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：简易分布式缓存系统

学　　号：

姓　　名：

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

简易分布式缓存系统（Simple Distributed Cache System，SDCS）

三、实验目的

完成一个简易分布式缓存系统：

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（即运行中无新节点加入，也无故障节点退出）；

i. 所有节点均提供 HTTP 访问入口；

ii. 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；

iii. 若数据所在目标存储服务器与接入服务器不同，则接入服务器需通过内部 RPC 向目标存储服务器发起相同操作请求，并将目标服务器结果返回客户端。

4. HTTP API 约定

i. Content-type: application/json; charset=utf-8

ii. 写入/更新缓存：POST /。使用 HTTP POST 方法，请求发送至根路径，请求体为 JSON 格式的 KV 内容。

iii. 读取缓存 GET /{key}。使用 HTTP GET 方法，key 直接拼接在根路径之后。为简化程序，对 key 格式不做要求（非 URL 安全字符需要进行 urlencode）。正常：返回 HTTP 200，body 为 JSON 格式的 KV 结果；错误：返回 HTTP 404，body 为空。

iv. 删除缓存 DELETE /{key}。永远返回 HTTP 200，body 为删除的数量。

四、实验内容

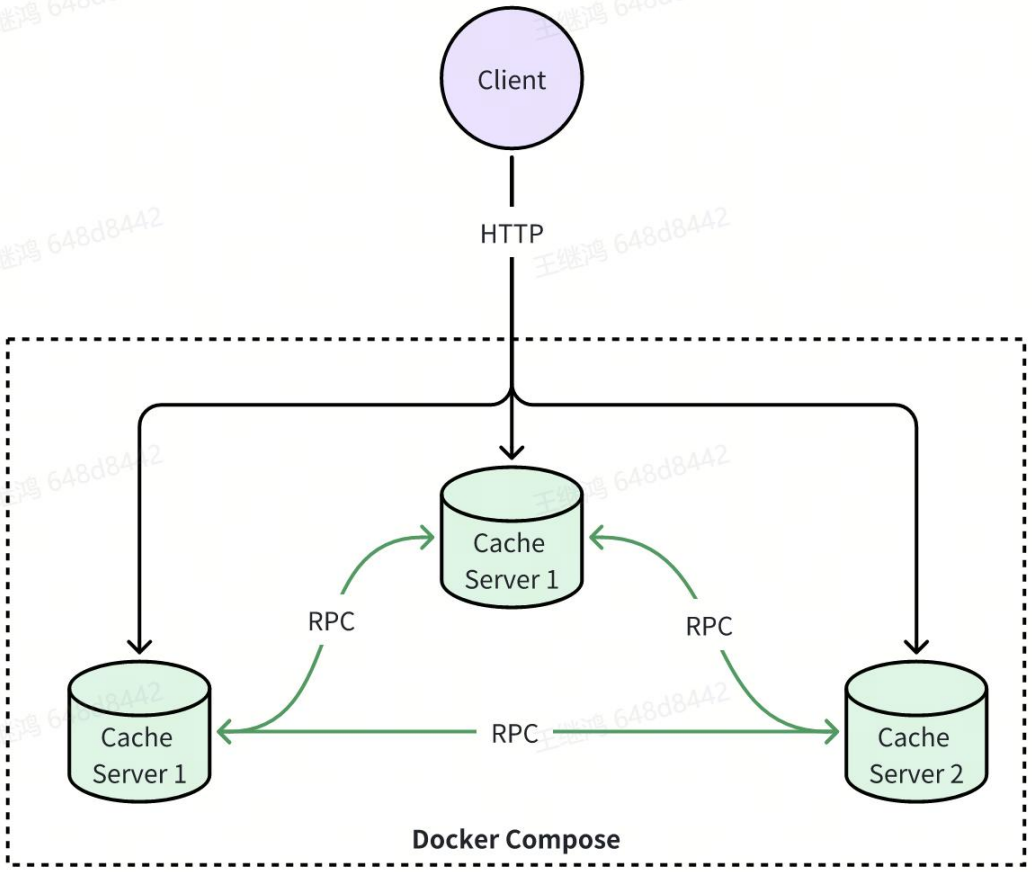

图 1 系统整体架构图

SDCS 系统的整体架构如图 1 所示。每个服务器在逻辑上包括三个部分：HTTP 服务模块、数据存取模块和 RPC 通信模块。功能的实现采用 Go 语言实现，代码通过 docker 打包并基于 Ubuntu 20.04 镜像进行构建。

