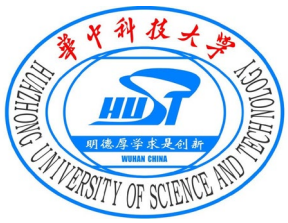


2019 级

# 《物联网数据存储与管理》课程

# 实 验 报 告

姓　名 张睿

学　号 U201912633

班　号 计算机校交 1902 班

日　期 2022.04.14

# 目　　录

## 一、实验目的

1. 熟悉对象存储技术，代表性系统及其特性；
2. 实践对象存储系统，部署实验环境，进行初步测试；
3. 基于对象存储系统，分析性能问题，架设应用实践。

## 二、实验背景

在如今数据规模支持增长，结构日趋复杂的时代，如何搭建一个兼具高性能和高可靠的存储系统是一个困扰相关人员的难题，对象存储便由此应运而生。对象存储（Object-based Storage）是一种新的网络存储架构，综合了 NAS 和 SAN 的优点，同时具有 NAS 的分布式数据共享和 SAN 的高速直通访问等优势，提供了具有高性能、高可靠性、跨平台、支持安全数据共享的存储体系结构。对象存储架构的核心在于将数据通路（数据读写）和控制通路（元数据）分离，并且基于对象存储设备来构建存储系统，每个对象存储设备都具有一定的职能，能够自动管理其上的数据分布。

分布式对象存储的代表之一为 Ceph，Ceph 是一个统一的分布式存储系统，最早起源于 Sage 就读博士期间的工作（最早的成果于 2004 年发表），随后贡献给开源社区。其设计初衷是提供较好的性能、可靠性和可扩展性。在经过多年的发展之后，目前已得到众多云计算厂商的支持并被广泛应用。RedHat 及 OpenStack 都可与 Ceph 整合以支持虚拟机镜像的后端存储。

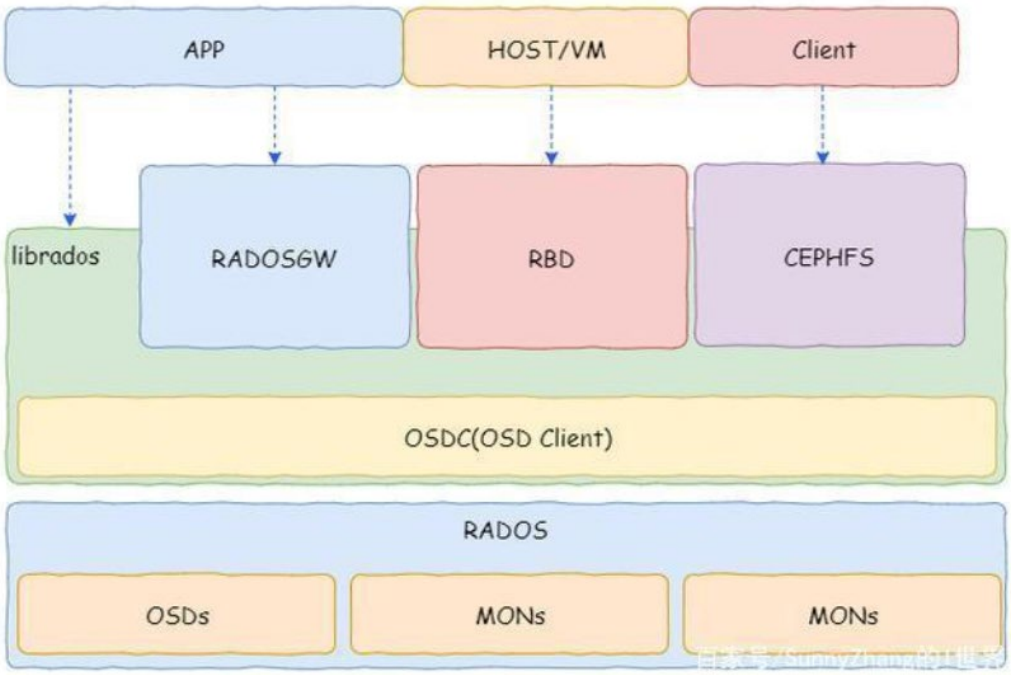


图 1 ceph 架构图（来源 Ceph 的整体架构介绍，这篇文章带入 Ceph 的大门 (baidu.com)）

## 三、实验环境

表格 1 实验环境说明

| | |
|---|---|
| 虚拟机操作系统 | WSL2+Ubuntu 20.04 LTS |
| 对象存储服务端 | Ceph 16.2.5 |
| 对象存储客户端 | Aws-cli 1.22.90 |
| 评测工具 | S3bench |

