2018 级

《物联网数据存储与管理》课程

实 验 报 告

姓　名　王英嘉

学　号　U201814500

班　号　物联网 1801 班

日　期　2021.06.21

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，架设实际应用，示范主要功能。

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。

三、实验环境

实验一开始在 Ubuntu 双系统中尝试，后来在启动对象存储服务的过程中出现了比较多的问题，与原有环境产生了冲突，为了不影响原有环境和配置，因此实验最终创建了新虚拟机，并在虚拟机中完成。

实验环境说明如表 1 所示。

表 1 实验环境说明

虚拟机操作系统	Ubuntu 16.04 LTS
虚拟机配置	Intel® Core™ i7-8550U CPU @ 1.80GHz × 2
对象存储服务端	Openstack Swift(Docker 20.10.6)
对象存储客户端	python-swiftclient 3.12.0
评测工具	COSBench 0.4.2c4

对于对象存储系统服务端，实验中选择 OpenStack Swift，并将环境配置在 Docker 容器中，客户端使用 python-swiftclient 包进行对象存储测试。

其中 OpenStack Swift 是 OpenStack 开源云计算项目的子项目之一，Swift 通过在软件层面引入一致性哈希技术和数据冗余性，牺牲一定程度的数据一致性来达到高可用性和高伸缩性，支持多租户模式、容器和对象读写操作，非常适合解决互联网应用场景下非结构化数据存储的问题。

搭建好的对象存储系统后，通过 COSBench 进行性能评测，环境配置在本机。COSBench 是一种应对云对象存储系统的分布式基准测评工具，由 Driver 和 Controller 组成，Driver 主要负责工作负载生成和收集性能数据；Controller 负责调度 Drivers 协调执行工作负载，并从 Driver 中收集运行时信息和最终结果。

四、实验内容

实验内容参考 https://github.com/cs-course/obs-tutorial，主要包括如下三方面：

1、熟悉基础环境，在本地配置好对象存储的服务端 OpenStack Swift、客户端 python-swiftclient，与对应的评测工具 COSBench；

2、实践对象存储，在提供 swift-config-sample.xml 基础上，修改配置文件并提交运行，分别实践和分析不同 Object size 和 Concurrency 情况下对对象存储的性能的影响；

3、进行实际研究，基于尾延迟的问题，2013 年 Jeff 论文《The Tale at Scale》提出了两种行之有效的方案，即 HedgedRequest 和 TiedRequest，使用 go 语言编写程序进行模拟，验证两个方案的可行性。

