CS 6343

Cloud Computing Project final report

Group: A1-1

Group Member:

Tong Xin

Tian Chenzi

Contents

Introduction

Goal of the project

This project includes three core goals which are installation of project infrastructure architecture, evaluate the trending distributed storage system like HDFS, CEPH and Swift by developing own test tool suite and optimize the current OSD cluster map data structure by adding extra virtual node layer.

Study of related work

Summary of related works

Study area	Study material source
HDFS	http://hadoop.apache.org/
HDFS	K. Shvachko, Hairong Kuang, S. Radia, R. Chansler, "The Hadoop distributed file system," IEEE Symposium on Mass Storage Systems and Technologies, May 2010
Openstack	http://www.openstack.org/

Swift	http://docs.openstack.org/developer/swift/
Keystone	http://docs.openstack.org/developer/keystone/
Ceph	Ceph: A Scalable, High-Performance Distributed File System
Ceph	CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data
Ceph	http://ceph.com/
CosBench	https://github.com/intel-cloud/cosbench
Openstack API	http://www.openstack4j.com/

How your project is different from or is similar to some existing works

1. Replace physical OSD node with virtual node plus mapping table in order to solve the load balance issue of Ceph.
2. Simulate frequent file I/O to specific OSD or Object Storage in CEPH and SWIFT to evaluation their performance under uneven traffic.
3. Integrate CEPH with Openstack Keystone as Authentication Entity.

Approach

System Architecture

Virtualization Architecture

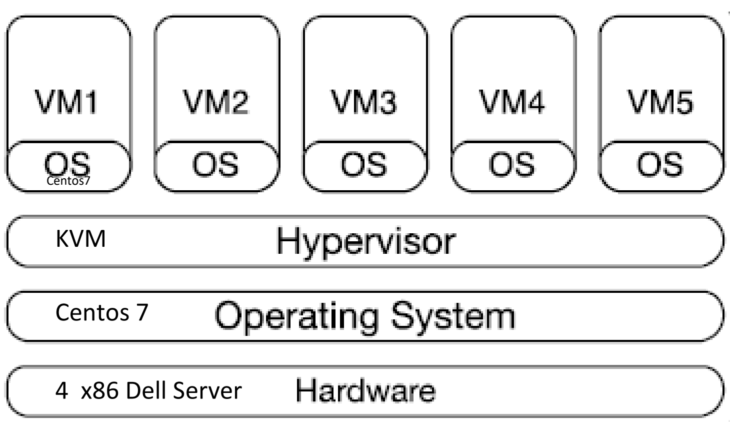


Figure 1 Virtualization Architecture

Guest VM allocation table on hosts

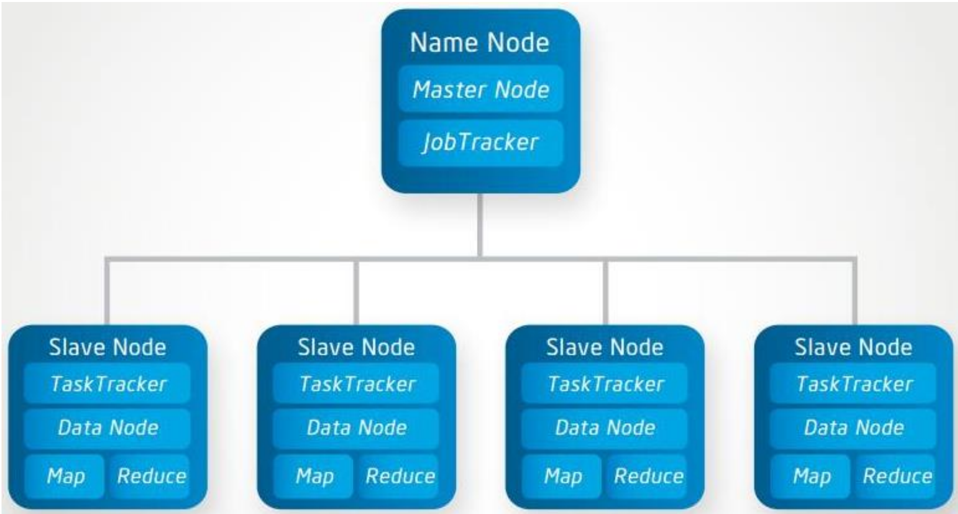
Host	Filesystem	Guest VM name	guest vm IP address
Host1	CEPH	deploy	192.168.1.124
Host1	CEPH	mon1	192.168.1.129
Host2	CEPH	osd1	192.168.1.108
Host3	CEPH	osd2	192.168.1.102
Host4	CEPH	osd3	192.168.1.109
Host1	CEPH	client	192.168.1.101
Host1	CEPH	mds1	192.168.1.104
Host1	CEPH	keystone-ceph	192.168.1.130
Host1	SWIFT	controller	192.168.1.131
Host1	SWIFT	proxy	192.168.1.133
Host2	SWIFT	object1	192.168.1.110
Host3	SWIFT	object2	192.168.1.112
Host4	SWIFT	object3	192.168.1.113
Host1	HDFS	namenode	192.168.1.121
Host2	HDFS	datanode1	192.168.1.120
Host3	HDFS	datanode2	192.168.1.119
Host4	HDFS	datanode3	192.168.1.118

Figure 2 Guest VM allocation table on hypervisors

HDFS Part

HDFS System Architecture

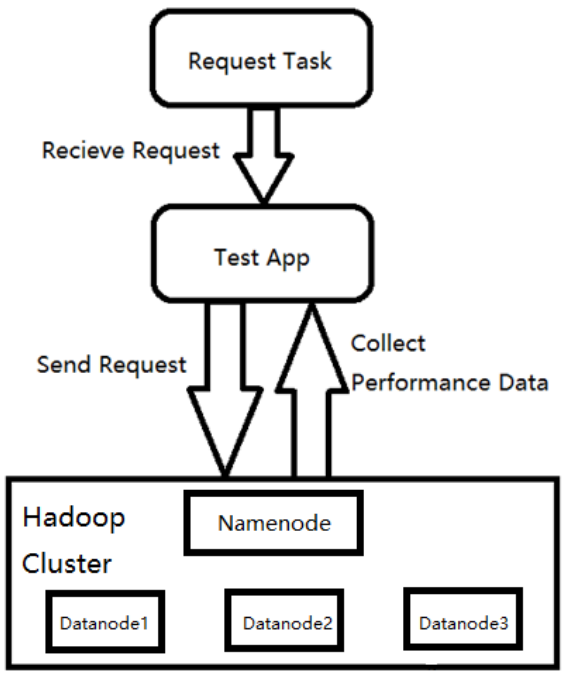
The following is the architecture our Hadoop cluster use. In our cluster, we have one namenode(namenode) and three datanodes(datanode1, datanode2, datanode3). We deploy the hdfs server, yarn server, Map-reduce history job server in out cluster.



Component	Hostname
Namenode	namenode
Datanodes	datanode1, datanode2, datanode3

Activity diagram

The following diagram shows how we test the HDFS cluster. We implement a test application which contains a request adaptor that can receive request sent from request generator and send it to Hadoop Cluster. There is also a performance tool to test basic performance such as write speed, read speed. To go further, we are trying to explore deeper to find how to test the look up time, load balancing feature and so on. This may need to modify the source code.



Some Implementation details

About Look up time on HDFS :

We read some source code of HDFS and found that while reading some file the program to use a method getLocatedBlocks(String src, long start) to get the block of the location. We will try to trace the method to find how much time it spends to look up for a file

Problems encountered and how they are resolved

Problem 1:

When we are try to do a java remote connect through hamachi to operate the HDFS, we are always fail to do so.

Reason:

Every node in the cluster has two network interfaces: one for the local environment in the lab and one for the hamachi. The cluster just listen the network interface for the local environment in the lab.

Solution:

Configure the <name>dfs.namenode.rpc-bind-host</name> to 0.0.0.0 so that namenode will listen all the network interface

Problem 2:

While trying to run the Map-reduce program on the cluster, there is always an error says can't find the MR app.

Solution:

Configure the mapred-site.xml and yarn-site.xml to set the correct class path.

CEPH part

CEPH high level architecture

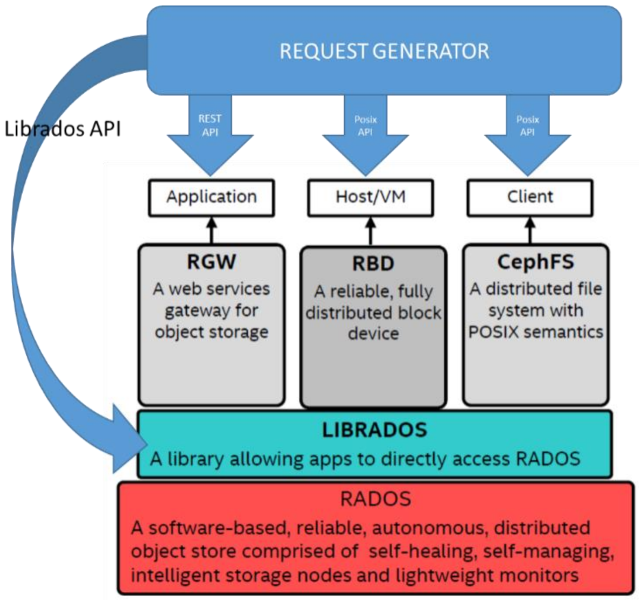
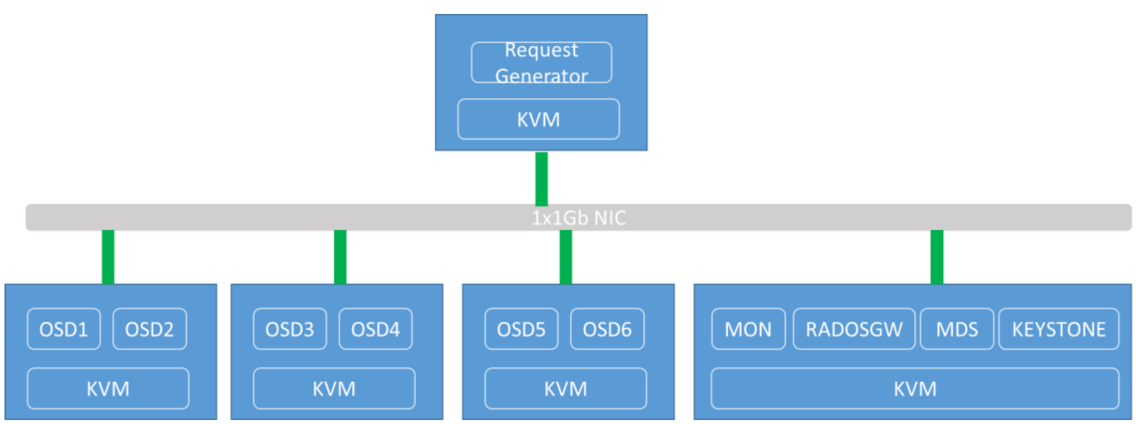


Figure 3High Level Architecture of CEPH part

CEPH detail level architecture



CEPH implementation details

There are three main methods to deploy Ceph at lab environment.

1. Manual installation
2. Install with deployment tool ceph-deploy provided by Ceph

3. Professional install tools like Chef,Puppet or Juju

We use ceph-deploy to install our ceph cluster. Basically ceph-deploy installation consists following steps.

