[译] Facebook 流量路由最佳实践：从公网入口到内网业务的全路径 XDP/BPF 基础设施（LPC, 2021）

Published at 2021-12-05 | Last Update 2021-12-11

译者序

本文翻译自 Facebook 在 LPC 2021 大会上的一篇分享： From XDP to Socket: Routing of packets beyond XDP with BPF。

标题可直译为**《从 XDP 到 Socket 的（全路径）流量路由：XDP 不够，BPF 来凑》**，因为 XDP 运行 在网卡上，而且在边界和流量入口，再往后的路径（尤其是到了内核协议栈）它就管不 到了，所以引入了其他一些 BPF 技术来“接力”这个路由过程。另外， 这里的“路由”并非狭义的路由器三层路由，而是泛指 L3-L7 流量转发。

翻译时加了一些链接和代码片段，以更方便理解。

由于译者水平有限，本文不免存在遗漏或错误之处。如有疑问，请查阅原文。

以下是译文。

1 引言

用户请求从公网到达 Facebook 的边界 L4LB 节点之后，往下会涉及到两个阶段（每个阶 段都包括了 L4/L7）的流量转发：

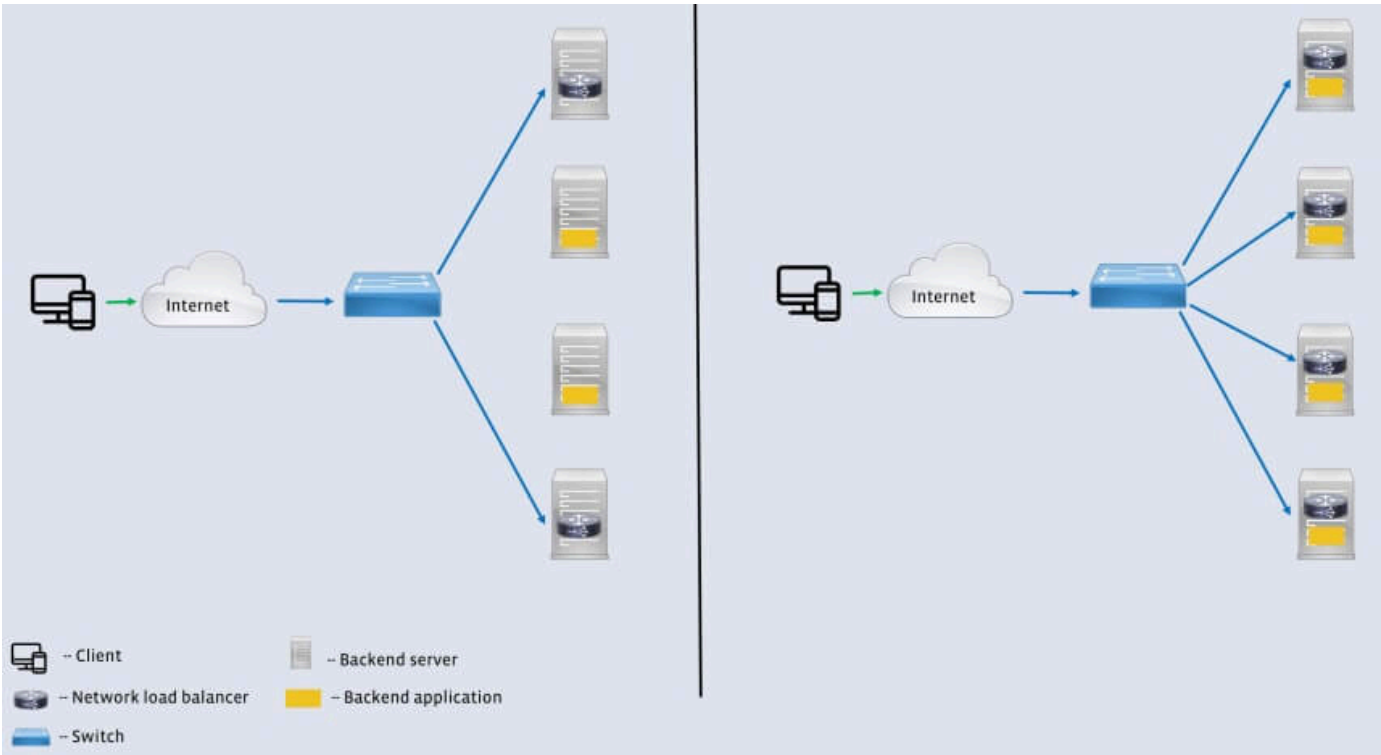
1. 从 LB 节点负载均衡到特定主机
2. 主机内：将流量负载均衡到不同 socket

以上两个阶段都涉及到流量的一致性路由（consistent routing of packets）问题。 本文介绍这一过程中面临的挑战，以及我们如何基于最新的 BPF/XDP 特性来应对这些挑战。

1.1 前期工作

几年前也是在 LPC 大会，我们分享了 Facebook **基于 XDP 开发的几种服务**，例如

1. 基于 XDP 的**四层负载均衡器（L4LB）** katran， 从 2017 年开始，每个进入 facebook.com 的包都是经过 XDP 处理的；
2. 基于 XDP 的**防火墙**（挡在 katran 前面）。

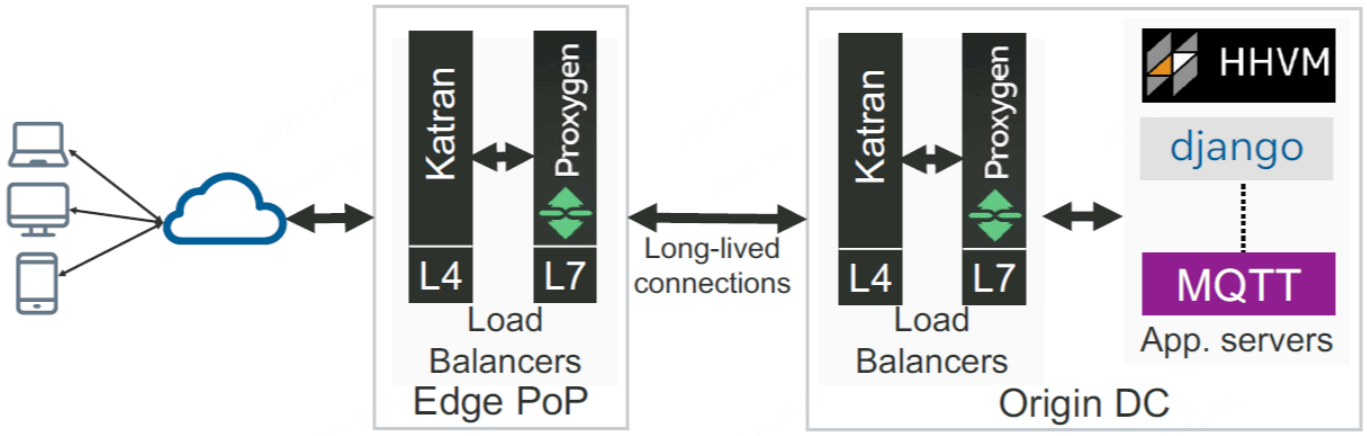


Facebook 两代软件 L4LB 对比。
左：第一代，基于 IPVS，L4LB 需独占节点；右：第二代，基于 XDP，不需独占节点，与业务后端混部。

1.2 Facebook 流量基础设施

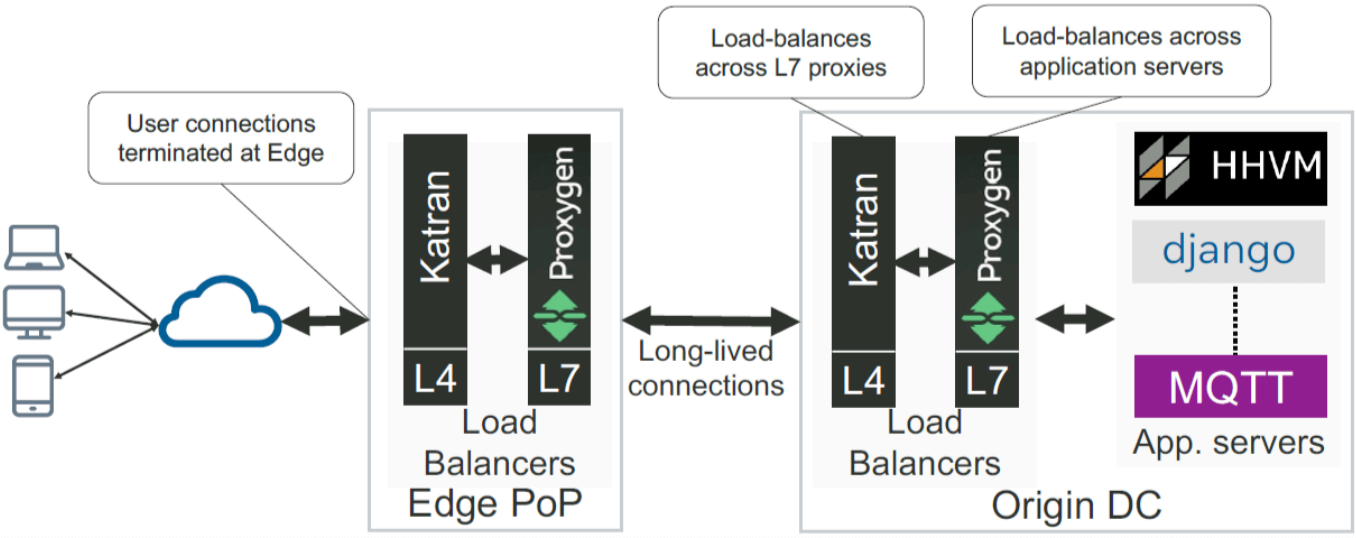
从层次上来说，如下图所示，Facebook 的流量基础设施分为两层：

1. **边界层**（edge tiers），位于 PoP 点
2. **数据中心层**，我们称为 Origin DC



- 每层都有一套**全功能 LB（L4+L7）**
- Edge PoP 和 Origin DC 之间的 LB 通常是长链接

从功能上来说，如下图所示：



1. **用户连接**（user connections）**在边界终结**，
2. Edge PoP LB 将 L7 流量路由到终端主机，
3. Origin DC LB 再将 L7 流量路由到最终的应用，例如 HHVM 服务。

1.3 面临的挑战

总结一下前面的内容：公网流量到达边界节点后，接下来会涉及 **两个阶段的流量负载均衡**（每个阶段都是 L4+L7），

1. 宏观层面：**LB 节点 -> 后端主机**
2. 微观层面（主机内）：**主机内核 -> 主机内的不同 socket**

这两个阶段都涉及到流量的高效、**一致性路由**（consistent routing）问题。

本文介绍这一过程中面临的挑战，以及我们是如何基于最新的 BPF/XDP 特性 来解决这些挑战的。具体来说，我们用到了两种类型的 BPF 程序：

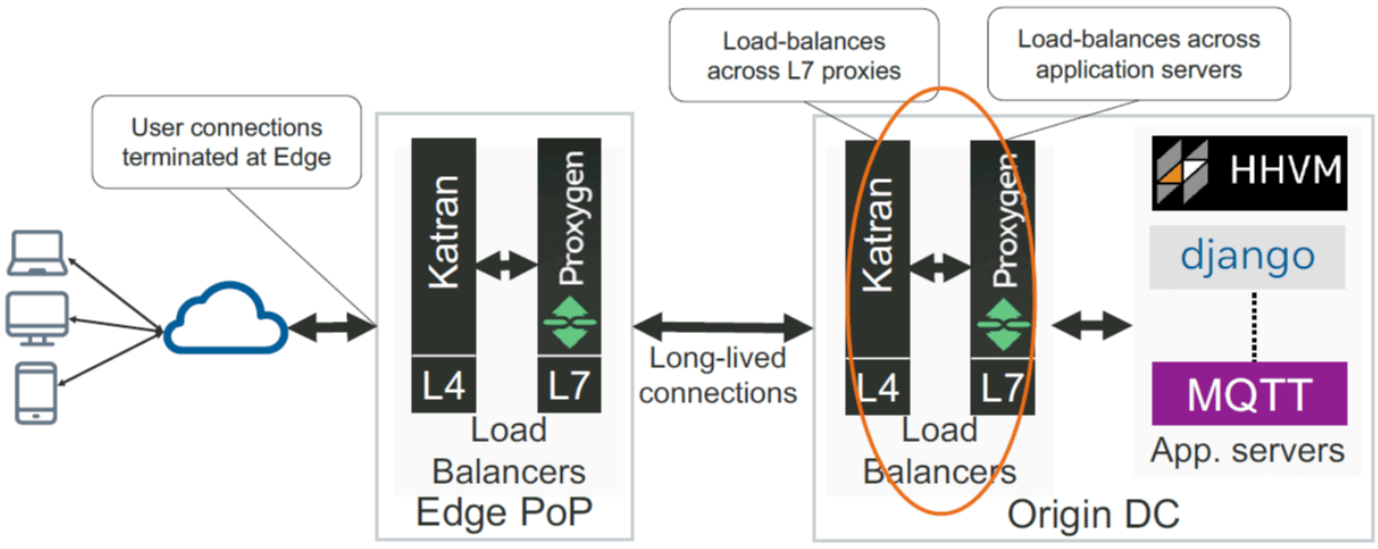
1. BPF TCP header options：解决主机外（宏观）负载均衡问题；
2. `BPF_PROG_TYPE_SK_REUSEPORT` （及相关 map 类型 `BPF_MAP_TYPE_REUSEPORT_SOCKARRAY` ）：解决主机内（微观）负载均衡问题。

