BiliBili: AF_XDP with QUIC 实践

01 背景

目前B站已在自建视频CDN下行中全量部署了基于QUIC和HTTP/3协议的网关服务（以下简称QUIC网关）。和TCP网关相比，QUIC网关在视频首帧、卡顿率以及加载失败率等常见的QoE/QoS指标方面都有不错的收益。另一方面，由于QUIC使用了更复杂的协议头和解析规则，此外Linux内核对UDP收发包的性能也不甚理想，这些方面都使得QUIC占用了更多的CPU负载，最终导致了更多资源成本的消耗。

为了给B站用户提供更稳定流畅的视频观看体验，同时降低成本，网络协议组团队通过技术选型，排除了DPU方案，决定使用AF_XDP技术来优化QUIC网关的收发包效率，减少CPU负载。

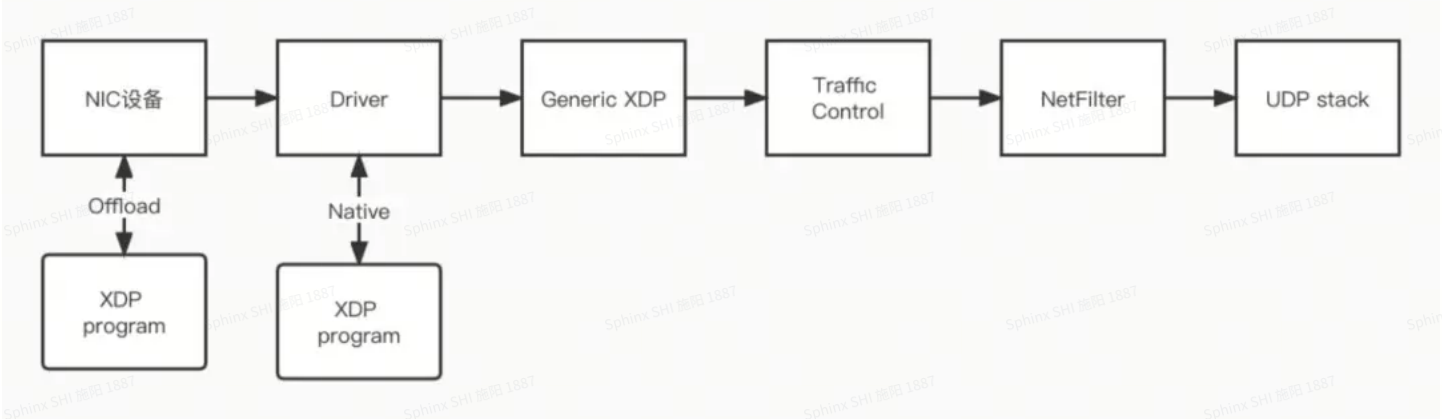
02 关于AF_XDP

2.1 概念

介绍AF_XDP，要从BPF说起。在Linux内核中，BPF（BSD Packet Filter）是一种高效灵活的类似虚拟机的工具，能够以安全的方式在不同的钩点执行用户注入到内核的字节码。它提供了一种过滤包的方法，并且避免了从内核空间到用户空间的无效的数据包复制行为。2013年的时候，Alexei Starovoitov对BPF进行了改造，在功能和性能方面有所改良，这就是eBPF（extended BPF）。

eBPF中包含许多钩子，因此它可以用于Linux内核中许多子系统中，而XDP就是Linux网络数据处理的一个eBPF hook点。XDP全称叫做eXPress Data Path，即快速数据路径，能够在网络数据包到达网卡驱动时对其进行处理。

XDP有三种运行模式，分别是generic模式、native模式以及offload模式。其中native模式下，XDP程序挂载在驱动接收路径上，是最传统的XDP的模式，需要驱动的支持，目前主流的网卡驱动，如ixgbe，i40e等都已实现了native XDP；generic是内核模拟出来的一种通用模式，不需要驱动支持，但是XDP挂载点更靠后，在内核协议栈接受路径上，性能不如native；offload则是直接在网卡中对XDP程序进行处理，挂载点更靠前，是性能最佳的模式，但是需要硬件特别支持。



