

```
kvstore-raft/  
├─ clerk/  
│  ├─ clerk.cpp  
│  ├─ clerk.h  
│  └─ Makefile  
├─ raft/  
│  ├─ raft.cpp  
│  ├─ raft.h  
│  └─ Makefile  
├─ kvserver/  
│  ├─ kvserver.cpp  
│  ├─ kvserver.h  
│  └─ Makefile  
├─ proto/  
│  ├─ kvstore.proto  
│  └─ Makefile  
├─ rpc/  
│  ├─ rpcchannel.cpp  
│  ├─ rpcchannel.h  
│  └─ Makefile  
├─ CMakeLists.txt  
└─ README.md
```

基本框架：

Clerk

与Raft代表的服务器进行交流：获取和设置 key-value pairs

每一个请求命令：clientId+commandId 组成

```
Clerk::Clerk(const std::string &leader_ip, uint16_t leader_port) {  
    rpc_channel = new MprpcChannel(leader_ip, leader_port);  
}
```

```
std::string Clerk::Get(const std::string &key) {  
    kvstore::GetRequest request;  
    request.set_key(key);  
  
    kvstore::GetReply reply;  
    MprpcController controller;  
  
    kvstore::KVStore_Stub stub(rpc_channel);  
    stub.Get(&controller, &request, &reply, nullptr);  
  
    if (controller.Failed()) {  
        std::cerr << "Get failed: " << controller.ErrorText() << std::endl;  
        return "";  
    }  
  
    return reply.value();  
}
```

```
void Clerk::Put(const std::string &key, const std::string &value) {  
    kvstore::PutRequest request;  
    request.set_key(key);  
    request.set_value(value);  
  
    kvstore::PutReply reply;  
    MprpcController controller;  
  
    kvstore::KVStore_Stub stub(rpc_channel);  
    stub.Put(&controller, &request, &reply, nullptr);  
  
    if (controller.Failed()) {  
        std::cerr << "Put failed: " << controller.ErrorText() << std::endl;  
    }  
}
```

设置号 RPC请求后，直接调用方法即可

```
service KVStore {  
    rpc Get(GetRequest) returns (GetReply);  
    rpc Put(PutRequest) returns (PutReply);  
}
```

KVStore 中设置了服务：

Raft

```
#include "raft.h"  
#include "../proto/kvstore.pb.h"  
#include "../rpc/rpcchannel.h"  
  
RaftNode::RaftNode(int id, const std::vector<std::string>& peers) : id(id), peers(peers)  
    state = FOLLOWER;  
}  
  
bool RaftNode::AppendEntries(const kvstore::AppendEntriesRequest& request, kvstore::Re  
    // Process the AppendEntries RPC from the leader  
    return true;  
}  
  
bool RaftNode::RequestVote(const kvstore::RequestVoteRequest& request, kvstore::Requ  
    // Handle vote requests from candidates  
    return true;  
}  
  
bool RaftNode::InstallSnapshot(const kvstore::InstallSnapshotRequest& request, kvsto  
    // Process snapshot installation from the leader  
    return true;  
}
```

kv-server

```

#include "kvserver.h"
#include "../proto/kvstore.pb.h"
#include "../raft/raft.h"

KVServer::KVServer(RaftNode* raft) : raft(raft) {}

std::string KVServer::Get(const std::string& key) {
    // Call Raft to ensure we are reading the latest committed data
    return kv_store[key];
}

void KVServer::Put(const std::string& key, const std::string& value) {
    // Replicate to other Raft nodes and update the local state
    kv_store[key] = value;
}

```

Protobuf Definitions

```

syntax = "proto3";

package kvstore;

service KVStore {
    rpc Get(GetRequest) returns (GetReply);
    rpc Put(PutRequest) returns (PutReply);
}

message GetRequest {
    string key = 1;
}

message GetReply {
    string value = 1;
}

message PutRequest {
    string key = 1;
    string value = 2;
}

message PutReply {}

service Raft {
    rpc AppendEntries(AppendEntriesRequest) returns (AppendEntriesReply);
    rpc RequestVote(RequestVoteRequest) returns (RequestVoteReply);
    rpc InstallSnapshot(InstallSnapshotRequest) returns (InstallSnapshotReply);
}

```

```
message AppendEntriesRequest {
  string leader_id = 1;
  int32 term = 2;
  repeated string entries = 3;
}

message AppendEntriesReply {
  bool success = 1;
}

message RequestVoteRequest {
  string candidate_id = 1;
  int32 term = 2;
}

message RequestVoteReply {
  bool vote_granted = 1;
}

message InstallSnapshotRequest {
  string leader_id = 1;
  bytes snapshot = 2;
}

message InstallSnapshotReply {
  bool success = 1;
}
```