五、实验过程

5.1 熟悉基础环境

基于如下命令安装 Docker 容器，如图 1 所示。

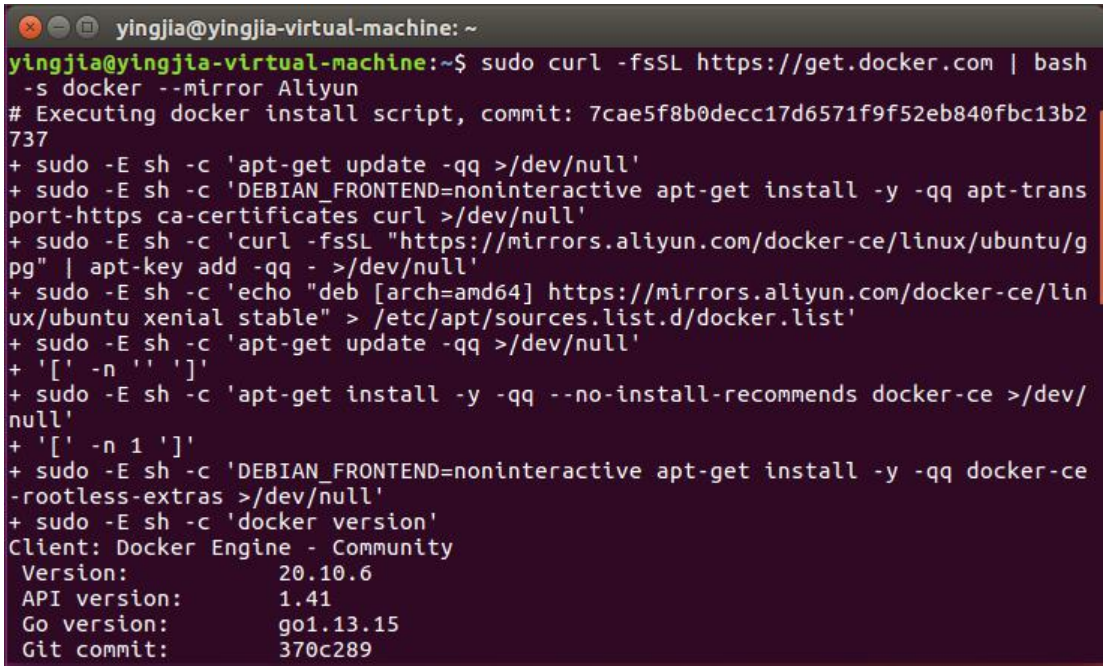

图 1 Docker 安装

安装成功后，通过 docker -v 命令，检查 docker 版本。

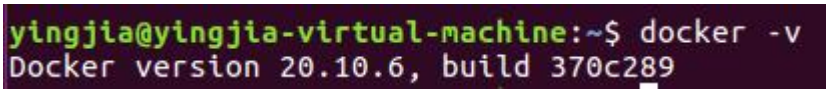

图 2 检查 Docker 版本

按照 https://github.com/cs-course/openstack-swift-docker 中的指导书依次执行命令，构建 Docker 镜像，准备数据分卷，并启动一个容器实例；之后通过 docker ps 命令验证容器成功启动，相关截图如图 3-图 5。

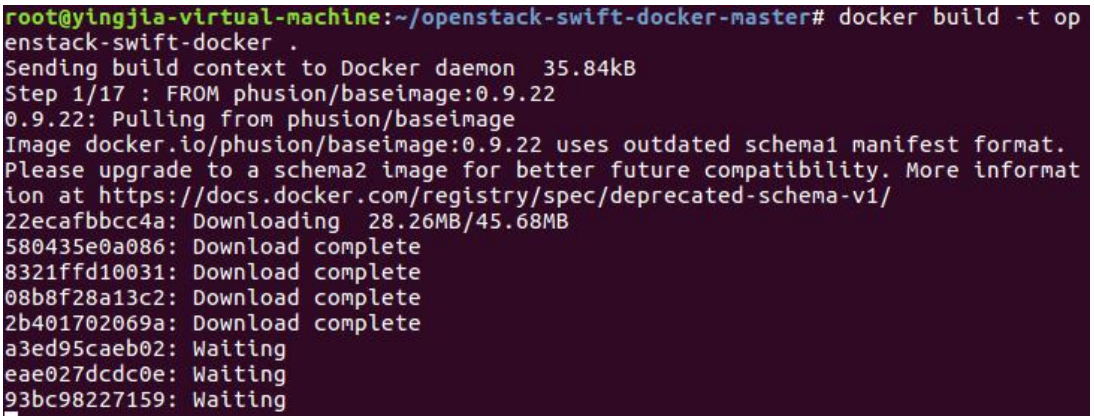

图 3 构建 Docker 镜像

```
root@yingjia-virtual-machine:~/openstack-swift-docker-master# docker run -v /srv
 --name SWIFT_DATA busybox
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
92f8b3f0730f: Pull complete
Digest: sha256:b5fc1d7b2e4ea86a06b0cf88de915a2c43a99a00b6b3c0af731e5f4c07ae8eff
Status: Downloaded newer image for busybox:latest
```

图 4 准备数据分卷

```
root@yingjia-virtual-machine:~/openstack-swift-docker-master# docker run -d --name openstack-swift -p 12345:8080 --volumes-from
 SWIFT_DATA -t openstack-swift-docker
367dfa160bece63d5b0e4ef5713f1b61d004d88cd2e62646092c0fbaca398692
root@yingjia-virtual-machine:~/openstack-swift-docker-master# docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED         STATUS         PORTS
               NAMES
367dfa160bec   openstack-swift-docker   "/bin/sh -c /usr/loc…"   8 seconds ago   Up 7 seconds   0.0.0.0:12345->8080/tcp, :::123
45->8080/tcp   openstack-swift
```