AF_XDP是为高性能数据包处理而生的地址族。在XDP程序中使用XDP_REDIRECT这样的返回动作，可以使用bpf_redirect_map()将数据帧重定向其他使能了XDP的网卡，而AF_XDP socket能够将数据帧重定向到用户空间的内存缓冲区中。

2.2 工作流程

AF_XDP的核心组件主要分为两个部分：AF_XDP socket和UMEM。其中AF_XDP socket（xsk）的使用方法和传统的socket类似，AF_XDP支持用户通过socket()来创建一个xsk。每个xsk包含一个RX ring和TX ring，其中收包是在RX ring上进行的，发包则是在TX ring上面执行。用户也是通过操作RX ring和TX ring来实现网络数据帧的收发。UMEM是由一组大小相等的数据内存块所组成的。UMEM中每个数据块的地址可以用一个地址描述符来表述。地址描述符被定义为这些数据块在UMEM中的相对偏移。用户空间负责为UMEM分配内存，常用的方式是通过mmap进行分配。UMEM也包含两个ring，分别叫做FILL ring和COMPLETION ring。这些ring中保存着前面所说的地址描述符。

在收包前，用户将收包的地址描述符填充到FILL ring，然后内核会消费FILL ring开始收包，完成收包的地址描述符会被放置到xsk的RX ring中，用户程序消费RX ring即可获取接收到的数据帧。在发包时，用户程序向UMEM的地址描述符所引用的内存地址写入数据帧，然后填充到TX ring中，接下来内核开始执行发包。完成发包的地址描述符将被填充到COMPLETION ring中。

为了让xsk成功地从网卡中收到网络数据帧，需要将xsk绑定到确定的网卡和队列。这样，从特定网卡队列接收到的数据帧，通过XDP_REDIRECT即可重定向到对应已绑定的xsk。