1. Download and Install prerequisites for ceph on Linux OS
2. Install deploy node which has deploy tool
3. Configure the cluster conf file
4. Invoke installation of different Ceph nodes via RPC based on conf file

Problems encountered during CEPH implementation

1. With deploy-tool, ceph installation is the much easier than Swift.

Main workflow of CEPH

Write file via HTTP REST API in Ceph

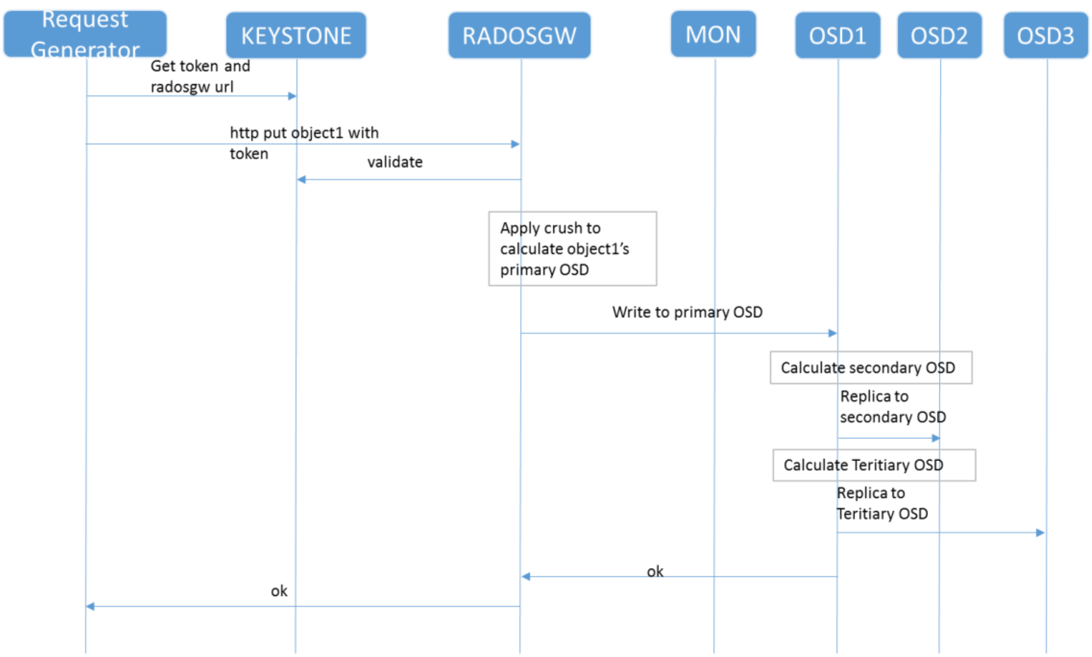


Figure 4Splay replication method implement by RADOS

Read via HTTP REST API in Ceph

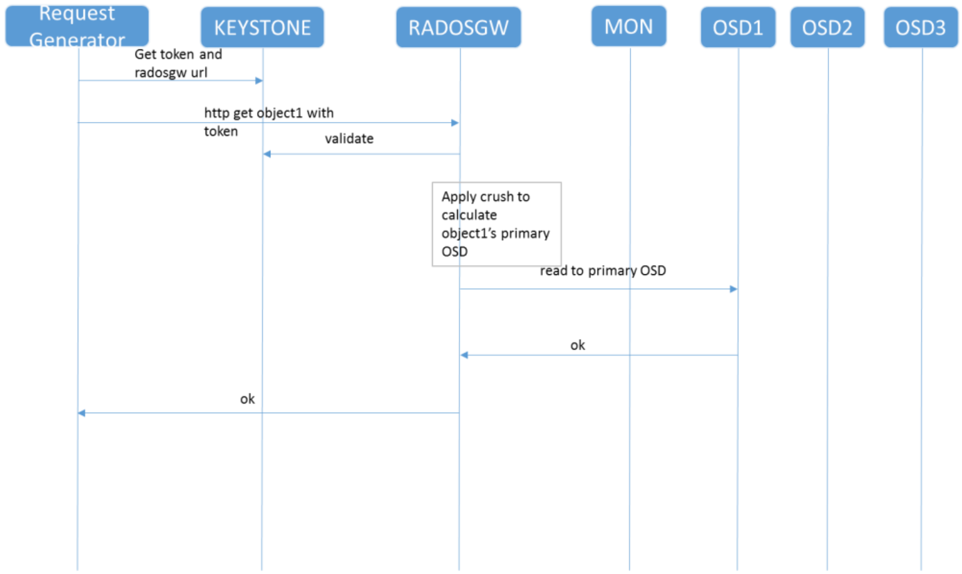
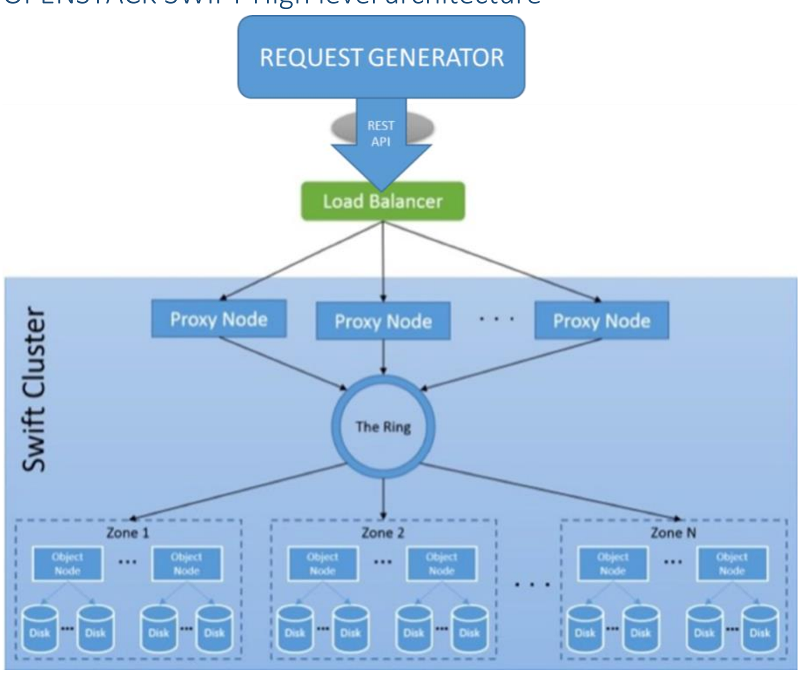


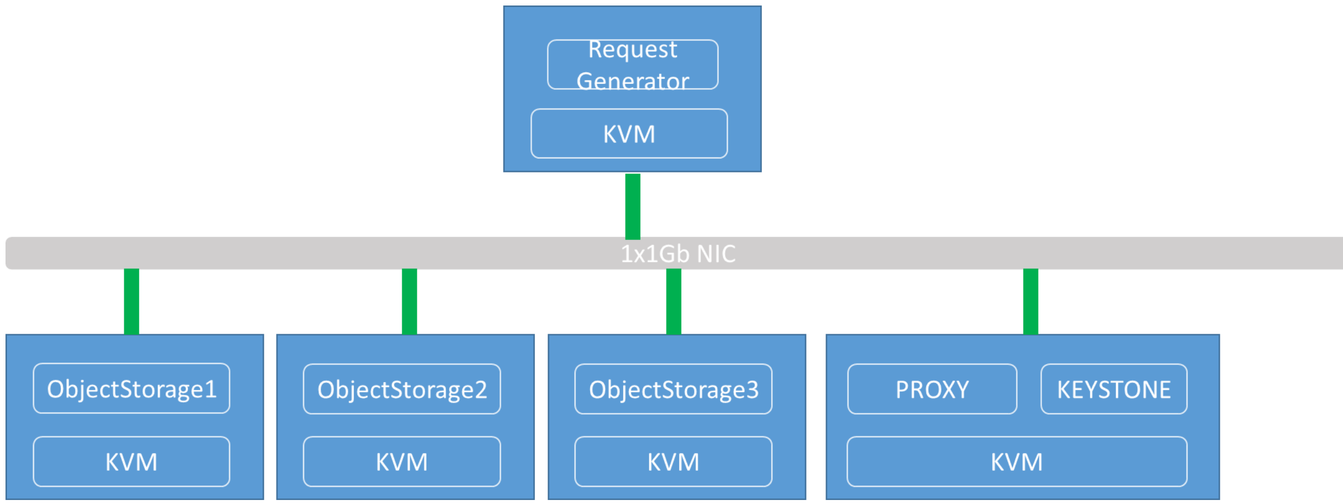
Figure 5 Call flow of CEPH read

OPENSTACK SWIFT Part

OPENSTACK SWIFT High level architecture



OPENSTACK SWIFT detail level architecture



OPENSTACK SWIFT Implementation Detail

OPENSTACK SWIFT installation has prerequisite on Authentication Component Keystone. The main steps are as following.

1. Install Keystone node and provisioned with Swift service and swift user subscription inside it.
2. Install SWIFT Storage nodes
    a. Attach free disk to Object storage and mount
    b. Configure Rsync on all storage node which is used to replication
3. Install SWIFT Proxy node
    a. Configure Swift Ring files which realize the DHT function of SWIFT
    b. Distribute the Swift Ring files to Object Storage nodes
4. Start Swift Service on all Storage nodes

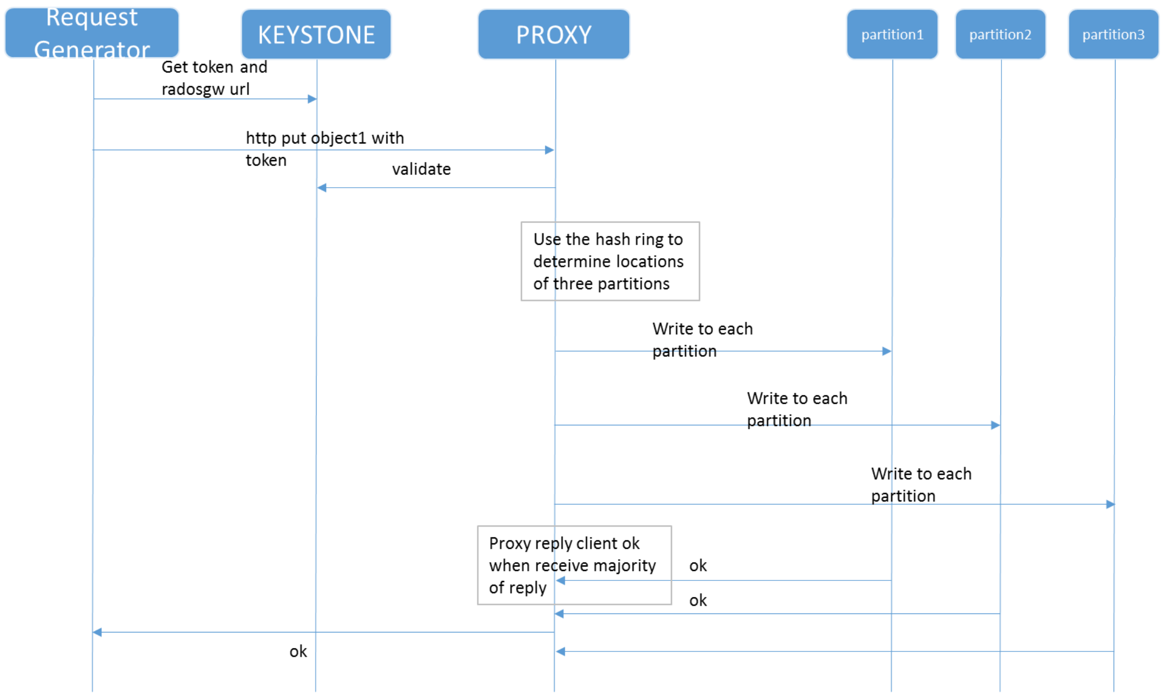
Problems encountered during OPENSTACK SWIFT implementation