本次实验利用 WSL2+Ubuntu20.04LTS 实验，为了减少配置环境的繁琐，使用 docker 镜像部署，这里使用 Ceph-daemon 镜像使得相关组件可以“all-in-one”配置，不用再额外增加相关组件的 image，对象存储的客户端使用 aws-cli，这是由于 ceph 的 RGW 支持 aws-cli，或者说绝大多数对象存储系统都支持 aws-cli。评测工具使用 s3bench，需要注意的是，s3bench 调用的是 aws-sdk，所以其实不安装 aws-cli 作为客户端也能够完成性能测评。

## 四、实验内容

整个实验由 3 个部分组成，对象存储系统搭建、对象存储系统性能分析以及尾延迟挑战，接下来分别叙述。

### 4.1 对象存储技术实践

这一部分主要为搭建 ceph 存储系统。由于是第一次使用 docker 以及 ceph，踩了许多坑，这一部分会稍微详细一点，为之后相关的环境配置作参考。根据架构图，我们需要搭建如下组件：monitor， osd， rgw， mgr。其中 osd 为对象存储节点，monitor 为观测节点， rgw 为接口连接节点， mgr 为集群管理节点。

1. docker 的安装

Docker 安装参考官网步骤 Install Docker Engine on Ubuntu | Docker Documentation

2. 拉取 ceph-daemon 镜像

$sudo docker pull ceph/daemon:latest

其中，latest 为镜像的 tag

3. 初始化 ceph 的挂载目录。

$sudo mkdir -p /etc/ceph /var/lib/ceph /var/log/ceph

4. 创建 ceph 的专用网路。

$sudo docker network create --driver bridge --subnet 172.20.0.0/16 ceph-network

其中，driver 为指定以 bridge 网络创建，需要说明的是，对于 docker 的网络来说，所有的新增的容器都是 attach 到 bridge 上的，subnet 为网段范围，ceph-network 为名字。

为了观察我们创建的网络，可以使用

$sudo docker inspect ceph-network

5. 创建 monitor 节点

$sudo docker run -itd --name monnode --network ceph-network --ip 172.20.0.10 --restart always -v /etc/ceph:/etc/ceph -v /var/lib/ceph/:/var/lib/ceph/ -v /var/log/ceph/:/var/log/ceph/ -e CLUSTER=ceph -e WEIGHT=1.0 -e MON_IP=172.20.0.10 -e MON_NAME=monnode -e CEPH_PUBLIC_NETWORK=172.20.0.0/16 ceph/daemon mon

其中，-itd 为交互、分配 tty、detach container， -v 为挂载映射。

可以通过如下方法查看容器状况。

$sudo docker inspect monnode # 查看 monnode 容器信息

$sudo docker logs monnode # 查看日志

$sudo docker exec monnode ceph -s # 查看集群的健康状态

6. 创建 osd 节点

首先从 monnode 节点获得 keyring

docker exec monnode ceph auth get client.bootstrap-osd -o /var/lib/ceph/bootstrap-osd/ceph.keyring。

接着利用 ceph/daemo 创建三个 osd 节点，并用 logs 查看创建日志

$sudo docker run -itd

--privileged=true --name osdnode1 --network ceph-network -e CLUSTER=ceph -e

WEIGHT=1.0 -e MON_NAME=monnode -e MON_IP=172.20.0.10 -v /etc/ceph:/etc/ceph -v

/var/lib/ceph/:/var/lib/ceph/ -v /var/lib/ceph/osd/1:/var/lib/ceph/osd -e

OSD_TYPE=directory -v /etc/localtime:/etc/localtime:ro ceph/daemon osd

其中 priviledged 参数需注意，没有此参数则登入后的 root 不是真正的 root 用户，也无法看到和 host 的挂载关系。

7. 创建 mgr 节点

$sudo docker run -itd --privileged=true --restart=always --name mgrnode

--network ceph-network -e CLUSTER=ceph --pid=container:monnode -v

/etc/ceph:/etc/ceph -v /var/lib/ceph/:/var/lib/ceph/ ceph/daemon mgr

其中-pid 表示 mgr 节点和 monnode 节点共享命名空间。

8. 创建 gateway 节点

$sudo docker exec monnode ceph auth get client.bootstrap-rgw -o /var/lib/ceph/bootstrap-rgw/ceph.keyring

