AI 副驾驶为您详解 DeepSeek 3FS 源码

Original Accela推箱子 Accela推箱子 2025年03月22日 23:21

(/≧∀≦)/ ~ ~★ AI 使我如获新生

★~ ~ 丶(≧∀≦丶)

AI 的发展速度超乎想象。例如 **VS Code Copilot Agent** [1]，可以快速阅读大量代码，理解主要组件和交互，甚至绘制类图、流程图。实在是研究开源项目的利器。本文用它阅读 DeepSeek 3FS [2] 的源码来生成讲解，~~以蹭热点~~。后文 **皆由 AI 生成**，包括绘图。

1. 引言

DeepSeek 3FS 是一个高性能、AI 原生的分布式文件系统，旨在满足大规模模型训练和推理的苛刻 I/O 模式。与传统存储系统不同，它以 AI 优先的方式重新构想存储堆栈——优化张量数据、并行 GPU 访问和 RDMA 加速传输。通过紧密集成元数据效率、分层缓存和工作负载感知的数据放置，DeepSeek 3FS 将存储从瓶颈转变为性能推动者，根本改变了数据在现代 AI 基础设施中的流动方式。

深入复杂的代码库如 DeepSeek 3FS，可能会让人感觉像是在没有地图的密林中徘徊。但借助正确的工具，曾经看似不可逾越的障碍变成了一条光明的洞察与发现之路。在本文中，我们将详细探索 DeepSeek 3FS 文件系统，从源代码追踪其架构和核心机制。在这个过程中，我们将大量依赖一个改变游戏规则的助手：VS Code Copilot Agent。凭借其从代码片段直接生成准确直观图表的能力，Copilot Agent 不仅帮助您阅读代码——它还帮助您看懂代码。从调用图到数据流再到模块关系，它在几秒钟内可视化复杂性，使其成为任何试图掌握像 DeepSeek 3FS 这样的大型系统的不可或缺的工具。

2. 3FS 客户端架构 (src/client)

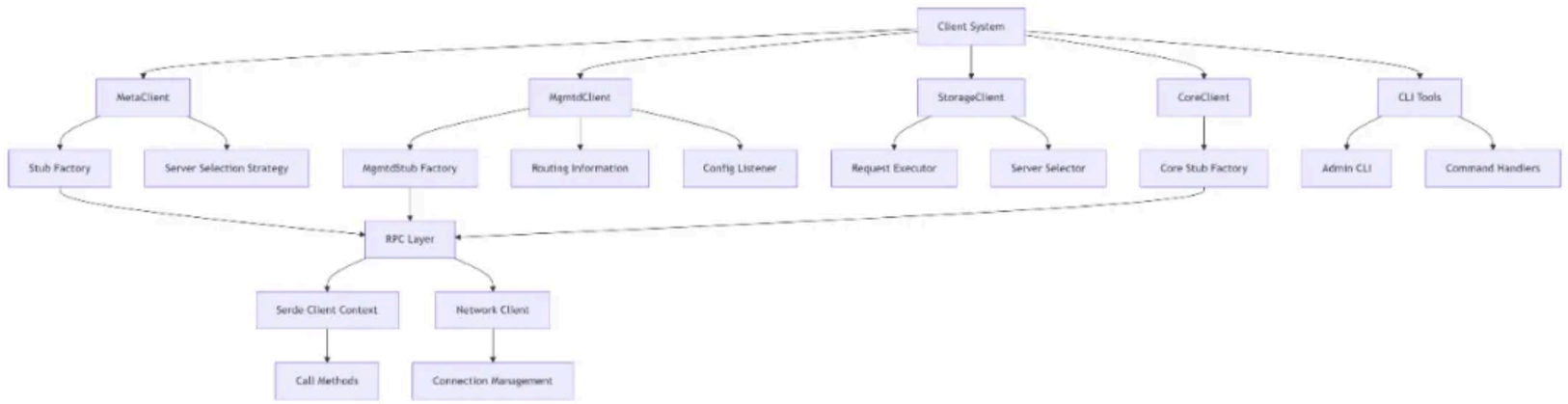
本文档提供了 3FS 系统中客户端组件的架构概述，包括它们的结构、关系和主要工作流程。

2.1. 主要组件和功能

- **MetaClient**
  : 管理文件元数据操作（创建、打开、关闭、状态、查找等）并与元数据服务器进行通信。
- **MgmtdClient**
  : 处理集群管理操作，维护路由信息，并管理客户端会话。
- **StorageClient**
  : 负责数据 I/O 操作（读/写块）并与存储服务器通信。
- **CoreClient**
  : 提供核心功能和系统级操作，贯穿整个集群。
- **CLI Tools**
  : 用于 3FS 系统的管理和维护的命令行工具。
- **Serde 层**
  : 用于 RPC 通信的序列化/反序列化框架。
- **网络层**
  : 管理与各种服务器的连接和网络通信。

2.2. 组件概述

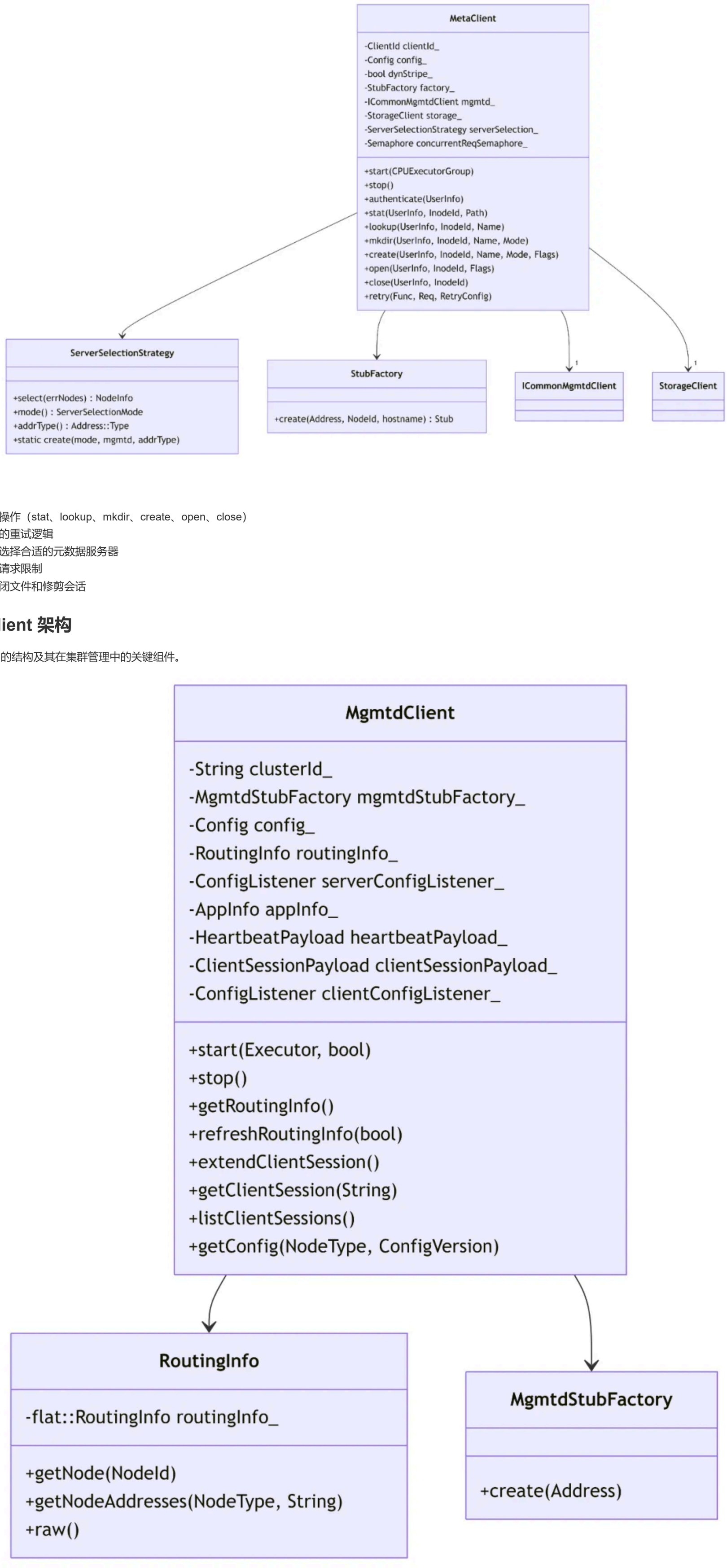
下图显示了客户端系统中主要组件之间的高层架构和关系。



2.3. 客户端组件结构

2.3.1. MetaClient 架构

此图示说明了 MetaClient 的内部结构及其与其他组件的关系。



MetaClient 功能：

- 提供文件系统元数据操作（stat、lookup、mkdir、create、open、close）
- 实现处理服务器故障的重试逻辑
- 使用服务器选择策略选择合适的元数据服务器
- 通过信号量管理并发请求限制
- 处理后台任务，如关闭文件和修剪会话

2.3.2. MgmtdClient 架构

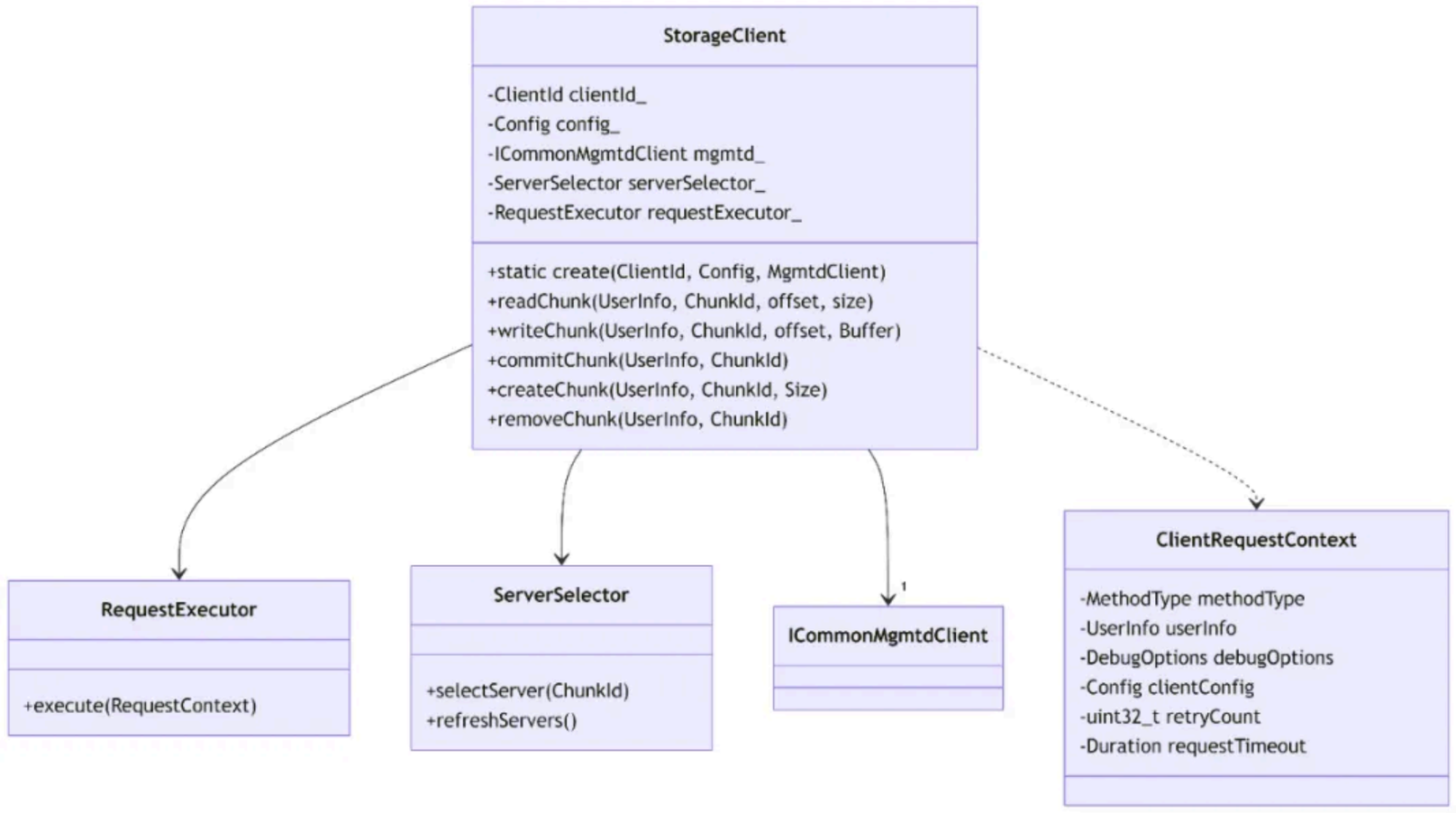
此图显示了 MgmtdClient 的结构及其在集群管理中的关键组件。

MgmtdClient 功能：

- 维护集群路由信息（哪些服务器可用）
- 通过创建和定期延长来管理客户端会话
- 提供配置管理和更新
- 为其他组件启用服务发现
- 发送心跳以保持与管理服务器的连接
- 促进客户端在集群中的识别和注册

2.3.3. 存储客户端架构

此图描绘了存储客户端的结构及其处理数据操作的组件。

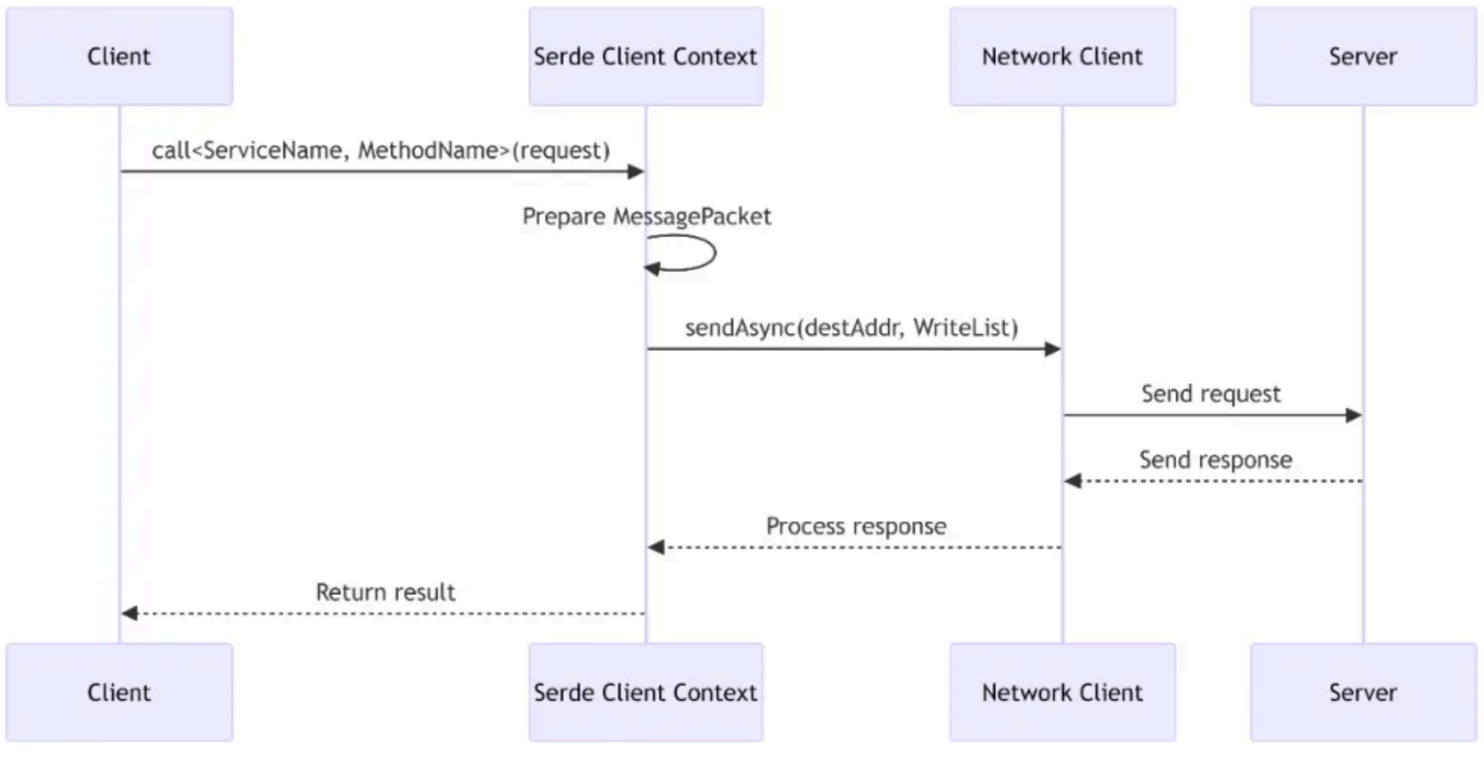


StorageClient 功能：

- 处理数据的输入/输出操作（读/写块）
- 创建、提交和删除存储块
- 为操作选择合适的存储服务器
- 实现存储操作的重试和错误处理
- 维护存储操作的指标和监控
- 通过可配置的请求处理优化数据传输

2.4. 网络通信

此图示说明了客户端和服务器之间 RPC 通信的事件顺序。

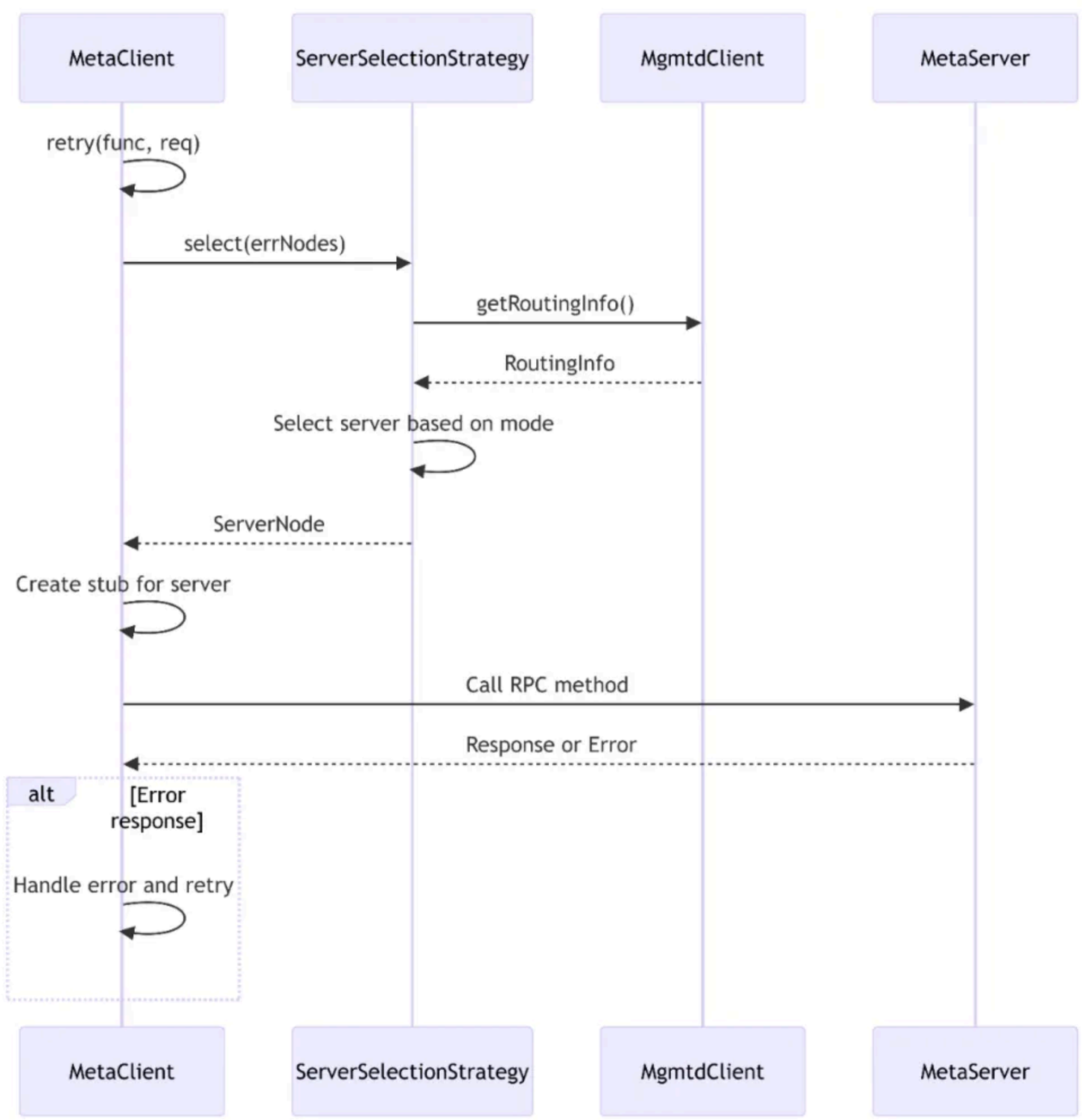


网络通信特性：

- 使用 Serde 框架序列化请求
- 支持同步和异步 RPC 调用
- 管理网络连接和重连策略
- 处理失败请求的超时和重试
- 报告网络延迟和吞吐量的指标
- 支持多种传输协议（TCP，RDMA）

2.5. 服务器选择工作流程

此图显示了客户端如何选择服务器进行元数据操作，包括错误处理和重试。

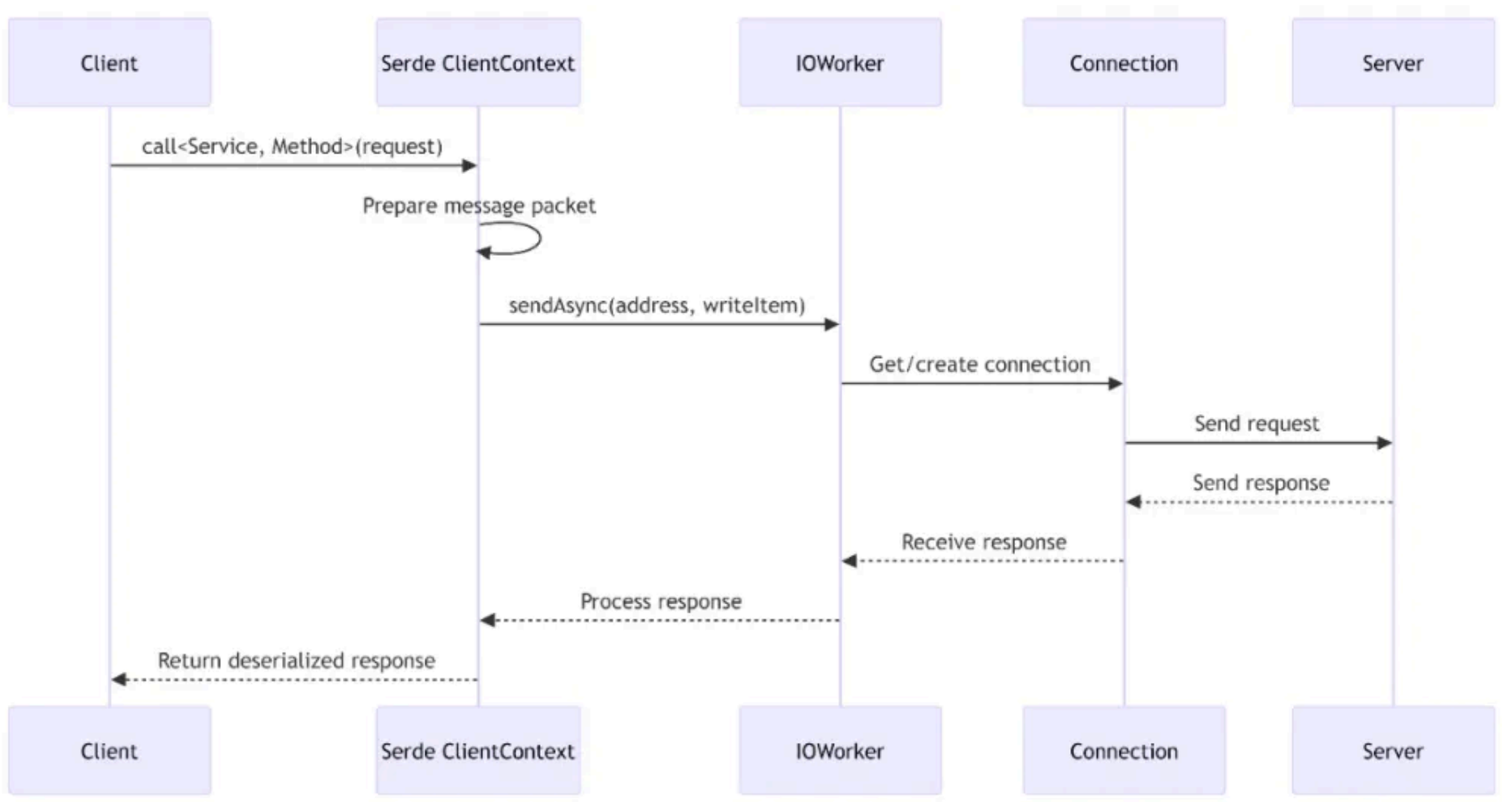


服务器选择功能：

- 多种选择策略（随机、跟随、随机跟随）
- 跟踪失败的服务器以避免再次选择它们
- 在需要时检索最新的路由信息
- 支持每个服务器不同的网络协议
- 实现可配置的重试策略，采用指数退避算法