RPC Communication

5. RPC Communication

Handle the communication between the clerk and the Raft leader using an RPC mechanism.

rpc/rpcchannel.cpp

cpp

复制代码

```
#include "rpcchannel.h"

void MprpcChannel::CallMethod(const google::protobuf::MethodDescriptor *method,
                              google::protobuf::RpcController *controller,
                              const google::protobuf::Message *request,
                              google::protobuf::Message *response,
                              google::protobuf::Closure *done) {
  // Handle RPC communication logic
}
```

RPC example: Clerk中的 Get()

1. Clerk中的 `stub.Get(&controller, &request, &reply, nullptr);` 语句的执行结果

是的, `stub.Get(&controller, &request, &reply, nullptr);` 这一行代码的执行会触发通过 RPC 调用在 **Protobuf Definitions** 中定义的 `KVStore` 服务的 `Get` 方法。这里的 `stub` 是通过 Protobuf 自动生成的 `stub` 类 `KVStore_Stub`, 它负责将客户端的请求序列化并通过 RPC 通道发送给服务端的 `Get` 方法, 返回结果并填充到 `reply` 中。

简化执行流程:

1. **`stub.Get` 调用:** Clerk 发起一个 RPC 调用, 向 Raft 节点的 KVServer 请求获取某个键的值。
2. **序列化请求:** Protobuf 将 `GetRequest` 序列化为字节流, 发送到服务端。
3. **RPC 通信:** 通过自定义的 RPC 通道 (即 `MprpcChannel`) 将请求发送到远程服务端的 KVServer 上。
4. **服务端处理:** 服务端的 KVServer 解包请求, 执行相应的 `Get` 方法, 处理后返回结果。
5. **反序列化响应:** Protobuf 将服务端返回的响应 (`GetReply`) 反序列化回对象, 并将结果填入 `reply` 中, 供 Clerk 使用。

2. `MprpcChannel` 是默认的吗？

`MprpcChannel` 并不是默认的，而是自定义的类。它继承自 `google::protobuf::RpcChannel`，你需要自己实现它的 `CallMethod` 方法，以处理 RPC 请求的实际网络通信和序列化/反序列化工作。

在 Protobuf 中，`RpcChannel` 是一个抽象类，提供了客户端和服务端进行远程过程调用的机制，但并不实现具体的通信逻辑。你需要实现自己的 RPC 通道类（这里是 `MprpcChannel`），用于处理底层的网络传输。

`MprpcChannel::CallMethod` 方法的作用：

- `CallMethod` 是 `RpcChannel` 类中的虚函数，负责处理 RPC 请求的整个生命周期。
- 当 Clerk 通过 `stub.Get()` 发起调用时，`CallMethod` 被自动调用，负责将 `request` 序列化、通过网络发送到目标服务器、接收服务器的响应并进行反序列化。

在上面的代码中，`MprpcChannel::CallMethod` 方法负责：

1. 将请求序列化并封装成 RPC 请求包。
2. 通过 `send()` 发送给远程的 Raft 节点或 KVServer。

3. 等待并接收服务器的响应。
4. 将响应反序列化为 Protobuf 消息类型并返回给调用者。

3. `CallMethod` 是 `MprpcChannel` 的默认方法吗？

`CallMethod` 不是默认实现的，而是你需要手动实现的。你需要在自定义的 `MprpcChannel` 类中实现该方法，以完成实际的 RPC 过程。

Protobuf 为你自动生成了 `KVStore_Stub` 这个类，`stub.Get()` 会在内部调用 `RpcChannel` 的 `CallMethod` 函数。在这里，你通过自定义的 `MprpcChannel` 实现了 `CallMethod`，让 `stub.Get()` 和 RPC 通道 `MprpcChannel` 进行实际的网络交互。

`CallMethod` 在执行时的简化流程：

1. 将请求消息（如 `GetRequest`）序列化。
2. 通过 TCP/UDP 网络发送请求。
3. 等待服务器的响应。
4. 接收到响应后将其反序列化为 Protobuf 消息，返回结果。

Get的具体实现： 客户端：

`Get` 函数的实现分为两部分：**客户端调用** 和 **服务端处理**。具体来说，**Clerk** 作为客户端发起 `Get` 请求，而服务端（Raft 的 `KVServer`）会处理这个请求，并返回相应的结果。

1. 客户端 (Clerk) 调用 `Get`

在客户端，Clerk 发起对 `Get` 方法的远程调用。客户端本身并不关心 `Get` 方法是如何具体实现的，它只会将请求封装成 RPC，并通过网络发送给服务端。