1. Keystone V# integration with OPENSTACK SWIFT was a big issue because it's very new version and lacks of support.

Main workflow of SWIFT

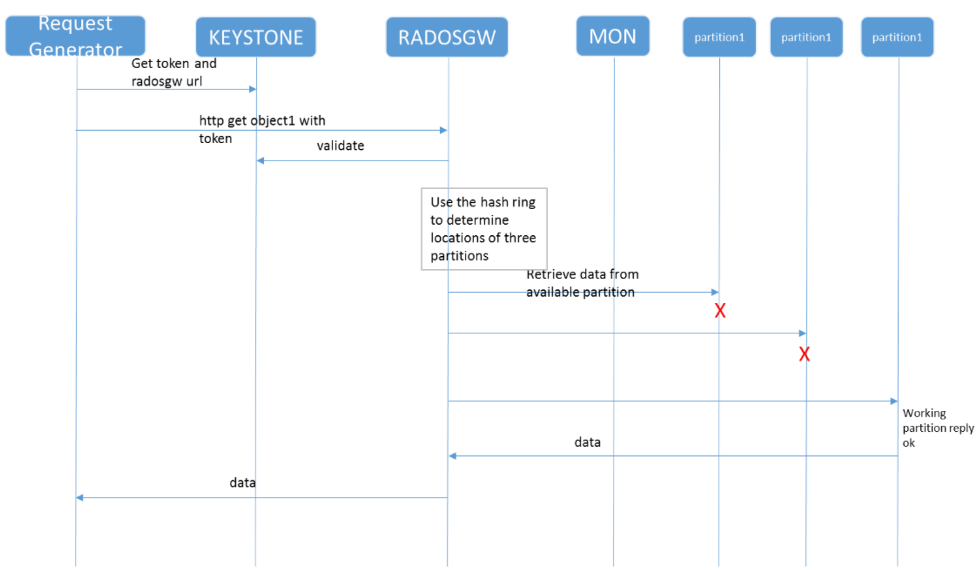
Write file via HTTP REST API in SWIFT

Write to swift



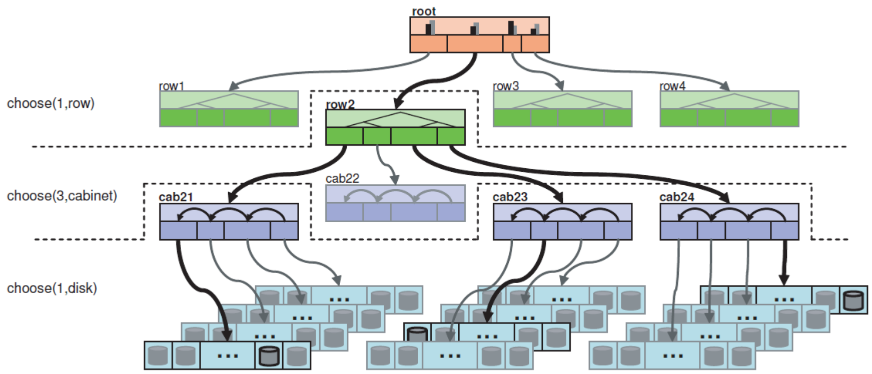
Read file via HTTP REST API in SWIFT

read on swift



OSD Map Data structure enhance Part

The basic OSD map data structure



initialization and lookup capabilities

```
0 :
| 1 :
| | 11 :
| | | 51 : ip:192.188.1.2 overload:false failed:false
| | | 49 : ip:1.1.1.1 overload:false failed:true
| | | 47 : ip:1.1.1.1 overload:false failed:true
| | | 42 : ip:192.168.1.130 overload:false failed:false
| | | 111 : ip:192.168.1.118 overload:false failed:false
| | | 112 : ip:192.168.1.119 overload:false failed:false
| | | 113 : ip:192.168.1.124 overload:false failed:false
| | | 114 : ip:192.168.1.126 overload:false failed:false
| | 12 :
| | | 121 : ip:192.168.1.127 overload:false failed:false
| | | 122 : ip:192.168.1.117 overload:false failed:false
| | | 123 : ip:192.168.1.108 overload:false failed:false
| | | 124 : ip:192.168.1.116 overload:false failed:false
| | 13 :
| | | 131 : ip:192.168.1.125 overload:false failed:false
| | | 132 : ip:192.168.1.128 overload:false failed:false
| | | 133 : ip:192.168.1.106 overload:false failed:false
| | | 134 : ip:192.168.1.107 overload:false failed:false
| | 14 :
| | | 141 : ip:192.168.1.109 overload:false failed:false
| | | 142 : ip:192.168.1.110 overload:false failed:false
| | | 143 : ip:192.168.1.113 overload:false failed:false
| | | 144 : ip:192.168.1.114 overload:false failed:false
| | | 145 : ip:192.168.1.129 overload:false failed:false
```

Client issuing node addition/removal and load balancing requests

Remapping part design

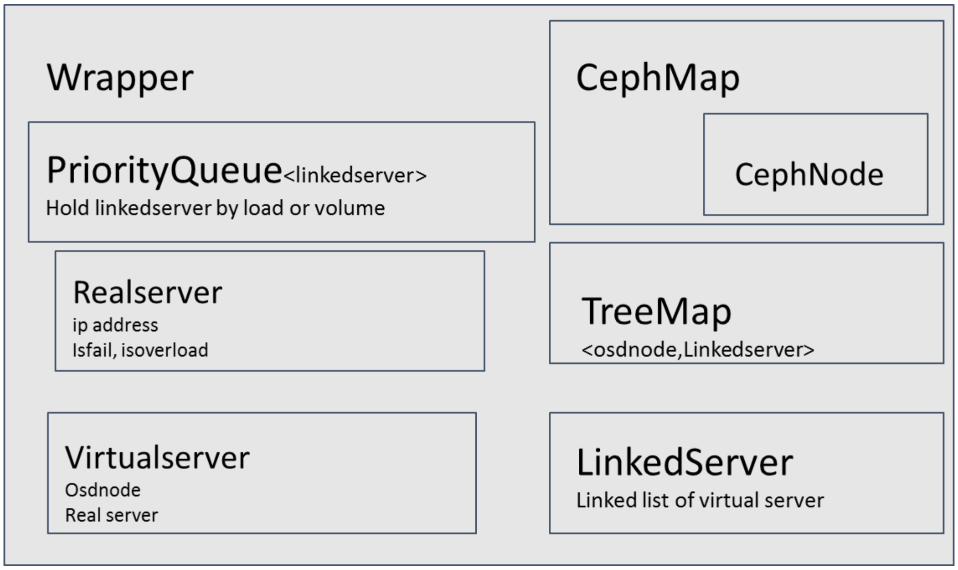


Figure 6data structures used in wrapper

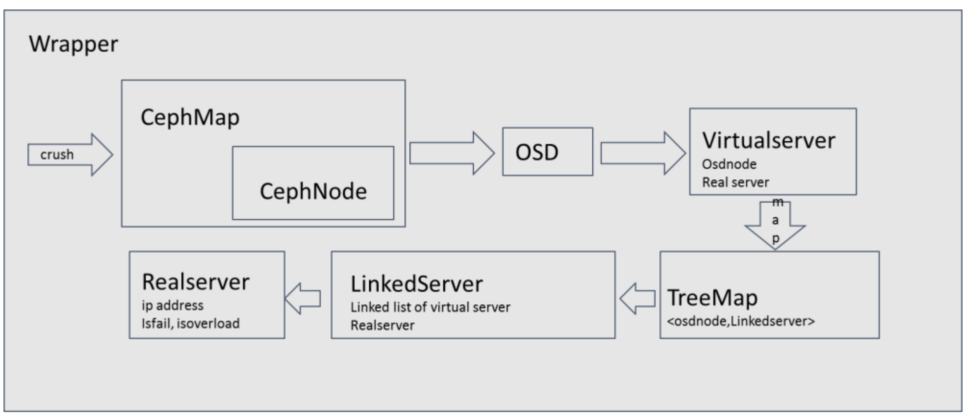


Figure 7Mapping from osd virtual node to real physical nodes

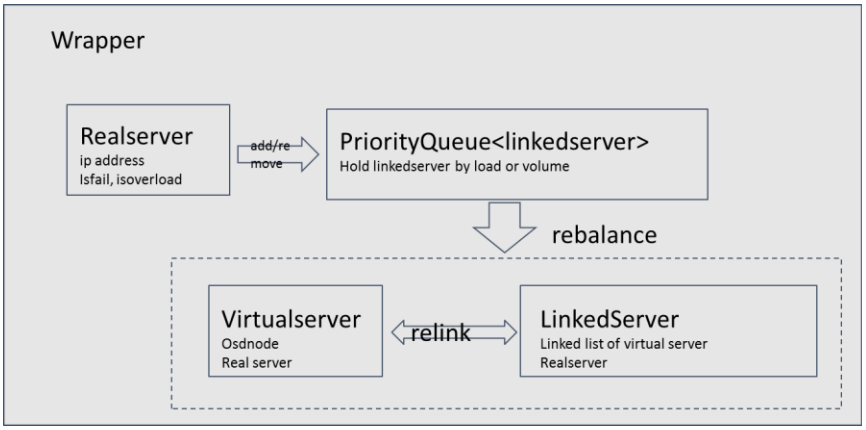
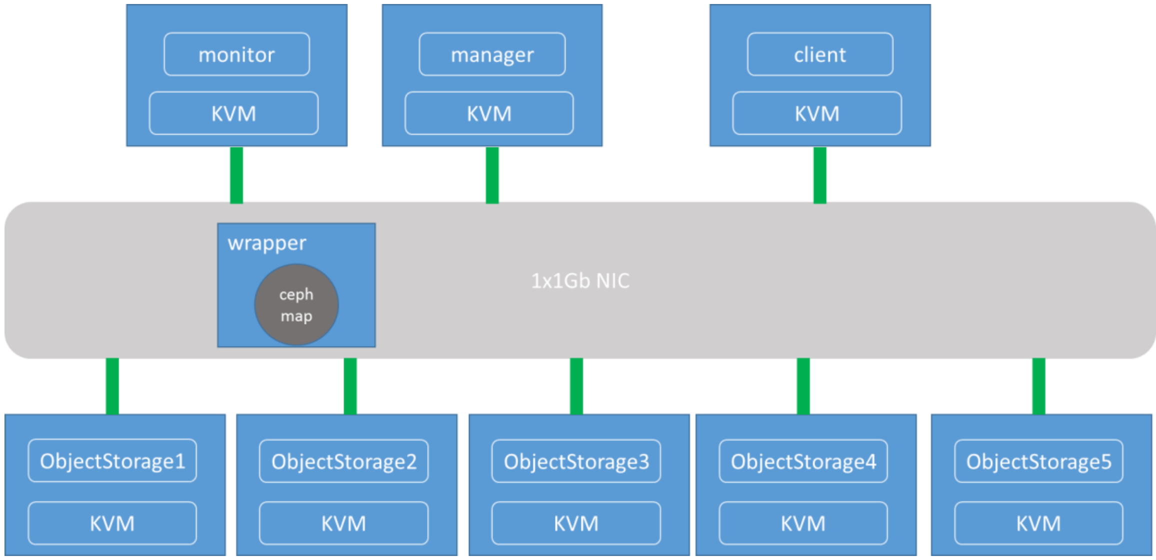


Figure 8Wrapper add/remove physical node

Remapping format

```
**********CephMap********************************physical node**************
0 :
| 1 :
| | 3 :
| | | 13 : ip:1.1.1.1 overload:false failed:false  --->  127.0.0.1:9561 failed
| | | 14 : ip:1.1.1.2 overload:false failed:false  --->  127.0.0.1:9565
| | | 15 : ip:1.1.1.3 overload:false failed:false  --->  127.0.0.1:9562 overloaded
| | | 16 : ip:1.1.1.4 overload:false failed:false  --->  127.0.0.1:9564
| | | 17 : ip:1.1.1.5 overload:false failed:false  --->  127.0.0.1:9563
| | | 18 : ip:1.1.1.6 overload:false failed:false  --->  127.0.0.1:9564
| | | 19 : ip:1.1.1.7 overload:false failed:false  --->  127.0.0.1:9563
| | | 20 : ip:1.1.1.8 overload:false failed:false  --->  127.0.0.1:9565
| | | 21 : ip:1.1.1.9 overload:false failed:false  --->  127.0.0.1:9561 failed
| | | 22 : ip:1.1.1.10 overload:false failed:false  --->  127.0.0.1:9562 overloaded
| | | 23 : ip:1.1.1.11 overload:false failed:false  --->  127.0.0.1:9561 failed
| | | 24 : ip:1.1.1.12 overload:false failed:false  --->  127.0.0.1:9562 overloaded
| | | 25 : ip:1.1.1.13 overload:false failed:false  --->  127.0.0.1:9563
| | | 26 : ip:1.1.1.14 overload:false failed:false  --->  127.0.0.1:9564
| | | 27 : ip:1.1.1.15 overload:false failed:false  --->  127.0.0.1:9565
| | | 28 : ip:1.1.1.16 overload:false failed:false  --->  127.0.0.1:9564
| | | 29 : ip:1.1.1.17 overload:false failed:false  --->  127.0.0.1:9565
| | | 30 : ip:1.1.1.18 overload:false failed:false  --->  127.0.0.1:9562 overloaded
| | | 31 : ip:1.1.1.19 overload:false failed:false  --->  127.0.0.1:9561 failed
| | | 32 : ip:1.1.1.20 overload:false failed:false  --->  127.0.0.1:9563
| | | 33 : ip:1.1.1.21 overload:false failed:false  --->  127.0.0.1:9561 failed
| | | 34 : ip:1.1.1.22 overload:false failed:false  --->  127.0.0.1:9563
| | | 35 : ip:1.1.1.23 overload:false failed:false  --->  127.0.0.1:9565
| | | 36 : ip:1.1.1.24 overload:false failed:false  --->  127.0.0.1:9564
| | | 37 : ip:1.1.1.25 overload:false failed:false  --->  127.0.0.1:9562 overloaded
```

