电子科技大学计算机科学与工程学院

实 验 报 告

课程名称：分布式系统

实验名称：SDCS: Simple Distributed Cache System

学　　号：xxxxxxxxxx

姓　　名：Koorye

电 子 科 技 大 学

实 验 报 告

一、 课程名称

分布式系统。

二、 实验项目名称

SDCS: Simple Distributed Cache System.

三、 实验原理

RPC 是一种远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。一个完整的 RPC 架构里面包含了四个核心的组件，分别是客户端、客户端存根、服务端、服务端存根。

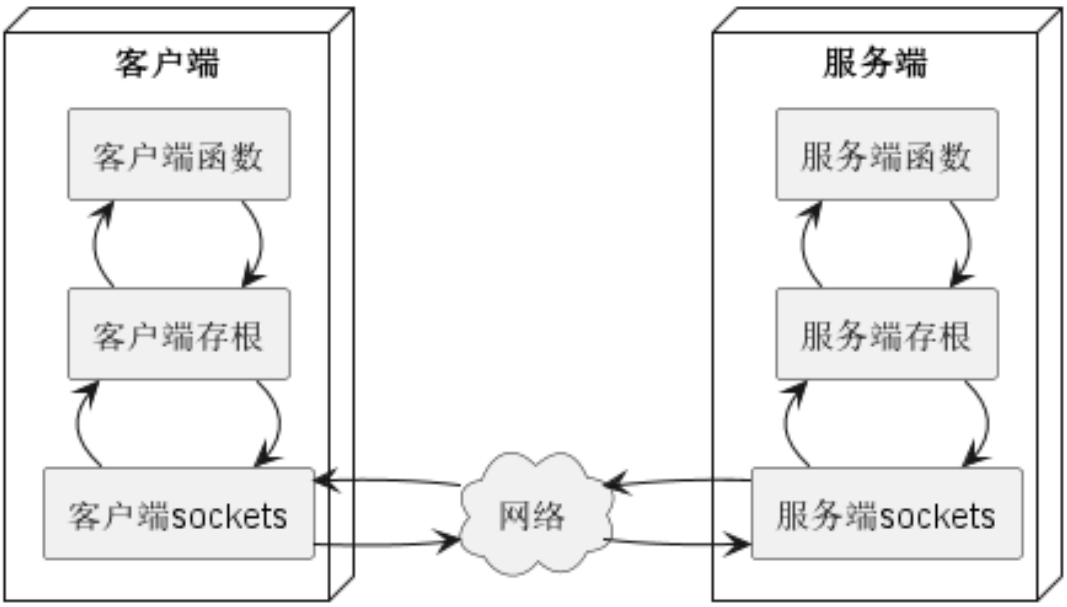

图 3.1: RPC 原理

如图**??**所示，当客户端请求 RPC 服务时，会通过存根打包请求参数，然后通过网络发送给服务端；服务端存根接收请求，将消息解包后调用本地方法，并将结果返回到客户端。为了使得请求可以在网络中传递，往往需要对数据进行序列化操作。

四、 实验目的

完成一个简易分布式缓存系统，并实现如下功能：

- 键值对的保存和读取。用户可将数据组织成键值对形式保存到数据库中，并通过指定的键读取数据。键的类型是字符串，而值可以是任意类型。
- 节点通信。各个节点之间通过 RPC 通信完成信息交换。
- 基本的负载均衡。数据将会根据键的内容被均匀分布给各个节点，读取时也会根据键的内容到指定的节点中获取值。
- 快速部署。本系统通过 docker 和 docker-compose 进行集成，用户可通过一行命令启动或停止分布式服务。如果要增删节点，用户只需编辑 docker-compose 中的内容。