$sudo docker run -itd

--privileged=true --name rgwnode --network ceph-network -e CLUSTER=ceph -v

/var/lib/ceph/:/var/lib/ceph/ -v /etc/ceph:/etc/ceph -v

/etc/localtime:/etc/localtime:ro -e RGW_NAME=rgw0 ceph/daemon rgw

9. 创建 ceph-client 节点

为了创建这个节点，我们拉取一个新的镜像 ubuntu，为了使得这个节点可以和 ceph 集群内的节点进行通信，我们也需要将其放入 ceph-network 网络内，除此之外，为了方便使用 vscode 等 app 操作，需要暴露其一个端口，即与主机端口建立映射关系。

$sudo docker pull ubuntu:latest

$sudo docker run -itd --network=ceph-network -p 8000:8074 --name=ceph-client ubuntu

其中将容器的 8074 端口和主机的 8000 端口建立了连接。

自此，服务端和客户端都搭建完成，可以看看当前的容器列表以及集群的状态

```
(base) ~  $ sudo docker container ls
CONTAINER ID   IMAGE         COMMAND                  CREATED      STATUS         PORTS                                        NAMES
2041702f0203   ceph-client   "/bin/sh"                2 days ago   Up 38 seconds  0.0.0.0:8000->8097/tcp, :::8000->8097/tcp    ceph-client
80d9e188bb83   ceph/daemon   "/opt/ceph-container…"   7 days ago   Up 38 seconds                                               rgwnode
6bc220c2afe4   ceph/daemon   "/opt/ceph-container…"   7 days ago   Up 14 seconds                                               mgrnode
a91aaf851106   ceph/daemon   "/opt/ceph-container…"   7 days ago   Up 37 seconds                                               osdnode3
df38d540b33c   ceph/daemon   "/opt/ceph-container…"   7 days ago   Up 37 seconds                                               osdnode2
b62dbce61960   ceph/daemon   "/opt/ceph-container…"   7 days ago   Up 36 seconds                                               osdnode1
a109450629c1   ceph/daemon   "/opt/ceph-container…"   7 days ago   Up 36 seconds                                               monnode
```

