Storage Solutions for Distributed Dockerized Cloud

Markus Peltola

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 11.4.2016

Thesis supervisor:

Assoc. Prof. Keijo Heljanko

Thesis advisor:

M.Sc. (Tech.) Mika Koskimäki

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Markus Peltola

Title: Storage Solutions for Distributed Dockerized Cloud

Date: 11.4.2016 Language: English Number of pages: 7+55

Department of Computer Science

Professorship: Computer Science

Supervisor: Assoc. Prof. Keijo Heljanko

Advisor: M.Sc. (Tech.) Mika Koskimäki

Distributed Object and Block Storages systems are studied in this thesis and their suitability as a storage solution for a dockerized cloud was evaluated. Docker is a relatively new virtualization framework. In beginning it was designed for containerizing processes on single host environments. However, it started to be used in multi host configurations and clouds, which has caused need for persistent storage solutions which are not relaying on host machine storage.
Two open source distributed storage solutions were studied. Swift is an eventually consistent Object Storage system developed for the Openstack project. Ceph is a consistent storage system including object, block and file system storage subsystems. Swift and Ceph Object Storage systems were compared against each other. The Ceph Block Storage performance was evaluated against the virtual machine disk. The results show that Ceph has double the throughput in small objects from 8KB to 128KB compared to Swift throughput, and 30% better performance in files from 256KB to 100MB. The main trend between Swift and Ceph is that Ceph has better throughput on read operations in all object sizes. The Ceph Block Storage system was able to utilize 88.5% of the virtual machine disk write throughput. Throughput efficiency was calculated by multiplying write throughput of Ceph block by three and it dividing by virtual machine disk write throughput. Ceph block throughput needed to be tripled because replication tripled amount of disk writes. Ceph journal files were not stored on the disk so those wont affect efficiency.

Keywords: Docker, Object Storage, Block Storage, scalable, cloud

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Markus Peltola

Työn nimi: Tallennusmenetelmät hajautetulle dockerisoidulle pilvelle

Päivämäärä: 11.4.2016 Kieli: Englanti Sivumäärä: 7+55

Tietotekniikan laitos

Professuuri: Tietotekniikka

Työn valvoja: Prof. Keijo Heljanko

Työn ohjaaja: DI Mika Koskimäki

Työssä käsitellään hajautettuja objekti- ja lohko -tallennusmenetelmiä sekä niiden sopivuutta pysyväistallennukseksi dockerisoituun pilveen. Docker on suhteellisen uusi virtualisointityökalu ja se oli alunperin suunniteltu pelkästään yhden koneen prosessien virtualisointiin. Sitä kuitenkin alettiin käyttämään pilvipalveluissa virtualisointityökaluna, mikä on aiheuttanut tarpeen hajautetulle tallentamiselle, sillä tallentaminen isäntäkoneen kovalevylle ei ole toimiva ratkaisu pilvipalveluissa.
Työssä käsiteltiin kahta avoimen lähdekoodin hajautettua tallennusjärjestelmää, Swift ja Ceph. Swift on Openstack projetin objekti-tallennusjärjestelmä. Ceph puolestaan tukee hajautettuja objekti, lohko, ja tietojärjestelmä tallennusmenetelmiä. Objekti-tallennuksessa Swiftin ja Cephin suorituskykyjä verratiin toisiinsa ja lohkotallennuksessa Cephin suorituskykyä verrattin virtuaalikoneen suorituskykyyn.
Tuloksissa huomattiin Cephin saavuttavan kaksinkertaisen suorituskyvyn verrattuna Swiftiin, kun testin objektien koko oli 8 kilotavusta 128 kilotavuun. Näitä suuremmilla objekteilla aina 100MB saakka suorituskyky ero oli enään 30% Cephin hyväksi. Yleisesti Cephi saavutti paremman suorituskyvyn objekteja luettaessa verrattuna Swiftiin. Cephin lohko tallennus osoitti testeissä hyvää suorituskykyä kyetessään 88,5% kirjoitus hyötysuhteeseen verrattaessa virtuaalikoneen kovalevyyn. Hyötysuhde laskettiin kertomalla Cephin lohkon surituskyky kolmella ja jakamalla se virtuaalikoneen kovalevyn suoritusteholla. Cephin suorituskyky kerrottiin kolmella sillä Cephi tallentaa kaiken kolmeen kertaan. Cephin lokikirjoituksia ei tarvinnut huomioida yhtälössä sillä niitä ei tallennettu kovalevylle.

Avainsanat: Docker, objekti, lohko, tallennus, skaalautuva, pilvi

Preface

I would like to thank Oy LM Ericsson Ab and a specially Joachim Ekman for giving me possibility to make this thesis. I would also like to thank Assoc. Prof. Keijo Heljanko for supervising my thesis and for all the guidance and excellent feedback he provided for the thesis. Last but not least would like to thank my thesis advisor M.Sc. (Tech.) Mika Koskimäki for his help to get the work finished.

Jorvas, 8.4.2016

Markus Peltola

Contents

Abbreviations

ACK	Acknowledgement code
API	Application programming interface
BTRFS	B-tree file system
COSBENCH	Cloud Object Storage Benchmark
CPU	Central Processing Unit
CRUSH	Controlled Replication Under Scalable Hashing
DNS	Domain Name System
EBOFS	Extent and B-tree based Object File System
EXT4	Fourth extended file system
I/O	Input/Output
IP	Internet Protocol
MD5	Message-Digest algorithm
MDS	Meta Data Storage
NAS	Network Access Storage
NIC	Network Interface Controller
OS	Operating System
OSD	Object Storage Daemon
PG	Placement Group
RADOS	Reliable, Autonomic Distributed Object Store
RAM	Random Access Memory
SAN	Storage area network
SSD	Solid-state drive
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	Virtual Machine

1 Introduction

Nowadays more and more applications are converted to a cloud environment or they have been designed from start to suit the cloud. This increases the flexibility and robustness of applications and makes global usage of same application possible.

However, as different parts of the application are not necessarily running physically in the same location, it means that some traditional services and methods are not any more suitable for a cloudified environment.

Virtualization has come to solve flexibility and cost efficiency problems. In traditional virtualization the whole operating system has been virtualized, so that there are two operating systems running on top of each other. This solution gives excellent partitioning so that different VMs (virtual machines) can not affect each other. Drawback in this is that two kernels will take twice the memory. Even it has been developed so that we have hardware acceleration for most of execution some processor commands some commands are still needed to be software emulated.

If an application does not need total separation from host machine it can be ran inside container. Containers will only virtualize application not the whole operating system. As containers enclose container application from other applications and containers the application will see as it would be running on its own host without others.

Docker containers were originally developed for single machine application virtualization. Therefore it has lacked some multi-host capabilities like multi-host networking and data storage. However, in recent Docker versions there have been added features to partially solve these problems.

Docker was selected as container virtualization platform for this paper as it is currently most used containerization solution and also easiest to use. This master thesis compares two open source Object Storage solution and one Block Storage solution and their suitability as storage solution for docker based cloud. For Object Storages performance is compared between these two different Object Storages and for the Block Storage comparison is done against the virtual machine local disk and results are adjusted to take count of block device data replication.

2 Docker

Docker is an open source project, which has been designed to virtualize applications by containerizing them. It was open sourced in 2013, since then it has growth its popularity as a virtualization solution among developers. Docker has three base elements which are called build, ship and run. Build means that containers are built to include all binaries what application will need for running. Shipping means that containers should be easily distributed from one machine to another without any third party applications. And last run means that if containers runs on one host it should be able to run in any host with Docker.

To be able to run and used Docker containers the host machine needs a daemon named Docker engine to be running in it. The Docker engine is the tool to build, ship, run and manage containers. So it is the hearth of whole Docker infrastructure.

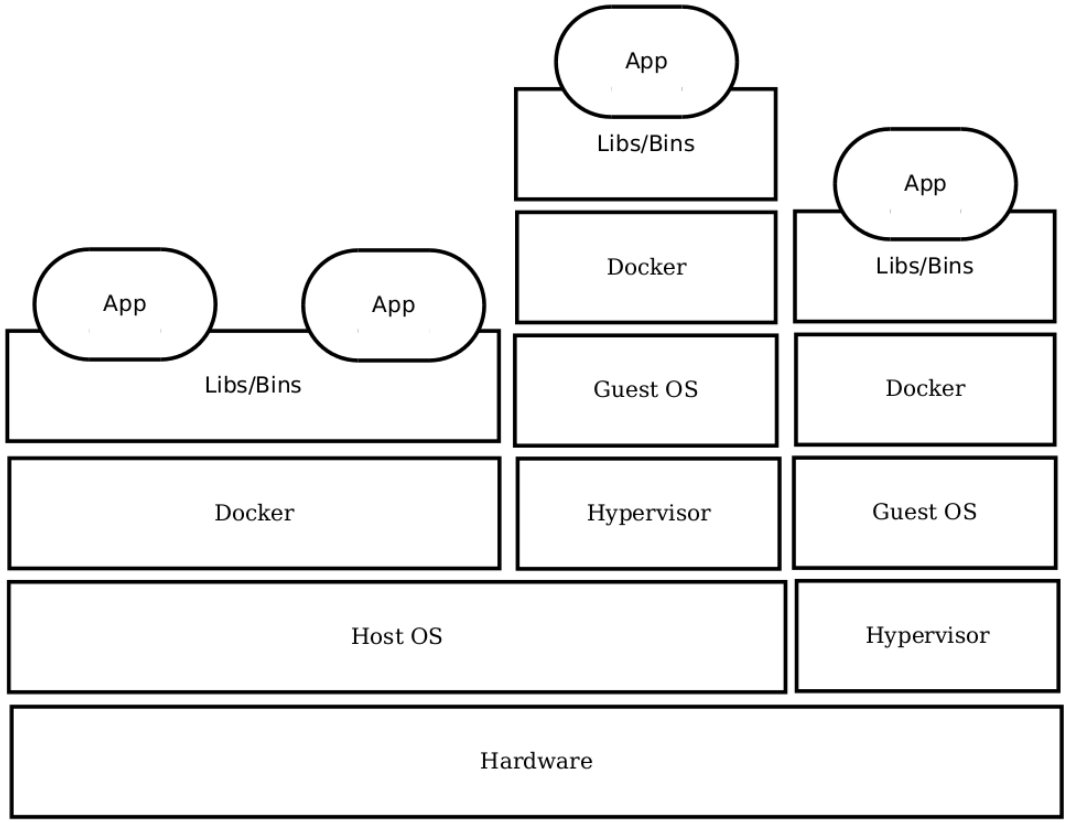

Figure 1: Layers in different configurations of running docker.

In Figure 1 are shown layers of three different Docker configurations where most left the Docker containers are running directly on a host OS. In middle containers run in virtual machine which is running on host OS. The most right containers are running in virtual machine running on top of hypervisor running directly on hardware. For the application point of view all environments are same and if container is made in one of those environments it will run in all of them.

Even the application would be using shared binary files it is not able to detect others containers using the same binary files. However, files can be shared among containers as long as containers are running on the same host.

Docker can be used as replacement for virtual machines at some extent. It will provide network, process and file system isolation. Advantage in the Docker approach is that almost no extra overhead will be added by using Docker containers compared application running on bare hardware [21]. All containers have all application binaries and its dependencies and only kernel is shared with host operating system and other containers.

Main strategies in Docker to isolate containers from each other and host machine is to use kernel cgroups and namespaces to hide information about other processes. With these abilities application in container will see as it would be running in system without any other processes. The kernel namespaces are used in Docker to isolate following properties:

- Proses id
- Networking
- Inter process communication
- Mount point management
- Unix timesharing system

Proses id isolation will ensure that only possesses running in container are shown. It combined with inter process communication will ensure that the process in container is not able to affect processes running in other containers or on host machine.

All containers have their own networking stack meaning that container can only access their own sockets and they see traffic as it would be in physically different machine. By linking ports from host machine to container. Container can get access to packets coming to that host port. However, it is possible to link host machine ports to container ports while container is running.

Docker containers are built using the Dockerfile script. It contains all information needed to build a specific container. First the base image of container is defined. It can be scratch, so nothing will be added or the base image can be any Linux distribution compatible with host machine kernel. In order to avoid the unnecessary data replication, the container building process in Docker is layered, so that many containers can use the same base image. Also if changes in Dockerfile are made, only layers after the change are needed to be rebuild.

The layered file structure is made possible with Docker own Union file system [23], which main purpose is to make container lightweight by making read-only layers that can be used by multiple containers. It enables in container rebuilding that only some layers will need to be rebuild. [24] Containers can also share volumes and host directories can be linked in container. [32] The sharing is currently working only in one host machine. this is one of the reasons, why this master thesis have been made.

More containers can be running on same host compared to VM as containers do not have everything duplicated as full VM will have. Containers are just isolated processes so they do not need to own a fixed amount of RAM, instead they can use shared kernel to share memory among containers. However, Docker supports memory usage limits per container. [36]

2.1 Security

Main difference between Docker container and VMs in security aspect is that container processes will communicate directly to host OS kernel, as every VM will have their own kernel on top of hypervisor and host OS kernel. This basically means that malicious application in VM will need to escape from VM before it can attack host OS, as in a Docker container applications have direct access to host kernel. However, the access to kernel is limited, but it will not prevent all attacks. If kernel has some vulnerability malicious container can try to use that to get control to host machine.

Docker has been designed to containerize applications. So different applications wont affect each other and they can use conflict libs without errors as processes shares only kernel and all libraries are container specific. [26] Security problem in Docker is that it do not have strict division between containers as all containers are using same kernel. Containers can not see other processes and or their data on a disk, there is still possibility for application to get out of it's container by misusing possible vulnerability in host kernel. Another problem is that users in container have the same privileges as host users, so container root is also root in host. Docker community is solving this problem by trying to make mapping so that container user will be different from host and other containers users, so that container user will never get full privileges. [25]

Other solution to tackle this security problem is that only signed containers should be ran so that unauthorized user could not run or modify containers in the system. Docker 1.8 have made this solution in user friendly form for the Docker developers. [27]

In version 1.10 Docker got new security features to prevent problems descried previously. Now system calls can be filtered so that if application do not need them they can be blacklisted for preventing exploits what uses those commands. Another great new feature is name-space mapping that finally allows mapping container root *pid*0 to other *pid* in host machine, so that even container root would get out of it container it is not root in host machine. [34]

2.2 Networking

In the local installation, Docker creates own bridged sub network named as *Docker*0 on the host machine. [22] Launched containers are added to it if network is not defined and all containers in that bridged network communicate each other using their subnet IP addresses. Containers can also use the *Docker*0 network for connecting outside the host machine.

It is possible to map host ports to container ports to expose container services as if they are running natively on the host. If a container needs to be hidden from other containers Docker allows creating private network and set specific containers to use those networks. Containers can be connected to more than one network or the network driver can be set to *none* so that container has no network end point connected to it. For network discovery containers can be linked so that they can find easily each other and with user specified networks there is DNS service for container discovery. [22]

However, locally defined networks are not enough for distributed systems. There are third party solutions like Weave [38] for multi host networking. However, Docker version 1.9 introduced production ready version of Docker multi-host networking. [33] Now it is possible to create multi-host wide networks with aliased containers. It would be great and simple solution if it would have good performance, but tests made for this thesis shows that it has a poor performance and it is not suitable for applications with heavy network usage.