五、 实验内容

完成一个简易分布式缓存系统，其具体要求如下。

1） Cache 数据以 Key-value 形式存储在缓存系统节点内存中（不需要持久化）。

2） Cache 数据以既定策略（round-robin 或 hash 均可，不做限定）分布在不同节点（不考虑副本存储）。

3） 服务至少启动 3 个节点，不考虑节点动态变化：

a) 所有节点均提供 HTTP 访问入口。

b) 客户端读写访问可从任意节点接入，每个请求只支持一个 key 存取。

c) 若数据所在存储服务器与接入服务器不同，接入服务器通过内部 RPC 从目标存储服务器获取数据，再返回至客户端。

六、 实验步骤

本章节将描述实验步骤，包括概要设计和详细设计。

（一）概要设计

1. 总体架构

本章节将展示该系统的总体架构。

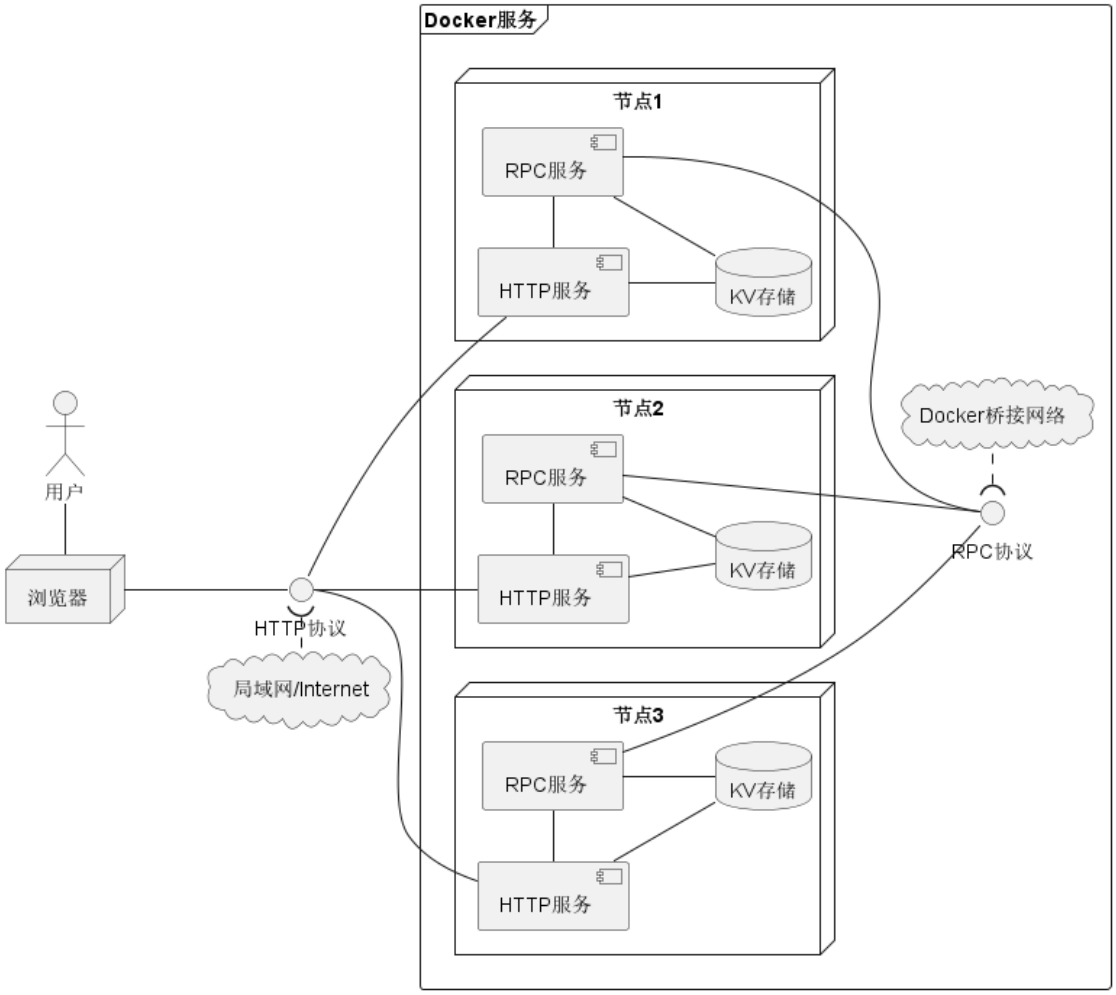

图 6.1: 总体架构

如图**??**所示，该系统的总体架构是 B/S 架构，用户可通过浏览器，基于 HTTP 协议，通过局域网或 Internet 与服务端进行交互，服务端对用户来说完全透明。

服务端内部则是分布式架构，不同节点拥有独立业务逻辑，能够完成完整的数据存储和读取业务。节点通过 HTTP 服务接收用户请求，之后与本地的 KV 存储进行交互来完成数据的存取。

当节点数量不只有 1 个时，节点之间就可以通过 RPC 服务进行通信。当请求

通过哈希函数分配给本机时，节点会自行处理逻辑，否则就会通过 RPC 发送给目标节点进行处理，并接收其返回值再返回到用户。节点之间通过 Docker 桥接网络进行通信。