图 2 docker 容器列表

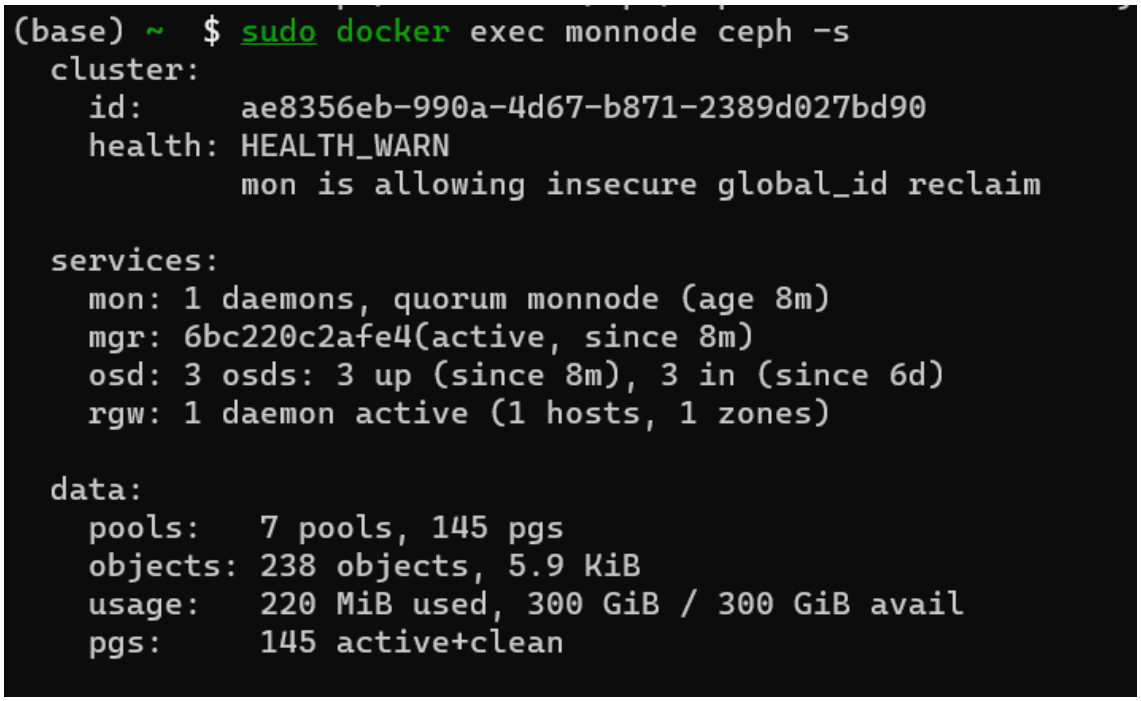


图 3 ceph 集群状态

11. 客户端配置

客户端配置主要为 aws-cli 的下载和配置，可参考 How To Configure AWS S3 CLI for Ceph Object Gateway Storage | ComputingForGeeks

其中需要注意的是，rgw 的端口也需要指定，可以通过进入 rgwnode 后，利用 ss 指令获得当前的监听端口，利用该监听端口达到 aws-cli 和集群通信的目的。

最后，简单验证一下该对象存储系统是否可用


图 4 客户端与 ceph 集群通信

## 4.2 对象存储性能分析

利用 s3bench 对该对象存储系统进行测评，为了批量测评，使用脚本的形式。

```
endpoint="http://172.20.0.3:7480"
bucket="test"
ObjectNamePrefix="loadgen"
AccessKey="0A9X9WW0UY99V15IK3IR"
AccessSecret="CF5FX3SxzvWEKx12WIcgtgg1AYyWpOElRTlDwiQI"
filepath="result4.txt"
verbose="true"

declare  -a  NumClient
declare  -a  NumSample
declare  -a  ObjectSize

NumClient=(1    2     3     4     5     6     7     8     9     10    11    12)
NumSample=(100  100   100   100   100   100   100   100   100   100   100   100)
ObjectSize=(102400 102400 102400 102400 102400  102400  102400  102400  102400  102400  102400  102400)

#display run progress
progress=11

for(( i=0;i<${#NumClient[@]};i++))
do
    # run sh
    $s3bench -accessKey=$AccessKey -accessSecret=$AccessSecret -bucket=$bucket -endpoint=$endpoint \
    -numClients=${NumClient[i]} -numSamples=${NumSample[i]} -objectNamePrefix=$ObjectNamePrefix -objectSize=${ObjectSize[i]} -verbose=$verbose >>
$filepath
```

图 5 s3bench 脚本

结果如下：

1. 并行度为 1 时，样本数量为 100 时对象大小的变化对各因素的影响

| 序号 | client num | object size(MB) | w_throughput(MB/s) | | w_latency(s) | w_90%latency(s) | r_throughput(MB/s) | | r_latency(s) | r_90%latency(s) |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0.001 | 0.05 | | 0.064 | 0.023 | 0.02 | | 0.06 | 0.05 |
| 2 | 1 | 0.002 | 0.1 | | 0.038 | 0.028 | 0.98 | | 0.003 | 0.002 |
| 3 | 1 | 0.004 | 0.22 | | 0.033 | 0.02 | 1.81 | | 0.003 | 0.002 |
| 4 | 1 | 0.01 | 0.56 | | 0.03 | 0.02 | 4.76 | | 0.006 | 0.002 |
| 5 | 1 | 0.02 | 1.12 | | 0.032 | 0.02 | 8.94 | | 0.003 | 0.002 |
| 6 | 1 | 0.04 | 2.15 | | 0.033 | 0.021 | 18.14 | | 0.004 | 0.003 |
| 7 | 1 | 0.1 | 5.28 | | 0.029 | 0.023 | 40.75 | | 0.004 | 0.003 |
| 8 | 1 | 0.2 | 8.56 | | 0.089 | 0.028 | 71.15 | | 0.007 | 0.004 |
| 9 | 1 | 0.4 | 13.13 | | 0.132 | 0.035 | 117.18 | | 0.005 | 0.004 |
| 10 | 1 | 1 | 19.29 | | 0.405 | 0.091 | 69.71 | | 0.039 | 0.029 |
| 11 | 1 | 2 | 14.44 | | 0.964 | 0.274 | 200.36 | | 0.014 | 0.011 |
| 12 | 1 | 4 | 16.22 | | 1.564 | 0.453 | 226.96 | | 0.057 | 0.021 |