简要步骤：

- Clerk 调用 `stub.Get()`，其中 `stub` 是由 Protobuf 自动生成的 `KVStore_Stub` 类。这个类负责将 `GetRequest` 发送给远程服务。
- 在这个过程中，`stub.Get()` 会调用 `RpcChannel`（你自定义的 `MprpcChannel`）的 `CallMethod`，执行 RPC 过程，发送请求并等待响应。

服务端：

2. 服务端 (Raft KVServer) 处理 `Get`

`Get` 方法的实际处理是在 **服务端** 完成的。也就是说，当客户端的请求通过 RPC 到达服务端后，Raft 节点的 `KVServer` 会接收并执行 `Get` 方法，查找所需的 key，并返回相应的 value。

实现步骤：

1. **Protobuf 定义：** 在 `.proto` 文件中定义了 `Get` 方法和请求/响应消息类型：

```
proto
service KVStore {
    rpc Get(GetRequest) returns (GetReply);
}

message GetRequest {
    string key = 1;
}

message GetReply {
    bool success = 1;
    string value = 2;
}
```

- `GetRequest` 包含客户端想要查询的 key。
- `GetReply` 包含服务端返回的查询结果（成功与否和对应的 value）。

2. **服务端的 `Get` 方法实现：** 服务端实现了 `KVStore` 的 `Get` 方法。这是处理实际业务逻辑的地方，例如从 KV 存储中查找指定的 key 并返回对应的 value。假设这个服务端在 `KVServer` 类中：

cpp

复制代码

```

class KVServer : public KVStore {
public:
    // Get 方法的具体实现
    void Get(::google::protobuf::RpcController* controller,
            const ::GetRequest* request,
            ::GetReply* response,
            ::google::protobuf::Closure* done) override {
        std::string key = request->key();

        // 假设 kv_map 是存储 key-value 数据的哈希表
        auto it = kv_map.find(key);
        if (it != kv_map.end()) {
            response->set_success(true);
            response->set_value(it->second); // 设置 value
        } else {
            response->set_success(false);
            response->set_value(""); // 设置空 value 表示没有找到
        }

        // 通知 RPC 框架调用完成
        if (done) {
            done->Run();
        }
    }
private:
    std::unordered_map<std::string, std::string> kv_map; // 模拟的 KV 存储
};

```

proto中 response是一个message { bool success=1; bytes value=2;}

这里的 `Get` 方法接收到 `GetRequest`，从服务端的 `kv_map` 中查找 `key`，然后将查找结果写入 `GetReply` 中，并返回给客户端。

3. RPC 框架的调用:

- 当客户端发起 `Get` 请求时，RPC 框架将请求转发到服务端的 `Get` 方法。
- 服务端的 `Get` 方法在 Raft 的 `KVServer` 上执行实际的业务逻辑，查找 `key`，并将结果填入 `GetReply`，然后通过 RPC 返回给客户端。

对于获取到的 value 结果，其返回是 Get 函数结束时，自动跟随 Response，返回给 Clerk，而不需要 KV-server 再构造返回发送函数了

kv-server 作为 KVDB、clerk 和 Raft 节点的连接，那么它是怎样把 clerk 的操作信息，传给 Raft 节点，并作为 entry 保留下来的呢？

好的，那我想再问一下，就是说，kv-server 是怎样收到 clerk 的 get 请求的呢？它是一直在等待 clerk 的发送信息吗？

是的，KVServer 会一直运行，并监听来自 Clerk 的 RPC 请求。具体的工作流程如下：

KVServer 接收 Clerk 请求的流程：

1. KVServer 启动并监听端口：

- 当 KVServer 启动时，它会通过一个 RPC 服务（例如 gRPC、Muduo 或其他网络库）绑定到一个特定的端口，开启一个服务器监听。
- 这个监听是阻塞的，会一直等待外部客户端（如 Clerk）发送的 RPC 请求。

2. Clerk 发起请求：

- Clerk 会通过 `stub.Get(&controller, &request, &reply, nullptr);` 发送请求。底层通过 `MprpcChannel::CallMethod`，将封装好的 RPC 请求发送给 KVServer 所在的地址和端口。

3. KVServer 监听并处理请求：

- KVServer 的 RPC 服务处于监听状态，当收到来自 Clerk 的请求时，RPC 框架（例如 gRPC 或 Muduo）会将该请求传递给相应的处理方法，例如 `KVServer::Get`。
- 在 KVServer 中，`Get` 方法是针对 Clerk 的 `GetRequest` 请求的具体处理逻辑。

4. 处理请求并返回响应：

- `KVServer::Get` 方法会根据 `request` 中提供的 `key` 查找对应的 `value`，将结果填入 `response`，并在函数结束时自动返回给 Clerk。
- RPC 框架会负责将这个 `response` 发送回给 Clerk。