2. 模块设计

本章节将展示该系统的模块设计。

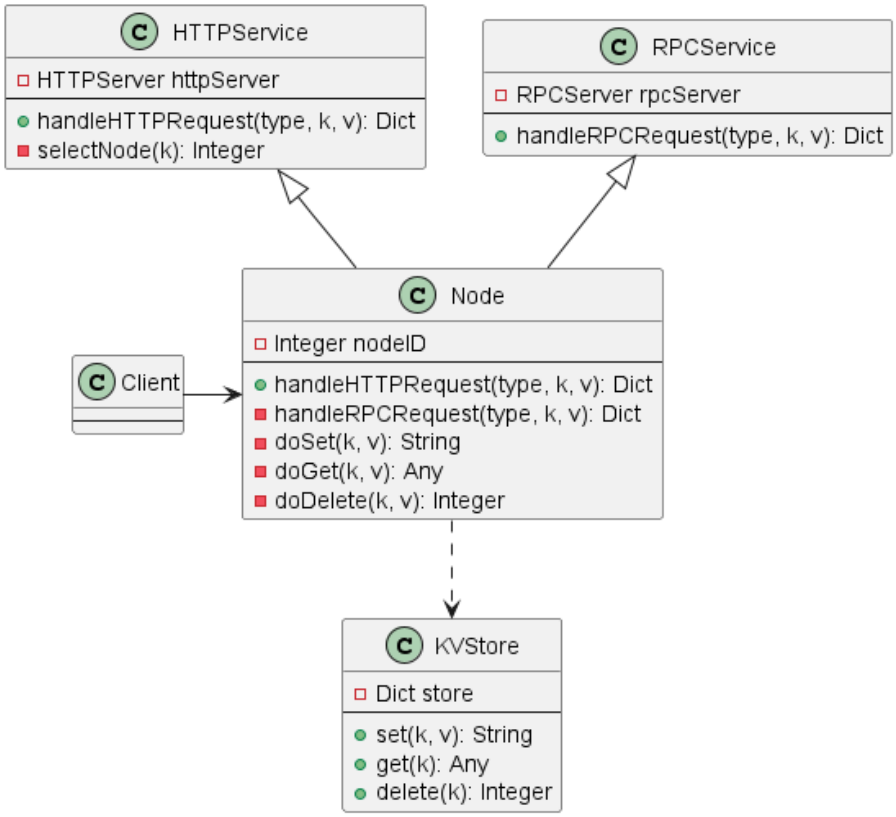

图 6.2: 模块设计

如图**??**所示，本系统可分为 4 个模块：

- HTTP 服务 (HTTPService)。该模块负责接收的 HTTP 请求，并选择处理请求节点。
- RPC 服务 (RPCService)。该模块负责接收并处理 RPC 请求。
- KV 存储 (KVService)。该模块负责存储和读取键值对。
- 节点 (Node)。通过 Python 多继承的特性，该模块负责整合所有模块并合理选择处理逻辑。如果请求节点属于本节点，该模块会调用本地的设置、获取、删除等处理函数；否则，将通过 RPC 调用目标节点的处理函数。