五、实验步骤

5.1 HTTP 服务模块

该 HTTP 服务模块的主要功能是通过 RESTful API 对用户提供分布式缓存系统的键值存取操作接口。

1. 路由解析和请求分发：

handler 函数作为 HTTP 请求的主入口，根据 URL 路径和请求方法将请求分发到合适的处理函数。如果路径是根路径 / 且请求方法是 POST，则调用 postHandler 处理 POST 请求。如果路径是 /{key}，则根据请求方法为 GET 或 DELETE 调用 getHandler 或 deleteHandler，以处理 GET 或 DELETE 请求。若无匹配的路由，返回 404 错误。

对应代码如下：

```
func handler(w http.ResponseWriter, r *http.Request) {
    // 根据请求路径拆分 URL
    path := r.URL.Path

    // 根路径（处理 POST 请求）
    if r.Method == "POST" && path == "/" {
        postHandler(w, r)
        return
    }

    parts := strings.Split(strings.Trim(path, "/"), "/")

    // 处理 GET /{key} 和 DELETE /{key}
    if len(parts) == 1 {
        key := parts[0]
        switch r.Method {
        case "GET":
            getHandler(w, key)
        case "DELETE":
            deleteHandler(w, key)
        default:
            http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
        }
        return
    }

    // 如果没有匹配的路由，返回 404
    http.Error(w, "Not found", http.StatusNotFound)
}
```

2. POST 请求处理 (postHandler):

接收请求体并将其解码为 JSON 格式，然后解析 JSON 对象的第一个键值作为唯一键 (unique_key)。通过 keyHashFunc 将该键哈希成目标节点 ID (target_id)。如果当前节点是目标节点，则将数据存储在本地 data 字典中；否则，通过 gRPC 向目标节点发送存储请求 (PostKV)。最后，返回请求的 JSON 数据作为响应。

对应代码如下：

```
func postHandler(w http.ResponseWriter, r *http.Request) {
    var temp map[string]interface{}
    var unique_key string

    // 读取请求体并将其存储为 json.RawMessage
          var rawMessage json.RawMessage
    err := json.NewDecoder(r.Body).Decode(&rawMessage)
```

```
        if err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
                return
        }

    // 将 json.RawMessage 解码为一个 map
    err = json.Unmarshal(rawMessage, &temp)
        if err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
                return
        }

        // 获取第一个键
        for key := range temp {
       unique_key = key
       break
        }

    target_id := keyHashFunc(unique_key)

    fmt.Printf("POST target_id is %d\n", target_id)

    if target_id == my_id {
        data[unique_key] = rawMessage
    } else {
        // Contact the server and print out its response.
              ctx, cancel := context.WithTimeout(context.Background(), time.Second)
        defer cancel()
        _, err := client[target_id].PostKV(ctx, &pb.PostRequest{Key: unique_key, Json:
rawMessage})
        if err != nil {
            log.Fatalf("could not post kv: %v", err)
        }
    }

    // fmt.Fprintf(w, "Received POST with key:%v; json: %+v\n", unique_key,
string(rawMessage))
    fmt.Fprintf(w, "%+v\n", string(rawMessage))
}
```

3. GET 请求处理 (getHandler):

使用请求的键 key 调用 keyHashFunc 计算目标节点 ID。如果目标节点是当前节点，则直接从本地 data 字典获取对应的 JSON 数据。如果当前节点不是目标节点，则通过 gRPC 向目标节点发送获取请求 (GetKV)。成功找到数据后，返回 JSON 数据；如果未找到匹配数据，返回 404 错误。

