

ebpf程序挂载位置选取

将处理过程直接放在内核进行处理，减少了内核态和用户态的交互。

1. Message Broadcasting on TC

在 Raft 协议中，当领导者收到来自客户端的提案时需要向所有跟随者广播复制消息，这涉及到内核态和用户态的频繁切换以及内核网络堆栈的遍历，为了降低该巨大的开销，我们实现广播操作的 eBPF 程序并将其附加到 TC 钩子上，因为只有 TC 钩子可以拦截和处理传出的数据包；在附加 eBPF 程序之后，用户空间应用程序可以调用 `elec_broadcast()` 函数来通过套接字将消息广播到相关目的地。在底层，eBPF 程序使用 `bpf_clone_redirect()` 辅助函数克隆数据包，相应地修改克隆数据包的目的地地址，并将这些数据包发送出去。处理数据包丢失：考虑到数据中心很少发生数据包丢失的情况以及减少复杂性，我们采用应用级的超时重传机制来处理数据包丢失的情况。具体来说，如果领导节点在发送请求一段时间后仍未收到响应，则会重新发送请求；一旦该请求经历多次超时后，领导节点则会将会把目标节点标记为死节点，并启动故障恢复机制。

2. Fast Acknowledging in XDP

Raft 请求延迟的很大一部分来自领导节点和跟随节点之间的往返延迟。对于内核网络栈下的 Raft 协议，这种往返延迟不仅包括物理传播和传输延迟，还包括内核网络栈引起的延迟。为了减少该延迟，我们优化 follower 节点中的准备处理，直接将准备消息缓冲到内核日志中，并提前向 leader 节点确认。同时，用户空间应用程序使用 `elec_poll_message()` 函数异步轮询和使用日志中的缓冲消息。在底层，该函数调用相应的 eBPF 系统调用来分批轮询消息，分摊内核交叉开销。

xdp

xdp输入输出参数

XDP 的输入参数如下，其中 `data` 指向当前正在处理的数据包的开始位置；`data_end` 指向数据缓冲区的结束位置，`data_meta` 通常用于存储与数据包相关的额外元数据，比如时间戳或者特定的标记信息等等；`ingress_ifindex` 存储了数据包进入网络接口的索引，它通过访问 `struct xdp_rxq_info` 中的 `rxq->dev->ifindex` 来获取；`rx_queue_index` 表示接收队列的索引，它通过访问 `struct xdp_rxq_info` 中的 `rxq->queue_index` 来获取；`egress_ifindex` 存储了数据包将要离开网络接口的索引，它通过访问 `struct xdp_rxq_info` 中的 `rxq->dev->ifindex` 来获取。

```

1 struct xdp_md {
2     __u32 data;
3     __u32 data_end;
4     __u32 data_meta;
5     /*Below access go through struct xdp_raq_info */
6     __u32 ingress_ifindex; /*rq->dev->ifindex*/
7     __u32 rx_queue_index; /*raq->queue_index*/
8     __u32 egress_ifindex; /*txq->dev->ifindex */
9 };

```

XDP 的输出参数如下，其中 XDP_ABORTED 表示程序错误，需要将数据包丢弃，但是使用会使用 trace_xdp_exception 记录错误行为；XDP_DROP 表示在网卡驱动层直接将该数据包丢弃，无需在进一步处理，也就无需再耗费任何额外的资源；XDP_PASS 表示将该数据包继续送往内核的网络协议栈，和传统的处理方式一致；XDP_TX 表示将该数据包从同一块网卡返回；XDP_REDIRECT 表示将该数据包重定向到其他的网卡或CPU, 结合 AF_XDP可以将数据包直接送往用户空间。

```

1 enum xdp_action {
2     XDP_ABORTED = 0,
3     XDP_DROP,
4     XDP_PASS,
5     XDP_TX,
6     XDP_REDIRECT,
7 };

```

针对Xdp的ebpf程序流程

eBPF程序的执行流程通常遵循以下步骤，结合提到的具体场景（如“PrepareFastReply”），这一流程可以概述为：

1. 编写eBPF程序：

- 使用C或C++编写eBPF程序代码，如“PrepareFastReply”，其中包含内核空间中执行的逻辑，比如处理网络数据包、修改包内容或执行性能监控等。

2. 编译到字节码：

- 利用LLVM/Clang编译器将eBPF程序编译成字节码。这个过程中，Clang首先将C/C++代码转换为LLVM中间表示(IR)，然后进一步编译成eBPF字节码，这是专为内核执行设计的格式。