2.6. 远程调用工作流程

此图详细说明了从客户端到服务器进行远程调用的过程，展示了涉及的内部组件。



远程调用功能：

- 支持异步（基于协程）和同步调用
- 处理连接池和重用
- 实现请求超时和取消
- 收集性能指标（延迟、吞吐量）
- 支持大负载的压缩
- 提供详细的错误信息以便调试

2.7. 命令行界面命令结构

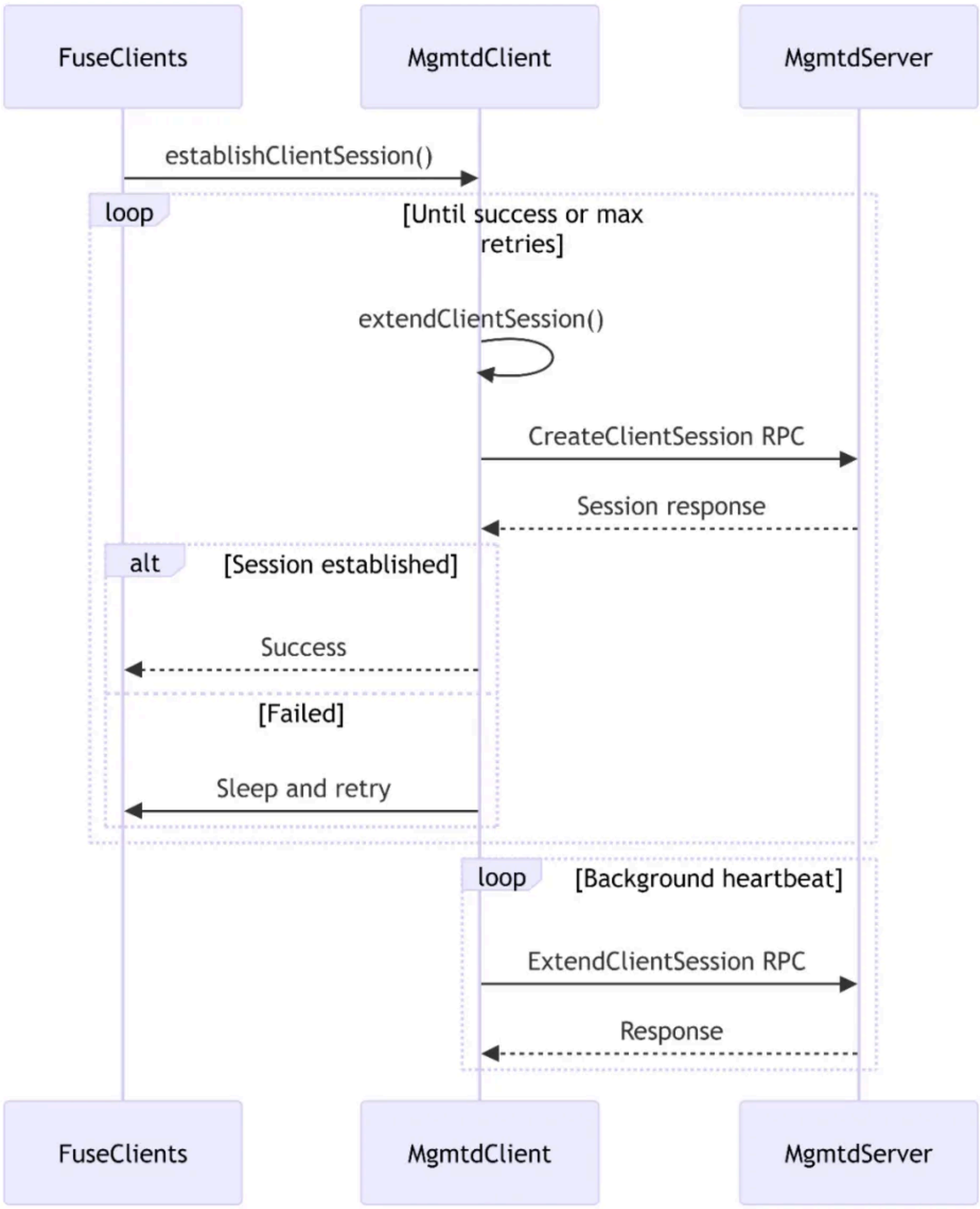
此图显示了管理员命令行工具及其命令处理程序的结构。

命令行功能：

- 提供 3FS 系统的管理界面
- 实现调试和监控的命令
- 允许与系统服务直接交互
- 支持配置文件信息的渲染
- 启用基准测试和性能测试
- 促进客户端会话管理

2.8. 客户端会话管理

此图示说明了客户端会话的生命周期，从建立到定期延续。



会话管理功能：

- 在集群中建立客户端身份
- 通过心跳保持持久会话
- 实现会话建立的重试逻辑
- 处理网络故障后的会话恢复
- 支持在会话生命周期内进行配置更新
- 使服务器能够跟踪和管理客户端

3. FoundationDB 集成架构在 3FS 中 (src/fdb)

本文提供了 3FS 系统中 FoundationDB（FDB）集成架构的概述。它涵盖了主要组件、它们的结构、关系和工作流程。

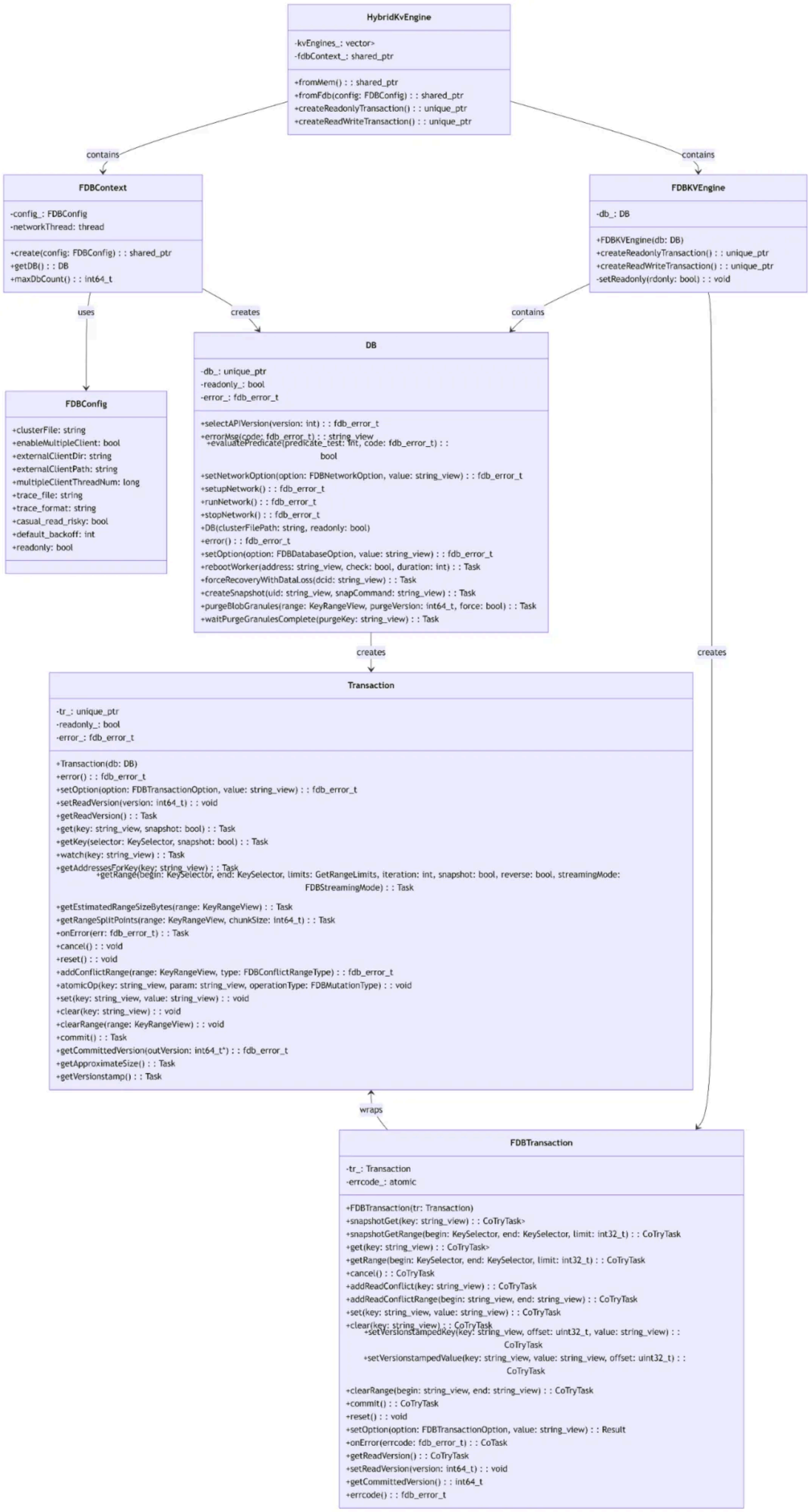
3.1. 概述

3FS 中的 FoundationDB 集成提供了一个强大的键值存储实现，能够与 FoundationDB 数据库接口。该集成围绕几个核心组件设计，这些组件处理初始化、事务和数据库操作。

3.2. 组件架构

3.2.1. 核心组件

下图显示了 FDB 集成的主要组件及其关系：



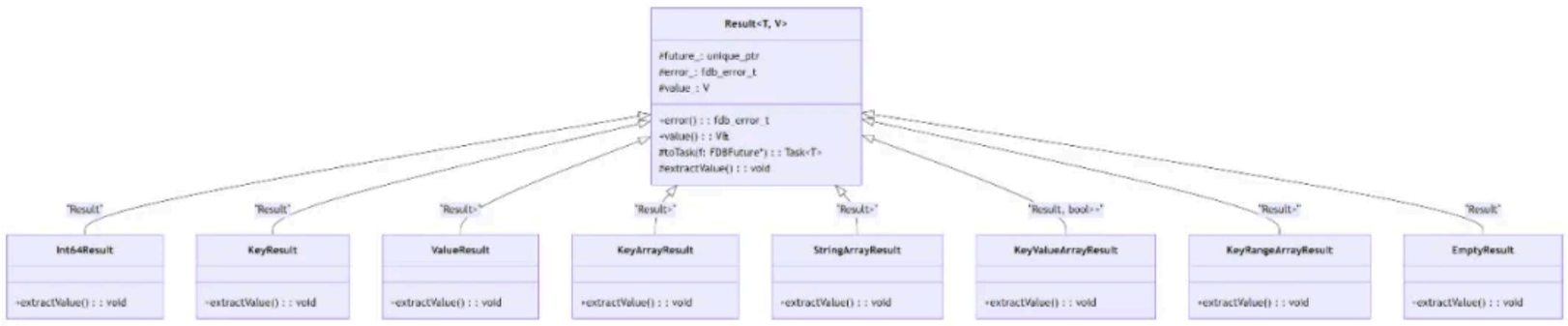
上图显示了 3FS 中 FoundationDB 集成的核心组件：

- `FDBConfig`
  : FoundationDB 连接的配置
- `FDBContext`
  : 管理 FoundationDB 连接的生命周期
- `DB`
  : 表示与 FoundationDB 数据库的连接
- `Transaction`
  : 封装原生 FoundationDB 事务
- `FDBTransaction`
  : 在 FoundationDB 之上实现 3FS 事务接口
- `FDBKVEngine`
  ： 提供一个使用 FoundationDB 的键值引擎实现

- `HybridKvEngine`
  ：允许使用多个键值引擎，包括 FoundationDB

3.2.2. 结果类层次结构

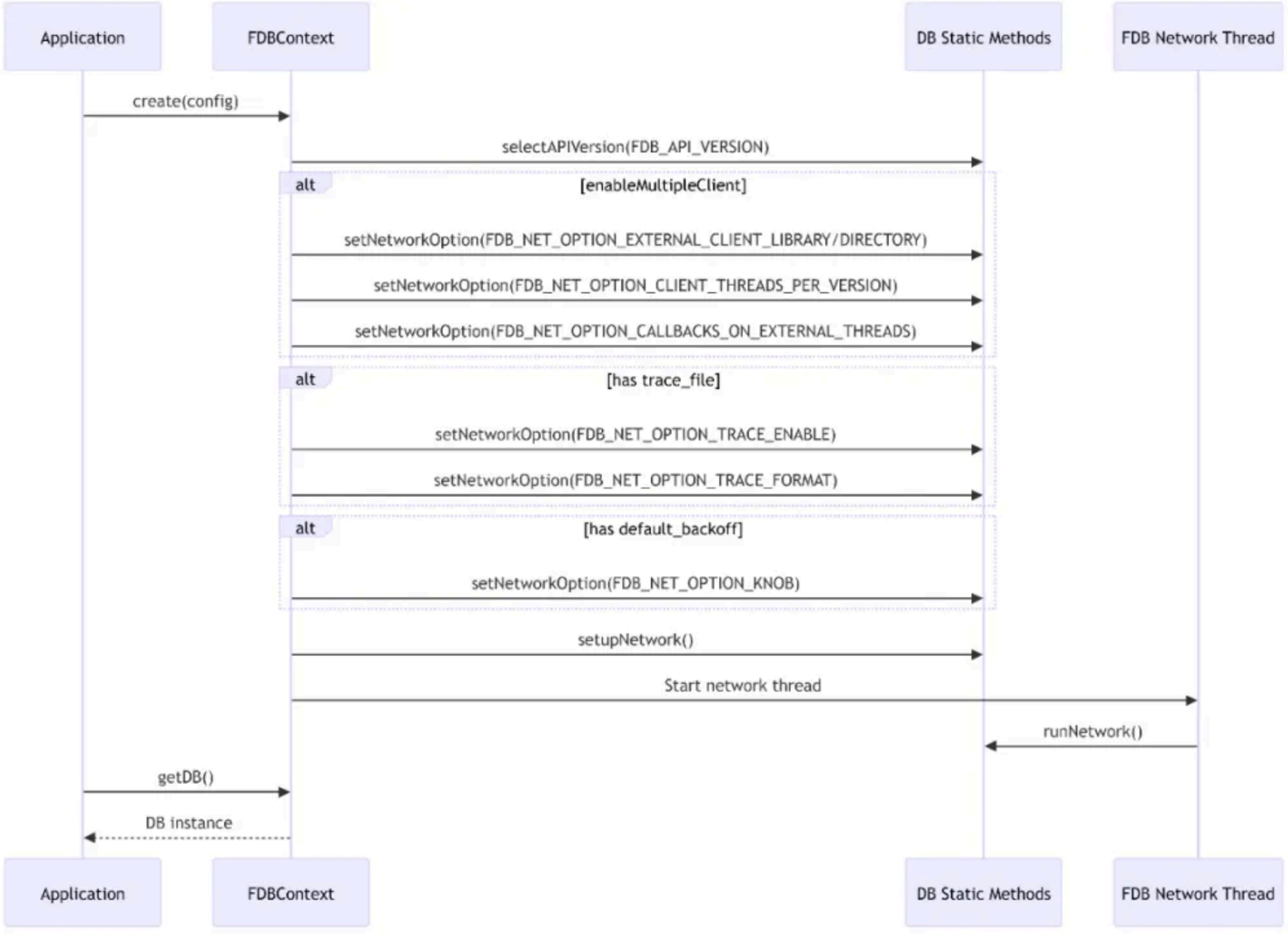
下图显示了用于处理与 FoundationDB 的异步操作的结果类：



结果模板类层次结构允许对不同 FoundationDB 操作结果进行类型安全处理。

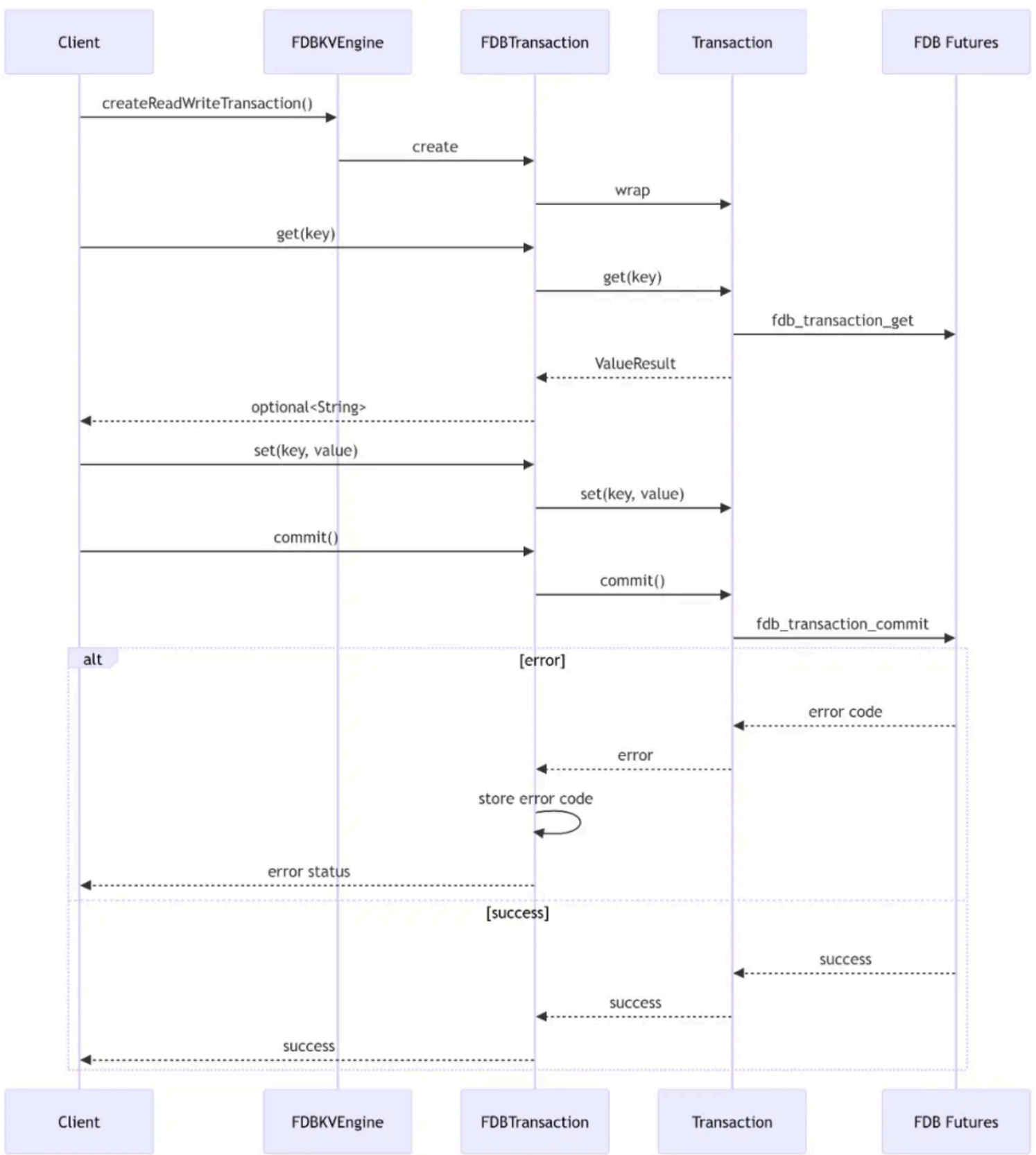
3.3. 操作工作流程

3.3.1. 初始化工作流程



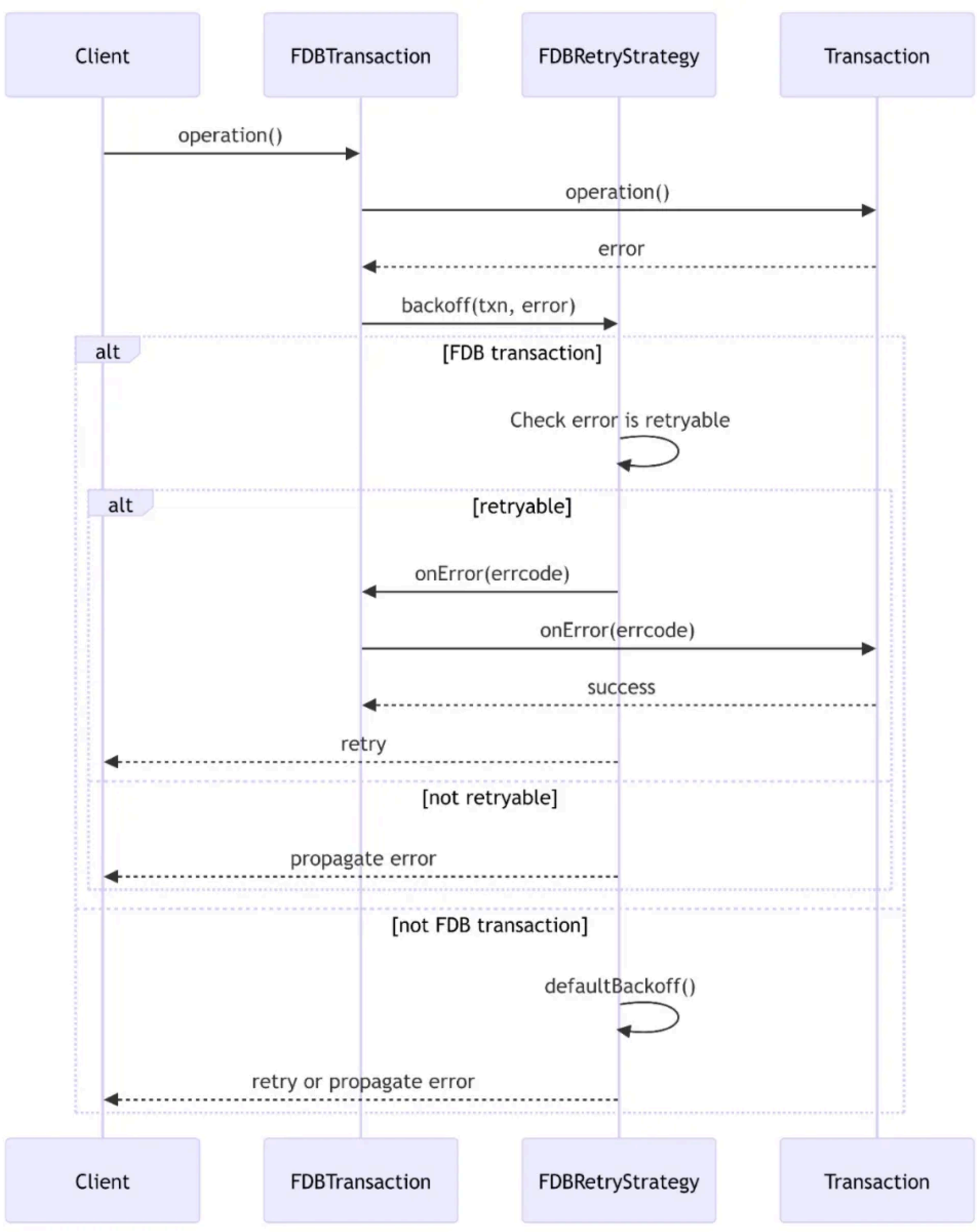
此图示说明了 FoundationDB 集成的初始化过程，展示了 FDB 网络的设置和数据库实例的创建。

