2020 级

《大数据存储系统与管理》课程

实 验 报 告

姓　名　付景洋

学　号　U202015424

班　号　计算机 2005 班

日　期　2023.5.25

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

对象存储是一种将数据集合作为单个、唯一可识别的不可分割项或对象存储和检索的方法。它适用于可以封装和管理为对象的任何形式的数据。

对象被视为一个原子单元。与文件系统中的目录层次结构相对应的结构不存在；每个对象在系统中通过唯一的对象标识符进行唯一标识。

在这种存储类型上创建对象时，整个数据集合都被处理和处理，而不考虑它可能具有的子部分。在从对象存储中读取时，可以读取整个对象，或者要求读取其部分。通常无法更新对象或对象的部分，需要重新编写整个对象。大多数对象存储允许删除对象。

存储系统不关心对象包含的内容。它们可以是简单的数据集合、文件、整个文件系统、视频、虚拟机或容器、数据库等等，因为存储系统只将对象视为与一块数据相关联的对象标识符。

对象存储如何存储对应于应用程序并不重要。许多系统使用基于文件或块的存储；有些系统专门设计了直接支持对象的设备。无论如何存储，所有对象存储系统都支持简单的 API 来创建、读取和删除对象，并搜索元数据。其中一些较为知名的是 SNIA 的云数据管理接口（CDMI）和 Amazon 的 S3，它们通过 HTTP 协议进行操作使得对象存储对 Web 和云的使用具有吸引力。

OpenStack Object Store 项目，也被称为 Swift，提供了云存储软件，可以通过简单的 API 存储和检索大量数据。它专为大规模操作而构建，并在整个数据集上进行了耐久性、可用性和并发性的优化。Swift 非常适合存储可以无限增长的非结构化数据。

三、实验环境

本实验在个人电脑采用 VMware 上 Ubuntu 虚拟机安装 docker 搭建对象存储系统。

```
❯ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.2 LTS
Release:        22.04
Codename:       jammy
```

为虚拟机分配了 12 个内核和 20G 的存储空间。

```
❯ df -h
Filesystem      Size  Used Avail Use% Mounted on
tmpfs           389M  2.1M  387M   1% /run
/dev/sda3        20G   11G  8.0G  56% /
tmpfs           1.9G     0  1.9G   0% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
/dev/sda2       512M  6.1M  506M   2% /boot/efi
tmpfs           389M  2.4M  387M   1% /run/user/1000
/dev/sr0        145M  145M     0 100% /media/pandora/CDROM
❯ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         45 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  12
  On-line CPU(s) list:   0-11
Vendor ID:               AuthenticAMD
  Model name:            AMD Ryzen 5 4600H with Radeon Graphics
```

安装了 docker、python-swiftclient 等相关软件。

```
❯ docker -v
Docker version 24.0.1, build 6802122
❯ python3 -V
Python 3.10.6
❯ swift --version
python-swiftclient 4.3.0
```

四、实验内容

4.1 对象存储技术实践

根据个人情况选择适合自己的对象存储实验系统并完成对象存储系统搭建，并体验对象存储系统的功能服务。使用客户端发送 PUT 、GET、DELETE 请求，在服务端终端以及本地文件夹下观测服务端的响应结果。

本实验选用 OpenStack Swift 搭建对象存储系统，使用 Docker OpenStack Swift onlyone 简易容器部署，使用 python-swiftclient 提供的 Swift API 编写 python 程序进行标准测试，通过测试吞吐率、延迟等指标，分析对象尺寸、并发率等因素对性能的影响。

4.2 对象存储性能分析

在进行性能观测中容易发现对象存储系统存在尾延迟问题，通过绘制 eCDF 量化观察尾延迟现象，思考解决尾延迟问题的方法。

解决尾延迟问题主要有对冲请求和关联请求两种方案。延迟高的情况为小概率事件，一个简单的想法是客户端同时对多个服务器发送请求，接受第一个响应便能多个延迟取最小，然而这种方法会使负载大大增加，对冲请求就是该方法的一个变种，它在第一个请求的响应时间超过 95%的延迟时才发送第二个请求，减少额外负载的同时显著缩短尾延迟。使用更激进的对冲请求方案(95%的比例改为更小)则会消耗更多的资源，当请求在某个服务器开始执行时，及时关闭其他服务器上的该请求有助于减少资源消耗。关联请求的方法就是将服务器相互关联，客户端同时向多个服务器发送请求，服务器收到的请求中包含其他服务器的信息，当第一个服务器开始执行时就能及时向其他服务器发送关闭该请求的信息。