图 6 并行度为 1 时，对象大小的变化对各因素的影响

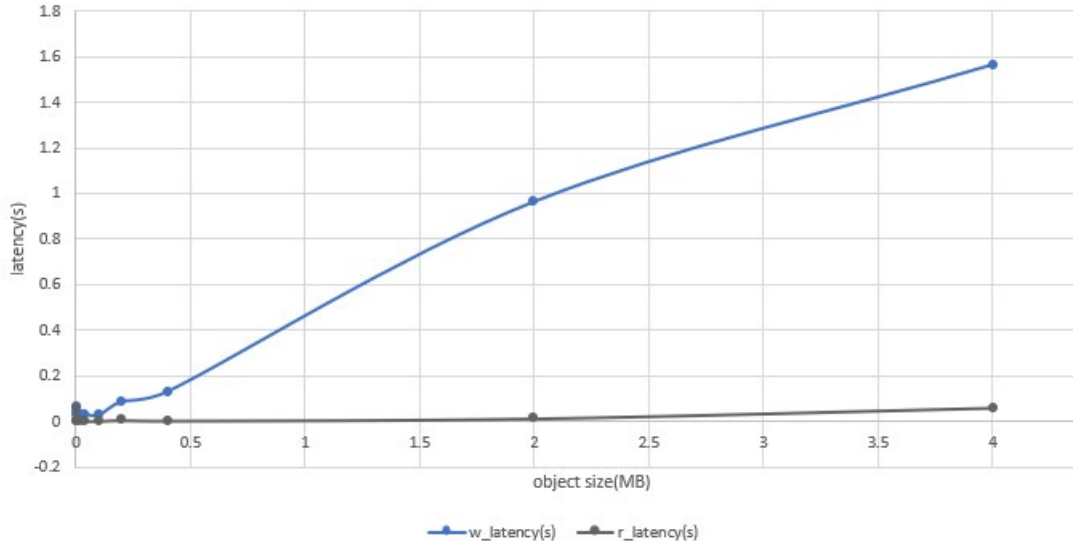


图 7 读写延迟随对象大小的变化情况

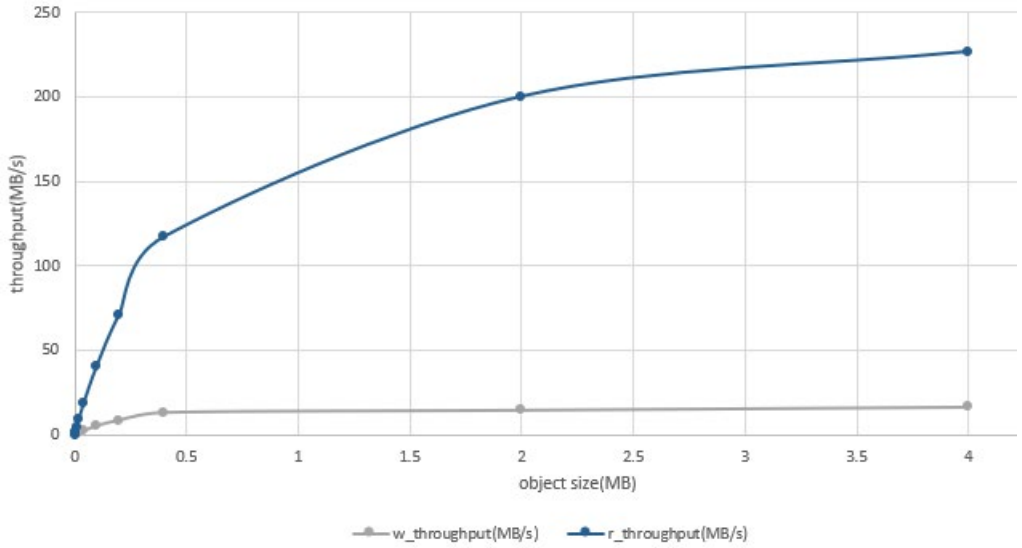


**图 8 读写吞吐量随对象大小的变化情况（去掉了 object size 为 1 的点）**

可以看到，当并行度不变的时候，吞吐量和时延随着对象大小的增大和增大，对于时延，随着对象大小的增大，时延大小类线性增加，这是和直觉比较相符的，而对于吞吐量来说，随着对象的增大，吞吐量先快速上升而后缓慢上升，这可能是由于对象存储底层实现结构所导致的，一次性取出的数据大小是有限制的。