3. 验证与优化：

- 编译后的eBPF字节码会通过内核的验证器进行检查，确保程序是安全的，不会导致系统崩溃或违反安全策略。验证通过后，内核还可能对其进行一些优化。

4. 加载到内核：

- 使用如 `bpf()` 系统调用或 `libbpf` 库中的函数（如 `bpf_object__load_skeleton` 和 `bpf_object__attach_skeleton`）将eBPF对象（包括程序和相关联的maps）加载到内核中。这一步骤会分配资源、设置程序类型（如XDP、kprobe等）并将其绑定到相应的挂钩

5. 触发执行：

- 当预定义的事件发生时，eBPF程序会被触发执行。例如，在XDP（eXpress Data Path）场景中，当网络数据包到达时，“PrepareFastReply”程序会在内核中自动运行，处理数据包并作出相应动作，如修改包内容或决定包的转发路径。

6. 数据交互：

- eBPF程序可以在内核空间执行期间读写eBPF maps（一种特殊的内核内存区域）。这些maps允许用户空间程序和内核空间eBPF程序共享数据。例如，“PrepareFastReply”可能会更新或查询之前提到的映射表来存储或检索状态信息。

7. 用户空间交互：

- 处理完数据或执行完任务后，eBPF程序可以通过ring buffer等机制向用户空间进程发送事件或数据。用户空间程序监听这些事件，进一步处理或展示结果。

8. 卸载与清理：

- 当不再需要eBPF程序时，可以通过相应的API卸载它，释放之前分配的资源。

针对“PrepareFastReply”的具体执行流程，就是按照上述步骤，首先将处理Paxos协议快速响应的逻辑编码并编译为eBPF字节码，然后加载到内核中，在网络数据包到达并触发相应的eBPF程序时执行快速响应的逻辑，包括修改包内容、更新状态等，最后通过eBPF maps或ring buffer与用户空间程序通信，完成整个处理流程。

ebpf函数 `PrepareFastReply_main` 解析

该函数 `PrepareFastReply_main` 是一个用C++编写的eBPF（Extended Berkeley Packet Filter）程序的核心部分，设计用于在内核空间直接处理网络数据包，以实现高性能的网络处理逻辑。具体功能步骤如下：

1. 数据检查

- **作用：**确保可以安全访问数据包内容。
- **实现：**通过 `xdp_md` 结构体提供的 `data` 和 `data_end` 指针转换为可直接访问的指针，分别代表数据包的起始位置和结束位置。

2. 解析网络头部

- **以太网头部**：定位到数据包的以太网头部。
- **IP头部**：基于以太网头部偏移量找到IP头部。
- **UDP头部**：在IP头部之后找到UDP头部。
- **有效载荷**：确定数据包的有效数据部分起始位置。

3. 检查数据包长度

- **目的**：验证数据包是否足够大以容纳特定的Paxos协议字段。
- **操作**：计算有效载荷加上预期附加字段的总长度，与数据包实际长度比较，若不足则直接放过此包 (`XDP_PASS`)。

4. 读取状态信息

- **操作**：从eBPF映射表 (`map_msg_lastOp` , `map_ctr_state`) 中查找当前状态信息，包括最近的操作编号 (`msg_lastOp`) 和计数器状态 (`ctr_state`)。
- **错误处理**：找不到对应信息时直接放过此包。

5. 获取Leader信息

- **来源**：从另一个eBPF映射表 (`map_configure`) 中根据计数器状态中的领导索引 (`leaderIdx`) 获取领导节点信息。
- **错误处理**：找不到Leader信息则放过此包。

6. 构建回复内容

- **修改有效载荷**：写入特定的标识符、消息类型、修改协议字符串为"MyPrepareOK"，以及插入状态信息如视图号、操作编号和发送者的索引等。
- **确保空间足够**：在写入每个字段前都检查剩余空间是否足够，不足则放弃处理。

7. 更新网络层头部

- **UDP**：修改UDP的源端口为目标端口，计算并更新UDP数据长度和校验和（此处未计算校验和，注释表明计算不是必须的）。
- **IP**：调整IP头部的源IP、目标IP、总长度和校验和（计算IP校验和）。

8. 调整以太网头部

- **交换MAC地址**：将源MAC地址与目标MAC地址互换，以准备数据包回传。

9. 数据包处理决策

- **调整数据包尾部**：使用 `bpf_xdp_adjust_tail` 调整数据包的长度，确保所有更改被正确反映。
- **返回指令**：最后函数返回 `XDP_TX`，指示数据包应被发送到网络接口进行转发，即作为对原Paxos prepare消息的快速响应。

综上所述，该函数负责在接收到Paxos协议的Prepare消息后，快速构造并准备一个同意（MyPrepareOK）的回复消息，同时处理所有必要的网络包头修改，以确保回复能准确无误地发回给Paxos集群的领导者。