5. KVServer 继续等待下一个请求：

- 处理完当前请求后，KVServer 会继续监听下一个来自 Clerk 或其他客户端的请求，保持服务的可用性。

监听实现细节:

在后台, `KVServer` 通过某种形式的事件循环或线程池来等待客户端的请求。具体的监听实现依赖于你选择的网络库, 例如:

- **gRPC**: gRPC 内部有一个事件循环, 等待并分配处理每个传入的 RPC 请求。
- **Muduo**: Muduo 作为 C++ 的高性能网络库, 也会有 `EventLoop` 和 `TcpServer` 来监听和处理网络连接和消息。

关键点:

- **持续监听**: `KVServer` 启动后会一直在运行, 等待来自 `Clerk` 的请求。它是一个持续的服务进程, 不会主动结束, 除非被手动终止。
- **网络传输层负责**: RPC 框架或网络库负责所有网络通信的底层细节, `KVServer` 只需实现相应的方法逻辑 (如 `Get`), 而不需要关注如何接收或发送数据, 框架会自动处理这些工作。

因此, `KVServer` 的角色是一个一直在监听的服务端, 只要 `Clerk` 发出请求, 它就会立即处理并返回响应。

网络传输层负责: RPC 框架或网络库负责所有网络通信的底层细节, `KVServer` 只需实现相应的方法逻辑 (如 `Get`), 而不需要关注如何接收或发送数据, 框架会自动处理这些工作。

kv-server作为KVDB、clerk和Raft节点的连接, 那么它是怎么把clerk的操作信息, 传给Raft节点, 并作为entry保留下来的呢?

在 Raft 一致性算法中，KVServer 作为外部与 Clerk 进行交互的接口，它会将来自 Clerk 的操作（如 Get、Put 或 Append）转发给 Raft 集群进行一致性处理，并确保每个操作在 Raft 集群中被达成共识（即作为日志 entry 保留下来）。这个过程涉及几个关键步骤，以下是一个典型的工作流程：

工作流程：

1. Clerk 向 KVServer 发送请求：

- Clerk 调用 Get、Put 或 Append 操作，这些操作通过 RPC 请求发送到 KVServer，即具体的 Raft leader 节点。

2. KVServer 将请求封装为 Raft 日志条目 (Entry)：

- KVServer 并不会自己直接处理这些操作，而是会把这些操作（如 Put 或 Append 的操作）作为命令封装成 Raft 日志条目（log entry）。
- 这个日志条目会包含操作的具体内容（比如 key-value 数据操作、请求的具体参数等），并通过 Raft 协议进行一致性提交。

3. Raft Leader 处理请求并启动共识流程：

- KVServer 会调用它所集成的 Raft 模块的 Start 方法，将这个日志条目传递给 Raft 协议进行处理。
- Raft 模块会启动日志复制的过程，将这个日志条目（包含 Clerk 的操作请求）发送给其他 Raft 节点（follower）以进行一致性达成。

4. Raft 达成共识并提交日志条目：

- Raft 集群经过投票机制，达成一致后，将该日志条目在集群中提交（commit）。
- 当日志条目在大多数节点上提交后，Raft leader 会将这个操作标记为已完成。

5. Raft Leader 执行操作并返回结果:

- 一旦日志条目被提交, Raft leader 会在自己的本地状态机中执行对应的操作 (如对 key-value 数据库进行更新操作)。
- 对于 Put 或 Append 操作, Raft leader 会将更新的结果写入其本地存储。
- 对于 Get 操作, Raft leader 会在本地状态机中查找 key, 并将查找到的值返回给 KVServer。

6. KVServer 返回结果给 Clerk:

- 当操作在 Raft 中达成一致并执行后, Raft leader 会将操作的执行结果 (例如 Get 返回的值、Put 和 Append 的确认) 返回给 KVServer。
- KVServer 接收结果后, 通过 RPC 响应将结果返回给 Clerk。

关键模块解析:

1. KVServer 作为中间层:

- KVServer 不会自己处理 Clerk 的请求, 而是将这些请求通过 Raft 日志来进行处理。它的作用是将外部请求翻译成 Raft 操作, 并通过 Raft 机制保证一致性。

2. Raft 的 Start 方法:

- Raft 的 Start 方法会将 KVServer 的操作封装为日志条目并开始 Raft 一致性协议。
- 日志条目一旦被提交, 意味着大多数节点都同意了这个操作, 并且可以安全地在状态机中执行。

3. 状态机和日志:

- 每个 Raft 节点都有一个状态机, 当 Raft 日志条目达成共识时, 日志会被提交并应用到状态机中。
- 这个状态机就是你的 `key-value` 数据库, 每个节点会在本地执行相同的操作, 从而保证一致性。