03 利用AF_XDP优化QUIC网关性能

3.1 QUIC协议性能问题

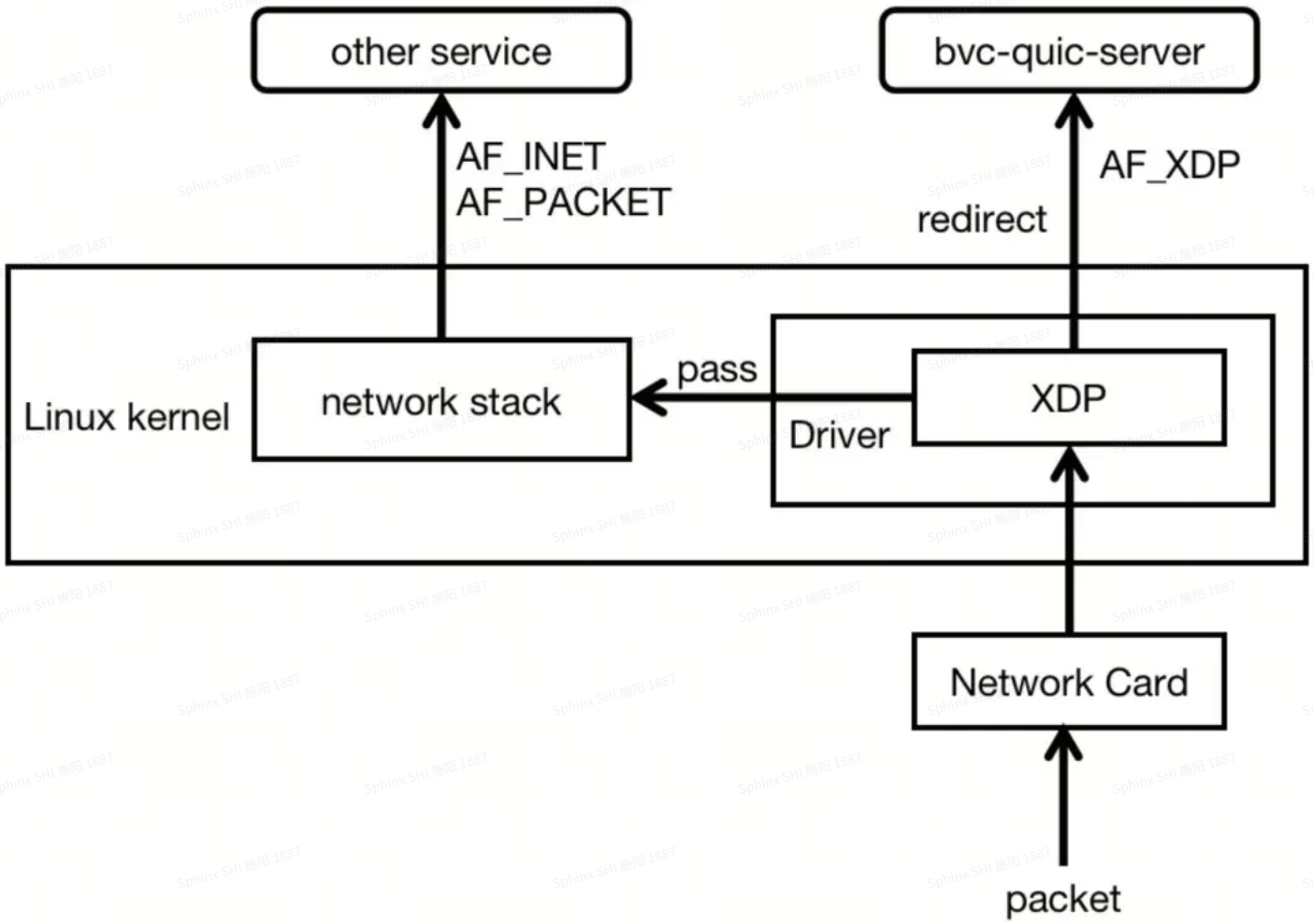
QUIC是google制定的一种基于UDP的低延时网络传输协议。基于QUIC协议的HTTP/3是新一代HTTP协议。目前我们团队已经在视频下行边缘CDN上全量部署了基于HTTP/3的QUIC网关：quic-server。在实践过程中，我们发现由于QUIC协议栈本身运行在用户态，协议逻辑复杂，加之内核对UDP的收发包效率远不如TCP，所以quic-server会使用更多的cpu负载。这不仅增加了服务的运行成本，在B站访问高峰期时还会有机器资源不足的风险。

3.2 基于AF_XDP的QUIC网关架构

对于quic-server来说，性能优化的重中之重就是UDP报文的收发。在这个过程中，我们也使用了不同的优化方式，包括UDP发包时使用sendmmsg来代替sendmsg，以及使用GSO来优化发包方式。这些方案一定程度上能够降低quic server的cpu负载，但是效果有限（10%左右），究其原因还是由于这些方案都是基于Linux内核的解决方案。例如我们使用的GSO实际上属于generic的GSO，还是在内核中实现的一种方案。如果想要进一步提升我们的程序收益，是否可能有kernel bypass的解决方案可用呢？于是，AF_XDP就进入了我们的视线。

