2020 级

《大数据存储系统与管理》课程

实 验 报 告

姓　名　李茗畦

学　号　U202015630

班　号　本硕博 2001 班

日　期　2023.05.25

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

伴随着数据规模的持续扩张，大数据存储越来越面领着挑战，包括数据量的爆炸性增长、数据的多样性和复杂性、数据的可靠性和可扩展性需求以及数据的安全性。

对象存储系统通常指的是一种用于存储和管理大规模数据的分布式存储系统。对象存储系统将数据以对象的形式进行存储，每个对象都有唯一的标识符，并且可以包含数据本身以及与之相关的元数据。这种存储方式使得对象存储系统能够扩展到极大规模，并提供高度可靠性和可用性。对象存储系统在面对大数据存储挑战时提供了解决方案，能够扩展到海量数据规模，保证数据的可靠性，支持多种数据访问模式。

常见的对象存储系统有 Amazon S3,Google Cloud Storage 以及 OpenStack Swift。本次实验将基于 OpenStack Swift 实现。Swift 采用分布式架构，数据被分散到多个存储节点。Swift 使用异步复制机制实现数据冗余，当存储节点发生故障，Swift 可以自动将数据从冗余副本中恢复，确保数据的可靠性。Swift 提供了多种数据访问方式，包括基于 RESTful 的 API 和命令行工具，使得用户可以方便地存储、检索和管理对象数据。它还支持与其他 OpenStack 组件和第三方工具的集成，提供灵活的数据管理和应用开发接口。本实验中我参考了 19 级张家荣学长的实验报告，利用 python-swiftclient 提供的的 API 加服务器上在 docker 部署的 openstack swift 完成对象存储系统的客户端和服务端的搭建。

三、实验环境

对象存储系统的客户端在本地部署，安装 python 中的 python-swiftclient 库，利用它提供的函数完成对服务端的访问。

• 对象存储系统客户端实验环境

操作系统	Microsoft Windows 11 家庭中文版
处理器	Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
内存	16G
对象存储客户端	python-swiftclient 4.3.0 + python 3.9.12

由于本地的 C 盘空间不够了，对象存储系统的服务端安装在服务器上，利用 github 课程仓库中的 docker 镜像部署服务端。

• 对象存储系统服务端实验环境

操作系统	Ubuntu 16.04 LTS
处理器	Intel(R)Xeon(R) Platinum 8378C CPU @2.80GHZ
内存	4GiB
对象存储服务端	docker 20.10.7 + OpenStack-Swift

四、实验内容

首先，将完成对象存储系统的客户端和服务端的搭建，并简单的进行数据传输，验证系统成功部署。接着，将在上一部分搭建的系统上分析影响对象存储系统性能的指标，包括对象尺寸，并发性等。在第三部分，将分析对象存储系统中的尾延迟现象。

4.1 对象存储技术实践

实验的第一部分是完成一个基于 OpenStack-Swift 的对象存储系统的客户端和服务端的搭建，并在客户端利用一些 swift 命令实现一些简单的操作，比如查看容器，向服务端写入或者下载一个文本文件，查看服务端的信息等。

4.2 对象存储性能分析

实验的第二部分是在基于 OpenStack-Swift 的对象存储系统上分析对象的尺寸和并发度对存储系统性能的影响。将改变单个传输对象的尺寸以及负责读写操作的线程数，分析吞吐率、平均延迟等指标的变化，探究对象尺寸和并发度对系统性能的影响。

实验的第三部分是分析对象存储系统的尾延迟现象。实验的第二部分关注的是一个传输任务的过程中整体的吞吐率。这一部分将关注一个传输任务中对每个传输对象的传输延迟，因为单个对象的较大的延迟会影响整个传输任务的总用时。这一部分将分别分析客户端向服务端发起的一次读操作和写操作的过程中出现的尾延迟。并尝试使用对冲请求对尾延迟进行优化。