（二）详细设计

本章节将描述本系统的详细设计，包括算法流程和接口描述。

1. 算法流程

接收 HTTP 请求。

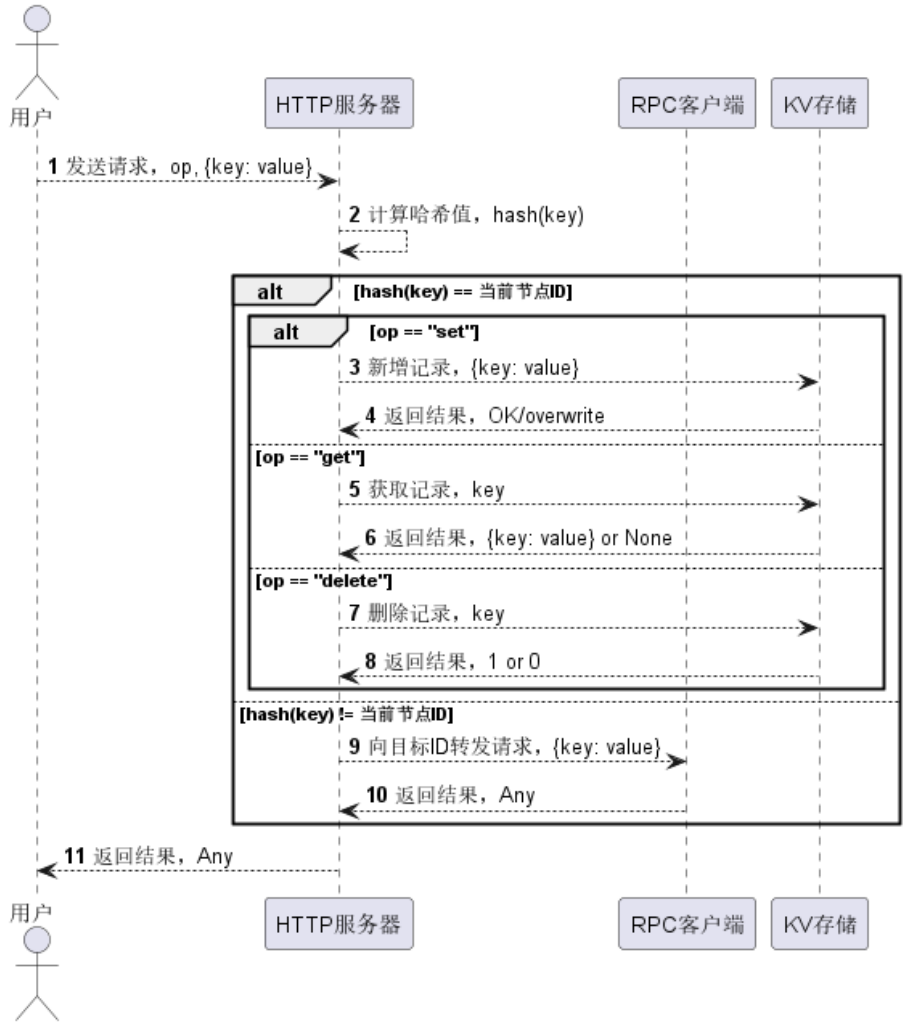

图 6.3: 接收 HTTP 请求流程

图**??**描述了任一节点接收 HTTP 请求的详细流程。当用户向任意服务器节点发送 HTTP 请求后，该节点通过对键进行哈希函数获得唯一 ID。如果 ID 与本地 ID 匹配，则与本地 KV 存储进行交互，通过操作字段 op 判断操作类型，之后选择对应的方法。如果 ID 与本地 ID 不匹配，则通过 RPC 客户端将请求转发给目标远程服务器。

接收 RPC 请求。

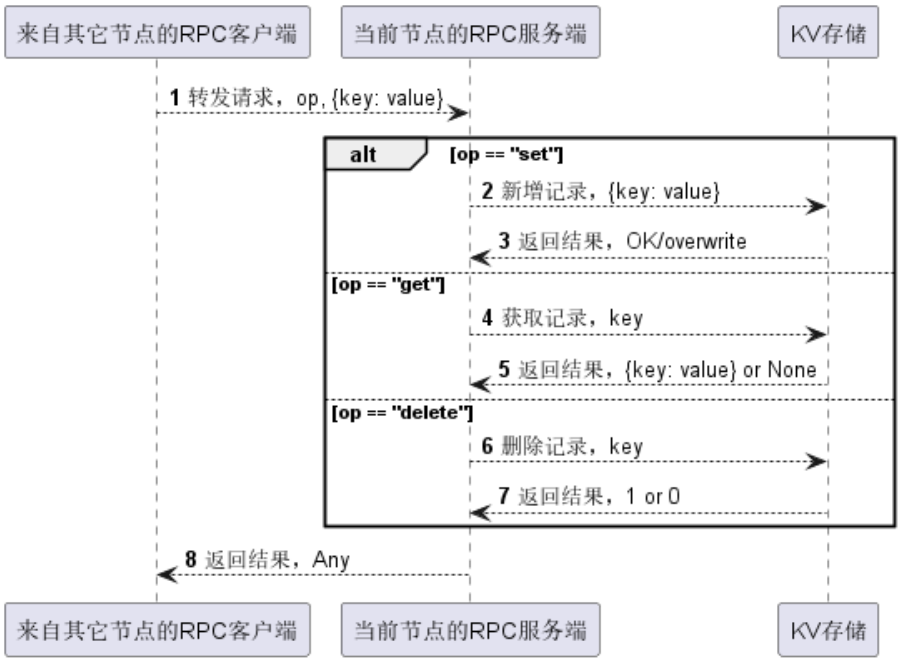

图 6.4: 接收 RPC 请求流程

图**??**描述了任一节点接收 HTTP 请求的详细流程。当 RPC 服务端收到来自任一节点的 RPC 客户端发送的 RPC 请求后，服务端所在节点就会通过操作字段 op 判断操作类型，并于本地 KV 存储交互，之后把请求返回给 RPC 客户端。