4. Leader 的角色:

- `KVServer` 直接与 Raft leader 节点通信。如果 `KVServer` 当前连接的节点不是 leader, 它会将请求重定向到当前的 Raft leader。

举例:

1. Put 请求的处理过程:

- Clerk 向 `KVServer` 发送一个 Put 请求。
- `KVServer` 调用 Raft 的 `Start` 方法, 将这个 Put 请求封装为日志条目。
- Raft leader 将日志条目复制到其他节点, Raft 通过投票达成一致。
- 日志条目一旦提交, leader 执行 Put 操作, 将 `key-value` 写入本地状态机, 并将结果返回给 Clerk。

举例:

2. Get 请求的处理过程:

- Clerk 向 `KVServer` 发送一个 Get 请求。
- `KVServer` 调用 Raft 的 `Start` 方法, 将 Get 请求封装为日志条目。
- Raft leader 将日志条目复制到其他节点, 达成一致后, leader 在状态机中查找 key 对应的 value。
- leader 查找到的值通过 RPC 返回给 Clerk。

When the Clerk calls the Put or Get method on the KVServer (via RPC), the KVServer's corresponding method is invoked. **KV-server类中实现**

```
// KVServer method for handling Put requests
void KVServer::Put(const std::string& key, const std::string& value) {
    // Create a new operation (command) to be replicated
    KVOperation op;
    op.type = PUT;
    op.key = key;
    op.value = value;

    // Propose the operation to the Raft node
    int index = raft->Start(op);

    // Optionally, wait for the operation to be committed
    // This can involve synchronization mechanisms like condition variables
    // to wait until the Raft node confirms commitment
}
```

The KVServer uses the Raft node's Start method (or similar) to propose a new command (operation) for replication. **Raft类中实现:**

```
// Raft node's Start method
int RaftNode::Start(const KVOperation& op) {
    // Check if the node is the leader
    if (state != LEADER) {
        // Return an error or redirect to the leader
        return -1;
    }

    // Append the operation to the local log as a new entry
    LogEntry entry;
    entry.index = lastLogIndex + 1;
    entry.term = currentTerm;
    entry.command = op;
    log.push_back(entry);

    // Update state variables
    lastLogIndex = entry.index;

    // Start the replication process to followers
    // (This involves sending AppendEntries RPCs to followers)

    return entry.index;
}
```

Raft类中实现

```
// Leader's method to send AppendEntries to followers
void RaftNode::SendAppendEntries() {
    for (auto& follower : followers) {
        AppendEntriesRequest request;
        request.term = currentTerm;
        request.leaderId = id;
        request.prevLogIndex = follower.nextIndex - 1;
        request.prevLogTerm = log[request.prevLogIndex].term;
        request.entries = log.slice(follower.nextIndex);
        request.leaderCommit = commitIndex;

        // Send the request to the follower (asynchronously)
        follower.rpcClient->AppendEntries(request,
&RaftNode::HandleAppendEntriesReply, this);
    }
}
```

Raft类中实现

```
// Handler for AppendEntries replies
void RaftNode::HandleAppendEntriesReply(const AppendEntriesReply& reply) {
    if (reply.success) {
        // Update matchIndex and nextIndex for the follower
        follower.matchIndex = reply.matchIndex;
        follower.nextIndex = follower.matchIndex + 1;

        // Check if the log entry is committed (replicated to majority)
        if (IsCommitted(reply.matchIndex)) {
            // Update commitIndex and apply the entry to the state machine
            commitIndex = reply.matchIndex;
            ApplyLogEntries();
        }
    } else {
        // Handle failures (e.g., decrement nextIndex and retry)
    }
}
```

ApplyLogEntries Once an entry is committed, the Raft node notifies the KVServer to apply the operation to the key-value store.

```
// Applying committed log entries to the state machine
void RaftNode::ApplyLogEntries() {
    // 提交一个 commit, 就把之前没有提交, 但是可以提交的commit都一起提交了
    while (lastApplied < commitIndex) {
        lastApplied++;
        const LogEntry& entry = log[lastApplied];

        // Notify KVServer to apply the operation
    }
}
```



```
        kvServer->ApplyOperation(entry.command);
    }
}

// KVServer applies the operation
void KVServer::ApplyOperation(const KVOperation& op) {
    std::lock_guard<std::mutex> lock(kv_mutex);

    if (op.type == PUT) {
        kv_store[op.key] = op.value;
    }
    // Handle other operation types if necessary
}
```

Kvserver Response Clerk For Put operations, once the operation is applied, the KVServer can send an acknowledgment back to the Clerk.