2 选择后端主机：数据中心内流量的一致性与无状态路由（四层负载均衡）

先看第一部分，从 LB 节点转发到 backend 机器时，如何来选择主机。 这是四层负载均衡问题。

2.1 Katran (L4LB) 负载均衡机制

回到流量基础设施图，这里主要关注 Origin DC 内部 L4-L7 的负载均衡，



katran 是基于 XDP 实现的四层负载均衡器，它的内部机制：

- 实现了一个 **Maglev Hash** 变种，通过一致性哈希选择后端；

- 在一致性哈希之上，还维护了自己的一个**本地缓存**来跟踪连接。 这个设计是为了在**某些后端维护或故障时，避免其他后端的哈希发生变化**，后面会详细讨论。

用伪代码来表示 Katran **选择后端主机的逻辑**：

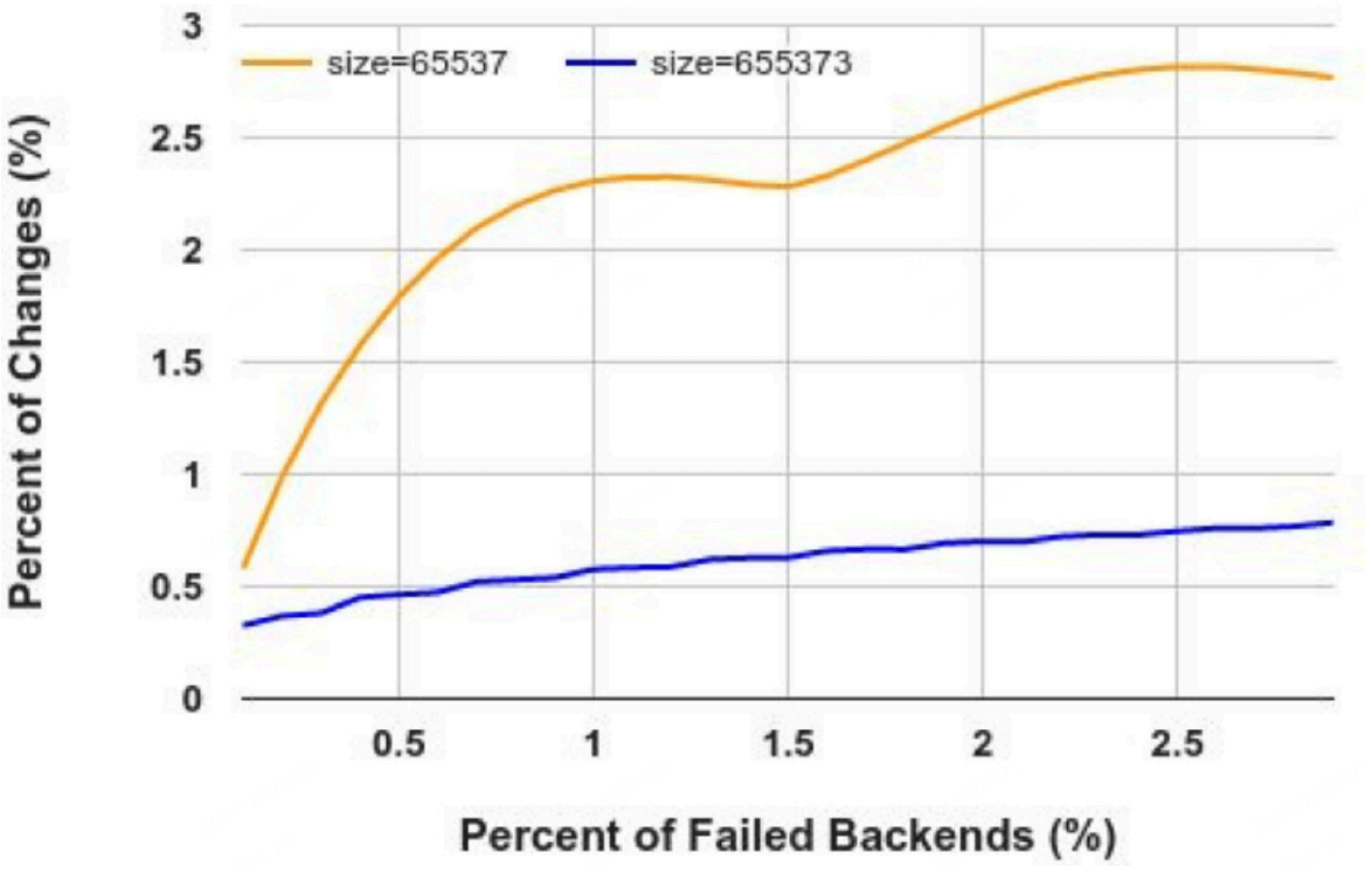
```
int pick_host(packet* pkt) {
    if (is_in_local_cache(pkt))
        return local_cache[pkt]

    return consistent_hash(pkt) % server_ring
}
```

这种机制非常有效，也非常高效（highly effective and efficient）。

2.2 一致性哈希的局限性

2.2.1 容错性：后端故障对非相关连接的扰动

一致性哈希的一个核心特性是具备**对后端变化的容错性**（resilience to backend changes）。 当一**部分后端发生故障时，其他后端的哈希表项不受影响**（因此对应的连接及主机也不受影响）。 Maglev 论文中已经给出了评估这种容错性的指标，如下图，



Resilience of Maglev hashing to backend changes

> Maglev: A fast and reliable software network load balancer. OSDI 2016

- 横轴表示 **backend 挂掉的百分比**
- 纵轴是**哈希表项（entries）变化**的百分比，对应**受影响连接**的百分比

Google 放这张图是想说明：一部分后端发生变化时，其他后端受影响的概率非常小； 但从我们的角度来说，以上这张图说明：即使后端挂掉的比例非常小， **整个哈希表还是会受影响，并不是完全无感知** —— 这就会 **导致一部分流量被错误路由**（misrouting）：

- 对于**短连接**来说，例如典型的 HTTP 应用，这个问题可能**影响不大**；
- 但对于 **tcp 长连接**，例如持续几个小时的视频流，这种扰动就不能忍了。

2.2.2 TCP 长连接面临的问题

首先要说明，**高效 != 100% 有效。** 对于 TCP 长连接来说（例如视频），有两种场景会它们**被 reset**：

```
int pick_host(packet* pkt) {
    if (is_in_local_cache(pkt))                // 场景一：ECMP shuffle 时（例如 LB 节点维护或
        return local_cache[pkt]

    return consistent_hash(pkt) % server_ring // 场景二：后端维护或故障时，这里的好像有（较小
}
```

解释一下：

1. 如果 **LB 升级、维护或发生故障**，会导致路由器 ECMP shuffle，那原来路由到某个 LB 节点的 flow，可能会被重新路由到另一台 LB 上；虽然我们维护了 cache，但它是 **LB node local** 的，因此会发生 cache miss；
2. 如果**后端节点升级、维护或发生故障**，那么根据前面 maglev 容错性的实验结果，会有一 部分（虽然比例不是很大）的 flow 受到影响，导致路由错误。

以上分析可以看出，“持续发布” L4 和 L7 服务会导致连接不稳定，降低整体可靠性。 除了发布之外，我们随时都有大量服务器要维护，因此哈希 ring 发生变化（一致性哈希 发生扰动）是日常而非例外。任何时候发生 ECMP shuffle 和服务发布/主机维护，都会导 致一部分 active 连接受损，虽然量很小，但会降低整体的可靠性指标。

解决这个问题的一种方式是**在所有 LB 节点间共享这个 local cache** （类似于 L4LB 中的 session replication），但这是个**很糟糕的主意** ，因为这就需要去解决另外一大堆分布式系统相关的问题，尤其我们不希望引入任何 会降低这个极快数据路径性能的东西。

2.2.3 QUIC 协议为什么不受影响

但对于 QUIC 来说，这都不是问题。

`connection_id`

QUIC 规范（RFC 9000）中允许 server 将任意信息嵌入到包的 `connection_id` 字段。

Facebook 已经广泛使用 QUIC 协议，因此在 Facebook 内部，我们可以

1. 在 server 端将路由信息（routing information）嵌入到 `connection_id` 字段，并
2. 要求客户端必须将这个信息带回来。

完全无状态四层路由

这样整条链路上都可以从包中提取这个 id，无需任何哈希或 cache 查找，最终实现的是一个 **完全无状态的四层路由**（completely stateless routing in L4）。

那能不能为 TCP 做类似的事情呢？答案是可以。这就要用到 BPF-TCP header option 了。

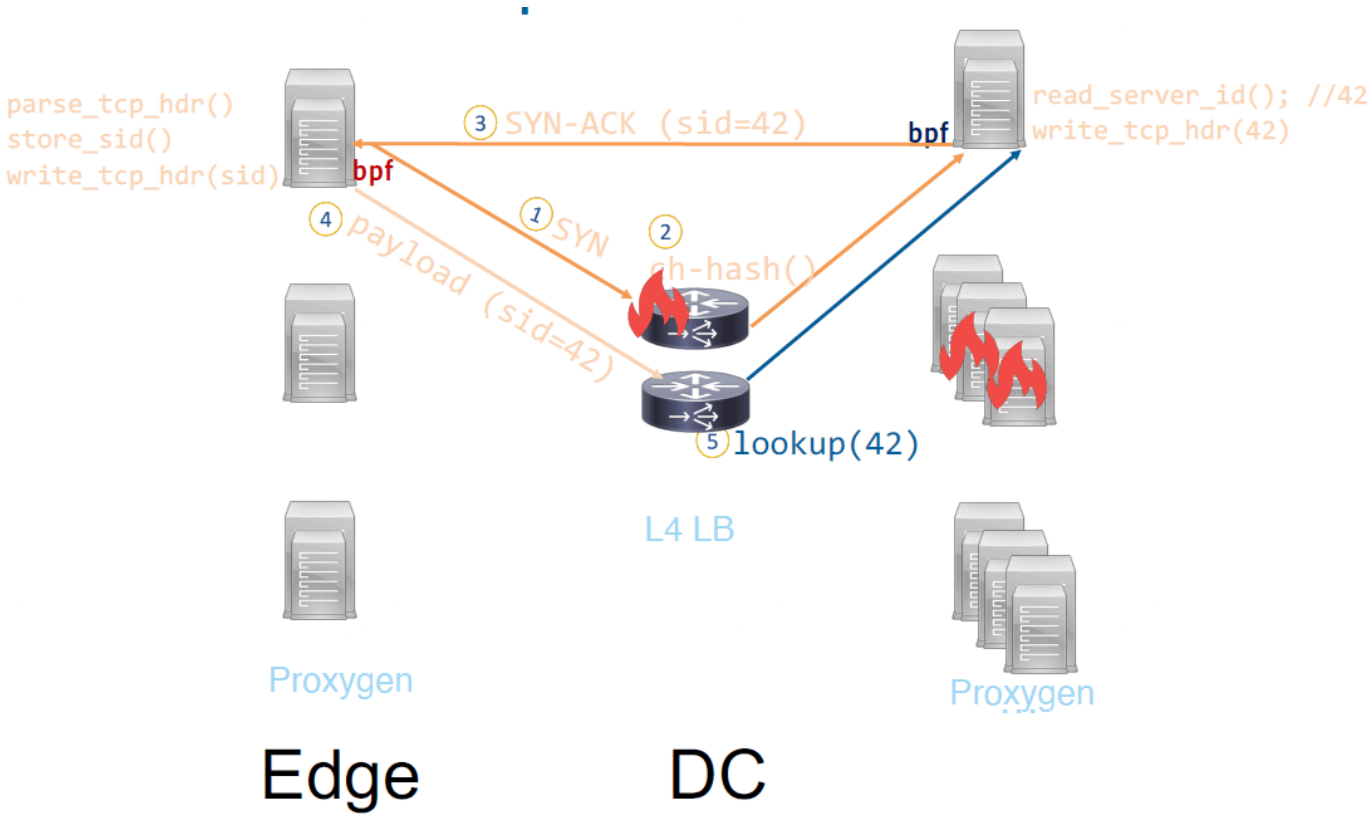
2.3 TCP 连接解决方案：利用 BPF 将 backend server 信息嵌入 TCP Header