五、实验过程

5.1 对象存储系统搭建

拉取 fnndsc/docker-swift-onlyone 镜像，准备 docker 数据卷，创建 swift-onlyone 容器并挂载到其中，并设置用户名和密码为 hustobs。

```
❯ docker pull fnndsc/docker-swift-onlyone
Using default tag: latest
latest: Pulling from fnndsc/docker-swift-onlyone
e706e0a9f423: Pull complete
6b25b47bb79d: Pull complete
593e9657e0bb: Pull complete
a439e0573b42: Pull complete
5211ad9e3be6: Pull complete
b5664dfbcd4d: Pull complete
7db7c9626be2: Pull complete
1a49ec5e6eaf: Pull complete
Digest: sha256:f914a677b542420a05e34c7fba36f750a4989e458bdb3e114c3e93fa5a96ed51
Status: Downloaded newer image for fnndsc/docker-swift-onlyone:latest
docker.io/fnndsc/docker-swift-onlyone:latest
❯ docker volume create swift_storage
swift_storage
❯  docker run -d --name swift-onlyone -p 12345:8080 -e SWIFT_USERNAME=hustobs:hustobs -e SWIFT_KEY=key -v swift_storage:/srv -t fnndsc/docker-swift-onlyone
cd9364d3a1165c87e9ed24a12c67286508ae24619913904e1c25c372da293546
```

查看容器创建成功。

```
❯ docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED         STATUS         PORTS                                                NAMES
cd9364d3a116   fnndsc/docker-swift-onlyone   "/bin/sh -c /usr/loc…"   2 minutes ago   Up 2 minutes   0.0.0.0:12345->8080/tcp, :::12345->8080/tcp   swift-onlyone
❯ swift -A http://127.0.0.1:12345/auth/v1.0 -U hustobs:hustobs -K key stat
              Account: AUTH_hustobs
           Containers: 0
              Objects: 0
                Bytes: 0
         Content-Type: text/plain; charset=utf-8
          X-Timestamp: 1684918383.32001
      X-Put-Timestamp: 1684918383.32001
           X-Trans-Id: tx8fa349c9999446baab3b7-00646dd06f
X-Openstack-Request-Id: tx8fa349c9999446baab3b7-00646dd06f
```

5.2 对象存储服务使用

在计算机领域中，create, read, update, and delete (缩写为 CRUD) 是访问持久存储的 4 项基本操作，在 HTTP 中体现为 PUT、GET、DELETE 三种核心操作。

python-swiftclient 同时提供了 CLI 和 API。

验证 Swift CLI 的功能，首先创建新容器 new-container，向容器中上传大小为 4K 的本地文件 test.bin，将其对象命名为 test-bin，再从容器中下载该对象，最后分别删除对象和容器。

```
❯ swift -A http://127.0.0.1:12345/auth/v1.0 -U hustobs:hustobs -K key list
❯ swift -A http://127.0.0.1:12345/auth/v1.0 -U hustobs:hustobs -K key post new-container
❯ swift -A http://127.0.0.1:12345/auth/v1.0 -U hustobs:hustobs -K key list
new-container
❯ swift -A http://127.0.0.1:12345/auth/v1.0 -U hustobs:hustobs -K key upload new-container ./test.bin --object-name test-bin
test-bin
❯ swift -A http://127.0.0.1:12345/auth/v1.0 -U hustobs:hustobs -K key list new-container
test-bin
❯ swift -A http://127.0.0.1:12345/auth/v1.0 -U hustobs:hustobs -K key download new-container test-bin
test-bin [auth 0.007s, headers 0.015s, total 0.015s, 0.479 MB/s]
❯ swift -A http://127.0.0.1:12345/auth/v1.0 -U hustobs:hustobs -K key delete new-container test-bin
test-bin
❯ swift -A http://127.0.0.1:12345/auth/v1.0 -U hustobs:hustobs -K key list new-container
❯ swift -A http://127.0.0.1:12345/auth/v1.0 -U hustobs:hustobs -K key delete new-container
new-container
❯ swift -A http://127.0.0.1:12345/auth/v1.0 -U hustobs:hustobs -K key list

 ~/hustcs/storage
❯
```