五、实验过程

5.1 实验一 系统搭建

a. 服务端的搭建

在本节实验中，在服务器中安装 Openstack Swift 作为对象存储系统的服务端。根据实验仓库中的 openstack-swift-docker 完成在 docker 中部署 Openstack Swift。之后，调用 docker ps 命令检查是否搭建成功，输出如下所示。接下来在服务端输入命令 docker run -d --name openstack-swift -p 12345:8080 --volumes-from SWIFT_DATA -t openstack-swift-docker 启动该容器，之后客户端边可通过服务器的 ip 地址加上 12345 端口号访问容器。

```
root@lmq:~# docker ps
CONTAINER ID   IMAGE                     COMMAND                  CREATED       STATUS
  PORTS                                         NAMES
9f0e7ef9fe5d   openstack-swift-docker    "/bin/sh -c /usr/loc…"   4 hours ago   Up 4 hours
  0.0.0.0:12345->8080/tcp, :::12345->8080/tcp   openstack-swift
```

b.客户端的搭建

客户端在本地搭建，使用 python 中的 python-swiftclient 库来和 Swift 进行交互。利用 pip install python-swiftclient 安装所需的库。输入以下命令来对服务端的 swift 进行一次访问，输出服务端的相关信息比如容器数等，验证 python-swiftclient 安装成功和服务端的成功启动。

```
lmq@LAPTOP-TTBI6Q3I:/mnt/c/Users/李茗畦/Desktop$ swift -A http://123.60.165.6:12345/
auth/v1.0 -U test:tester -K testing stat
                        Account: AUTH_test
                     Containers: 3
                        Objects: 6780
                          Bytes: 4001097
Containers in policy "policy-0": 3
   Objects in policy "policy-0": 6780
     Bytes in policy "policy-0": 4001097
                   X-Timestamp: 1684248489.74169
                  Content-Type: text/plain; charset=utf-8
                 Accept-Ranges: bytes
                    X-Trans-Id: txbe232e76699b443883730-006464c3d8
```

接下来我们调用 swift 命令向服务端传送一个文档，-A 选项指定服务端的 url，由服务器端的 ip 地址和端口号组成，-U 选项指定用户名，-K 选项指定密码，upload swift ./helloworld.txt 表示由客户端向服务端中的容器传送文件 helloworld.txt。

```
lmq@LAPTOP-TTBI6Q3I:/mnt/c/Users/李茗畦/Desktop$ swift -A http://123.60.165.6:1
2345/auth/v1.0 -U test:tester -K testing upload swift ./helloworld.txt
```

通过在 swift 命令中添加 list 选项查看服务端的容器，输出如下图，可见新建了一个名称为 swift 的容器。

```
lmq@LAPTOP-TTBI6Q3I:/mnt/c/Users/李茗畦/Desktop$ swift -A http://123.60.165.
6:12345/auth/v1.0 -U test:tester -K testing list
lab2bucket
swift
testbucket
```

在 swift 命令中添加 download 选项即可从服务端下载文件到客户端，下载客户端 swift 容器中的 helloworld.txt 到本地，该命令的输出如下所示。

```
lmq@LAPTOP-TTBI6Q3I:/mnt/c/Users/李茗畦/Desktop$ swift -A http://123.60.165.6:1
2345/auth/v1.0 -U test:tester -K testing download swift ./helloworld.txt
./helloworld.txt [auth 0.058s, headers 0.144s, total 0.146s, 0.000 MB/s]
```

5.2 实验二 性能观测

a.探究对象尺寸对性能的影响

我参考了 19 级张家荣学长的程序编写了测试程序。该实验的测试程序利用 python-swiftclient 库提供的 API 实现对 openstack swift 的读写请求。利用 swiftclient 中的 Connection 方法建立和 swift 服务端的连接，利用 put_container 和 delete_container 建立和删除，利用 get_object，put_object 和 delete_object 来对指定容器中的指定对象完成读取、写入和删除的操作。我们保持一次读写的总数据量 total_size 不变，每一次只修改对象尺寸 object_size，并保证对象尺寸和对象数量的乘积不变，计算相应的对象数量 num_sample。

在这里，我们将数据总量设置为 1MB，令 object_size 分别为 1，2，4，8，16，32，64，128，512，1024KB，并发数设置为 10，从线程池中申请 10 个线程负责对这些对象文件的读写。