2.3.1 原理和流程

基本思想：

1. 编写一段 `BPF_PROG_TYPE_SOCK_OPS` 类型的 BPF 程序，**attach 到 cgroup**：
   - 在 LISTEN, CONNECT, CONN_ESTD 等事件时会触发 BPF 程序的执行
   - BPF 程序可以获取包的 TCP Header，然后往其中写入路由信息（这里是 server_id），或者从中读取路由信息
2. 在 L4LB 侧维护一个 server_id 缓存，记录仍然存活的 backend 主机

以下图为例，我们来看下 LB 节点和 backend 故障时，其他 backend 上的原有连接如何做到不受影响：



- 1) 客户端发起一个 SYN；
- 2) L4LB 第一次见这条 flow，因此通过一致性哈希为它选择一台 backend 主机，然后将包转发过去；
- 3) 服务端应答 SYN+ACK，其中 **服务端 BPF 程序将 server_id 嵌入到 TCP 头中**；
  - 图中这台主机获取到自己的 server_id 是 42，然后将这个值写到 TCP header；
  - 客户端主机收到包后，会解析这个 id 并存下来，后面发包时都会带上这个 server_id；

假设过了一会发生故障，前面那台 L4LB 挂了（这会导致 **ECMP 发生变化**）； 另外，某些 backend hosts 也挂了（这会 **影响一致性哈希**，原有连接接下来有小概率会受到影响），那么接下来，

- 4) 客户端流量将被（数据中心基础设施）转发到另一台 L4LB；
- 5) 这台新的 L4LB 解析客户端包的 TCP header，提取 server_id，**查询 server_id 缓存**（注意不是 Katran 的 node-local 连接缓存）之后发现 **这台机器还是 active 的**，因此直接转发给这台机器。

可以看到在 TCP Header 中引入了路由信息后，未发生故障的主机上的长连接就能够避免 因 L4LB 和主机挂掉而导致的 misrouting（会被直接 reset）。

2.3.2 开销

数据开销：TCP header 增加 6 个字节

```
struct tcp_opt {
    uint8_t  kind;
    uint8_t  len;
    uint32_t server_id;
}; // 6-bytes total
```

运行时开销：不明显

需要在 L4LB 中解析 TCP header 中的 server_id 字段，理论上来说，这个开销跟代码实 现的好坏相关。我们测量了自己的实现，这个开销非常不明显。

2.3.3 实现细节

监听的 socket 事件

```
switch (skops->op) {
    case BPF_SOCK_OPS_TCP_LISTEN_CB:
    case BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB:
    case BPF_SOCK_OPS_TCP_CONNECT_CB:
    case BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB:
    case BPF_SOCK_OPS_PARSE_HDR_OPT_CB:
    case BPF_SOCK_OPS_HDR_OPT_LEN_CB:
    case BPF_SOCK_OPS_WRITE_HDR_OPT_CB:
    . . .
}
```

维护 TCP flow -> server_id 的映射

在每个 LB 节点上用 bpf_sk_storage 来存储 **per-flow server_id**。 也就是说，

1. 对于建连包特殊处理，
2. 建连之后会维护有 flow 信息（例如连接跟踪），
3. 对于建连成功后的普通流量，从 flow 信息就能直接映射到 server_id， **不需要针对每个包去解析 TCP header**。

server_id 的分配和同步

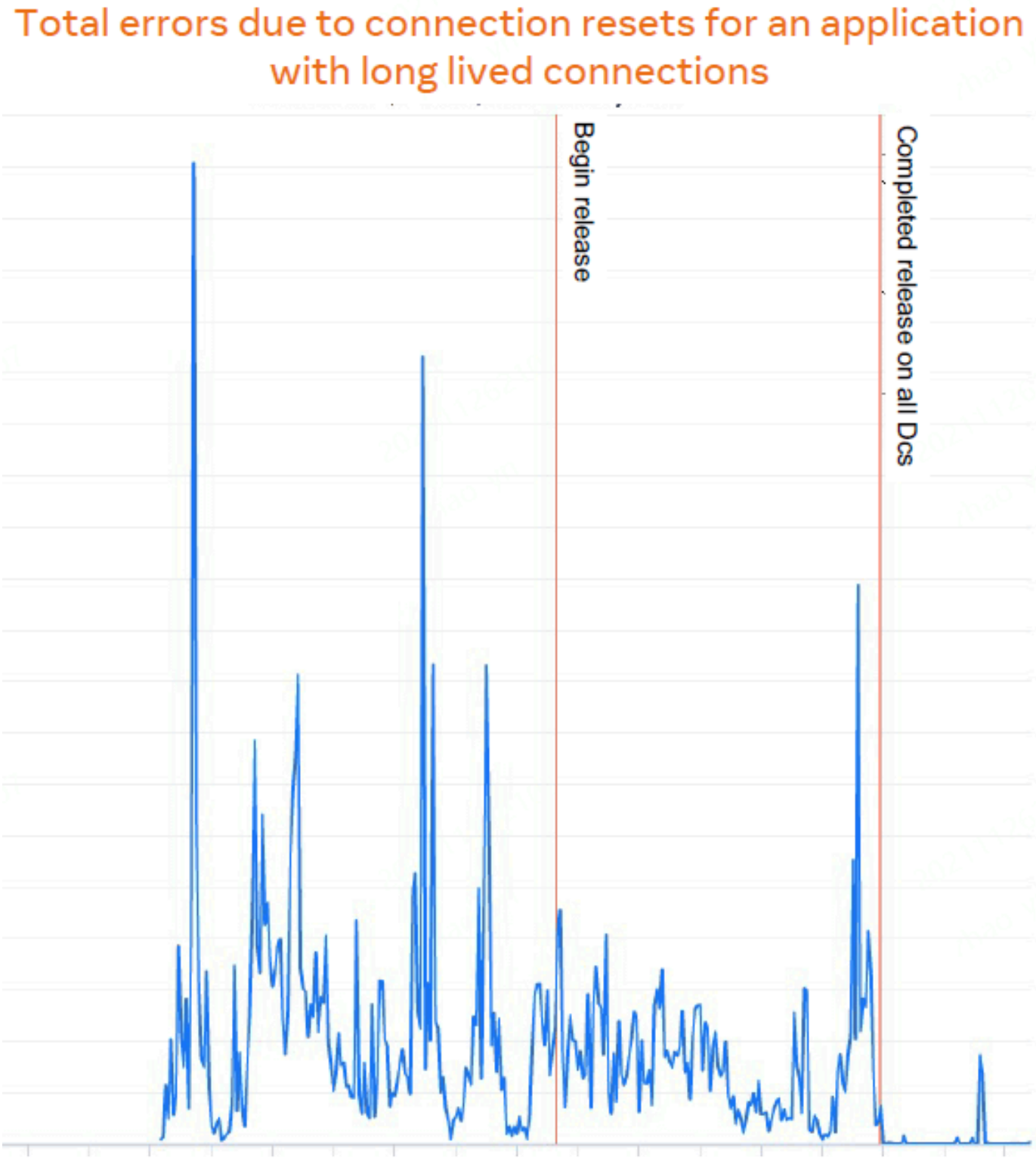
前面还没有提到**如何分配 server_id**，以及如何保证这些后端信息在负 载均衡器侧的时效性和有效性。

我们有一个 **offline 工作流**，会给那些有业务在运行的主机随机分配 一个 id，然后将这个信息**同步给 L4 和 L7 负载均衡器**（Katran and Proxygen）， 后者拿到这些信息后会将其加载到自己的控制平面。因此这个系统不会有额外开销，只要 **保证 LB 的元信息同步**就行了。

由于这个机制同时适用于 QUIC 和 TCP，因此 pipeline 是同一个。

2.3.4 效果

下面是一次发布，可以看到发布期间 connection reset 并没有明显的升高：



2.3.5 限制

这种方式要求 TCP **客户端和服务端都在自己的控制之内**，因此

- 对典型的**数据中心内部访问**比较有用；

- 要用于数据中心外的 TCP 客户端，就要让后者将带给它们的 server_id 再带回来，但这个基本做不到；

  即使它们带上了，**网络中间处理节点（middleboxes）和防火墙（firewalls）**也可能会将这些信息丢弃。

2.4 小结

通过将 server_id 嵌入 TCP 头中，我们实现了一种 stateless routing 机制，

- 这是一个**完全无状态**的方案
- 额外开销（CPU / memory）非常小，基本感知不到
- 其他竞品方案都非常复杂，例如在 hosts 之间共享状态，或者将 server_id 嵌入到 ECR (Echo Reply) 时间戳字段。

3 选择 socket：服务的真正优雅发布（七层负载均衡）

前面介绍了流量如何从公网经过内网 LB 到达 backend 主机。 再来看在主机内，如何路由流量来保证七层服务（L7 service）发布或重启时不损失任何流量。

这部分内容在 SIGCOMM 2020 论文中有详细介绍。想了解细节的可参考：

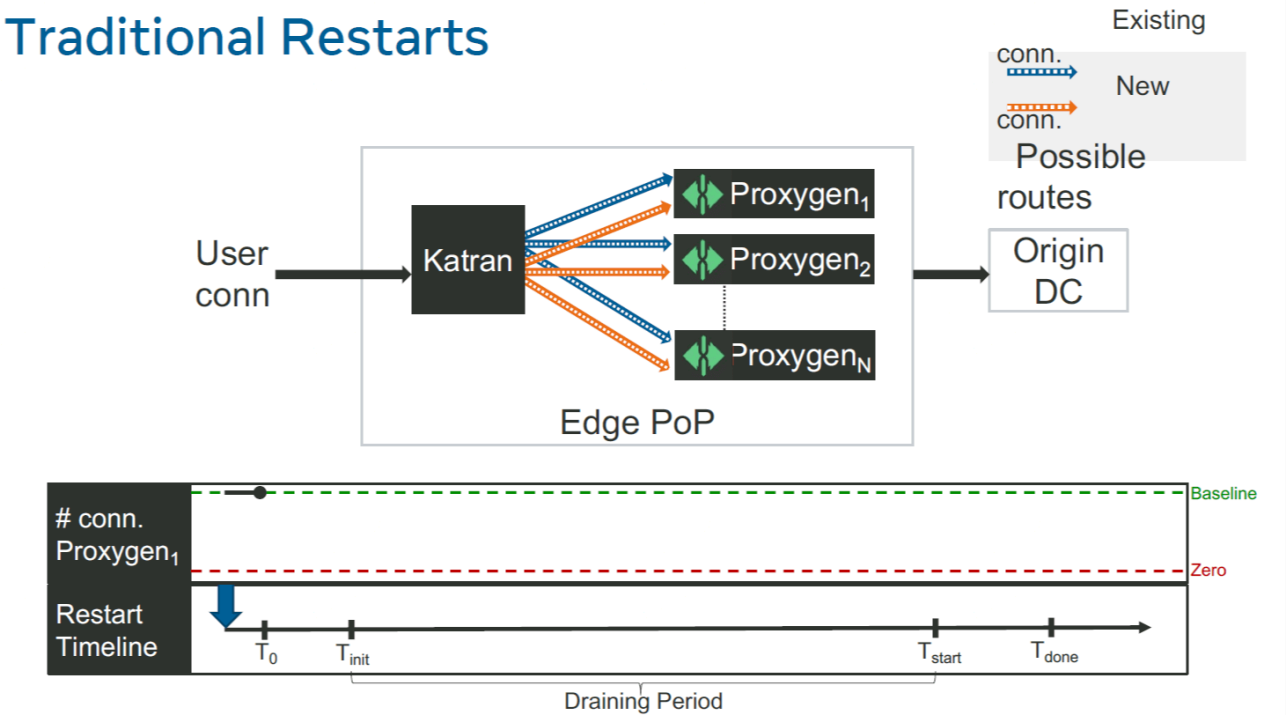
> Facebook, Zero Downtime Release: Disruption-free Load Balancing of a Multi-Billion User Website. SIGCOMM 2020