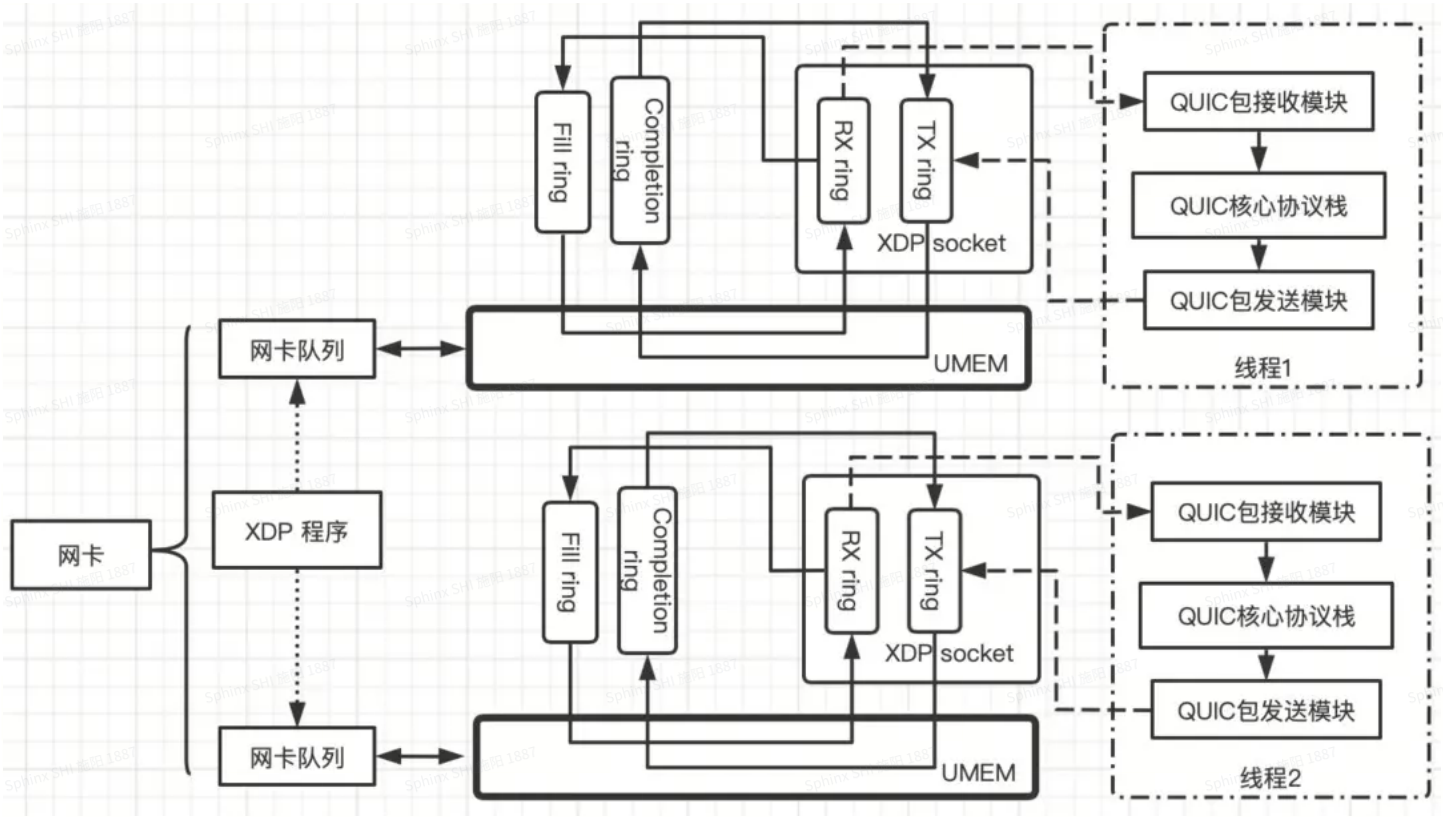
AF_XDP是一种半bypass的网络收发包机制。之所以叫做“半bypass”，是因为它的工作流程中还是需要内核的协助，并不能完全绕过内核，但是它的收发报的路径可以做到对内核协议栈的绕开。前文我们也介绍了AF_XDP的概念和工作流程，这里我们很容易用一句话来概括quic server如何使能AF_XDP：quic server采用AF_XDP来进行UDP报文的收发来提升其收发报性能，降低CPU负载。

下图是quic server和AF_XDP具体结合起来的一个网络数据走向示意图：



上图主要展示了网络数据的接收流转路径。数据帧从网卡到达驱动处理位置时，会运行挂载的XDP程序。XDP程序可由我们自行编写，利用XDP程序我们可以对数据帧进行过滤，即把要发给quic-server的HTTP/3请求所对应的数据帧重定向到quic-server维护的xsk中。

下图展示的是quic-server经过AF_XDP改造之后的架构图：



quic-server依然是多线程的架构，每个线程独立管理自己的xsk，每个xsk通过xskmap与AF_XDP ring进行绑定。这样驱动处理数据时，通过XDP程序就将特定队列的数据帧重定向到特定的xsk，也就可以通过quic-server特定的线程进行数据的处理，尽可能保证quic-server多线程并发处理数据包的性能。