测试程序的具体步骤如下：

1. 利用 swiftclient 中的 Connection 方法和服务端建立连接，服务端的 url 由远程服务器的 ip 加端口号 12345 组成，用户名和密码都为默认值，用户名为 test:tester，密码为 testing；
2. 在每一轮的测试的开始，新建一个大小为 object_size 的测试对象文件，用于客户端和用户端之间传输。object_size 依次取 1，2，4…1024KB；
3. 在服务端新建一个测试用的名字为”lab2bucket”的容器，如果发现该容器已存在，则先该旧的容器删除，在新建一个名字为”lab2bucket”的容器，否则新建一个已有的容器会报错；
4. 接下来，先进行有客户端向服务端的写操作测试。调用 ThreadPoolExecutor 方法申请一个线程数为 num_client，也就是并发数为 10 的线程池。将由这 10 个线程负责 num_sample 个对象的写操作，num_sample 即对象数量，为总数据量/对象尺寸。我们记录写入的总用时，以及每一个对象的延迟。
5. 然后由客户端向服务端发送读请求，将上一步写入的 num_sample 个对象下载到本地。该操作依旧由一个线程数为 num_client 的线程池负责。记录下

载的总用时，以及每一个对象的延迟。

6. 计算读写的平均延迟，和吞吐率。

最终利用将运行结果绘制图像，图 5.1 是对象尺寸依次取 1，2，4…1024KB 时是客户端向服务端读、写时的吞吐率，图 5.2 和图 5.3 是客户端向服务端写、读时的平均延迟。由于写平均延迟和读平均延迟的数值相差较大，因此分为两个折线图绘制。

以写吞吐率为例说明对象尺寸对性能的影响，由图 5.1 可知，当对象尺寸由 1KB 增长到 256KB 时，客户端向服务端的写吞吐率显著提升。处理一个较大的对象比处理若干的小对象性能更佳。因为相比处理多个小对象，处理少个大对象将不必花费多余的时间和资源在建立连接等和数据本身的传输无关的操作。

当对象尺寸由 256KB 再增长到 512KB，写吞吐率则没有明显增加。该实验的传输数据总量为 1MB，当对象尺寸超过 256KB 时，我们只需要不超过 4 个线程负责。因此只需要从线程池中选取一部分线程执行一次写操作即可，此时增加对象尺寸，建立连接、发送 http 请求等操作的开销也不会减少。因此，当对象尺寸从 256KB 继续增长，写吞吐率不再增加。读吞吐率和写吞吐率具有相似的变化趋势，但是读吞吐率的值却远远小于写吞吐率。我使用 speedtest 命令查看本地的 updoad 带宽为 75.85 Mbit/s，Download 带宽为 64.71Mbit/s，服务端的 upload 带宽为 1.10Mbit/s， Download 带宽为 32.60Mbit/s。因此写操作比读操作要快的多。