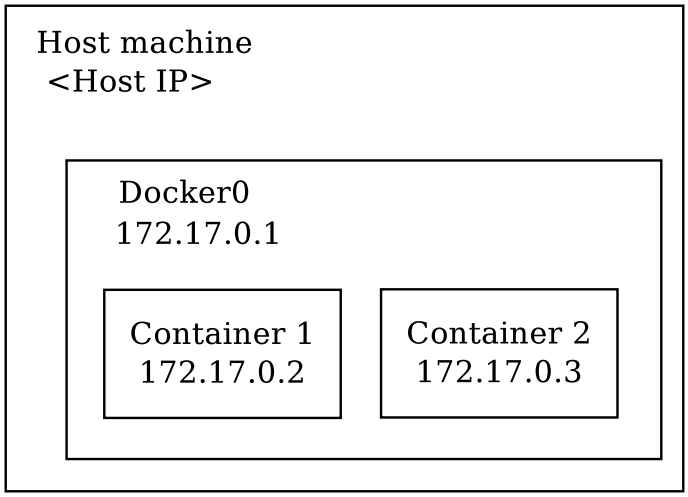

Figure 2: Docker networking structure. With default configurations containers are in the Docker0 network.

In Figure 2 is shown one host Docker network structure where the host machine Docker daemon have made the *Docker*0 network. There the host machine has the first IP what in this example is 172.17.0.1 then for containers are given next free addresses.

2.3 Docker Machine

Docker machine is tool to create and install Docker engines to physical or virtual machines. It supports wide variety of different platforms from bare metal to OpenStack and VMware virtualization environments. Environments can be mixed so different cloud infrastructures can be used parallel. It is also designed to manage

these different machines and their Docker engines. With Docker machine tool it is possible to use *ssh* in machines, copy files between host and Docker machine and start containers in those machines. With this tool it is also possible to restart machine, upgrade it's Docker engine or remove the whole virtual machine or just clean modification made by it in bare metal installation. [35]

2.4 Docker Swarm

Docker Swarm has been developed to manage a pool of Docker hosts and divide container payload among them. As single Docker daemon does not have any build in method to communicate to other Docker daemons on different host. There is need for hypervisor like application to control group of Docker hosts. Docker community has solved this problem with the Docker Swarm. Swarm contains many necessary tools to orchestrate pool of Docker daemons on different hosts. Main features in Swarm 1.0 are multi host networking and third party API to persistent storage support. [28]

Swarm uses network for communicating between nodes and all Swarm instances are similar no matter on what hardware or cloudification platform is used. It is possible to make hybrid cloud where different parts are running on different environments as long as each Swarm instance can access each other. [28]

Distributed Swarm cloud needs a way to discover nodes in a cluster and containers running in them to be able to work as one. Swarm can use different discovery services and currently it supports Consul, Etcd and Zookeeper. These are distributed key/value stores where all swarm nodes can store their IPs and port information for discovery. To keep cluster consistent Swarm uses one master node to manage containers and other services in Swarm cloud. In cloud can be multiple master, but only one is active at time. If current master drops from cluster other masters will vote which will be new active master. [29]

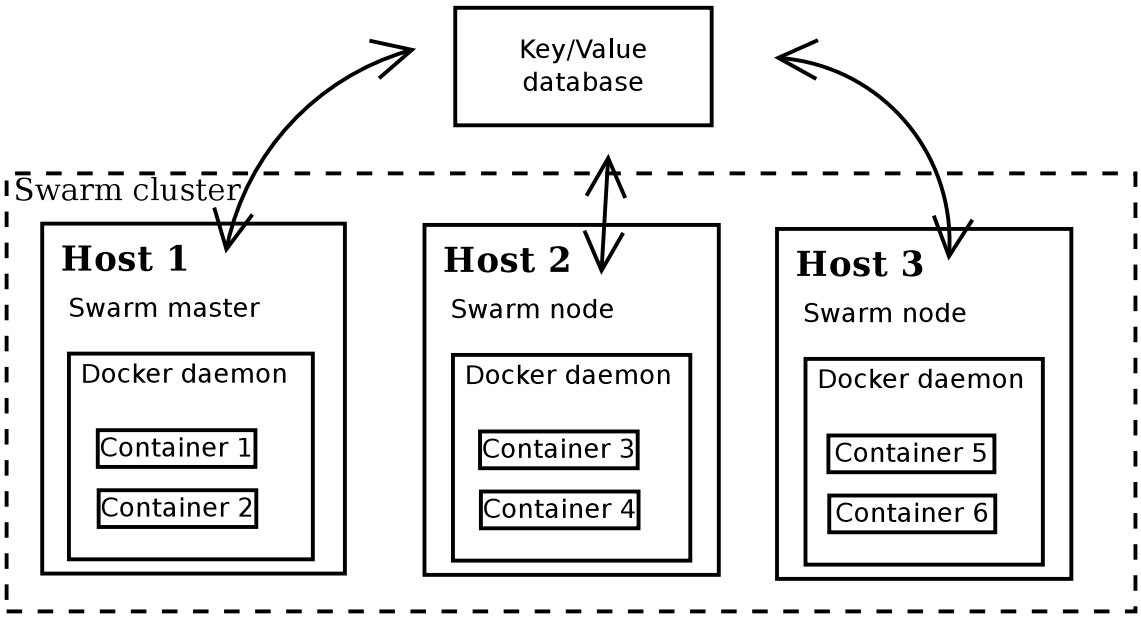

Figure 3: Swarm architecture. Swarm has one active master and multiple slave nodes. Swarm services uses external key/value store for distributing their ip:port information and also the current state of local Docker daemon information.

Figure 3 shows high level architecture of Swam where all host machines have docker daemon and swarm service running on them. Those will inform other nodes by its state by writing data to external key/value store. The key value store is exclude in image from Swarm cluster as it can be storing data of other services than Swarm.

2.5 Kubernetes

Kubernetes is an open source orchestration system made by Google for Docker containers. It has been designed to be lightweight and simple, but it still is able to run on multi cloud environment. It has automatic self-healing properties for replacement, restart and replication of containers. [20] Kubernetes have been developed based on Googles previous experiences from making Borg cluster manager [58]. So in Kubernetes many flaws found in design of Borg has been fixed.

Kubernetes is designed to deploy, schedule, maintain and scale the container cluster. So that for administer running new applications in cluster would be as easy as possible. Scheduling means that administer just starts pod in cluster and scheduler finds correct place for it based on scheduling rules and node stress.

In high level Kubernetes has the master node which orchestrates whole cluster. Nodes are running on top of VMs or bare hardware servers and inside those are Pods which contains application containers. Every node has one Kubelet service that keeps pods running and restart them if any pod is unhealthy. Also every node contains one proxy which will route API calls to applications. It is capable to make

load balancing for pod, so that it is possible to run same pods on different hosts and distribute load among those. [20]

In micro service application structure there is usually multiple small applications which should be communicating with each other. In this situation pods have their advantage as it ties applications needing connectivity to each other to one group. So when applications are needed to be relocated it is easy to just start the whole pod on new host. So the service responsible for relocating applications do not need to know anything about needs of applications inside pod. Grouping containers helps also to ensure low latency between containers in pod as they have to be running on the same host machine.

2.6 Flocker

Flocker is an open-source data volume management tool made by ClusterHQ. It allows to combine Docker containers and volumes so that when container is moved from one host to another the volume will also follow. Flocker also allows moving volumes without container binding. [39]

Data replication and the actual data replacement in Flocker is implemented by using third party network connected Block Storage. [40] For managing Flocker cluster needs one Flocker control service running in it. The controller service is interface for other services to manage Flocker global state and it commands all Flocker agents on all nodes. All docker nodes will need Flocker plugin to enable usage of Flocker movable volumes. Also every node needs Flocker agent which keeps node state correct based on configurations. When the agent founds out that state of node is incorrect it will notify the controller about situation and after notification it will fix state differences. [41]

So data safety in Flocker depends mostly on underlying blocks storage solution and how well its data replication strategies are designed. Another flaw of Flocker design is its control service as it is currently running only on one node and without redundancy support. However, if the cloud infrastructure supports restarting of services, died controller can be restarted on another host.

3 Cloud storage

More and more applications are running on public and private cloud. There is a need for cloud based storage solutions as traditional storage solutions (NAS, SAN) are too strict to adapt flexible need of cloud, in addition those systems were not designed to scale big enough for distributed cloud solutions. In many cases cloud services are growing fast and huge amount of calculation power and storage are needed. And it is not economically feasible to just scale up old solutions. Another challenge in large systems is that hardware and networking failures should be considered more common than in traditional storage solutions. System should be designed to work when some parts of systems would be in fault or unreachable state.

Clouds size is growing every year. Needs to store data increases at the same rate. When data sizes are in petabyte size it becomes critical that storage solution is efficient to be able to store the data at generation speed. Different application uses storage in different ways and in many cases it is impossible to make one solution that would be suitable for all applications. It is necessary to know what storage solution suits best for specific use case.

As hardware failures in cloud systems are common, data should be stored in multiple different locations to ensure that no data will be lost even some physical devices would break. This data replication will increase used disk space many times if data replication is done just by replicating all data to multiple devices. Erasure coding can be used to reduce used disk space, but it will need for more CPU usage to calculate coding and recalculate data from coded blocks. With high IOPS that can be impractical.

Cloud storages can be divided in different groups depending data accessing rate. Hot storage has a high data I/O throughput and it is typically the main storage solution. There latency is usually tens of millisecond or even less. [59] Cold storages are solutions where data will be accessed rarely. Typical example is backups what are accessed seldom. There latency can be from couple of seconds to hours or even days depending storage solution. Cold storages are mainly used due their low price per TB of data compared to hot storage. [42] Warm storage have been designed to be solution between hot and cold, as it has low access latency as hot storage, but with lower I/O rate. [59]

3.1 Object Storage

In Object Storage data can be stored in variable sizes objects [3]. Typically objects contain data itself and meta data related to object properties. Mainly Object Storages are designed to store big files from tens of megabytes to multi gigabyte files, like images, videos and backups.

Swift and Ceph have been selected as examples as they are a open source and quite commonly used in enterprise solutions. Also these two cloud storages have different approaches as Swift is eventually consistent and Ceph is consistent storage solution. Main advantage for Swift from eventually consistency is that it can have lower latency

as replication can be done background as Ceph need to write all replicates before it can give acknowledge to client. On the other hand if Ceph has written data to disk it will be same for all clients no matter from what node the data is searched. Swift will synchronize the data between different replicates after the modification have be done. If another client tries to accesses same data and the object is read from another storage node with replicated data it is possible that those two replicated are not in synchronized state and another client got old information. So this have to be considered when designing applications using Swift.

3.2 Block Storage

Block Storage is storage solution type where data is stored in volumes which usually are spoken as blocks. The block will be presented for operating system as hard drive and it can be used as it would be physical device on that machine. In many cases file system is installed on top of the block device for better operability of various applications.

4 Openstack Swift Object Storage

Swift is open source project belonging to Openstack family. It is designed to be highly available, distributed an eventually consistent object store. It can be used through http API to store efficiently big files like videos or backups. [47]

4.1 Architecture

Openstack Swift is build from following parts: proxy servers, rings, zones, accounts, containers, objects and partitions. This chapter will cover architecture of Swift Object Storage.

Figure 4 describes Swift high level architecture. Beginning from left clients are users or applications what are using Swift restful API for accessing objects in the Object Storage. In real application there are usually load balancer between clients and proxy servers to evenly distribute load, but clients could also be directly communicating to proxy servers as presented in figure.

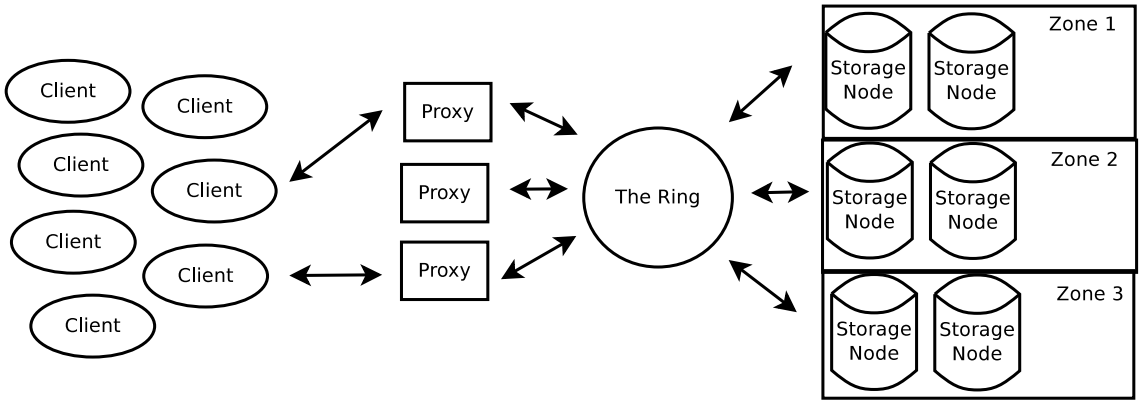

Figure 4: Openstack Swift architecture.

When the proxy server gets client request it will first validate if the user has valid access rights to the data it would be assessing. If everything is correct proxy will look from its local ring data structure which storage node would be responsible for that data. After founding responsible node it will try to access that node to execute the user request command. And finally the proxy will pass the traffic through it. The client will be only communicating with the proxy server and will never see the infrastructure behind it.

In default Swift configurations objects are stored as one blob to disk and only bigger files are sliced to multiple smaller objects. If the object is stored in parts manifest file will be created to contain information how the object is constructed. The actual data is then sliced in client side and named with a specific prefix then individual parts can be stored to cluster as any other object. When the sliced object is downloaded all parts can be loaded same time so read the performance can be

increased. If big object is created or modified the manifest file should be changed after all objects are stored to cluster to prevent loading partial objects. [44]

4.1.1 Proxy server

Proxy servers are used to handle incoming API request to Swift and route requests accordingly where data is located based on the information read from the ring. When objects are streamed to or from object servers, the stream goes through the proxy without any buffering. [48]

No data will be directly shared from proxy to another so if one proxy fails others can take it workload on the fly. Proxy servers are also responsible for making erasure coding and decoding, if erasure coding is enabled in system. When Proxy server is working as an interface to Object Storage, it needs to handle a large amount of errors. For example if client tries to write file but the responsible storage node is not available the proxy server will ask from the a ring for handoff server where to store the object. [48]

4.1.2 The Rings

The Ring is a data structure used to determine object locations in a Swift cluster. There are different rings for account database, container database and individual objects. All these three works in the same manner and they are maintained by manual application called ring builder. [43]

Ring builder can make the rings or update old ones. It will makes optimized Python structures and writes is to compressed, serialized file on disk, where it should be distributed to servers. Servers time to time checks modifications to their local copy and will load it to memory if it detects newer version of ring. As ring builder has clever way to rebuild rings, so servers with only slightly old version will only have one replicate in wrong location. [43]

Ring is structured in three parts. First structure has list of devices, where every list object is dictionary with following keys:

- *id* (*int*)
- *zone* (*int*)
- *weight* (*float*)
- *ip* (*string*)
- *port* (*int*)
- *device* (*string*)
- *metadata* (*string*)

This information can be used to locate physical storage devices in cluster. Second structure contains array of arrays, where there are one array for each replica. Internal arrays have the same length as there are partitions in ring. These arrays maps partitions to devices for all replicates. And last there are partition Swift value, used to modify default MD5 hash result. [43]

Ring builder needs cluster specific ring file with the ring information and additional information so it can update rings based on this file information. If ring file is lost it will mean that the ring have to be totally rebuild, which means that all data in cluster will be relocated.

