

一、目标描述

1.1 背景

Raft 协议是一种分布式一致性算法，旨在解决分布式系统中多个节点之间数据一致性的问题。它通过选举领导者来协调日志复制，确保所有节点数据一致。目前市面上的 Raft 协议基本在用户态实现，其执行过程中会引发用户态和核心态的频繁切换，且需经过内核网络协议栈，会产生较大的性能开销。

在当今的云计算和容器化环境中，eBPF (extended Berkeley Packet Filter) 技术的应用日益广泛，特别是在加速网络转发、提高系统性能方面展现出了巨大的潜力。使用 eBPF 加速 raft 集群的现实意义不仅体现在提升网络效率上，还包括增强系统安全性、降低运维成本等多个方面。下面将深入探讨使用 eBPF 加速 raft 集群的现实意义：

1.1.1.提升网络效率

(1) 降低延迟

通过绕过复杂的内核栈，eBPF 能够显著降低网络通信的延迟。

(2) 提高吞吐量

eBPF 通过优化数据路径，提高了网络吞吐量，使得数据传输更加高效。

(3) 减少 CPU 负载

传统的网络栈处理需要占用大量的 CPU 资源，eBPF 通过简化数据处理流程，减轻了

CPU 的负担。

1.1.2.增强系统安全性

(1) 内核级别的安全防护

eBPF 运行在内核空间，可以实施更为严格和细粒度的安全策略。

(2) 动态安全监控

利用 eBPF 可以实现实时的网络安全监控，及时发现并阻止潜在的攻击行为。

(3) 减少安全漏洞

eBPF 程序的执行是受限的，这减少了因程序错误导致的安全漏洞的风险。

1.1.3.降低运维成本

(1) 简化网络配置

eBPF 可以自动分析和优化网络通信，减少了手动配置网络的复杂性和出错率。

(2) 自动化故障排除

eBPF 程序可以自动检测和修复一些常见的网络问题，减少了运维人员的工作量。

(3) 持续的性能优化

eBPF 程序可以根据系统的实际运行情况动态调整，实现持续的性能优化。

1.1.4.提升应用性能

(1) 支持高并发

eBPF 的效率优势使得服务器能够支持更高的并发连接数,满足大规模用户访问的需求。

(2) 优化资源分配

eBPF 可以帮助更合理地分配系统资源,确保关键应用的性能需求得到满足。

1.1.5.提高资源利用率

(1) 更高效的数据处理

eBPF 通过优化数据路径,提高了数据处理的效率,从而提升了资源利用率。

(2) 减少能耗

降低 CPU 负载和网络延迟不仅提升了性能,也有助于减少整个系统的能耗。

(3) 优化硬件使用

eBPF 可以使硬件资源得到更有效的利用,延长硬件的使用寿命。

1.2 赛题分析

本次作品要求要在集群中实现 eBPF 技术，以加速 raft 集群的运行。参考顶级会议论文《“Electrode: Accelerating Distributed Protocols with eBPF”. Symposium on Networked Systems Design and Implementation》中 eBPF 对 paxos 协议的修改，本项目使用现有的 eBPF 技术对现有的 raft 开源程序 dragonboat 进行修改。

二、相关调研

2.1 Raft 协议基本原理

Raft 协议是一种分布式一致性算法（共识算法），共识就是多个节点对某一个事件达成一致的计算，即使出现部分节点故障，网络延时等情况，也不影响各节点，进而提高系统的整体可用性。

2.1.1 复制状态机

Raft 算法基于复制状态机 Replicated State Machine 模型，本质上就是一个管理日志复制的算法。客户端向服务端集群发送请求，服务端集群中每个节点都是由复制状态机（State Machine）、日志（Log）和一致性模块（Consensus Module）三个组件构成的。复制状态机负责处理节点的状态，日志则用于记录节点的操作历史，由日志条目（Entry）组成；一致性模块则确保各个节点之间的数据一致性。三者关系如图 1 所示：

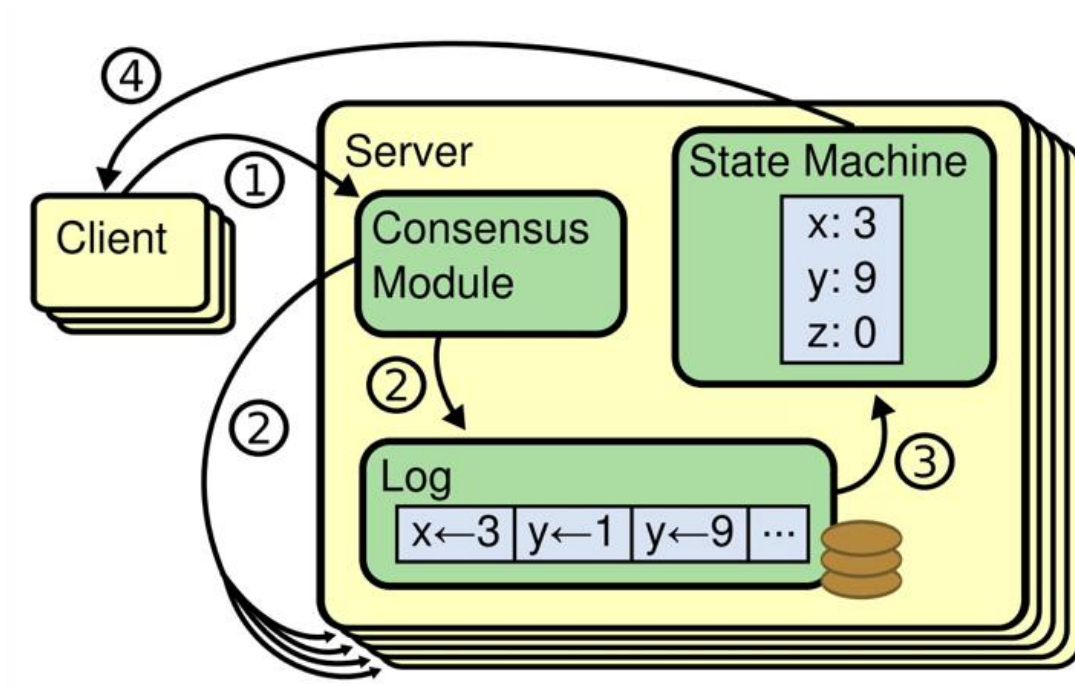


图 1 节点组成

2.1.2 角色

Raft 将系统中的角色分为领导者(Leader)、跟从者(Follower)和候选者(Candidate)。

1. Leader

发送心跳、管理日志复制与提交。接受客户端请求，并向 Follower 同步请求日志。

2. Follower

接受并持久化 Leader 同步的日志，在 Leader 告知日志可以提交后，提交日志。当 Leader 出现故障时，主动推荐自己为候选人。

3. Candidate

Leader 选举过程中的临时角色。向其他节点发送请求投票信息，如果获得大多数选票，则晋升为 Leader。

其基本的转换关系如图 2 所示：

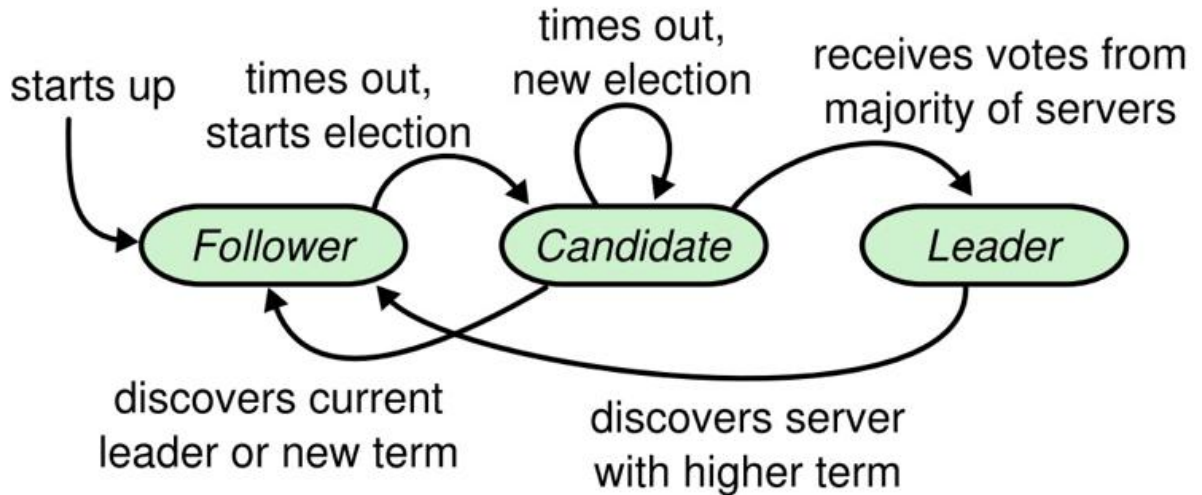


图 2 角色转换关系图

2.1.3 任期

出于可用性考虑，当前 Leader 下线后，集群需要从存活的节点中挑选一个新的 Leader，这个过程被称为选举 election。每次选举都会产生一个新的任期号 term（单调递增），如果选举中产生了一个新的 Leader，那么这个任期号会伴随这个 Leader 直到其下线。