Design topology



Experiment data of DHT

Time	lambda	alpha	duration	thread	osd node	lookuptime(ms)	table update(ms)	read time (lookup all osd node)
300s	10	0.5	1000000	8	5	17	39	48
300s	100	0.5	1000000	8	5	17	44	31
300s	100	0.5	1000	8	5	17	36	36
300s	200	0.5	1000000	8	5	16	45	30
300s	300	0.5	1000000	8	5	16	38	35
300s	1000	0.5	1000000	8	5	16	33	38
300s	300	0.5	1000000	1	5	17	41	25
300s	300	0.5	1000000	4	5	17	45	33
300s	300	0.5	1000000	8	5	16	44	25
300s	300	0.5	1000000	12	5	16	39	29
300s	300	0.5	1000000	16	5	16	43	30

DHT lookup time

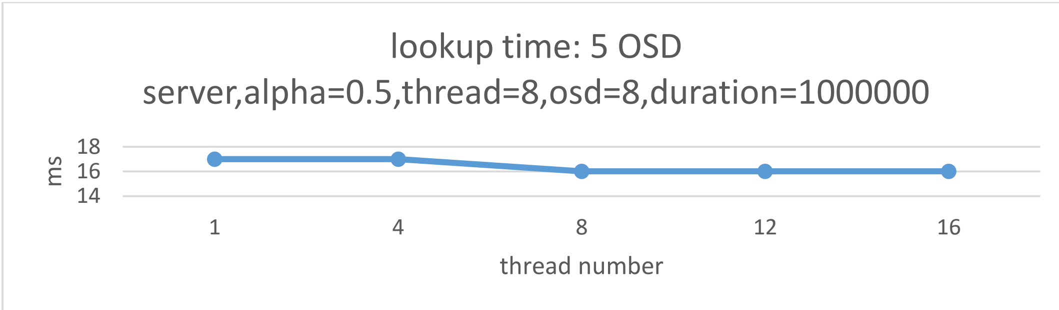


Figure 9lookup time relationship with thread number

Table update time

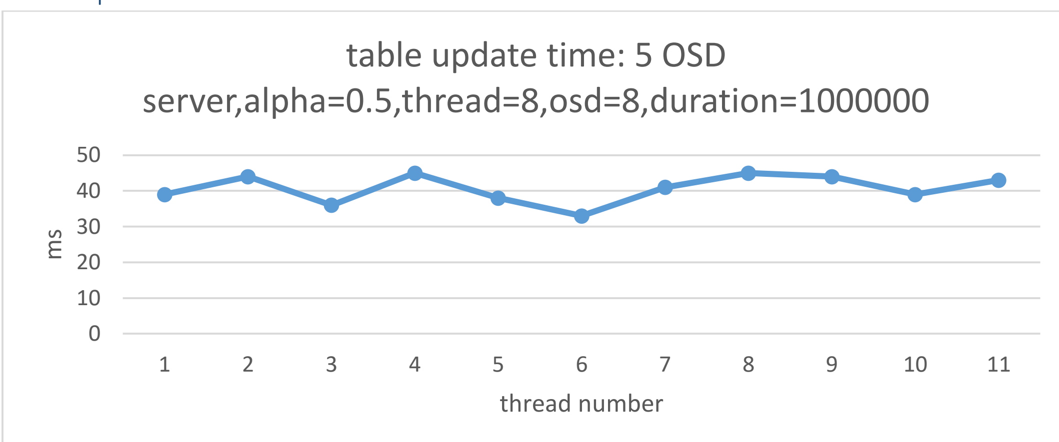


Figure 10 table update time relationship with thread number

Read time

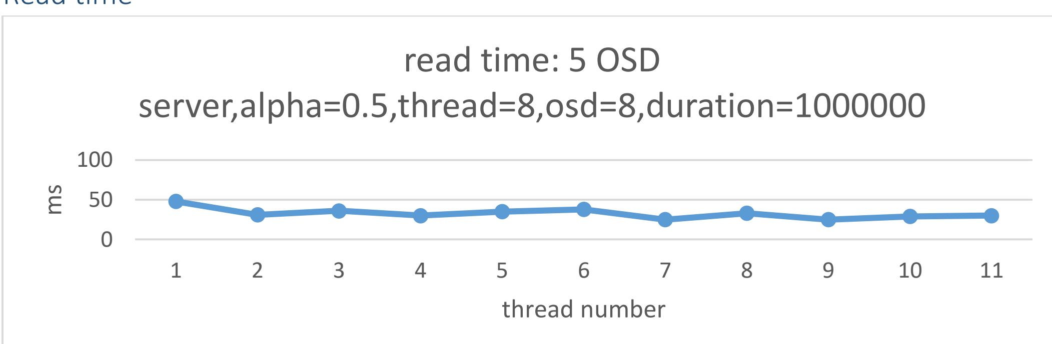


Figure 11relathion ship between thread number and read time

Request Generator Adapter

Class Design

Package	Code file	Role
Hadoop4j.hadoop4j	RequestAdapterHDFSAPI.java	Parse input request and call the method in HDFS_Client.java to communicate with Hadoop server
	HDFS_Client.java	Methods that implements actual write, read, create file operations
	filepropagate_test	A demo shows that how to do the file propagation task
	requestgenerator_test	A demo shows that how to do the request generator task
Swift4j	RequestAdapterSwift4j	Convert request into REST API foramt

Experimental results

HDFS Adapter part

Hadoop

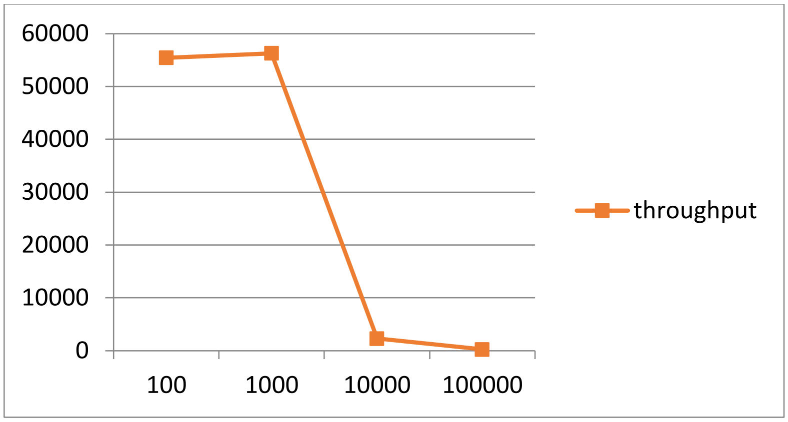
Experiment on duration/ lambda

We change the parameter lambda and duration several times to find out the relationship between the duration/lambda and the performance of HDFS (we use reading speed, writing speed and throughput as the metrics).

The experiment result is as following:

	Lambda	Duration	Duration /Lambda	Alpha	Reading speed	Writing speed	Throughput
1	100	10000	100	0.5	58.603 MB/s	45.65 MB/s	54.11 MB/s
2	100	100000	1000	0.5	61.443 MB/s	47.922 MB/s	54.958 MB/s
3	100	1000000	10000	0.5	23.056 MB/s	44.222 MB/s	2.247 MB/s
4	100	10000000	100000	0.5	22.461 KB/s	52.313 MB/s	232.422 KB/s

Relationship between the duration/lambda and the throughput:



As we can see from the chart above, as the Duration/Lambda increases, the throughput of the HDFS system decreases. We can also find the some information from the log of the Request Generator:

```
E1: Total req: 216, average request time: 1166 ms, overhead: 210 ms, interval: 1398 ms
E2: Total req: 150, average request time: 1386 ms, overhead: 231 ms, interval: 1984 ms
E3: Total req: 27, average request time: 517 ms, overhead: 77 ms, interval: 11100 ms
E4: Total req: 2, average request time: 5106 ms, overhead: 1 ms, interval: 78721 ms
```

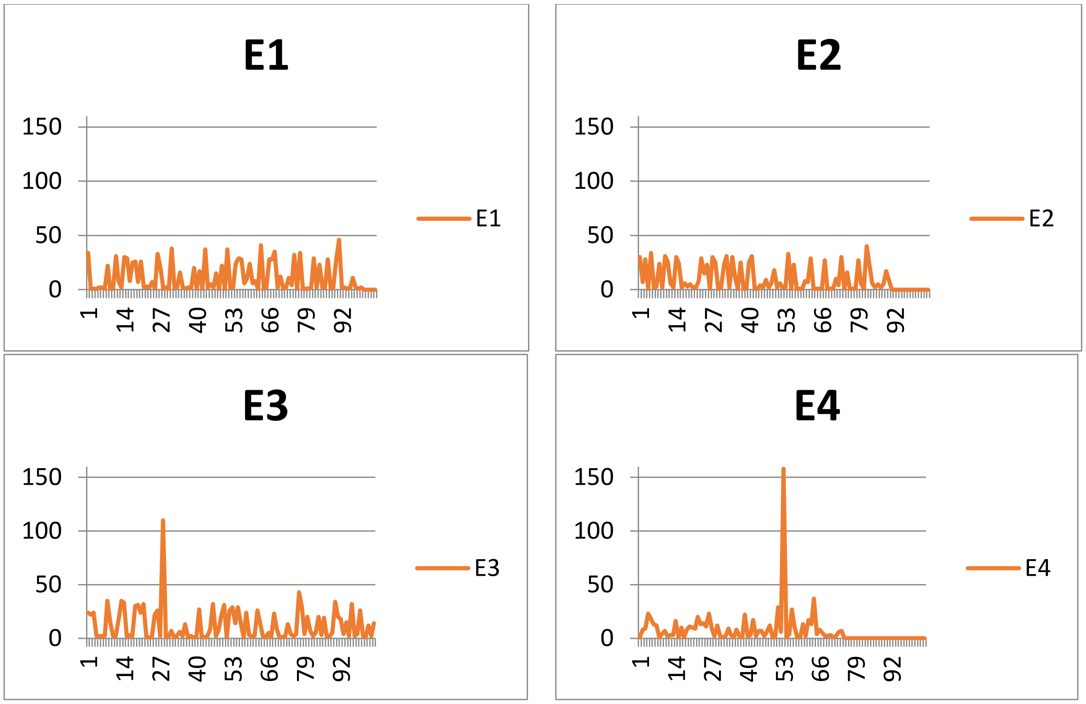
As the Duration/Lambda increase, the request generated sharply decreases and the interval time increases a lot.

So our conclusion is that the parameter Duration/Lambda is related to the interval time. When the parameter Duration/Lambda increases, the request generator will use a longer interval time to generate requests which means the HDFS system will spend more time waiting for the next request. That is the reason that the throughput decreases.