对应代码如下：

```
func getHandler(w http.ResponseWriter, key string) {
    target_id := keyHashFunc(key)
    var json json.RawMessage

    fmt.Printf("GET target_id is %d\n", target_id)

    if target_id == my_id {
```

```
        var exists bool
        json, exists = data[key]
        if !exists {
            // 如果没有匹配的 key，返回 404
            // http.Error(w, "Not found key", http.StatusNotFound)
            http.Error(w, "not found", http.StatusNotFound)
            return
        }
    } else {
        // Contact the server and print out its response.
              ctx, cancel := context.WithTimeout(context.Background(), time.Second)
        defer cancel()
        r, err := client[target_id].GetKV(ctx, &pb.GetRequest{Key: key})
        if err != nil || !r.Success {
            // http.Error(w, "Not found key", http.StatusNotFound)
            http.Error(w, "not found", http.StatusNotFound)
            return
        }
        json = r.GetJson()
    }

    // fmt.Fprintf(w, "Received GET return %+v\n", string(json))
    fmt.Fprintf(w, "%+v\n", string(json))
}
```

4. DELETE 请求处理 (deleteHandler):

使用 key 计算目标节点 ID。如果目标节点是当前节点，删除本地 data 字典中的数据并返回 1 表示成功；如果未找到键，返回 0。如果目标节点不是当前节点，则通过 gRPC 向目标节点发送删除请求 (DeleteKV)。删除成功时返回 1，失败返回 0。

对应代码如下：

```
func deleteHandler(w http.ResponseWriter, key string) {
    target_id := keyHashFunc(key)

    fmt.Printf("DELETE target_id is %d\n", target_id)

    if target_id == my_id {
        _, exists := data[key]
        if !exists {
            fmt.Fprintf(w, "%d\n", 0)
            return
        }
        delete(data, key)
    } else {
        // Contact the server and print out its response.
              ctx, cancel := context.WithTimeout(context.Background(), time.Second)
        defer cancel()
        r, err := client[target_id].DeleteKV(ctx, &pb.DeleteRequest{Key: key})
        if err != nil || !r.Success {
            fmt.Fprintf(w, "%d\n", 0)
            return
        }
    }
```

```
    fmt.Fprintf(w, "%d\n", 1)
    // fmt.Fprintf(w, "Received DELETE request for key: %v\n", key)
}
```

5. 服务器初始化与启动：

在 main 函数中，启动了 gRPC 服务和 HTTP 服务。首先启动 gRPC 服务（在单独的 goroutine 中），以监听端口 50051，用于接收和处理其他节点的分布式缓存请求。启动 HTTP 服务，监听端口 8080，用于接收客户端的 POST、GET、和 DELETE 请求。

对应代码如下：

```
// Start HTTP service.
http.HandleFunc("/", handler)
fmt.Println("Starting server at :8080")
err := http.ListenAndServe(":8080", nil)
if err != nil {
    fmt.Println("Error starting server:", err)
}
```

5.2 RPC 服务模块

该 RPC 服务模块实现了一个分布式缓存系统的 gRPC 通信接口，允许节点间进行键值的存储、读取和删除操作。每个节点提供 gRPC 服务供其他节点调用，同时建立 gRPC 客户端连接以便在请求转发时与目标节点通信。

1. gRPC 服务定义与实现：

定义了分布式缓存的 gRPC 服务接口 (kvrpc.proto)，包括 PostKV、GetKV 和 DeleteKV 三个方法，用于处理键值的存储、读取和删除操作。