3.3.2. 事务工作流程



此图展示了事务的工作流程，包括如何通过架构的各个层处理获取、设置和提交等操作。

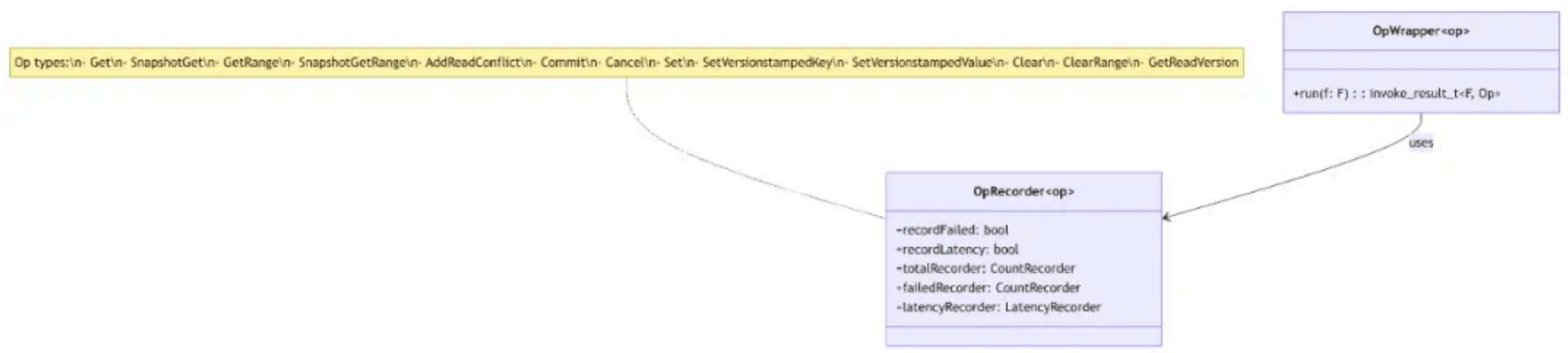
3.3.3. 错误处理和重试工作流程



此图说明了如何处理错误以及在 FoundationDB 事务中对瞬态错误的重试机制。

3.4. 操作监控

FDB 集成包括通过操作记录器的监控功能：



监控系统使用专用记录器来跟踪不同类型的操作，跟踪的指标包括：

- 总操作计数
- 失败的操作计数
- 操作延迟

3.5. 关键组件功能

3.5.1. FDB 上下文

`FDBContext` 类管理 FoundationDB 连接的生命周期，包括：

- 选择 API 版本
- 配置网络选项
- 设置和运行网络线程
- 创建数据库实例

3.5.2. 数据库

`DB` 类表示与 FoundationDB 数据库的连接，并提供：

- 用于全局 FoundationDB 操作的静态方法
- 数据库级别的操作和选项
- 事务对象的创建

3.5.3.事务

`Transaction` 类封装了一个原生的 FoundationDB 事务，并提供：

- 读写操作
- 事务选项和管理
- 使用 folly::coro::Task 的异步操作

3.5.4. FDBTransaction

`FDBTransaction` 类实现了 3FS 事务接口并提供：

- 高级事务操作
- 错误处理和转换
- 与监控系统的集成

3.5.5. FDBKVEngine

`FDBKVEngine` 类提供了一个使用 FoundationDB 的键值引擎实现：

- 创建实现 3FS 事务接口的事务
- 管理数据库连接

3.5.6. HybridKvEngine

`HybridKvEngine` 允许使用多个键值引擎：

- 可以使用 FoundationDB 作为后端创建
- 可以使用内存作为后端创建
- 将操作路由到适当的底层引擎

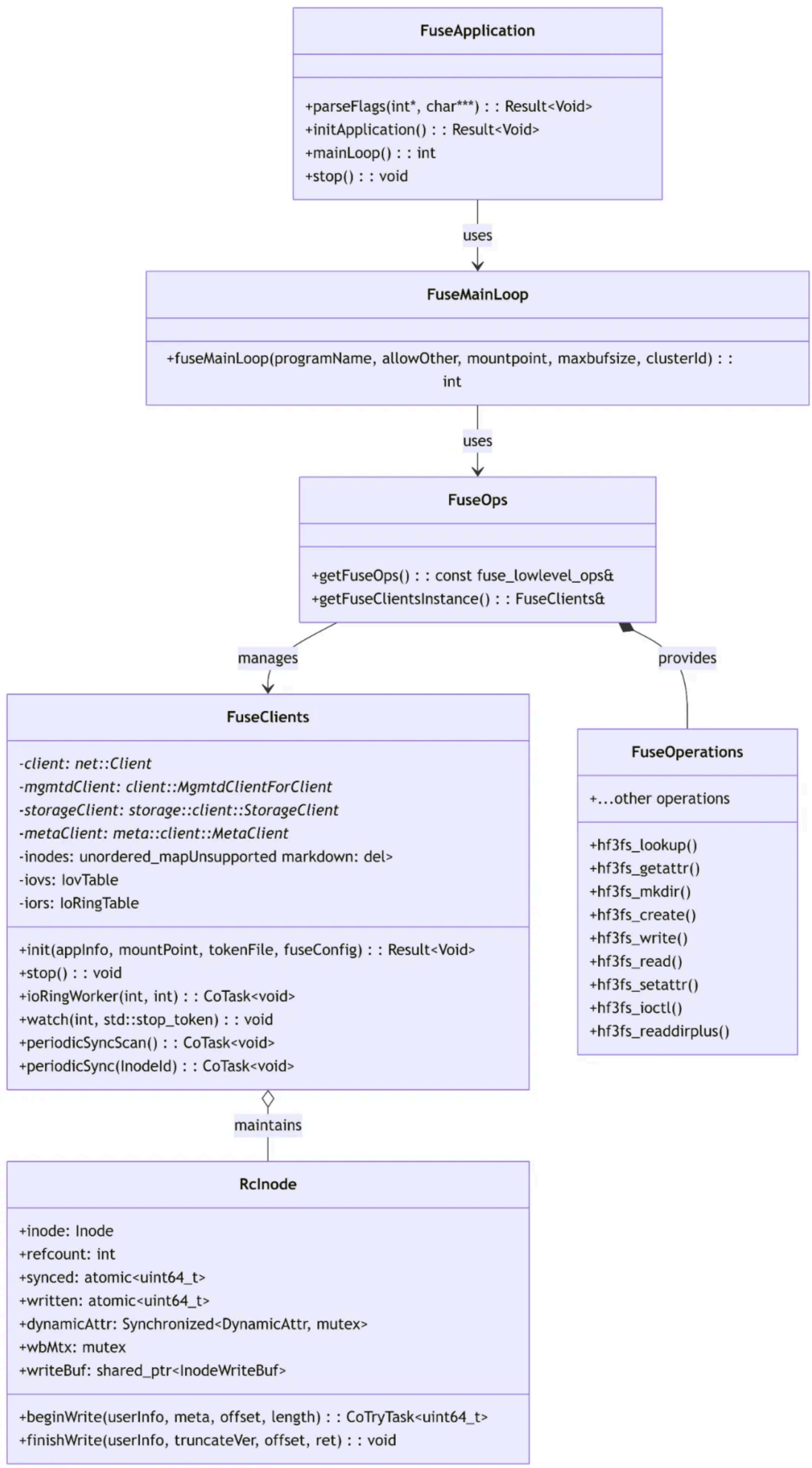
4. 3FS FUSE 架构 (src/fuse)

本文档提供了 3FS 系统中 FUSE（用户空间文件系统）实现的概述。它包括组件图、工作流程插图和主要功能的描述。

4.1. 概述

3FS FUSE 模块实现了一个 FUSE 接口，允许将 3FS 文件系统作为操作系统中的标准文件系统挂载。它将文件系统操作（例如，读取、写入、创建目录）转换为对底层存储和元数据服务的适当调用。

4.2. 主要组件



此图显示了 FUSE 实现的主要组件及其关系。

4.3. 组件详细信息

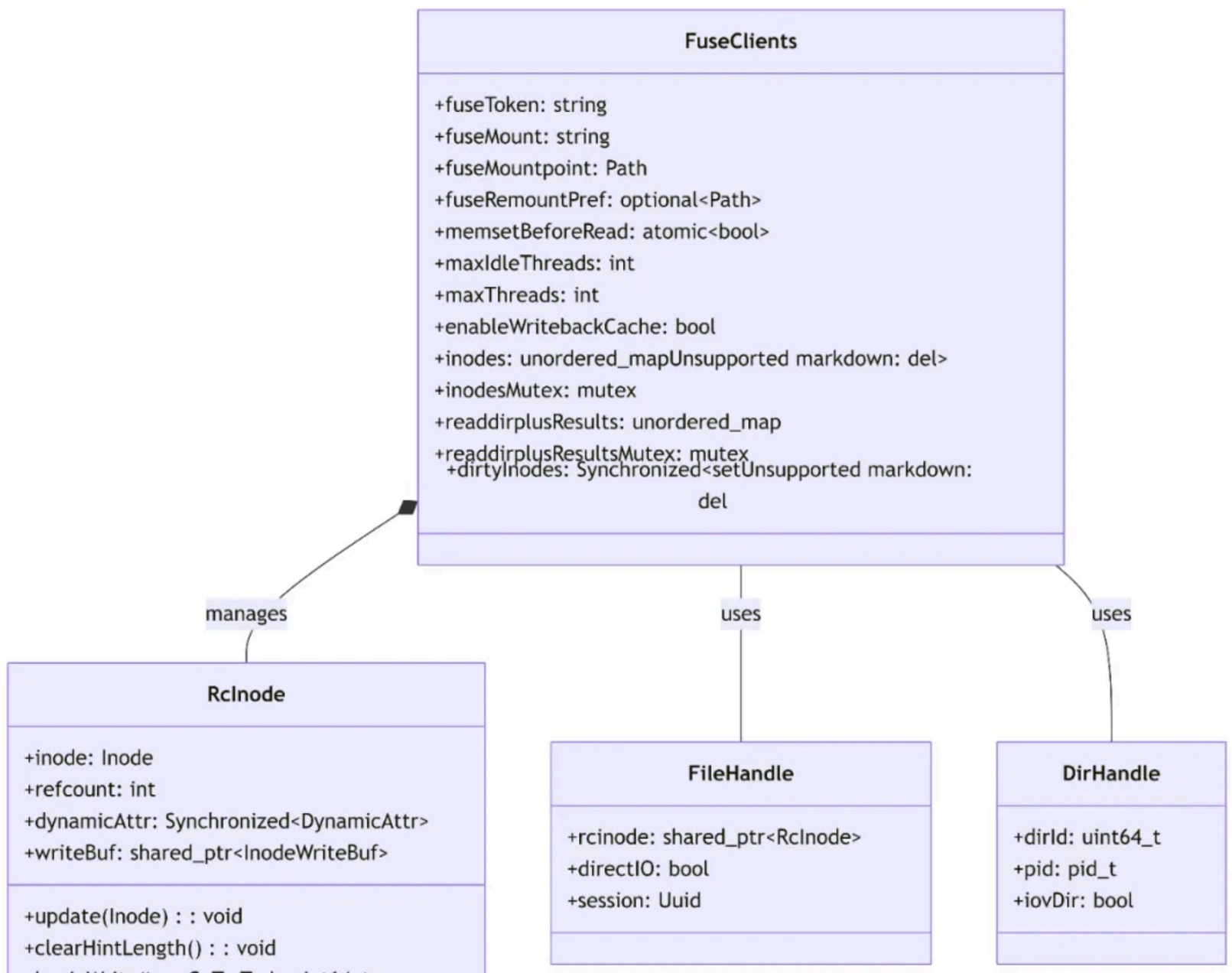
4.3.1. FuseApplication

初始化 FUSE 文件系统并处理应用程序生命周期的主要应用程序类。

4.3.2. FuseClients

中央管理类：

- 维护与元数据和存储服务的连接。
- 管理 inode 缓存和状态
- 处理 I/O 操作和后台任务
- 协调定期同步



此图显示了 FuseClients 及相关类的内部结构。

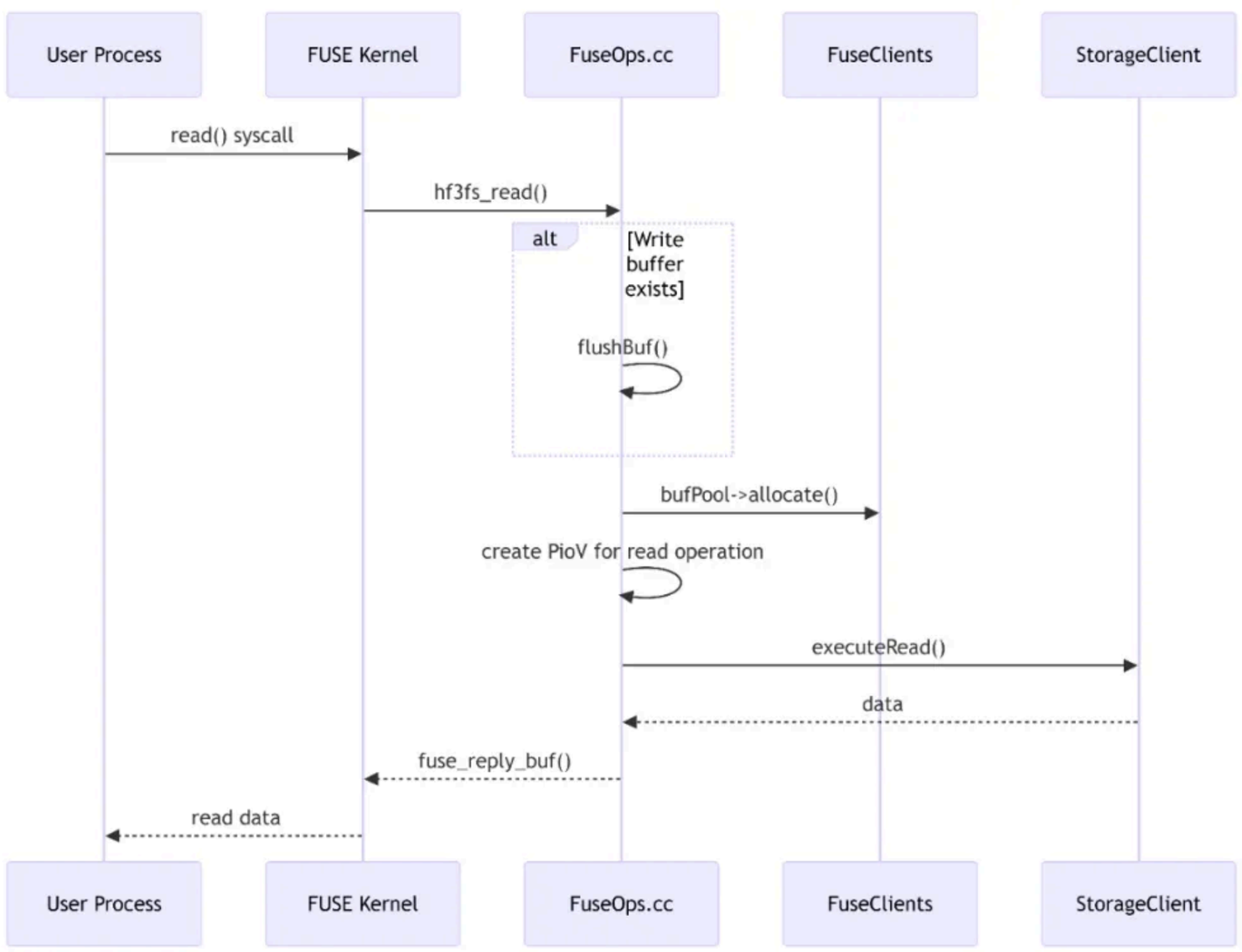
4.3.3. FuseOperations (在 FuseOps.cc 中)