每个 Follower 进程都会维护一个 `current_term` 用于表示已知的最新任期，进程之间通过彼此交换该值来感知 Leader 变化。

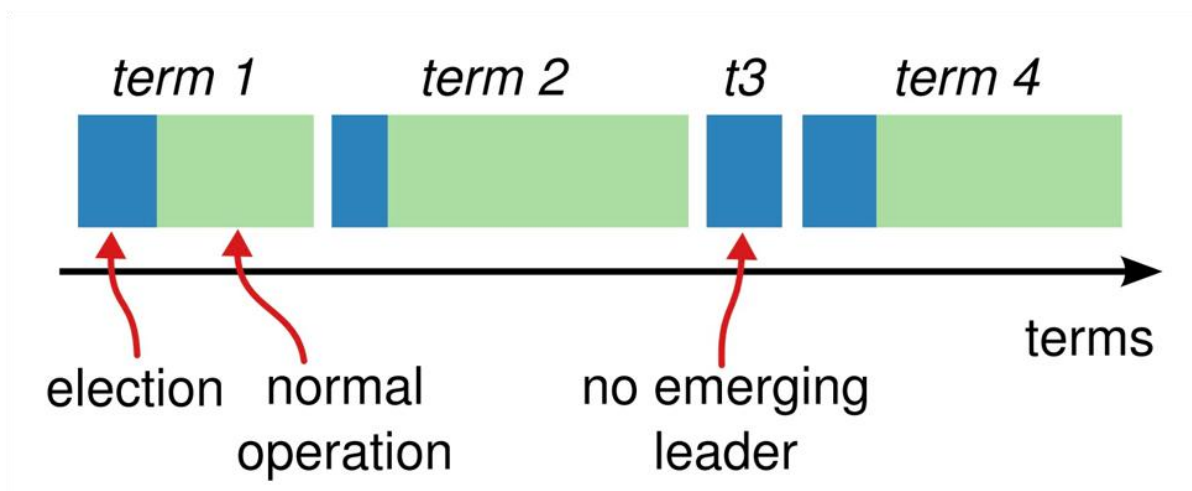


图 3 任期演变

2.1.4 日志

日志是 Raft 的核心概念。Raft 保证日志是连续且一致的，并且最终能够被所有进程按照日志索引的顺序提交。每条日志记录包含：

- 任期 term：生成该条记录的 Leader 对应的任期
- 索引 index：其在日志中的顺序
- 命令 command：可执行的状态机指令

一旦某条日志中的命令被状态机执行了，那么我们称这条记录为已提交 committed，

Raft 保证已提交的记录不会丢失。

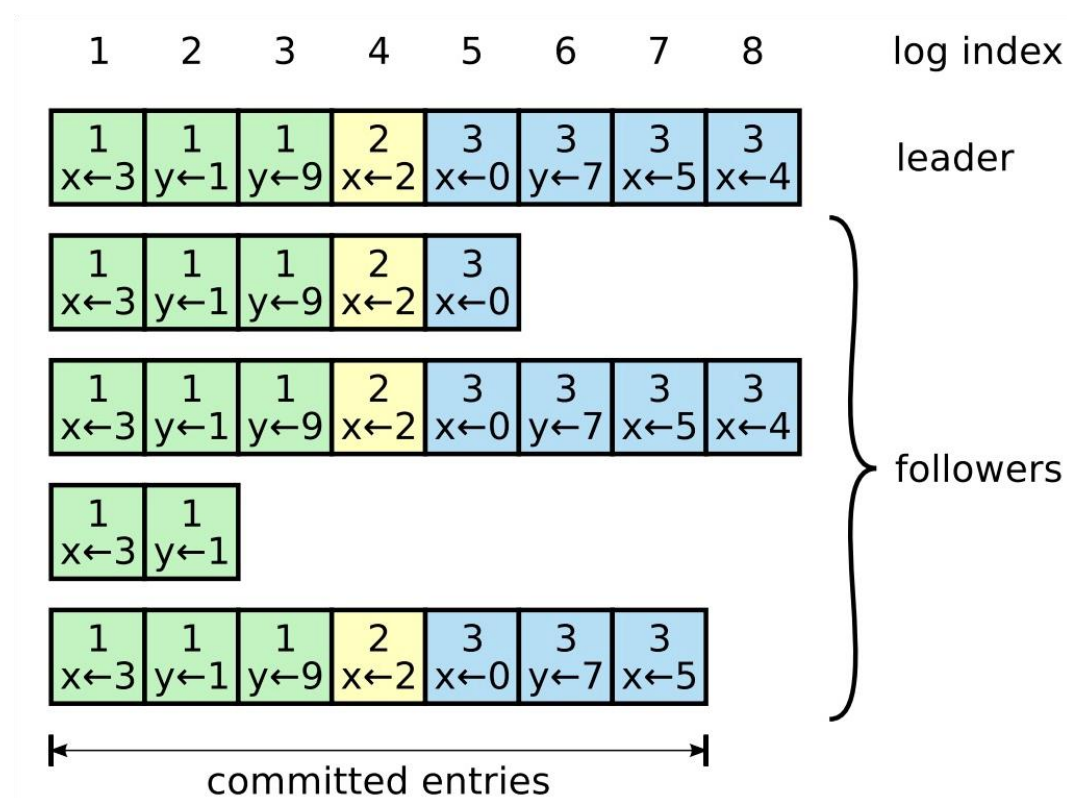


图 4 日志图示

2.1.5 算法流程

Raft 算法将分布式一致性分解为多个子问题，包括 Leader 选举 (Leader election)、

日志复制 (Log replication)、安全性 (Safety)、日志压缩 (Log compaction) 等，主要讲解其中关键的几个部分。

(1) Leader 选举

Raft 使用心跳超时 (heartbeat timeout) 机制来触发 Leader 选举。节点启动时默认处于 Follower 状态，如果 Follower 超时未收到 Leader 心跳信息，会转换为 Candidate 并向其他节点发起 RequestVote 请求。

当 Candidate 收到半数以上的选票之后成为 Leader，开始定时向其他节点发起 AppendEntries 请求以维持其 Leader 的地位。Leader 失效之后停止发送心跳，Follower 的心跳超时机制又会被触发，开始新一轮的选举。

(2) 日志复制

在节点集群中，只有 Leader 对外提供服务，客户端与 Leader 进行通信时，每个请求包含一条可以被状态机执行的命令。当 Leader 在接收到命令之后，首先会将命令转换为一条对应的日志记录 log entry，并追加到本地的日志中。然后调用 AppendEntries 将这条日志复制到其他节点的日志中。

当日志被复制到过半数节点上时，Leader 会将这条日志中包含的命令提交 commit 状态机执行，最后将执行结果告知客户端。

(3) 状态安全

在选举时，Raft 会保证新的 Leader 拥有所有已经提交的日志，每个 Follower 节点在投票时会检查 Candidate 的日志索引，并拒绝为日志不完整的 Candidate 投赞成票，



半数以上的 Follower 节点都投了赞成票,意味着 Candidate 中包含了所有可能已经被提交的日志

在提交日志时, Leader 只主动提交自己任期内产生的日志,如果记录是当前 Leader 所创建的,那么当这条记录被复制到大多数节点上时, Leader 就可以提交这条记录以及之前的记录。

如果记录是之前 Leader 所创建的,则只有当前 Leader 创建的记录被提交后,才能提交这些由之前 Leader 创建的日志。

2.2 Dragonboat 实现原理

Dragonboat 是一个高性能的多组 Raft 共识协议库,由 Go 语言实现,并且也支持 C++11。它旨在提供强大的故障容忍能力,保证在大多数服务器可用的情况下系统仍能正常运行。Dragonboat 通过将多个 Raft 组整合到一个集群中,使得每个组都可以独立处理读写请求,从而提高了系统的可扩展性和性能。

2.2.1 NodeHost

NodeHost 结构是 dragonboat 提供的所有功能的门面接口。每个 NodeHost 实例(由其 RaftAccess 属性标识)通常在管理其中央处理器、存储和网络资源的单独主机上运行。

每个 NodeHost 都可以管理来自许多不同 Raft 组（称为 Raft 集群）的 Raft 节点。每个 Raft 集群都由其 ButterID 标识。每个 Raft 集群通常由多个节点组成，这些节点由其 NodeID 值标识。来自同一 Raft 集群的节点应该分布在整个网络的不同 NodeHost 实例上，这为节点故障带来了故障的容忍性，因为只要其大部分管理 NodeHost 实例（即其底层主机）可用，存储在此类 Raft 集群中的应用程序数据就可以使用。

用户应用程序可以通过实现其 IStateMachine 组件来利用 Dragonboat 中实现的 Raft 协议的强大功能。每个集群节点都与一个 IStateMachine 实例相关联，它负责更新、查询和快照应用程序数据。

用户应用程序可以使用 NodeHost 的 API 来更新其 IStateMachine 实例的状态，这称为提出建议。一旦被 Raft 集群的大多数节点接受，该提案将被视为已提交，并将应用于 Raft 集群的所有成员节点。应用程序还可以进行线性化读取来查询其 ISateMachine 实例的状态。Dragonboat 使用 ReadIndex 协议来实现线性化读取。虽然在 leader 节点启动的负载最低，但是读和写操作支持在任何节点上启动。

2.2.2 待提议的请求

当用户提交一个写请求（称为 Propose，即提议）后，从性能考虑一般它并不是被立刻处理，而需要暂存。Go 实现中，一般是用 channel 来暂存，并通过 channel 将待提议的请求传递给负责处理的部分。

显然，这会带来一定的性能问题，每次只能从 channel 中取出一个请求后逐一处理。而 channel 是个功能丰富的内建类型，比如协程的特性决定对 channel 的操作甚至可以带来协程的调度。

在 Dragonboat 中处理单个请求时，均避免使用 channel 做任何数据的传递。在处理待提议请求上，使用的是由两个 slice 来回切换而构成的一个可以整体访问的 queue，每次取出当前 slice 中的所有待提议请求，一次合并（batching）地完成提议，由此完成了性能的优化。

2.2.3 Raft Log Entry 的磁盘保存

在 Raft 的实现中，其要求被 commit 的 entry 至少在过半数的机器上完成了落盘的保存，这确保只要 quorum 存活，磁盘上必然有所有已 commit 的 entry。当每秒面对千万规模的 Log Entry 的写入，因为基数巨大，任何冗余的数据的写入都浪费磁盘空间与带宽，进而拖累延迟与吞吐性能。

以使用 RocksDB 等 Key-Value store 库来存储 Raft 的 Log Entry 库为例，对于某个 Raft 组而言，通常每个 log entry 被作为一条记录存储，它的 key 和 value 如图 5 所示。

Keys			Values			
Type	ClusterID	Index	Index	Term	Session	Payload
1	100	1	1	1	S1	P1
1	100	2	2	1	S2	P2
1	100	3	3	1	S3	P3
1	100	4	4	1	S4	P4
1	100	5	5	1	S5	P5
1	100	6	6	1	S6	P6
1	100	7	7	1	S7	P7
1	100	8	8	1	S8	P8

locality of reference

图 5 每一 log entry 的 key 和 value 值结构

这样的设计会带来一定的问题，

1. Key 和 Value 里存在大量的重复值，如上图中的 ClusterID 和 Term 字段，
2. 同时 Key 和 Value 里均含有大量低信息量的值，如上图中的 Index 字段。

3. 每个 entry 被设定为一个 Key-Value store 的记录，可被独立地读写访问。但对共识库而言，磁盘上 entry 的访问具有很强的访问局部性，某 entry 被读写以后，它下一条 Log Entry 很可能是接着马上要被访问。

对上述简单的设计，面对大量的如每秒 1000 万次写，则每秒需添加 1000 万条记录，内含 1000 万个值完全重复的 ClusterID 与 Term 记录，这必然是低效的。

以 etcd 为例，其 WAL 包负责 Raft Log Entry 的保存，虽没有使用 Key-Value Store，且因为单组而无需保存 ClusterID，但上述三个问题 etcd 依然都具备。

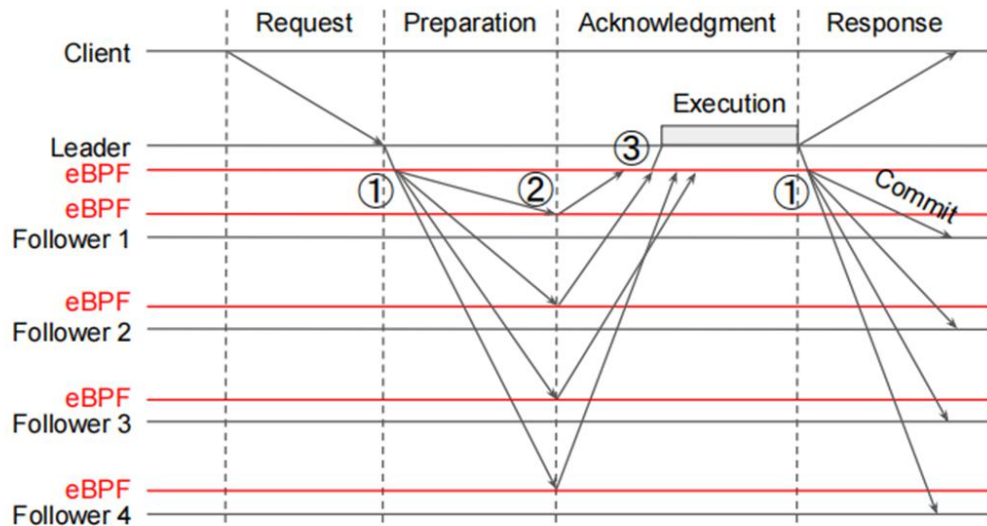
Dragonboat 中，Log Entry 的存储避免上述归纳的三个问题。在 Leader 不变，从而 Term 不变 Index 连续递增的通常情况下，大幅降低了空间的浪费和写入带宽的需求，对延迟的改善也很明显。