There can be multiple object rings in one Swift cluster. This will allow that not all objects need to have same replication strategy. And because object rings specifies replication multiple rings are needed to enable multi replication strategy in one cluster. Those object rings can also have different devices so that container which needs fast data access can use drives with SSD and container with archive data can have hard drives with erasure coding to handle it data. [50]

Data policies are determined on container level so that many container can use the same policy, but once container is created with specified policy it can not be changed. If client wants to change container policy then only way is to make a new container with wanted policies and copy all data from old container. [50]

Handoff partitions are partitions in the Swift ring used to store data while some storage nodes are down. [48] So handoff partition is temporary storage that can be used to store any data during failure. Once old node have become up again or the ring is rebuild and the data is now located to working node the data is copied from the handoff node to correct node and after it the handoff copy can be removed. Then the handoff partition goes back to free handoff pool where it can be again assigned to new partitions if storage nodes go down.

4.1.3 Object, Container and Account servers

Object servers are simple servers responsible for save, read and delete objects stored to its local drive. Objects are stored in binary file format and metadata will be stored in files extended attributes (xattrs). So Swift requires underlying file system to support xattrs. [48]

Data is stored with version information to ensure that in any case newest data will be loaded. When data is deleted it is marked with tombstone, which is considered as file version, so that replication can not bring old version back due failure scenario. [48]

The container servers store listings of objects in containers. It does not store any information about object data or location. All those listings are stored in sqlite database. In data base is also stored information how many objects are in the container and how much storage space the container uses. Account server is similar as the container server but it stores listings of containers. [48]

4.1.4 Zones

Purpose of zones is to enable robust data distribution. Zones can be used to divide different failure zones from each other, so that when the data is replicated it will be located as widely as possible so that cluster can survive from big failures without losing any data.

Zones have a hierarchy so that largest are regions, then zones, then servers, and last drives. So when replication is planned, different replicates are stored in to different zones starting distribution from regions and going to smaller and smaller zones. Administrator can specify how zone borders goes. For example all server racks behind the same power line can be specified to be in same zone. [45]

4.2 Scalability

Cluster can be scaled in different ways in Swift. If number of proxies are increased ability to handle requests increases and if storage node count is increased then storage capacity grows as well as throughput. [57] However, all factors are needed to be increased when systems, otherwise unscaled parts can become bottlenecks. This strategy also gives flexibility as if query count is high but relatively low throughput is needed on storage nodes side it is only needed to grow the proxy node count to handle load.

4.3 Data replication

Swift is eventual consistent storage system. This approach will keep writing latencies low no matter how big system will grow. Client writing data will only need to wait approval from node where it is writing the data. Drawback will be that other clients can see the old state until replication is done if they are using a different node. [47]

In Swift, replication is not made on object level, but there are partitions which contains stored data, account and container databases. These collections of different objects are replicated as one whole part to different zones.

Replication is kept synchronous by replicators which continuously examine each partition for differences between different copies. Replicators will use hashes, that are made for all directories in partition to determine if data have been changed and is needed to be replicated. If modifications occurs the newest modifications will be resynced to other replicas. Only modified directories will be copied to another replicates and all changes in one partition will be send in once, which will reduce TCP overhead what could occur if many small TCP connections would be made. Replicators also handles the cleaning of tombstones, so it will ensure that when tombstones are removed then all old copies of object have been removed from entire system. [48]

For containers and accounts there is different replication strategy than objects. As data is stored in database the whole database synchronization can be checked with low-cost hash comparison. If difference have been found replicator will share

records added since last sync point. Sync point is last point when data bases have been known to be in sync so only records after it have to be checked. However, if whole database is missing then it is copied from another node with rsync(1). [49]

Swift replication strategy has bottleneck when number of object on one storage node increases over 1000 objects with three times replication. This is caused by periodical multi casting all objects to replicate servers to ensure consistency. So when number for broadcast objects increase over 1000 the network overhead starts to be massive and also the synchronization delay increases due high number of messages need to be send during multi casting.[53]

4.4 Consistency

Swift is eventual consistent object storage so it can not promise same view for all clients about current objects. However, Swift will promise to be consistent after time periodic known as consistency window [49].

This behaviour does not cause problems if objects are once written to storage and after it read. Then there is inconsistency only before the consistency window have been closed. This strategy also allows keeping low write latencies even different replicates have been distributed to different continents, as only the node where data is written need to ACK user.

4.5 Docker integration

Swift uses rest API for managing objects and containers. Modifications in Swift require always full rewriting of objects [51]. Therefore Object Storage is most suitable for data which is written once and modifications on those objects will require updating the whole object. Examples of this kind of objects are images and videos. Also Swift Rest API is one of supported Docker Registry storage back ends and can therefore be used for storing all Docker images in a cluster own Docker Registry [37].

Drawback of rest API is that storage interface have to be integrated in application or another service need to copy local object from local disk to Object Storage. However, the integration of Object Storage in application is straightforward.

Strength of Swift is that it can be used in multi data center solutions as the replication will not increase latency for user as it is done background and replication consistency is achieved after specific time window. Swift can be also be forced to make replicates to different data center, which can lower latency as client can read object from closest replica. This will also improve data resiliency as even losing one data center will not delete any data as at least one replica is on another data center. So for bigger dockerized clouds with multiple data center with need of cloud wide shared objects the Swift Object Storage is considerable option.

4.6 Hardware requirements

For all type of servers in Swift is recommended to have dual quad core processors. However, only the proxy server is really CPU intensive as all data is flowing thought them. Object, container and account servers should have 8 to 12 GB of RAM. Whereas for proxy server the RAM is not so important. [46]

Networking should construct from private and public networks where in private network is only for cluster internal communication and public network is only connected to proxy server. Network bandwidth should be from 1Gbits/s to 10Gbits/s. [46]

5 Ceph Object Storage

Ceph is a highly reliable distributed storage cluster with self-management and healing features. Ceph is designed to not have any single point of failure as any resource can be replicated on the fly.

Ceph cluster has three different storage solutions, Block Storage, Object Storage and file system, which can be running in same cluster simultaneously [9]. This chapter will only cover Ceph Object Storage and Block Storage. The CephFS distributed file system will be left outside of this scope.

5.1 RADOS Architecture

Ceph cluster is based on RADOS (Reliable Autonomic Distributed Object Store). Base components in the Ceph are Object Storage daemons (see OSD Section 5.1.4), Ceph monitors (see Section 5.1.5) and Ceph metadata servers, which is only used in CephFS and will not be covered in this work.

Object mapping is not made with distributed mapping, but locations can be calculated by using pseudo random CRUSH algorithm (see Section 5.1.1) and the global cluster map. The cluster map is maintained by Ceph monitors, which votes at current cluster map using Paxos protocol, that will ensure that modifications can be only made if majority of monitors agree the modification.

5.1.1 CRUSH Algorithm

CRUSH-algorithm (Controlled Replication Under Scalable Hashing) is pseudo random placing algorithm designed to remove need of centralized system to store and retrieve object locations in cluster. However, the CRUSH need current state of cluster to be able to calculate responsible OSD for the data. [1]

Also replication placement is done based on CRUSH. In cluster map (see at Section 5.1.2) can be determined how different failure domains are divided so CRUSH is able to calculate correct replicate placements by using this information. [1]

As CRUSH pseudo randomly distributes objects over the cluster it is crucial that all OSDs have some free space, as if even one OSD would be full it could cause situation where the CRUSH would place a new object in it and write would fail as there would be no free space. So in the Ceph all writes are prevented as long as even one OSD is full.

5.1.2 Cluster map

The Cluster map contains five different maps:

- *Monitor map*
- *OSD map*

- *PG map*
- *CRUSH map*
- *MDS map*

The monitor map contains information about all cluster monitors including monitors positions, addresses and ports. The OSD map has a list of OSDs and their statuses. The PG map has list of placement groups, their statuses and usage statistics. The MDS map is for metadata servers information. [9]

The CRUSH map itself has four main sections. First section of lists Object Storage devices which should mapped to OSDs. Second section contains bucket types which includes buckets which are used in CRUSH hierarchy (i.e. row, rack, host etc.). Third section list what host machines are in which bucket type. Last section determines the rules how buckets are selected. [4]

Each map contains iterative history of map modifications and indexes, so that map user can determine if it's local version of the map is out dated and needs to be update. If old and newer maps iterative histories overlap, the old map can be updated using only information from the newer map. Otherwise the user with older map needs to request newest map from monitor. [9]

5.1.3 Placement Groups

In Ceph all objects are mapped to placement groups (PG). This approach will make object tracking and replicating less expensive to compute as smallest trackable items in cluster are PGs. To balance cluster there should be about 100 PGs per OSD divided by replica count rounded up to nearest power of two. Rounding is optional, but it will ensure that all groups are approximately same size and this will help balancing load in the cluster. [6] In Figure 5 can be seen that objects are mapped to PGs which are then written to OSDs.

When scaling the cluster size, also the PG count should be adjusted, but if size is converted directly from one power of two to another, this would cause that half of the data should be replaced to new locations and cause huge performance drop in the cluster. To smooth this transaction it is possible to increase PG size in steps so that the object replacement can be distributed over longer time periodic. [2, p88]

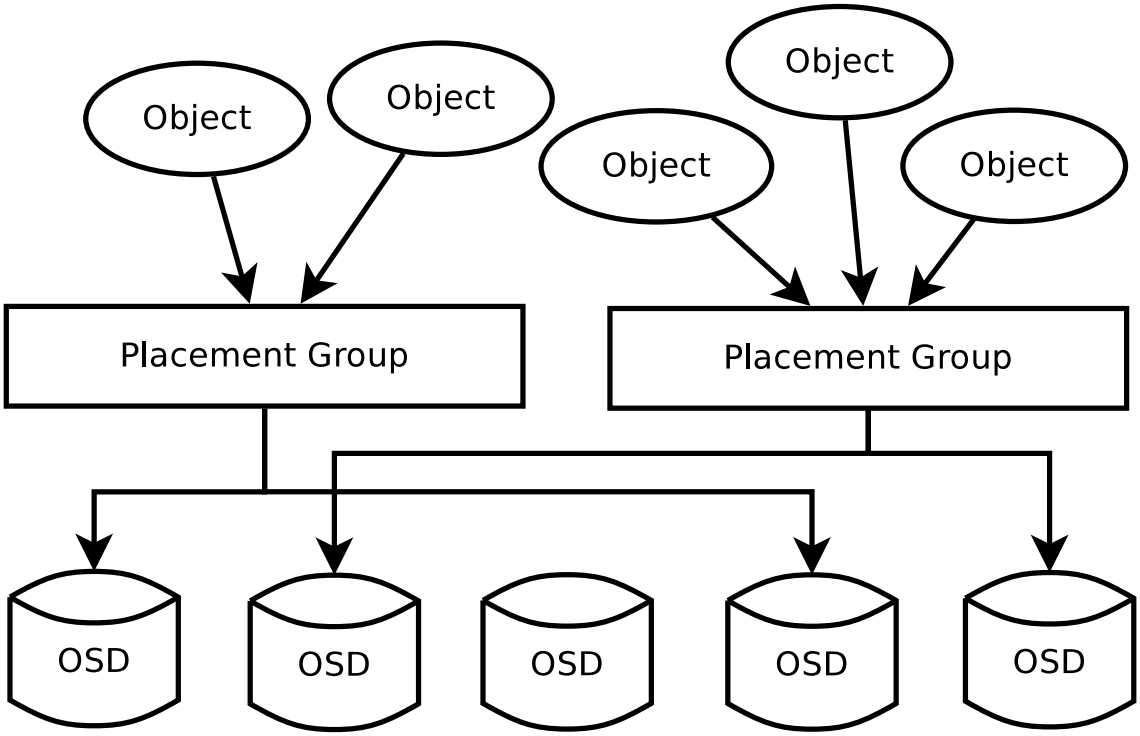

Figure 5: Placement group

5.1.4 Ceph OSD Daemon

Objects in Ceph cluster are stored in a flat namespace and every object has cluster widely unique ID to identify objects. Additionally for ID every object has data section where actual object data is stored and objects metadata in key/value pairs. Semantics of the metadata can be competitively be specified by Ceph client. [9]

When the Ceph client writes to OSD the replication will be done simultaneously by that OSD, according to replication rules. As in default configurations there will be two copies of every object, so when client writes to OSD, it will write same object to specific a OSD, determined by the CRUSH. The client will get acknowledgement for successful operation only after all OSDs where the object is written have informed about successful write. [9]

Journaling is used in OSD for two main reasons. First it will increase performance and latency for small writes, as the sequential journal write is only needed for retuning successful write for the client. It also gives ability to collect multiple object modification together and make only one bigger commit to the storage disk. Another advantage is increased consistency as the compound operations can be made atomically. Because changes are written to the journal, in case of failure the daemon can replay its journal to get the storage disk synchronized. [13]

In Ceph context journaling means that all modifications for objects are written to ring buffer file in serialized format. With serialized writing can be accomplice faster writing speed.

Storing the actual data to disk OSDs uses ext4, btrfs (B-tree file system) or XFS

file systems. The XFS is currently most recommended due its excellent stability, but btrfs will take its place when it will be proved to be stable. [12] In early stages Ceph used own EBOFS (Extent and B-tree based Object File System) [2, p162], but it was discontinued in later versions.

OSDs can use a cache to increase performance for most busiest files. There are two different cache types, if caching is used at all. First option is the writeback cache, where all read and write operations the data is added to the cache so that most recently assessed data is always in the cache. This will be suitable for mutable data as the data is updated in the cache. Time to time the cache will be flushed to the disk to apply changes. Second option is read-only cache where data is only loaded to cache when data is read. Now if objects are modified after they are loaded to the cache they will not be updated in the cache. So this option should be only used for the immutable data. [14]

In a big cluster, network problems are common and storage cluster should be able to detect errors and change it behavior based on the errors.That is why, OSD has heartbeat functionality which test connections to related OSDs.

By default OSDs send heartbeat messages with 6 second intervals and if another OSD will not answer for the heartbeat after certain time interval it will be marked as down by that OSD and one of monitors will be informed about it. When the monitor have got three times the OSD down message it will mark that OSD down and the cluster map will be updated. Not only OSDs send messages to each other but they also report their status to monitors within specified intervals. And if the OSD have not been reported it status for monitor in that specified time interval it will be removed from the cluster map. [11]

OSD can delay reads until write operation is done to disk if there is concurrently reader and writer on same object. With this approach the Ceph can keep data consistency even the OSD breaks during writing operation. As read will be made only after data have been written to disk, it ensures that all data given to client is on the disk. It will cause some latency to readers, but it is quite uncommon to have concurrent reads and writes for the same data. [2, p77]

5.1.5 Ceph Monitor

When the client need to write or read the data from the Ceph cluster it will need copy of the cluster map. It can be queried from one of Ceph monitors, which are responsible for keeping the cluster map updated. Map consensus is maintained with the Paxos protocol that will ensure that only when majority of monitors end up in consensus about map data, then changes to cluster map can be accepted. [9]

Monitors manages OSD, PG, monitor and MDS maps and they also provides authentication and logging services. All of these are in the same Paxos, so changes in any of them will trigger new Paxos round, but it will also gives ability get combine group of changes to one Paxos round. The data of the cluster map is stored to key/value database in every monitor. Those databases are only updated after successful Paxos round so all monitors will have synchronized databases. [5]

Changes to the cluster map will not be broadcast to OSDs but OSDs them self will

distribute it along messages are send. So if another OSD communicates with another OSD and finds out that it has older version of the cluster map it will automatically load the newer version from this another OSD. Meaning all changes are not needed to be applied to all OSDs if it wont affect them. [5]

All decisions needs majority of monitors to agree. The cluster will need at least three or more monitors before any redundancy can be provided. In order to get best efficiency, cluster should have odd number of monitors, because even number of monitors will just increase number of monitors but not the number of failed monitors cluster can handle.