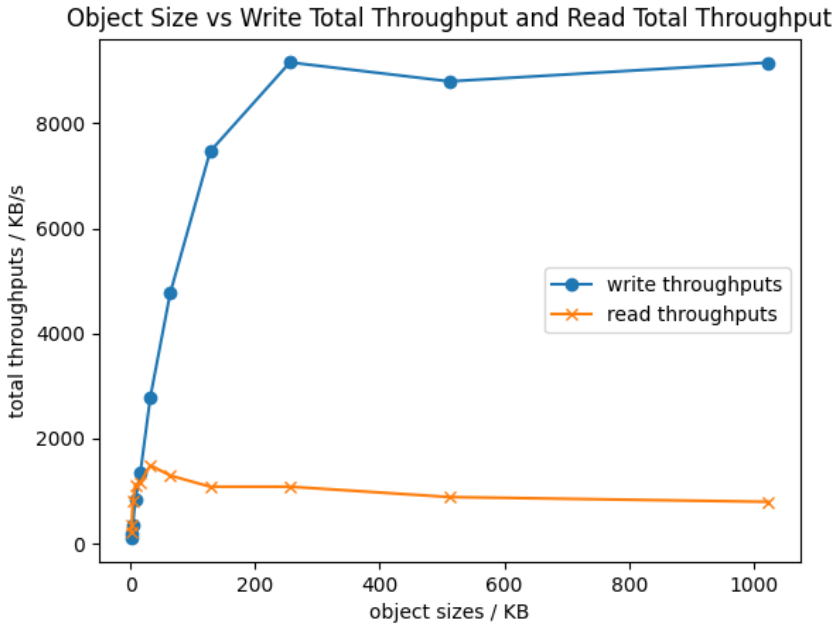

图 5.1 读、写吞吐量随对象尺寸的变化曲线

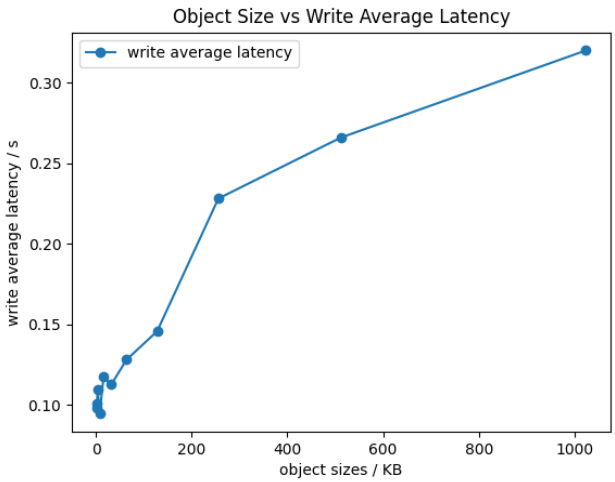

图 5.2 写平均延迟随对象尺寸的变化曲线

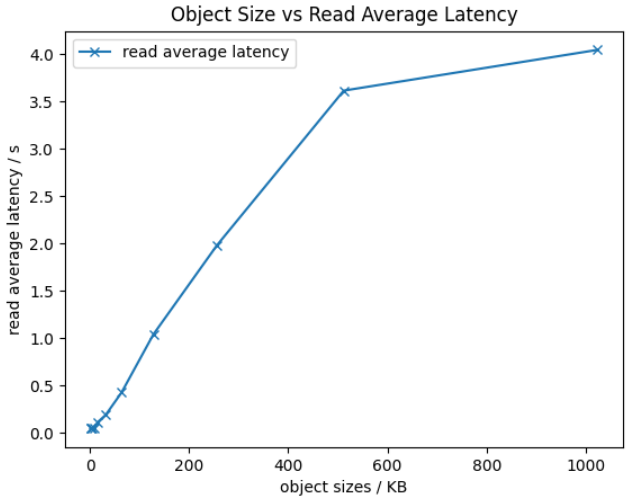

图 5.3 写平均延迟随对象尺寸的变化曲线

以写平均延迟来说明对象尺寸对性能的影响。由图 5.2 可知，当对象尺寸有 1KB 增长到 32KB，写平均延迟有微小的波动，没有明显增长；当对象尺寸由 64KB 增长到 1024KB 时，写平均延迟开始有明显的增长，随对象尺寸大约呈正比例增长。这可能是由于，当对象尺寸较小时，I/O 带宽并没有跑满，因此随着对象尺寸的增加，写平均延迟保持较平稳。当对象尺寸较大时，I/O 的带宽跑满，成为制约服务端和客户端之间传输的主要制约因素，因此随着对象尺寸的增加，完成一个对象的传输的延迟也呈正比例增长。

b.探究并发度对性能的影响

本节实验中将探究并发数，即程序中的参数 num_client，对对象存储系统的读、写吞吐率和平均延迟的影响。将传输的数据总量 total_size 设为 4MB，对象尺寸 object_size 为 8KB，对象数量 num_samples 为 total_size/object_size=512，这三个参数保持恒定。并发数 num_client 从 1,2,…10,20,30…130,140 中逐个取值。

本节实验的测试程序和上一节的程序较类似，只是申请线程池时修改申请的线程数。在 num_client 取不同值的情况下，客户端向服务端分别读和写总量为 total_size，对象尺寸为 object_size，对象数量为 num_samples 的数据。记录传输的总时间，以及每个任务的延迟，计算出吞吐率和平均延迟。将实验数据绘制出图像，读写吞吐和并发数的图像如图 5.4 所示，读写平均延迟和并发数的图像如图 5.5 所示。