实现了 fuse_lowlevel_ops 接口所需的所有 FUSE 文件系统操作，包括：

- 文件操作（读取、写入、创建）
- 目录操作（创建目录、读取目录）
- 属性操作（getattr，setattr）
- 特殊操作（ioctl，xattr）

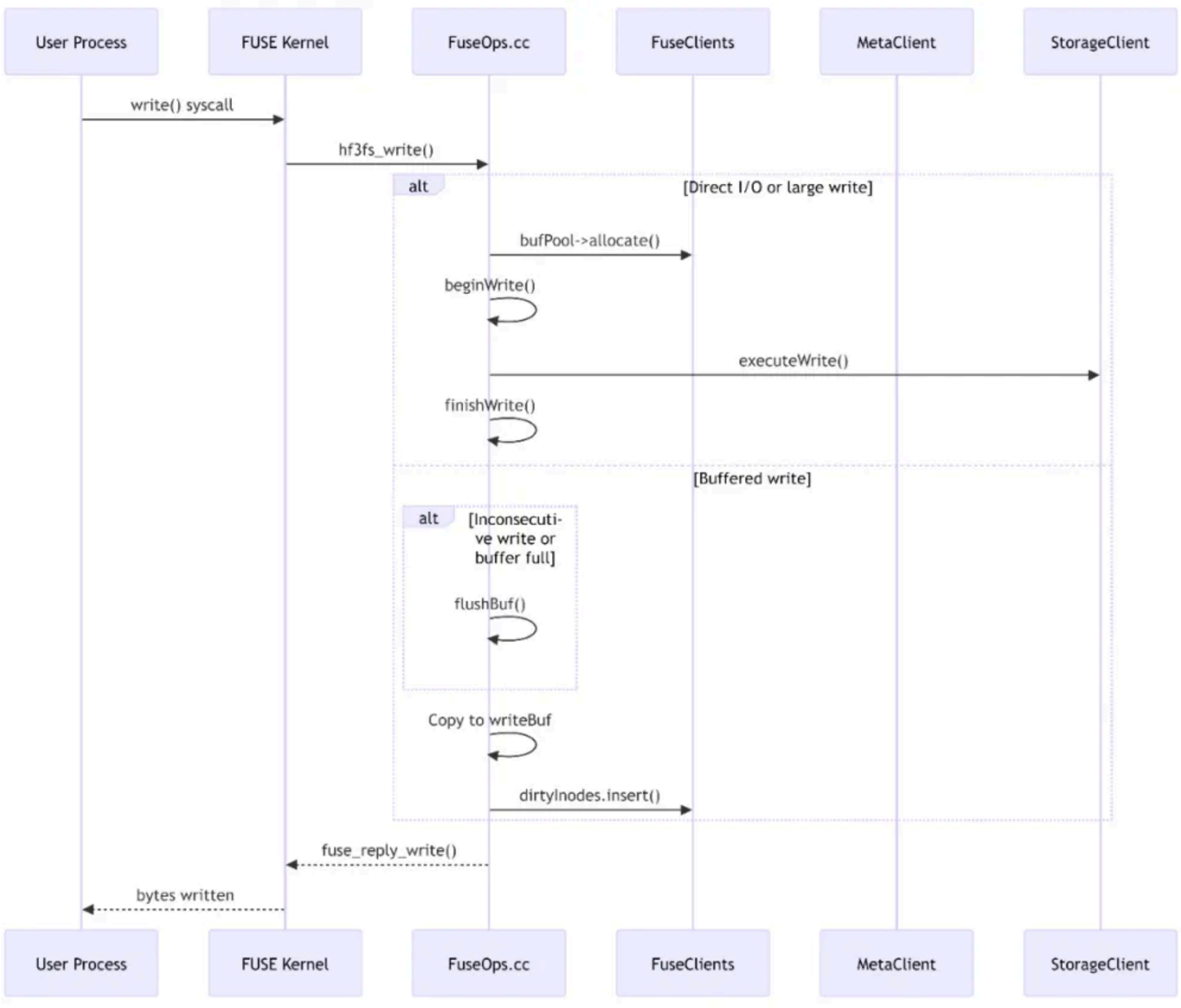
4.4. 主要工作流程

4.4.1. 文件读取工作流程



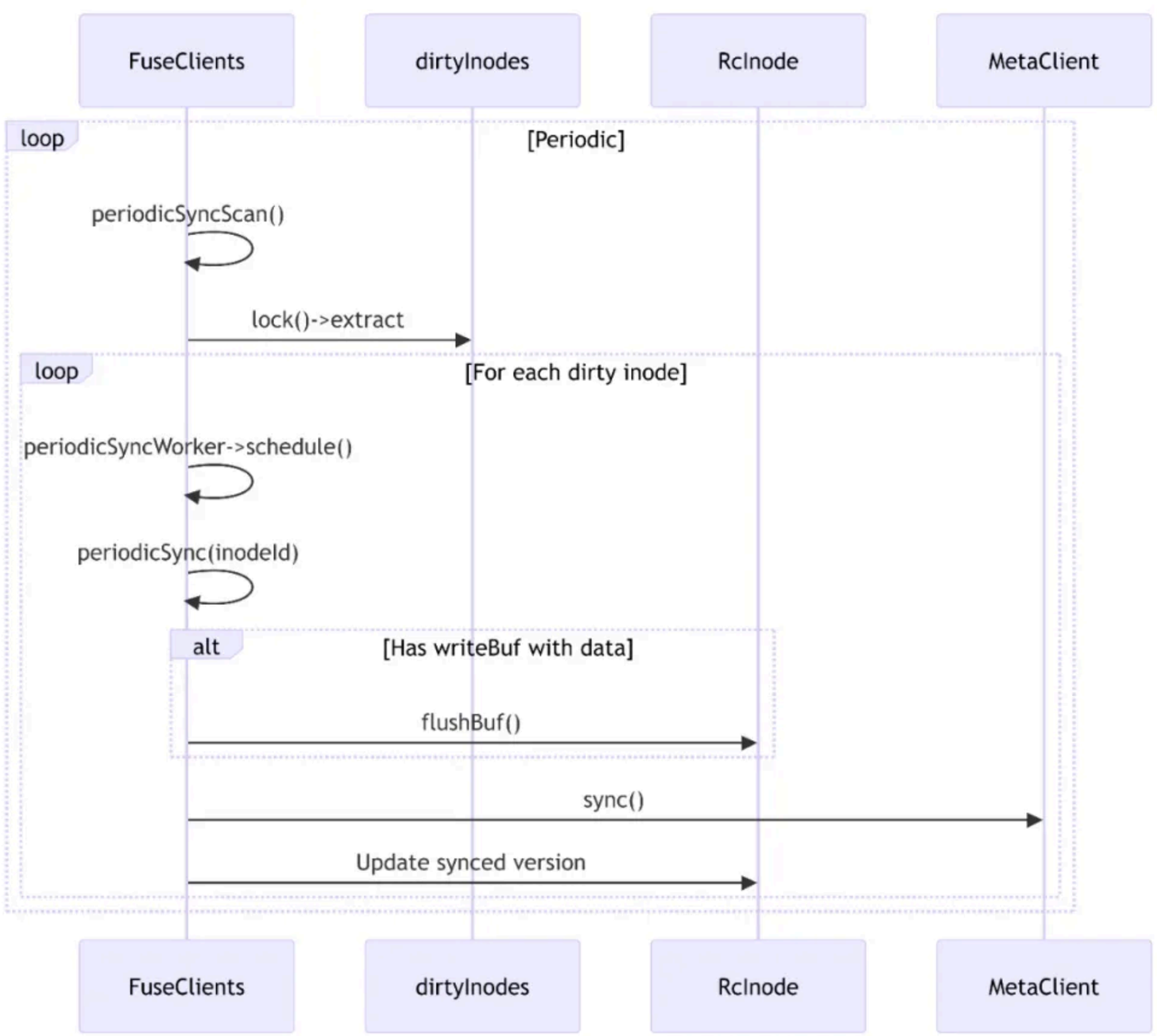
此图示说明了用户进程通过 FUSE 层到存储后端的读取操作流程。

4.4.2. 文件写入工作流程



此图示展示了写入操作的处理方式，包括直接 I/O 和缓冲写入情况。

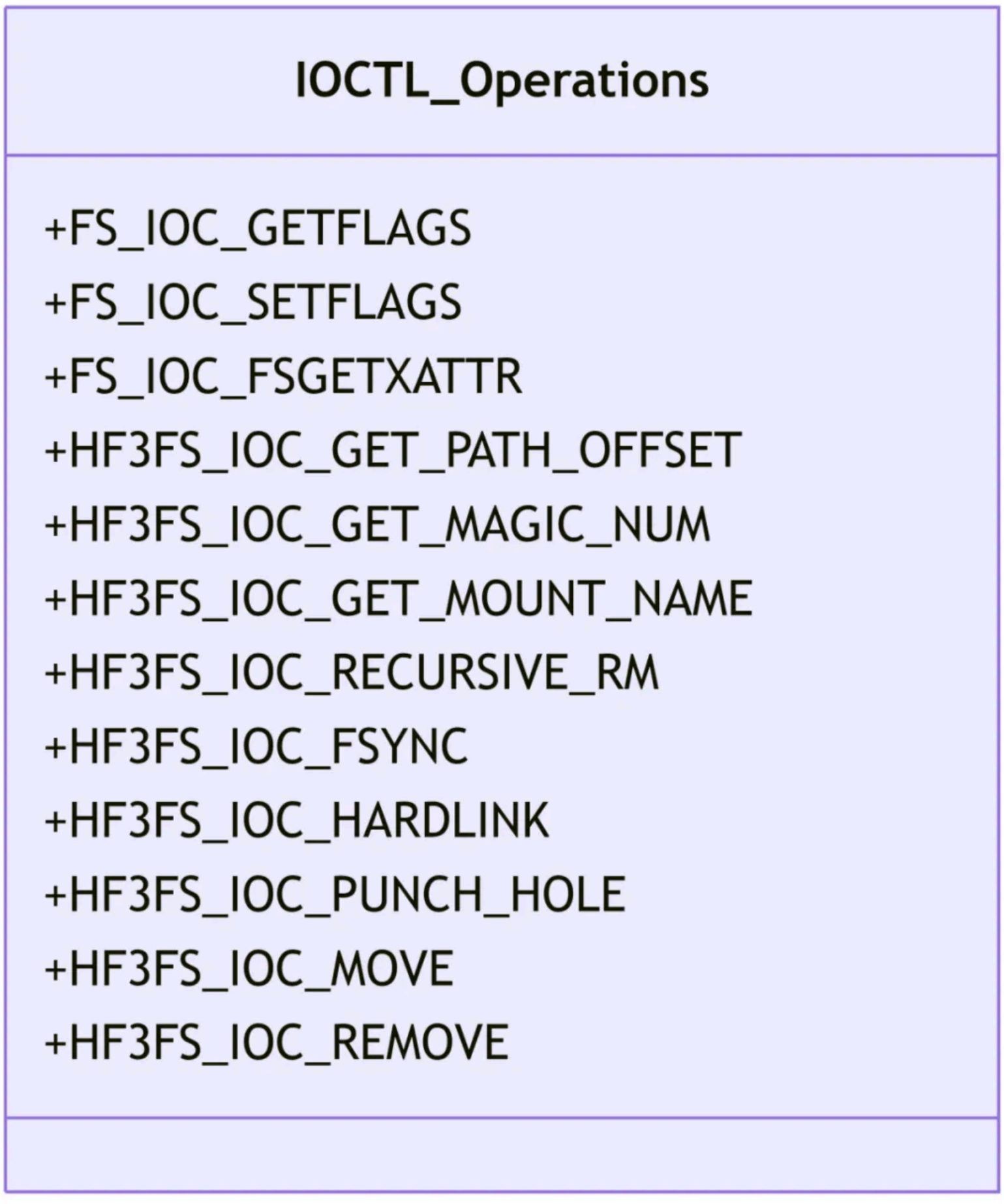
4.4.3. 定期同步工作流程



该图表显示了脏 inode 如何定期与元数据服务器同步。

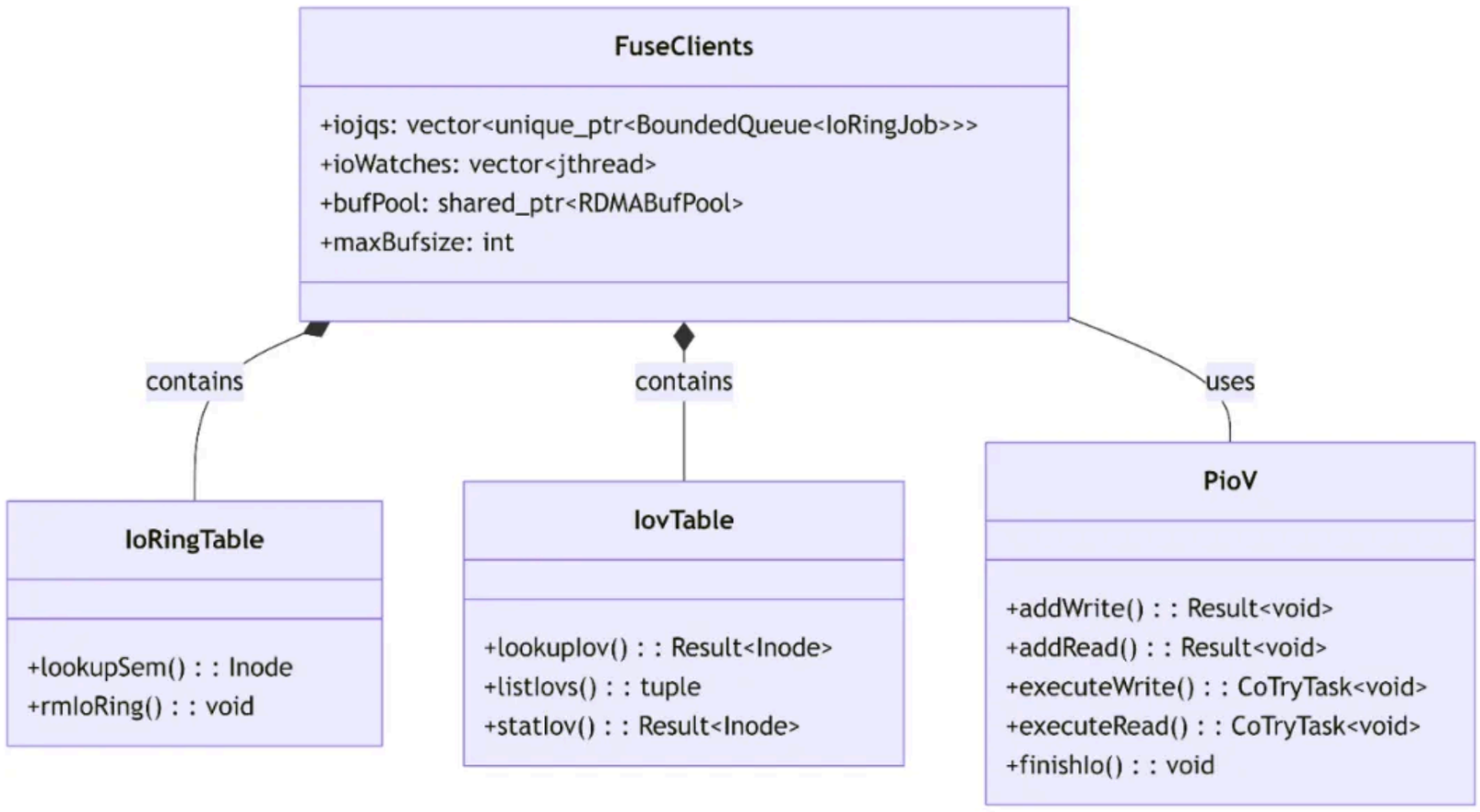
4.5. IOCTL 操作

3FS 实现了用于特殊文件系统功能的自定义 IOCTL 操作：



这些操作为文件系统提供了扩展功能，而这些功能并不包含在标准 POSIX 操作中。

4.6. I/O 环和缓冲区管理



此图显示了如何通过环形缓冲区和内存池管理 I/O 操作。

4.7. 关键特性和优化

1. **写入缓冲**
   : 小写入被缓冲以通过减少存储操作的数量来提高性能。
2. **定期同步**
   : 背景线程定期同步脏的 inode，以确保数据的持久性。
3. **内存管理**
   : 使用缓冲池高效管理 I/O 操作的内存。
4. **缓存**
   : 实现 inode 元数据和目录条目的缓存，以减少对元数据服务器的调用。
5. **可配置性**
   : 提供广泛的配置选项，如读/写超时、缓冲区大小和同步间隔。
6. **优化的 I/O**
   ： 使用 PioV（并行 I/O 向量）进行高效的 I/O 操作，以便于存储后端。
7. **直接 I/O 支持**
   ： 支持直接 I/O，以在需要时绕过内核页面缓存。

4.8. 结论

3FS FUSE 实现提供了一个完整的 POSIX 文件系统接口，构建在分布式 3FS 存储系统之上。通过精心的缓冲管理、缓存和同步策略，它在性能和可靠性之间取得了平衡。

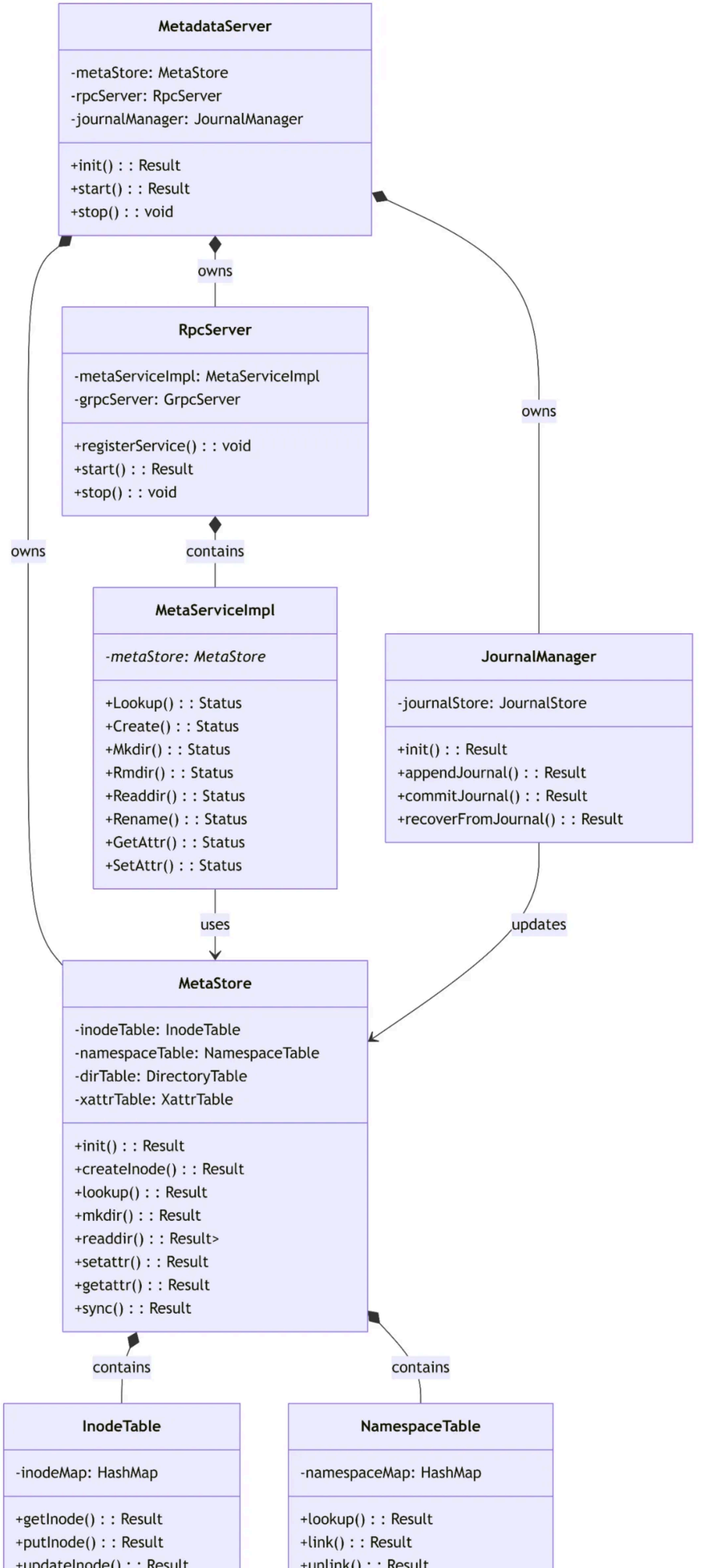
5. 3FS 元数据服务架构 (src/meta)

本文档提供了 3FS 系统中元数据服务实现的架构概述，包括组件图、内部结构、工作流程插图以及主要功能的描述。

5.1. 概述

元数据服务是 3FS 分布式文件系统的关键组件，负责管理文件系统元数据，包括 inode、目录、文件属性和命名空间操作。它确保元数据在分布式系统中的一致性、持久性和高可用性。

5.2. 主要组件

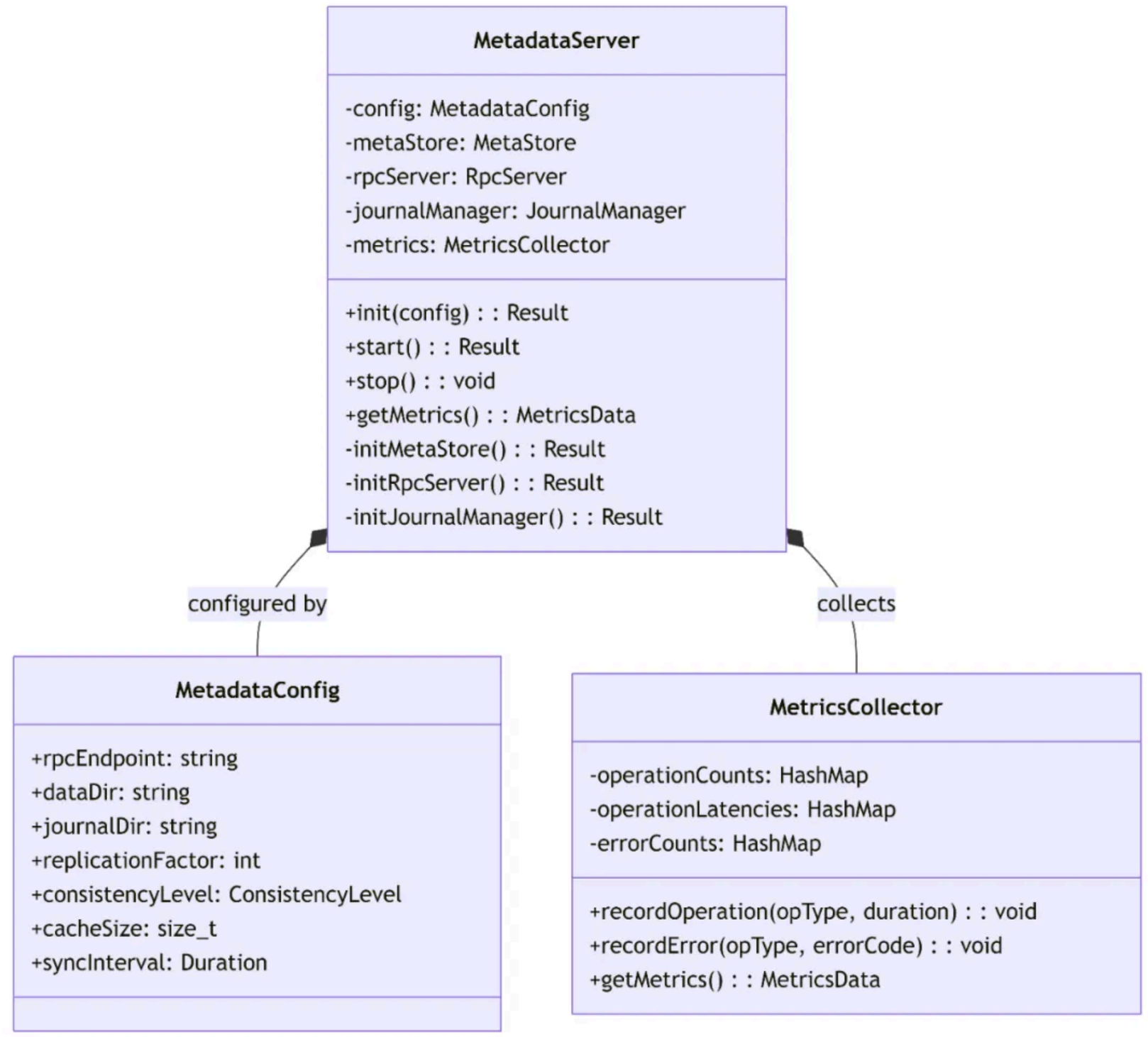

MetadataServer
-metaStore: MetaStore
-rpcServer: RpcServer
-journalManager: JournalManager
+init() : : Result
+start() : : Result
+stop() : : void
owns
RpcServer
-metaServiceImpl: MetaServiceImpl
-grpcServer: GrpcServer
+registerService() : : void
+start() : : Result
+stop() : : void
owns
contains
MetaServiceImpl
-metaStore: MetaStore
+Lookup() : : Status
+Create() : : Status
+Mkdir() : : Status
+Rmdir() : : Status
+Readdir() : : Status
+Rename() : : Status
+GetAttr() : : Status
+SetAttr() : : Status
owns
JournalManager
-journalStore: JournalStore
+init() : : Result
+appendJournal() : : Result
+commitJournal() : : Result
+recoverFromJournal() : : Result
uses
updates
MetaStore
-inodeTable: InodeTable
-namespaceTable: NamespaceTable
-dirTable: DirectoryTable
-xattrTable: XattrTable
+init() : : Result
+createInode() : : Result
+lookup() : : Result
+mkdir() : : Result
+readdir() : : Result>
+setattr() : : Result
+getattr() : : Result
+sync() : : Result
contains
contains
InodeTable
-inodeMap: HashMap
+getInode() : : Result
+putInode() : : Result
+updateInode() : : Result
NamespaceTable
-namespaceMap: HashMap
+lookup() : : Result
+link() : : Result
+unlink() : : Result