2. 对象大小不变时，并行度改变对各因素的影响

| | client num | object size(MB) | w_throughput(MB/s) | total duration | w_latency(s) | w_90%latency(s) | r_throughput(MB/s) | total duration | r_latency(s) | r_90%latency(s) |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0.1 | 3.12 | 3.126 | 0.053 | 0.038 | 25.6 | 0.382 | 0.06 | 0.05 |
| 2 | 2 | 0.1 | 4.29 | 2.278 | 0.072 | 0.059 | 59.06 | 0.165 | 0.007 | 0.005 |
| 3 | 3 | 0.1 | 6.23 | 1.566 | 0.07 | 0.059 | 81.41 | 0.12 | 0.007 | 0.005 |
| 4 | 4 | 0.1 | 6.28 | 1.555 | 0.094 | 0.078 | 88.54 | 0.11 | 0.008 | 0.006 |
| 5 | 5 | 0.1 | 8.58 | 1.138 | 0.081 | 0.073 | 123.97 | 0.079 | 0.007 | 0.005 |
| 6 | 6 | 0.1 | 10.04 | 0.972 | 0.084 | 0.074 | 117.31 | 0.083 | 0.01 | 0.007 |
| 7 | 7 | 0.1 | 11.58 | 0.844 | 0.089 | 0.07 | 135.47 | 0.072 | 0.012 | 0.007 |
| 8 | 8 | 0.1 | 12.36 | 0.79 | 0.108 | 0.077 | 158.26 | 0.062 | 0.01 | 0.007 |
| 9 | 9 | 0.1 | 11.82 | 0.827 | 0.132 | 0.091 | 169.3 | 0.058 | 0.012 | 0.007 |
| 10 | 10 | 0.1 | 13.47 | 0.725 | 0.138 | 0.093 | 159.72 | 0.061 | 0.01 | 0.008 |
| 11 | 11 | 0.1 | 7.32 | 1.334 | 0.404 | 0.253 | 153.29 | 0.064 | 0.016 | 0.01 |
| 12 | 12 | 0.1 | 4.02 | 2.431 | 0.563 | 0.437 | 101.99 | 0.096 | 0.031 | 0.027 |