3.1 当前发布方式及存在的问题

L7LB Proxygen 自身也是一个七层服务，我们以它的升级为例来看一下当前发布流程。

3.1.1 发布流程

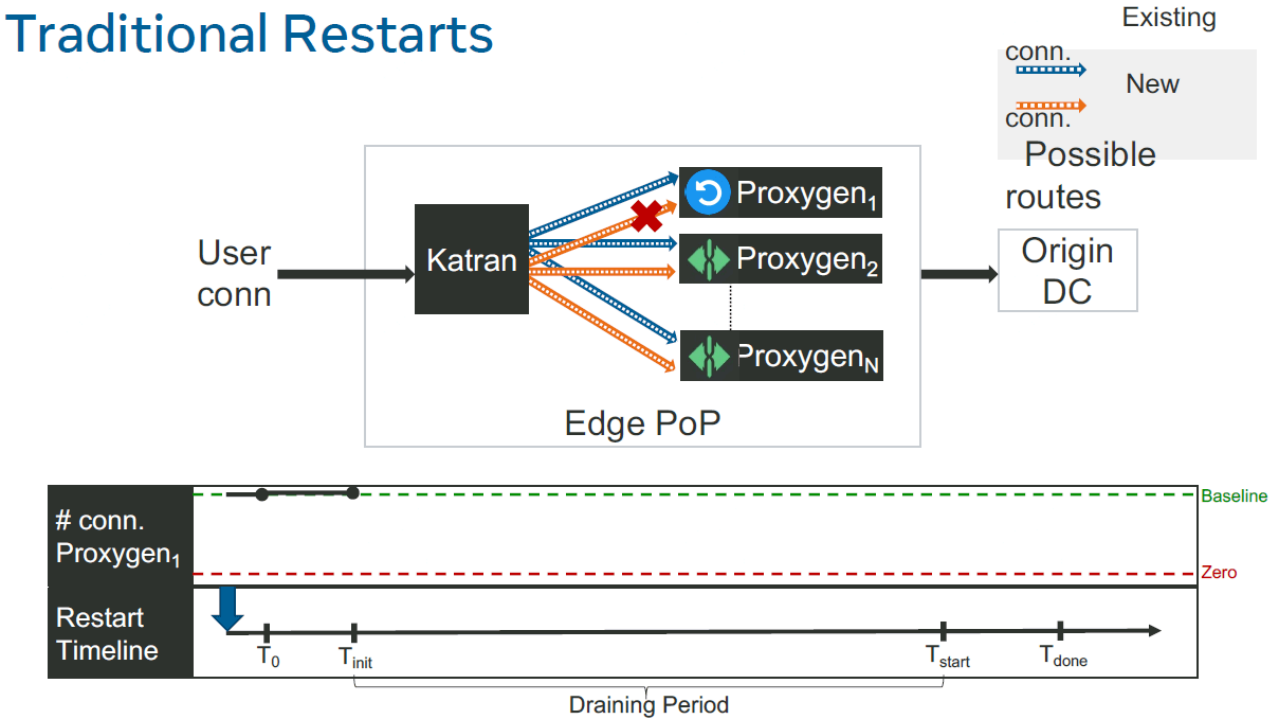
1. **发布前状态**：Proxygen 实例上有一些老连接，也在不断接受新连接，



2. **拉出**：拉出之后的实例不再接受新连接，但在一定时间窗口内，继续为老连接提供服务；

   i. 这个窗口称为 **graceful shutdown（也叫 draining） period**，例如设置为 5 或 10 分钟；

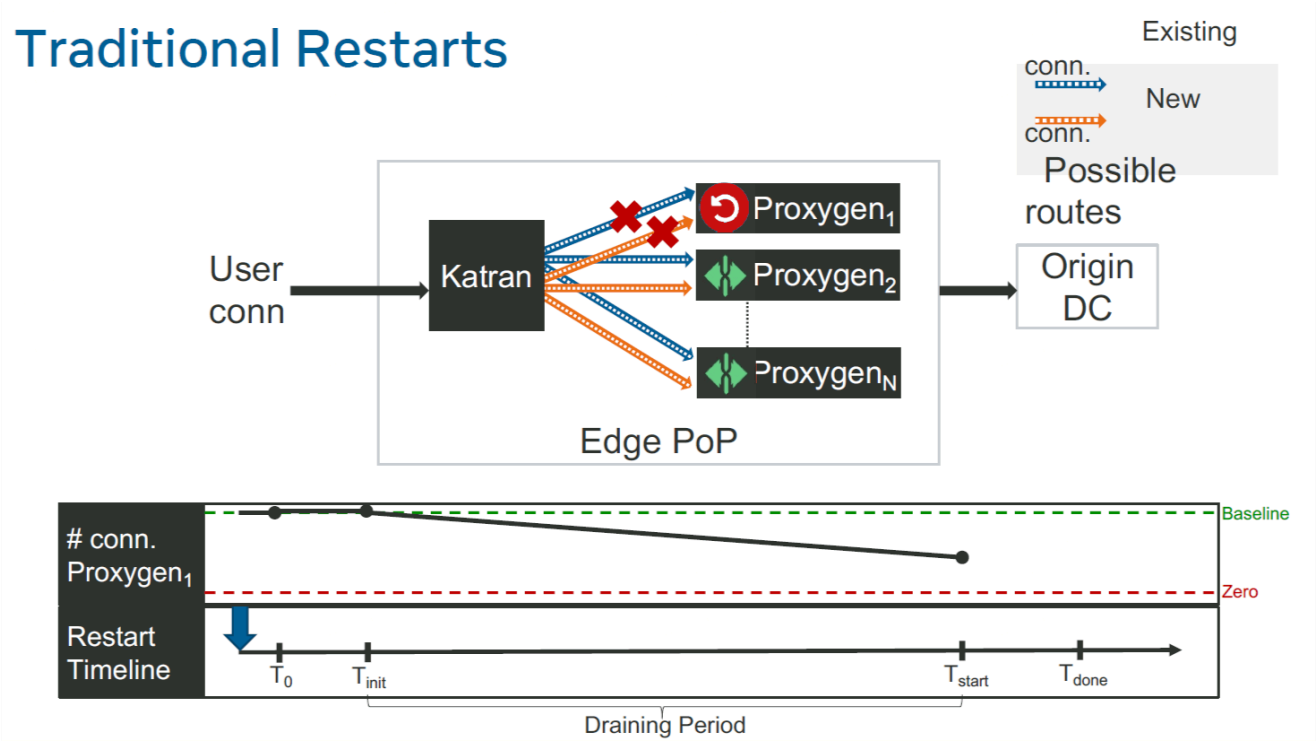
   ii. 拉出一般是通过**将 downstream service 的健康监测置为 false** 来实现的，例如在这个例子中，就是让 Proxygen 返回给 katran 的健康监测是失败的。



3. **发布新代码**：graceful 窗口段过了之后，不管它上面还有没有老连接，直接开始升级。

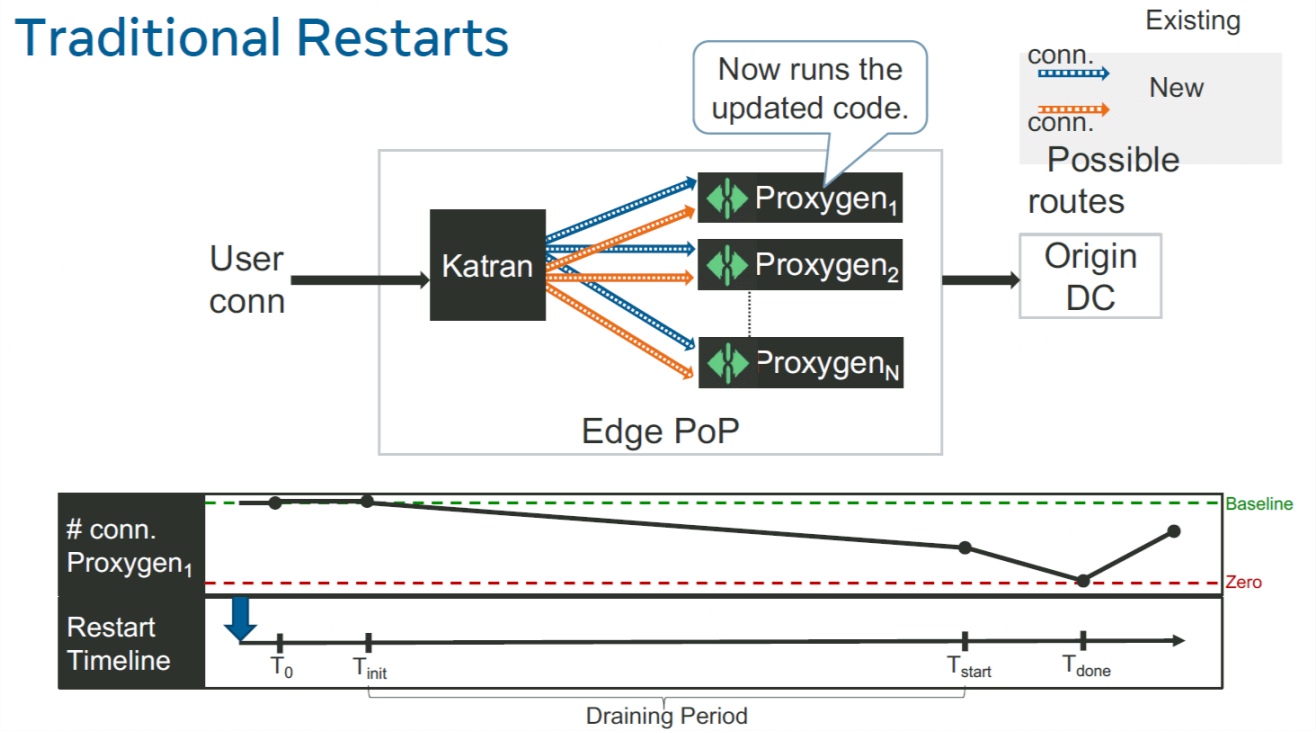
   i. 部署新代码，

   ii. 关闭现有进程，创建一个新进程运行新代码。



一般来说，只要 graceful 时间段设置比较合适，一部分甚至全部老连接能够在这个 窗口内正常退出，从而不会引起用户可见的 spike；但另一方面，如果此时仍然有老 连接，那**这些客户端就会收到 tcp reset。**

4. **监听并接受新连接**：升级之后的 Proxygen 开始正常工作， 最终达到和升级之前同等水平的一个连接状态。



3.1.2 存在的问题

很多公司都是用的以上那种发布方式，它的实现成本比较低，但也存在几个问题：

1. 发布过程中，**系统容量会降低。**

   从 graceful shutdown 开始，到新代码已经接入了正常量级的流量，这段时间内 **系统容量**并没有达到**系统资源所能支撑的最大值**， 例如三个 backend 本来最大能支撑 3N 个连接，那在升级其中一台的时间段内，系统能支撑的最大连接数就会小于 3N，在 2N~3N 之间。 这也是为什么很多公司都避免在业务高峰（而是选择类似周日凌晨五点这样的时间点）做这种变更的原因之一。

2. **发布周期太长**

   假设有 100 台机器，分成 100 个批次（phase），每次发布一台， 如果 graceful time 是 10 分钟，一次发布就需要 1000 分钟，显然是不可接受的。

   本质上来说，这种方式**扩展性太差**，主机或实例数量一多效率就非常低了。

3.2 不损失容量、快速且用户无感的发布

以上分析引出的核心问题是：如何在用户无感知的前提下，不损失容量（without losing capacity）且非常快速（very high velocity）地完成发布。