此图显示了元数据服务的主要组件及其关系。

5.3. 组件详细信息

5.3.1. 元数据服务器

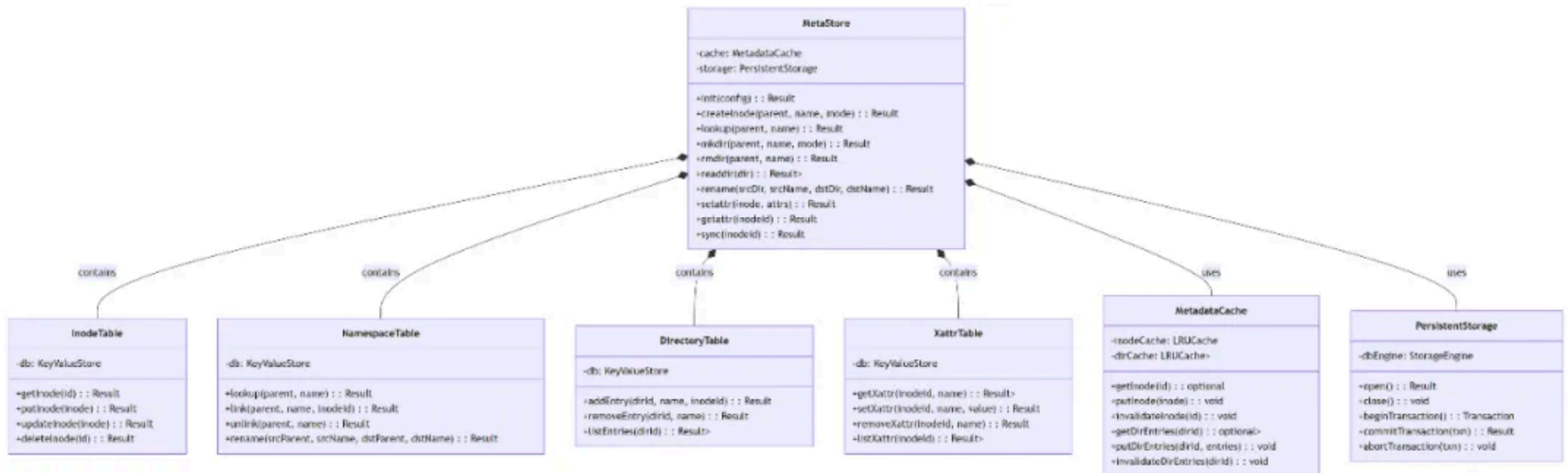
初始化和协调所有元数据服务功能的中央服务器组件。



此图显示了 MetadataServer 的内部结构及其配置。

5.3.2. 元存储

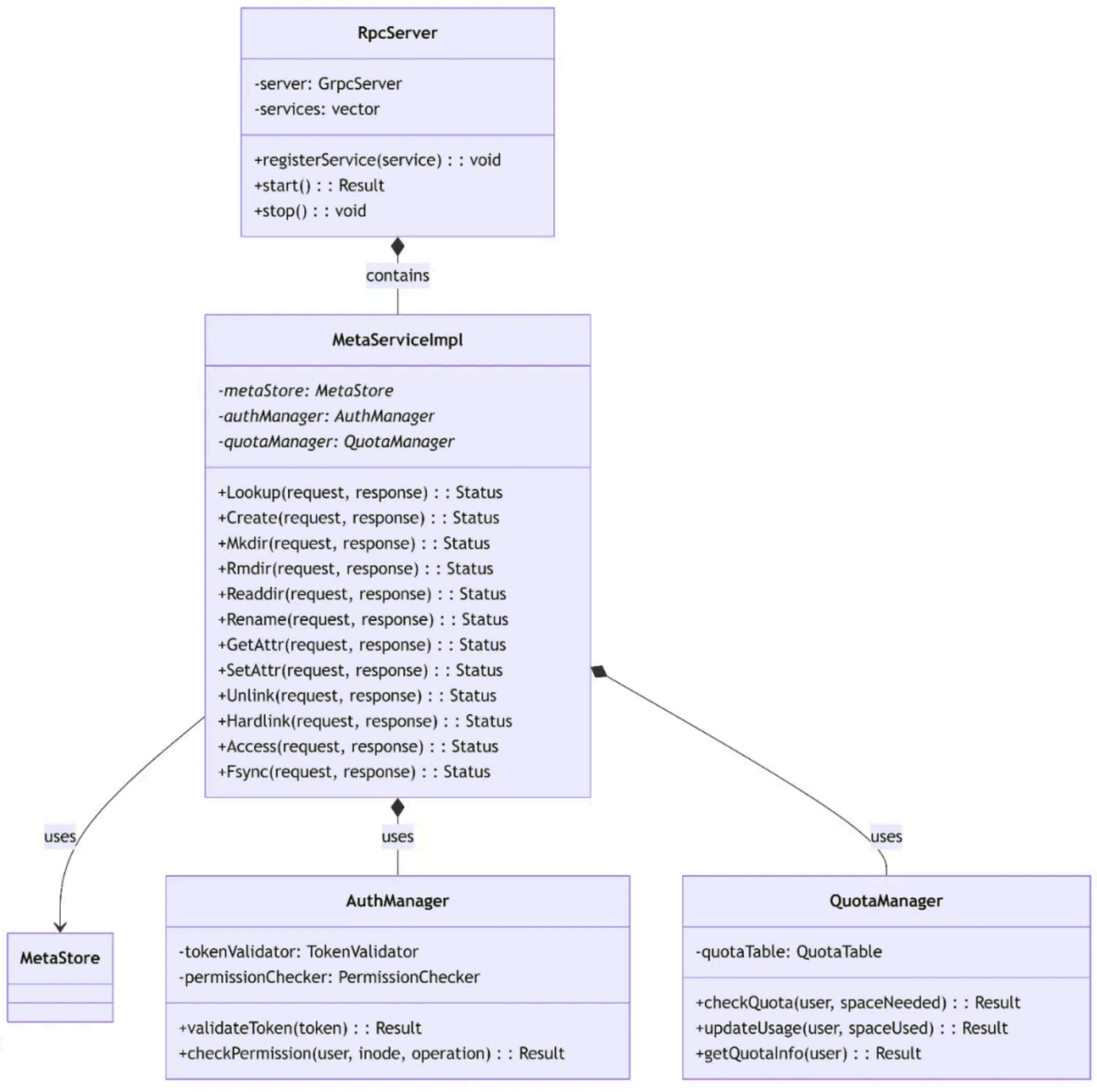
管理 inode、目录条目和命名空间关系的核心元数据存储系统。



此图显示了 MetaStore 及其子组件的详细结构。

5.3.3. RpcServer 和 MetaServiceImpl

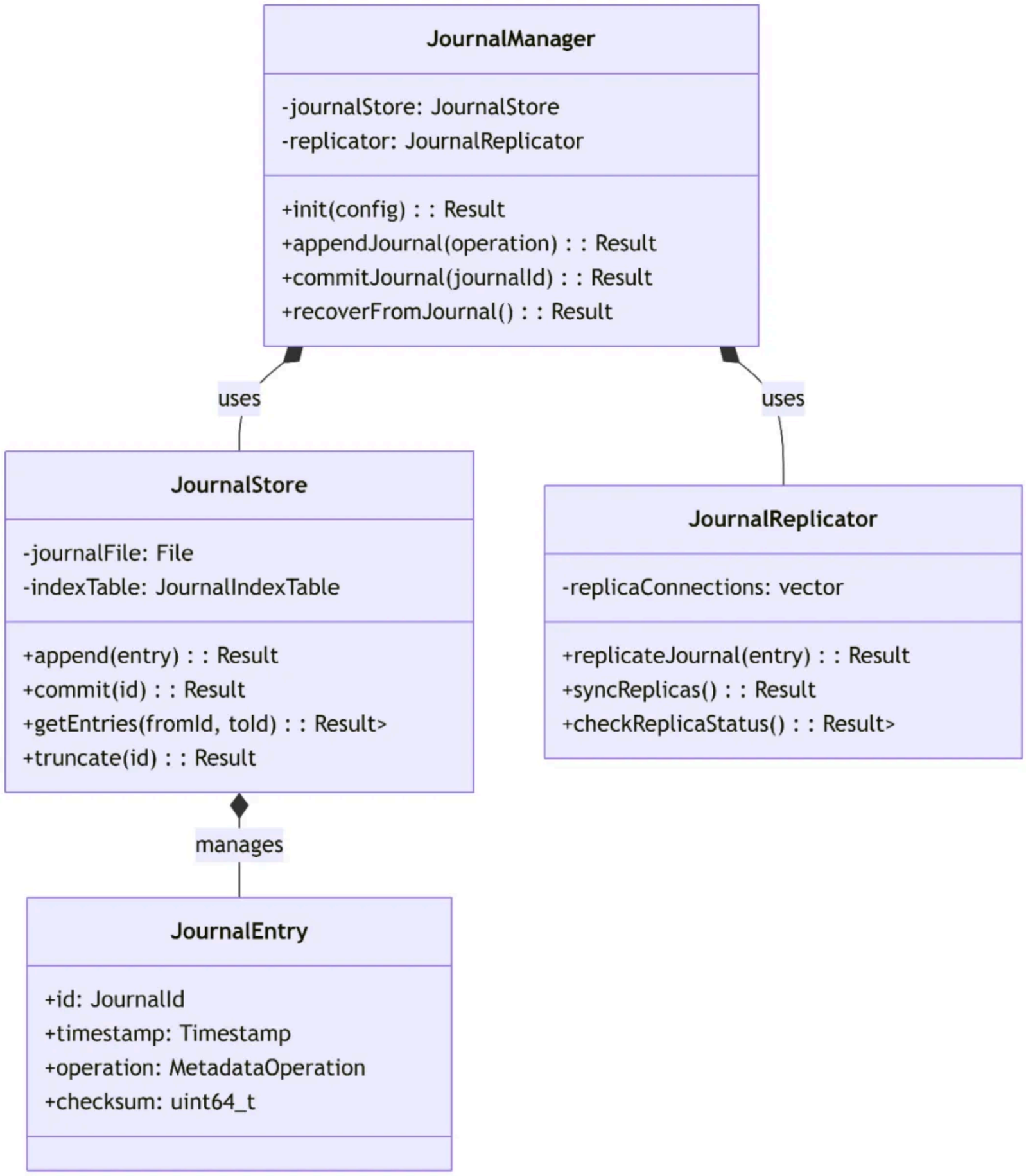
处理客户端请求并将其转换为元数据操作的 RPC 接口。



此图显示了 RPC 服务器组件及其与身份验证和配额管理的关系。

5.3.4. 日志管理器

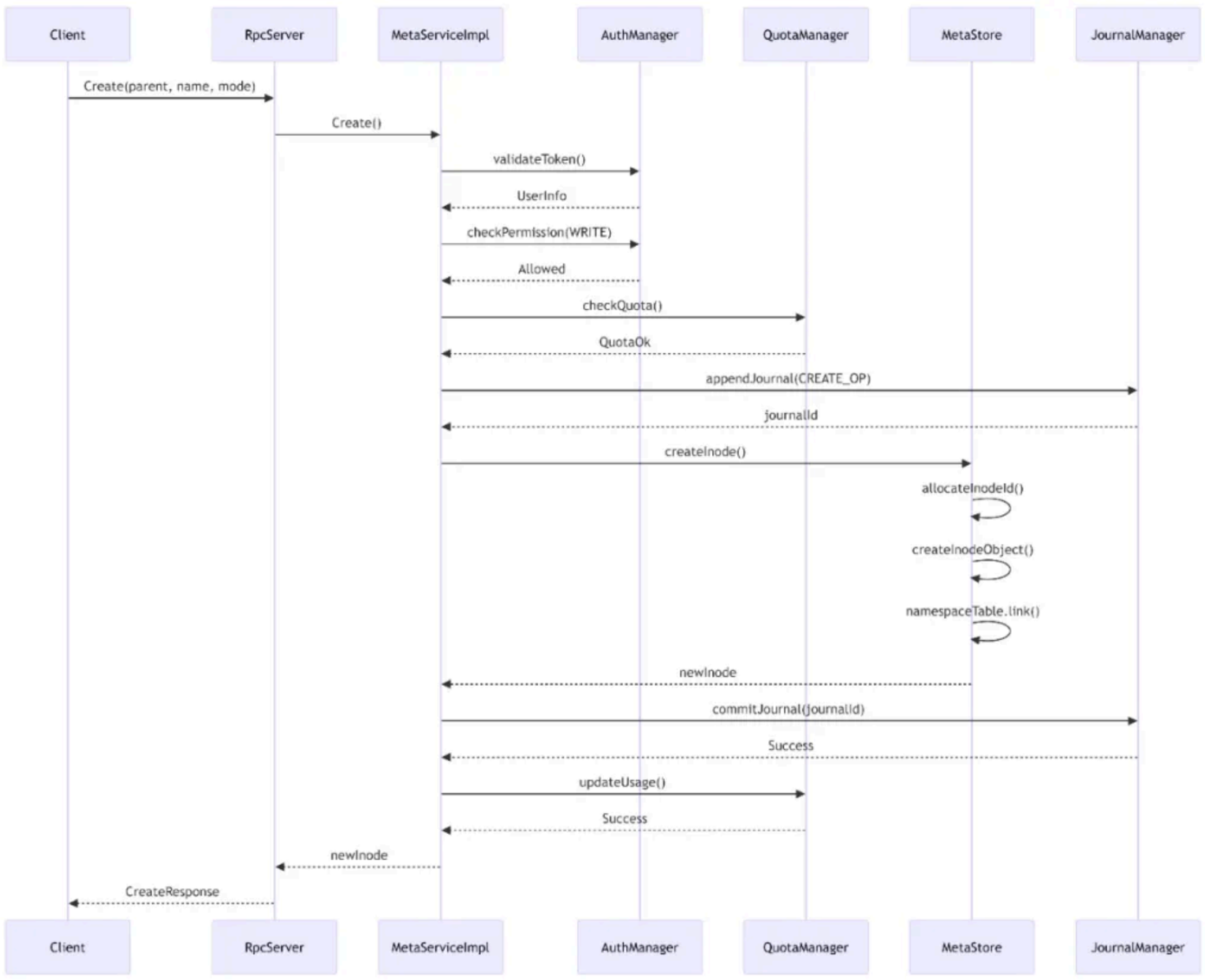
管理元数据操作的预写日志和恢复。



此图示说明了预写日志的日志管理系统。

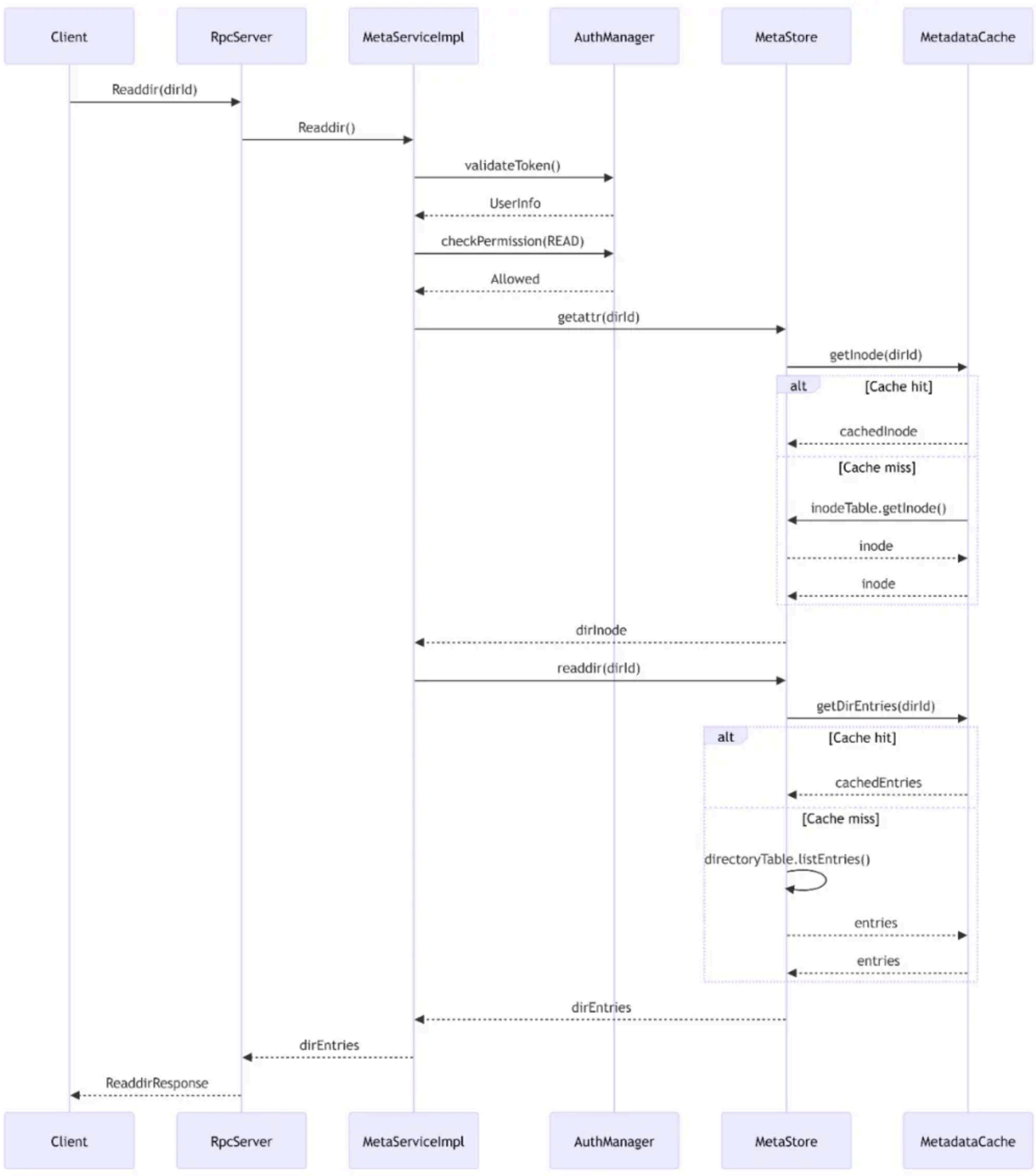
5.4. 主要工作流程

5.4.1. 文件创建工作流程



此图显示了创建新文件的工作流程，包括权限检查、日志记录和配额管理。

5.4.2. 目录列出工作流程



此图说明了与缓存交互的目录列出工作流程。

5.4.3. 重命名操作工作流程

此图显示了重命名文件或目录的工作流程，包括命名空间管理。

5.5. 关键特性

5.5.1. 层次命名空间管理

元数据服务提供了一个完整的层次命名空间，包括目录、文件和链接。NamespaceTable 将（父目录 ID，名称）对映射到 inode ID，而 DirectoryTable 维护每个目录中的条目列表。

5.5.2. 缓存和性能优化

MetadataCache 组件缓存频繁访问的 inode 和目录列表，以减少数据库查找并提高性能。它实现了 LRU（最近最少使用）驱逐策略来管理内存使用。

5.5.3. 日志记录和崩溃恢复

日志管理器实现了预写日志，以确保在崩溃情况下可以恢复元数据操作。所有修改元数据的操作首先记录在日志中，然后再应用到主数据库。

5.5.4. 访问控制和安全

认证管理器组件验证用户令牌，并根据存储在索引节点属性中的标准 Unix 风格权限执行权限检查。它支持基于用户和组的访问控制。

5.5.5. 配额管理

配额管理器跟踪并强制执行每个用户或组的存储配额。它维护使用统计信息，并防止超出分配配额的操作。

5.5.6. 分布式协调

对于多节点部署，元数据服务包括协调、领导者选举和复制的机制，以确保集群的高可用性和一致性。

5.5.7. 可扩展属性支持

XattrTable 提供了文件和目录的扩展属性（xattrs）的存储和检索，支持超出标准文件属性的自定义元数据。

5.6. 结论

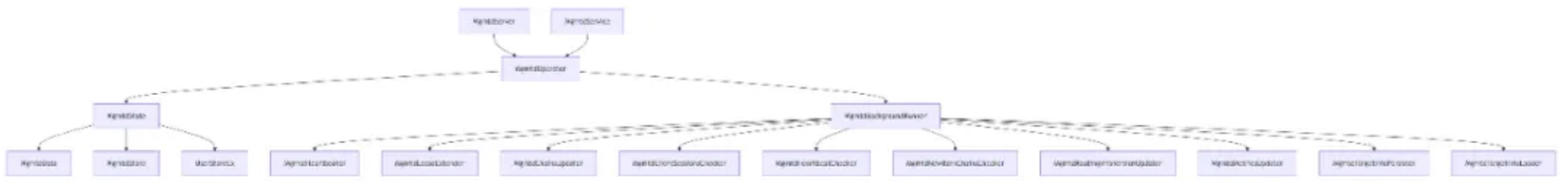
3FS 元数据服务提供了一个强大而高效的系统来管理文件系统元数据。其模块化设计将存储、缓存、访问控制和网络之间的关注点分开，允许灵活的部署和扩展选项。日志系统确保数据完整性，即使在系统故障的情况下，而缓存机制则优化了常见操作的性能。

6. 管理守护进程（MGMTD）架构（src/mgmtd）

本文档概述了 3FS 系统中管理守护进程（MGMTD）的架构。MGMTD 作为中央协调和管理服务，处理集群配置、节点注册、心跳和路由信息。

6.1. 主要组件概述

MGMTD 服务由多个核心组件组成，这些组件协同工作以提供集群管理功能。



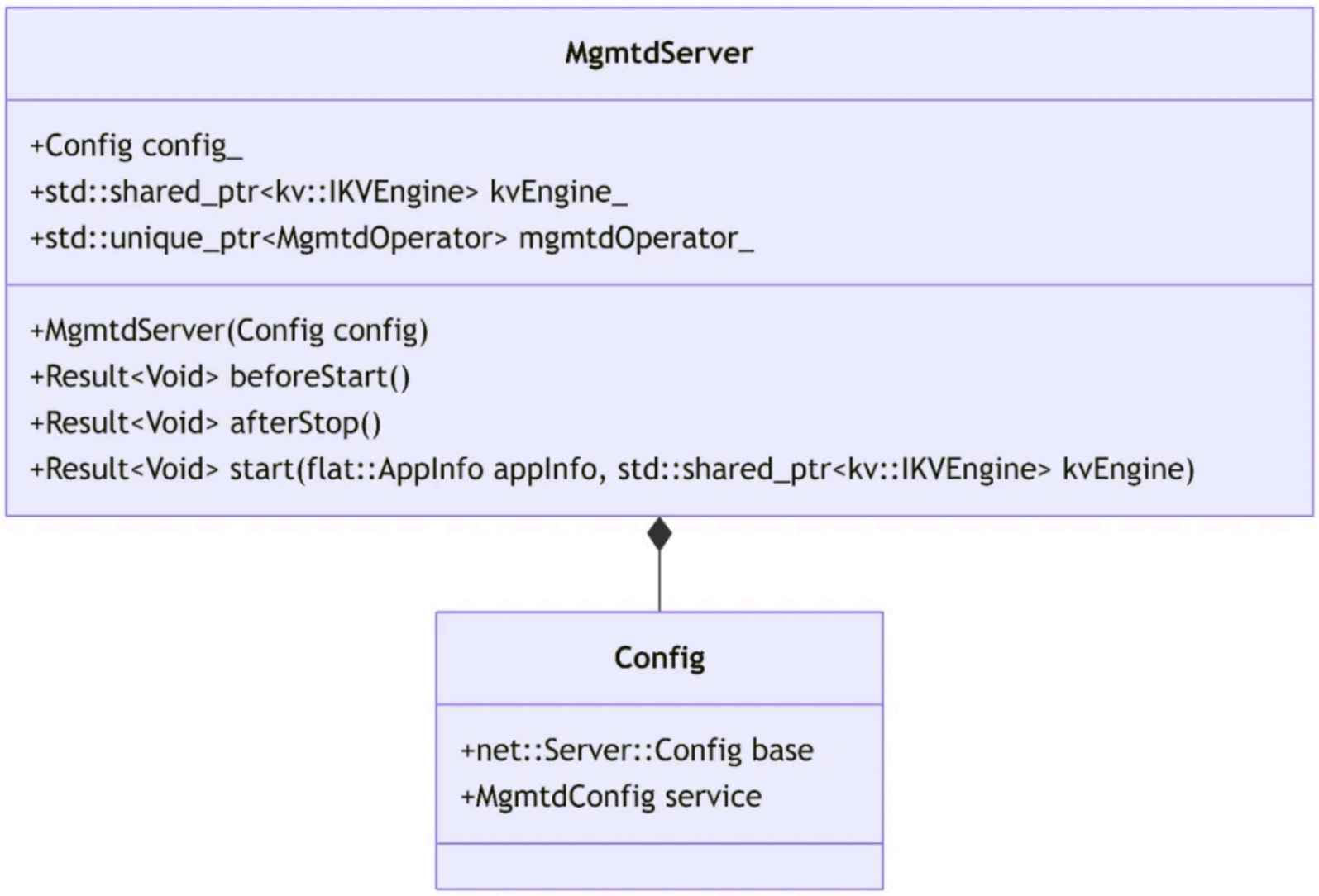
关键组件：

- **管理守护进程服务器**
  : MGMTD 服务的主要入口点
- **MgmtdOperator**
  : 处理服务接口暴露的操作
- **MgmtdState**
  : 维护集群的当前状态
- **MgmtdData**
  : 存储路由信息、配置和其他集群数据
- **MgmtdBackgroundRunner**
  : 管理后台任务，如心跳和租约延续
- **MgmtdStore**
  : 处理 MGMTD 数据的持久性
- **MgmtdService**
  : 客户端交互的 RPC 服务接口