We also notice that, as the parameter Duration/Lambda increases, reading speed decrease a lot while writing speed doesn't change much. The reason is that files in HDFS are written once, so the write operation comes from the append and create operation. The append operations generated by the Request Generator are normally very large. At the same time, as the there are fewer requests, system get more chance to receive some small file read operation. That is the reason reading speed decrease a lot while writing speed doesn't change much.

Experiment on alpha

The alpha is parameter of Zipf distribution on requested files. From the following charts we can find that as the alpha increases, the distribution of the requested files is more uneven and more requests will focus on some fewer files.



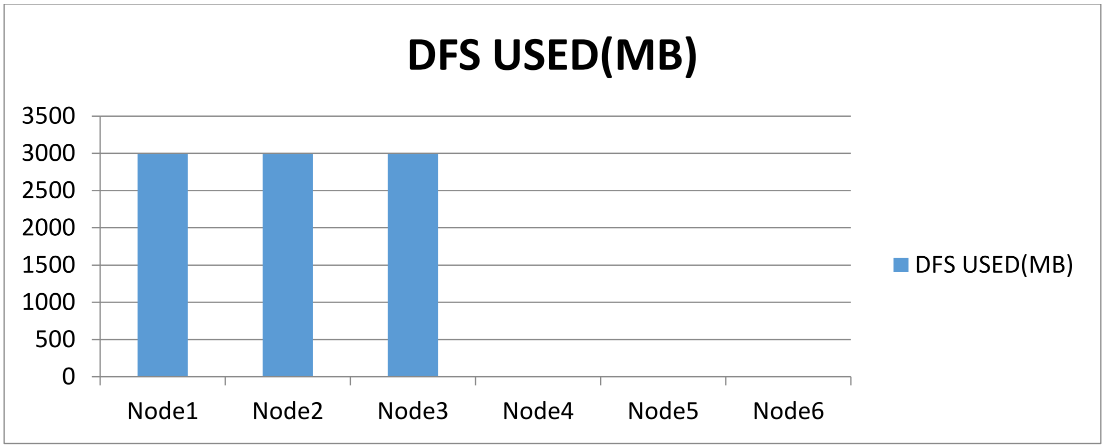
We also test the performance while we change the alpha. The result is as the following:

	Lambda	Duration	Alpha	Reading speed	Writing speed	Throughput
1	100	10000	0.01	59.494 MB/s	50.261 MB/s	55.552 MB/s
2	100	10000	0.1	59.295 MB/s	50.018 MB/s	55.792 MB/s
3	100	10000	0.5	58.603 MB/s	45.65 MB/s	54.11 MB/s
4	100	10000	0.9	59.307 MB/s	50.791 MB/s	57.128 MB/s

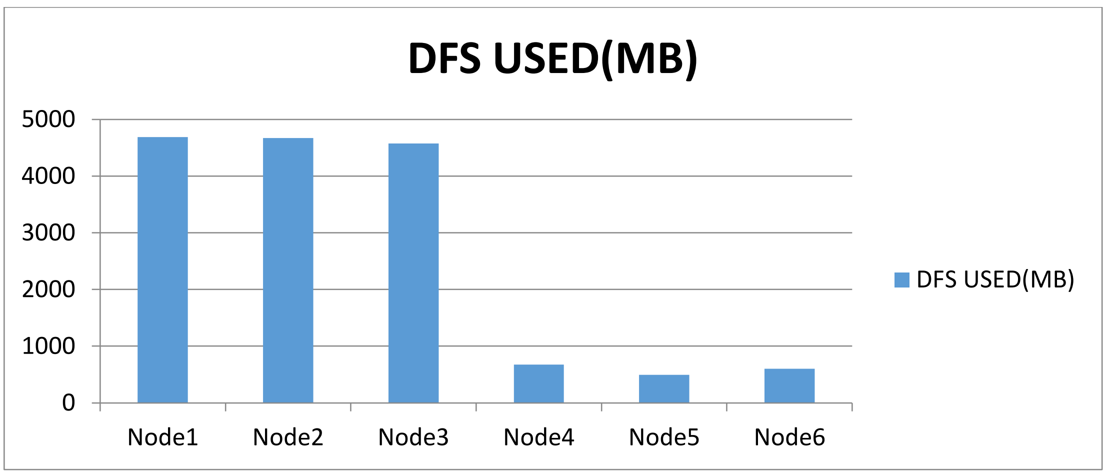
According to the data above, we find that the parameter alpha doesn't have much effect on the performance of the HDFS system. Because of the limited time, we didn't do more experiments on the parameter. However, we believe that the parameter alpha do have effect on the performance. In our scenario, we only have 8 threads and our test files are medium-size files. If we increase the test pressure (by increase the thread and the test file size), large amount of operations on some few files will definitely make the HDFS system throughput decreases. That is why load balance in HDFS is so important.

Load balance on HDFS

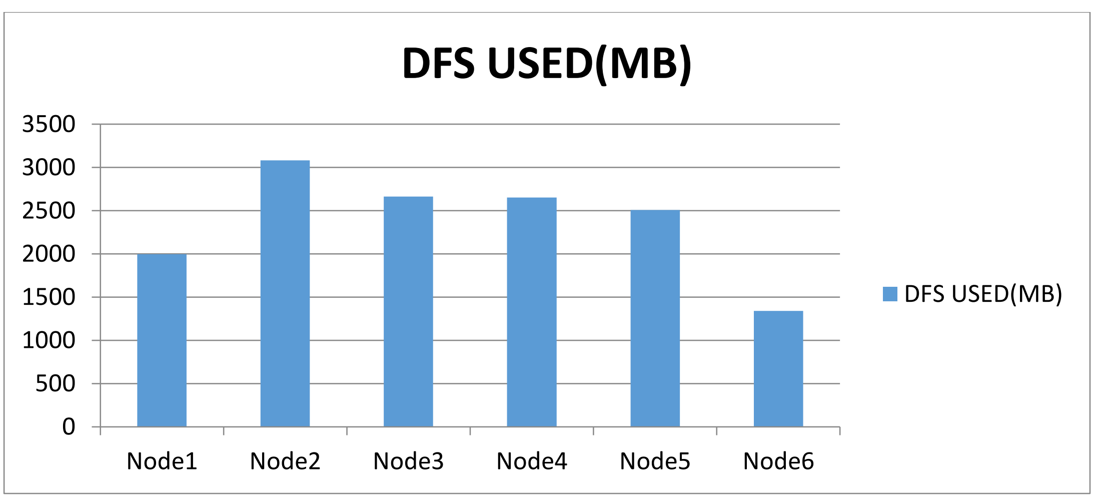
In HDFS, the load balance task is completed by the load balancer. We create the following scenario: do the file propagation on 3-node cluster and add extra 3 nodes to the cluster :



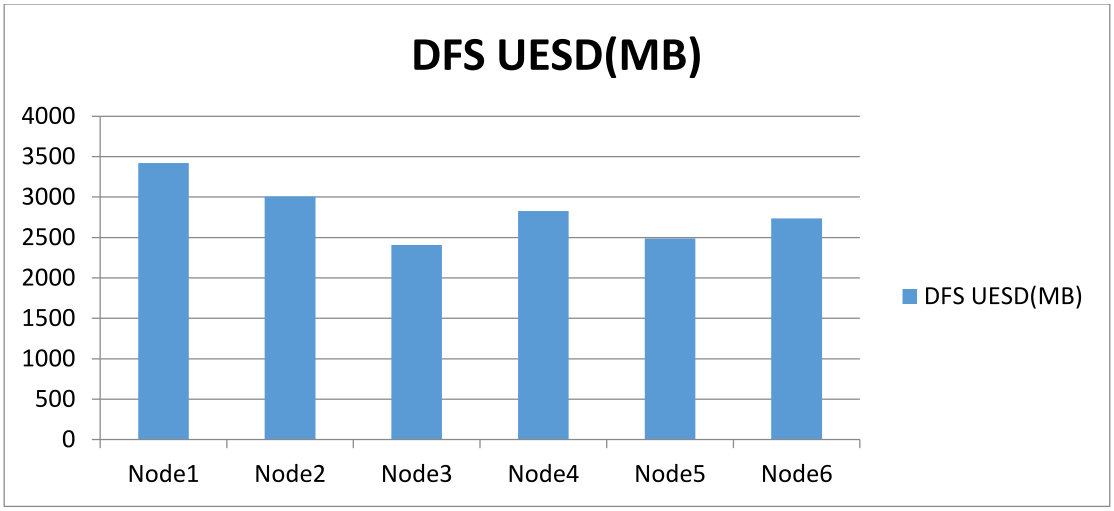
Then we run the request generator on such system, collect the performance data:



After that, we run the load balancer with default threshold 10%, it takes around 9 minutes to balance the data:



Finally, we run the load balancer with threshold 1%, it takes around 2 minutes to balance the data:



As we can see from the charts above, because the HDFS storage is based on block, so it can't get perfectly balanced especially when we only have limited number of files.

Balance with threshold 5%	9 Min
Balance with threshold 1%	2 Min

	Nodes	Reading speed	Writing speed	Throughput
1	3	58.907 MB/s	46.439 MB/s	55.128 MB/s
2	6	60.573 MB/s	52.218 MB/s	57.642 MB/s

As we can see, performance increased after we add 3 more nodes.

CEPH REST-API Adapter part

Experiment data

Propagate speed

propagate speed	ceph-REST
test1.txt	392.578 KB/s
test2.txt	764.648 KB/s

Request Generator data

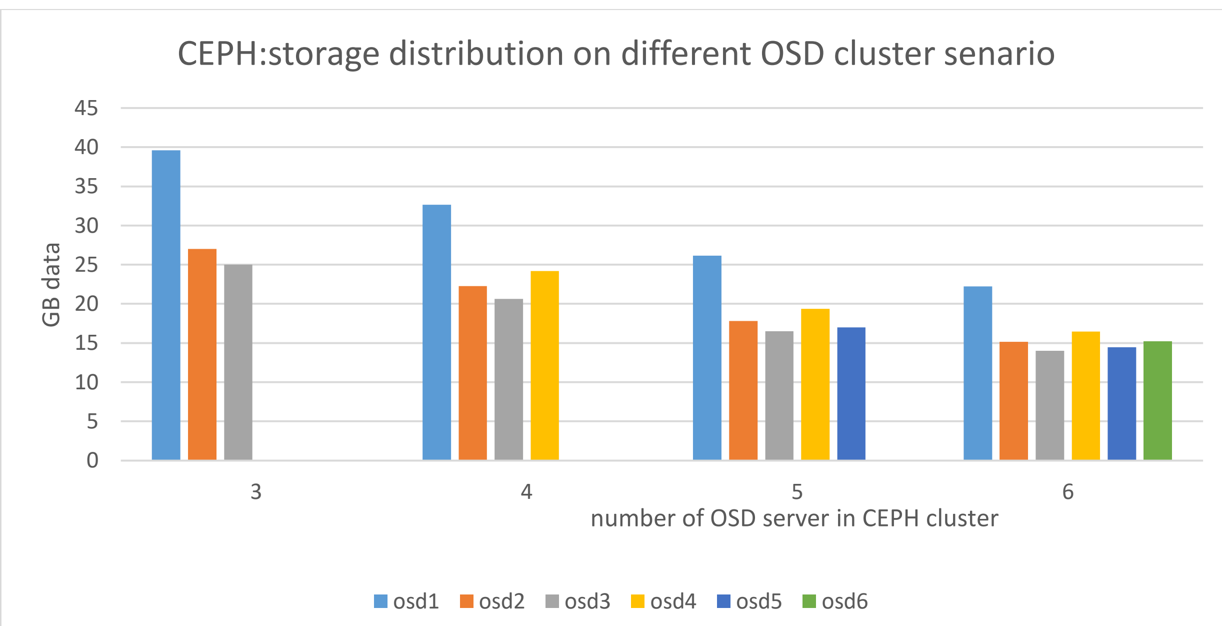
Time	lambda	alpha	duration	thread	osd node	Reading_speed	Writing_speed	Throughput	Lookup time
300s	10	0.5	1000000	8	8	17.578 KB/s	1.096 MB/s	91.797 KB/s	103.0 ms
300s	100	0.5	1000000	8	8	22.009 MB/s	1.019 MB/s	746.094 KB/s	95.0 ms
300s	200	0.5	1000000	8	8	17.874 MB/s	947.266 KB/s	849.609 KB/s	99.0 ms