下面以写吞吐率说明并发数对性能的影响。当并发数由 1 增加到 20，随着并发数的增加，写的吞吐率明显增加，从大大约 171.35KB/s 增长到大约 1008KB/s。这是因为使得负责客户端向服务端写入的线程数目的增加提升了系统的并行度，因此吞吐率有了明显的提高。当并发数继续增加，读、写的吞吐率则不再继续增长，且呈现不稳定的上下波动。这时客户端和服务端之间的 I/O 带宽跑满，因此增

加并行度不会再提升吞吐率。读吞吐率和写吞吐率有着相似的变化趋势。

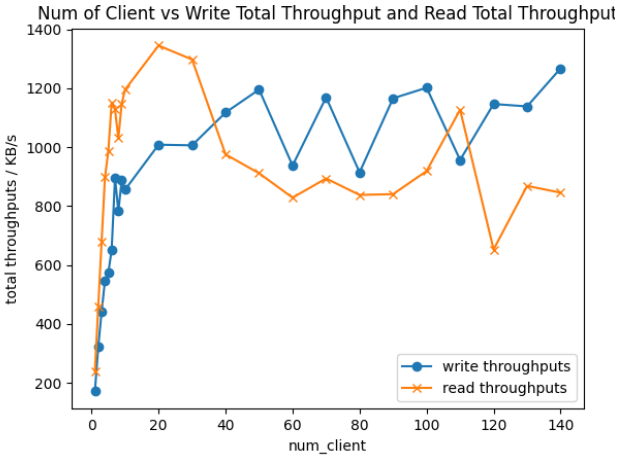

图 5.4 读、写吞吐量随并发度的变化曲线

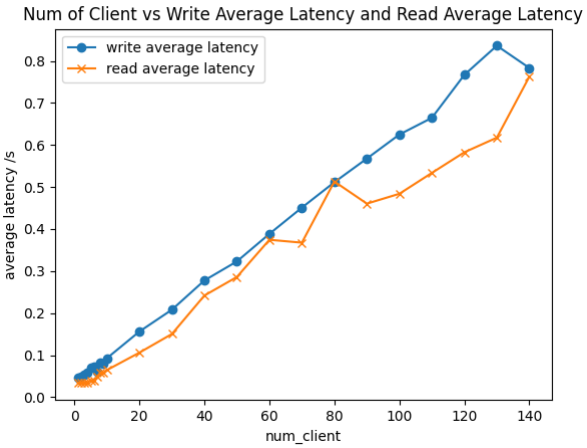

图 5.5 读、写平均延迟随并发度的变化曲线

对于平均延迟来说，写操作和读操作的平均延迟都随着并行度的增加而增加。这是由于线程数的增加，每个线程可利用的带宽随着并行度的增长而减小。

这里我只把并发度测试到 140 而没有再继续增加。在实验中，当并发度超过 150 后，程序运行过程中就会出现 409 Confilct 的报错，我还没有弄懂出现这个问题的原因。

c.观察尾延迟

尾延迟指的是在一个对象存储系统中，有少数的请求的响应时间远远大于绝大多数的请求的响应时间。在实际场景中，这会导致用户由于少数的请求而经历较长时间的等待。

下面我们将通过一个由客户端向服务端发起的写请求来观察对象存储系统中的尾延迟现象。将写操作的数据总传输量设为 1MB，单个对象的尺寸在 4KB，一共有 256 个对象，并发度为 10。接下来将 256 个对象的传输任务分配给 10 个线程，并记录每一个对象写入任务的开始和结束时间，计算该对象的传输延迟。当这 256 个对象都写入完成后，我们统计这些对象的传输延迟。对于读操作我们也做类似的测试。

最终，在数据总传输量为 1MB，单个对象尺寸为 4KB，并发度为 10 的条件下，一次写操作和一次读操作的测试结果如表 5-1 所示。

参考实验仓库中的 latency-plot.ipynb，我绘制出 256 个对象的传输延迟的图像。其中，写操作的传输延迟的图像如图 5.6 和图 5.7 所示。由图 5.6 可以看出，超过 90%的对象的传输延迟都在 120ms 以下，在图 5.7 中，大部分对象的传输延迟都在 100ms 附近上下波动，只有少数的几个对象的传输延迟远远高于其他对象的传输延迟，是大部分对象传输延迟的二倍以上。在横坐标为 0 和 200 附近可以观察到明显的峰值。