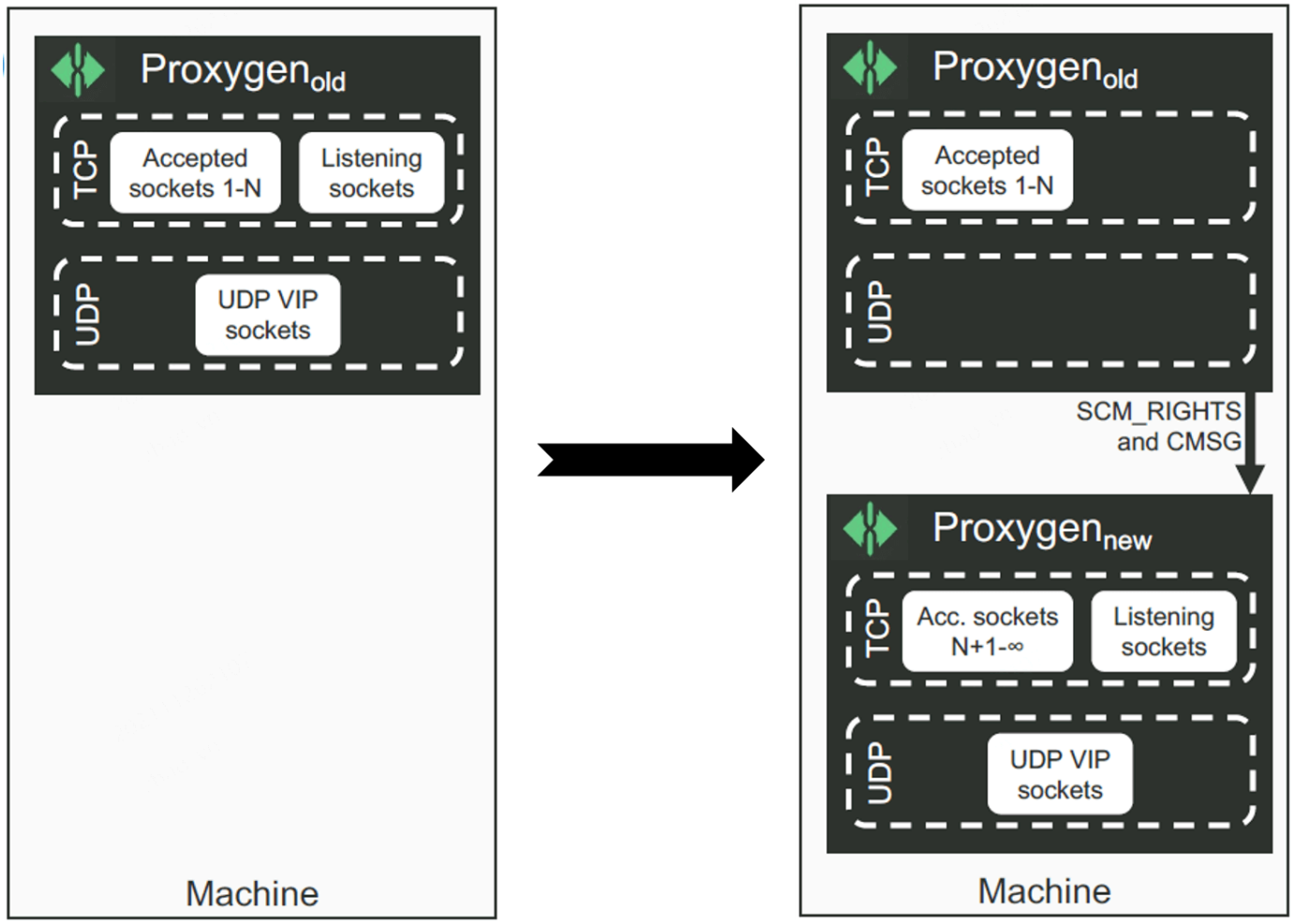
3.2.1 早期方案：socket takeover (or zero downtime restart)

我们在早期自己实现了一个所谓的 zero downtime restart 或称 socket takeover 方案。 具体细节见前面提到的 LPC 论文，这里只描述下大概原理： 相比于等待老进程的连接完全退出再开始发布，我们的做法是直接创建一个新进程，然后通过一个唯 一的 local socket **将老进程中 TCP listen socket 和 UDP sockets 的文件描述符** （以及 SCM rights）转移到新进程。

发布流程

如下图所示，**发布前**，实例正常运行，同时提供 TCP 和 UDP 服务，其中，

- TCP socket 分为两部分：**已接受的连接**（编号 1~N）和监听新连接的 **listening socket**
- UDP socket，bind 在 VIP 上



接下来**开始发布**：

1. 创建一个新实例
2. **将 TCP listening socket 和 UDP VIP 迁移到新实例**；老实例仍然 serving 现有 TCP 连接（`1 ~ N`），
3. **新实例开始接受新连接**（`N+1 ~ +∞`），包括新的 TCP 连接和新的 UDP 连接
4. **老实例等待 drain**

可以看到，这种方式：

1. 在发布期间不会导致系统容器降低，因为我们完全保留了老实例，另外创建了一个新实例
2. 发布速度可以显著加快，因为此时可以并发发布多个实例
3. 老连接被 reset 的概率可以大大降低，只要允许老实例有足够的 drain 窗口

那么，这种方式有什么缺点吗?

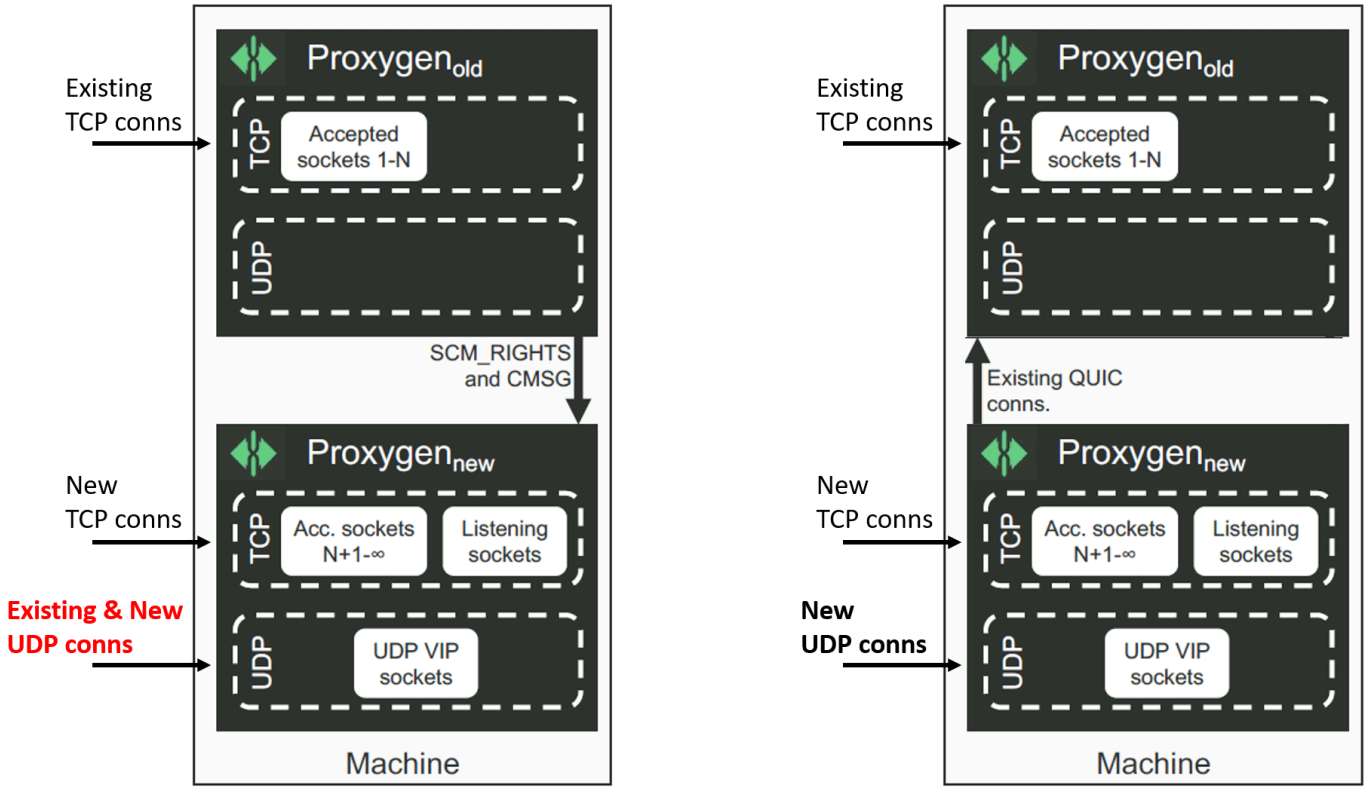
存在的问题

一个显而易见的缺点是：这种发布方式需要更多的系统资源，因为对于每个要升级的实例，它的新老实例需要并行运行一段时间；而在之前发布模型是干掉老实例再创建新实例， 不会同时运行。

但我们今天要讨论的是另一个问题：**UDP 流量的分发或称解复用（de-multiplex）**。

- TCP 的状态维护在内核。
- UDP 协议 —— 尤其是维护连接状态的 UDP 协议，具体来说就是 QUIC —— 所有 **状态维护在应用层而非内核**，因此内核完全没有 QUIC 的上下文。

由于 socket 迁移是在内核做的，而内核没有 QUIC 上下文（在应用层维护），因此 当**新老进程同时运行时，内核无法知道对于一个现有 UDP 连接的包，应该送给哪个进程**（因为对于 QUIC 没有 listening socket 或 accepted socket 的概念），因此有些包会到老进程，有些到新进程，如下图左边所示；



为解决这个问题，我们引入了用户空间解决方案。例如在 QUIC 场景下，会查看 ConnectionID 等 QUIC 规范中允许携带的元信息，然后根据这些信息，通过另一个 local socket 转发给相应的老进程，如以上右图所示。

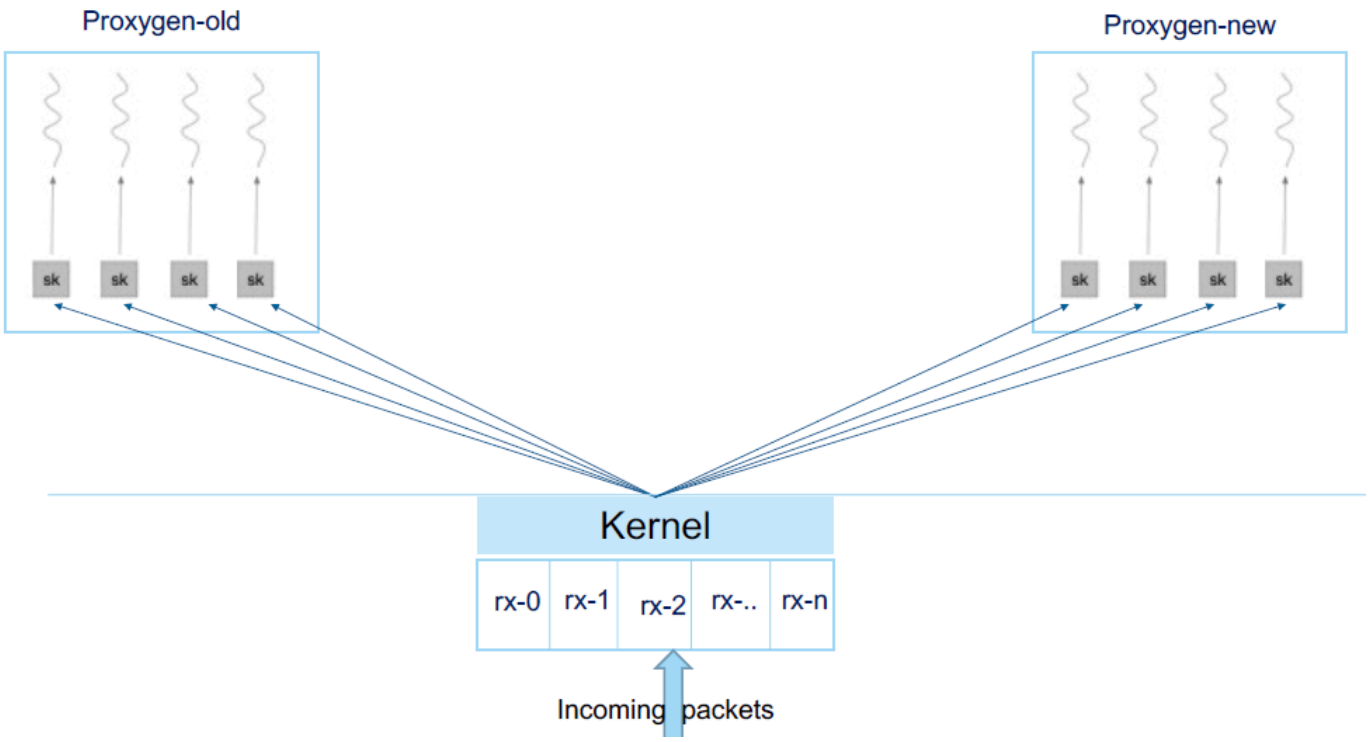
虽然能解决 QUIC 的问题，但可以看出，这种方式非常复杂和脆弱，涉及到大量进程间通信，需要维护许多状态。 有没有简单的方式呢?