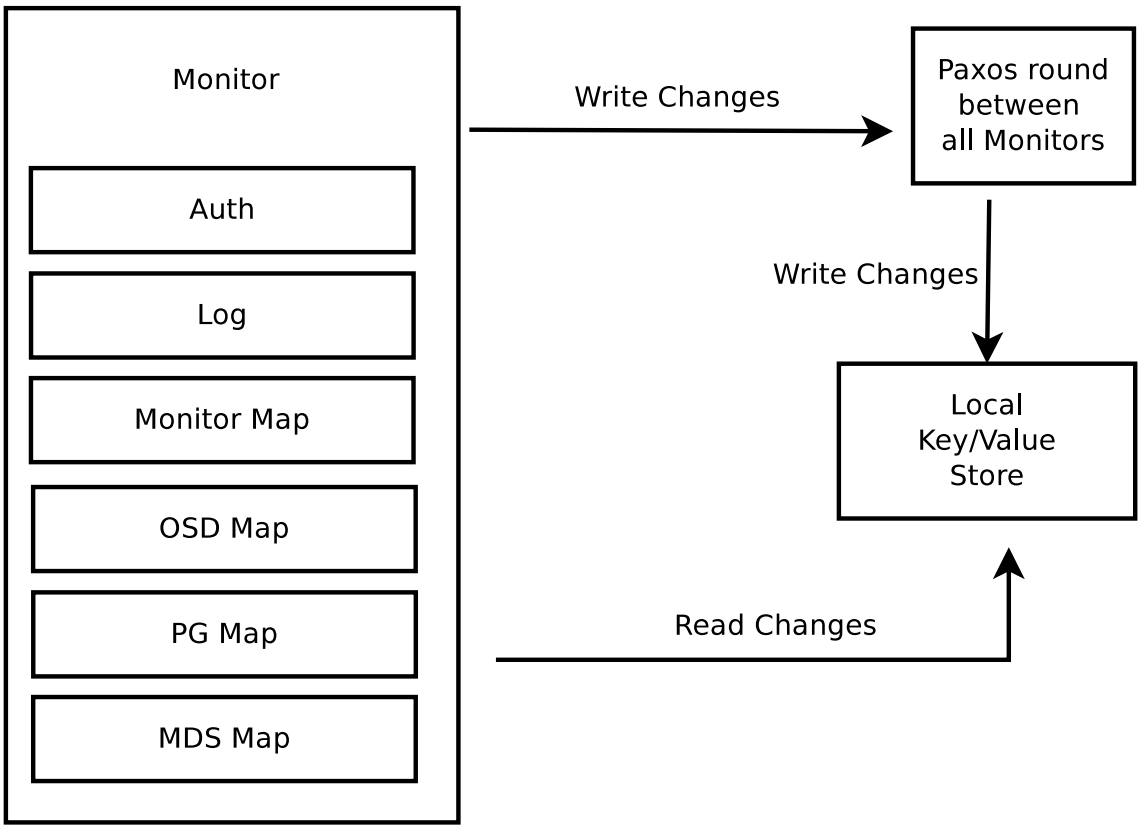

Figure 6: Monitor cluster state management.

Figure 6 shows architecture how reads and changes for ceph cluster maps are done. All data in Ceph monitors are stored in local key/value database. As all monitors have their own copy of that data structure it is important that all changes in it have to be globally accepted by majority. So when any of monitors need to change any data it will start new Paxos round and only after successful round all monitors can write changes to their local databases.

5.2 Ceph Object Storage Architecture

Ceph Object Storage is based on three different parts:

- *Object gateway*
- *Monitor*
- *Object storage daemon* (*OSD*)

The monitor and OSD are base part of the Ceph RADOS cluster. Object gateways are providing the API interface for clients to access and manipulate object stored in cluster. They also keep track of object buckets so that client can search objects stored to the Object Storage. More information about these parts can be found in following subsections, where each part will be covered in more detail.

When objects are stored to Object Storage, the Gateway will make mapping from client object to RADOS objects, so that header of file is one RADOS object and actual data is stripped to multiple objects. Client objects are stored in buckets so that the client can search for the object from the bucket. [10]

In Ceph Object Storage there can be multiple underlying Ceph RADOS clusters. All RADOS clusters in Ceph Object Storage share the same namespace so that there can not be two different objects with same name in different RADOS clusters. Object Storage objects can be written to any of those clusters. However, if object is written to one cluster it can be only read from there. So if request is made to wrong cluster it will be automatically redirected to the correct one. [10]

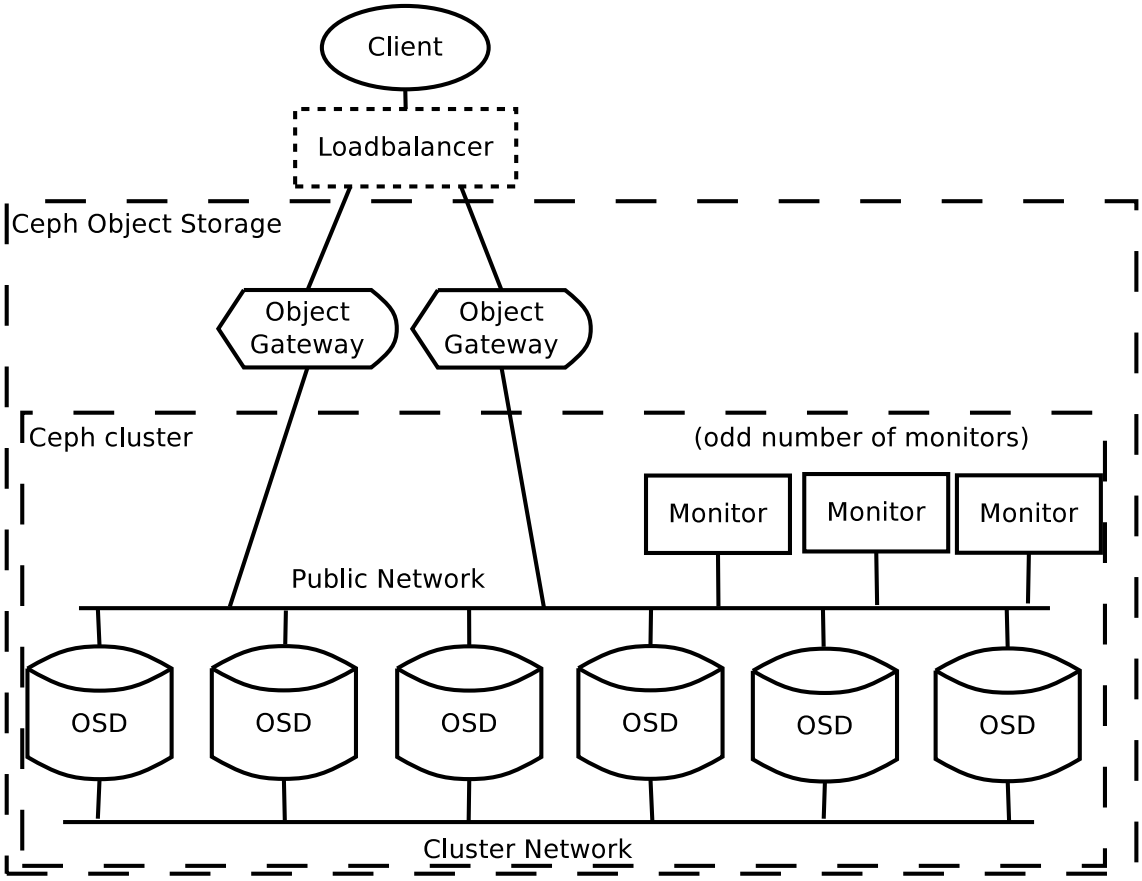

Figure 7: Ceph architecture. Load balancer is optional and will be external for Ceph cluster. OSDs are Ceph Object Storage daemons.

In Figure 7 are shown in high level the Ceph architecture in the Object Storage. Object Storage client will communicate directly to one object gateway or communication can go through external load balancer, which will select gateway. There gateway will communicate to the Ceph storage cluster. The actual cluster has group of OSDs (see Section 5.1.4) and odd number of monitors. Gateways will directly read and write RADOS objects to Ceph cluster. The cluster usually has at least two network interfaces where one is public network used by different Ceph clients and monitors to communicate each other and to OSDs. The another network is just for OSDs for moving objects from one OSD to another due replication or data replacement.

5.2.1 Ceph Object Gateway

Ceph object gateway is used by client for accessing Ceph Object Storage using either Amazon S3 or Swift APIs. Both APIs can be used simultaneously, for example one client writes object using S3 API and another reads it with Swift API. The object gateway have been built on top of the RADOS gateway, mainly just adding those third party API interfaces. The object gateway is also responsible for the storing

information how stored objects are mapped to RADOS objects, as the RADOS it self will not store any object mapping.

Object Storage maps client object to RADOS objects so that a single RADOS object will be used as the object head, containing the object metadata, it will be followed with the actual data which will be stripped to multiple RADOS objects if it will not fit in one RADOS object. These objects are then mapped to buckets specified by client. [15] By default buckets are single RADOS object which can cause performance issues if bucket is under heavy load or it has thousands or millions of objects stored in it. However, is it also possible to strip it to multiple RADOS objects to divide load. [15]

In data deletion of object or bucket, it will only mark it for removal, but data stays readable until it is purged. Admin should periodically run cleaning script to purge removed data. [10]

5.3 Ceph Block Storage Architecture

Ceph Block Storage (RBD) can be used with librbd or Linux rbd kernel module. The Ceph block device do not make difference for the usability from user perspective. The Ceph Block device uses underlying Ceph RADOS to store data in cluster and manage data replication. With this design there is no need for redesigning replication and data placement as existing RADOS can be used to help in it.

Ceph block device will not store any data on system, excluding possible caches. This will ensure that no data will be lost if data have been reported to be successfully written to disk. So even in situation where a system with mounted Ceph block device crashes or connectivity is lost, all saved data is in the cluster. This makes possible that the same block can be remounted to another host and block usage can be continued.

The *librbd* supports also write caching [16], which can break the philosophy that the data is never stored to the client side. The writecache can cause situation where data have not been written to cluster and if crash occurs before writes are flushed. However, write cache can increase RBD performance it selection between performance and data safety.

RBD supports snapshot so that it is possible to make snapshot of mounted block device. This is made possible by the RADOS watch notify messaging system. So that the one which is making snapshot can inform other about it. Informing others is important as snapshots means that all those snapshotted objects are now write protected. So any further writes have to go in copy-on-write manner, so that new objects are created when data is modified.

To operate block devices every RBD pool have following object types, where first four are metadata objects and last type is the actual data.

- *rbd_directory*
- *rbd_children*
- *rbd_id.* < *image_name* >
- *rbd_header.* < *image_id* >
- *rbd_object.* < *pefix_* < *id* >>

In *rbd_directory* store all name to id and vice versa information about all images in that pool. *rbd_children* object is store information whit images has clones. It will be used in situations where the image is tried to be deleted and before deletion have to be checked if any other image is cloned from this one. They can have shared objects and those dependencies have to be solved before deletion. Metadata object *rbd_id.* < *image_name* > is created for every image and it is used to map user defined name to real object id. All images have also *rbd_header.* < *image_id* > metadata object which has all information for the operating image. It has unique prefix used for naming images data objects, image size, list of snapshots and snapshot metadata and last locks held on image. [16]

Actual image data is stored in *rbd_object.* < *pefix_* < *id* >> RADOS objects. All data objects starts with same prefix and are numbered in increasing order as they seen in image by the user. Every data object has the same size and by default the configurations object size is defined to be 4MB as it is optimized size for accessing data in spinning disks [16]. Reason for uniform objects size is that it will reduce load variance of OSD which is important for the Ceph as even one full OSD will prevent any further writes to cluster [16]. Since the RADOS random object replacement for different size object could cause one OSD to get more bigger objects than others and cause it become full significantly earlier than others.

Snapshots can be made for any image in any time even if the image is in use. However, all I/O to the image should be stopped during snapshot to ensure the data consistency. [17] Snapshots are read-only and have copy-on-write behaviour so that if image is snapshoted any further writes will need to copy modified objects and make modifications to those new copies. Snapshot metadata is stored to the RBD header as mentioned previously. In every write, the list of snapshots is send so that writer is able to copy newest snapshot object and apply changes on that or if object have already been modified then changes can be made directly to that object.

As client using the image have to be able to know if the image have been snapshotted every client will use the RADOS watch-notify implementation to watch header object for changes [17]. When client makes a snapshot from image all clients watching that header will be notified [16].

5.4 Scalability

Ceph is designed to be easy to scale out by adding more hardware to cluster. RADOS throughput will scale linearly when adding new OSDs to cluster as long as network will not be saturated. [2, p. 148]

When adding new OSDs to Ceph cluster, number of PGs should be checked if it is suitable for that number of OSDs. Every time when a OSD is removed or added to the cluster, some data distribution will be done either to keep the number of replicas correct or to keep object distribution even. When scaling up number of monitors should not be increased excessively as all working monitors are voting for the cluster map changes and changes can be done only after majority of monitors have accepted voting. High number of monitors will make voting more complex and cause higher latencies. Also number of monitor should be odd to get best performance, as even number of monitors will just add one monitor without giving any benefits.

When scale of the Object Storage cluster grows, it can be practical to use multiple Ceph RADOS clusters instead only one. Clusters can be divided to different regions, when they do not share object data or they can be on different zones to increase durability. For every region there should be one master zone which will handle all write requests and slave zones will only be used for reading.

Between regions it is possible to make requests to any of object gateways, if requested object is located to in another region the client will be redirected to correct cluster hosting that object.

5.5 Data replication

Ceph can use different data replication rules for replication. The default replication rule is to save object and one additional copy of it. It can be other n-way replications or rules can have more specific definitions as three way replication so that data should be stored to two different data center. Also erasure coding is possible, but it will require more CPU resources than simple data cloning. Replication in the Ceph id made on pool level which enables different replication strategies for different pools. The whole cluster does not need to use the same replication strategy. [7]

When the client writes a data to the Ceph, it will be written to the node and this node will simultaneously write data to replicas and when write has been ended. The client will be informed success only after all copies have also reported success. In bigger system this can cause latencies if some replicas has higher latency. Different replication methods can also affect to write latencies. Client will get message about successful write after all replicas are written to their OSDs. Meaning that write latency is always determined by the slowest OSD part of the write.

5.6 Docker integration

Ceph Object Storage behaves similar to Swift so it also fits only for objects what should be written and modified as whole. Ceph design of delaying ACK for client until all replicates are finished write causes increasing delays as latency depends on slowest replicate. Then if different Ceph cluster parts would be in different data center it could cause high latencies and therefore lower throughput of whole cluster. So Ceph cluster should be located in only one data center. And therefore integrating

Ceph to multi data center dockerized cloud can need some design decisions how data is transferred from one cluster to another.

However, with Ceph Object Storage it is possible make multi data center storage as one Object Storage can be using multiple Ceph cluster for storing data. However, objects can only be stored in one Ceph cluster and replication is done inside that cluster. So even client accessing one data center is able to access all objects in Object Storage, objects located in another Ceph cluster in another data center can have higher latency as the query is needed to be done to that data center.