For Get operations, since reads may not need to go through the Raft consensus (depending on your consistency requirements), the KVServer can read the value directly if it's guaranteed to be up-to-date. **KV-server类中实现**

```
// KVServer method for handling Get requests
std::string KVServer::Get(const std::string& key) {
    // Optionally, ensure the latest committed state is applied
    // This may involve interacting with the Raft node

    std::lock_guard<std::mutex> lock(kv_mutex);
    auto it = kv_store.find(key);
    if (it != kv_store.end()) {
        return it->second;
    } else {
        // Handle key not found
        return "";
    }
}
```

奥，意思就是说，KV-server收到请求后，需要先向Raft节点打报告，然后Raft节点同意操作后，并把entry记录到Logs中，然后commit之后，KV-server才可以去对DB进行擦欧总

是的，对于写操作（例如 `Put` 或 `Append`），流程如下：

1. **KVServer 收到 Clerk 的请求**：当 Clerk 向 KVServer 发送写操作（如 `Put` 或 `Append`）的请求时，KVServer 不能直接更新它的本地 key-value 存储。
2. **KVServer 向 Raft 提交请求**：
 - KVServer 将写操作（如更新 key-value 的操作）作为一个 **entry** 提交给 Raft 集群。
 - Raft 将该 entry 添加到它的日志中，并且开始在集群内复制日志，保证日志在多数节点上都存在。
3. **Raft 达成共识**：
 - 如果 Raft 集群中多数节点同意了该 entry（通过日志复制机制），Leader 会将这个 entry 标记为 **已提交 (committed)**。
 - 这个时候，Leader 知道这个写操作已经在多数节点中达成共识，可以认为这个操作是安全的，可以应用到状态机中。
4. **应用到状态机 (KV 存储)**：
 - KVServer 通过 Raft 得知该操作已提交 (committed)，然后在本地的 key-value 存储中应用该写操作。
 - 例如，Raft 通知 KVServer 执行 `Put` 操作，KVServer 更新它的本地数据库。
5. **返回结果给 Clerk**：
 - 写操作成功后，KVServer 可以向 Clerk 返回结果，表示操作已经成功完成。
 - 如果写操作失败（例如 Raft 没有达成共识，或当前 KVServer 不是 Leader），KVServer 会返回相应的错误信息，Clerk 可能需要重试或重定向到正确的 Leader。

整个写操作的关键流程可以总结如下：

1. Clerk 发送写请求到 KVServer。
2. KVServer 将请求作为 entry 提交到 Raft。
3. Raft 在集群中复制日志并达成共识。
4. Raft Commit 操作，KVServer 应用到本地数据库。
5. KVServer 返回结果给 Clerk。

对于读操作（如 `Get`），流程会略有不同：

- Clerk 发送读请求到 KVServer。
- KVServer 可能需要先确认自己是 Leader，确保它可以提供最新的数据。
- 如果确认自己是 Leader，KVServer 直接从本地的 key-value 存储中读取数据并返回给 Clerk。

在这个架构下，写操作通过 Raft 保证一致性，而读操作可以直接从本地读取，避免了性能开销。

好的非常感谢。最后几个问题：1、对于一个项目具体部署而言，clerk怎么知道Raft节点的ip+端口的呢，可以通过配置文件写好后，让clerk进程去读吗？并且读完之后，clerk每次去与Raft节点进行沟通时，如果发送的请求很多的话，会发生什么事？或者，由于raft集群处理的速度没有clerk发请求的速度快，那么这个时候怎么办？可以用LockQueue队列解决吗？2、对于一个项目具体部署而言，我现在写好了Raft相关的所有文件，我怎么在一台电脑上体现一个Raft集群呢？我需要写一个Producer文件，生成好几个Raft节点的运行进程，模拟集群工作吗？那么，这些Raft节点之间的配置信息要怎么提前共享呢？

1. Clerk与Raft节点的通信配置和高请求负载：

- Raft节点的IP和端口配置：** 可以通过配置文件来管理Raft节点的IP和端口信息。在项目中，你可以在配置文件中预定义所有Raft节点的IP地址和端口号。Clerk进程在启动时读取这些配置信息，并将它们用于与Raft节点建立连接。
- 处理大量请求：** 如果Clerk向Raft节点发送了大量的请求，而Raft集群的处理速度跟不上，这可能会导致请求积压。在这种情况下，使用队列（如 `LockQueue` ）可以帮助管理请求的顺序和流量控制。通过队列，可以将请求按顺序处理，避免请求丢失或混乱。同时，也可以在Raft节点上实施请求的批处理或限流策略，确保系统能够平稳运行。