2. 接口描述

本系统的接口描述如下。

表 6.1: 接口描述

名称	URL	方法	请求参数	说明
新增	/	POST	JSON	上传格式为 {”key”: ”value”} 的 JSON
获取	/{key}	GET	键	当键存在时返回格式为 {”key”: ”value”} 的 JSON，否则返回错误码 404
删除	/{key}	DELETE	键	当键存在时返回 1 表示成功，否则返回 0 表示失败

如表**??**所示，本系统的接口描述遵循 Restful 规范，通过 POST/GET/DELETE 等请求表示操作类型。

（三）实现与部署

本章节将介绍本系统是如何实现与部署的，包括开发工具和开发流程。

1. 开发工具

开发工具：

- Visual Studio Code 1.82.2
- Fluent Terminal 0.77.0
- Docker 24.0.6
- Docker Compose v2.21.0-desktop.1

开发环境：

- Python 3.8.16
- Miniconda 4.12.0

开发框架：

- Fastapi 0.103.1。Fastapi 是一个简单高性能的 HTTP 框架，可以快速响应 HTTP 请求。
- Rpyc 5.3.1。Rpyc 是一个易用的 RPC 框架。
- Uvicorn 0.23.2。Uvicorn 是一个轻量级的服务器框架，可用于运行服务器程序。

2. 开发流程

本章节将简述开发的总体流程，避免任何不必要的细节赘述，仅展示最核心语句和代码。

建立 HTTP 服务。基于 Fastapi 和 Uvicorn，一个 HTTP 服务可以相当容易被建立。

```
import uvicorn
from fastapi import Fastapi

app = Fastapi()

@app.post('/')
def handle_set(data: Dict):
    for k, v in data.items():
        return service.handle_http_req('set', k, v)

if __name__ == '__main__':
    uvicorn.run(app, port=<HTTP_PORT>)
```

上面的代码清晰展示了 HTTP 服务是如何被建立的，只需导入 fastapi 和 uvicorn 包，创建 app，设定路由，最后通过 uvicorn 启动 app 即可。

建立 RPC 服务。基于 Rpyc，RPC 服务的建立相当容易。

```
from rpyc import Service
from rpyc.utils.server import ThreadedServer

class RPCService(Service):
    def exposed_handle_rpc_req(self, op, k, v=None):
        func = getattr(self, 'do_' + operation)
        return func(k, v) if v is not None else func(k)

if __name__ == '__main__':
    service = RPCService()
    rpc_server = ThreadedServer(service, <RPC_PORT>)
    t = threading.Thread(rpc_server.start, True)
    t.start()
```

上面的代码清晰展示了 RPC 服务的建立流程。Rpyc 框架可以直接暴露继承自 Service 类下开头为 exposed_ 的方法，这样来自其他节点的客户端可以直接通过方法名使用该方法。该方法会通过 op 字段获取本地的 set/get/delete 等方法句柄，之后调用该方法获得结果。需要注意的是，RPC 服务在一个额外线程上启用，这样才能与 HTTP 服务器同时运行。

3. 节点的选择和转发

主程序中包含基于哈希的节点选择和转发策略。

```
class Node(object):
    def handle_http_request(self, operation, k, v=None):
        target_id = self._select_rpc(k)
        if self._is_local(target_id):
            return self.handle_local_req(operation, k, v)
        else:
            target_host = self.hosts[target_id]
            conn = rpyc.connect(target_host, <RPC_PORT>)
            return conn.root.handle_rpc_req(operation, k, v)

    def _select_rpc(self, k):
        return hash(k) % len(self.hosts)

    def _is_local(self, id_):
        return id_ == self.id
```

上面的代码展示了节点的选择和转发，其逻辑非常简单。首先根据 hash() 函数获取唯一 ID 号，当 ID 与本地 ID 匹配时，就调用本地的处理函数；否则，向 ID 对应的目标节点转发 RPC 请求。