2.3 Electrode 实现原理

Electrode 是一个将内核网络栈下的 Paxos 协议卸载到内核 eBPF 程序的框架，以减少用户-内核交叉和内核网络栈穿越。电极在设计其 eBPF 卸载方面有两个目标：

- 大幅减少内核堆栈开销以提高性能；
- 仔细划分用户和内核空间功能，以保持卸载在 eBPF 子系统内的可实现性和效率。

为了实现第一个目标，需要选择那些普遍的对性能具有决定性影响的操作进行卸载。如下图所示，Electrode 卸载了消息广播、快速确认和定量等待三个操作。从图上可以看出 eBPF 作用的时间点，大致简明地表现了 Electrode 的优化方式。



(b) Electrode-accelerated Multi-Paxos/Viewstamped Replication.

为了实现第二个目标，就不能把所有的 Paxos 操作都转移到内核中实现。Electrode 将一些复杂的操作留在用户空间中实现。具体而言，Electrode 将故障恢复和消息丢包及失序的处理按照与 VR 协议及 NOPaxos 相似的机制进行处理。这些程序包含内存的动态空间处理，将难以通过 eBPF 的静态验证。

综合以上两个因素，图中的关键节点就显现出来。文章将围绕这些节点进行实现和验证。

2.3.1 工作流程（实现内容）

首先，用户程序将 eBPF 程序绑定到各个网络接口的 hook 点上，随后用户程序使用 Electrode 提供的 API 来调用 eBPF 卸载的函数或获取 eBPF 处理的结果，最后 eBPF 程序解释和处理目标网络包，直接在内核中完成而不再通过网络栈和用户空间应用。

Paxos 协议中的各个元素，包括锁（locks）、障碍（barriers）、配置参数等能够存入单个以太网包中，正好满足 eBPF 的处理方式。其它包则转由用户程序处理。

2.3.2 总体框架

主要通过 ebpf 进行修改的是三个部分：TC 中的消息广播、XDP 中的快速确认、TC +

XDP 中的 Wait-on-Quorums。

电极实现修改内容的的主要架构为六个 ebpf 程序：

tc_broadcast_and_quorum：此程序拦截传出的准备消息。它实现了消息广播机制和定量等待的 tc_ebpf 函数。

xdp_dispatcher：此程序将检查传入消息的类型，并调用相应的消息处理程序，它仅拦截 ACK（仅在领导节点上接收）和准备（仅在跟随节点上接收）消息，并调用相应的 handle_ACK 和 handle_preparation 程序。

handle_ACK：xdp_ebpf 函数，主要用于实现 wait-onquorums 部分的功能，它会丢弃大多数 ACK 消息，而仅将达到法定数量的 ACK 消息转发给用户空间应用程序。

handle_preparation：检测非关键路径的情况，检查到时，将消息转发到用户空间的应用程序。在正常情况下（大多数情况下），它将调用 write_buffer 以开始 fast_ACK。

write_buffer：此程序将消息/数据包数据存储在内核日志中，供用户空间应用程序进行轮询和使用。如前所述，主要使用 eBPF 环缓冲区实现日志数据结构，然后，该程序调用 fast_ACK 程序

fast_ACK：重用并修改接收到的包缓冲区，以创建一个 ACK 包并将其发送出去。

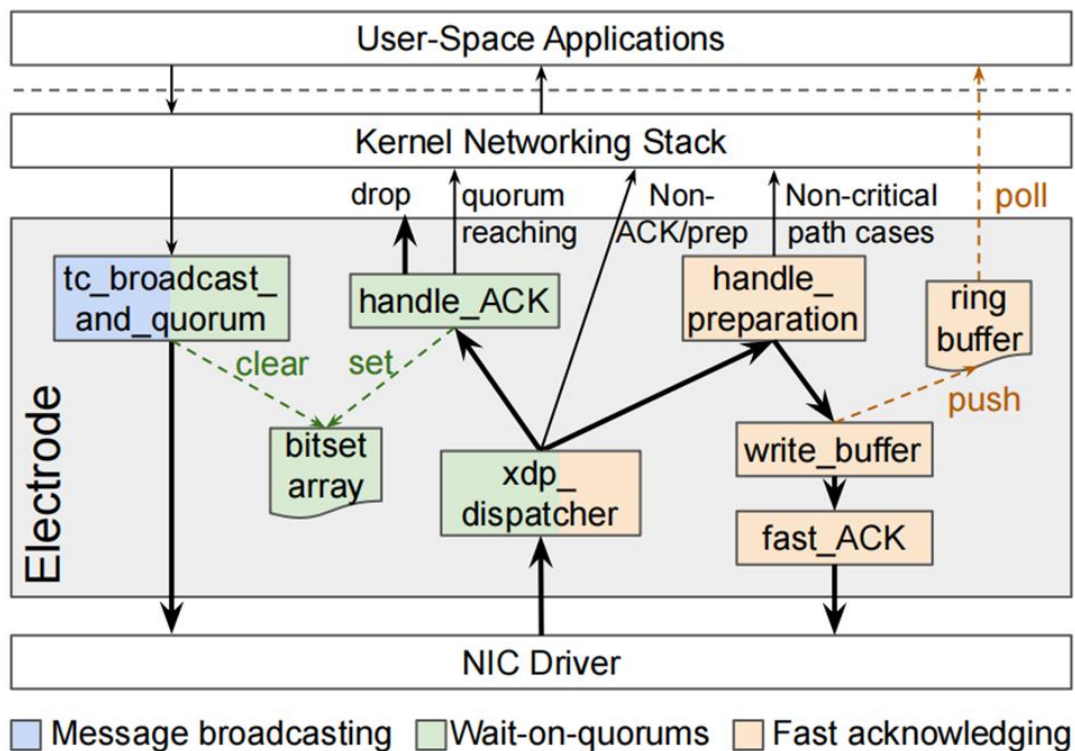


图 6 Electrode 结构

(1) 消息广播

消息广播在 Paxos 中使用广泛，Leader 节点向 follower 节点发送 prepare 消息，及 leader 节点向 follower 节点发送 commit 消息。

当前实现方法主要存在的性能缺陷是从用户空间向不同地址多次发送相同的报文，其 overhead 随 follower 的增加而增加，而每个 follower 节点的 overhead 不变。经典的 leader 瓶颈问题。IP 多播方式则需要硬件和软件支持。

使用 eBPF 的 TC hook，使用 helper 函数 `bpf_clone_redirect()` 在内核中复制报文包，更改其目的地址并分别发送。

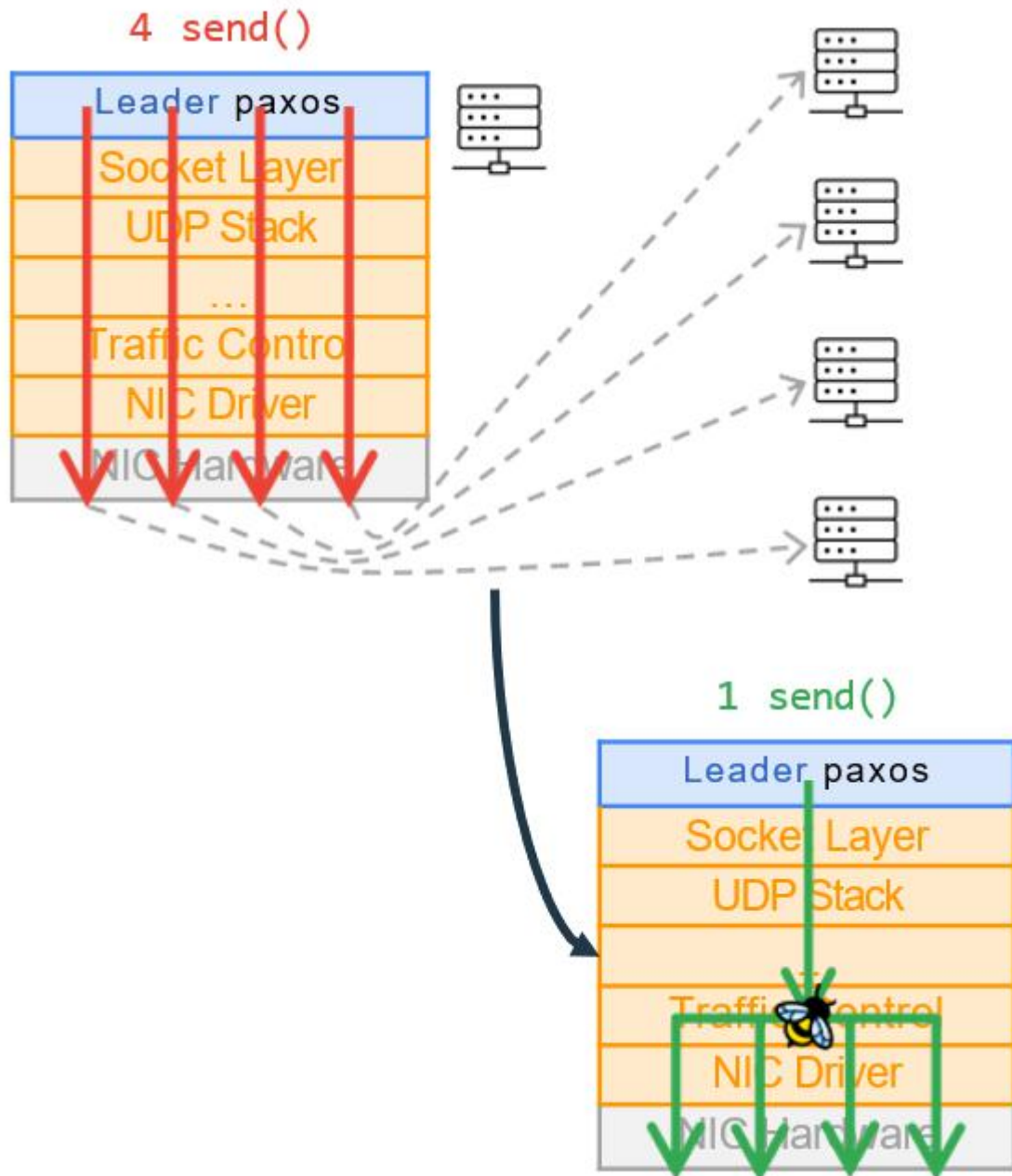


图 7 使用 eBPF 对消息广播的改进示意图

(2) 快速确认

快速确认是 Paxos 中的请求延迟主要来源之一，在 follower 节点中将送达的 prepare 报文缓存在内核日志中，由用户空间应用异步轮询。内核日志由 BPF_MAP_TYPE_RINGBUF 这个 eBPF map 数据结构实现，具有很高的效率。