下面简要介绍XDP程序的逻辑：

代码块

```
SEC(&quot;xdp_sock&quot;) int xdp_sock_prog(struct xdp_md *ctx){void *data_end = (void *)(long)ctx-&gt;data_end;void *data = (void *)(long)ctx-&gt;data;
    struct ethhdr *eth = data;
    unsigned short h_proto;if (eth + 1 &gt; data_end) {return XDP_DROP;}
    h_proto = eth-&gt;h_proto;if (h_proto == htons(ETH_P_IP)) {return handle_ipv4(ctx);}else if (h_proto == htons(ETH_P_IPV6))return handle_ipv6(ctx);else {return XDP_PASS;}}
```

上述代码片段是处理网络数据帧的主逻辑。根据TCP/IP协议分层体系，对数据帧进行解封装操作，检查数据帧的网络层协议，对于IPv4或IPv6数据方进行下一步处理，其他类型则返回XDP_PASS，这意味着非IPv4或IPv6的数据帧后续仍由内核协议栈进行处理。

代码块

```
static __always_inline int handle_ipv4(struct xdp_md *ctx){void *data_end = (void *)(long)ctx-&gt;data_end;void *data = (void *)(long)ctx-&gt;data;
    int dport;
    unsigned int key = 0;
```

```
4     struct iphdr *iph = data + sizeof(struct ethhdr);if (iph + 1 &gt; data_end) {return XDP_DROP;}if (iph-&gt;protocol != IPPROTO_UDP) {return XDP_PASS;}
5     dport = get_udp_dport(iph + 1, data_end);if (dport == -1) {return XDP_DROP;}else if (dport == expect_udp_dport) {
6         key = ctx-&gt;rx_queue_index;return bpf_redirect_map(&amp;xsks_map, key, 0);}return XDP_PASS;}
```

handle_ipv4函数对ipv4报文进行处理，继续解封装，过滤出传输层协议为UDP且目的端口为expect_udp_dport的数据帧，expect_udp_dport即为quic-server绑定的udp端口。最后通过bpf_redirect_map将数据帧重定向到队列绑定的xsk中。quic-server通过epoll管理xsk的读事件，通过消费Rx ring获取到所收到的包，实现了从xsk读取数据的过程。至此，通过AF_XDP，quic-server完成了网络数据的读取，这些数据通常包含的是公网HTTP/3请求。

在quic-server内部，UDP报文管理主要是由PacketWriter/BatchWriter模块负责的。我们实现了一个XskPacketWriter/XskBatchWriter，由它来对xsk的写数据进行管理。基于性能考虑，我们最终采用批量写的XskBatchWriter进行QUIC数据包的写管理。

架构图中QUIC包发送模块即为quic-server的XskBatchWriter，通过epoll将xsk的写事件管理起来，当xsk可写时，通过操作xsk的TX ring进行写操作，待网卡驱动完成数据帧的发送之后，由XskBatchWriter对Completion ring中的地址描述符进行回收。

3.3 性能分析

在不启用AF_XDP时，quic-server使用内核协议栈UDP socket进行UDP数据的读写。这种情况下，UDP数据的收发需要经过Linux内核协议栈，调用链路较长，同时还涉及到内核态和用户态的交互，存在比较大的性能开销。启用AF_XDP后，quic-server的性能可以得到明显的提升，具体原因在于：

1. 利用native XDP模式，在驱动早期处理数据时，即可将数据重定向到用户态维护的xsk中，调用链路明显缩短。反之，走内核协议栈的UDP收发报调用链路较长。
2. native XDP模式支持ZeroCopy，即驱动维护的RX ring和TX ring所指向的内存区域可以直接映射到用户空间，此空间通常由用户程序创建的UMEM来维护。这直接减少了数据的内存拷贝的次数，提升了收发包效率。反之，内核协议栈需要构建sk_buff对数据报文进行处理，还需要拷贝到用户空间。

3.4 收益

目前基于AF_XDP的QUIC网关已经在自建CDN下行中全量部署。我们专门就CPU负载这一指标进行了压力测试和上线效果对比。单线程压测结果表明，使用AF_XDP方案，在服务同等用户带宽的前提下，CPU负载比非AF_XDP方案低50%左右。进一步地，我们又在多线程，多range请求的条件下再次进行了两组压测，压测结果如表1和表2所示。两组测试的差异在于第二组测试采用的请求range范围更大。