Ceph Block Storages are a really suitable for storing data in dockerized cloud if data is needed only by one client by time. As blocks can only be allocated for one client at time. However, one block can be cloned for multiple clients, but all modifications for that blocks are only visible for that client. Also the one data center limitation is applied to Block Storages due to the replication latency. Also as clients of Ceph Block Storage needs to be in same local network as client will be communicating directly to OSDs.

Strengths of Ceph Block Storage are that it can be mounted as any storage device, data is immediately stored and replicated to Ceph cluster and moving block from one host to another needs only remounting device on the new host and no data is needed to be transferred between host. So it fits fluently with containerized cloud where all resources should be flexible and never fixed to specific host machines.

5.7 Consistency

Ceph is always consistent storage system. It promises that when the ACK have been send back to the client, data is then stored to all replicas. [2, p. 130-132] Ceph also delays reads to objects if there is ongoing write operation to that RADOS object as long as the whole write have been accomplished [2, p. 142]. This ensures that data is always newest version of object and all clients will get same result no matter from which replicate the data is fetched.

In Ceph Object Storage objects are always consistent and objects are always read as whole even if they are simultaneously rewritten. This is made possible by making all Object Storage objects immutable and the object header is rewritten only after the object data and meta data have been added to storage [15]. When clients reads object that other client is rewriting it can read the old object as long as the object header have not been modified. When object header is modified it will be pointing to new data and as header is always only one RADOS object it is ensured to be hanged in all replicates simultaneously.

5.8 Hardware requirements

Different Ceph services need different resources for running fluently. OSDs will need minimum of two cores per daemon as they are constantly doing CPU intensive calculating of CRUSH-algorithm for the data replacement. In normal operation

OSDs are able to survive with only 500MB of RAM. However, in recovery they will need about 1GB of memory per 1TB of disk space. [8]

Monitors are much lighter as they only maintain the cluster map and do not need to make any CPU intensive tasks so single core it enough for them. However, as they should be able to give the cluster map in fast manner it should be kept all times in the memory so at least 1GB of RAM is required. [8]

For networking there should be at least two network interface controllers (NIC) connected to OSDs and one for monitor and gateway. OSDs would use one private NIC for data replication and another would be used as the public network for clients to communicate with OSDs. Network bandwidth for public network should be at least 1Gbits/s and the private network should have 10Gbits/s. [8]

The private OSD network should have higher bandwidth that the public network as writes will cause double traffic to the private network due to replication. Of course different replication strategies will cause different network load. However, in case of hard disk failure all data stored to that disk should be then replaced to new OSDs. So it is realistic that over 2TB of data is needed to be moved through network. It would take with 1Gbits/s network over four hours to accomplish it and with 10Gbits/s network time is reduced to under half an hour.

6 Testing setup

In this section will be discussed how actual tests where planed and setup for testing performance of systems. To be able to get reliable results it is important that tested systems have as equal configurations as possible and underlying hardware and VMs is same for both systems.

The test environment was first tested on Oracle VirtualBox virtual machines in local laptop before creating the real Openstack test environment. This was possible due Docker machine and Swarm which hide underlying virtualization structure so that for software both installations where similar. This allowed more efficient resource usage as only real test needed to be running in the Openstack system and functionality testing could be done in the local installation.

Test environment have been deployed on single OpenStack server with 32 vCPU and 63GB of RAM and with 314GB of disk space. Both Ceph and Swift are deployed in Docker containers running in Docker Swarm. Base image for VMs in Openstack was selected Ubuntu 14.04 cloud as most tools have been tested in Ubuntu 14.04 environment. Especially COSBench Object Storage benchmarking tool has some issues on different distributions of Linux and even with different version of Ubuntu. So to simplify deployment all virtual servers had same Linux image installed on them.

The high level architecture of the test cluster is shown in Figure 8. The Swarm cluster part contains six virtual machines where five of them are computational nodes where actual containers will run and last one is running the keystore based on Consul distributed database for distributing Swarm global state and configurations for computational nodes. These five VMs each have 2 virtual core, 4GB of RAM and 40GB of disk space. The keystore VM has only one core and 2GB of RAM.

For networking first idea was to use Docker Swarm own overlay [22] network driver as it has own DNS server for container discovery. So that no hard coded IPs would have been needed. However, turned out that it has quite low maximum data transfer capacity and the whole cluster could use only about 60MB/s network bandwidth, which would limit in best case cluster write speed to 20MB/s in only traffic going to storage nodes would go through Docker overlay network. And for read speed it would be limited to that 60MB/s. Those bandwidths where unacceptable as single VM to VM connection was capable of 900MB/s bandwidth. (more information at subsection 6.2). Because that all Swarm containers where using directly their host VM networking stack.

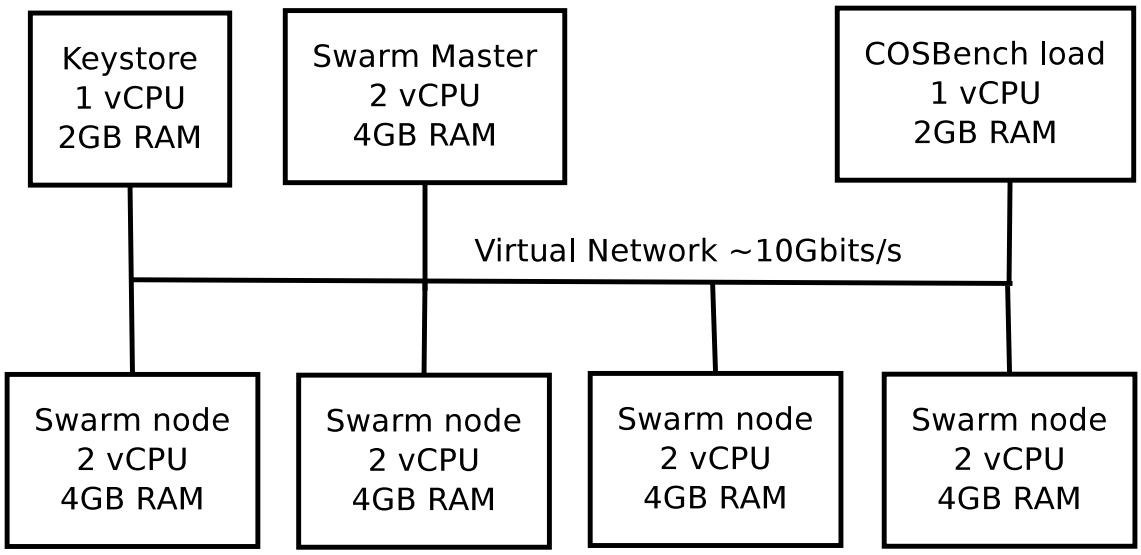

Figure 8: Openstack test environment virtual machine architecture. Network is divided for all VMs so that simultaneously connections will affect bandwidths.

6.1 Disk performance testing

Disk performance where benchmarked with flexible I/O tester [54], from now on fio. Following command where ran to execute test for single test case:

```
fio --name=test --filename=test --randrepeat=1 --ioengine=libaio
--direct=1 --size=1G --gtod_reduce=1 --iodepth=64
--readwrite=randrw --rwmixread=20 --bs=4k
```

In command the first row of command file location and name are set, then random seed for all runs is set to be same in all runs and last disk I/O engine is selected. On second row direct flag will remove cache from use then the size is just size of test file. Flag *gtod_reduce* reduces time stamp querying to 0.4% so it would not cause high CPU usage and possibly interference with measurement. Last on that row *iodepth* will set how many parallel file access are kept on file. In last row accessing is set random for both read and writing. Then read-write ration percent is set, so that 20 means 20% of reading and 80% writing. Last flag determines the block size used in accessing.

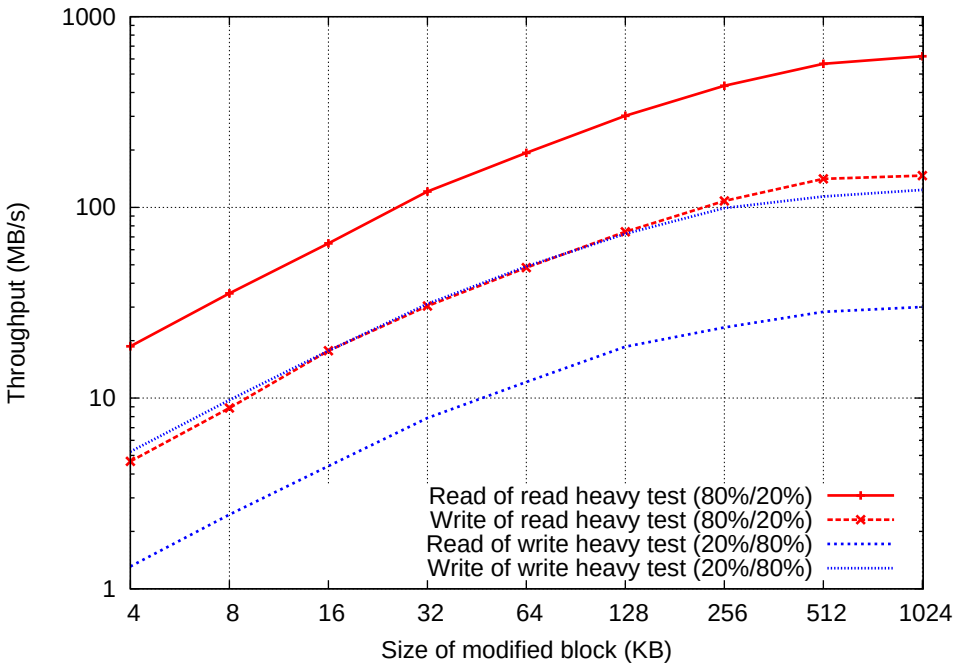

Figure 9: Virtual machine hard disk performance test. File on disk where random accessed with simultaneously read and write operations with 80% to 20% division.

In Figure 9 we can see that disk reading can get over 600MB/s for 1MB file size and writing is then limited to 130MB/s. From these figures three times replication should get to 43 MB/s as most likely limiting factor is disk write speed. For reading the speed can be even higher than disk speed as storage systems will be using cache. Over decade difference between accessing 4KB blocks compared to 1MB blocks is normal behaviour for random accessing.

6.2 Network performance testing

Testing network performance was used Iperf software. It have been designed to test maximum bandwidth of IP network using TCP or UDP protocols [56]. It also allows multiple simultaneously connections ran same time for both receiving and sending [56].

The network is tested between all VMs used in test excluding the keystore VM as it will have only small amount of traffic. In the test all expect two links reached a bit over 7Gbits/s (900MB/s) repeatable bandwidth. The one odd connection between two data VMs was randomly 4Gbits/s and other times 7Gbits/s. With the another connection bandwidth was only 3.3Gbits/s (about 400MB/s) in all tests. These

connections where only between nodes where storage services ran. So it can only partially limit the maximum replication throughput.

Second phase of the network testing was to simulate traffic caused by reading objects from Object Storages and writing it to them. Reading is simple to simulate as all nodes storing data will be streaming data to the gateway which will sends it to load VM. In Image 10 can be seen how write and read simulations will be configured. There arrow tip shows direction of stream and presents single or bidirectional depending if it has one or two tips. In writing situation the load VM makes one stream to the gateway which makes two streams for storage nodes. And both storage nodes make again two steams to other nodes to simulate replication writing. Those streams are selected as worst case so that nodes have uneven load distribution. In the real system all nodes should be getting evenly traffic, but these benchmarks should only give some rough estimates about network limits.

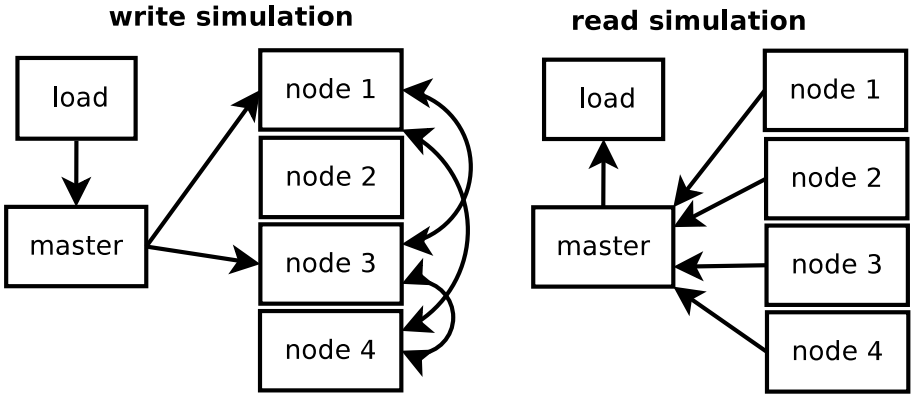

Figure 10: In left side of image is shown data streaming directions when simulating traffic while writing to Object Storage. Right side are stream directions when simulating reading from Object Storage.

For starting docker containers following command was ran:

```
docker run -d -e constraint:node==master --name a1 \
       -p 5001:5001 moutten/iperf
```

There $-e$ *constraint* : *node* == *master* flag will select the responsible swarm node what in this case was the *master* node. After it the container is named with cluster widely unique name and then $-p$ is used to map host port to container port. In this case it is just straight one to one mapping. Last *moutten/iperf* is just the name of the Docker image. The image will be automatically loaded to host from the Docker Hub if it is not found locally.

Test where ran so that in each VM had one container with Iperf server running in it. Test where executed with script which started Iperf clients on each node as specified in the test scenario. All clients did not start exactly same time, but test reliability where checked by starting clients multiple times in different order and running test for 60 seconds at time. All links behaved similar and different

start sequences did not change results. Also those two links with lower performance behaved similar as all other so those links most likely have only something that limits only the maximum throughput.

The network overall performance was that a single node could get average 2.5Gbits/s bandwidth in and out simultaneously. However, sum of all connection throughputs in network was in around 15-20Gbits/s depending how many simultaneously connections where in network.

Results will only give simple view of the network performance as they simplify a lot how the real traffic would be behaving. Like in the gateway situation it can be getting more traffic in that sending what would not be happening in real system as it can only pass so much traffic as gets and is able to send. However, reason for benchmarking the network is just to be able to detect if the network is limiting factory in the test environment. So based on these test can be said that the network is able to handle at least 3Gbits/s (almost 400MB/s) object reading traffic and about 2.5Gbits/s (300MB/s) writing traffic.

6.3 Setup for Swift cluster

In the Swift cluster there was one gateway server and four storage nodes. Each storage node has account, container and object servers running in one container. This approach will ensure good load distribution and easy scalability, as adding new storage nodes will increase count of all storage server types. Account and container servers are quite light services so they can be running along the more heavily used object servers.

The cluster setup starts by creating five identical containers to five different VMs in the Swarm cluster. Those containers base image is modified version of all-in-one Swift docker container [52], so that wanted services can be started after container start. All containers uses directly their host VM networking stack to keep network configurations as simple as possible.

When all five containers are started and finished initialization proses, their IP addresses can be collected for ring building script. The script is copied to the gateway container were it is executed. Then finished ring files are copied to all storage nodes.

Last node specific configuration files are copied to responding nodes and based on those configurations node can start correct services. After all nodes have started all needed services the storage cluster is now fully functional and it can be started to be used for testing.

6.4 Setup for Ceph cluster

Ceph setup was as similar as possible to Swift setup. This approach would reduce possible errors caused by different configurations. Main difference is that Ceph OSDs all have own journal disk mounted to RAM. Reason for using RAM mounted disk was that if Ceph would have used the disk to store journal it would have halved

disk write throughput as there would be six simultaneous writes as Swift would only make three simultaneous writes.