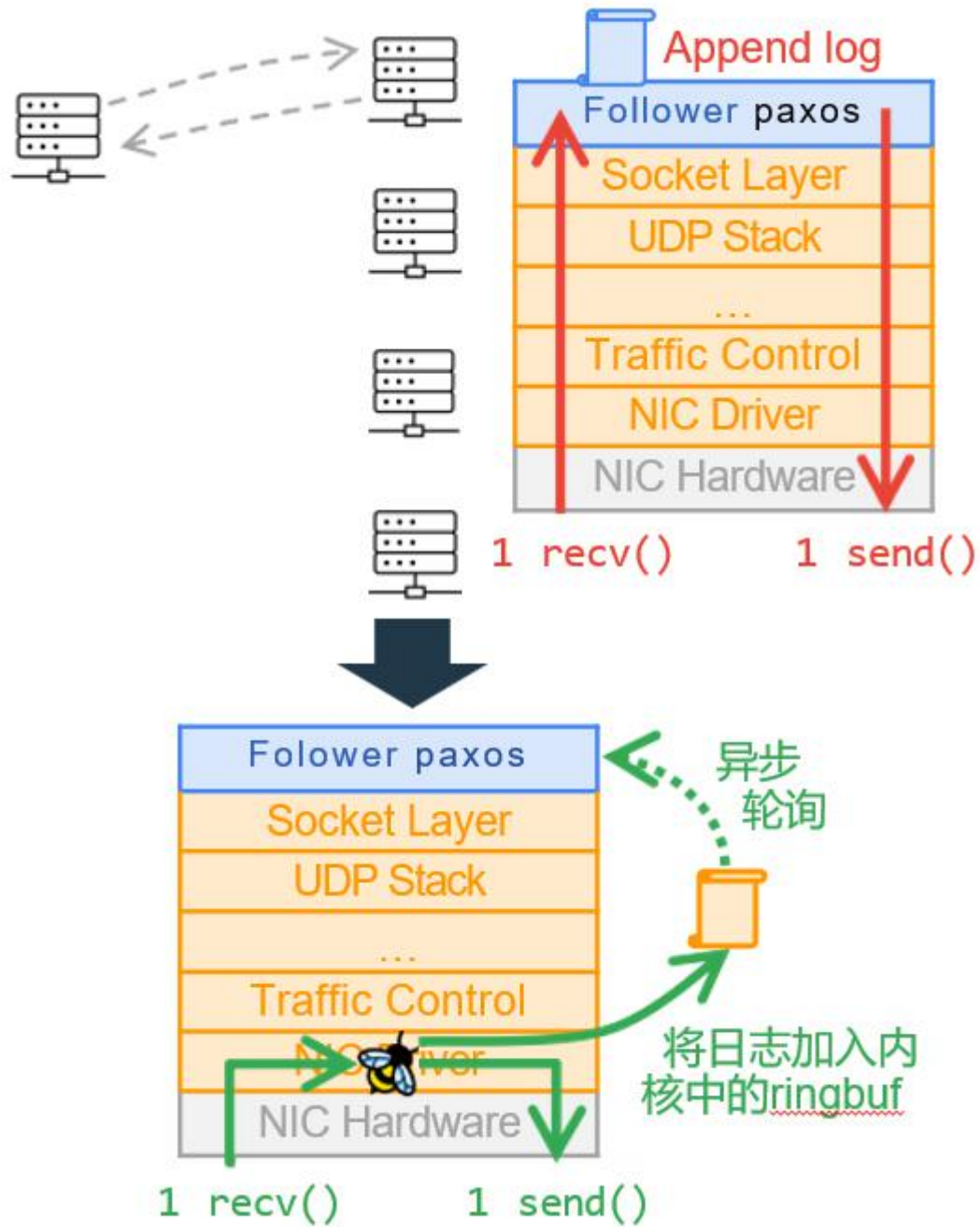


图 8 使用 eBPF 对快速确认的改进示意图

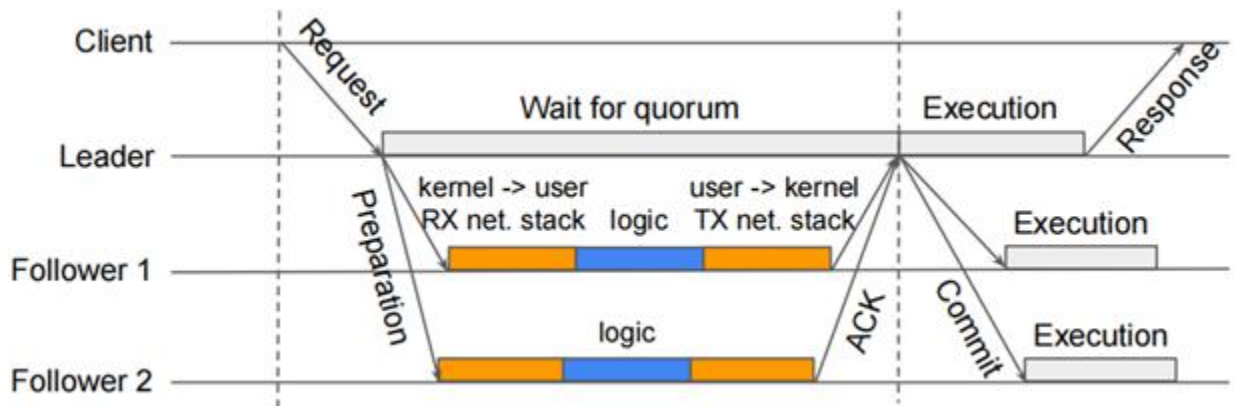


图 9 原系统性能

使用 eBPF 进行优化后，系统的性能会得到明显的提升。

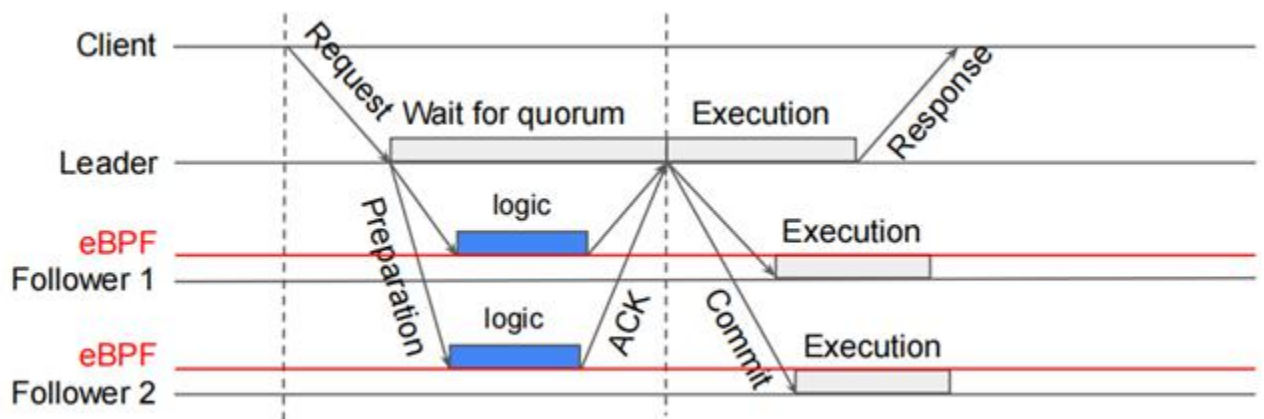


图 10 eBPF 优化后性能

(3) 定量等待

定量等待是 Paxos 中的关键路径，Leader 节点需要保证绝大多数 follower 节点都已经有了当前请求的副本，才能执行 commit 完成请求。

当前实现方法主要的性能缺陷是在用户空间中收集所有 follower 节点的 ACK 信息，进行计数。有大量报文不被需要却占用了 leader 节点的 CPU。

用 eBPF 将定量等待转移到内核，存留一个比特数组用于记录信息，当达到定量时再将消息传递到用户态应用。在 TC 钩上清除比特位，在 XDP 钩上置位。

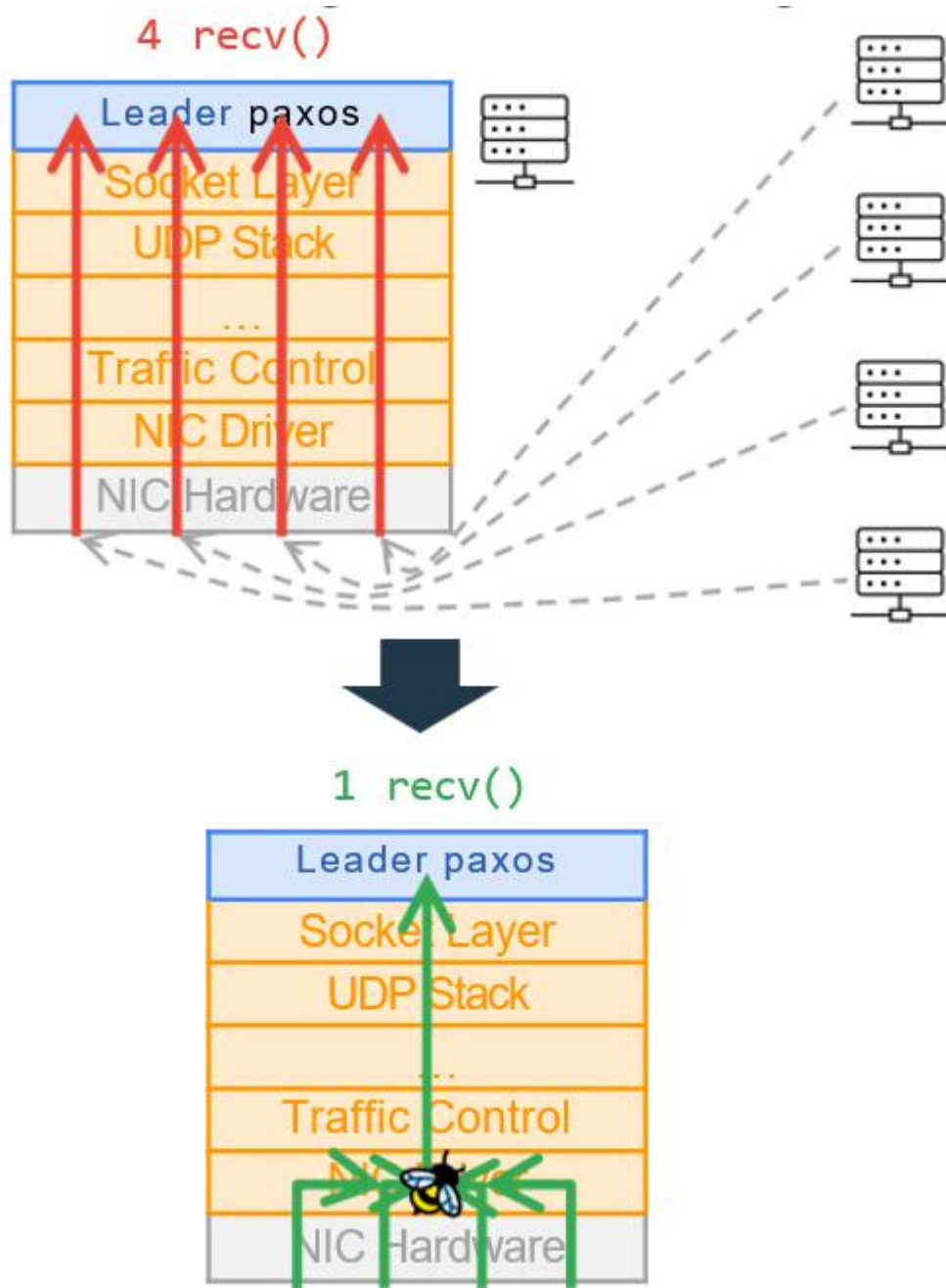


图 11 使用 eBPF 对定量等待的改进示意图

2.3.3 代码分析

(1) 源码结构分析

Electrode 源码中主要包含两个部分内容：

第一，在“VR/”中实现了 VR (Viewstapped Replication) 协议，代码来自推测性 Paxos (<https://github.com/UWSysLab/specpaxos>). 其中做了一些修改来实现功能。基本结构如下：