表 5.1 对象的写、读延迟

指标	写操作	读操作
吞吐率 KB/ s	409.0457	125.8906
最大延迟 / ms	188.3337	2247.0383
99th 延迟 / ms	185.2571	1764.7695
90th 延迟 / ms	113.5320	659.7630
75th 延迟 / ms	100.2464	350.8756
50th 延迟 / ms	90.3663	141.8530
25th 延迟 / ms	83.7922	77.6975
平均延迟 / ms	95.4167	282.3885

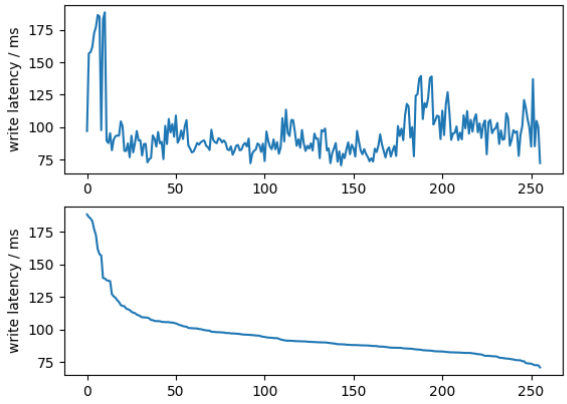

图 5.6

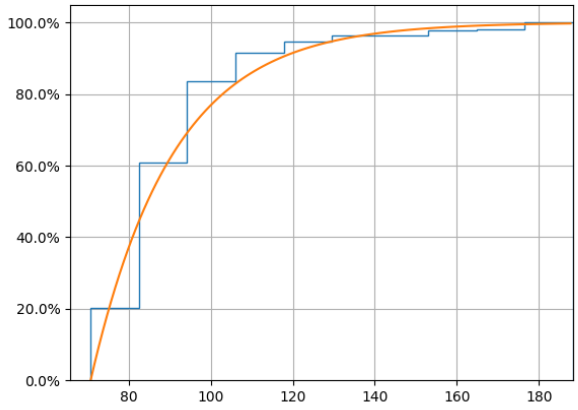

图 5.7

读操作的对象的传输延迟的图像如图 5.8 和图 5.8 所示。和写操作的图像相比，读操作的尾延迟的现象则更加明显。由图 5.9 可以观察出，超过 90%的对象的传输延迟都在 250ms 以下，对于剩下的对象，他们的传输延迟则大大增加。正如表 5-1 所呈现的，读操作和写操作中 50%th 的传输延迟相差不大，而 99%th 延迟和最大延迟却相差将近一个数量级。写操作的 99%th 延迟仅为平均延迟的 2 倍，而读操作中 99%th 延迟为平均延迟的将近 10 倍，可见在这个对象存储系统中，读操作的尾延迟相较于写操作更为严重，读操作的性能大大受限于尾延迟。这是由于客户端的 Download 带宽小于 Upload 带宽而服务端的 Upload 带宽小于 Downolad 带宽，使得读操作可利用的最大带宽小于写操作。