Fast journal virtual disk should only cause distorting to short writes as Ceph can write journal as long as disk is able to flush the same data to disk. In test every test configuration would be written 120 seconds where small 100MB journal will be easily filled and write speed is limited to the disk and Ceph algorithm performance and other factors.

The Ceph cluster has four OSDs, one monitor and one RADOS gateway. All OSDs are installed in Docker containers located in different VMs so that VM internal networking would not have effect on results. The both monitor and gateway are in same container running on different VM than any of OSDs. Mapping one VM to one container will ensure similar resource distribution for all nodes as every VM have individual resources, excluding the hard drive which is shared between all VMs as the host server only has one disk.

Ceph containers used in this work are modified versions of Ceph official demo container [18]. Setuping Ceph cluster starts by starting the monitor-gateway node. After it four identical OSD containers are started on different VMs in Swarm cluster. The monitor-gateway node has generated the Ceph configuration files in start up. Those configurations are copied for all OSDs so that they can be communicating with the monitor and by that way join in the cluster.

After all OSDs have copies of configurations, each of those will initialize their file systems for storage and journal. In this section there was some race conditions in Ceph which caused the OSD process to turn in to zombie and consuming 100% CPU. Then only way to rescue situation was to restart that VM and construct the OSD again and try if the file system construction succeed that time.

Once all OSDs have file systems working they can be added to CRUSH map and joined to cluster. Now the RADOS part of Ceph cluster is fully functional and Ceph RBD blocks could be created and used. To enable Ceph Object Storage user for Swift API needed to be created and after it the Object Storage Swift API was fully functional.

6.5 Test configuration for testing Object Storage

Intel COSBench benchmarking tool was used for all object storage tests in this paper. It have been designed for testing performance of various Object Storages. It has one controller which distributes load among driver nodes based on test configurations. It also provides web based graphical user interface for test result monitoring. Those drivers are just load generators and test setup can have multiple of them, but at least one is needed. In this test setup the storage cluster capacity was so low that multiple drivers would not have changed results as one driver was enough for saturating test system single hard disk.

As both Ceph and Swift supports Swift API it was used as the API interface in both systems. So changing from Ceph to Swift only required changing the API path from Ceph path */auth/swif/v1* to swift */auth/v1.0* to be able to run tests.

The clusters will be tested with different read write ratios. All tests used one object size at a time so that test started with smallest object size of 4KB and continued increasing power of two intervals to 1MB object size. There were 10 parallel workers loading the Object Storage in each test. In test cases there were 2500 objects which were divided to 50 containers with each having 50 objects. In combined read and write test object ids where divided so that objects which are read where in range of 1 to 25 and written objects where with ids of 26 to 50. So no single object where simultaneously read and written.

With objects size of 10M and 100M there were only 5 containers with 10 object in each container. They were similarly, as small objects, divided to two groups from object id 1 to 5 and 6 to 10. Reason for limiting big objects to only 50 object was that disk space would not allowed to write 2500 objects size of 100MB on disk as it would have needed 750GB of disk space with all three replicas.

Also one test scenario was to load cluster with Ziphian distributed random read accessing as it is quite common for data accessing [55]. But as COSbench allows only access percentages for objects and object ranges the Ziphian distribution is divided to ten different sequences.

The first object gets 12% of all traffic then second object to fifth gets 6, 4, 3 and 2 percent respectively. Then next 150 objects group gets 38%, second 150 objects group gets 18%, third 350 objects group gets 9%, fourth 650 objects group gets 5% and lastly 1250 objects group gets the last 3% of the load. The five first objects are included in the last 1250 object group. However, single object in last group gets only 0.0024% so this overlapping will not have any affect in final results as the rounding errors in are many times greater than the added load to those objects.

6.6 Test configuration for Block Storage

Block Storage should act as any mounted volume so it will be benchmarked with fio which was used previously for validating the VM disk performance. The mounted Ceph RBD size is defined to be 4GB and it will be using 12GB of disk space as rbd pool was configured to have three times replication. The RBD block was mounted to same load VM where COSBench was running, so that it has similar environment as Object Storage tests had. Kernel module rbd will be used to map the Ceph block to the file system. After mapping the block to the VM and initializing EXT4 file system on it. Then block was mounted to selected directory and Block Storage was ready for using.

Following test cases where ran for the Block Storage:

- Read-only (2GB file)
- Write-only (1GB file)
- Write 80% and Read 20% (1GB file)

All test cases where repeated with different block sizes starting from 4KB and increasing it power of two steps to 1MB block size. Data assessing will be random

over the whole test file. Read test has bigger file size as reading is so much faster so test will run only couple of seconds and results are not so reliable. Also tests where ran on three different environments. First test ran on natively on VMs disk, to see actual disk performance. After it same test where ran on the RBD mounded to VM and RBD linked to the Docker container.

It could have been possible that Docker container would have mounted RBD block inside itself, but it would have needed container to have full access to VM devices what would basically means same as VM mounting directly the RBD on disk. As same kernel modules does the mounting. And it is not good practice to give container full access to host devices as the container could then fault the whole system if some critical errors occur.

While testing read-only performance of RBD in the Docker container found out that some measurements got better results than from the native disk. It should not be possible as test in VM mounted RBD got similar results as native disk. So most likely some sort of caching in the Docker container where done as results were better.

7 Test results

This section introduces all test results made for benchmarking Object and Block Storages. All Object Storage test were ran for both Ceph and Swift using same the test arrangements and configurations as described in Sections 6.3 and 6.4. In couple of tests there was problem that cluster storage space ran out. This caused inaccurate test results for write tests with 10MB and 100MB objects as over 90% writes failed.

Solution for this problem was to reinstall the Swarm cluster as it could be done in under half an hour. There most time consuming task was to rebuild needed Docker images to all host VMs as this setup does not have own Docker repository for storing build images.

The reason for losing storage space over time was most likely caused by Docker volumes and images. In some cases Docker do not remove volumes linked to container when the container is removed. Then those ghost volumes just consumes disk space. Also when building new images old unnecessary image layers will stay in hard disk of host. There is scripts to remove all stored volumes and Docker images from machine but those usually really slow and unreliable.

After the Swarm cluster rebuild performance was quickly retested for possible change in cluster performance. However, no difference in host's performance was found by comparing results for previous measurements. Also some Object Storage test where ran for comparison of previous cluster. All the test gave similar results as with previous cluster so testing cloud be continued and all remaining test could be ran without further problems.

7.1 Object Storage test

Figure 11 shows read only performance of Swift and Ceph Object Storages. For reading have been used Zipfian distribution for emulating common object storage usage where only few objects get most of the queries. There can be seen that Ceph has over 5 times throughput compared to Swift when accessing 8KB objects. However, with 100MB objects the difference have been decreased to 1.8 times throughput for Ceph. With under 128KB objects average throughput difference is 4.6 times for Ceph. With bigger objects the average difference was 2.8 for Ceph.

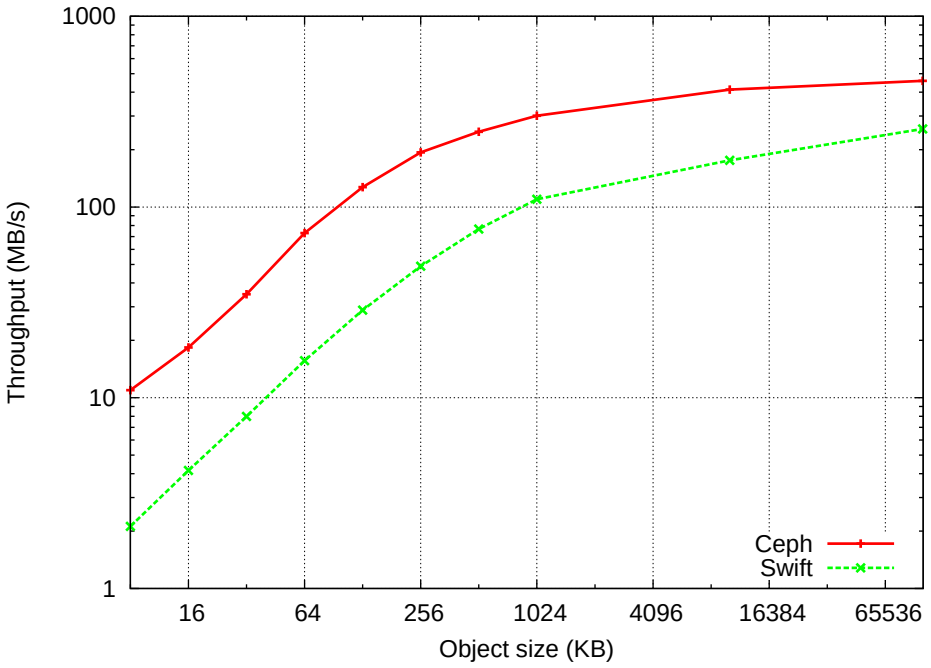

Figure 11: Object Storage read only test results with Zipfian distributed accessing. Single point in graph presents one test run.

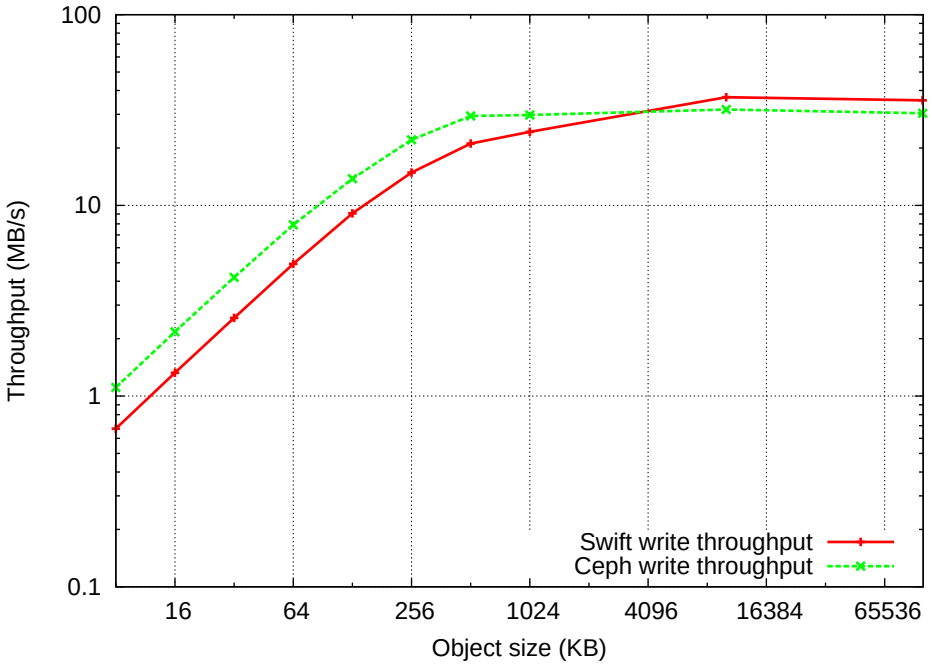

Figure 12: Random access writes evenly distributed over all objects in Object Storage.

Figure 12 shows the write-only performance of both Object Storages. With objects from 8KB to 128KB throughput difference was 1.6 times better for Ceph compared to Swift throughput. From 256KB to 1M objects the difference was 1.4 times for Ceph. However, with 10MB and 100MB objects Swift got 1.7 times better throughput compared to Ceph.

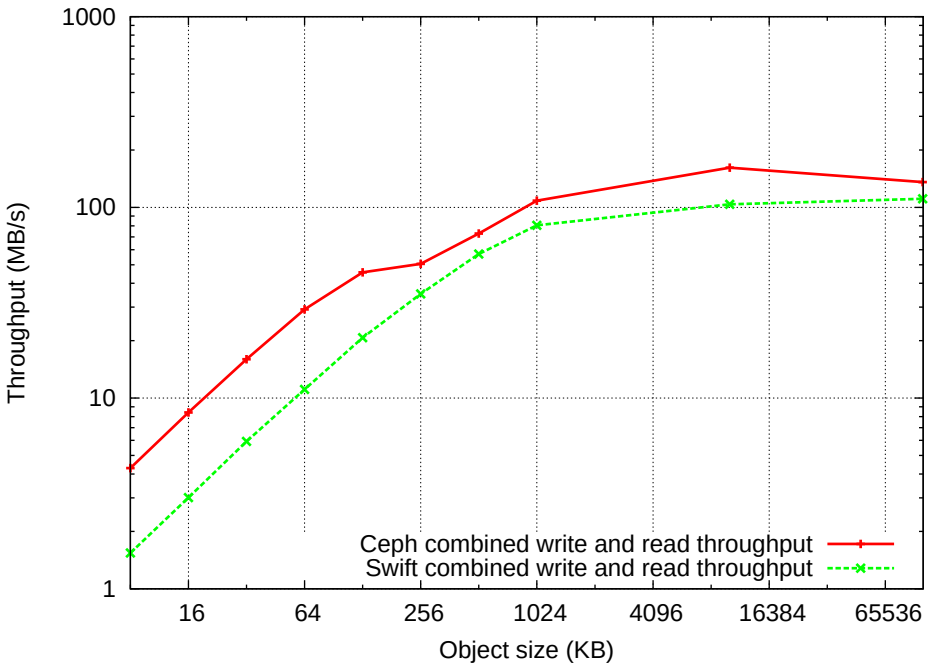

Figure 13: Random access reads and writes evenly distributed over all objects in Object Storage. From all operations 80% are reads and rest 20% are write operations.

In read heavy test read-write ration was 80% to 20% reading and writing respectively. From Figure 13 it can be seen that Ceph has average 2.6 times throughput compared to Swift when object size was from 8KB to 128KB. For bigger objects the average difference decreases to 1.4 times throughput for Ceph.

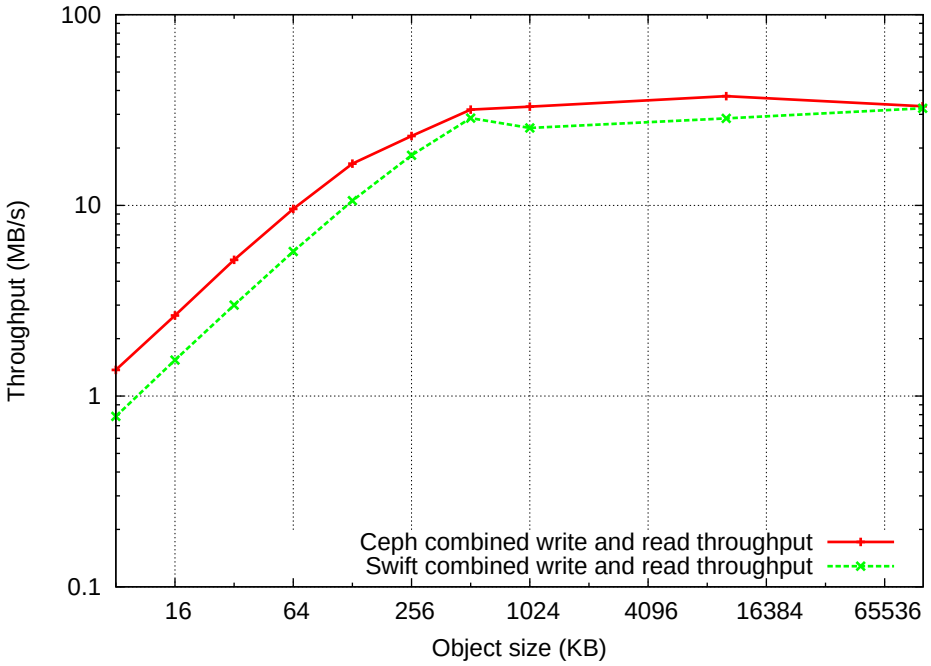

Figure 14: Random access reads and writes evenly distributed over all objects in Object Storage. From all operations 20% are reads and rest 80% are write operations.