在 Go 语言中实现了这些 RPC 方法：

PostKV：接收键和值，将其存储在本地 data 字典中并返回成功状态。

GetKV：根据键查找本地字典中的值，如果找到则返回成功状态和数据，若未找到则返回失败状态。

DeleteKV：删除指定键的值，若键存在则返回成功状态，否则返回失败。

对应代码如下：

```
func (s *server) PostKV(ctx context.Context, in *pb.PostRequest) (*pb.PostReply, error)
{
    key := in.GetKey()
    json := in.GetJson()
    data[key] = json
          log.Printf("(%v) Post Received Key: %v Json: %v", rpc_port, key, string(json))
          return &pb.PostReply{Success: true}, nil
}

func (s *server) GetKV(ctx context.Context, in *pb.GetRequest) (*pb.GetReply, error)
{
    key := in.GetKey()
          log.Printf("(%v) Get Received Key: %v", rpc_port, key)
```

```
    json, exists := data[key]
    if !exists {
        return &pb.GetReply{Success: false, Json: nil}, nil
    }
          return &pb.GetReply{Success: true, Json: json}, nil
}

func (s *server) DeleteKV(ctx context.Context, in *pb.DeleteRequest) (*pb.DeleteReply,
error) {
    key := in.GetKey()
          log.Printf("(%v) Delete Received Key: %v", rpc_port, key)
    _, exists := data[key]
    if !exists {
        return &pb.DeleteReply{Success: false}, nil
    }
    delete(data, key)
          return &pb.DeleteReply{Success: true}, nil
}
```

2. gRPC 服务初始化：

在 main 函数中，启动 gRPC 服务，用于处理来自其他节点的缓存请求。

在一个单独的 goroutine 中监听端口 50051，启动 gRPC 服务器，并注册 ServiceKV 服务，使其他节点能够通过 gRPC 调用分布式缓存操作。

```
go func(){
    // Start KV RPC server.
    lis, err := net.Listen("tcp", fmt.Sprintf(":%s", rpc_port))
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    s := grpc.NewServer()
    pb.RegisterServiceKVServer(s, &server{})
    log.Printf("server listening at %v", lis.Addr())
    if err := s.Serve(lis); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }

}()
```

3. 客户端连接初始化：

程序启动后，逐个节点建立 gRPC 客户端连接，以便在非本地节点上执行请求。每个节点都会创建到其他节点的 gRPC 连接（除自身节点）。这里有个小 Tips：在各个服务器开启 RPC 服务后，先睡眠 1 秒再进行 RPC 客户端的连接，避免出现 RPC 连接时发现对方服务尚未开启的情况。

```
// Sleep for 1s
time.Sleep(1000 * time.Millisecond)

// Start KV RPC client.
for i := 0; i <= 2; i++ {
    if i == my_id {
        client[i] = nil
```

```
        continue
    }

    // Set up a connection to the server.
    conn, err := grpc.Dial("cache_server_" + fmt.Sprintf("%d:", i) + rpc_port,
grpc.WithTransportCredentials(insecure.NewCredentials()))
    if err != nil {
        log.Fatalf("did not connect: %v", err)
    }
    defer conn.Close()
    client[i] = pb.NewServiceKVClient(conn)
}
```

六、实验数据及结果分析

1. 代码编写完成后，使用 docker-compose up 命令启动容器（若尚未构建容器，使用 docker-compose up --build 先构建后运行）。开启三个服务器之后终端输出如图所示：