图 5 启动容器实例

如图 6 所示，启动容器过程中可能会提示纠删码相关库动态链接错误，经老师指点此处对实验没有影响，如需解决 warnings，需要安装下列库。

https://github.com/openstack/liberasurecode、https://github.com/ceph/gf-complete

https://github.com/tsuraan/Jerasure、https://github.com/intel/isa-l

```
liberasurecode[42]: liberasurecode_backend_open: dynamic linking error libisal.so.2: cannot open shared object file: No such fi
le or directory

liberasurecode[44]: liberasurecode_backend_open: dynamic linking error libshss.so.1: cannot open shared object file: No such fi
le or directory
```

图 6 动态链接错误

python-swiftclient 可以通过 pip 直接安装，图 7-图 9 首先创建了一个用户，并检查状态；然后分别将实验指导 PPT 上传和下载，如图 10 所示，下载的文件完好如初，验证了流程的正确性。

```
root@yingjia-virtual-machine:~/openstack-swift-docker-master# swift -A http://127.0.0.1:12345/auth/v1.0 -U test:tester -K testing stat
       Account: AUTH_test
    Containers: 0
       Objects: 0
         Bytes: 0
X-Put-Timestamp: 1622531511.05044
    X-Timestamp: 1622531511.05044
   Content-Type: text/plain; charset=utf-8
     X-Trans-Id: tx3bca18d8c2be42b995459-0060b5ddb7
```

图 7 客户端创建用户

```
root@yingjia-virtual-machine:~/files# swift -A http://127.0.0.1:12345/auth/v1.0
-U test:tester -K testing upload --object-name test.pptx SWIFT_DATA ./iot-storag
e-experiment.pptx
test.pptx
```

图 8 上传文件

```
root@yingjia-virtual-machine:~/files# swift -A http://127.0.0.1:12345/auth/v1.0
-U test:tester -K testing download SWIFT_DATA test.pptx
test.pptx [auth 0.009s, headers 0.058s, total 0.122s, 68.854 MB/s]
```

图 9 下载文件

图 10 验证文件

在 Github 中安装 cosbench 0.4.2.c4 版本，运行 start-all.sh 启动，如图 11 所示，启动了一个 Driver 运行在 18088 端口，启动了一个 Controller 运行在 19088 端口。

```
yingjia@yingjia-virtual-machine:~/0.4.2.c4$ ./start-all.sh
Launching osgi framwork ...
Successfully launched osgi framework!
Booting cosbench driver ...
Starting    cosbench-log_0.4.2    [OK]
.
Starting    cosbench-tomcat_0.4.2    [OK]
Starting    cosbench-config_0.4.2    [OK]
Starting    cosbench-http_0.4.2    [OK]
Starting    cosbench-cdmi-util_0.4.2    [OK]
Starting    cosbench-core_0.4.2    [OK]
Starting    cosbench-core-web_0.4.2    [OK]
Starting    cosbench-api_0.4.2    [OK]
Starting    cosbench-mock_0.4.2    [OK]
Starting    cosbench-ampli_0.4.2    [OK]
Starting    cosbench-swift_0.4.2    [OK]
Starting    cosbench-keystone_0.4.2    [OK]
Starting    cosbench-httpauth_0.4.2    [OK]
Starting    cosbench-s3_0.4.2    [OK]
Starting    cosbench-librados_0.4.2    [OK]
Starting    cosbench-scality_0.4.2    [OK]
Starting    cosbench-cdmi-swift_0.4.2    [OK]
Starting    cosbench-cdmi-base_0.4.2    [OK]
Starting    cosbench-driver_0.4.2    [OK]
Starting    cosbench-driver-web_0.4.2    [OK]
Successfully started cosbench driver!
Listening on port 0.0.0.0/0.0.0.0:18089 ...
Persistence bundle starting...
Persistence bundle started.
---------------------------------------------
!!! Service will listen on web port: 18088 !!!
---------------------------------------------

=====================================================

Launching osgi framwork ...
Successfully launched osgi framework!
Booting cosbench controller ...
.
Starting    cosbench-log_0.4.2    [OK]
Starting    cosbench-tomcat_0.4.2    [OK]
Starting    cosbench-config_0.4.2    [OK]
Starting    cosbench-core_0.4.2    [OK]
Starting    cosbench-core-web_0.4.2    [OK]
Starting    cosbench-controller_0.4.2    [OK]
Starting    cosbench-controller-web_0.4.2    [OK]
Successfully started cosbench controller!
Listening on port 0.0.0.0/0.0.0.0:19089 ...
Persistence bundle starting...
Persistence bundle started.
---------------------------------------------
!!! Service will listen on web port: 19088 !!!
```