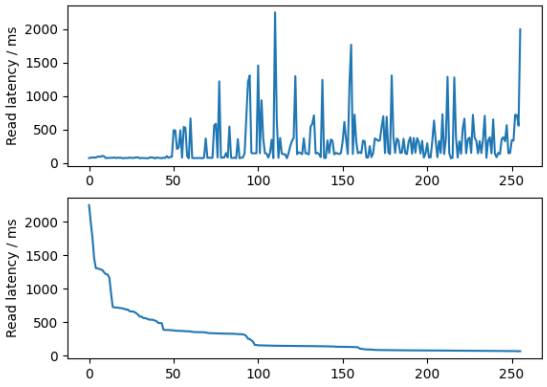

图 5.8

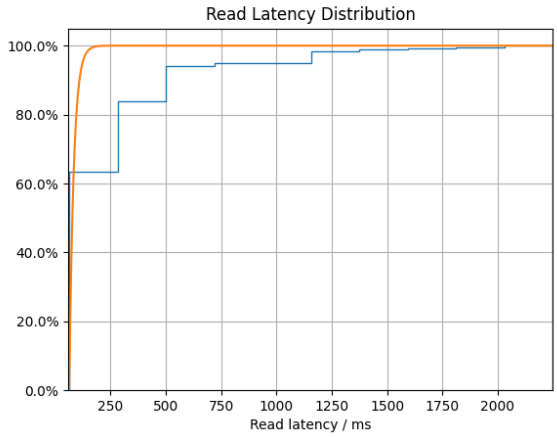

图 5.9

5.3 实验三 尾延迟挑战

在本节，将尝试使用对冲请求优化尾延迟。在上一个实验中读操作有很严重的尾延迟，因此在这一节将对读操作进行优化。

具体的步骤是如下：

1. 对于每一个对象，首先发出了一个读请求；
2. 父线程将等待一个对冲等待时间，如果子线程在这之前结束，那么则继续处理其他请求；
3. 如果超过这个时间，父线程将再申请若干个线程，只要其中一个线程结束，父进程就将其他子线程结束，继续执行下一个请求。

开始我采用的参数是上一个实验的参数，但是发现对冲请求的优化很有限，不管如何调整等待时间和对冲请求的线程数，传输延迟都没有明显减少甚至会增加。考虑到可能是客户端和服务端之间的带宽限制，因此我将传输任务的传输数据总量，对象尺寸和并行度都减少了一些，并换用了带宽更大的服务器。传输任务为向服务端读取 512KB 的数据，对象尺寸为 0.5KB，并行度为 4。首先进行一次不采用对冲请求的读操作，收集数据计算其平均延迟和其他延迟变量。我们将 90th 延迟作为对冲请求时间，对冲请求线程数暂时设为 2，采用对冲请求并收集延迟数据。

经过统计，采用对冲请求和不采用对冲请求的延迟对比图如图 5.10 所示。由图可以看出，通过对冲请求，最大延迟和最慢的 1%的延迟都得到了很大的程度的优化，但是由于带宽优先，当较慢的请求发起对冲请求时会和其他请求抢占带宽，导致其他请求的延迟有了些许的增长。两种情况下每个对象的延迟情况和分布情况如图 5.11-5.14 所示。由图 5.11 和图 5.12，图 5.13 和图 5.14 的对比可见，在对冲请求下，各个对象之间的延迟较为均衡，不再有少部分对象的延迟远远高于其他对象。