3.2.2 其他方案调研：SO_REUSEPORT

Socket takeover 方案复杂性和脆弱性的根源在于：**为了做到客户端无感，我们在两个进程间共享了同一个 socket。** 因此要解决这个问题，就要避免在多个进程之间共享 socket。

这自然使我们想到了 SO_REUSEPORT: 它允许 **多个 socket bind 到同一个 port。** 但这里仍然有一个问题：UDP 包的路由过程是非一致的（**no consistent routing for UDP packets**），如下图

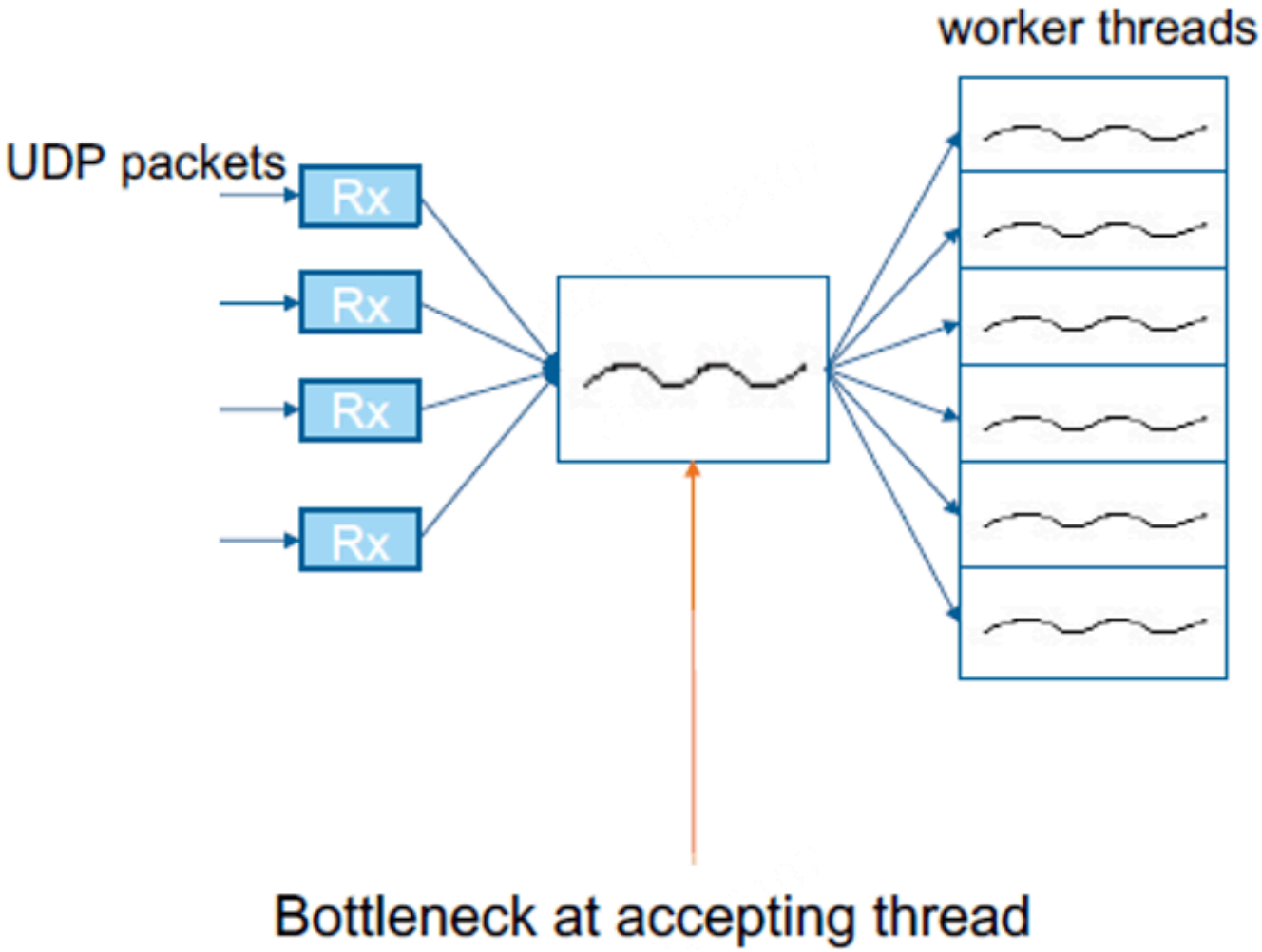
所示：



如果新老实例的 UDP socket bind 到相同端口，那一个实例重启时，哈希结果就会发生变化，导致这个端口上的包发生 misrouting。

另一方面，SO_REUSEPORT 还有**性能问题**，

- TCP 是有一个**独立线程负责接受连接**，然后**将新连接的文件描述符转给其他线程**，这种机制在负载均衡器中非常典型，可以认为是在 socket 层做分发；
- UDP **状态在应用层**，因此**内核只能在 packet 层做分发**，负责**监听 UDP 新连接的单个线性不但要处理新连接，还负责包的分发**，显然会存在瓶颈和扩展性问题。



因此直接使用 SO_REUSEPORT 是不行的。

3.2.3 思考

我们后退一步，重新思考一下我们的核心需求是什么。有两点：

1. 在内核中实现流量的无损切换，以便客户端完全无感知；
2. 过程能做到快速和可扩展，不存在明显性能瓶颈；

内核提供了很多功能，但并没有哪个功能是为专门这个场景设计的。 因此要彻底解决问题，我们必须引入某种创新。

- 理论上：只要我们能**控制主机内包的路由过程**（routing of the packets within a host），那以上需求就很容易满足了。
- 实现上：仍然基于 SO_REUSEPORT 思想，但同时解决 UDP 的一致性路由和瓶颈问题。

最终我们引入了一个 **socket 层负载均衡器** bpf_sk_reuseport。

3.3 新方案： `bpf_sk_reuseport`

3.3.1 方案设计

简单来说，

1. 在 socket 层 attach 一段 BPF 程序，控制 TCP/UDP 流量的转发（负载均衡）：
2. 通过一个 BPF map 维护配置信息，业务进程 ready 之后自己配置流量切换。

3.3.2 好处

这种设计的好处：

1. 通用，能处理多种类型的协议。
2. 在 VIP 层面，能更好地控制新进程（新实例）启动后的流量接入过程，例如

   Proxygen 在启动时经常要做一些初始化操作，启动后做一些健康检测工作， 因此在真正开始干活之前还有一段并未 ready 接收请求/流量的窗口 —— 即使它此时已经 bind 到端口了。

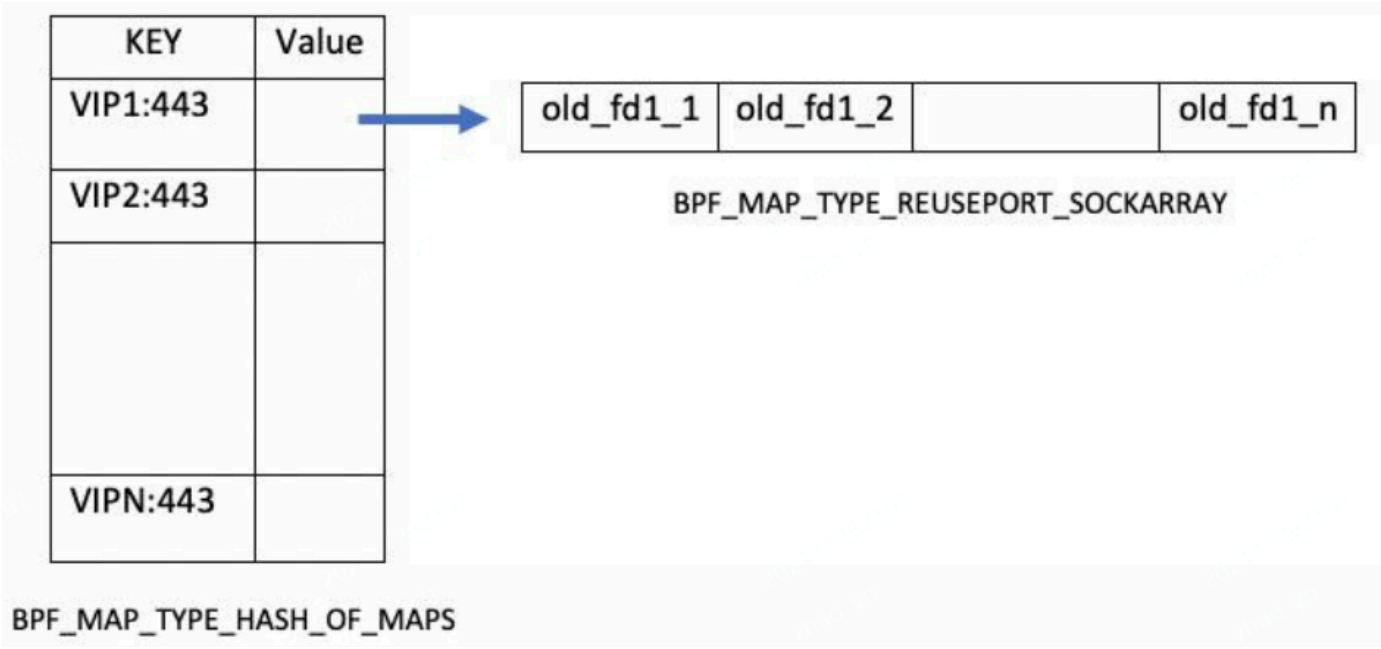
   在新方案中，我们无需关心这些，**应用层自己会判断新进程什么时候可以接受流量** 并通知 BPF 程序做流量切换；
3. 性能方面，也解决了前面提到的 UDP 单线程瓶颈；
4. 在包的路由（packet-level routing）方面，还支持根据 CPU 调整路由权重（adjust weight of traffic per-cpu）。 例如在多租户环境中，CPU 的利用率可能并不均匀，可以根据自己的需要实现特定算法来调度，例如选择空闲的 CPU。
5. 最后，未来迭代非常灵活，能支持多种新场景的实验，例如让每个收到包从 CPU 负责处理该包，或者 NUMA 相关的调度。

3.3.3 发布过程中的流量切换详解

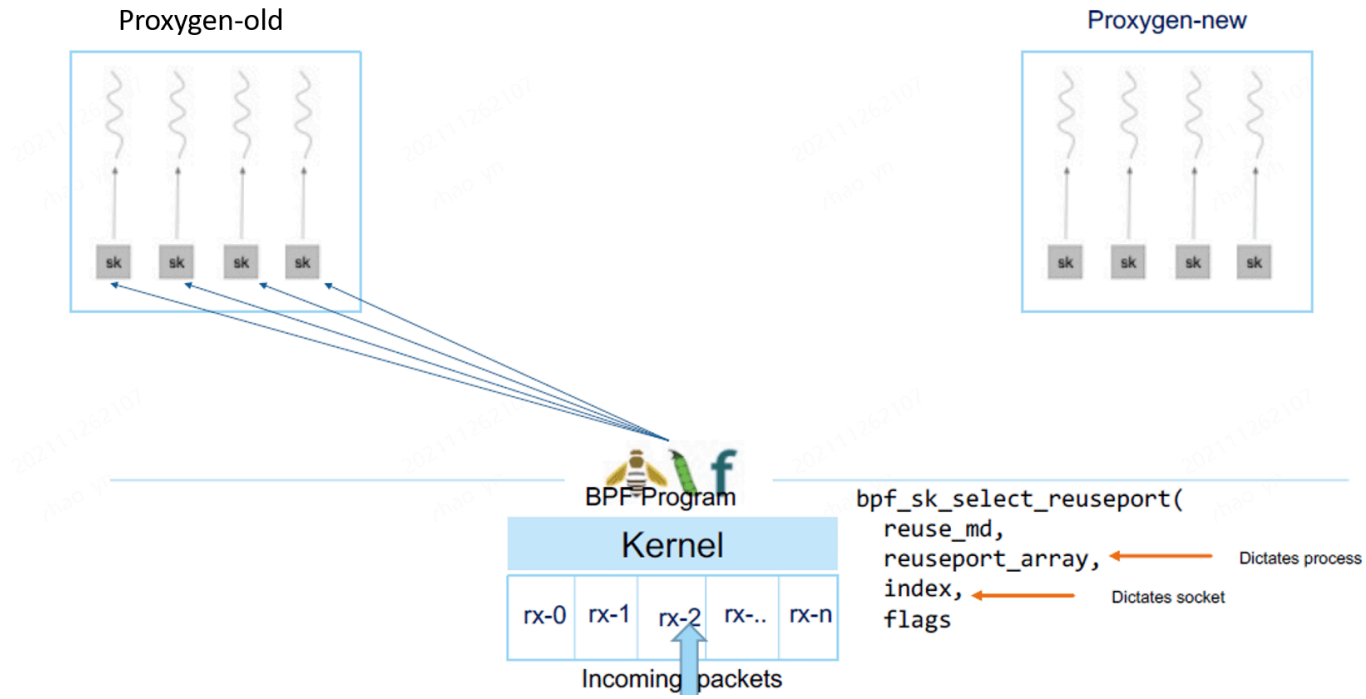
用一个 `BPF_MAP_TYPE_REUSEPORT_SOCKARRAY` 类型的 BPF map 来配置转发规则，其中，

