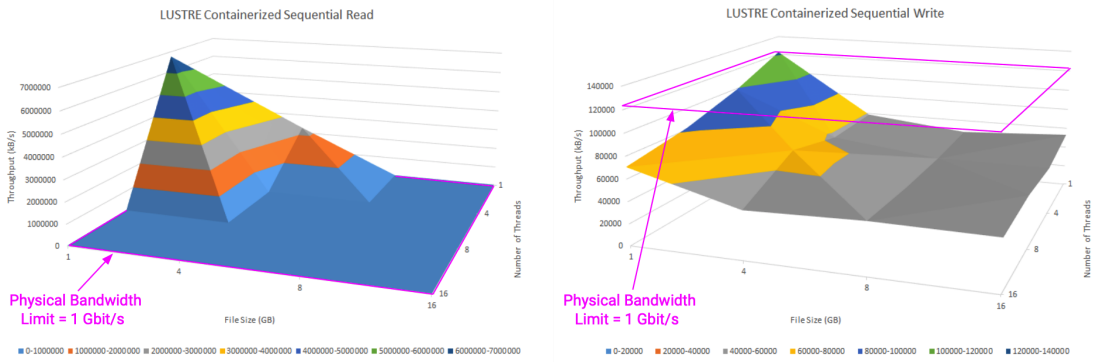


**Figure 20:** Lustre read/write throughput measured by `iozone` after tuning on configuration of kernel modules in case of containerized OSS.

## 3. Conclusions

The outcome of the proof of concept described above has been successful: Kubernetes can be used to deploy distributed filesystem clusters or simply client containers capable to access distributed filesystems from already existing clusters. GPFS and Ceph have shown to be able to saturate ethernet bandwidth without necessity of configuration adjustments, while Lustre has required some fine tuning on specific configuration parameters to gain performances comparable to the other filesystems. However, the most important observation here is the lack of any relevant difference between non-containerization and containerization of disk servers, leading to the possibility of implementing containerized setups without performance loss.

## References

[1] https://www.docker.com (Last seen: April 2021)

[2] https://kubernetes.io (Last seen: April 2021)

[3] https://www.ibm.com/products/spectrum-scale (Last seen: April 2021)

[4] https://ceph.io (Last seen: April 2021)

[5] https://www.lustre.org (Last seen: April 2021)

[6] https://www.openstack.org (Last seen: April 2021)

[7] https://iperf.fr (Last seen: April 2021)

[8] https://www.ibm.com/docs/en/spectrum-scale/5.0.5?topic=reference-spectrum-scale-gui (Last seen: April 2021)

[9] https://github.com/IBM/ibm-spectrum-scale-bridge-for-grafana (Last seen: April 2021)

[10] https://grafana.com (Last seen: April 2021)

[11] https://www.ansible.com (Last seen: April 2021)

[12] https://helm.sh (Last seen: April 2021)

[13] https://github.com/ffornari90/ceph-helm (Last seen: April 2021)

[14] https://www.drbd.org (Last seen: April 2021)

[15] https://baltig.infn.it/fornari/kube-lustre (Last seen: April 2021)

[16] https://wiki.lustre.org/Lustre_Monitoring_Tool (Last seen: April 2021)

[17] https://www.iozone.org (Last seen: April 2021)