6.2. 组件结构和职责

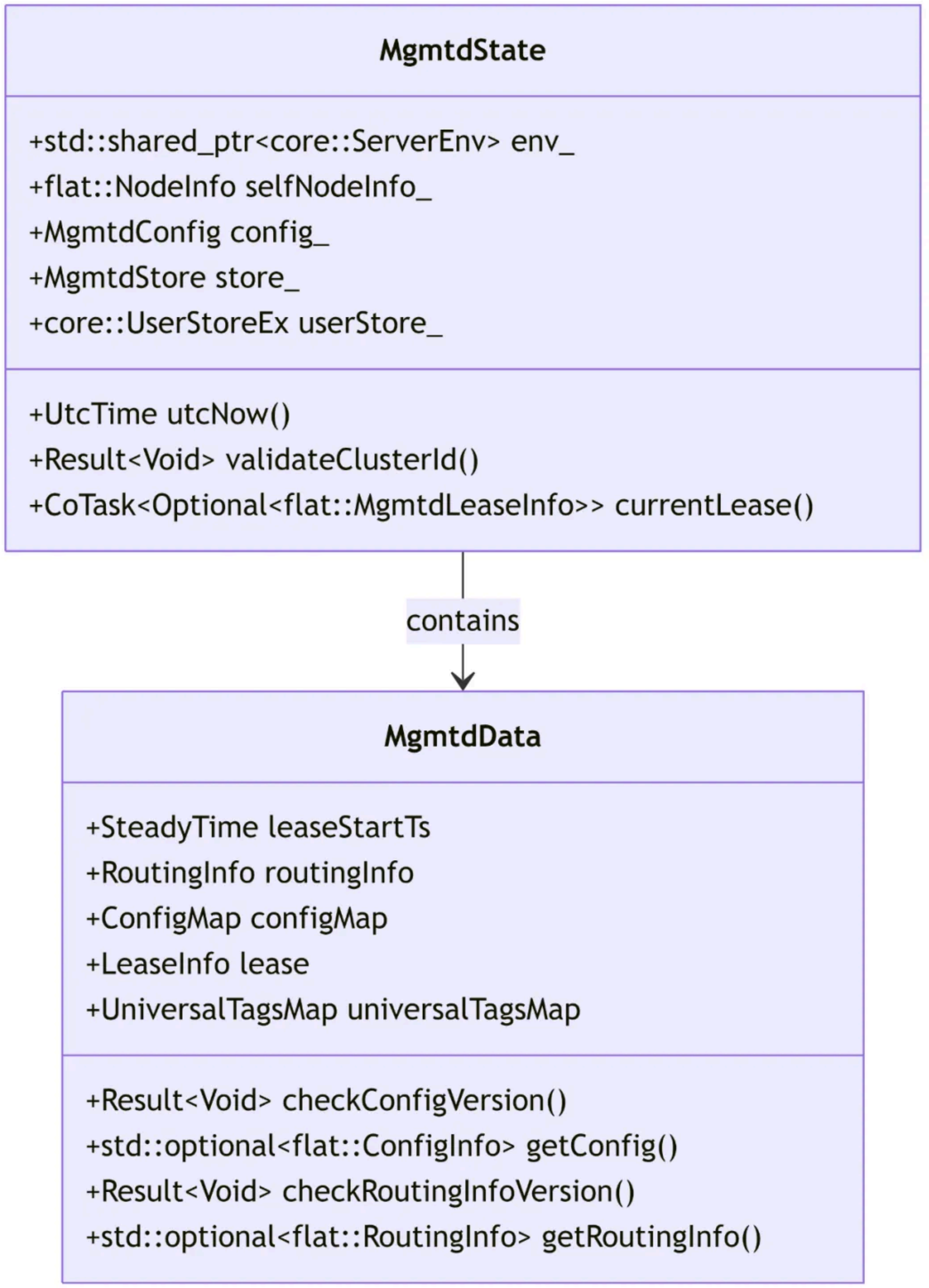
6.2.1. MgmtdServer

MGMTD 服务的入口点，初始化并协调所有组件。



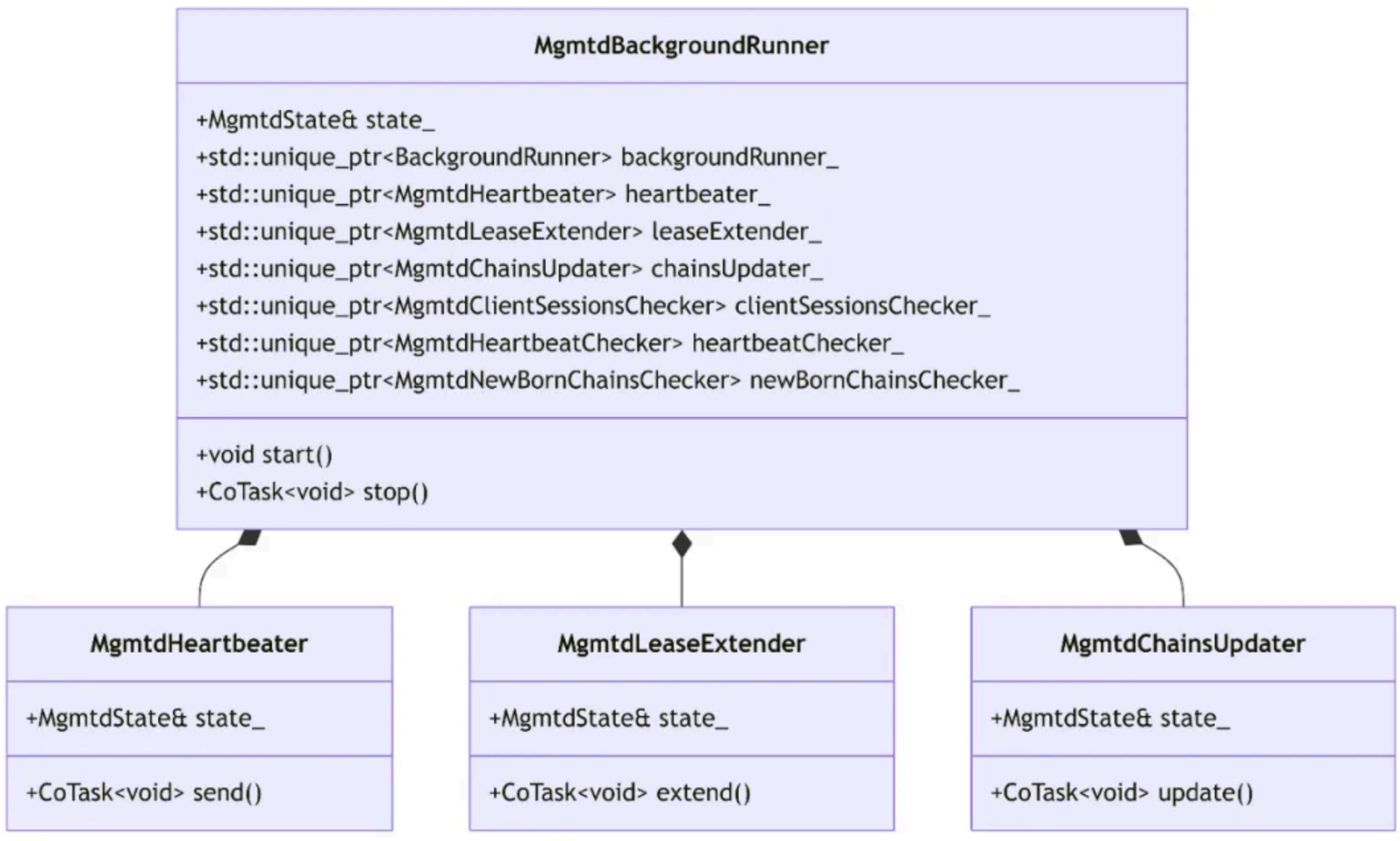
6.2.2. MgmtdState 和 MgmtdData

这些组件维护集群的状态和数据。



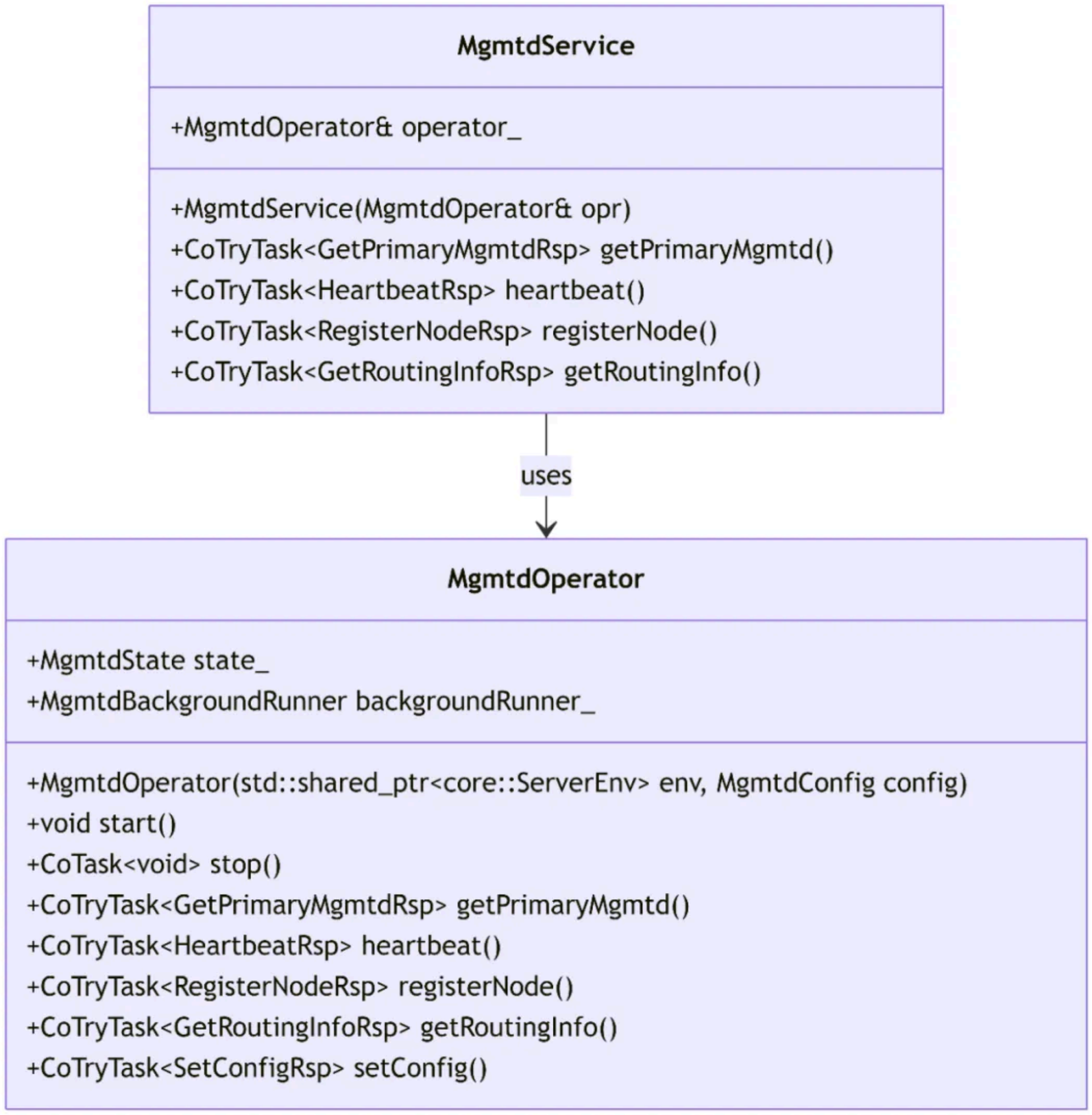
6.2.3. MgmtdBackgroundRunner 和后台服务

管理各种后台进程，以保持集群的运行。



6.2.4. MgmtdOperator 和服务接口

处理 RPC 接口和服务操作。



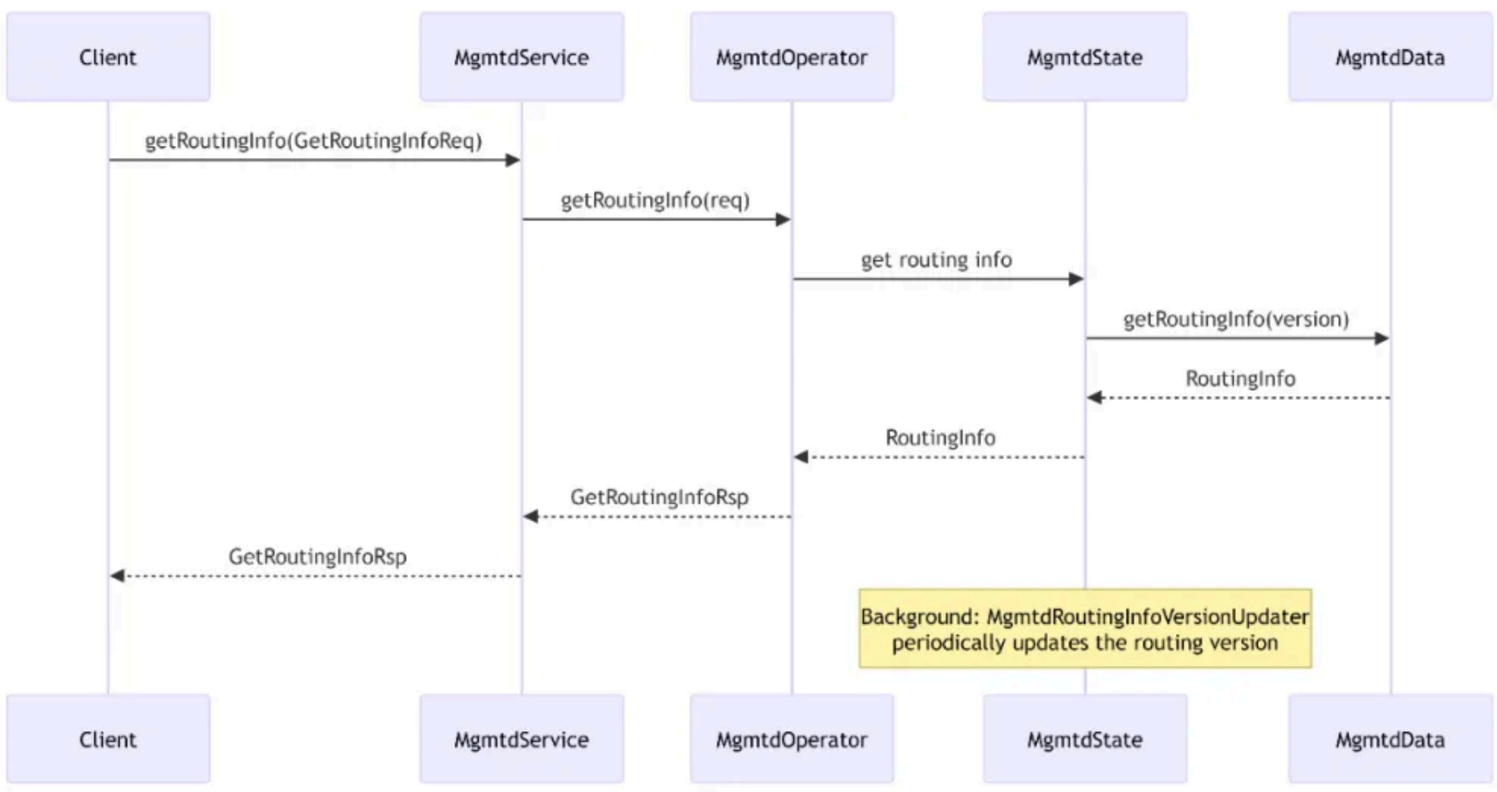
6.3. 关键工作流程

6.3.1. 节点注册和心跳流程

显示节点如何与 MGMTD 注册并通过心跳保持其存在。

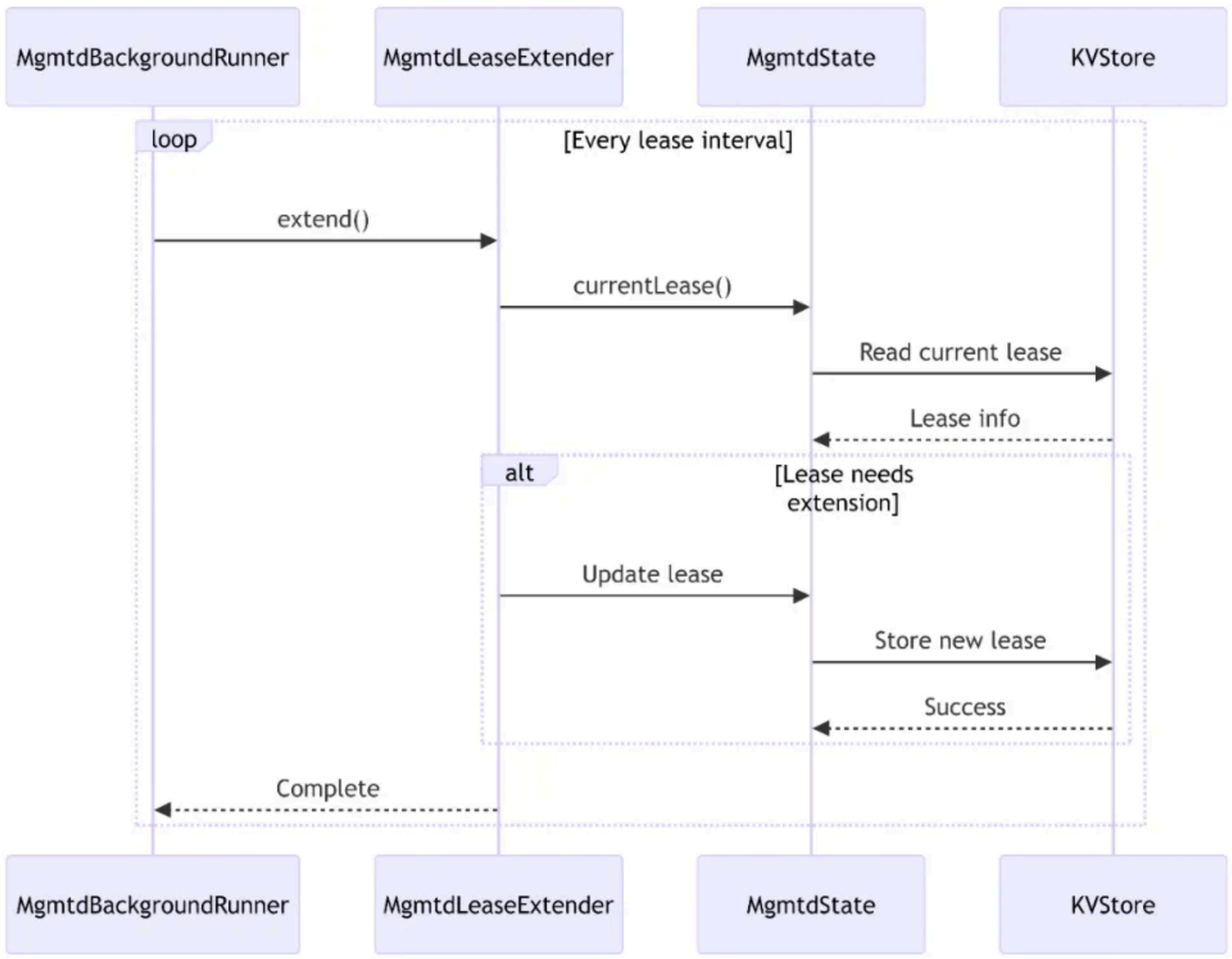
6.3.2. 路由信息分发

说明路由信息如何维护并分发给客户端。

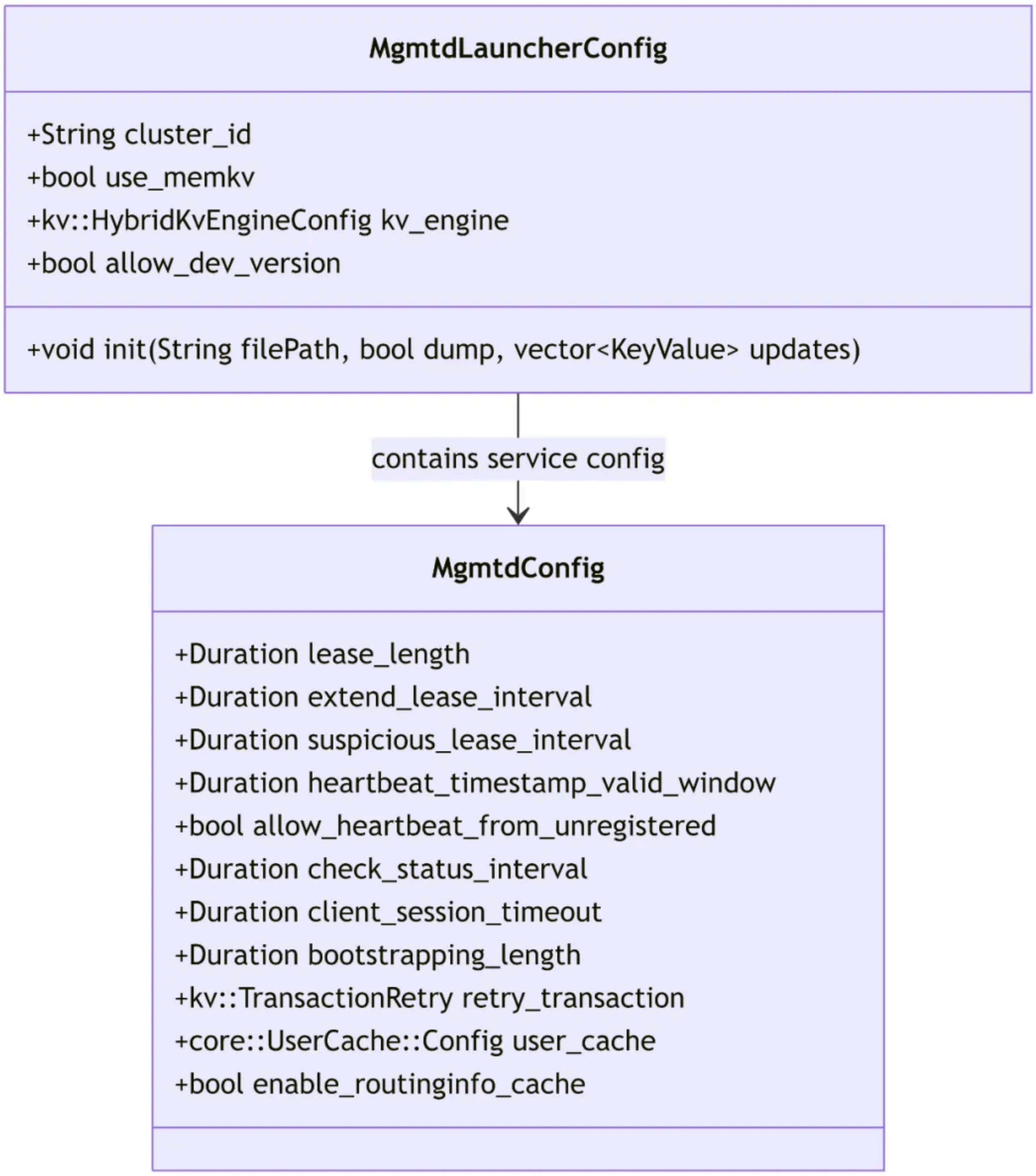


6.3.3. 租赁管理

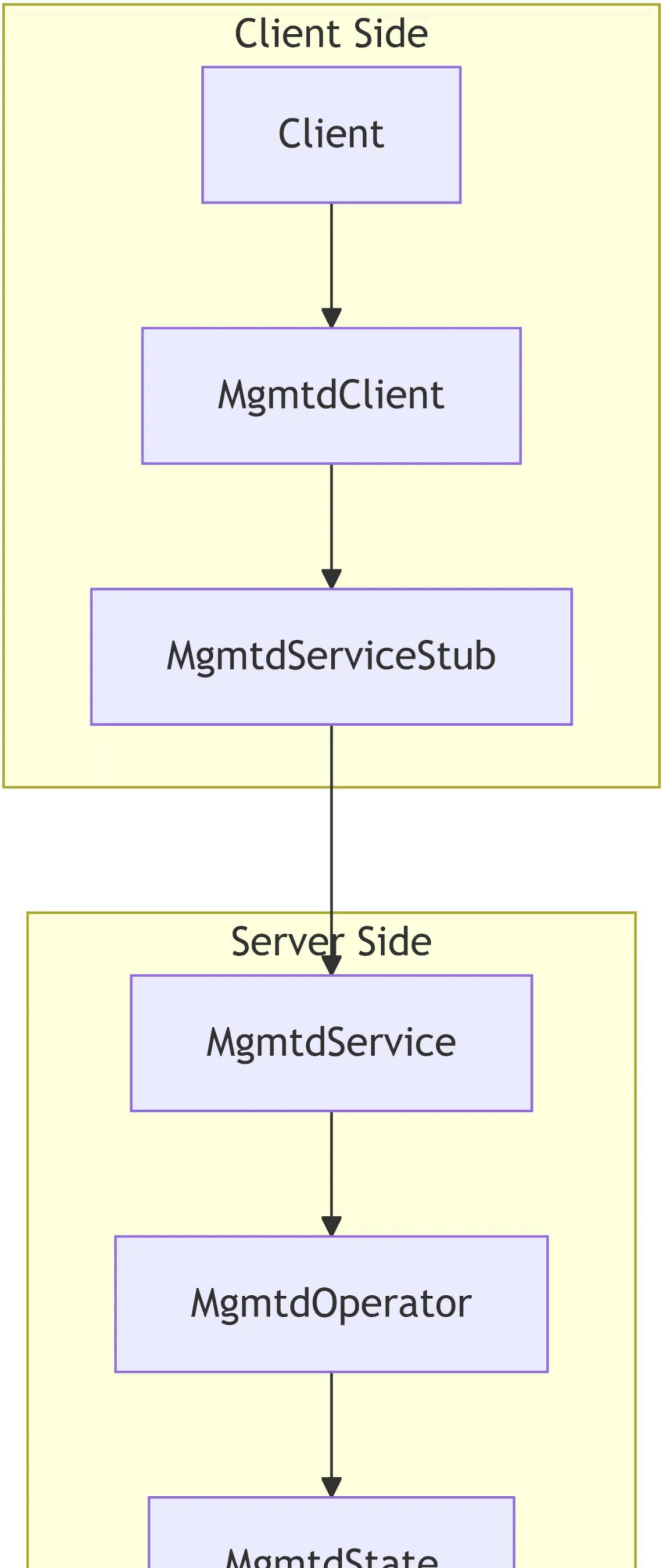
显示 MGMTD 如何管理主 MGMTD 协调的集群租赁。

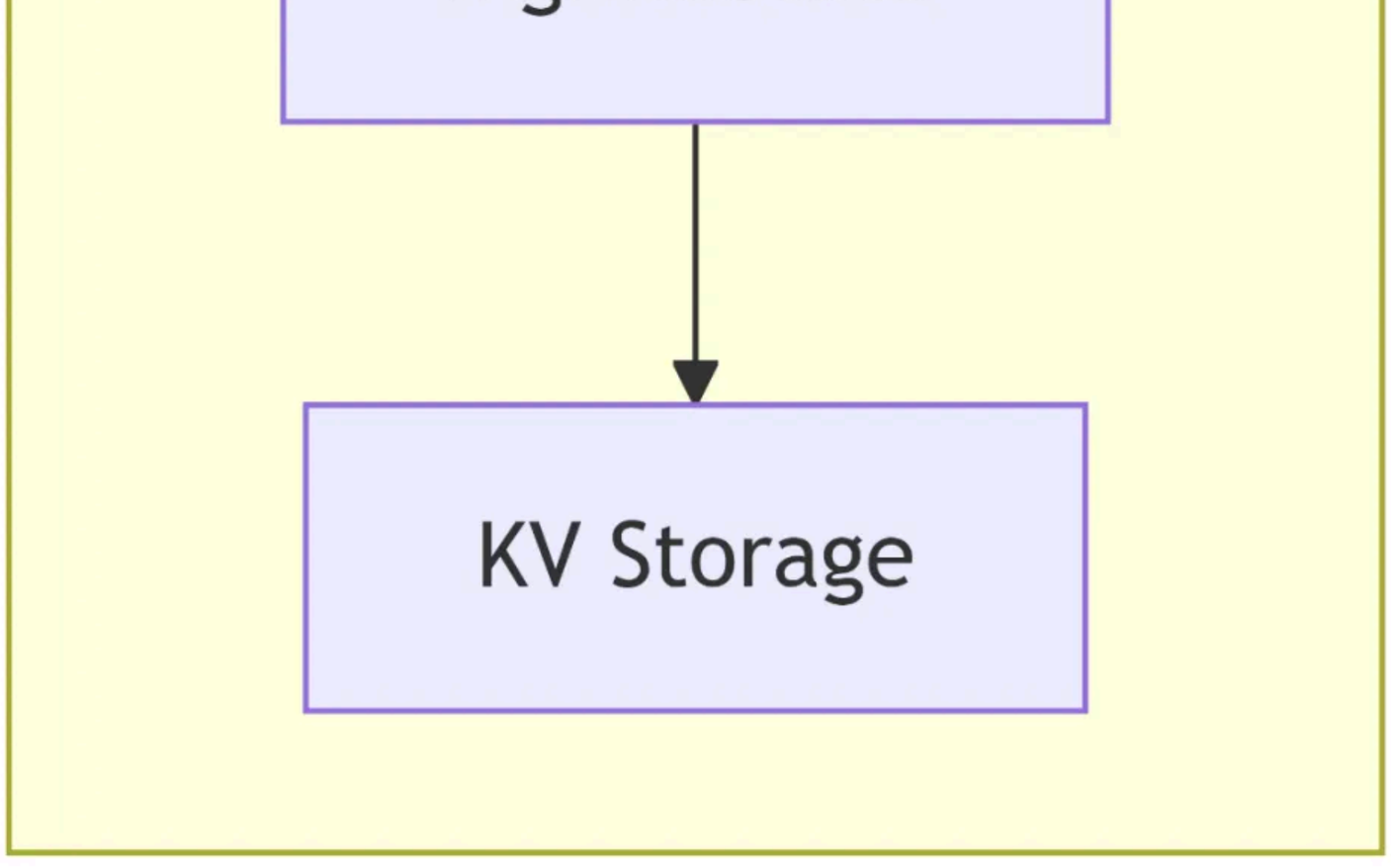


6.4. 配置管理



6.5. 客户端与服务的交互


Client Side
Client
MgmtdClient
MgmtdServiceStub
Server Side
MgmtdService
MgmtdOperator
MgmtdState