实际上在进行存储性能分析时采用 Connection API，其中并发请求部分参考了 obs-tutorial 中的 latency-collect.ipynb，通过设置不同的客户端线程数量、对象尺寸和对象数量测试吞吐率、延迟等指标。

客户端进行连接的代码如下图所示。

```
import swiftclient

endpoint = 'http://localhost:12345/'
_user = 'hustobs:hustobs'
_key = 'key'

conn = swiftclient.Connection(
    authurl=endpoint+'auth/v1.0',
    user=_user,
    key=_key
)
```

调用 Swift API 接口 PUT、GET、DELETE 的代码如下图所示。

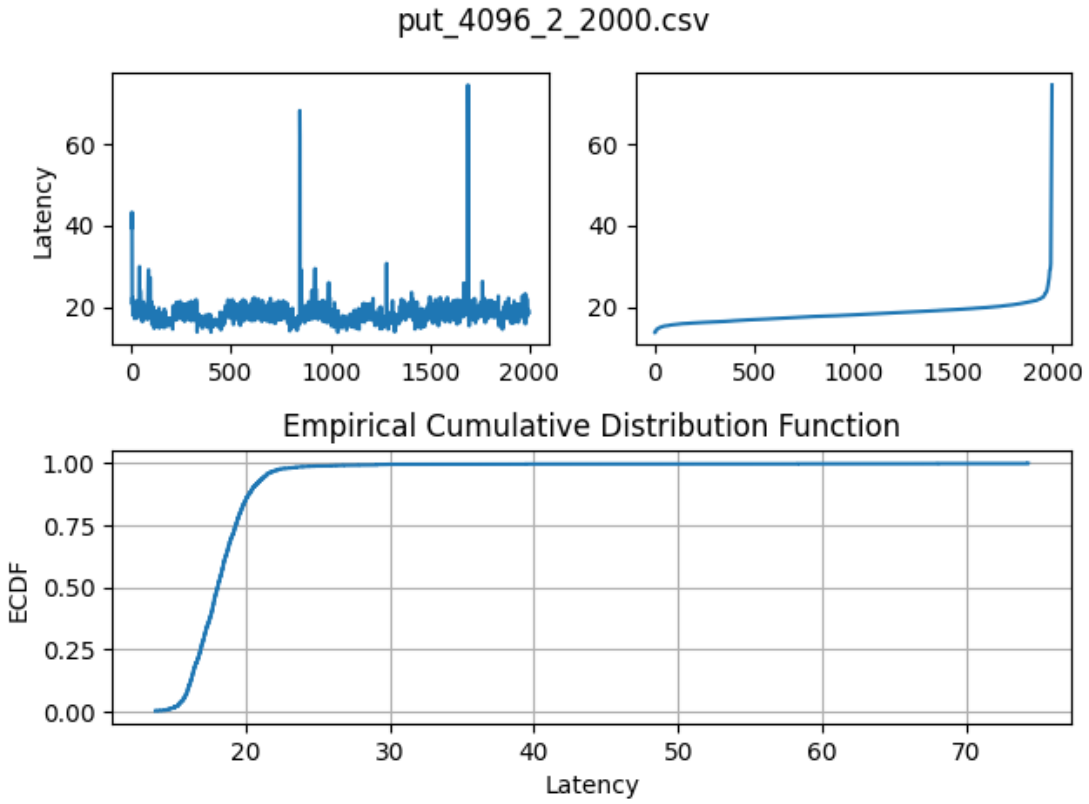
```
def bench_put(i):
    obj_name = object_name_prefix + str(i)
    start = time.time()
    with open(local_file, 'rb') as f:
        conn.put_object(bucket_name, obj_name, f)
    end = time.time()
    return (end - start) * 1000


def bench_get(i):
    obj_name = object_name_prefix + str(i)
    start = time.time()
    _, obj_contents = conn.get_object(bucket_name, obj_name)
    with open(obj_name, 'wb') as f:
        f.write(obj_contents)
    end = time.time()
    return (end - start) * 1000


def bench_delete(i):
    obj_name = object_name_prefix + str(i)
    start = time.time()
    conn.delete_object(bucket_name, obj_name)
    end = time.time()
    return (end - start) * 1000