TC

TC 的输入参数如下，这个结构是一种UAPI（User space API of the kernel），允许访问内核中 socket buffer 内部数据结构中的某些字段。它具有与 struct xdp_md 相同意义的两个指针 data 和 data_end，同时还有更多的信息可以获取，因为在TC 层面上，内核已经解析了数据包以提取与协议相关的元数据，因此传递给BPF 程序的上下文信息更丰富。

Listing 16 TC 输入参数

```
struct __sk_buff {
    __u32 len;
    __u32 pkt_type;
    __u32 mark;
    __u32 queue_mapping;
    __u32 protocol;
    __u32 vlan_present;
    __u32 vlan_tci;
    __u32 vlan_proto;
    __u32 priority;
    __u32 ingress_ifindex;
    __u32 ifindex;
    __u32 tc_index;
    __u32 cb[5];
    __u32 hash;
    __u32 tc_classid;
    __u32 data;
    __u32 data_end;
    __u32 napi_id;

    /* Accessed by BPF_PROG_TYPE_sk_skb types from here to ... */
    __u32 family;
    __u32 remote_ip4;    /* Stored in network byte order */
    __u32 local_ip4;     /* Stored in network byte order */
    __u32 remote_ip6[4]; /* Stored in network byte order */
    __u32 local_ip6[4];  /* Stored in network byte order */
    __u32 remote_port;   /* Stored in network byte order */
    __u32 local_port;    /* stored in host byte order */
    /* ... here. */

    __u32 data_meta;
    __bpf_md_ptr(struct bpf_flow_keys *, flow_keys);
    __u64 tstamp;
    __u32 wire_len;
    __u32 gso_segs;
    __bpf_md_ptr(struct bpf_sock *, sk);
};
```

和XDP一样，TC的输出代表了数据包如何被处置的一种动作，如下是常用的5种动作：

Value	Action	Description
0	TC_ACT_OK	Delivers the packet in the TC queue.
2	TC_ACT_SHOT	Drop packet.
-1	TC_ACT_UNSPEC	Uses standard TC action.
3	TC_ACT_PIPE	Performs the next action, if it exists.
1	TC_ACT_RECLASSIFY	Restarts the classification from the beginning.

Figure 3.1: TC 输出参数

函数解析

该C++函数 `FastBroadcast_main` 是一个高度定制化的数据包处理函数，专为Paxos分布式一致性协议设计，用于在内核态（可能是使用eBPF即Extended Berkeley Packet Filter技术）高效地处理和转发特定格式的UDP数据包。以下是其详细步骤分析：

- 数据检查与指针初始化
 - 数据指针定位：**首先，通过 `skb->data_end` 和 `skb->data` 获取数据包的结束和起始指针，并转换为可操作的指针形式。
- 解析网络头部
 - 解析以太网头、IP头、UDP头：**分别从数据包中偏移量处找到以太网头部(`ethhdr`)、IP头部(`iphdr`)、UDP头部(`udphdr`)的位置。
 - 有效载荷定位：**计算出数据包的有效载荷部分的起始位置。
- 数据包类型与头部检查
 - 基本验证：**确保数据包结构完整，检查IP协议类型是否为UDP，并且UDP头部在数据包范围内。
 - Paxos协议验证：**进一步验证UDP源端口是否为12345，这是预设的Paxos协议端口，并检查Paxos协议特有的“魔术字节”及类型长度字段。
- Paxos协议特殊字段处理
 - 协议字段检查：**深入检查Paxos协议的特定字段，如“魔术位”和类型长度，确保它们符合预期格式。
 - 消息内容定位：**根据协议结构，定位到实际消息内容的开始位置。
- 处理广播消息

- **提取关键信息：**从消息中提取视图号(`msg_view`)和广播标志位(`is_broadcast`)，并据此调整视图号。
- **消息类型计算：**基于消息类型字符串计算消息类型。

6. 更新并转发消息

- **状态更新逻辑：**对于特定类型的消息（如FAST_PROG_XDP_HANDLE_PREPARE），更新维护的Quorum状态映射中的条目。
- **广播逻辑处理：**如果消息标识为广播，根据当前集群状态动态修改消息类型字符串，以指示多播，并可能触发数据包的多播转发（通过调用`bpf_clone_redirect`）。
- **数据包头重写：**为了适应多播或转发需求，函数还会根据需要更新UDP目标端口、IP目标地址、校验和以及以太网目标地址等字段，确保数据包能正确送达目的地。

7. 错误与边界检查

- 在每个关键处理步骤后，都有边界检查来确保不会访问到数据包之外的内存，以防止安全问题。

总结

此函数是一个复杂的内核态数据包处理器，针对特定Paxos协议设计，涉及深度的数据包解析、协议合规性检查、状态更新以及智能的多播转发逻辑，体现了在网络协议栈底层进行高效、精确数据处理的能力。