4. 部署流程

本章节将阐述本系统是如何部署的，仅展示最核心的步骤和代码。

打包 Docker 镜像。基于 Docker，可以避免繁杂且容易出错和冲突的环境配置工作，直接以容器的形式运行。

```
FROM ubuntu:20.04
MAINTAINER Koorye <a1311586225@gmail.com>
COPY requirements.txt /tmp
WORKDIR /tmp
RUN apt clean && apt update
RUN apt install -y python3 python3-dev python3-pip
RUN pip3 install -r requirements.txt
COPY simplekv simplekv
WORKDIR simplekv
ENTRYPOINT ["python3", "app.py"]
CMD ["--host", "(host)", "--all-hosts", "(host1)", "(host2)", "(host3)"]
```

上面的代码展示了一个 Dockerfile 的完整编写逻辑：

- FROM。指定基础镜像，后面的打包工作将在该镜像上进行。
- MAINTAINER。指定作者信息。
- COPY。复制文件或目录到镜像中，这里复制了 python 程序所需的依赖以及项目目录。
- WORKDIR。指定工作目录，相当于 cd 命令。
- RUN。运行命令，这里运行了多个命令：通过 apt 包管理器安装 python 程序、通过 pip 包管理器安装 python 依赖。
- ENTRYPOINT。运行命令，与 RUN 大致相同，这里启动 python 程序。
- CMD。运行命令或传递必要的参数，这里传入了当前节点的 host 信息和所有节点的 host 信息。

在编写上述 Dockerfile 之后，可以调用如下命令进行打包。

```
$ docker build -t simplekv:v1 .
```

如果一切顺利，就可以通过如下命令查看打包后的镜像。

```
$ docker images
REPOSITORY TAG IMAGE ID     CREATED     SIZE
simplekv   v1  3ccc296dffc2 13 days ago 458MB
```

之后可以通过如下命令启动容器。

```
$ docker network create simplekv_rdp
$ docker run -d --name server1 -p 9527:80 -v log/path/here/0:/tmp/simplekv/
    logs --net simplekv_rdp simplekv:v1 --host server1 --all-hosts server1
    server2 server3
$ docker run -d --name server2 -p 9528:80 -v log/path/here/1:/tmp/simplekv/
    logs --net simplekv_rdp simplekv:v1 --host server2 --all-hosts server1
    server2 server3
$ docker run -d --name server3 -p 9529:80 -v log/path/here/2:/tmp/simplekv/
    logs --net simplekv_rdp simplekv:v1 --host server3 --all-hosts server1
    server2 server3
```

首先通过 docker network create 命令名为 simplekv_rdp 的网络，模式默认为桥接。在该模式下，网络内部的容器可以通过容器名称相互通信。

之后通过 docker run 命令从镜像运行容器，上述命令中传递了一些参数：

- -d。指定容器以后台模式运行，否则将占用当前窗口的会话。
- –name。指定容器名称。
- -p。指定映射端口，可以将外部端口映射到容器内部端口。
- -v。指定挂载目录，可以将宿主机中的目录或文件挂载到容器中或是将容器中的目录或文件挂载到宿主机，这里挂载了容器中的日志目录。
- –net。指定连接的网络。
- –host, –all-hosts。镜像中指定的命令行参数。

如果一切顺利，可以通过如下命令查看网络和镜像。

```
$ docker network ls
NETWORK ID     NAME             DRIVER    SCOPE
7ffa7f92a8df   simplekv_rdp     bridge    local
$ docker ps
CONTAINER ID IMAGE        COMMAND          ... PORTS         NAMES
61ec299498db simplekv:v1 "python3 ..." ... 9527->80/tcp server1
e9c3170487e0 simplekv:v1 "python3 ..." ... 9527->80/tcp server2
5e444de115f2 simplekv:v1 "python3 ..." ... 9527->80/tcp server3
```

基于 Docker Compose 的容器编排。虽然上述方式可以启动容器，但是当容器数量很多时，这样逐个启动/关闭容器的管理方式非常麻烦，尤其是容器之间相互联系时。因此，更好的方案是采用 Docker Compose 或 Kubernates 进行容器管理。这里只展示基于 Docker Compose 的方式，首先需要编写一个 YAML 文件。

```
version: '1'
services:
  server1:
    image: simplekv:v1
    container_name: server1
    volumes:
    - log/path/here/0:/tmp/simplekv/logs/
    ports:
    - "9527:80"
    networks:
    - simplekv_rdp
    command:
    - "--host"
    - "server1"
    - "--all-hosts"
    - "server1"
    - "server2"
    - "server3"
```

```
  server2:
    ...
  server3:
    ...

networks:
  simplekv_rdp:
    external: true
    driver: bridge
```