**图 9 对象大小为 0.1MB 时，各因素随并行度的变化**

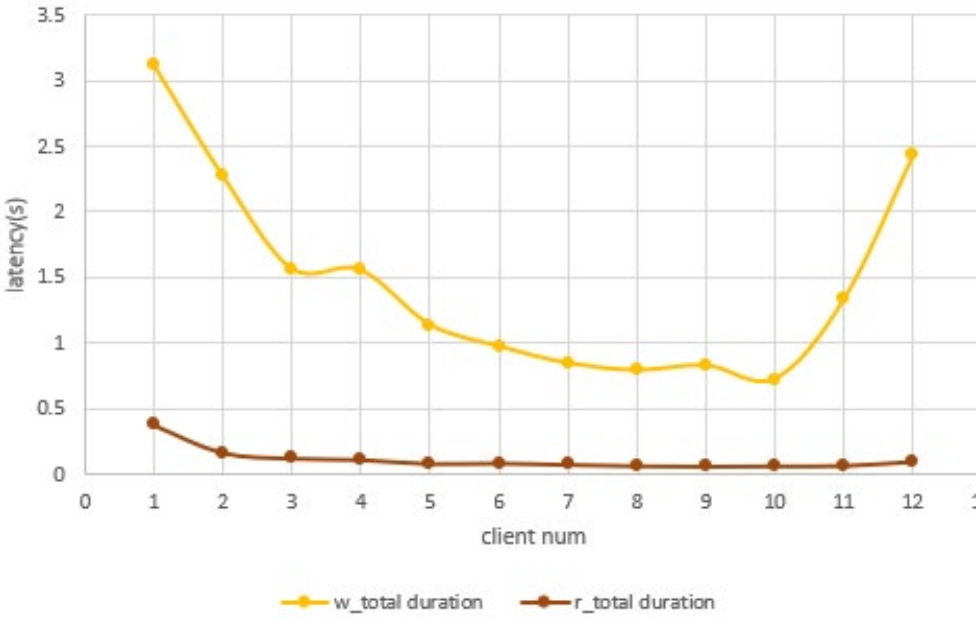


图 10 总时延随并行度的变化情况

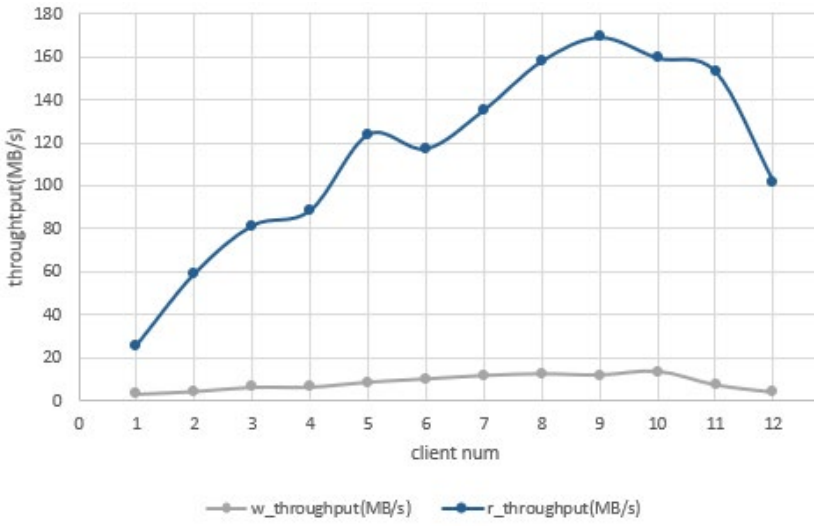


图 11 读写吞吐量随并行度的变化情况

可以看到，随着并行度的增加，读写时延先下降，后上升，而读写吞吐量先上升后下降。随着并行度的增加，时延下降，吞吐量上升这个结果是比较直观的，那为什么后面又会反过来呢，猜测可能的原因一方面集群有并发数的限制，另一方面可能是由于创建并发 client 的时间过长。

## 4.3 尾延迟探究

以 object size 为 0.2MB，client num 为 1，num sample 为 100 的请求为例，我们观察其尾延迟现象。

```
Results Summary for Write Operation(s)
Total Transferred: 1.953 MB
Total Throughput:  1.05 MB/s
Total Duration:    1.858 s
Number of Errors:  0
-----------------------------------
Write times Max:       0.096 s
Write times 99th %ile: 0.096 s
Write times 90th %ile: 0.023 s
Write times 75th %ile: 0.019 s
Write times 50th %ile: 0.016 s
Write times 25th %ile: 0.015 s
Write times Min:       0.013 s
```

图 12 object size 为 0.2MB 时的写测试结果

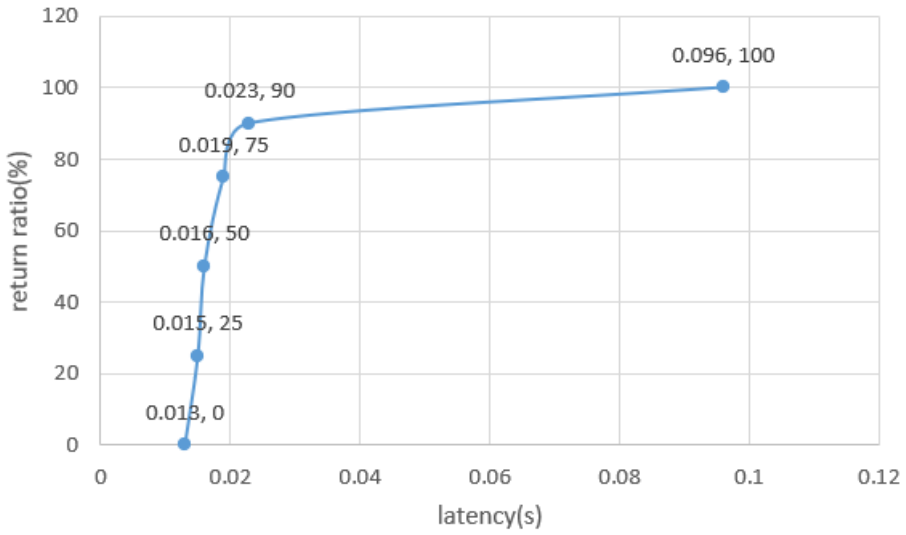


图 13 object size 为 0.2MB 时的尾延迟现象

可以看到，90%的请求都在 0.028s 内完成了，而剩下的 10%的请求远大于 0.028s 的延时，这说明了尾延迟现象是确实存在的。

现尝试使用对冲请求来减弱尾延迟现象，对冲请求的思想为当前请求的时延如果在 95%的请求时延内没有完成，则重新发起一个新的相同的请求。为了简化难度，这里对齐进行简化，通过设置一个阈值，比如当前的实验结果 90%的请求到