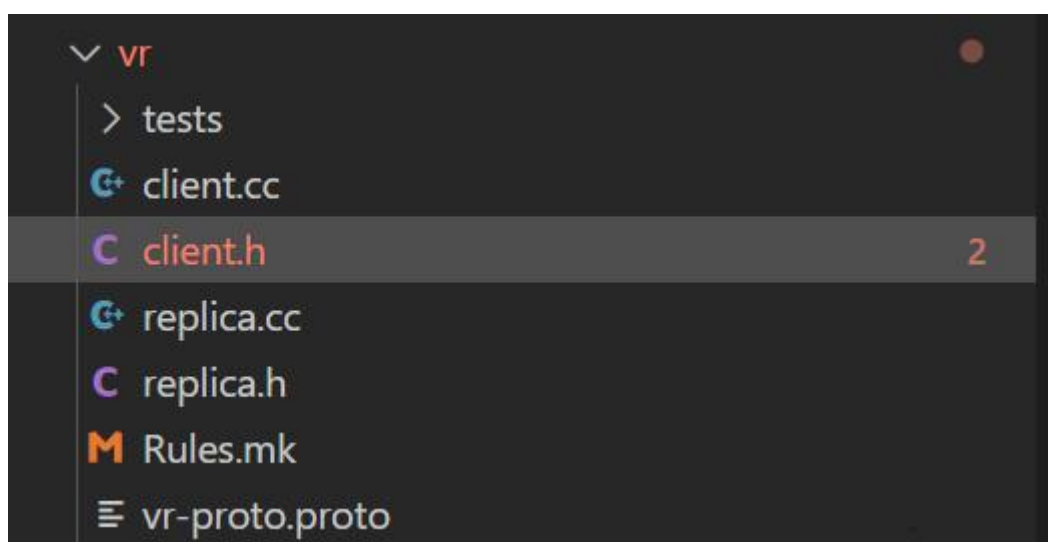


图 12 VR/ 目录结构

第二，通过 eBPF 在“xdp-handler/”中实现了三个优化，其原始结构来自[BMC]的源代码(<https://github.com/Orange-OpenSource/bmc-cache>).

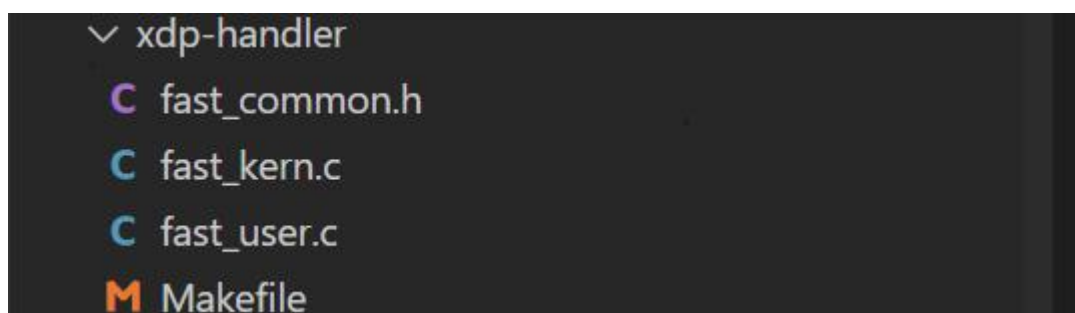


图 13 xdp-handle/r 目录结构

先前分析中可知，Electrode 中关于 paxos 协议的代码都来自于推测性 Paxos，下载该项目源码，对其进行分析，和电极中的代码进行比较。

specpaxos 项目结构：

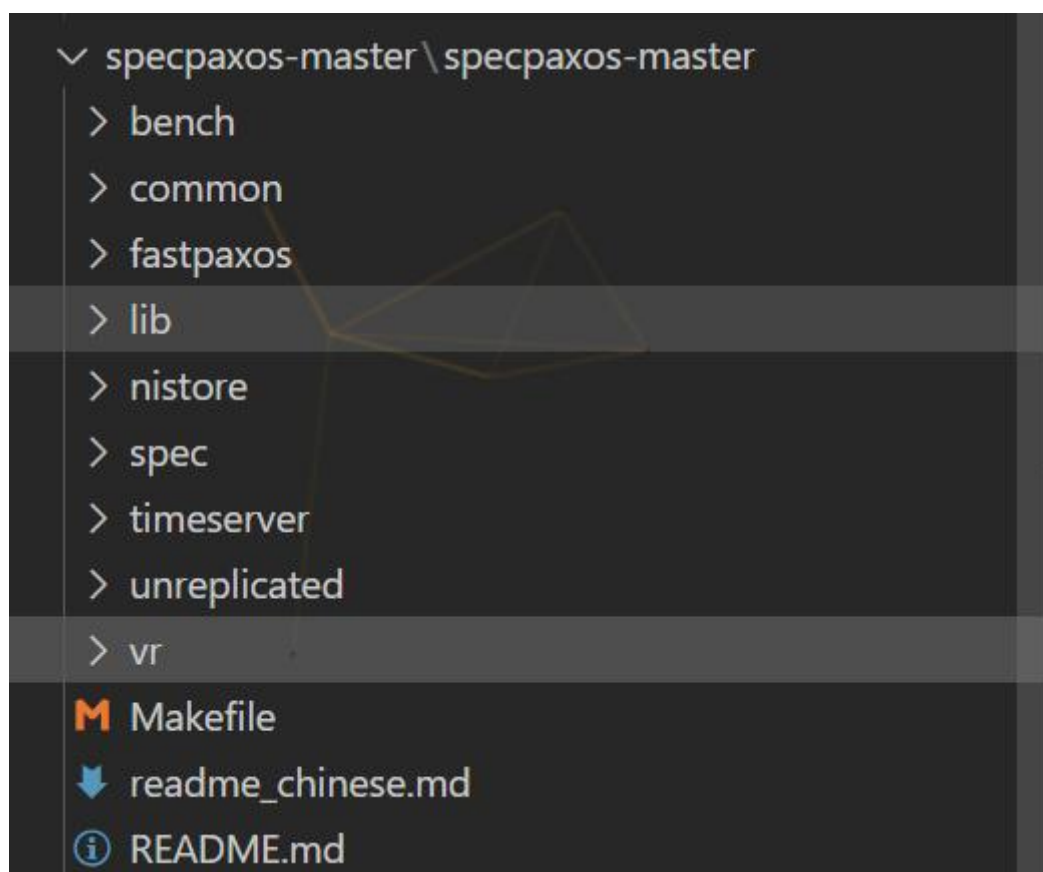


图 14 specpaxos/ 目录结构

- bench : 一个以尽可能快的速度进行的基准测试
- fastpaxos : fast paxos , 只实现了基本情况 (不知此处的 fast paxos 指的是什么, 要探究可能需要进一步阅读论文)
- nistore : 一个事务性键值存储 ("nistore"), 支持使用任意一个严格的多版本控制和并发控制两阶段锁定或乐观并发控制。 (这是源自中使用的代码库 TAPIR)
- spec : Speculative Paxos , 推测性 paxos , 实现包括正常操作、同步和协调协议。 (同样不知区别)
- timeserver : 一个用于分布式的存储系统简单的时间戳服务器。
- unrepliated : 一个简单用于比较的原始版本
- vr : Viewstapped Replication (VR) , 又名 Multi-Paxos , 论文 "Viewstamped Replication Revisited" 中实现的, 包括可选的批处理优化

比较代码的变化，Electrode 改变的代码内容集中在 replica.cc 和 replica.h 两个文件中。

(2) Electrode 关键代码

replica.cc

归纳 replica.cc 中主要函数及其主要功能：

1. request_polling 函数

```
1. int request_polling(void *ctx, void *data, size_t data_sz) /
   / 函数接受三个参数：ctx (上下文指针)，data (数据指针)，data_sz (数据
   大小) by:wbh

2. {
3.     VRReplica *pt = (VRReplica *)ctx;
4.     // 将 ctx 转换为 VRReplica 指针类型
5.     sockaddr_in sender;
6.     sender.sin_family = AF_INET;
7.     sender.sin_port = *(in_port_t *)data;
8.     data += sizeof(in_port_t);
9.     sender.sin_addr.s_addr = *(in_addr_t *)data;
10.    data += sizeof(in_addr_t);
11.    // 从 data 中解析出发送者的信息，包括端口号和 IP 地址，并将其保存
    在 sender 结构体中。
12.    uint64_t size = *(uint64_t *)data;
13.    static RequestMessage request;
14.    request.ParseFromString(string((char *)data + sizeof(uint64_t), size));
15.
16.    UDPTransportAddress senderAddr(sender);
17.    pt->HandleRequest(senderAddr, request);
```

```

18.    // 创建一个 UDPTransportAddress 对象，表示发送者的地址，并调
        用 HandleRequest 方法处理请求
19.    return 0;
20.}

```

该函数是在定义了 FAST_BATCH 前提下运行的，函数接受三个参数：ctx（上下文指针），data（数据指针），data_sz（数据大小）创建一个 UDPTransportAddress 对象，表示发送者的地址，并调用 HandleRequest 方法处理请求

2. prepare_polling 函数

```

1. int prepare_polling(void *ctx, void *data, size_t data_sz)
2. {
3.     VRReplica *pt = (VRReplica *)ctx;
4.     // 将 ctx 转换为 VRReplica 指针类型 by:wbh
5.     uint64_t size = *(uint64_t *)data;
6.     static PrepareMessage prepare;
7.     prepare.ParseFromString(string((char *)data + sizeof(uint64_t), size));
8.     // 创建了一个 PrepareMessage 类型的静态变量 prepare，然后调
        用 prepare.ParseFromString 方法将数据解析到 prepare 变量中
9.
10.    pt->HandlePrepare_Kernel(prepare);
11.    return 0;
12.}