- key： `<VIP>:<Port>`
- value：**socket 的文件描述符**，与业务进程一一对应

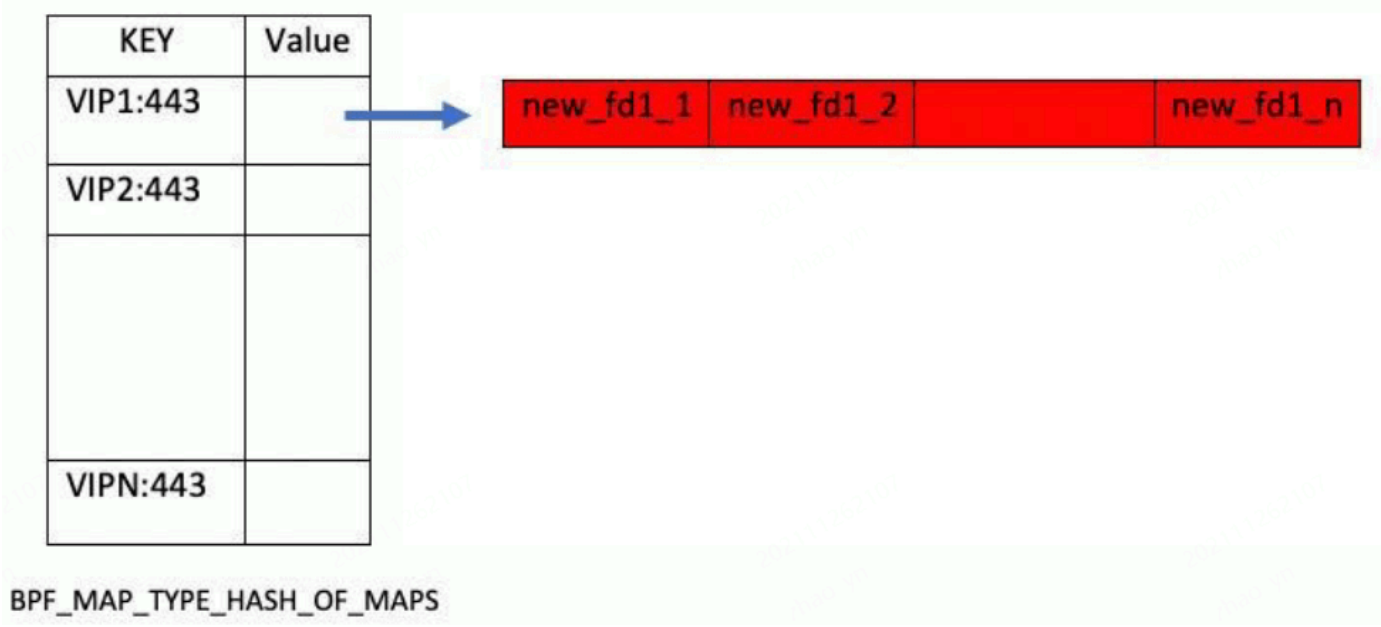
如下图所示，即使新进程已经起来，但只要还没 ready（BPF map 中仍然指向老进程），



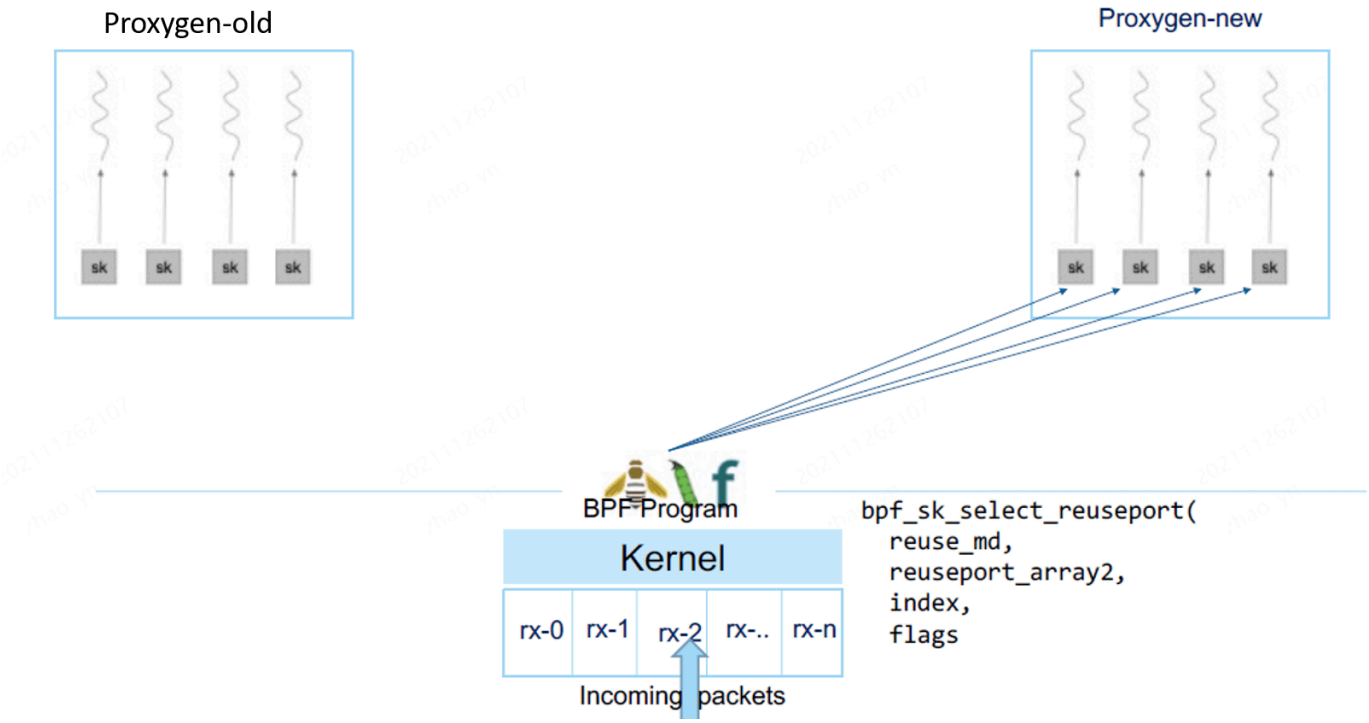
BPF 就继续将所有流量转给老进程，



新进程 ready 后，更新 BPF map，告诉 BPF 程序它可以接收流量了：



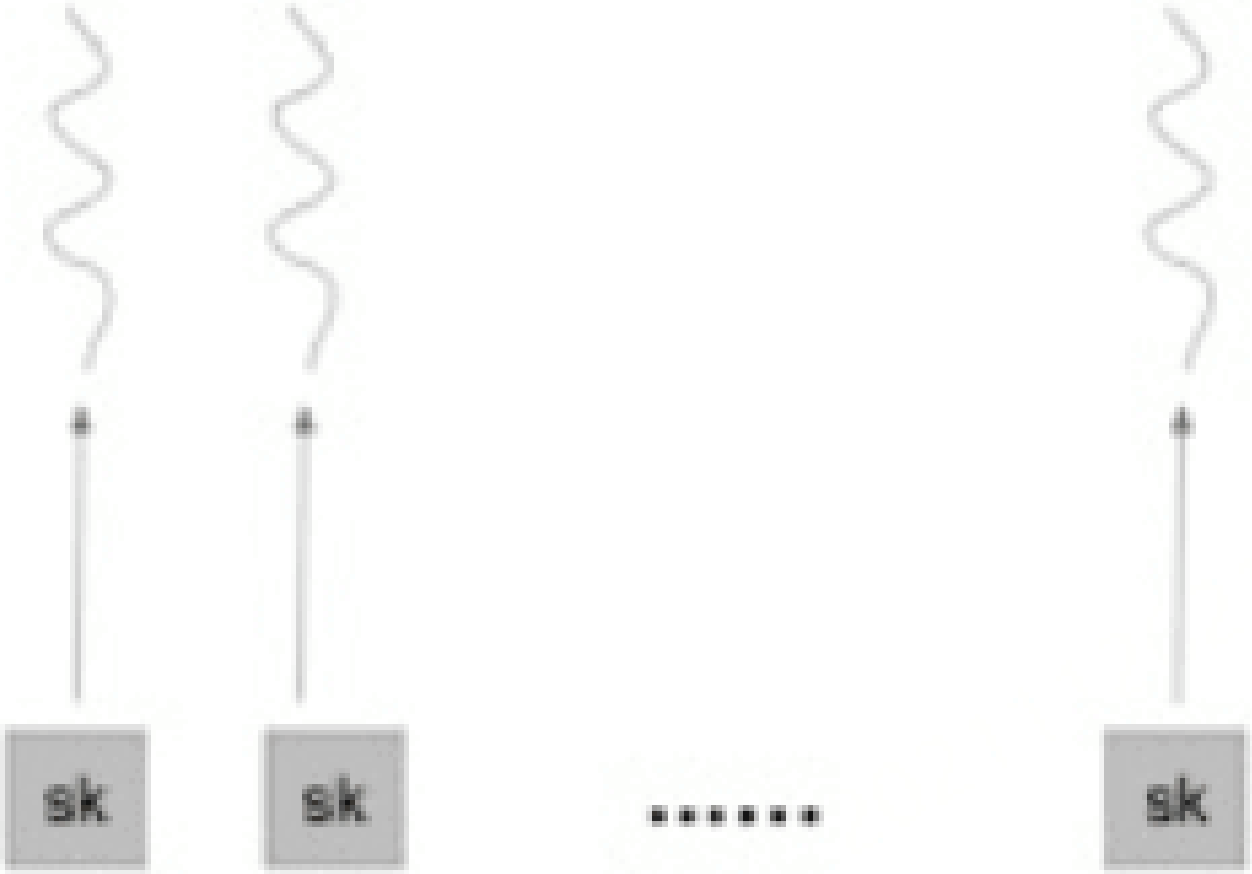
BPF 程序就开始将流量转发给新进程了：



前面没提的一点是：我们仍然希望将 UDP 包转发到老进程上，这里实现起来其实就非常简单了：

1. 已经维护了 flow -> socket 映射
2. 如果 flow 存在，就就转发到对应的 socket；不存在在创建一个新映射，转发给新实例的 socket。

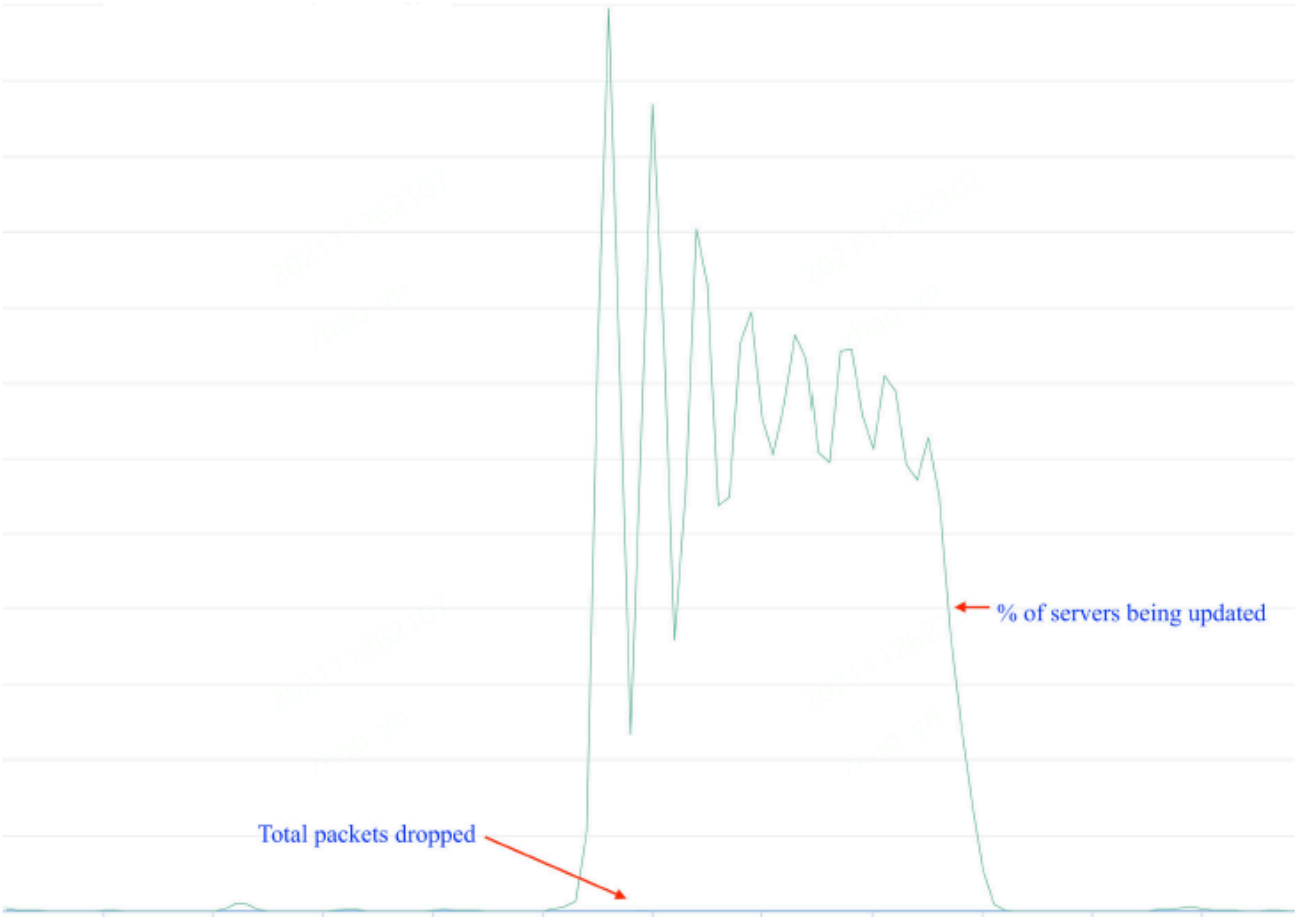
这也解决了扩展性问题，现在可以并发接收包（one-thread-per-socket），不用担心新进程启动时的 disruptions 或 misrouting 了：