Write heavy test read-write ratio was 20% to 80% reading and writing respectively. Figure 14 shows that average throughput difference was 1.7 times for Ceph when objects where 8KB to 128KB. For 256KB objects to 100MB objects the average difference was 1.2 for Ceph. With 100MB objects both system suffered lowered read throughput and for Swift only 78% operations where successful. There where not obvious reason for this behaviour and Ceph had always 100% success rate.

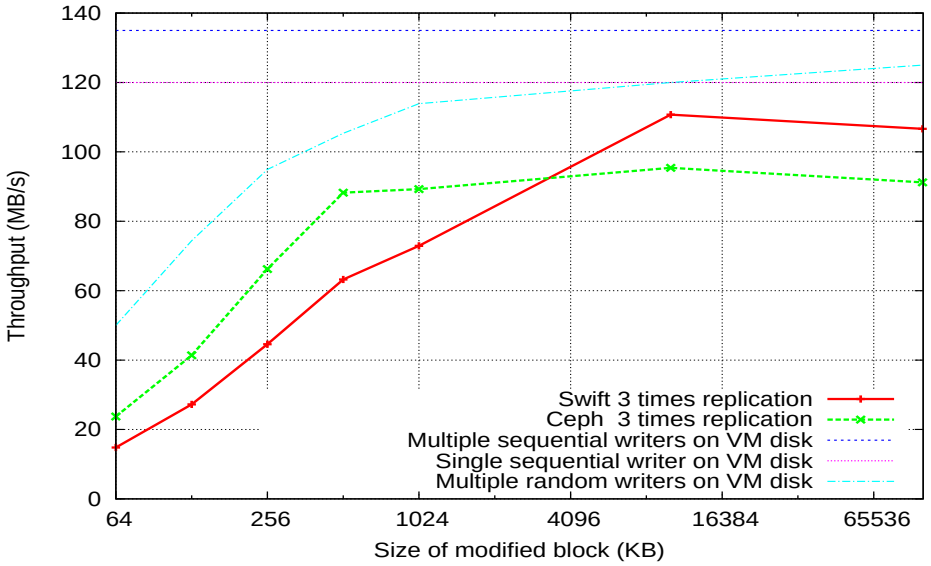

Figure 15: Approximated hard disk throughput of Swift and Ceph Object Storages when performing random access writes to the storage. Other three lines are representing the disk behaviour when loaded from VM.

Figure 15 shows how Ceph and Swift should be using the disk on write situation as three times replication will triple writes on the disk. The light blue curve is for comparison what would be the limiting factor of VMs hard disk if objects where written as that size of blocks. However, objects have some metadata and other factors what will effect on throughput efficiency of writing objects to storage compared to writing them directly on the disk. Other two lines are disk sequential write throughputs of VM hard disk. Violet line represent single sequential writer and dark blue line represent combined throughput of four parallel sequential writers.

7.2 Block Storage tests

Figure 16 shows difference in write throughput between VM hard disk and Ceph RBD. Due the three times replication RBD can even theoretically achieve one third of the throughput of the disk. However, at red dotted line shows approximated disk throughput what RBD was actually using to achieving its throughput. Comparing the VM disk throughput and RBD approximated throughput can be seen that RBD achieves 88.5% of the throughput of VM disk.

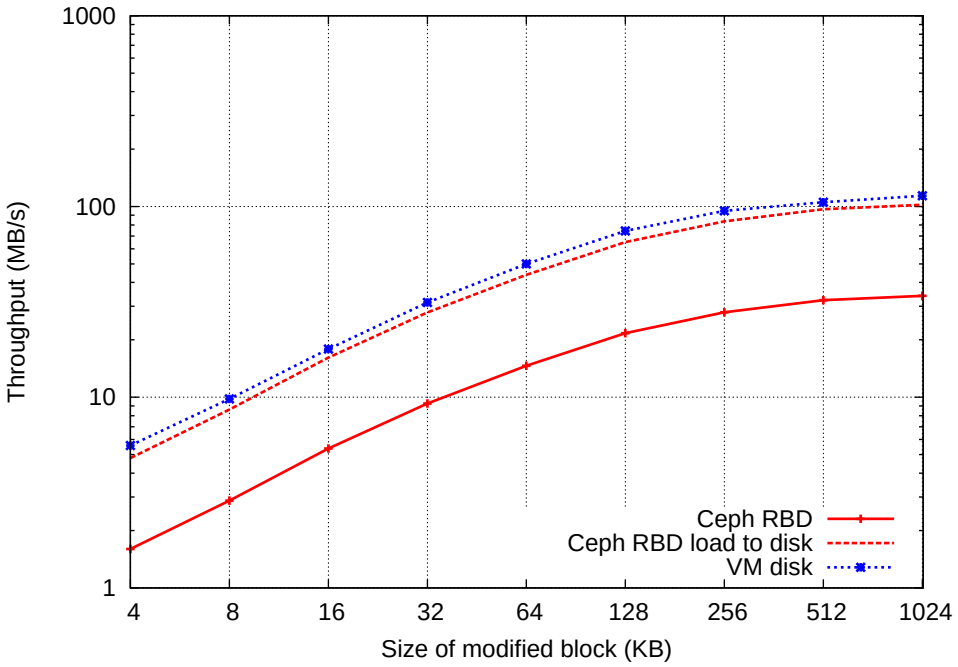

Figure 16: Block Storage write capacity compared to VM disk. The red line shows how much data in RBD have to write to disk because three times replication.

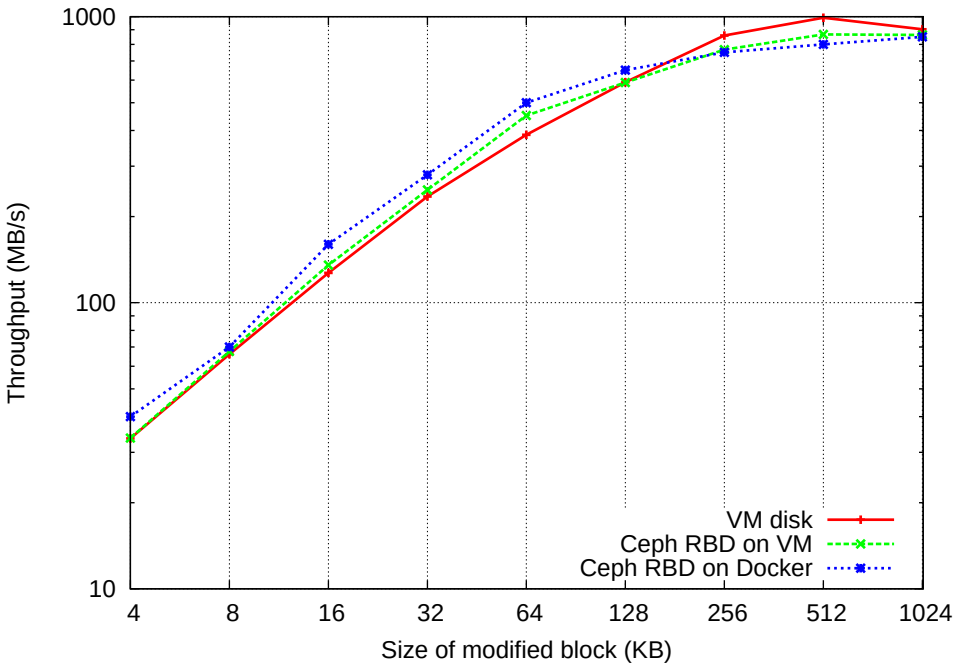

Figure 17: Read only performance for both VM disk and RBD. Read file was only 2GB cause limited disk space in test system.

In Figure 17 are compared read performances of VM disk, Ceph RBD in VM and Ceph RBD in Docker container. Main trend in this figure is that there is no significant difference between different configurations. However, the whole test block size was only 2GB and with bigger block sizes it meant that the test took only couple of second to execute so most likely some level caching was causing this high performance.

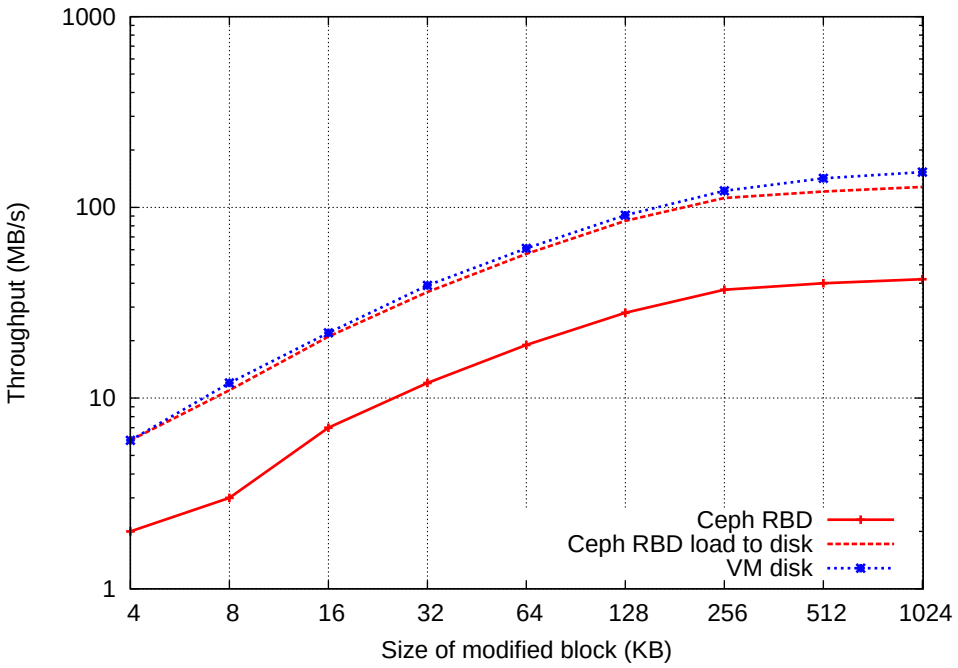

Figure 18: Mixed random access write (80%) and read (20%) results to RBD. Compared performance between RBD and VM disk.

In Figure 18 Ceph RBD performance with write heavy load is shown. Load is distributed so that 80% of operations to storage was write operations and 20% where read operations. Again the RBD throughput is about one third of disk throughput due the replication.

The approximated throughput of RBD have been directly calculated so that RBD throughput is just multiplied with three. Even read operations will not cause three times traffic the are not excluded from calculation as the test setup is limiting read throughput to be 20% of the whole throughput. Then if throughput of writes would increase it would also mean that throughput of reads should also increase. When compensating the replication the average throughput efficiency of RBD compared to VM disk was 92.2%.

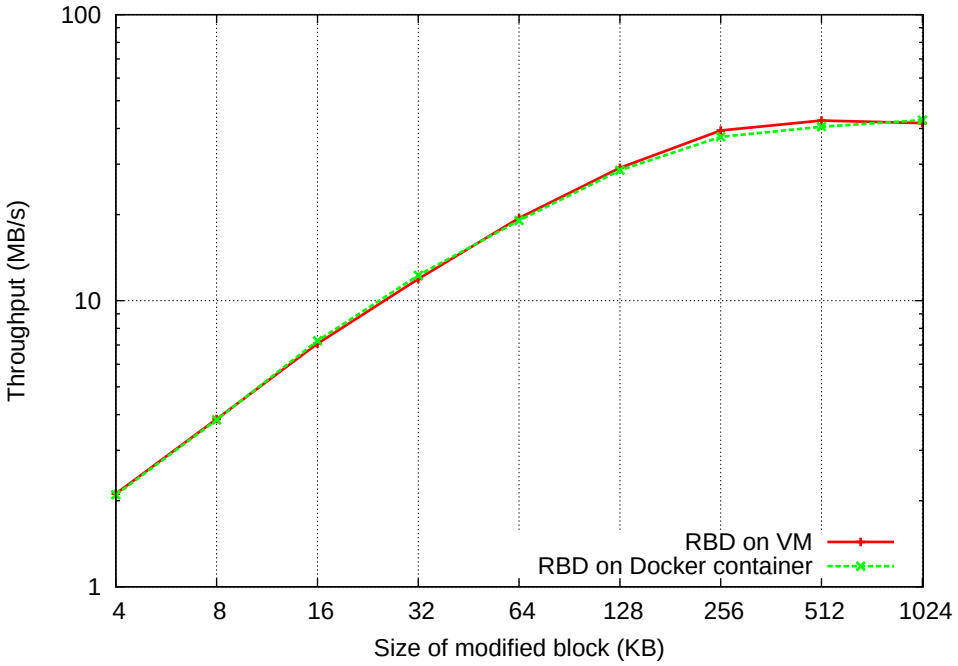

Figure 19: Random accessing write heavy testing (80% write and 20% read ration) on RBD directly mounted on VM compared to RBD block linked in Docker container.

In Figure 19 shows comparison in throughput between RBD block mounded in VM and RBD block linked to Docker container, when load is mix of 20% read operations and 80% of write operations. This test was made to show if there where any throughput reduction caused by linking RBD volume to Docker container. As the average difference between RBD in VM and it linked to Docker container was only under 1% can be said that it is inside natural variation.

8 Test results analysis

It can be concluded that Ceph has better over all performance especially with small objects. In bigger objects Swift gets closer to Ceph performance and in the big objects read-only test Swift was even faster. However, Ceph gives strong data consistency and it provides both Object Storage and Block Storage in the same packet.

In Figure 20 relative performance of Swift when Ceph is normalized to one is shown. There can be seen that difference between Sift and Ceph is constant form 8KB objects to 64KB objects. Then with bigger Swift starts to reach Ceph performance. Still winning Ceph only with write-only test with 10MB and 100MB object sizes.

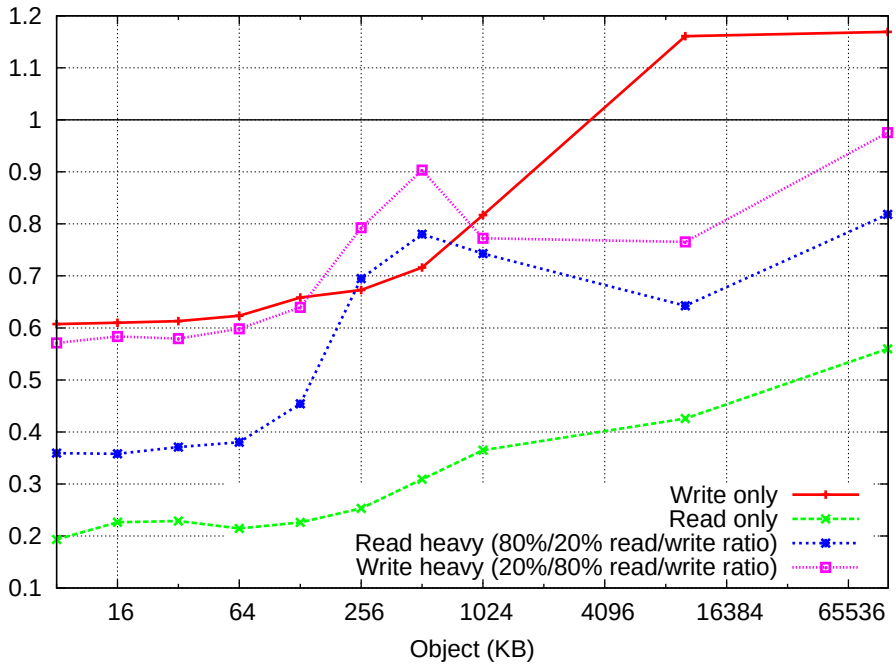

Figure 20: Relative performance of Swift when Ceph is normalized to 1.0.