6.6. 功能概述

6.6.1. MGMTD 的主要职责

1. 集群成员管理

- 节点注册和跟踪
- 心跳监测
- 节点状态管理（启用/禁用）

2. 配置管理

- 存储和分发节点配置
- 配置的版本控制
- 配置更新和验证

3. 路由信息管理

- 维护集群拓扑
- 分配路由表
- 路由信息的版本控制

4. 租赁管理

- 主管理选举
- 租赁获取和延续
- 故障转移协调

5. 链管理

- 链表配置
- 链的创建、更新和监控
- 目标排序和旋转

6. 客户会话管理

- 会话跟踪
- 会话超时监控
- 会话延长

7. 背景监控与维护

- 心跳检查
- 目标信息持久性
- 链接更新
- 指标收集与报告

MGMTD 服务在 3FS 系统中发挥着核心作用，通过维护集群状态、协调节点，并确保将配置和路由信息正确分发到所有组件。

7. 3FS 存储架构 (src/storage)

本文档概述了 3FS 分布式文件系统中存储子系统的架构。存储层负责高效地持久化和检索数据块，同时保持数据完整性和高性能。

7.1. 概述

3FS 存储子系统提供了一种可靠的高性能分布式存储解决方案。它管理多个存储目标上的数据块，处理并发的读/写操作，并提供数据复制和恢复机制。

7.2. 关键组件

存储子系统由几个关键组件组成，采用分层架构：

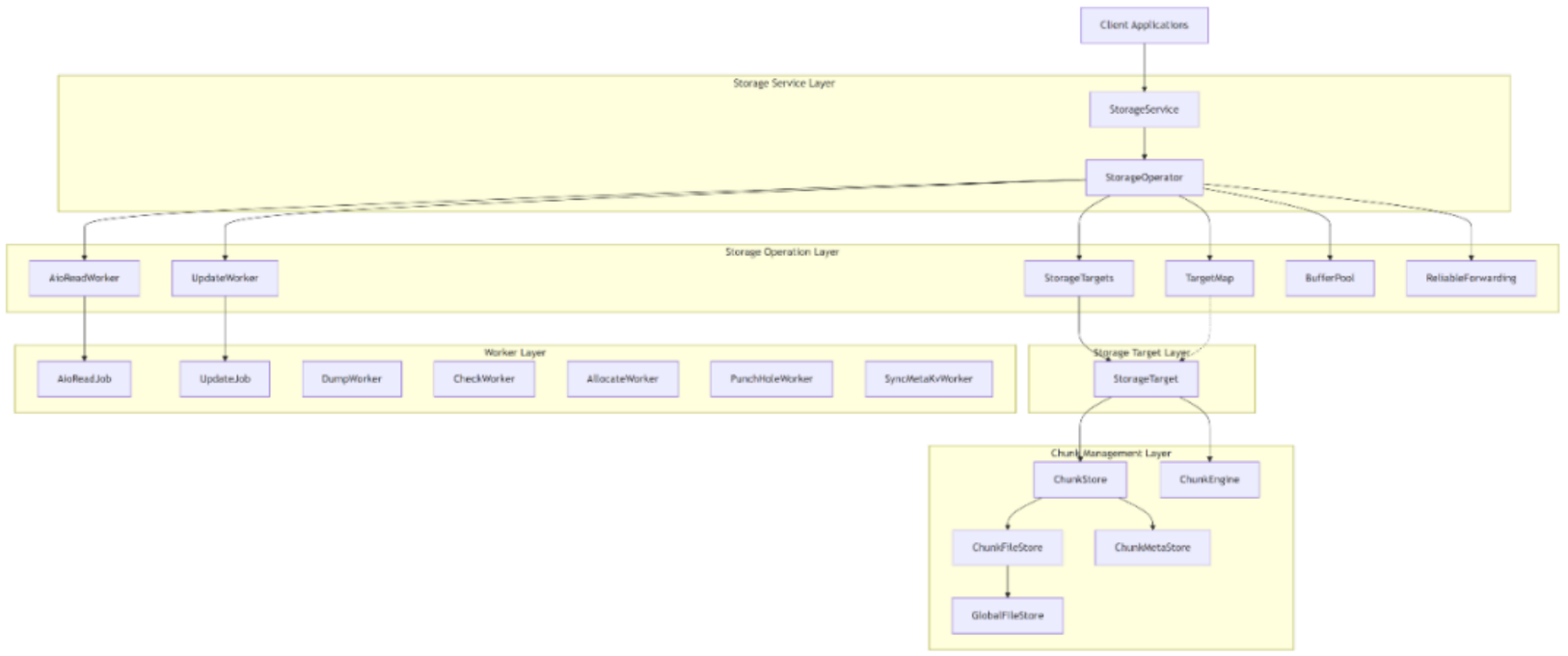
1. **存储服务层**
   - 处理客户端请求的外部接口
2. **存储操作员**
   （StorageOperator） - 协调存储目标之间的操作
3. **目标管理**

（Target Management） - 处理存储目标生命周期和分配

4. **块存储**
   - 管理数据块的物理存储
5. **复制协议**
   - 用于原子可查询复制（CRAQ）的链复制

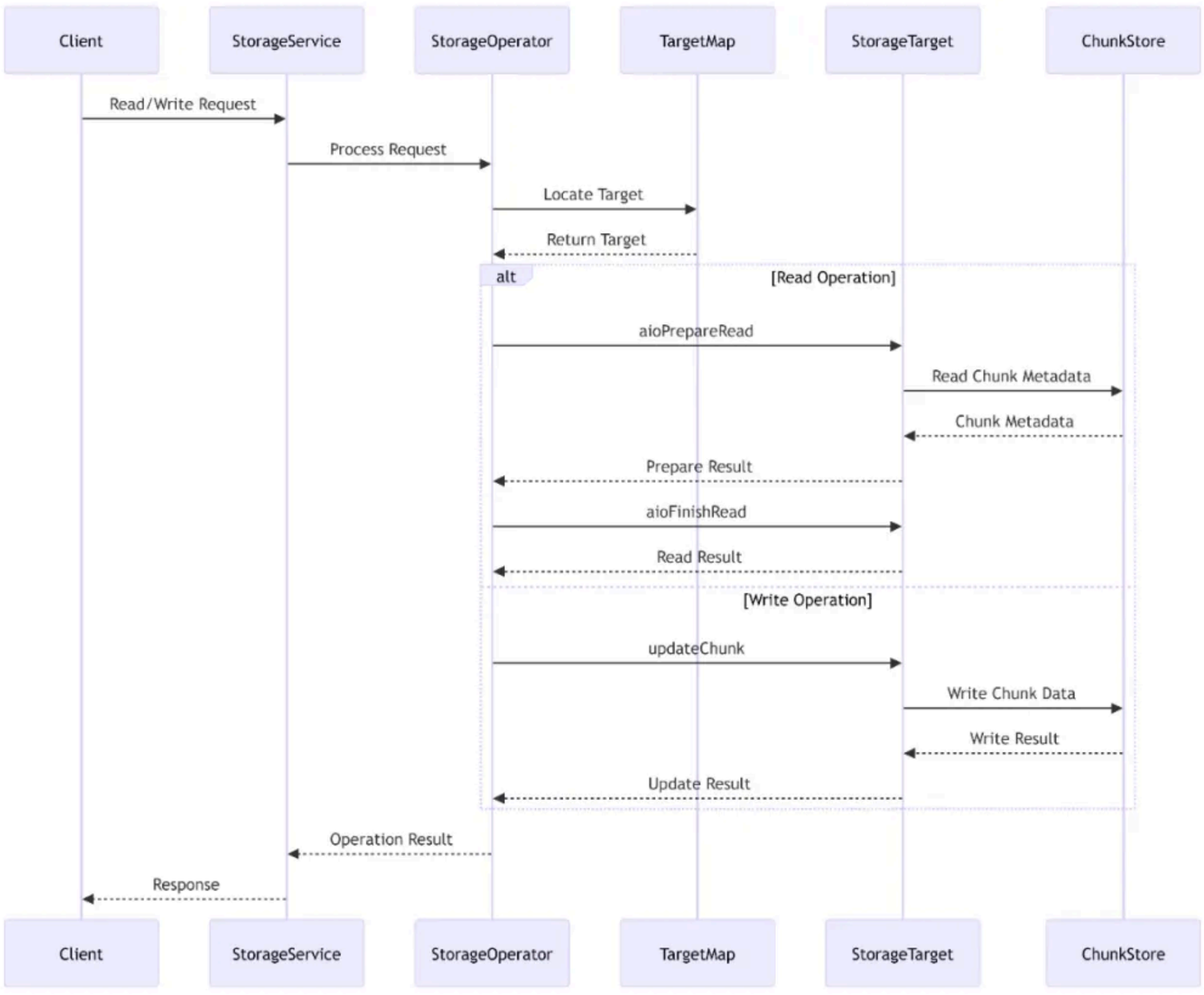
7.3. 架构图

7.3.1. 组件架构



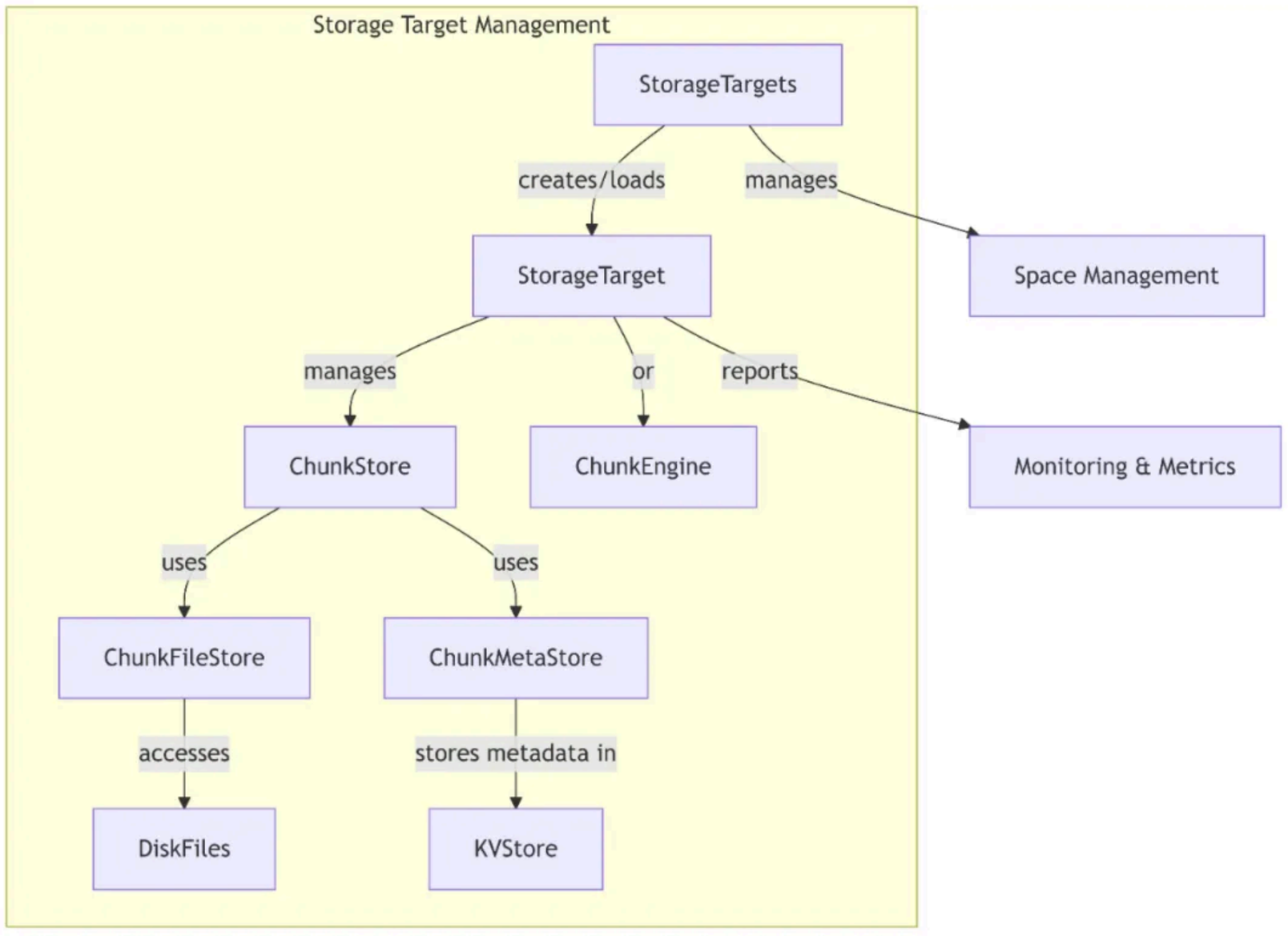
上面的图表显示了存储架构的主要组件及其关系。存储系统分层组织，每一层负责存储功能的不同方面。

7.3.2. 请求处理工作流程



此图说明了读写请求如何在存储系统中流动，展示了请求处理过程中不同组件之间的交互。

7.3.3. 存储目标管理



此图显示了系统中如何管理存储目标，说明了目标与底层存储机制之间的关系。

7.4. 主要组件描述

7.4.1. 存储服务层

- **存储服务**
  : 作为 RPC 服务暴露存储功能，处理客户端对读取、写入、更新和块管理等操作的请求。

7.4.2. 存储操作层

- **存储操作员**
  : 协调存储目标之间的操作，实现读取/写入操作、空间管理和目标协调的核心功能。
- **缓冲池**
  : 管理 I/O 操作的内存缓冲区，以优化内存使用并减少分配。
- **可靠转发**
  : 确保在复制链中的存储节点之间可靠地转发数据。

7.4.3. 存储目标层

- **存储目标**
  : 管理存储目标的集合，处理目标创建、加载和空间信息。
- **目标映射**
  : 维护链 ID 与存储目标的映射，以便高效查找目标。
- **存储目标**
  : 表示一个单独的存储目标，管理特定目标的块操作和元数据。

7.4.4. 块管理层

- **块存储**
  : 管理块数据和元数据，提供读取、写入和查询等操作的接口。
- **块引擎**
  : 使用基于 Rust 的引擎的块管理的替代实现。
- **ChunkFileStore**
  ： 处理文件中块的物理存储。
- **ChunkMetaStore**
  ： 管理键值存储中的块元数据存储。
- **GlobalFileStore**
  ： 提供存储系统中文件描述符的全局视图。

7.4.5. 工作层

- **AioReadWorker**
  : 处理异步 I/O 读取操作。
- **UpdateWorker**
  : 处理块更新操作（写入、删除、截断）。
- **DumpWorker**
  : 执行存储数据的后台转储操作。
- **CheckWorker**
  : 执行验证和健康检查任务。
- **分配工作者**
  : 管理块分配操作。
- **打孔工作者**
  : 通过在稀疏文件中打孔来回收存储空间。
- **同步元数据键值工作者**
  : 在键值存储中同步元数据。

7.5. 关键工作流程

7.5.1. 读取操作

1. 客户端向存储服务发送读取请求
2. 存储服务委托给存储操作员
3. 存储操作员通过目标映射识别目标
4. 存储操作员使用 AioReadWorker 准备读取
5. 存储目标从 ChunkStore 或 ChunkEngine 访问块数据
6. 数据通过服务层返回给客户端

7.5.2. 写操作

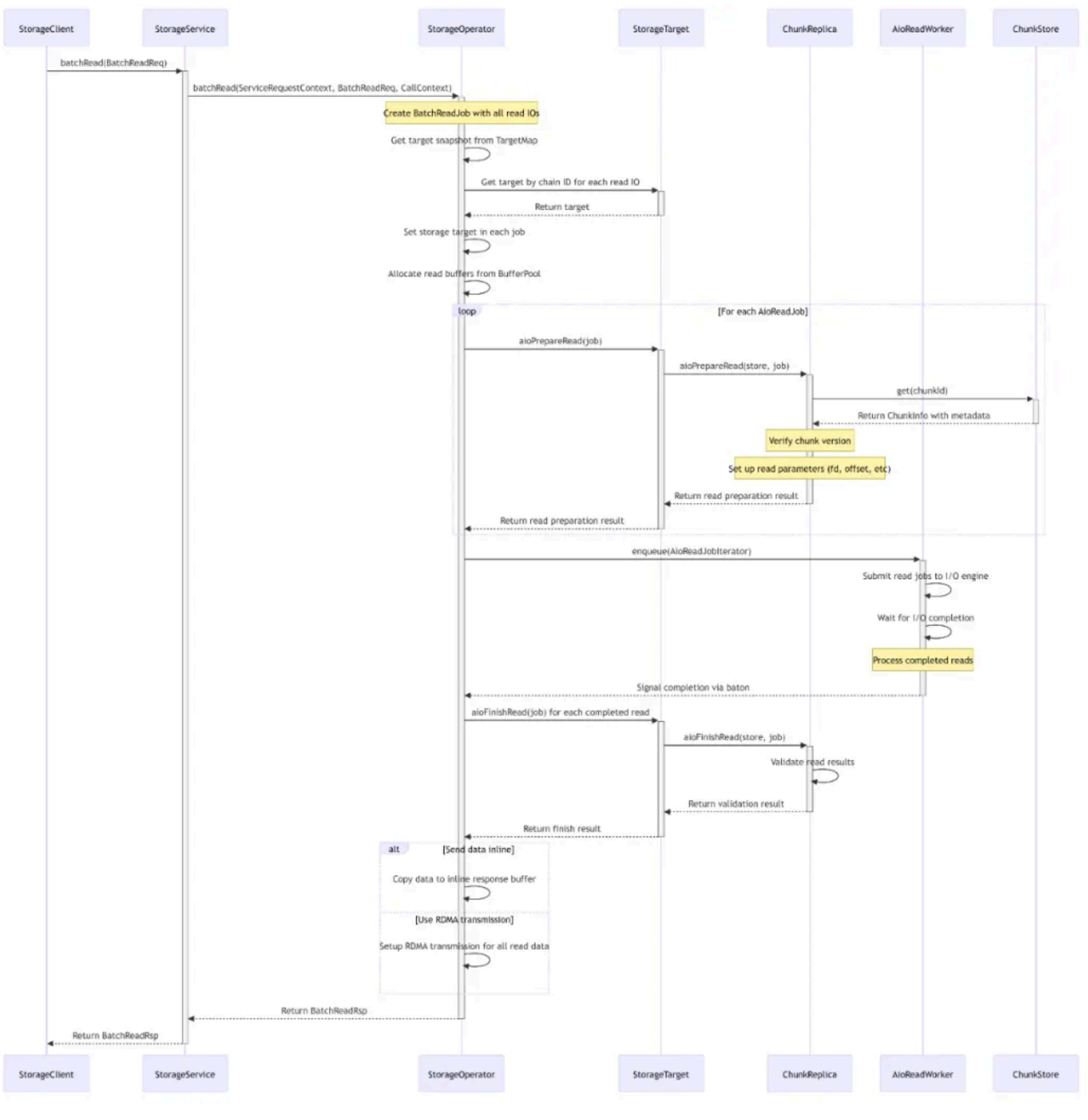
1. 客户向存储服务发送写请求
2. 存储服务委托给存储操作员
3. 存储操作员通过目标映射识别目标
4. 存储操作员启动写操作
5. StorageTarget 使用 ChunkStore 或 ChunkEngine 更新块
6. 可选地通过 ReliableForwarding 转发到下一个副本
7. 结果返回给客户端

7.5.3. 目标管理

1. StorageTargets 根据配置创建或加载目标
2. 每个 StorageTarget 初始化 ChunkStore 或 ChunkEngine
3. StorageTargets 在 TargetMap 中注册目标
4. 收集并维护空间信息以便于管理决策