sk
sk
sk

3.3.4 新老方案效果对比

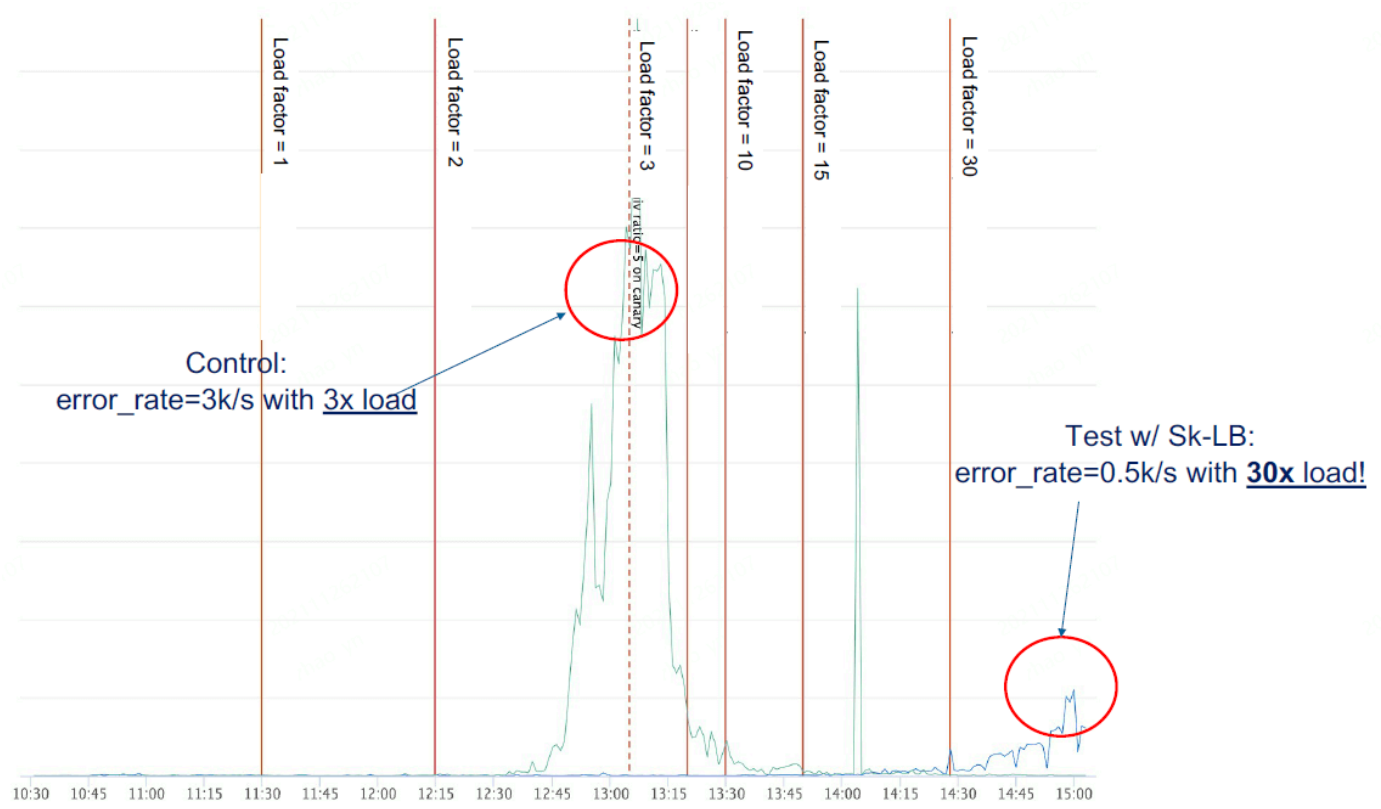
先来看**发布过程对业务流量的扰动程度**。下图是我们的生产数据中心某次发布的统计，图中有两条线：

- 一条是已发布的 server 百分比，
- 另一个条是同一时间的丢包数量，



可以看到在整个升级期间，丢包数量没有明显变化。

再来看**流量分发性能**，分别对 socket takeover 和 bpf_sk_reuseport 两种方式加压：



- 控制组/**对照组**（左边）：3x 流量时开始丢包，
- 实验组（右边）：30x，因此还没有到分发瓶颈但 CPU 已经用满了，但即使这样丢包仍然很少。

3.3.5 小结

本节介绍了我们的基于 BPF_PROG_TYPE_SK_REUSEPORT 和 BPF_MAP_TYPE_REUSEPORT_SOCKARRAY 实现的新一代发布技术，它能实现**主机内新老实例流量的无损切换**，优点：

1. 简化了运维流程，去掉脆弱和复杂的进程间通信（IPC），减少了故障；
2. 效率大幅提升，例如 UDP 性能 10x；
3. 可靠性提升，例如避免了 UDP misrouting 问题和 TCP 三次握手时的竞争问题。

4 讨论

4.1 遇到的问题：CPU 毛刺（CPU spikes）甚至卡顿

生产环境遇到过一个严重问题：新老进程同时运行期间，观察到 **CPU spike 甚至 host locking**；但测试环境从来没出现过，而且在实现上我们也没有特别消耗 CPU 的逻辑。

排查之后发现，这个问题跟 BPF 程序没关系，直接原因是

1. 在同一个 netns 内有大量 socket，
2. 新老实例同时以支持和不支持 bpf_sk_reuseport 的方式 bind 到了同一端口，

```
bind("[::1]:443"); /* without SO_REUSEPORT. Succeed. */
bind("[::2]:443"); /* with    SO_REUSEPORT. Succeed. */
bind("[::]:443");  /* with    SO_REUSEPORT. Still Succeed */
```

3. bind() 实现中有一个 spinlock 会遍历一个 hashtable bucket，这个哈希表**只用 dst_port 作为 key 去哈希**，

   如果有大量 http endpoints，由于它们的 **dst_port 很可能都是 443 和 80**， 因此会导致对应哈希槽上的链表特别长，在遍历时就会导致 CPU 毛刺甚至机器卡住。 这一问题下一小节专门介绍。

这个问题花了很长时间排查，因此有人在类型场景下遇到类似问题，很可能跟这个有关。 相关内核代码， 修复见 patch。

4.2 Listening socket hashtable

进一步解释上一小节提到的 hashtable 导致的 CPU 毛刺甚至卡顿问题以及 Facebook 的改进。 这个问题在 Cloudflare 2016 年的分享 The revenge of the listening sockets 中有详细介绍。

```
// include/net/inet_hashtables.h

static inline struct sock *__inet_lookup(struct net *net,
                     struct inet_hashinfo *hashinfo,
                     struct sk_buff *skb, int doff,
                     const __be32 saddr, const __be16 sport,
                     const __be32 daddr, const __be16 dport,
                     const int dif, const int sdif,
                     bool *refcounted)
{
    u16 hnum = ntohs(dport);
    struct sock *sk;

    // 查找是否有 ESTABLISHED 状态的连接
    sk = __inet_lookup_established(net, hashinfo, saddr, sport, daddr, hnum, dif, sdif);
    if (sk)
        return sk;

    // 查找是否有 LISTENING 状态的连接
    return __inet_lookup_listener(net, hashinfo, skb, doff, saddr, sport, daddr, hnum, dif
}
```

如以上代码所示，查找一个包对应的 socket 时，

1. 首先会查找是否有 ESTABLISHED 状态的 socket，如果没有
2. 再确认是否有 LISTENING 状态的 socket；这一步会查一下 listen hashtable，
    - 它的 bucket 数量非常小，内核宏定义为 32，此外，
    - 这个哈希表 **只根据目的端口（dst_port）来做哈希**，因此 **IP 不同但 dst_port 相同的 socket 都会哈希到同一个 bucket**（在 Cloudflare 的场景中，有 16K entry 会命中同一个 bucket，形成一个非常长的链表）。

`__inet_lookup_listener()` 老代码就不看了，直接看 5.10 的新代码，这已经包含了 Facebook 的 BPF 功能：

```
// net/ipv4/inet_hashtables.c

struct sock *__inet_lookup_listener(struct net *net,
                    struct inet_hashinfo *hashinfo,
                    struct sk_buff *skb, int doff,
                    const __be32 saddr, __be16 sport,
                    const __be32 daddr, const unsigned short hnum,
                    const int dif, const int sdif)
{
    struct inet_listen_hashbucket *ilb2;
    struct sock *result = NULL;
    unsigned int hash2;

    // 如果这里 attach 了 BPF 程序，直接让 BPF 程序来选择 socket
    /* Lookup redirect from BPF */
    if (static_branch_unlikely(&bpf_sk_lookup_enabled)) {
        result = inet_lookup_run_bpf(net, hashinfo, skb, doff, saddr, sport, daddr, hnum);
        if (result)
            goto done;
    }

    // 没有 attach BPF 程序或 BPF 程序没找到 socket： fallback 到常规的内核查找 socket 逻辑

    hash2 = ipv4_portaddr_hash(net, daddr, hnum);
    ilb2 = inet_lhash2_bucket(hashinfo, hash2);

    result = inet_lhash2_lookup(net, ilb2, skb, doff, saddr, sport, daddr, hnum, dif, sdif
    if (result)
        goto done;

    /* Lookup lhash2 with INADDR_ANY */
    hash2 = ipv4_portaddr_hash(net, htonl(INADDR_ANY), hnum);
    ilb2 = inet_lhash2_bucket(hashinfo, hash2);

    result = inet_lhash2_lookup(net, ilb2, skb, doff, saddr, sport, htonl(INADDR_ANY), hnu
```

```
done:
    if (IS_ERR(result))
        return NULL;
    return result;
}
```

4.3 bpf_sk_select_reuseport vs bpf_sk_lookup

这种两种类型的 BPF 程序，分别是 Facebook 和 Cloudflare （根据各自需求）引入内核的， 功能有些相似，因此拿来对比一下。

先看一段 Cloudflare 引入 `bpf_sk_lookup` 时的 commit message，

```
This series proposes a new BPF program type named BPF_PROG_TYPE_SK_LOOKUP,
or BPF sk_lookup for short.

BPF sk_lookup program runs when transport layer is looking up a listening
socket for a new connection request (TCP), or when looking up an
unconnected socket for a packet (UDP).

This serves as a mechanism to overcome the limits of what bind() API allows
to express. Two use-cases driving this work are:

(1) steer packets destined to an IP range, fixed port to a single socket

192.0.2.0/24, port 80 -> NGINX socket

(2) steer packets destined to an IP address, any port to a single socket

198.51.100.1, any port -> L7 proxy socket
```

> 更多信息，可参考他们的论文：
>
> The ties that un-bind: decoupling IP from web services and sockets for robust addressing agility at CDN-scale, SIGCOMM 2021

可以看到，它也允许多个 socket bind 到同一个 port，因此与 `bpf_sk_select_reuseport` 功能有些重叠，因为二者都源于这样一种限制：**在收包时，缺少从应用层直接命令内核选择哪个 socket 的控制能力。**

但二者也是有区别的：

- `sk_select_reuseport` 与 IP 地址所属的 socket family 是紧耦合的
- `sk_lookup` 则**将 IP 与 socket 解耦** —— lets it pick any / netns

« [译] 为 K8S WORKLOAD 引入的一些 BPF DATAPATH 扩展（LPC, 2021）

TRIP.COM: FIRST STEP TOWARDS CLOUD NATIVE SECURITY »