The reason for the better performance of Ceph with writes with small objects is most likely related on its journalling strategy where everything is first written sequentially to the journal and every few seconds data from the journal is constructed to bigger writes what are then flushed to disk. Whereas in Swift every write is single write to disk which causes inefficiency on writing.

However, with bigger object Swift approach of writing objects sequentially to storages node, and not splitting them to many smaller objects as Ceph does, can partially explain why Swift reaches Ceph performance with bigger objects. By splitting Ceph causes many small random writes to disk which makes write operations less efficient. However, if There would be multiple hard disk this approach would also

distribute load more evenly over OSDs.

The big difference with small objects reading should not be related on journalling as it is only used for writing. So it is most likely related on how efficiently Ceph and Swift are handling queries. Cause Ceph have been written on C++ it should have many times greater performance than Swift as Swift is written on python.

Results in this thesis can not be straightly be scaled to bigger systems as there where only one hard disk so throughput had to be divided among all storage nodes. Also sharing hard drive caused more seeks on the disk as simultaneously access from multiple nodes. However, both systems had same configurations so characteristics should be similar in bigger installations.

In Block Storage side Ceph RBD showed good efficiency compared to VM hard disk. However, the test where made only using relatively small blocks which caused some unreliability on the results as internal caches can cause result to show better performance. Even in all test the Linux file system caches where bypassed the Ceph OSDs could cache most of the block content to their memory and give better throughput than with bigger blocks or with multiple users.

9 Summary and Conclusion

All three tested storage solutions are suitable solutions as persistent storage solutions for dockerized cloud. However, nature of data will have great impact on which storage solution will be most beneficial or should there be multiple storages in use for different type of data.

Objects Storages are used with rest API and they are generally suitable for data which is written once and read many times or if the whole object is needed to be any case rewritten as objects storages do not allow partial writes on objects. Whereas Block Storage has better support for small writes and its is easier to be integrated with legacy applications as it behaves as any storage device. However, Blocks Storages will not support multiple simultaneously users.

When selecting between Swift or Ceph Object Storages the scale of cloud should be estimated and what kind of workload the storage needs to be handling. If cluster will be running in multiple data centers the Swift could be more straightforward solution as single Swift cluster can be running in multiple data centers. Whereas Ceph cluster is only efficient when it runs in one data center as write latency increases if latency between replicas increases. However, Ceph Object Storage can use multiple Ceph clusters so that every data center has own Ceph cluster where local objects are stored. This approach increases latency when client and object which client need are in different data center as all replicas are in that one Ceph cluster.

Administrator side of different system is also quite different as Ceph can automatically add and remove OSDs from cluster as they fail and recover. Whereas with Swift administrator have to manually remove storage nodes from cluster and update ring files in all nodes. However Swift has temporary storages for data of failed node so that object has always correct amount of replicates even some storage nodes would have failed.

Block Storages are straightforward storage service for applications as it can be used as any storage device. In Docker it is possible to mount block device on host and link it in container. Application inside container will not see any difference between linked block device and normal docker volume. When container have to be moved from one host to another the block device can be mounted on that new host and again linked to container.

This behaviour gives good data resilience as data is always stored in Ceph storage cluster so no data will be lost in case of host machine failure. However, drawback of Ceph Block Storage is that it can only have one client at time using it. This means that block device can not be used as such as a data sharing service. If data is need to be shared the application have to handle sharing or it has sharing service beside it.

In tests Ceph Object Storage showed better performance compared to Swift. Only test case where Swift outperformed Ceph was in write only test with 10MB and 100MB objects. Generally Ceph has 1.7 to 2.6 times throughput compared to Swift when object size was from 8KB to 128KB. With bigger object Ceph throughput was from 1.2 to 1.4 times Swift throughput.

Block Storage test showed that Ceph has high efficiency of using disk throughput capacity. While writing, the Ceph Block Storage was able to utilize 88.5% disk

throughput compared to VM disk. On reading side no difference were found. However, the test block was only 2GB which caused read to be cached, even Linux file system cache was by passed. Also test showed that linking block to Docker container will not cause significant difference on disk throughput compared if block was used directly on host where it was mounted.

Conclusion based on tests and resources was that Ceph is always better option if Cluster is running only in one data center or applications and their data can be located in only one data center. Also it has more advanced fault recovery and supports both object and block storage. However, if objects are needed simultaneously in multiple data centers, Swift can be a better solution as it is able to distribute replicas to different data center without performance losses.

References

[1] Sage A. Weil Scott A. Brandt Ethan L. Miller Carlos Maltzahn *CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data* Santa Cruz, Storage Systems Research Center University of California, 2006.

[2] Sage A. Weil *CEPH: Reliable, Scalable, and High-performance distributed storage* Santa Cruz, University of California, 2007.

[3] *Object Storage: A New Approach to Long-Term File Storage Object Storage* A Dell Technical White Paper. Dell Product Group. 2010 http://www.dell.com/downloads/global/products/pvaul/en/object-storage-overview.pdf [Accessed: 27.11.2015]

[4] *CRUSH Maps — Ceph Documentation* Inktank Storage, Inc. http://docs.ceph.com/docs/hammer/rados/operations/crush-map/ [Accessed: 25.11.2015]

[5] *Monitor Config Reference — Ceph Documentation* Inktank Storage, Inc. http://docs.ceph.com/docs/hammer/rados/configuration/mon-config-ref/ [Accessed: 27.11.2015]

[6] *Placement Groups — Ceph Documentation* Inktank Storage, Inc. http://docs.ceph.com/docs/hammer/rados/operations/placement-groups/ [Accessed: 19.11.2015]

[7] *Erasure code — Ceph Documentation* Inktank Storage, Inc. http://docs.ceph.com/docs/hammer/rados/operations/erasure-code/ [Accessed: 19.11.2015]

[8] *Hardware Recommendations — Ceph Documentation* Inktank Storage, Inc. http://docs.ceph.com/docs/hammer/start/hardware-recommendations/ [Accessed: 3.3.2016]

[9] *Architecture — Ceph Documentation* Inktank Storage, Inc. http://docs.ceph.com/docs/hammer/architecture/ visited 27.11.2015

[10] *Ceph Object Gateway — Ceph Documentation* Inktank Storage, Inc. http://docs.ceph.com/docs/hammer/radosgw/ [Accessed: 19.11.2015]

[11] *Configuring Monitor/OSD Interaction — Ceph Documentation* Inktank Storage, Inc. http://docs.ceph.com/docs/hammer/rados/configuration/mon-osd-interaction/ [Accessed: 1.12.2015]

[12] *Hard Disk and File System Recommendations — Ceph Documentation* Inktank Storage, Inc. http://docs.ceph.com/docs/hammer/rados/configuration/filesystem-recommendations/ [Accessed: 16.2.2016]

[13] *Journal Config Reference — Ceph Documentation* Inktank Storage, Inc. http://docs.ceph.com/docs/hammer/rados/configuration/journal-ref/ visited 24.11.2015

[14] *Cache Tiering — Ceph Documentation* http://docs.ceph.com/docs/hammer/rados/operations/cache-tiering/ visited 16.3.2016

[15] Yehuda Sadeh *Ceph Tech Talks: RGW, 26.2.2016 in Ceph tech talks* 2016 https://www.youtube.com/watch?v=zvfv3pXq0Ww&list=PLrBUGiINAakM36YJiTT0qYepZTVncFDdc&index=1 visited 27.11.2015

[16] Josh Durgin *2015-FEB-26 – Ceph Tech Talks: RBD* 2015 https://www.youtube.com/watch?v=qA-Xqu2hCqU visited 3.3.2016

[17] Inktank Storage, Inc. *Snapshots — Ceph Documentation* http://docs.ceph.com/docs/hammer/rbd/rbd-snapshot/ visited 3.3.2016

[18] *ceph/ceph-docker: Docker files and images to run Ceph in containers* https://github.com/ceph/ceph-docker visited 4.2.2016

[19] Drew Roselli Jay Lorch Tom Anderson *A Comparison of File System Workloads.* In Proceedings of the 2000 USENIX Annual Technical Conference San Diego, CA, June 2000. USENIX Association

[20] *kubernetes/kubernetes: Container Cluster Manager from Google* https://github.com/kubernetes/kubernetes [Accessed: 26.11.2015]

[21] Wes Felter Alexandre Ferreira Ram Rajamony Juan Rubio *IBM Research Report: An Updated Performance Comparison of Virtual Machines and Linux Containers* IBM Research Division Austin Research Laboratory, 11501, Burnet Road Austin, TX 78758, USA, 2014

[22] *Docker container networking* https://docs.docker.com/engine/userguide/networking/dockernetworks/ [Accessed: 8.3.2016]

[23] *Docker Glossary* https://docs.docker.com/v1.8/reference/glossary/#union-file-system [Accessed: 22.11.2015]

[24] *What is Docker?* https://www.docker.com/what-docker visited 16.11.2015

[25] Thanh Bui *Analysis of Docker Security* Aalto University School of Science. 13 Jan 2015

[26] *Docker security* https://docs.docker.com/engine/articles/security/ [Accessed: 27.11.2015]

[27] Diogo Mónica *Introducing Docker Content Trust | Docker Blog* 2015 https://blog.docker.com/2015/08/content-trust-docker-1-8/ [Accessed: 27.11.2015]

[28] *Docker Swarm* https://docs.docker.com/swarm/ [Accessed: 21.11.2015]

[29] *Discovery | Docker Swarm* https://docs.docker.com/swarm/discovery/ [Accessed: 8.3.2016]

[30] Andrea Luzzardi *Announcing Swarm 1.0: Production-ready clustering at any scale | Docker Blog* 2015 https://blog.docker.com/2015/11/swarm-1-0/ [Accessed: 22.11.2015]

[31] *Understand the architecture | Docker* https://docs.docker.com/engine/understanding-docker/ [Accessed: 26.2.2016]

[32] *Manage data in containers | Docker* https://docs.docker.com/engine/userguide/containers/dockervolumes/ [Accessed: 26.2.2016]

[33] *Docker 1.9 multi host networking.* https://blog.docker.com/2015/11/docker-multi-host-networking-ga/ [Accessed: 26.2.2016]

[34] *Docker Engine 1.10 Security Improvements* by Docker Core Engineering, 2016 https://blog.docker.com/2016/02/docker-1-10/ [Accessed: 26.2.2016]

[35] *Machine Overview | Docker* https://docs.docker.com/machine/overview/ [Accessed: 26.2.2016]

[36] *Docker run reference | Docker* https://docs.docker.com/engine/reference/run/ [Accessed: 14.3.2016]

[37] *Deploying a registry server | Docker* https://docs.docker.com/registry/deploying/ [Accessed: 21.3.2016]

[38] *Docker container networking monitoring - Weaveworks* https://www.weave.works/ [Accessed: 26.2.2016]

[39] Bukhary Ikhwan Ismail Ehsan Mostajeran Goortani Mohd Bazli Ab Karim Wong Ming Tat Sharipah Setapa Jing Yuan Luke Ong Hong Hoe *B. I. Ismail et al., "Evaluation of Docker as Edge computing platform,"* Open Systems (ICOS), 2015 IEEE Confernece on, Melaka, 2015, pp. 130-135. doi: 10.1109/ICOS.2015.7377291

[40] *Storage solutions & data volume manager for Docker | Flocker ClusterHQ* https://clusterhq.com/flocker/introduction/ [Accessed: 3.3.2016]

[41] *Flocker Cluster Architecture* https://docs.clusterhq.com/en/latest/flocker-features/architecture.html [Accessed: 3.3.2016]

[42] *Google cloud storage nearline* Google Cloud Platform https://cloud.google.com/files/GoogleCloudStorageNearline.pdf [Accessed: 26.11.2015]

[43] *The Rings — swift 2.6.1.dev261 documentation* OpenStack Foundation http://docs.openstack.org/developer/swift/overview_ring.html [Accessed: 16.11.2015]

[44] *Large Object Support — swift 2.6.1.dev261 documentation* OpenStack Foundation http://docs.openstack.org/developer/swift/overview_large_objects.html [Accessed: 8.3.2016]

[45] *Object Storage zones - OpenStack Configuration Reference - kilo* OpenStack Foundation http://docs.openstack.org/kilo/config-reference/content/swift-zones.html visited at 26.11.2015

[46] *System requirements for Object Storage - OpenStack Installation Guide for Red Hat Enterprise Linux 7, CentOS 7, and Fedora 20 - juno* OpenStack Foundation http://docs.openstack.org/juno/install-guide/install/yum/content/object-storage-system-requirements.html visited at 27.11.2015

[47] *Welcome to Swift's documentation! — swift 2.6.1.dev261 documentation* OpenStack Foundation http://docs.openstack.org/developer/swift/ [Accessed: 22.11.2015]

[48] *Swift Architectural Overview — swift 2.6.1.dev261 documentation* OpenStack Foundation http://docs.openstack.org/developer/swift/overview_architecture.html [Accessed: 25.11.2015]

[49] *Replication — swift 2.6.1.dev261 documentation* OpenStack Foundation http://docs.openstack.org/developer/swift/overview_replication.html [Accessed: 9.3.2016]

[50] *Storage Policies — swift 2.6.1.dev261 documentation* OpenStack Foundation http://docs.openstack.org/developer/swift/overview_policies.html [Accessed: 1.12.2015]

[51] *OpenStack API Documentation* OpenStack Foundation http://developer.openstack.org/api-ref-objectstorage-v1.html [Accessed: 21.3.2016]

[52] *bouncestorage/docker-swift: Docker image for Swift all-in-one demo deployment* https://github.com/bouncestorage/docker-swift [Accessed: 4.4.2016]

[53] Thierry Titcheu, Ennan Zhai, Zhenhua Li, Yong Cui, and Kui Ren *On the Synchronization Bottleneck of OpenStack Swift-Like Cloud Storage Systems* 35th IEEE International Conference on Computer Communications (INFOCOM'16), Apr, 2016.

[54] *fio(1): flexible I/O tester - Linux man page* http://linux.die.net/man/1/fio [Accessed: 25.2.2016]

[55] Lada A. Adamic Bernardo A. Huberman *Zipf's law and the Internet* Glottometrics 3, 2002,143-15

[56] *iPerf - The TCP, UDP and SCTP network bandwidth measurement tool* https://iperf.fr/ [Accessed: 24.2.2016]

[57] *The OpenStack Object Storage system. Deploying and managing a scalable, open-source cloud storage system with the SwiftStack Platform* By SwiftStack, Inc

[58] Abhishek Verma Luis Pedrosa Madhukar Korupolu David Oppenheimer Eric Tune John Wilkes *Large-scale cluster management at Google with Borg* Google, Inc.

[59] Subramanian Muralidhar Wyatt Lloyd Sabyasachi Roy Cory Hill Ernest Lin Weiwen Liu Satadru Pan Shiva Shankar Viswanath Sivakumar Linpeng Tang Sanjeev Kumar *f4: Facebook's Warm BLOB Storage System* Facebook Inc, University of Southern California, Princeton University

[60] *The Tail at Scale* Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. Commun. ACM 56, 2 (February 2013), 74-80. http://dx.doi.org/10.1145/2408776.2408794