图 11 COSBench 启动

尝试提交 conf 文件夹中 workload-config.xml，如图 12 所示，COSBench 启动了一个 workload，并赋予 ID 为 w1，同时交付 driver1 成功开始执行。

```
yingjia@yingjia-virtual-machine:~/0.4.2.c4$ sh cli.sh submit conf/workload-config.xml
Accepted with ID: w1
yingjia@yingjia-virtual-machine:~/0.4.2.c4$ sh cli.sh info
Drivers:
driver1 http://127.0.0.1:18088/driver
Total: 1 drivers

Active Workloads:
w1      Tue Jun 01 15:53:00 CST 2021    PROCESSING      s2-prepare
Total: 1 active workloads
```

图 12 benchmark 测试

5.2 实践对象存储

下面进行 swift 对象存储测试，首先在 swift-config-sample.xml 中修改 auth 信息如下，与用 client 创建用户的信息一致。

```
<!-- MODIFY ME  -->
<auth type="swauth" config="username=test:tester;password=testing;auth_url=http://127.0.0.1:12345/auth/v1.0" />
```

图 13 修改 auth 信息

提交 swift benchmark 并在浏览器中观测，运行解释后 report 信息如图 14 所示，显示了各个阶段读/写的统计量信息，其中：

Op-Type 表示操作类型，Op-Count 表示操作总数，Byte-Count 表示操作产生的 Byte 总数；Avg-ResTime 表示操作产生的平均时间，Avg-ProcTime 表示操作的平均时间，Throughput 表示吞吐量，Bandwidth 表示带宽，Succ-Ratio 表示操作的成功率。

其中比较重要的是 Avg-ProcTime 和 Throughput，前者反映了平均每次请求的时延，后者反映了操作的并发程度。

Final Result

General Report

Op-Type	Op-Count	Byte-Count	Avg-ResTime	Avg-ProcTime	Throughput	Bandwidth	Succ-Ratio
op1: init -write	0 ops	0 B	N/A	N/A	0 op/s	0 B/S	N/A
op1: prepare -write	1.6 kops	102.4 MB	23.48 ms	22.65 ms	42.43 op/s	2.72 MB/S	100%
op1: read	31.85 kops	2.04 GB	52.92 ms	52.89 ms	106.17 op/s	6.8 MB/S	100%
op2: write	7.85 kops	502.66 MB	90.8 ms	90.41 ms	26.18 op/s	1.68 MB/S	100%
op1: cleanup -delete	3.2 kops	0 B	17.88 ms	17.88 ms	55.86 op/s	0 B/S	100%
op1: dispose -delete	0 ops	0 B	N/A	N/A	0 op/s	0 B/S	N/A

show peformance details

图 14 swift benchmark

首先探究 object size 大小对各项指标的影响，修改 size 大小分别为 4、16、64、256，得到 main stage 阶段 read/write 性能指标结果如表 2 所示。

表 2 不同 object size 下各项指标统计

Op-Type	Op-Count	Byte-Count	Avg-ResTime	Avg-ProcTime	Throughput	Bandwidth	Succ-Ratio
Read-4	40.97	163.88M	40.35	40.34	136.58	546.31	100%
Write-4	10.38	41.54M	71.78	71.72	34.62	138.46	100%
Read-16	40.36	661.43M	42.45	42.45	134.53	538.13	100%
Write-16	10.06	160.23M	68.14	68.09	33.53	134.1	100%
Read-64	38.17	2.44G	42.93	42.9	127.24	8.14	100%
Write-64	9.76	0.62G	77.88	77.61	32.54	2.08	100%
Read-256	31.85	8.15G	53.28	46.58	106.17	27.18	100%
Write-256	8.01	2.05G	87.58	86.84	26.72	6.84	100%