```

```
#ifdef FAST_REPLY
```

创建了一个 PrepareMessage 类型的静态变量 prepare，然后调用 prepare.ParseFromString 方法将数据解析到 prepare 变量中，HandlePrepare_Kernel 处理请求

3. ModifyKernelState 函数

```

1. void VRReplica::ModifyKernelState() // 修改内核状态 by:wbh
2. {

```

```

3.      // 定义了一个结构体 paxos_ctr_state , 包含了一些状态信息
4.      struct paxos_ctr_state
5.      {
6.          enum ReplicaStatus state;
7.          int myIdx, leaderIdx, batchSize;
8.          __u64 view, lastOp;
9.      } state;
10.
11.     state.state = status;                                // 当前状态
12.     state.view = view;                                    // 当前视图号( ?
13.     state.lastOp = lastOp;                                // 最后一次操作
14.     state.myIdx = myIdx;                                    // 当前节点的索引
15.     state.leaderIdx = configuration.GetLeaderIndex(view); //
        领导者节点的索引
16.     state.batchSize = batchSize;                            // 批处理大小
17.
18.     unsigned int zero = 0;
19.     // 调用 bpf_map_update_elem 函数将其更新到名为
        paxos_ctr_state_fd 的 BPF 映射
20.     if (bpf_map_update_elem(paxos_ctr_state_fd, &zero, &state, 0) < 0)
21.     {
22.         fprintf(stderr, "Error: bpf_map_update_elem \"ctr_state\" failed\n");
23.         exit(1);
24.     }
25.     // asd123www: maybe here is a polling?
26. }

```

```
#if defined FAST_REPLY || defined FAST_BATCH || defined
```

FAST_QUORUM_PRUNE 修改内核状态。

定义了一个结构体 `paxos_ctr_state`，包含了一些状态信息 当前视图号、当前状态、最后一次操作、当前节点的索引、领导者节点的索引、批处理大小 调用 `bpf_map_update_elem` 函数将其更新到名为 `paxos_ctr_state_fd` 的 BPF 映射

4. GenerateNonce 函数

```
1. VRReplica::GenerateNonce() const
2. {
3.     std::random_device rd;
4.     std::mt19937_64 gen(rd());
5.     std::uniform_int_distribution<uint64_t> dis;
6.     return dis(gen);
7. }
```

该函数功能较为简单，使用硬件提供的随机数初始化随机数引擎，创建 64 位整数均匀分布，最后生成并返回一个 64 位随机整数。

5. AmLeader 函数

通过 `GetLeaderIndex(view) == myIdx` 的判断，查看当前视图是不是领导结点。

6. CommitUpTo 函数

```
1. void
2. VRReplica::CommitUpTo(opnum_t upto)
3. {
4.     // 循环提交日志中的操作，直到达到指定的操作号
5.     while (lastCommitted < upto)
6.     {
7.         Latency_Start(&executeAndReplyLatency); // 开始测量执
            行和回复的延迟
8.
9.         lastCommitted++; // 提交编号递增
10.
11.        // 在日志中查找当前操作
12.        const LogEntry *entry = log.Find(lastCommitted);
13.        if (!entry)
```

```
14.      {
15.          RPanic("Did not find operation " FMT_OPNUM " in
log", lastCommitted);
16.      }
17.
18.      // 执行找到的操作
19.      RDebug("Executing request " FMT_OPNUM, lastCommitted)
;
20.      ReplyMessage reply;
21.      Execute(lastCommitted, entry->request, reply);
22.
23.      // 设置回复中的视图和操作号
24.      reply.set_view(entry->viewstamp.view);
25.      reply.set_opnum(entry->viewstamp.opnum);
26.      reply.set_clientreqid(entry->request.clientreqid());
27.
28.      // 标记操作为已提交
29.      log.SetStatus(lastCommitted, LOG_STATE_COMMITTED);
30.
31.      // 更新客户端表，记录回复
32.      ClientTableEntry &cte =
33.          clientTable[entry->request.clientid()];
34.      if (cte.lastReqId <= entry->request.clientreqid())
35.      {
36.          cte.lastReqId = entry->request.clientreqid();
37.          cte.replied = true;
38.          cte.reply = reply;
39.      }
40.      else
41.      {
42.      // 如果有后续的操作被准备，那么这个请求被认为已在客户端完成，不需要记录
结果
43.      }
44.      // 如果找到客户端的地址，发送回复
45.      auto iter = clientAddresses.find(entry->request.clie
ntid());
46.      if (iter != clientAddresses.end())
```

```

47.         {
48.             transport->SendMessage(this, *iter->second, repl
            y);
49.         }
50.         Latency_End(&executeAndReplyLatency); // 结束测量
51.     }
52. }

```

该函数实现了一个循环提交日志的过程，其目的是同步并执行一系列操作直至达到指定的操作号（inclusive）。具体步骤包括：首先，从当前状态开始遍历日志文件中的操作；其次，对每个操作执行相应的业务逻辑，并生成回复消息；然后，更新客户端的状态信息；接着，将处理后的回复发送回客户端；最后，记录从接收请求到发送回复整个过程的时间消耗，以监控性能。这一过程通常应用于分布式系统中的状态机复制场景，确保多个节点上的操作一致性。@param upto 指定需要提交到哪个操作号。

7. SendPrepareOKs 函数

```

1. void
2. VRReplica::SendPrepareOKs(opnum_t oldLastOp)
3. {
4.     // 遍历日志中的操作，寻找需要发送 PREPAREOK 的消息的操作
5.     for (opnum_t i = oldLastOp; i <= lastOp; i++)
6.     {
7.         // 如果操作已经被提交，则跳过
8.         if (i <= lastCommitted)
9.         {
10.             continue;
11.         }
12.
13.         // 在日志中查找当前操作
14.         const LogEntry *entry = log.Find(i);
15.         // 如果找不到操作，则panic
16.         if (!entry)

```

```

17.      {
18.          RPanic("Did not find operation " FMT_OPNUM " in
            log", i);
19.      }

20.      // 断言：找到的操作必须处于 PREPARED 状态
21.      ASSERT(entry->state == LOG_STATE_PREPARED);
22.      // 更新客户端表
23.      UpdateClientTable(entry->request);
24.
25.      // 准备 PREPAREOK 消息
26.      PrepareOKMessage reply;
27.      reply.set_view(view);
28.      reply.set_opnum(i);
29.      reply.set_replicaidx(myIdx);
30.
31.      // 打印调试信息，说明正在发送 PREPAREOK 消息
32.      uint32_t my_data[3] = {(uint32_t)view, (uint32_t)i,
            (uint32_t)myIdx};
33.      RDebug("Sending PREPAREOK " FMT_VIEWSTAMP " for new
            uncommitted operation",
34.             reply.view(), reply.opnum());
35.
36.      // 发送 PREPAREOK 消息给领导者。如果发送失败，则打印警告信息。
37.      if (!(transport->SendMessageToReplica(this,
38.                                             configuratio
            n.GetLeaderIndex(view),
39.                                             reply, my_da
            ta)))
40.      {
41.          RWarning("Failed to send PrepareOK message to le
            ader");
42.      }
43.  }
44. }

```

向其他副本发送 PREPAREOK 消息，用于确认新的未提交操作。遍历从 oldLastOp 到 lastOp 的所有操作，对那些新且未提交的操作，发送 PREPAREOK 消息。PREPAREOK 消

息包含操作号、视图号和发送者的索引。

@param oldLastOp 是旧的最后一个操作号,观察到 SendMessageToReplica 函数,多了参数,所有发送文件的消息的函数都多了这个参数,后面详解。

8. SendRecoveryMessages 函数

发送恢复信息给所有副本,以请求恢复操作。

9. RequestStateTransfer 函数

请求状态传输的函数。当发现当前状态陈旧时,会调用此函数来请求更新状态。

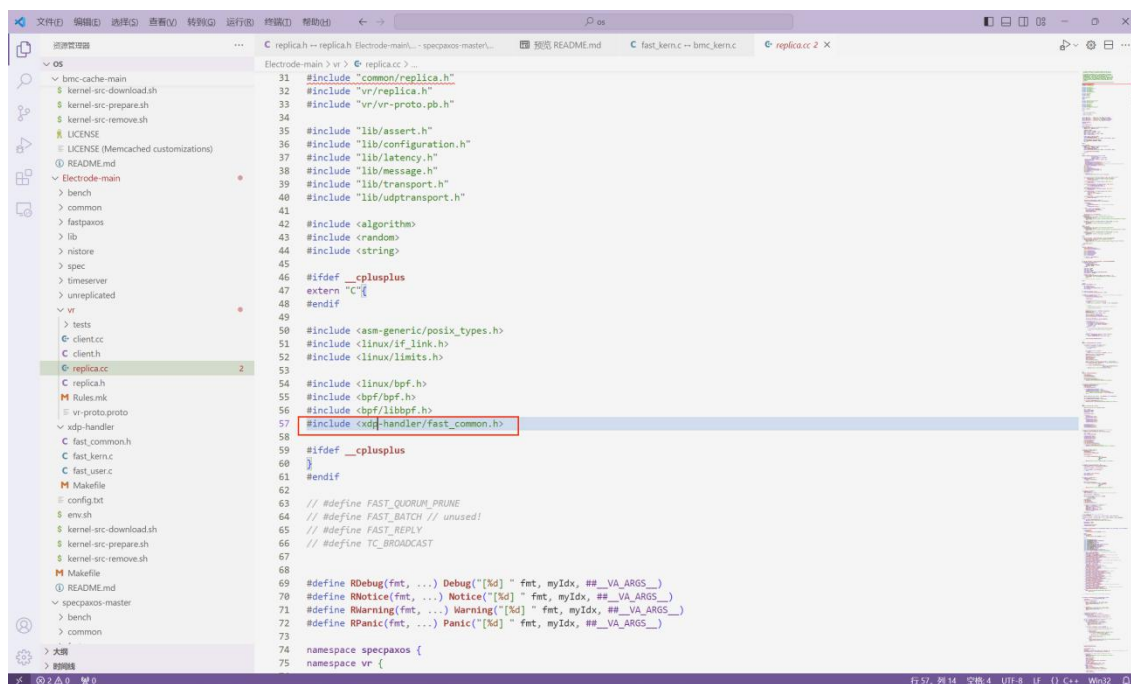
10. EnterView 函数

用来进行视图切换操作的,根据当前节点是否为领导者进行不同的逻辑处理,并且在进入新视图时进行相应的状态更新和超时计时器管理

xdp-handler

eBPF 主要的优化代码实现在 xdp-handler/中,这个结构来自于 bmc 的源代码,即此优化代码是根据 bmc 改编的。

对于 xdp-handler 中 fast_common.h 进行直接调用,其位于文件 replica.cc 中:



fast_common.h 是一个头文件，主要定义了一些常量、枚举类型和结构体，以及一些预定义的 XDP 程序和 TC 程序的操作码。

1. 定义了常量：

- ETH_ALEN：以太网地址的字节数
- CLUSTER_SIZE：集群大小，值为 3
- FAST_REPLICA_MAX：最大副本数量，值为 100
- NONFRAG_MAGIC 和 FRAG_MAGIC：用于识别非分片和分片数据的魔数
- MAGIC_LEN、REQUEST_TYPE_LEN、PREPARE_TYPE_LEN、PREPAREOK_TYPE_LEN、MYPREPAREOK_TYPE_LEN、FAST_PAXOS_DATA_LEN：用于定义数据的长度
- BROADCAST_SIGN_BIT：广播标志位
- QUORUM_SIZE：法定数量大小
- QUORUM_BITSET_ENTRY：用于记录法定数量的位集合条目大小