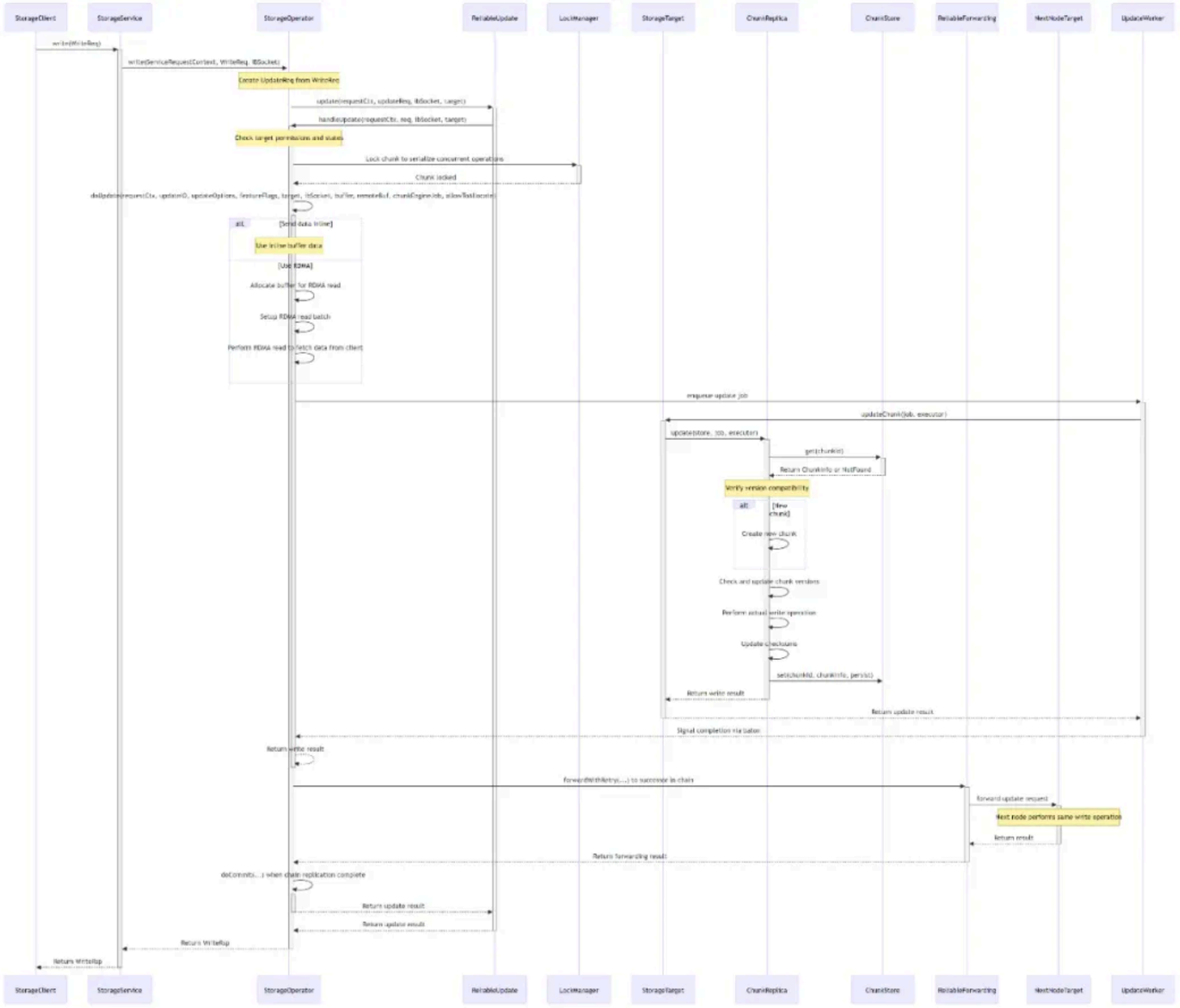
7.6. 读取处理流程

以下序列图说明了在 3FS 存储系统中如何处理读取请求：



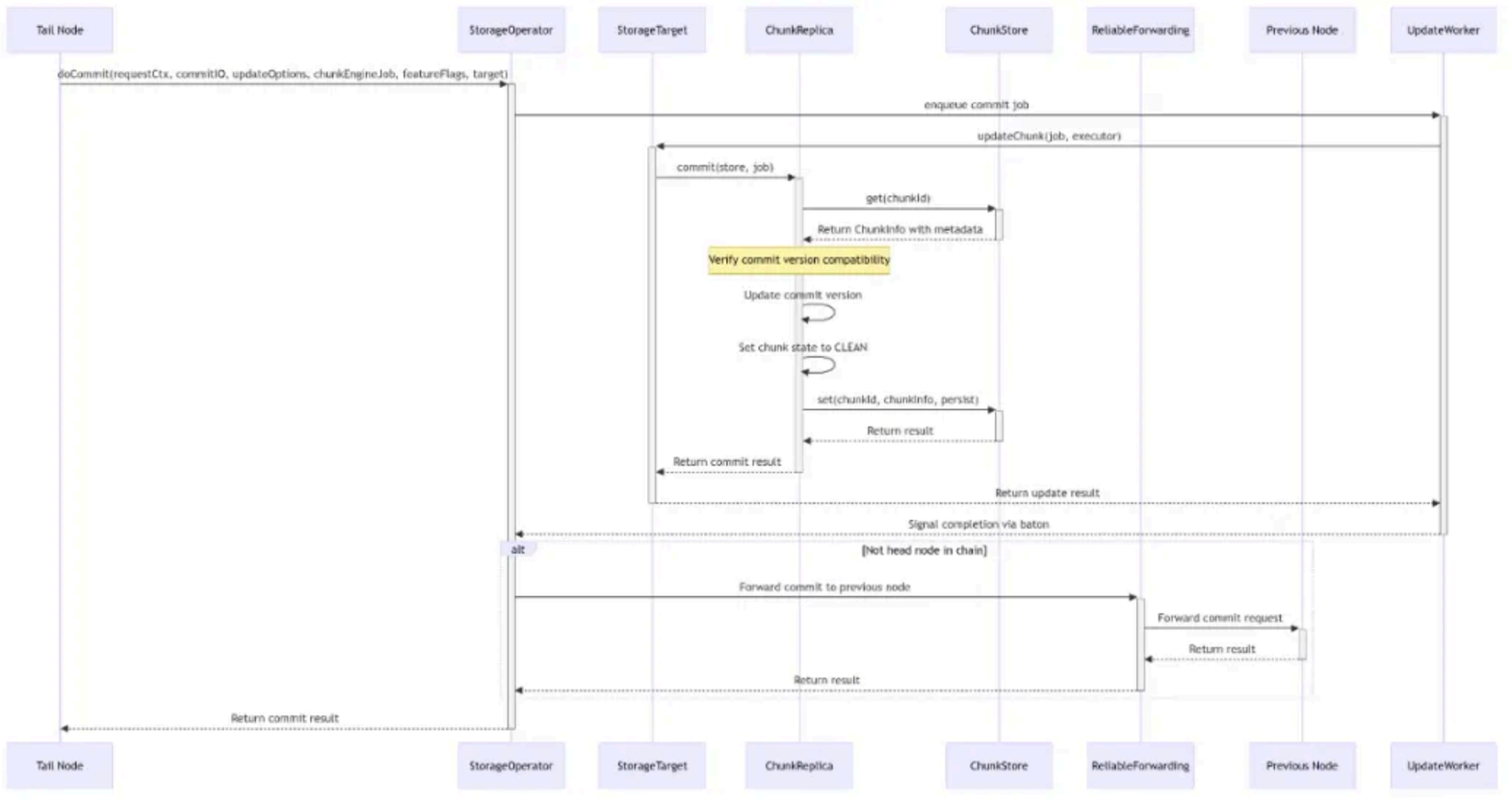
7.7. 写入处理流程

以下序列图说明了在 3FS 存储系统中如何处理写入请求，包括链复制协议：



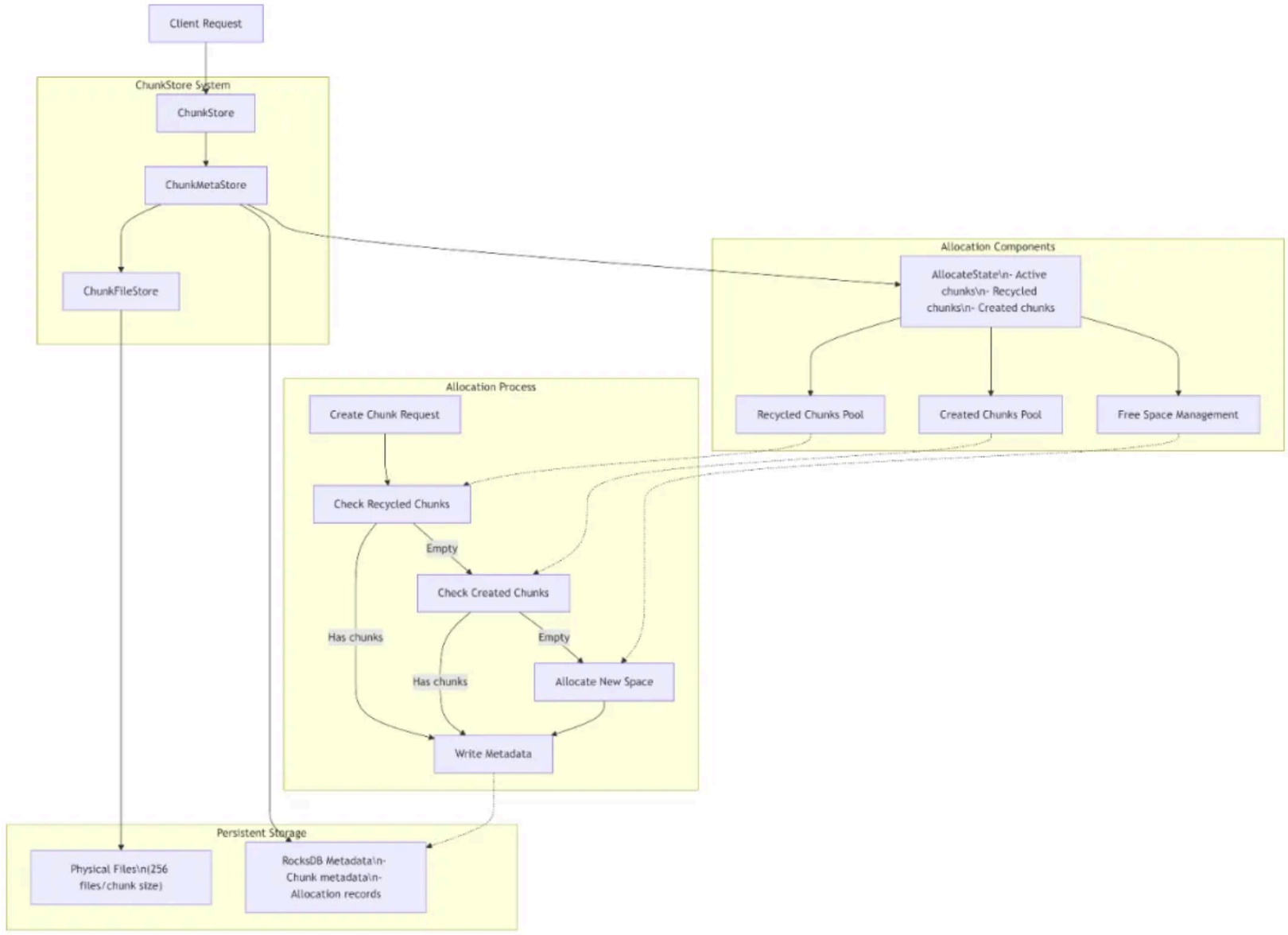
7.8. 提交过程流程

提交操作是 CRAQ 协议的关键部分，确保数据在副本之间持久存储且一致。以下序列图说明了在 3FS 中提交过程的工作原理：



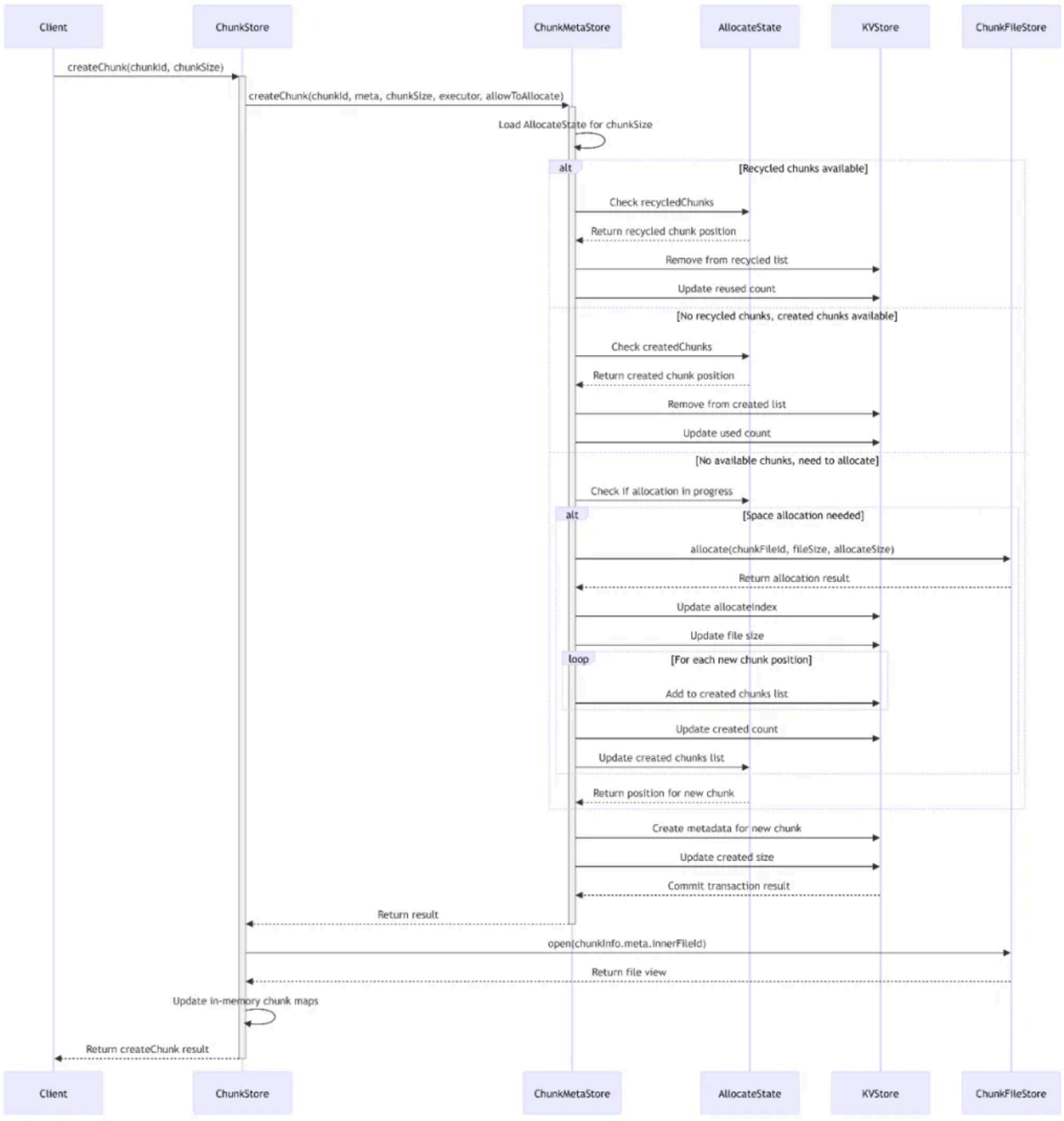
7.9. 块分配管理

块分配系统负责高效管理存储系统中块的生命周期。下面是一个详细的架构图，展示了与块分配相关的组件：



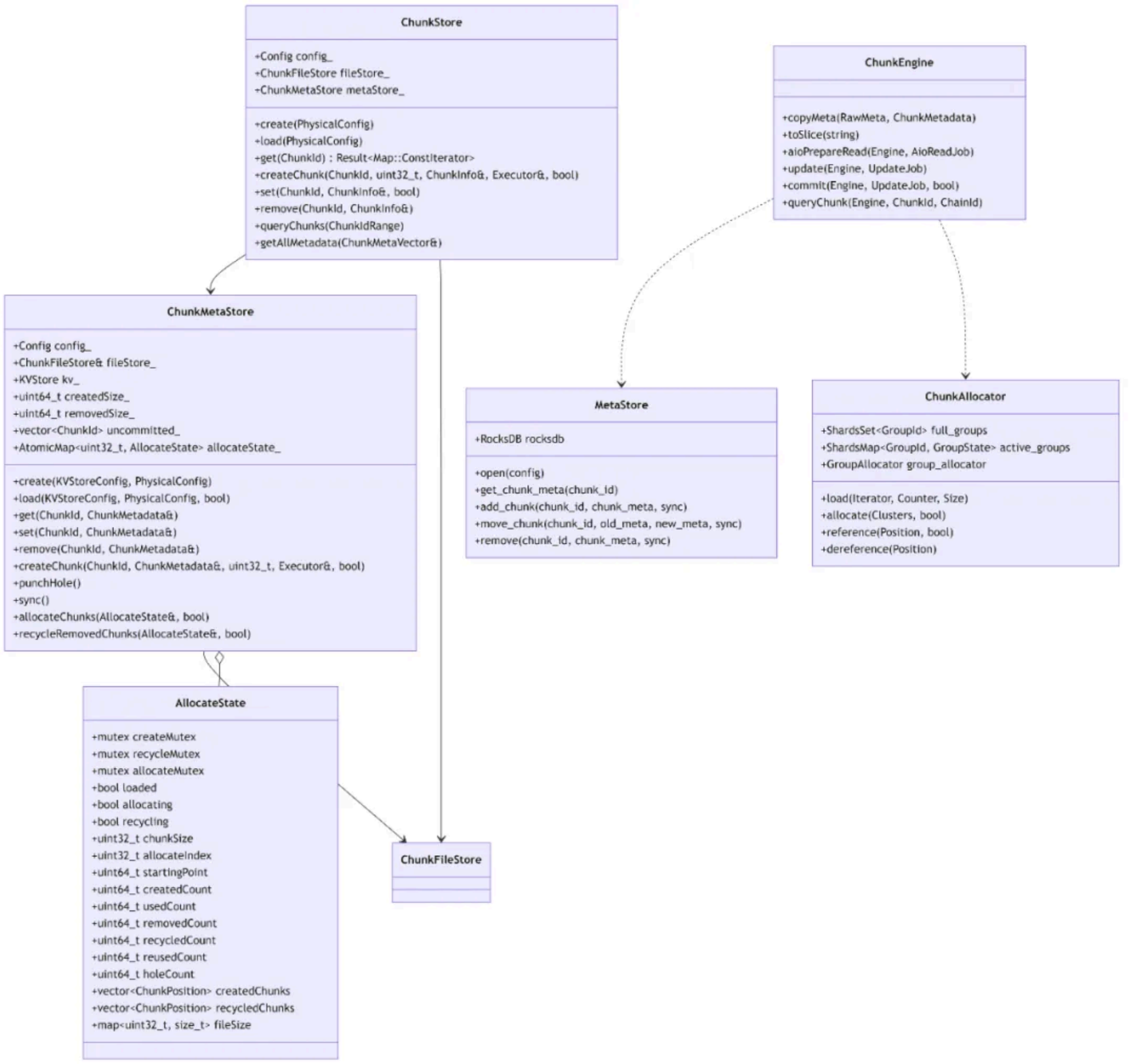
7.9.1. 详细的块分配过程

块分配系统遵循一种复杂的过程，以高效地分配和回收存储空间：



7.10. 元数据管理系统

元数据管理系统负责维护存储系统中所有与块相关的元数据，提供一种可靠和高效的方式来跟踪块的状态、版本和物理位置。

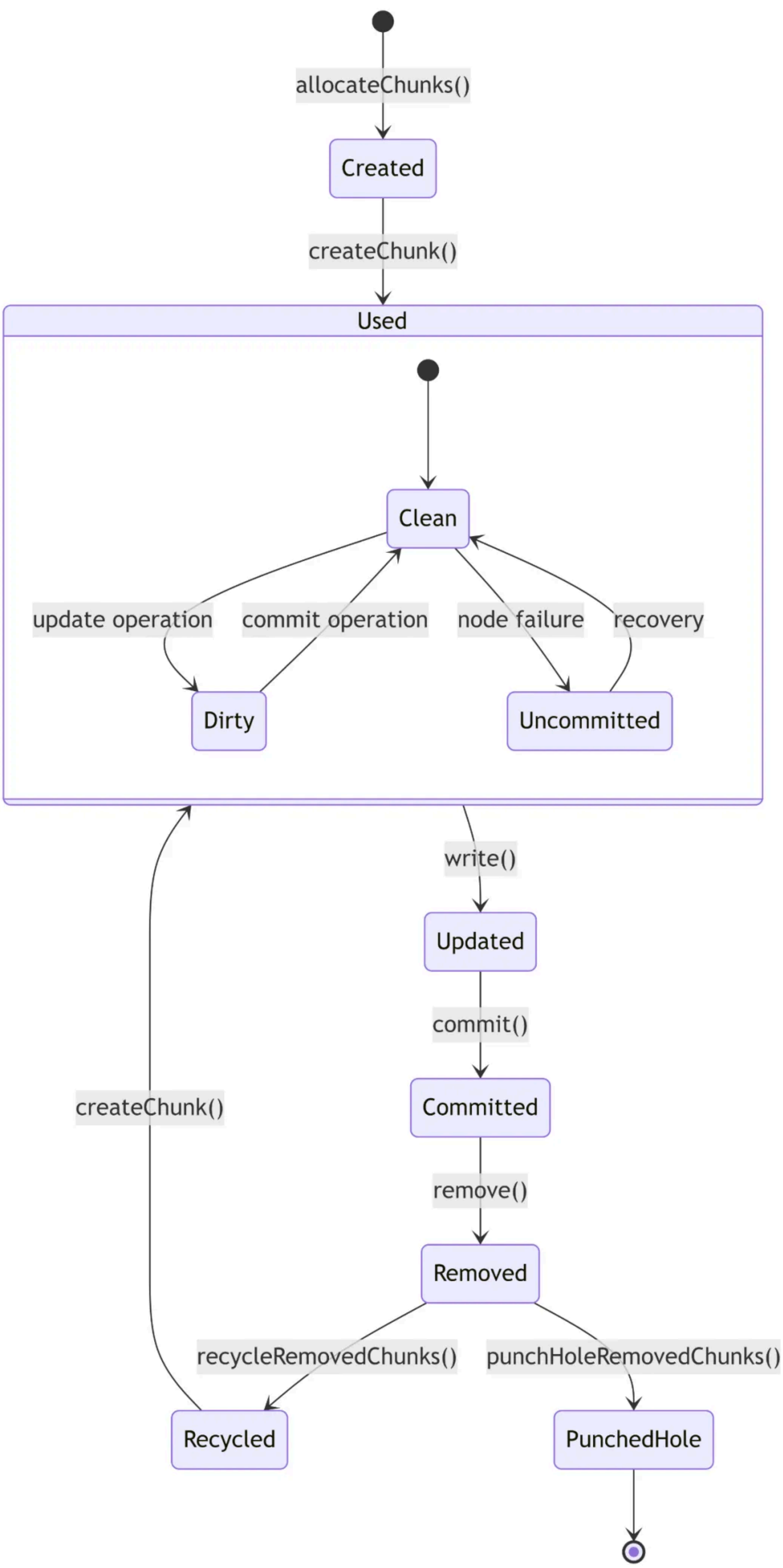


7.10.1. 元数据键结构和组织

系统使用结构化前缀系统在 RocksDB 中组织元数据键，以优化检索和管理：

7.11. 块生命周期管理

下图说明了存储系统中块的完整生命周期：



7.12. 存储系统组件

7.12.1. 存储操作员（StorageOperator）

存储操作员是处理所有存储操作的中央协调组件。它管理：

1. 读/写请求处理

2. 数据传输的缓冲区分配
3. 目标选择与协调
4. RDMA 数据传输
5. 链复制协调

7.12.2. AioReadWorker

AioReadWorker 提供异步 I/O 功能用于读取操作：

1. 管理 I/O 线程池
2. 处理来自存储设备的异步读取
3. 支持传统的 AIO 和 io_uring 接口
4. 高效处理批量读取请求

7.12.3. ChunkReplica

ChunkReplica 处理块的实际存储操作：

1. 阅读准备和验证
2. 带版本控制的写操作
3. 提交操作
4. 校验和验证与更新

7.12.4. 目标管理

存储目标通过以下方式进行管理：

1. 动态添加和移除目标
2. 健康监测和状态管理
3. 副本分发与平衡
4. 链的形成与维护

7.12.5. 可靠转发

ReliableForwarding 组件确保更新通过复制链正确传播：

1. 请求转发到后继节点
2. 重试和故障转移机制
3. 一致性验证
4. 链版本管理

8. 引用和资料

[1] VS Code Copilot Agent: https://github.blog/news-insights/product-news/github-copilot-the-agent-awakens/

[2] DeepSeek 3FS: https://github.com/deepseek-ai/3FS/tree/main/src/storage

(注：本文为个人观点总结，作者工作于微软。)