其中各列单位分别为 kops，B，ms，ms，op/s，MB/s，无。

可以看到，随着 object size 大小的增加，读写过程中 Byte-Count 的值发生显著增加，且近似为正比例关系，因为 Byte-Count 反映的是数据迁移的总量（total data transferred）。

Avg-ResTime 和 Avg-ProcTime 随着 object size 大小增加有小幅度增加，因为 object 越大，处理产生时间和处理时间相对会越长；吞吐量表征的是 1s 内完成操作数量，因此有小幅度减少也合理；带宽由于不稳定没有明显特征。

再探究 worker 数量（Concurrency）对各项指标的影响，修改 workers 数量分别为 1、2、4、8，即分别使用 1 核、2 核、4 核、8 核四种情况进行测试，得到 main stage 阶段 read/write 性能指标结果如表 3 所示。

表 3 不同 concurrency 下各项指标统计

Op-Type	Op-Count	Byte-Count	Avg-ResTime	Avg-ProcTime	Throughput	Bandwidth	Succ-Ratio
Read-1	21.81	348.9M	9.54	9.53	72.69	1.16M	100%
Write-1	5.56	89.04M	16.39	16.27	18.55	296.8K	100%
Read-2	35.33	565.28M	11.6	11.59	117.77	1.88M	100%
Write-2	8.78	140.45M	21.54	21.45	29.26	468.19K	100%
Read-4	30.31	484.94M	27.83	27.81	101.05	1.62M	100%
Write-4	7.63	122.11M	46.56	46.38	25.44	407.1K	100%
Read-8	36.26	580.11M	45.58	45.56	120.87	1.93M	100%
Write-8	8.84	141.42M	84.41	84.41	29.47	471.45K	100%

其中各列单位分别为 kops，B，ms，ms，op/s，MB/s，无。

首先注意到一个问题，当 workers 数量为 2、4 和 8 时的数据无较大差异，但与 workers 数量为 1 时的数据差异较大，这可能是因为虚拟机只分配了两个核心，所以当 workers 数量大于 2 时意义不大，反而由于多路线程阻塞影响了性能。

下面逐个分析指标的变化：

（1）Op-Count：增加，猜测是因为多个 workers 并发执行时增加了一部分调度的操作；

（2）Byte-Count：增加，猜测同上；

（3）Avg-ResTime：近似正比例增加，这里是比较困惑的一点，注意到 workers=1 和 workers=2 时差别微小，而 workers=2、4、8 时相对差别较大，经老师指点，此处应是因为由于 I/O 显著慢于计算处理，因此越多线程就导致阻塞越严重，近乎是正比例关系。

（4）Avg-ProcTime：近似正比例增加，猜测同上；

（5）Throughput：增加，并行操作使得每秒可以完成更多 operations；

（6）BandWidth：无明显差异，说明这段时间带宽较稳定；

（7）Succ-Ratio：全部为 100%，说明操作执行都成功。

5.3 尾延迟探究

由 Jeff 论文中提到，在分布式系统中，经常出现因为极少数事件延迟很长导致大概率整体响应时间变长的 Tail Latency 的问题，因此论文中提出了 Hedged Request 和 Tied Request 两种优化方案。

5.3.1 Hedged Request

Hedged Request 的思想说明如下：

1、client 首先发送一个请求给服务端，并等待服务端返回响应

2、如果客户端在 95%的请求响应时间内没有收到服务端的响应，则马上发送同样的请求到另一台（或多台）服务器

3、客户端等待第一个响应到达之后，终止其他请求的处理

Google 的测试数据表明，利用这种方法，可以仅用 2%的额外请求，将系统 99%的请求的响应时间从 1800ms 降低到 74ms。

根据上述思想，用 go 语言编写并发程序进行模拟，验证 Hedged Request 的性能优势，参数配置如图 15 所示，核心代码 server 函数如图 16 所示。

```
const(
    lowDelay = 5 // 低延迟请求时间 ms
    highDelay = 1000 // 高延迟请求时间 ms
    highDelayRate = 0.1 // 请求遇到高延迟的概率
    maxTimeout = 10 // 从发送请求后允许的最大等待时间 ms
    candidates = 10 // 超过最大等待时间后下一次发出请求的数量
    testCount = 1000 // 测试次数
)
```