2. 定义了枚举类型 ReplicaStatus

其包括三个状态：STATUS_NORMAL、STATUS_VIEW_CHANGE、STATUS_RECOVERING，表示副本的状态

3. 定义了两个枚举类型，分别表示 XDP 程序和 TC 程序的操作码
4. 定义了一个结构体 paxos_configure，用于存储 Paxos 配置信息，包括 IP 地址、端口和以太网地址

三、开发思路与具体代码

3.1 kern.c 改写部分

快速广播通过 bpf_clone_redirect 实现，在内核层面将一条数据复制成多条，并分发到不同的目标。在此过程中，使用 Appl 头中的 Message Type 作为中间变量来标识每条复制的数据是第几条。为了配合等待仲裁（wait on quorum），广播需要清空一些位集合（bitset）。

```

1. if (type_str[1] == 'p' && type_str[0] == 's')
2. {
3.     id = !ctr_state -> leaderIdx;
4.     nxt = id + 1;
5.     nxt += ctr_state -> leaderIdx == nxt;
6.     type_str[0] = nxt;
7.     type_str[1] = 'M';
8.     if (nxt < CLUSTER_SIZE) bpf_clone_redirect(skb, skb -> ifi
        index, 0);
9. }
10.
11. else

```

```

12.{
13.  nxt = type_str[0] + 1;
14.  nxt += ctr_state -> leaderIdx == nxt;
15.  type_str[0] = nxt;
16.  if (nxt < CLUSTER_SIZE) bpf_clone_redirect(skb, skb -> ifi
    ndex, 0);
17.}
18.
19.if (msg_type == FAST_PROG_XDP_HANDLE_PREPARE)
20.{
21.  __u32 idx = msg_lastOp & (QUORUM_BITSET_ENTRY - 1);
22.  struct paxos_quorum *entry = bpf_map_lookup_elem(&map_quor
    um, &idx);
23.  if (entry && entry -> view != msg_view || entry -> opnum !
    = msg_lastOp)
24.  {
25.    entry -> view = msg_view;
26.    entry -> opnum = msg_lastOp;
27.    entry -> bitset = 0;
28.  }
29.}

```

快速确认 (fast ack) 需要模拟 follower 在内核中进行回复，因此需要知道用户态的状态，从而正确回复。这通过一个名为 ctr_state 的映射来实现。然后将信息暂存在一个 ringbuffer 中，由上层代码来消费。快速回复缩短了用户请求的关键路径，但需要注意内核可能遇到无法处理的丢包情况，需要交由上层处理。

```

1. char *pt = bpf_ringbuf_reserve(&map_prepare_buffer, MAX_DATA
    _LEN, 0);
2. if (pt)
3. {
4.   for (int i = 0; i < MAX_DATA_LEN; ++i)
5.     if (payload + i + 1 <= data_end) pt[i] = payload[i];
6.   bpf_ringbuf_submit(pt, 0);
7.   bpf_tail_call(ctx, &map_progs_xdp, FAST_PROG_XDP_PREPARE_R
    EPLY);
8. }

```

等待仲裁 (wait on quorum) 主要是 leader 在内核中等待某个索引 (Index) 过半数后一次性返回给上层，表示该条目 (entry) 已经可以提交 (commit)。这里的思路与广

播 (broadcast) 类似, 旨在减少网络协议栈的遍历和用户态与内核态之间的切换。在此过程中, 使用位集合 (bitset) 进行计数, 它具有幂等性。位集合与快速广播 (fast broadcast) 配合使用, 一个负责清空, 另一个负责置一。

```
1. // case 1
2.  if (ctr_state -> state != STATUS_NORMAL)
3.      return XDP_DROP;
4.
5.  // case 2
6.  if (msg_view < ctr_state -> view)
7.      return XDP_DROP;
8.
9.  // case 3
10. if (msg_view > ctr_state -> view)
11.     return XDP_PASS;
12.
13. // case 4
14. if (msg_lastOp <= ctr_state -> lastOp)
15. {
16.     bpf_tail_call(ctx, &map_progs_xdp, FAST_PROG_XDP_PREPARE_REPLY);
17.     return XDP_PASS;
18. }
19.
20. // case 5
21. if (msg_batchStart > ctr_state -> lastOp + 1)
22.     return XDP_PASS;
```

3.2 user.c 改写部分

此部分主要负责加载映射 (map) 和 eBPF 程序。映射除了正常载入内核, 如果需要在用户态访问, 如环形缓冲区 (ringbuffer), 还需要将其固定在一个约定的位置。这里还有一类特殊的映射用于存放 eBPF 程序, 这是为了实现尾递归调用, 从而可以拆分 eBPF 程序以避

免超出限制。此代码还负责将预先规定的转发地址写入内核映射中。

eBPF 主要包括 5 个 XDP 程序和 1 个流量控制 (TC) 程序, 它们分别固定两个主程序, 其他程序通过尾递归启动。

vr 改写部分

VR 是一种多 Paxos 实现, Electrode 优化是基于 VR 进行的。用户态需要通过映射 (map) 更新内核状态来维护正确性。在快速确认 (fast ack) 中, eBPF 在 XDP 中提前回复信息, 并将数据包存储在一个环形缓冲区 (ringbuffer) 中。用户需要对环形缓冲区进行轮询, 但由于非关键路径已经在内核中处理, 此处无需重复验证。

```
1. if (progs[i].pin)
2. {
3.     if (snprintf(filename, PATH_MAX, "%s/%s", BPF_SYSFS_ROOT,
        progs[i].name) < 0)
4.         exit(-1);
5.     else if (snprintf(filename, PATH_MAX, "%s/%s", BPF_SYS
        FS_ROOT, progs[i].name) >= PATH_MAX)
6.         exit(-1);
7.
8.     FLAG:
9.     if (bpf_program__pin_instance(progs[i].prog, filename, 0))
10.    {
11.        fprintf(stderr, "Panic: Fail to pin program '%s' to path
            %s\n", progs[i].name, filename);
12.        if (errno == EEXIST)
13.        {
14.            fprintf(stdout, "BPF program '%s' already pinned, unpin
                ning it to reload it\n", progs[i].name);
15.            if (bpf_program__unpin_instance(progs[i].prog, filename,
                0))
16.            {
```

```

17.     fprintf(stderr, "Panic: Fail to unpin program '%s' at
    %s\n", progs[i].name, filename);
18.     exit(-1);
19. }
20.     goto FLAG;
21. }
22.     exit(-1);
23. }
24. }

```

为了给内核提供额外信息，用户态需要配合增加 Appl 头的报文，即增加一个 FAST_PAXOS_DATA_LEN 字段，因此涉及到整个传输模块的相应修改。

```

1. void VRReplica::ResendPrepare()
2. {
3.     ASSERT(AmLeader());
4.     if (lastOp == lastCommitted)
5.         return;
6.     RNotice("Resending prepare");
7.     if (!(transport->SendMessageToAll(this, lastPrepare
8.                                     #ifdef TC_BROADCAST
9.                                     , sgn_bits
10.                                    #endif
11.                                    )))
12.         RWarning("Failed to ressend prepare message to all r
    eplicas");
13. }
14.
15. void VRReplica::ReceiveMessage(const TransportAddress &remote,
    const string &type, const string &data)
16. {
17.
18.     #ifdef FAST_BATCH
19.     if (AmLeader())
20.     {
21.         assert(ring_buffer__poll(rb_request, 0) >= 0);
22.     }
23.     #endif
24.     #ifdef FAST_REPLY
25.     if (!AmLeader())

```

```
26.     {
27.         assert(ring_buffer__poll(rb_prepare, 0) >= 0);
28.     }
29.     #endif
30....
31. }
```

四、测试框架

在此项目的测试中，我们主要使用了 Github 上的 raft-bench 框架 (<https://github.com/alexanderstephan/raft-bench>)，raft-bench 是一个用于比较 etcd、hashicorp 和 dragonboat 的 RAFT 库性能的基准测试工具，它提供了由 Raft 一致性算法支持的关键值存储集群的原始性能洞察。

4.1 测试框架入口

```
1. package main
2.
3. import (
4.     "flag"
5.     "log"
6.     "os"
7.     "time"
8.
9.     "github.com/thanhphu/raftbench/dragonboat"
10.    "github.com/thanhphu/raftbench/etcd"
11.    "github.com/thanhphu/raftbench/hashicorp"
12.    "github.com/thanhphu/raftbench/util"
13.)
14.
15. // Command line defaults
16. const (
17.     DefaultHTTPAddr = ":11000"
```

```
18. DefaultRaftAddr = ":12000"
19.)
20.
21. // main 函数是程序的入口点。
22. // 它解析命令行参数并根据选择的 Raft 引擎启动相应的服务。
23. func main() {
24. // 配置 Raft 引擎的命令行标志
25. engine := flag.String("engine", "etcd", "etcd/hasbi/dragonb
    oat select the raft engine")
26. cluster := flag.String("cluster", "http://127.0.0.1:9021",
    "comma separated cluster peers")
27. id := flag.Int("id", 1, "node ID")
28. kvPort := flag.Int("port", 9121, "key-value server port")
29. join := flag.Bool("join", false, "join an existing cluster"
    )
30. enabled := flag.Bool("test", false, "use this node for r/w
    tests")
31. logfile := flag.String("logfile", "result.csv", "name of cs
    v log file (used together with --test)")
32. numKeys := flag.Int("numKeys", 1000, "Run the benchmark num
    Keys * mil times")
33. mil := flag.Int("mil", 1, "Run the benchmark numKeys * mil
    times")
34. runs := flag.Int("runs", 5, "Number of time to run the benc
    hmark")
35. wait := flag.Int("wait", 3000, "Time to wait before each st
    ep and before read / write")
36. firstWait := flag.Int("firstWait", 10000, "Time to wait bef
    ore starting benchmark")
37. step := flag.Int("step", 100, "If read fails, wait this muc
    h before trying to avoid overloading the system")
38. maxTries := flag.Int("maxTries", 10, "Only retry an operati
    on this many times")
39.
40. // 配置 HTTP 和 Raft 地址的命令行标志
41. var httpAddr string
42. var raftAddr string
43. var joinAddr string
44. var nodeID string
45.
```

```
46. flag.StringVar(&httpAddr, "haddr", DefaultHTTPAddr, "Set the HTTP bind address")
47. flag.StringVar(&raftAddr, "raddr", DefaultRaftAddr, "Set Raft bind address")
48. flag.StringVar(&joinAddr, "joinaddr", "", "Set join address, if any")
49. flag.StringVar(&nodeID, "nodeid", "", "Node ID")
50.
51. addr := flag.String("addr", "", "Nodehost address")
52.
53. // 自定义命令行帮助信息
54. flag.Usage = func() {
55.     log.Printf("Usage: %s [options] <raft-data-path> \n", os.Args[0])
56.     flag.PrintDefaults()
57. }
58.
59. // 解析命令行参数
60. flag.Parse()
61.
62. // 创建测试参数结构体
63. testParams := &util.TestParams{
64.     NumKeys:    *numKeys,
65.     Mil:        *mil,
66.     Runs:       *runs,
67.     Wait:       time.Duration(*wait) * time.Millisecond,
68.     FirstWait:  time.Duration(*firstWait) * time.Millisecond,
69.     Step:       time.Duration(*step) * time.Millisecond,
70.     MaxTries:   *maxTries,
71.     Enabled:    *enabled,
72.     LogFile:    *logFile,
73. }
74.
75. // 根据选择的引擎启动相应服务
76. switch *engine {
77. case "etcd":
78.     etcd.Main(*cluster, *id, *kvPort, *join, *testParams)
79. case "hashi":
80.     hashicorp.Main(httpAddr, raftAddr, joinAddr, nodeID, *testParams)
```

```

81. case "dragonboat":
82.     dragonboat.Main(*cluster, *id, *addr, *join, *testParams)
83. }
84. }