2. 在单台电脑上模拟Raft集群：

- 启动多个Raft节点：** 要在一台电脑上模拟一个Raft集群，你需要启动多个Raft节点进程。每个进程代表一个Raft节点，它们需要监听不同的端口，并能够相互通信。可以编写一个脚本或程序（类似于 `Producer` 文件），用于启动这些节点进程并配置它们的网络设置。
- 共享Raft节点配置：** Raft节点的配置信息（如节点的IP和端口）需要在集群启动时共享。你可以通过配置文件来定义集群的成员信息，并在每个节点启动时读取这些信息。通常，这些配置信息包括所有节点的地址和端口，以便每个节点能够彼此发现和通信。

在单台机器上模拟时，你可能需要为每个节点分配不同的端口，并确保这些端口在本地机器上是开放的和可用的。通过这种方式，你可以在本地环境中测试和调试Raft算法及其实现。

1、

1. Clerk与Raft节点的通信配置和高请求负载

配置文件 `config.txt`：

plaintext

复制代码

```
# 配置Raft节点的IP和端口
raft_nodes = [
    {"ip": "127.0.0.1", "port": 5001},
    {"ip": "127.0.0.1", "port": 5002},
    {"ip": "127.0.0.1", "port": 5003}
]
```

(Json数据样式 读ip+port)

```
#include <fstream>
#include <sstream>
#include <vector>
#include <string>
#include <iostream>
#include <json/json.h> // 需要安装 jsoncpp 库

struct RaftNode {
    std::string ip;
    int port;
};

// 读取配置文件
std::vector<RaftNode> readConfig(const std::string& filename) {
    std::vector<RaftNode> nodes;
    std::ifstream file(filename);
    if (!file.is_open()) {
        std::cerr << "Failed to open config file!" << std::endl;
        return nodes;
    }
}
```



```
    Json::Value root;
    file >> root;
    const Json::Value raft_nodes = root["raft_nodes"];
    for (const auto& node : raft_nodes) {
        RaftNode raftNode;
        raftNode.ip = node["ip"].asString();
        raftNode.port = node["port"].asInt();
        nodes.push_back(raftNode);
    }
    return nodes;
}
```

```
// 模拟高请求负载
void sendRequests(const std::vector<RaftNode>& nodes) {
    // 假设这里的 sendRequestToNode 函数负责向每个节点发送请求
    for (const auto& node : nodes) {
        // 处理高负载
        for (int i = 0; i < 100; ++i) {
            // 发送请求
            // sendRequestToNode(node.ip, node.port, "GET", "key");
            std::cout << "Sending request to " << node.ip << ":" << node.port << std::endl;
        }
    }
}


int main() {
    std::vector<RaftNode> nodes = readConfig("config.txt");
    sendRequests(nodes);
    return 0;
}
```

2、

2. 在单台电脑上模拟Raft集群

启动Raft节点的示例脚本（start_nodes.sh）：

bash

 复制代码


```
#!/bin/bash

# 启动Raft节点
./raft_node --port 5001 &
./raft_node --port 5002 &
./raft_node --port 5003 &

wait
```

Raft节点代码示例（C++）：

cpp

 复制代码

```
#include <iostream>
#include <unistd.h> // 用于 sleep 函数

void startRaftNode(int port) {
    // 启动Raft节点服务
    std::cout << "Starting Raft node on port " << port << std::endl;
    // 假设这里是Raft节点的主要处理逻辑
    while (true) {
        // 节点服务代码，处理请求、与其他节点通信等
        sleep(10); // 休眠一段时间，模拟节点运行
    }
}

int main(int argc, char* argv[]) {
    if (argc != 2) {
        std::cerr << "Usage: " << argv[0] << " <port>" << std::endl;
        return 1;
    }

    int port = std::stoi(argv[1]);
    startRaftNode(port);
    return 0;
}
```

chmod 添加脚本文件的运行权限

该bash文件怎么被启动的？是raft_node.cpp 调用的吗？



`start_nodes.sh` 脚本通常是独立运行的，用来启动多个Raft节点的进程，而不是由 `raft_node.cpp` 调用的。这个脚本的目的是在你的系统中启动多个Raft节点，以便你可以模拟一个完整的Raft集群。

启动脚本的使用

1. **创建脚本文件:** 将 `start_nodes.sh` 保存到你的项目目录中。你可以使用任何文本编辑器来创建这个文件，例如 `nano` 或 `vim`。
2. **赋予执行权限:** 在终端中，给脚本文件赋予执行权限，以便可以运行它。

```
bash
```

[复制代码](#)

```
chmod +x start_nodes.sh
```

3. **运行脚本:** 通过执行脚本来启动多个Raft节点。

```
bash
```

[复制代码](#)

```
./start_nodes.sh
```

启动流程说明

脚本内容:

```
bash 复制代码  
  
#!/bin/bash  
  
# 启动Raft节点  
./raft_node --port 5001 &  
./raft_node --port 5002 &  
./raft_node --port 5003 &  
  
wait
```