```

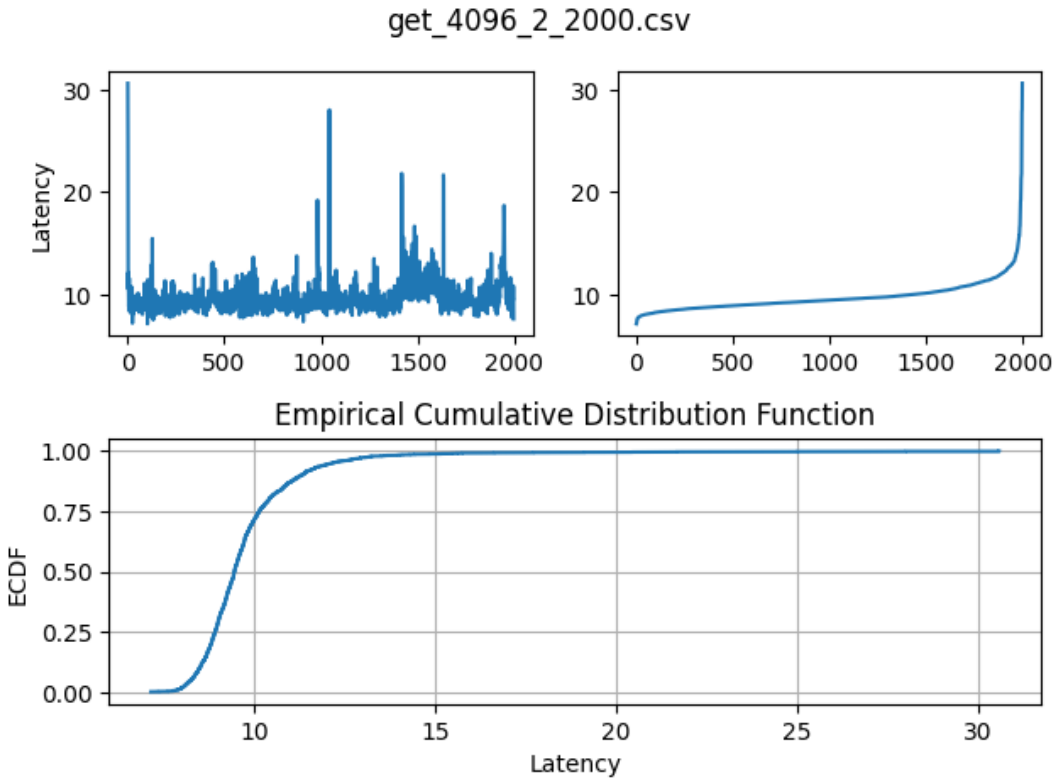
输入参数 object_size、num_clients、num_samples 分别为 4096B、10、2000 运行如下图所示。

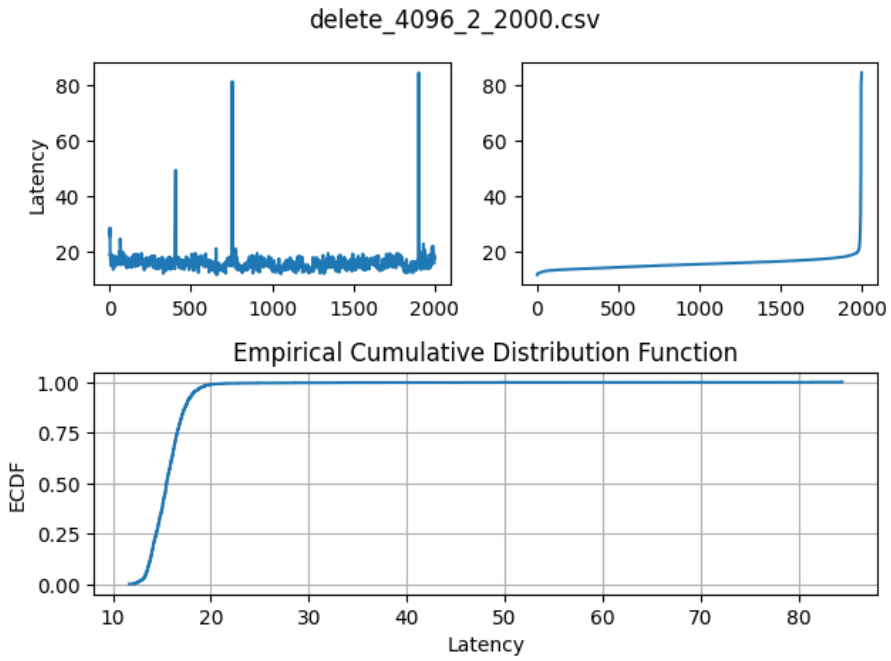
```
❯ python3 latency_collect.py 4096 10 2000
endpoint:  http://localhost:12345/
bucket_name:  test-bucket
object_name_prefix:  test-obj
object_size:  4096 B
num_clients:  10
num_samples:  2000
Test bucket test-bucket created.
Test file test.bin created.
Running put test...
total duration:  14.292045831680298 s
average latency:  71.24705123901367 ms
total transferred:  8000.0 KB
total throughput:  559.7519133521733 KB/s
success rate:  100.0 %
writing latency to put_4096_10_2000.csv
-----------------------------------------------------
Running get test...
total duration:  9.87982988357544 s
average latency:  49.18830871582031 ms
total transferred:  8000.0 KB
total throughput:  809.7305413425659 KB/s
success rate:  100.0 %
writing latency to get_4096_10_2000.csv
-----------------------------------------------------
Running delete test...
total duration:  12.605906009674072 s
average latency:  62.804670572280884 ms
total transferred:  8000.0 KB
total throughput:  634.6231674153853 KB/s
success rate:  100.0 %
writing latency to delete_4096_10_2000.csv
-----------------------------------------------------
Downloaded files deleted.
Test file test.bin deleted.
Test bucket test-bucket deleted.
```