图 15 HedgedRequestSim 参数配置

server 函数内部允许递归调用，以模拟如果超过 maxTimeout 后需要再次发送 candidates 个请求的过程，请求的响应通过 golang 中的计时器模拟，一旦有某个请求响应，则通过通道一直向上层传递直到被 main 函数中主通道接收。

server.go main.go config.go

```
func server(n int, c chan int) {
    var latency int

    for i := 0; i < n; i++{
        r := rand.Intn( n: 100)
        if r < 100 * highDelayRate{
            latency = highDelay
        } else {
            latency = lowDelay
        }
        select{
        case <- time.After(time.Millisecond * time.Duration(latency)):
            c <- latency
        case <- time.After(time.Millisecond * maxTimeout):
            Chan := make(chan int, candidates)

            go server(candidates, Chan)
            select{
                case nextTime := <-Chan:
                    c <- maxTimeout + nextTime
                case <-time.After(time.Millisecond * time.Duration(latency-maxTimeout)):
                    c <- latency
            }
        }
    }
    close(c)
}
```

图 16 HedgedRequestSim 核心代码

修改参数不断测试，结果如表 4。

表 4 HedgedRequest 参数探究

Id	HighLatencyRate	maxTimeout	candidates	HedgedRequest	Avg-Latency
1	0.01	10		false	15.945
2	0.01	10	3	true	5.12
3	0.05	10		false	63.705
4	0.05	10	3	true	5.59
5	0.1	10		false	113.455
6	0.1	10	3	true	6.18
7	0.01	10	5	true	5.12
8	0.01	10	10	true	5.11
9	0.2	10	1	true	8.81
10	0.2	10	3	true	7.91
11	0.2	10	5	true	7.39

由 1-6 分析，当使用 HedgedRequest 时，即使 HighLatencyRate 达到了 10%，平均延迟仍然与低延迟相差无几，而不使用 HedgedRequest 时的平均延迟最多已经达到了 20 多倍。

由 2、7、8 分析，当 HighLatencyRate 为 1%时，修改 candidates 数量，即一定时间内没有收到请求时再次转发的服务器数量，结果几乎不变，这说明 HighLatencyRate 很低时增加转发次数意义不大，反而在实际应用中会增大许多不必要的开销。

由 9-11 分析，当 HighLatencyRate 为 20%时，修改 candidates 数量，此时平均延迟从 8.81ms 降低到了 7.91ms 和 7.39ms，这启发我们：当前网络状况不佳时，可以动态调整转发服务器的个数，以发挥极致整体性能。

5.3.2 Tied Request

Tied Request 的思想说明如下：

每个服务器有一个任务队列 TaskQueue 维护当前并发执行的任务和排队等待执行的任务，系统同时发送相同任务给多个服务器运行，当某个任务被其中一个服务器运行完成时，将通知其他服务器不用再处理该请求；为了防止由于任务队列为空而导致所有服务器同时开始处理任务，系统发给多个服务器要引入一个延

迟，该延迟时间足够第一个服务器完成并通知其他服务器。

值得注意的是，为什么系统在发送任务前不先进行探测哪个服务器任务队列最短？原因有三：

（1）探测和响应间有一定时间间隔，这段间隔后服务器负载很可能早已经发生了变化。

（2）如果多个系统同时发现服务器负载低，会造成“瞬间过热”。

（3）队列长度小并不能保证处理时间一定短。

由于 Tied Request 比较复杂，以下做了一定的简化：

（1）没有实现不同服务器间的通信，而是通过一个全局的并发安全的 map 记录某个任务的最小响应时间。每个服务器最多允许 concurrencyLimit 个任务同时执行（默认为 10），当有高延迟任务在其他服务器中已经完成，在本服务器中已经开始执行的任务不会被打断直到执行结束，由于 concurrencyLimit 一般比较大，且 highLatencyRate 一般比较小，假设一个服务器中有一两个高延迟任务阻塞导致最多有效并发任务数略小于 concurrencyLimit 的影响，是可以忽略的。