300s	300	0.5	1000000	8	8	13.366 MB/s	825.195 KB/s	830.078 KB/s	101.0 ms
300s	300	0.5	1000000	1	8	1.899 MB/s	1.057 MB/s	137.695 KB/s	93ms
300s	300	0.5	1000000	4	8	12.234 MB/s	998.047 KB/s	721.68 KB/s	99.0 ms
300s	300	0.5	1000000	8	8	13.366 MB/s	825.195 KB/s	830.078 KB/s	101.0 ms
300s	300	0.5	1000000	12	8	29.603 MB/s	1.032 MB/s	1.172 MB/s	108.0 ms
300s	300	0.5	1000000	16	8	26.546 MB/s	796.875 KB/s	810.547 KB/s	298.0 ms

Relationship between duration/lambda and writing/reading speed

Reweight situation when there are 6,8,10,12 osd node.

node volume[GB]	node storage (20g)					
OSD nodes number [2 osd per osd physical node]	osd1	osd2	osd3	osd4	osd5	osd6
3	39.6	27	25			
4	32.67	22.275	20.625	24.2		
5	26.136	17.82	16.5	19.36	17	
6	22.2156	15.147	14.025	16.456	14.45	15.2



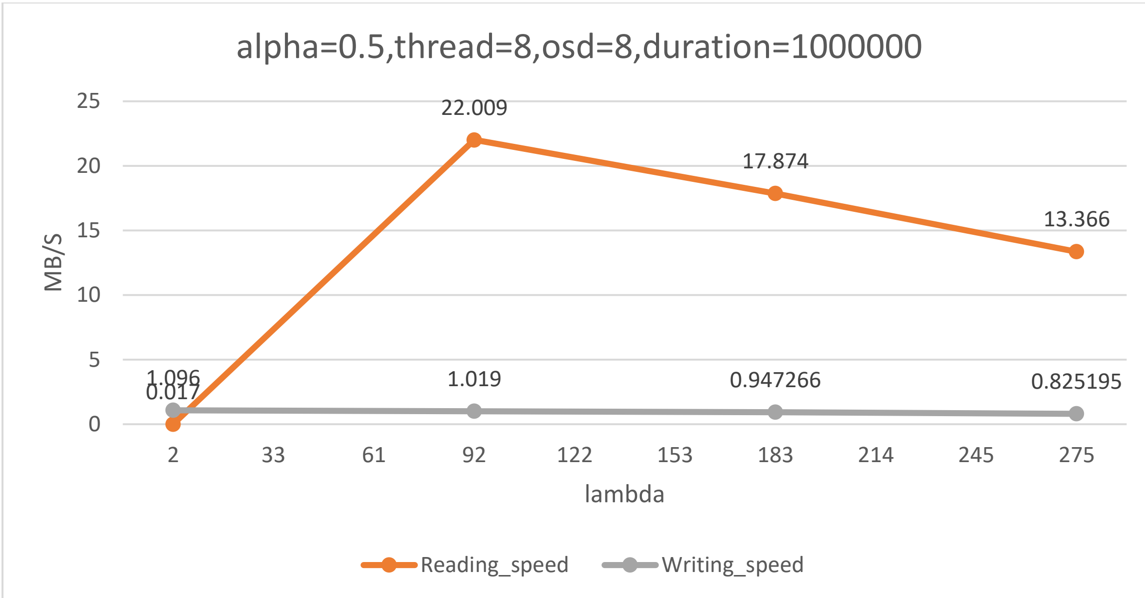


Figure 12writing reading speed relationship with duration/lambda

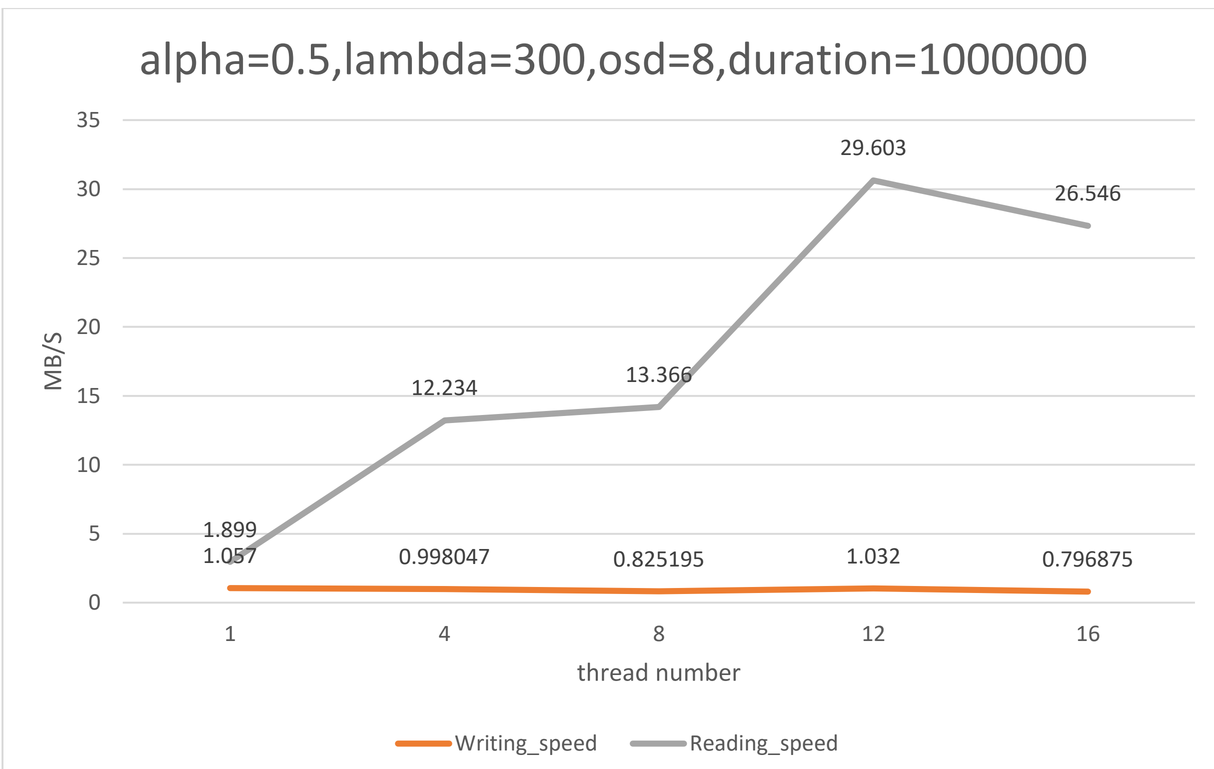


Figure 13relationship between writing reading speed and thread number

CEPH API-RADOS Adapter part

Experiment data

Propagate data speed

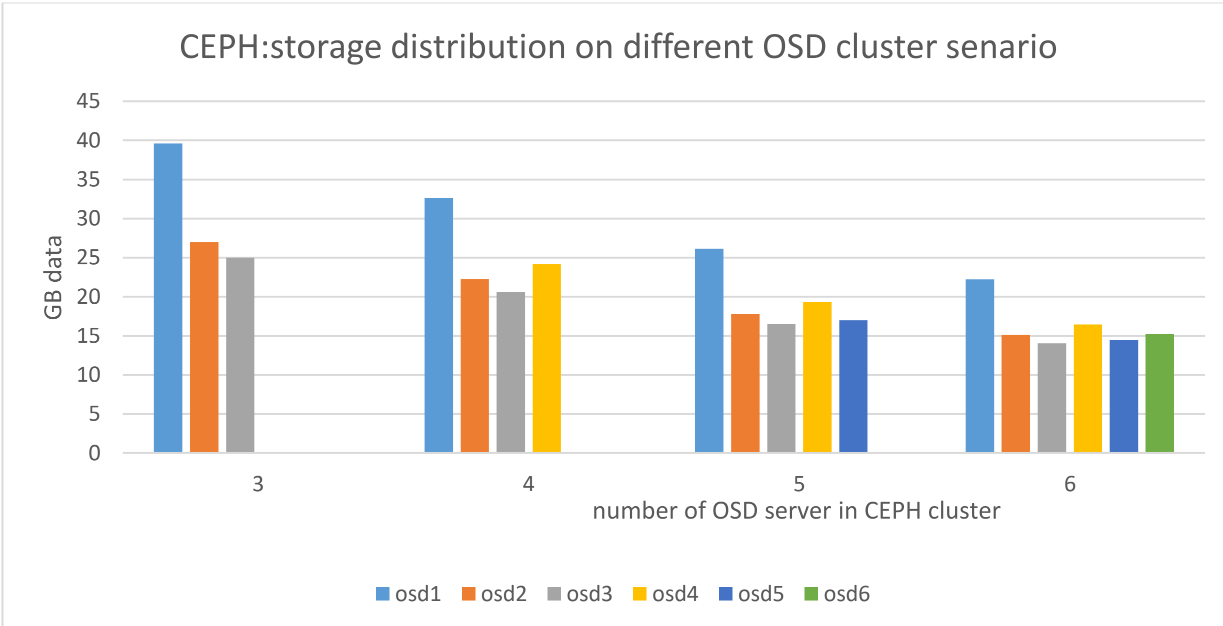
propagate	ceph-rados
test1.txt	6.795 MB/s
test2.txt	6.302 MB/s

Reading writing data

Time	lambda	alpha	duration	thread	osd node	Reading_speed	Writing_speed	Throughput
60s	10	0.5	1000000	8	8	53.946 MB/s	4.908 MB/s	5.705 MB/s
60s	100	0.5	1000000	8	8	40.189 MB/s	4.891 MB/s	4.922 MB/s
60s	200	0.5	1000000	8	8	54.464 MB/s	5.02 MB/s	5.447 MB/s
60s	300	0.5	1000000	8	8	43.166 MB/s	35.389 MB/s	19.773 MB/s
60s	300	0.5	1000000	1	8	46.026 MB/s	4.613 MB/s	4.797 MB/s
60s	300	0.5	1000000	4	8	52.456 MB/s	4.679 MB/s	5.513 MB/s
60s	300	0.5	1000000	8	8	43.166 MB/s	35.389 MB/s	19.773 MB/s
60s	300	0.5	1000000	12	8	22.296 MB/s	33.768 MB/s	27.293 MB/s
60s	300	0.5	1000000	16	8	46.006 MB/s	4.543 MB/s	2.141 MB/s

Ceph Storage data distribution under different cluster scenario

node volume	node storage (20g)					
OSD nodes number [2 osd per osd physical node]	osd1	osd2	osd3	osd4	osd5	osd6
3	25	33	27			
4	20.625	27.225	22.275	24.2		
5	16.5	21.78	17.82	19.36	15.26	
6	14.025	18.513	15.147	16.456	16.86	14.98