```

这段代码主要作为测试框架的入口。

命令行参数:

- 可以选择测试的 Raft 引擎 (etcd、hashi、dragonboat)。
- 集群的 HTTP 和 Raft 地址，节点 ID，以及是否加入现有集群。
- 是否用于测试，日志文件名，测试的键数量，测试次数等。
- 等待时间和重试策略，用于调节测试过程。

程序逻辑:

- 定义了默认的 HTTP 和 Raft 地址。
- 解析命令行参数并设置默认值。
- 根据选择的引擎 (etcd、hashicorp、dragonboat)，调用相应的 Main 函数进行测试。

主要功能:

- 支持对不同 Raft 引擎进行基准测试。
- 可配置测试参数，如测试轮数、等待时间、重试次数等。
- 支持将测试结果记录到 CSV 文件中，便于后续分析。

4.2 Dragonboat 测试框架

```

1. package dragonboat

```

```
2.
3. import (
4.     "context"
5.     "encoding/json"
6.     "fmt"
7.     "os"
8.     "os/signal"
9.     "path/filepath"
10.    "runtime"
11.    "strings"
12.    "syscall"
13.    "time"
14.
15.    "github.com/lni/dragonboat/v4"
16.    "github.com/lni/dragonboat/v4/config"
17.    "github.com/lni/dragonboat/v4/logger"
18.
19.    "github.com/thanhphu/raftbench/util"
20.)
21.
22.// RequestType 定义了请求操作的类型。
23.type RequestType uint64
24.
25.// exampleClusterID 是一个预定义的集群 ID，用于演示目的。
26.const (
27.    exampleClusterID uint64 = 128
28.)
29.
30.// Main 是主入口函数，用于启动一个 dragonboat 节点，并根据传入的参数
    配置节点的行为。
31.// 参数：
32.//     cluster: 集群配置字符串，包含集群成员的信息。
33.//     nodeID: 当前节点的 ID。
34.//     addr: 当前节点的地址。
35.//     join: 是否以加入现有集群的方式启动。
```

```
36. // test: 测试参数结构体。
37. func Main(cluster string, nodeID int, addr string, join bool,
    test util.TestParams) {
38. if len(addr) == 0 && nodeID != 1 && nodeID != 2 && nodeID !=
    3 {
39. fmt.Fprintf(os.Stderr, "node id must be 1, 2 or 3 when add
    ress is not specified\n")
40. os.Exit(1)
41. }
42. // 解决 macOS 下特定信号处理的问题。
43. if runtime.GOOS == "darwin" {
44. signal.Ignore(syscall.Signal(0xd))
45. }
46. initialMembers := make(map[uint64]string)
47. if !join {
48. // 解析集群配置字符串, 获取初始成员列表。
49. addresses := strings.Split(strings.Replace(cluster, "http:
    //", "", -1), ",")
50. for idx, v := range addresses {
51. initialMembers[uint64(idx+1)] = v
52. }
53. }
54. var nodeAddr string
55. if len(addr) != 0 {
56. nodeAddr = addr
57. } else {
58. nodeAddr = initialMembers[uint64(nodeID)]
59. }
60. fmt.Fprintf(os.Stdout, "节点地址: %s\n", nodeAddr)
61. // 设置日志级别。
62. logger.GetLogger("raft").SetLevel(logger.ERROR)
63. logger.GetLogger("rsm").SetLevel(logger.WARNING)
64. logger.GetLogger("transport").SetLevel(logger.WARNING)
65. logger.GetLogger("grpc").SetLevel(logger.WARNING)
66. // 初始化配置对象。
67. rc := config.Config{
68. ReplicaID:          uint64(nodeID),
69. ShardID:            exampleClusterID,
```

```
70. ElectionRTT:          10,
71. HeartbeatRTT:         1,
72. CheckQuorum:          true,
73. SnapshotEntries:      0,
74. DisableAutoCompactions: true,
75. MaxInMemLogSize:      0,
76. CompactionOverhead:   5,
77. }

78. // 指定数据目录。
79. datadir := filepath.Join(
80.     fmt.Sprintf("wal-dragonboat-%d", nodeID))
81. nhc := config.NodeHostConfig{
82.     WALDir:      datadir,
83.     NodeHostDir: datadir,
84.     RTTMillisecond: 200,
85.     RaftAddress:  nodeAddr,
86. }
87. nh, err := dragonboat.NewNodeHost(nhc)
88. if err != nil {
89.     panic(err)
90. }
91. if err := nh.StartReplica(initialMembers, join, NewMemKV, r
    c); err != nil {
92.     fmt.Fprintf(os.Stderr, "failed to add cluster, %v\n", err)
93.     os.Exit(1)
94. }
95.
96. cs := nh.GetNoOPSession(exampleClusterID)
97.
98. // 等待分片准备就绪。
99. time.Sleep(2 * time.Second)
100.
101.     ctx, _ := context.WithTimeout(context.Background(), 10
        00*time.Second)
102.
103.     // 执行基准测试。
104.     util.Bench(test, func(k string) bool {
105.         _, err := nh.SyncRead(ctx, exampleClusterID, k)
106.         return err == nil
107.     }, func(k string, v string) bool {
108.         kv := &KVData{
```

```
109.     Key: k,  
110.     Val: v,  
111. }  
112. data, err := json.Marshal(kv)  
113. if err != nil {  
114.     return false  
115. }  
116.  
117. _, err = nh.SyncPropose(ctx, cs, data)  
118. return err == nil  
119. })  
120. }
```

- 忽略特定操作系统信号，以避免程序在接收到信号时退出。
- 解析命令行参数，配置节点地址和初始成员信息。
- 设置日志记录器的不同级别，以减少不必要的日志输出。
- 定义了用于龙舟（Dragonboat）库的配置参数，如复制 ID、快照条目数等。
- 指定数据目录，为龙舟节点主机配置 WAL（Write-Ahead Logging）目录和节点主机目录。
- 创建并启动一个龙舟节点主机实例，加入到初始成员中。
- 创建一个无操作会话（NoOP Session），用于与龙舟交互。
- 等待节点准备好后，执行读写操作的性能测试。

代码的主要功能是初始化分布式状态机环境，通过 Dragonboat 库实现 Raft 协议，并对状态机进行读写操作的性能测试。

4.3 基准测试框架

```
1. package util  
2.  
3. import (  
4.     "fmt"  
5.     "log"
```

```
6.  "math/rand"
7.  "os"
8.  "time"
9. )
10.
11. type TestParams struct {
12.     NumKeys    int
13.     Mil        int
14.     Runs       int
15.     Wait       time.Duration
16.     FirstWait  time.Duration
17.     Step       time.Duration
18.     MaxTries   int
19.     Enabled    bool
20.     LogFile    string
21. }
22.
23. func Bench(testParams TestParams, read func(string) bool, write func(string, string) bool) {
24.     defer WaitForCtrlC()
25.     if !testParams.Enabled {
26.         return
27.     }
28.
29.     f, err := os.Create(testParams.LogFile)
30.     if err != nil {
31.         log.Fatal("unable to create csv log")
32.     }
33.     defer f.Close()
34.
35.     time.Sleep(testParams.FirstWait)
36.     log.Printf("Starting benchmark...\n")
37.     for i := 0; i < testParams.Runs; i++ {
38.         log.Printf("BENCHMARK %v OF %v\n", i+1, testParams.Runs)
39.         time.Sleep(testParams.Wait)
40.
41.         start := time.Now()
42.         failure := 0
43.         for k := 0; k < testParams.NumKeys*testParams.Mil; k++ {
44.             v := rand.Int()
45.             tries := 0
46.             for ok := false; !ok; ok = write(fmt.Sprintf("%d", k), fmt.Sprintf("%d", v)) {
47.                 time.Sleep(testParams.Step)
```

```

48.     tries++
49.     if tries > testParams.MaxTries {
50.         break
51.     }
52. }
53. failure += tries
54. }
55. _, _ = f.WriteString(fmt.Sprintf("write,%v,%v,%v,%v\n", i+
    1, failure, testParams.NumKeys*testParams.Mil, time.Since(st
    art).Microseconds()))
56.
57. time.Sleep(testParams.Wait)
58. start = time.Now()
59. failure = 0
60. for k := 0; k < testParams.NumKeys*testParams.Mil; k++ {
61.     tries := 0
62.     for ok := false; !ok; ok = read(fmt.Sprintf("%d", k)) {
63.         time.Sleep(testParams.Step)
64.         tries++
65.         if tries > testParams.MaxTries {
66.             break
67.         }
68.     }
69.     failure += tries
70. }
71. _, _ = f.WriteString(fmt.Sprintf("read,%v,%v,%v,%v\n", i+1,
    failure, testParams.NumKeys*testParams.Mil, time.Since(star
    t).Microseconds()))
72. }
73. log.Printf("BENCHMARK COMPLETE\n")
74. }

```

该函数主要功能如下：

- 初始化与配置：

接受一组测试参数 TestParams 及两个操作函数 read 和 write。如果 Enabled 为 false

则直接返回。创建日志文件用于记录测试结果。

- 准备与延迟：



应用初始延迟 FirstWait 后开始基准测试。

- 循环执行测试：

执行指定次数 Runs 的测试循环。每轮之间有 Wait 延迟。记录当前轮次信息。

- 写操作测试：

对每个键执行写操作，键数量为 NumKeys * Mil。生成随机值并尝试写入。

记录失败重试次数。将写操作统计数据写入日志。

- 读操作测试：

对相同的键执行读操作。记录失败重试次数。将读操作统计数据写入日志。