- `./raft_node --port 5001 &`: 启动一个Raft节点, 监听端口5001。& 符号表示在后台运行该进程。
- `./raft_node --port 5002 &`: 启动另一个Raft节点, 监听端口5002。
- `./raft_node --port 5003 &`: 启动第三个Raft节点, 监听端口5003。
- `wait`: 等待所有后台进程完成。

启动Raft节点:

- 这个脚本直接启动了 `raft_node` 可执行文件的多个实例, 每个实例都指定了一个不同的端口。每个 `raft_node` 实例会运行 `raft_node.cpp` 文件中的代码。

`raft_node.cpp` 的作用

- **主程序:** `raft_node.cpp` 实际上是编译生成的 `raft_node` 可执行文件的源代码。每个Raft节点的实例将执行这个主程序, 创建一个 `Raft` 类对象, 并启动该节点。

关系说明

- **独立性:** 启动脚本 (`start_nodes.sh`) 和主程序 (`raft_node.cpp`) 是独立的, 启动脚本用来启动多个Raft节点进程, 而每个Raft节点进程的行为由 `raft_node.cpp` 文件中的代码定义。
- **互不依赖:** `raft_node.cpp` 不会调用 `start_nodes.sh`, 而是通过脚本来启动多个实例。脚本和程序配合工作, 实现了Raft集群的模拟。

对于Raft集群, 如果出现极端情况, 比如一个Leader与其它成员失去联系, 那么其它成员会选出一个新Leader, 但是旧Leader还会认为自己是Leader, 并且, 很奇怪的是, Clerk和该旧Leader还保持着正常的通信连接。那么我想请问: clerk如果第一个选择沟通的Raft节点就是该旧Leader, 并且该旧Leader因为还认为自己是整个集群的Leader, 所以它的状态还是Leader, 所以其

给clerk isLeader的反馈是 true。那么我想问，这个时候，clerk向该旧Leader发送Get和Put请求时，会怎么样呢？

其实，如果 raft节点对于任何操作，都必须征得一半以上的节点同一时，该情况不会有影响，因为 旧Leader无法达成此条件。但是，如果对于 读操作，只要保证是 Raft返回是Leader 状态就可以正常请求的话，此情况下：读操作很可能读到旧数据！！

处理clerk请求的同时，“心跳”RPC在同时进行着

我想问，再处理clerk请求的同时，Raft节点之间还同时在进行着“心跳”等RPC的通信吗？



是的，在Raft集群中，Raft节点之间会同时进行心跳（Heartbeat）和其他RPC通信，即使在处理Clerk请求的同时。Raft协议的设计是为了确保集群的高效性和一致性，因此它需要处理多种类型的通信。

Raft节点间的RPC通信

1. 心跳（Heartbeat）

- **用途：**心跳是一种特殊的RPC请求，用于维持领导者的地位。领导者定期向所有跟随者发送心跳消息，以防止选举超时，并保持其领导地位。
- **实现：**心跳消息通常是空的，不携带实际的操作数据，仅用于告知跟随者“我还活着”。
- **频率：**心跳的频率由Raft算法的心跳间隔决定，通常是很短的时间间隔（比如100ms）。

2. 日志复制（Log Replication）

- **用途：**领导者将它的日志条目复制到跟随者，以保证集群的一致性。每次有新的操作（如客户端的 `Put` 或 `Get` 请求）时，领导者会将这些操作作为日志条目追加到自己的日志中，并将这些条目复制到所有的跟随者。
- **实现：**这涉及到RPC调用，例如 `AppendEntries` 请求，它会包含日志条目和相关的日志信息。

3. 选举（Election）

- **用途：**当集群中的领导者失效时，节点会启动选举过程来选举一个新的领导者。这涉及到多个节点之间的投票RPC。
- **实现：**选举过程通过投票请求（`RequestVote`）和投票响应（`RequestVoteResponse`）来实



处理Clerk请求的同时的RPC通信

在Raft集群中，心跳和日志复制与Clerk请求的处理是并行进行的，主要有以下几个方面：

- **多线程或异步处理：**Raft节点通常使用多线程或异步机制来同时处理多个RPC请求。这样，节点可以同时处理心跳、日志复制和客户端请求，而不会相互阻塞。
- **优先级和资源管理：**虽然心跳和日志复制是关键的集群一致性操作，但通常这些操作的处理优先级不会高于客户端请求的处理。然而，Raft协议设计确保了所有的RPC请求都能够被有效地处理，并且不会因为某一种操作的处理导致整个系统的阻塞。
- **网络 and 消息队列：**Raft节点在内部使用网络 and 消息队列机制来管理和调度RPC请求。心跳、日志复制、选举等操作会通过独立的网络消息队列进行处理，保证了集群的高效运作。