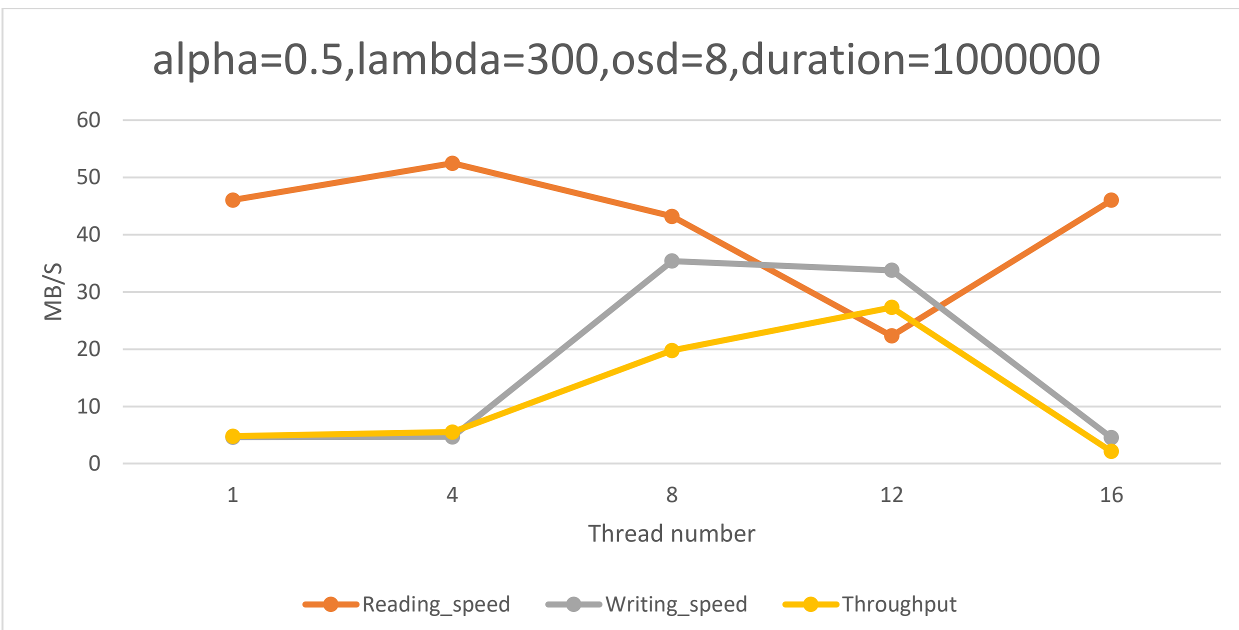


Figure 14 relationship between writing reading speed and thread number

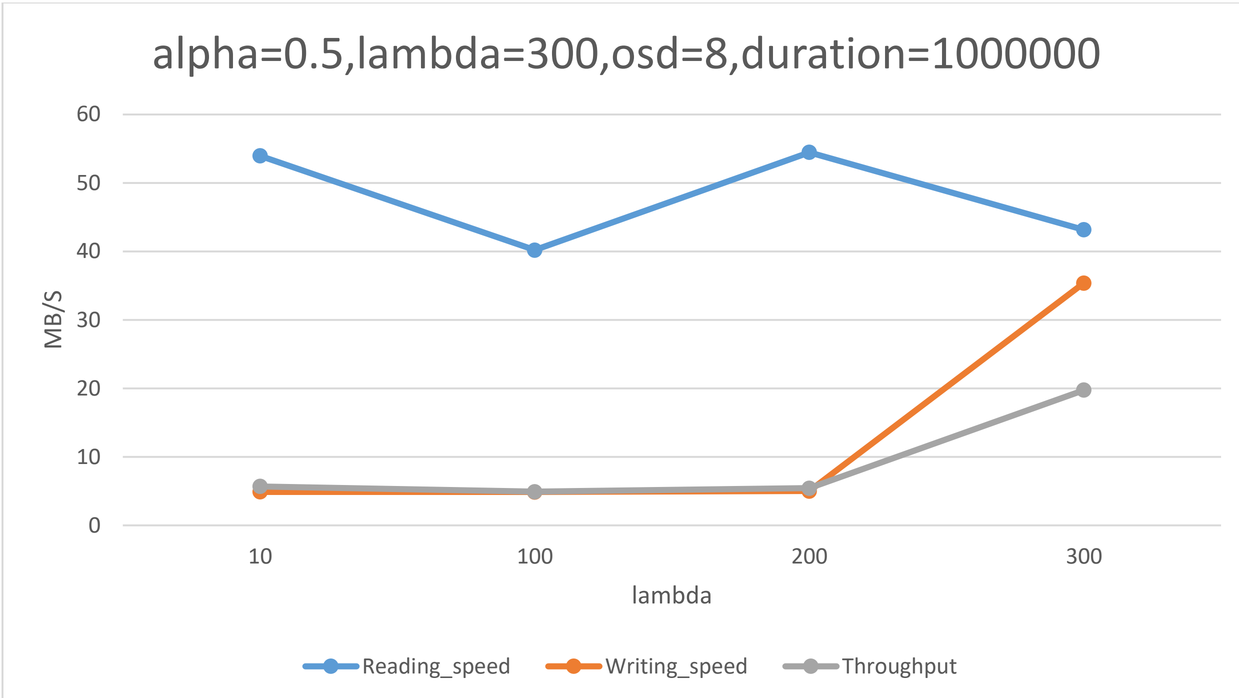
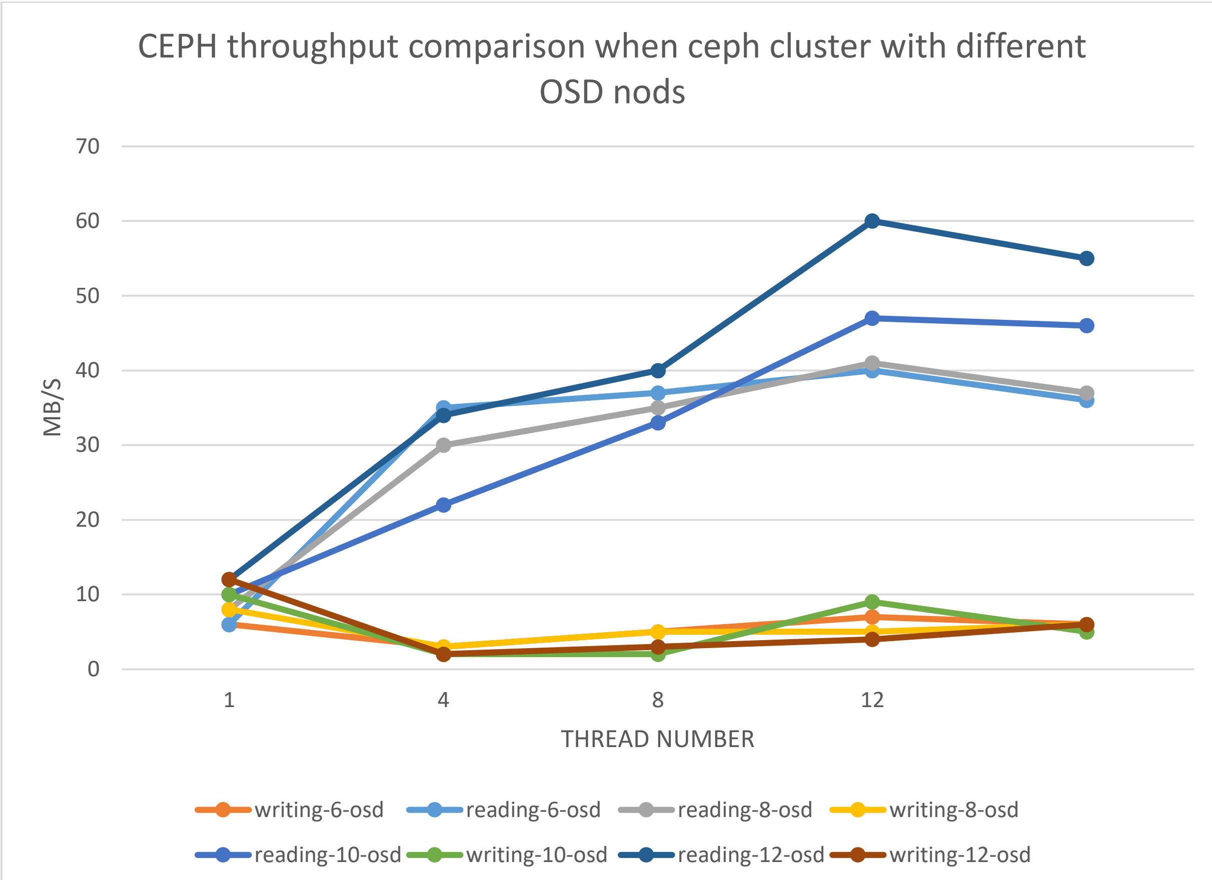


Figure 15 Relationship between reading && writing speed and lambda

alpha=0.5,lambda=300,duration=1000000								
CEPH-RADOS	reading-6-osd	writing-6-osd	reading-8-osd	writing-8-osd	reading-10-osd	writing-10-osd	reading-12-osd	writing-12-osd
thread	6	6	8	8	10	10	12	12
1	35	3	30	3	22	2	34	2
4	37	5	35	5	33	2	40	3
8	40	7	41	5	47	9	60	4
12	36	6	37	6	46	5	55	6



SWIFT REST-API Adapter part

Experiment data

propagate	swift-REST
test1.txt	849.609 KB/s
test2.txt	869.141 KB/s

Raw data

Time	lambda	alpha	duration	thread	osd node	Reading_speed	Writing_speed	Throughput
300s	10	0.5	1000000	8	8	6.413 MB/s	1.261 MB/s	159.18 KB/s
300s	100	0.5	1000000	8	8	2.626 MB/s	1.324 MB/s	628.906 KB/s
300s	100	0.5	1000	8	8	10.653 MB/s	1.29 MB/s	1.645 MB/s
300s	200	0.5	1000000	8	8	6.299 MB/s	1.313 MB/s	1.428 MB/s

300s	300	0.5	1000000	8	8	18.746 MB/s	1.308 MB/s	1.243 MB/s
300s	1000	0.5	1000000	8	8	5.568 MB/s	1.312 MB/s	1.427 MB/s
300s	300	0.5	1000000	1	8	9.953 MB/s	1.234 MB/s	1.234 MB/s
300s	300	0.5	1000000	4	8	10.754 MB/s	5.516 MB/s	1.167 MB/s
300s	300	0.5	1000000	8	8	18.746 MB/s	1.308 MB/s	1.243 MB/s
300s	300	0.5	1000000	12	8	9.8 MB/s	1.307 MB/s	1.144 MB/s
300s	300	0.5	1000000	16	8	3.547 MB/s	1.312 MB/s	972.656 KB/s

Swift Storage data distribution under different cluster senario

node volume	node storage (20g)					
OSD nodes number	osd1	osd2	osd3	osd4	osd5	osd6
3	25	33	27			
4	20.625	27.225	22.275	24.2		
5	16.5	21.78	17.82	19.36	15.26	
6	14.025	18.513	15.147	16.456	16.86	14.98

alpha=0.5,lambda=300,duration=1000000								
SWIFT	reading-6-osd	writing-6-osd	reading-8-osd	writing-8-osd	reading-10-osd	writing-10-osd	reading-12-osd	writing-12-osd
thread	1.26	0.8	1.68	0.75	2.1	0.9	2.28	1.2
1	6.3	0.9	4.8	0.98	3.52	1.2	6.8	1.1
4	8.14	1.2	7.7	1.5	7.26	1.3	10	1.2
8	10	1	7.79	1.1	8.93	1.06	13.2	1.1
12	5.76	1.4	9.25	1.6	11.5	1.12	14.85	1