（2）没有实现系统将相同任务发给多个服务器间的延迟，结合（1）考虑，实现意义不大。

首先实现一个任务队列 TaskQueue 来模拟服务器，队列定义如图 17 所示，可以支持最多 concurrencyLimit 个任务并发执行。

```
type queue struct {
    Handler          func(interface{})
    ConcurrencyLimit int
    push             chan interface{}
    pop              chan struct{}
    stop             chan struct{}
    stopped          bool
    buffer           []interface{}
    count            int
    wg               sync.WaitGroup
}
```

图 17 TaskQueue 定义

参数配置如图 18 所示，要考虑的参数与 HedgedRequest 相差不多。

```
const(
    lowLatency = 5 // 低延迟
    highLatency = 1000 // 高延迟
    highLatencyRate = 0.01 // 高延迟率
    queueCount = 3 // 任务队列个数
    concurrencyLimit = 10 // 每个任务队列可以并发执行的任务数量
    testCount = 1000 // 测试次数
    ifTiedRequest = true // 是否使用TiedRequest
)
```

图 18 TaskQueue 参数配置

使用一个并发安全的 map 记录最小的响应时间，代码实现如图 19。

```
var concurrentMap map[int]int
var lock sync.Mutex

func record(i interface{}, latency int){
    lock.Lock()
    defer lock.Unlock()

    concurrentMap[i.(int)] = latency
}

func checkTask(i interface{}) bool{
    lock.Lock()
    defer lock.Unlock()

    _, ok := concurrentMap[i.(int)]
    return ok
}
```

图 19 并发安全 map 实现

TiedRequest 模式下，系统将任务分发给全部任务队列，每个任务队列当下一个需要执行某任务时，会先到 concurrentMap 中确认是否已经有了最小响应时间，如果没有，才会开始执行，执行完毕后再次确认是否有了最小响应时间，如果仍然没有，就会将结果写入；因此如果在上述“第一次确认”和“写入”之间有其他任务队列确认并开始执行的话，会造成任务的重复执行，造成的干扰在简化假设中有讨论，可以忽略不计。

非 TiedRequest 模式下，系统直接将任务分发给某一个服务器并交付执行，无论是低延迟还是高延迟都只能盲等。

修改参数不断测试，结果如表 5。

表 5 TiedRequest 参数探究

Id	highLatencyRate	queueCount	TiedRequest	Avg Latency
1	0.01	3	false	14.95
2	0.01	3	true	5
3	0.05	3	false	55.745
4	0.05	3	true	19.925
5	0.1	3	false	107.485
6	0.1	3	true	59.725
7	0.2	3	false	215.94
8	0.2	3	true	66.69
9	0.01	5	false	15.945
10	0.01	5	true	5
11	0.01	10	false	15.945
12	0.01	10	true	5

由 1-8 分析，随着高延迟率的增加，TiedRequest 模式和非 TiedRequest 模式相比可以有效减少平均延迟，主要体现在通过转发额外请求换取可能更小的响应时间；

由 1-2 和 9-12 分析，在高延迟率不变的情况下，在分发服务器数量增加时，两种模式提升都不明显，分别分析：

（1）TiedRequest 模式平均延迟以及与低延迟相当，自然不可能再提升，如果高延迟率比较高的话，增大 queueCount 的效果比较明显；

（2）非 TiedRequest 模式平均延迟差异不大，在目前的代码实现情况下，认为增大 queueCount 只是增大了可选服务器的范围，在高延迟率一定的情况下，在哪个服务器上执行没有差别。

六、实验总结

在实验一和实验二中，我了解了什么是对象存储，从头搭建了一个完整的对象存储系统，并学会了用评测工具评测一个对象存储系统的性能，同时也更加熟练了 docker 的使用，对评测工具中一些常见的性能指标有了更深刻的理解；在实

验三中，我理解论文并动手模拟两种方案的可行性，虽然实现的代码进行了一定的抽象和简化，但仍然提高了我的文献阅读能力和代码能力。

实验中由于时间原因，对 docker 容器构建和 COSBench 评测仍然停留在应用阶段，没有理解地特别深入，之后有时间会继续研究。

最后感谢施展老师对我的答疑解惑。

参考文献

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

[2] 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.

[3] DEAN, J., AND BARROSO, L.A. The Tail at Scale. Communications of the ACM 56, 2(Feb.2013), 74-80.