达的时间 0.028s，如果超过这个阈值则重新发送请求，即将剩余的 10%的请求重新发送。

由于还不是很熟悉 go 语言，这里进行进一步简化，发送剩余 10%的请求数量的新请求，即不是原来的请求。所以只需要再发送 10%的请求，并用这些请求的数据替代之后 10%的数据即可。

```
Results Summary for Write Operation(s)
Total Transferred: 0.195 MB
Total Throughput:  1.82 MB/s
Total Duration:    0.107 s
Number of Errors:  0
-----------------------------------
Write times Max:       0.025 s
Write times 99th %ile: 0.025 s
Write times 90th %ile: 0.025 s
Write times 75th %ile: 0.022 s
Write times 50th %ile: 0.021 s
Write times 25th %ile: 0.020 s
Write times Min:       0.019 s
```

图 14 10%数据的写测试结果

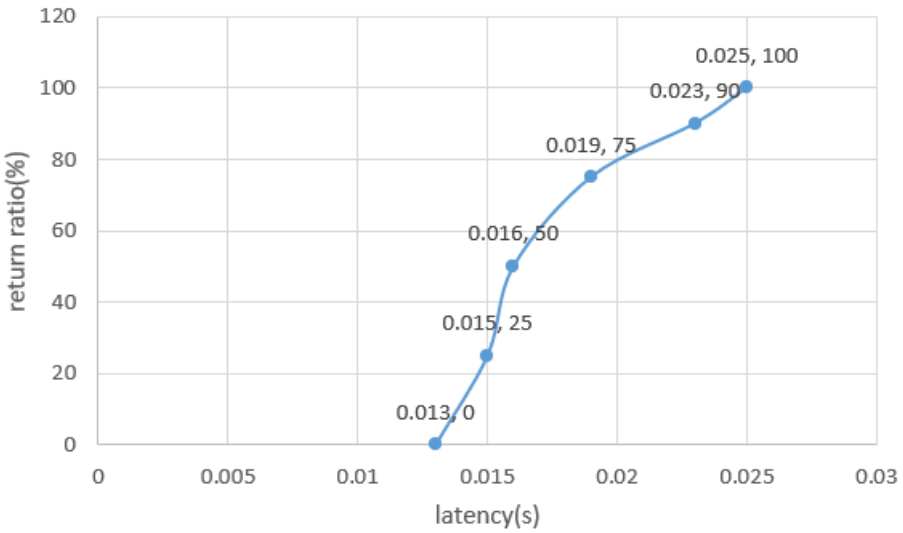


图 15 使用对冲请求后的结果

由此可见，使用对冲请求后，尾延迟效应被大幅降低了。

## 五、实验总结

经过这次实验，我了解了什么是对象存储系统，并从 0 开始搭建了一个对象存储集群 ceph 以及对应的 aws 客户端，还学会了使用相关工具去分析系统的性能并观测和简单解决了尾延迟的现象。

这个实验确实挺硬的，主要是搭建系统确实比较困难，困难的点不在于类似数学那种难想的困难，我个人认为困难的主要有两点，一方面是搜索相关资料的困难，虽然 ceph 社区目前比较繁荣，但中文资料比较少，而且也参差不齐，走了很多弯路，另一方面还是相关工具比如 docker 等使用不熟悉。

总之，经过这次实验后，感觉收获还是很大的，是专选课里面少数能真正学到东西的课程。

最后，感谢施展老师对我的答疑解惑！

## 参考文献

[1] ZHENG Q, CHEN H, WANG Y 等. COSBench: A Benchmark Tool for Cloud Object Storage Services[C]//2012 IEEE Fifth International Conference on Cloud Computing. 2012: 998－999.

[2] ARNOLD J. OpenStack Swift[M]. O’Reilly Media, 2014.

[3] WEIL S A, BRANDT S A, MILLER E L 等. Ceph: A Scalable, High-performance Distributed File System[C]//Proceedings of the 7th Symposium on Operating Systems Design and Implementation. Berkeley, CA, USA: USENIX Association, 2006: 307－320.

[4] Dean J, Barroso L A. Association for Computing Machinery, 2013. The Tail at Scale[J]. Commun. ACM, 2013, 56(2): 74－80.

[5] Delimitrou C, Kozyrakis C. Association for Computing Machinery, 2018. Amdahl’s Law for Tail Latency[J]. Commun. ACM, 2018, 61(8): 65－72.