5.3 对象存储性能分析

以 object_size=4096B，num_clients=2，num_samples=2000 为例，绘制图表如下所示，程序见 latency_plot.ipynb。

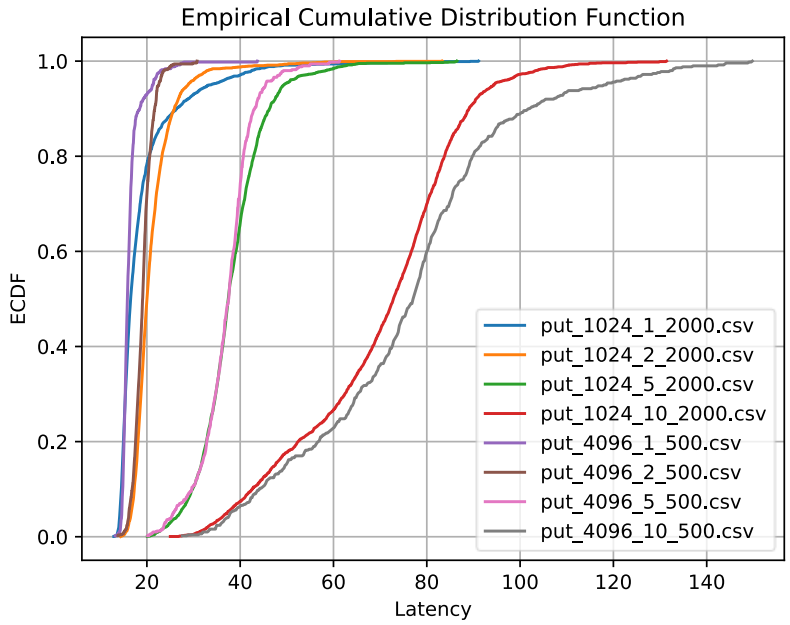






可以观察到大部分请求的响应延迟较小，而少部分请求响应延迟较大，即为尾延迟现象。另外 get、delete 的延迟明显高于 put 的延迟，可能是因为前者在向对象系统写入、删除数据的同时产生了一些副作用，而后者只需要下载数据，这也与一般 HTTP 请求的特点相同。

探究对象尺寸和并发数对性能的影响，将 num_samples 和 object_size 的固定，选取 put 操作，改变 object_size 和 num_clients 的大小，分析实验数据可知增大对象尺寸能够有效增大吞吐率，并发数增加也能减小总延迟，但是当并发数增大到一定程度时平均延迟增加，可能是线程之间发生阻塞的原因。



六、实验总结

本次实验为我打开了对象存储技术的新大门，我初步接触了解了对象存储技术的特点和相关经典架构，并对搭建简易的对象存储系统服务端和客户端进行了实践，并对对象存储系统的性能进行了观测，对对象存储系统性能的影响因素有了更深刻的认识，本次实验提升了我设计实验分析数据的能力。

实验的很多地方都值得我进一步学习研究，为解决尾延迟问题，我还阅读了相关文献，对缓解尾延迟现象的方法有了一定的了解，但受限于时间和自身的知识储备，很遗憾没有编程实现相关算法。

参考文献.

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

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