Figure 16relationship between reading writing data and lambda

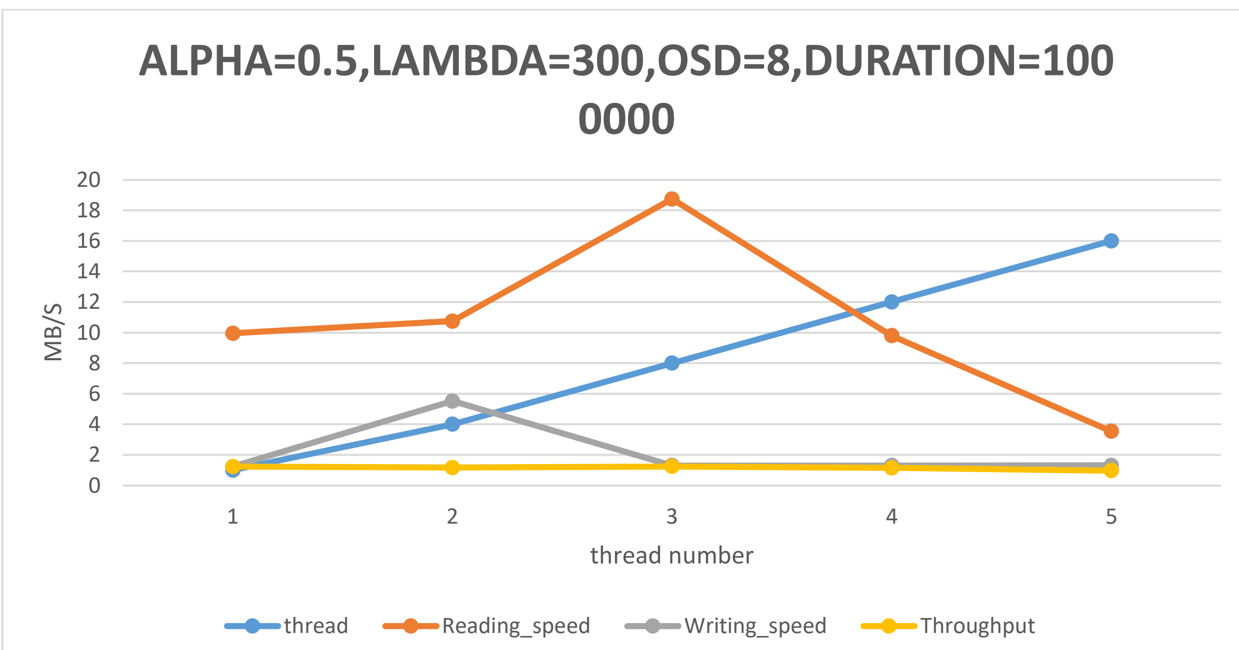
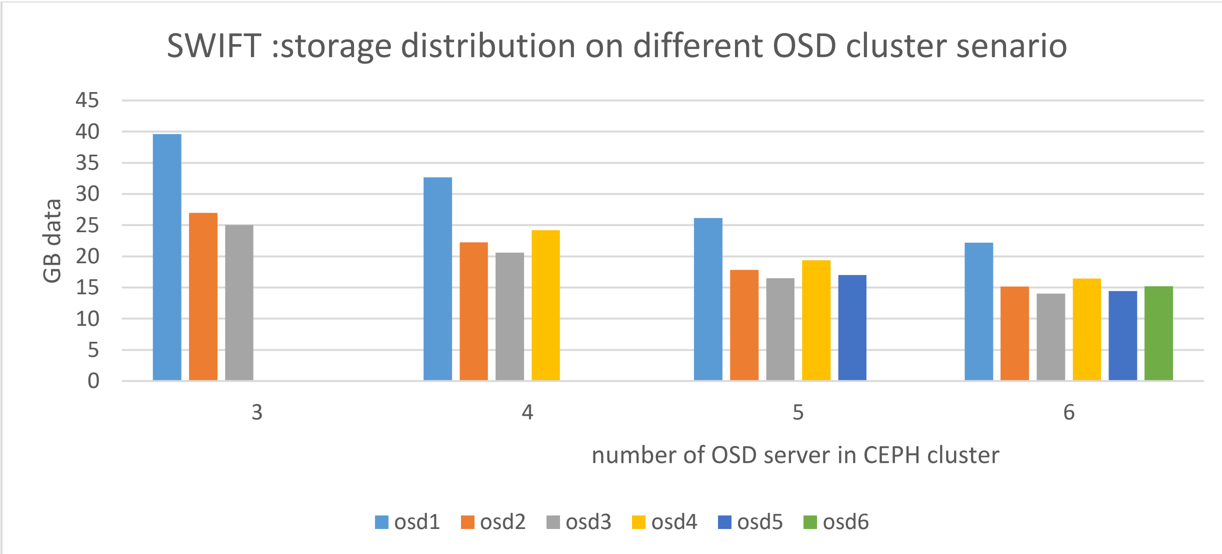
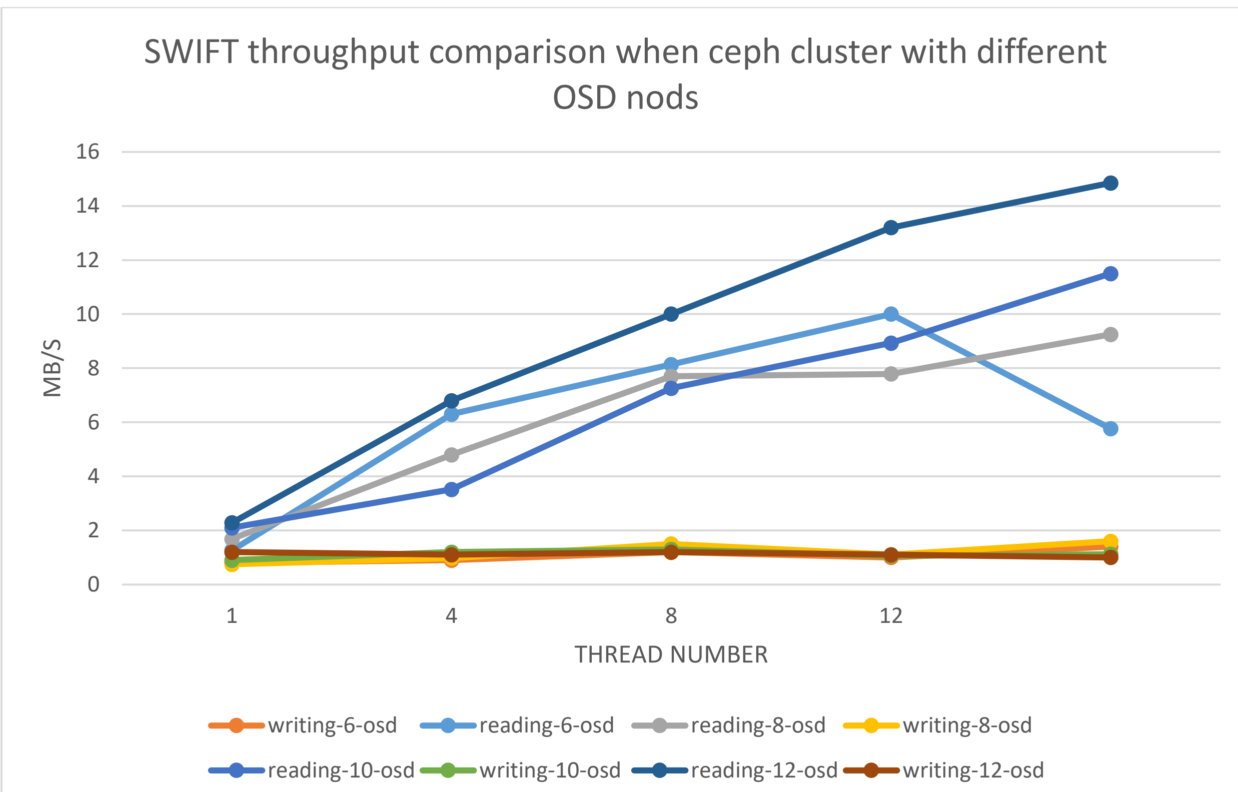


Figure 17 relationship between writing reading data and thread number



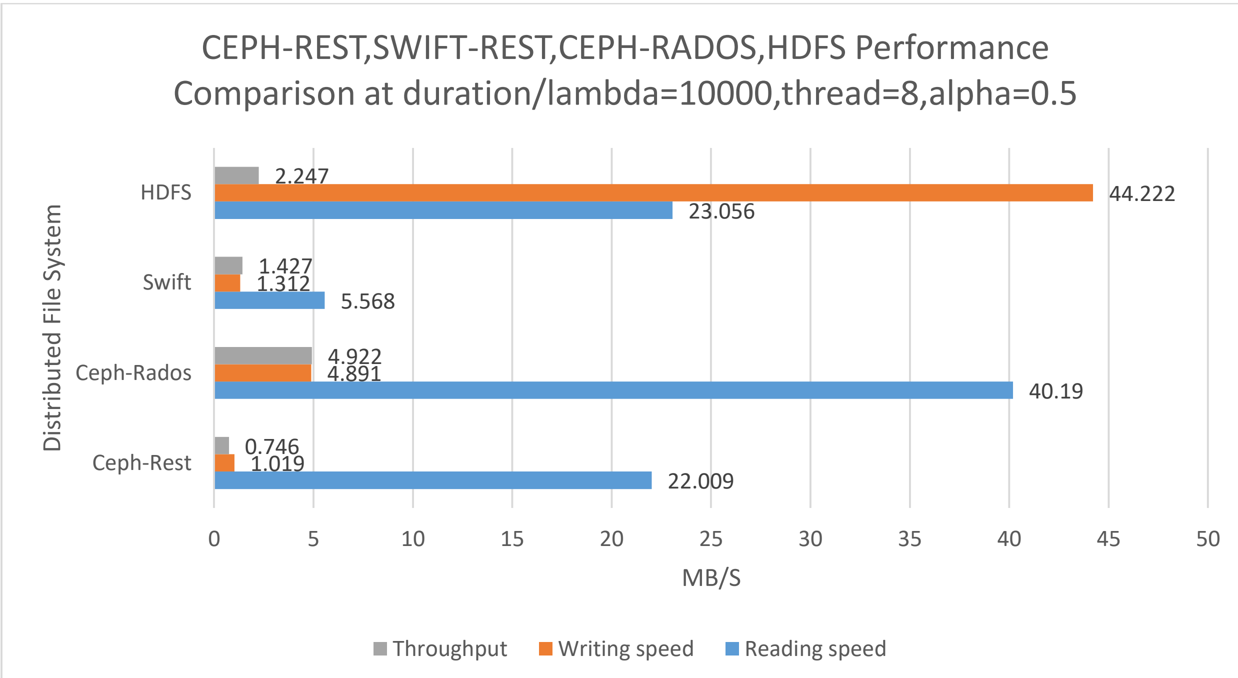


Comparison between HDFS, Ceph and Swift

Performance

FS	duration/lambda	Alpha	running time	Thread	Node	Reading speed	Writing speed	Throughput
Ceph-Rest	10000	0.5	300s	8	8	22.009	1.019	0.746

Ceph-Rados	10000	0.5	300s	8	8	40.19	4.891	4.922
Swift	10000	0.5	300s	8	8	5.568	1.312	1.427
HDFS	10000	0.5	300s	8	4	23.056	44.222	2.247



Installation Manual

HDFS

How to setup your programs

Import the code folder into IDE and you can run *Demo in every separatepackage (For example: hadoop4j.hadoop4j. filepropagate_test.java) to run the test task.

HDFS

For the Hadoop part, you need to make sure your machine can access and naming all machines in the cluster (namenode, datanode1, datanode2, datanode3)

How to get access to our system

To log into our system, you can ssh to the IP address listed in the VMlist table above in the lab environment.

Work load distribution

Include a high level summary

Xin Tong: Ceph , Swift, OSD map

Chenzhi Tian: Hadoop, external performance data collection

Detailed list

Assignment	Subtasks	Assigned to	Status
File system installations on VMs	Install Centos on Hypervisors	Both	100%
	Prepare KVM virtualization environment on centos 7	Both	100%
	Install Ceph file system	Tong Xin	100%
	Install Swift file system	Tong Xin	100%
	Install HDFS file system and MapReduce environment	Tian Chenzhi	100%
Identify the metrics for evaluating the file systems	Identify the metrics for evaluating the file systems	Tian Chenzhi	100%
	Put in report and discuss during midterm demo	Tian Chenzhi	100%
Request Generator for FS propogate	Swift RESTFUL API Requester Generator	Tong Xin	100%
	Ceph RESTFUL API Requester Generator	Tong Xin	100%
	Ceph Rados API Requester Generator	Tong Xin	100%
	DHT API Request Generator	Tong Xin	100%
	HDFS API Requester Generator	Tian Chenzhi	100%
Request Generator performance data collector component	Calculate performance value from log	Tian Chenzhi	100%
DHT design and coding	To implement the OSD map	Tong Xin	100%
DHT auto deployment in VMs environment	To implement the OSD map	Tian Chenzhi	100%
Automation deploy Ceph OSD	Automation deployment of CEPH osd nodes by cloning VM	Tong Xin	100%
Automation deploy HDFS data nodes	Automation deployment of HDFS data node	Tian Chenzhi	100%

Source Code link

https://github.com/exinton/CloudComputing