上述文件中包含了 services 服务和 networks 网络的配置。Services 中配置了 server1/server2/server3 共 3 个节点，与上述配置方式相似的，每个节点进行了镜像、容器名称、目录挂载、端口映射、网络、命令行参数的配置。Networks 中则配置了网络名称、模式，并指定为外部网络。

之后，可以通过一行命令启动所有容器。

```
$ docker-compose up -d
[+] Running 3/3
 ✔ Container server1  Started                                0.1s
 ✔ Container server2  Started                                0.1s
 ✔ Container server3  Started                                0.1s
```

基于 Docker Compose，容器可以很容易被编排和管理，集群服务的启动/停止变得非常容易。

七、 实验数据及结果分析

基于测试示例，对本系统的工作情况进行验证，首先是写入/更新操作测试。

```
$ curl -XPOST -H "Content-type: application/json" http://localhost:9527/ -d '{"myname": "电子科技大学@2023"}'
"ok"
$curl -XPOST -H "Content-type: application/json" http://localhost:9528/ -d '{"tasks": ["task 1", "task 2", "task 3"]}'
"ok"
$curl -XPOST -H "Content-type: application/json" http://localhost:9529/ -d '{"age": 123}'
"ok"
```

写入成功时，返回 ok，测试通过。

接下来是读取测试。

```
$ curl http://localhost:9528/myname
{"myname": "电子科技大学@2023"}
$ curl http://localhost:9527/tasks
{"tasks": ["task 1", "task 2", "task 3"]}
$ curl http://localhost:9527/notexistkey
{"detail":"Not Found"} # status code is 404
```

读取成功时返回 JSON，否则返回 404，测试通过。

最后是删除测试。

```
$ curl -XDELETE http://localhost:9529/myname
1
$ curl http://localhost:9527/myname
{"detail":"Not Found"} # status code is 404
$ curl -XDELETE http://localhost:9529/myname
0
```

删除成功时返回 1，值不存在时返回 0，删除后读取相应 key 返回 404，测试通过。

基于测试脚本进行完整测试，结果如下。

```
$ bash tests/test.sh 3
"ok""ok""ok""ok""ok"..."ok"
{"key-70":"value 70"}{"key-473":"value 473"}{"key-43":"value 43"}{"key-431":"
    value 431"}{"key-134":"value 134"}{"key-475":"value 475"}{"key-88":"value
    88"}...{"key-394":"value 394"}"overwrite""overwrite""overwrite"..."
    overwrite"
11111111111111111011111111111111011101100...00
{"detail":"Not Found"}{"detail":"Not Found"}{"detail":"Not Found"}{"detail":"
    Not Found"}{"detail":"Not Found"}{"key-133":"value 133"}{"key-209":"value
    209"}{"key-267":"value 267"}{"key-86":"value 86"}{"key-402":"value 402"
    }...{"detail":"Not Found"}{"detail":"Not Found"}{"detail":"Not Found"}{"
    detail":"Not Found"}
```

上述结果是基于脚本中大量模拟请求得到的结果，分别包括 set/get/delete/set/get 操作，可以看到所有操作都被顺利执行。

八、 实验结论

本实验包括一个分布式 KV 存储引擎从设计、开发、部署、测试的完整流程，最终顺利完成全部实验流程，设计并实现了一个功能简单的分布式 KV 存储引擎，通过测试，实现了实验目标。

九、 总结及心得体会

我顺利完成了分布式 KV 存储的实验，这是一个非常有挑战性的任务。在这个实验中，我不仅学习了分布式系统的设计和实现，还学习了如何测试和评估分布式系统的性能。

这个实验是一个非常有价值的学习经历，它不仅让我了解了分布式系统的工作原理，还让我学会了如何设计、实现和测试一个分布式 KV 存储系统，对我以后的工作也很有帮助。

对于这个实验我有一些建议，例如可以在测试阶段使用一些基准测试工具来验证系统的功能并评估系统的性能；还可以实现更多功能，如故障检测、副本、动态节点更新等。