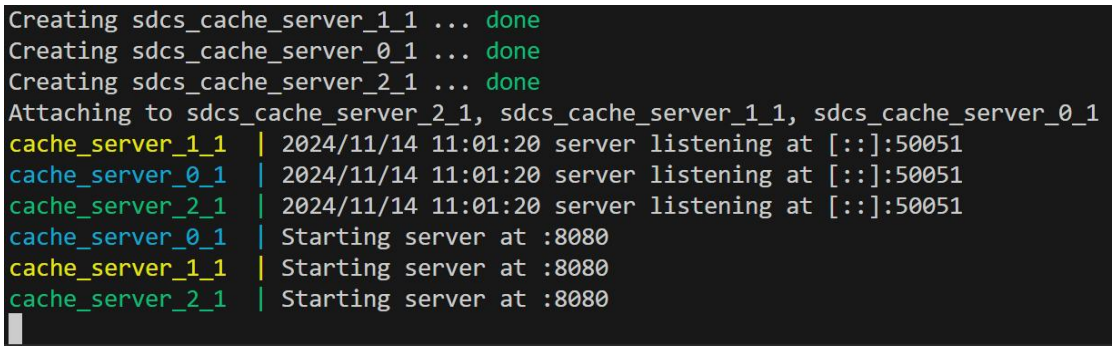

图 2 docker-compose 构建结果

2. 另起一个终端，使用 curl 工具先对 SDCS 系统进行简单的测试：

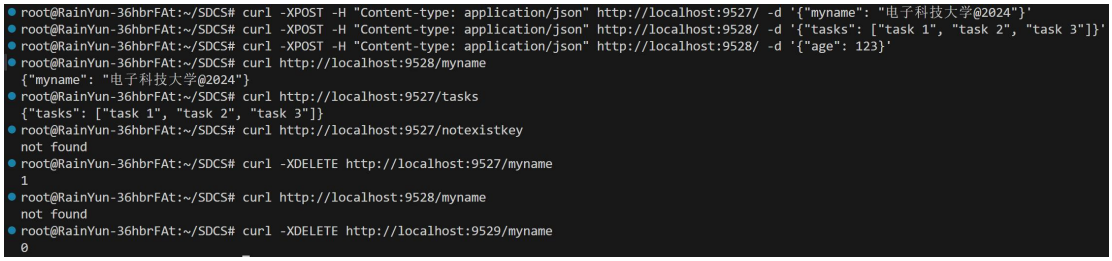

图 3 使用 curl 工具自行输入测试结果

3. 最后使用 sdcs-test.sh 脚本对 SDCS 系统进行测试：

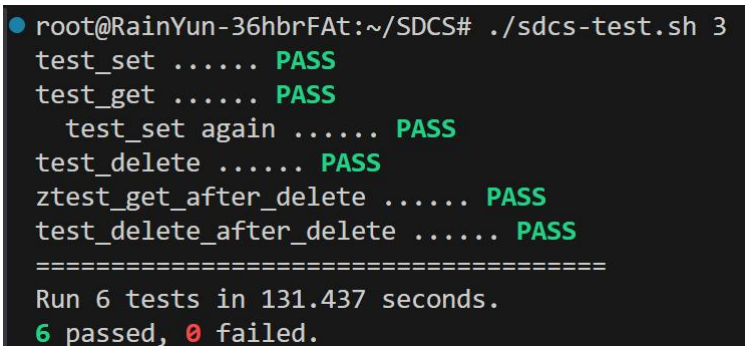

图 4 SDCS 测试脚本运行结果

七、实验结论

通过本次实验，我们成功地实现了一个简易的分布式缓存系统（Simple Distributed Cache System，SDCS）。该系统包含多个节点，每个节点提供了 HTTP 和 gRPC 接口，支持分布式缓存数据的存取操作。实验的核心目标已基本完成，具体结论如下：

1. 分布式缓存系统的实现：每个节点能够独立存储数据，并通过 gRPC 与其他节点进行通信，支持数据的远程存储、读取和删除操作。这验证了分布式系统中节点之间的互操作性。

2. 数据分布与访问：通过哈希函数（或其他策略，如轮询等），系统能将缓存数据分布到多个节点上。每个节点都提供了 HTTP API，客户端可以通过任意节点进行缓存数据的存取操作，若目标节点不在本地，接入节点能够通过 gRPC 向目标节点发起请求，确保请求的正确处理。

3. 系统稳定性与可扩展性：实验在容器化环境中通过 Docker 和 Docker Compose 实现了多节点部署，并成功进行了基础测试。每个节点都能正常启动、运行，并且相互之间能够顺利进行通信。实验表明，该系统可以支持基本的负载分配和请求处理，具备一定的稳定性和扩展性。

4. 容器化与自动化部署：利用 Docker 和 Docker Compose，成功将该分布式缓存系统容器化，简化了部署和管理流程。通过 docker-compose 工具，可以轻松启动多个节点，快速进行测试和扩展。

八、总结及心得体会

在本次实验中，我深刻体会到了分布式系统的设计与实现过程中的复杂性，特别是在数据一致性、节点通信和故障恢复等方面。通过此次实验，我不仅加深了对分布式缓存系统的理解，还学习到了如何使用现代开发工具（如 Go 语言、gRPC、Docker）进行分布式系统的开发和部署。