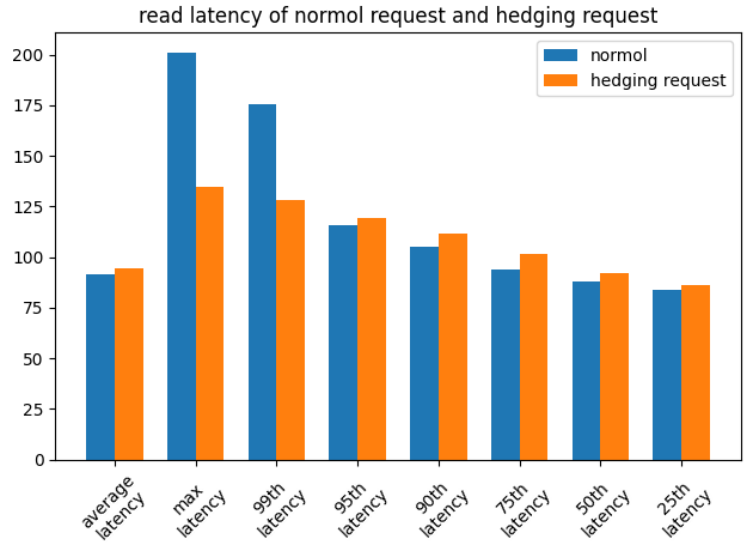

图 5.10 无对冲请求和对冲请求时的延迟对比

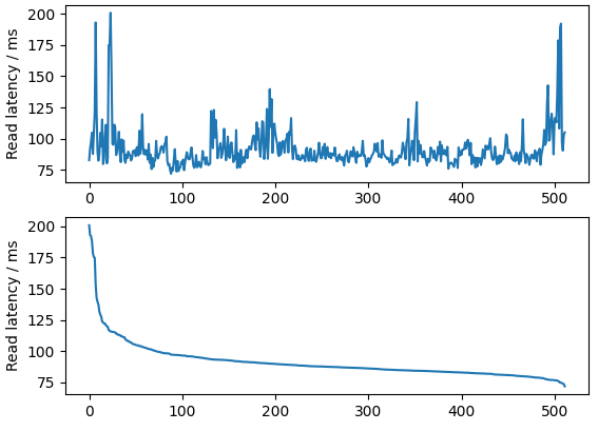

图 5.11 无对冲请求下的延迟

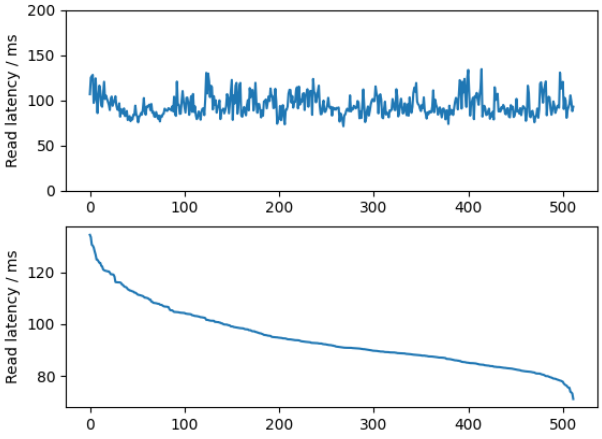

图 5.12 对冲请求下的延迟

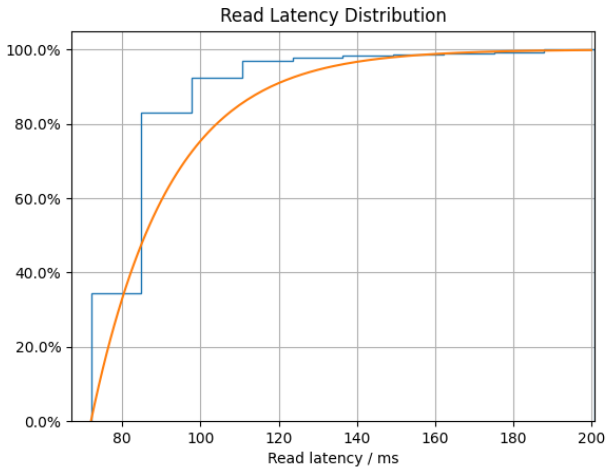

图 5.13 无对冲请求下的延迟分布

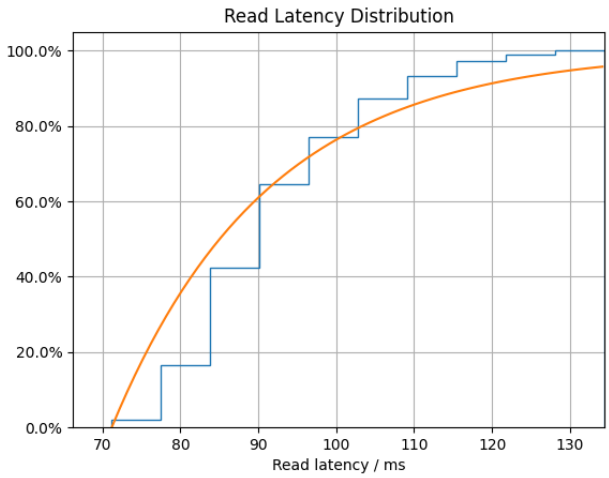

图 5.14 有对冲请求下的延迟分布

六、实验总结

在本实验中我了解了对象存储系统，开始部署了 minio 对象存储系统，然后又进一步部署了 OpenStack-Swift 对象存储系统，对对象存储系统有了基本的了解。

接下来我探究了对象存储系统中对象尺寸和并发度对存储系统的性能的影响。这启示我们在设计存储系统时应根据 I/O 带宽等因素采用合适的对象尺寸和并发度。我观察了存储系统中的尾延迟现象，并尝试使用对冲延迟对尾延迟进行了一定程度的优化。

经过本次实验，我对大数据系统的存储有了进一步的认识和了解，并将在将来进行进一步的学习。

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.