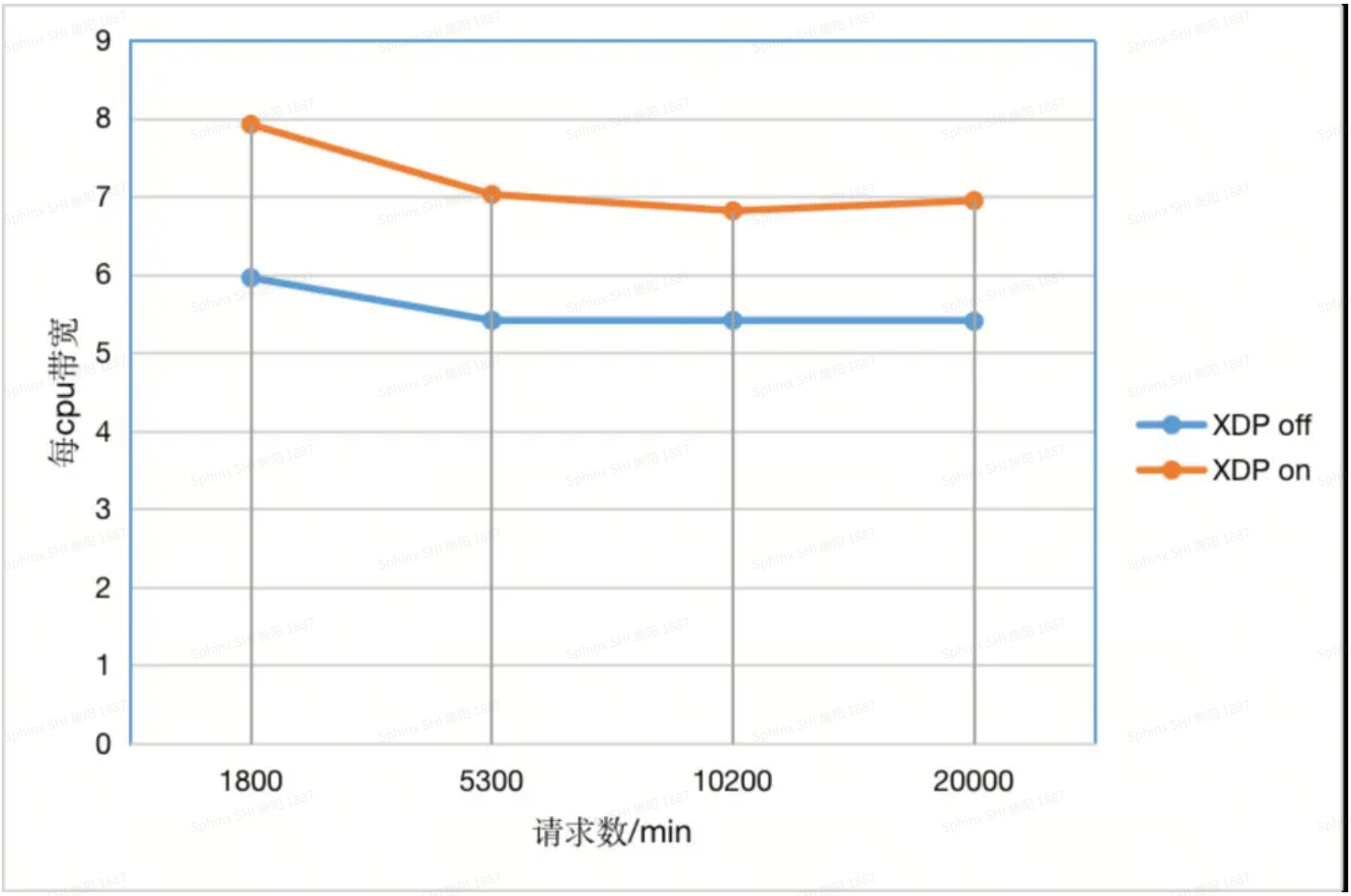


表1 压测对比结果第一组

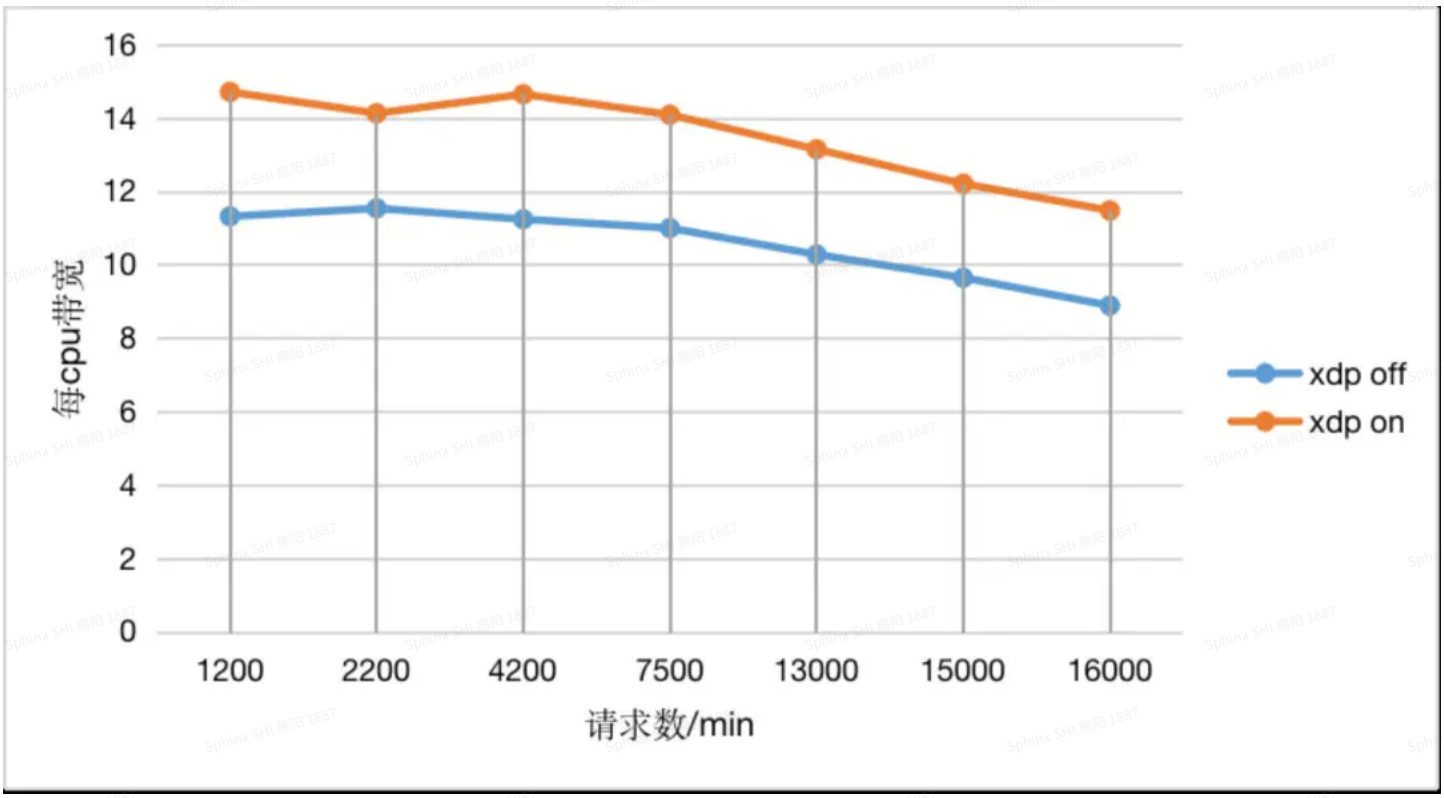


表2 压测对比结果第二组

可以明显看到，使用AF_XDP，在同等CPU load的前提下可以服务更多的用户带宽。

接下来，在同集群两台机器上进行对比测试，一台开启AF_XDP，一台关闭AF_XDP，对比效果如图所示。

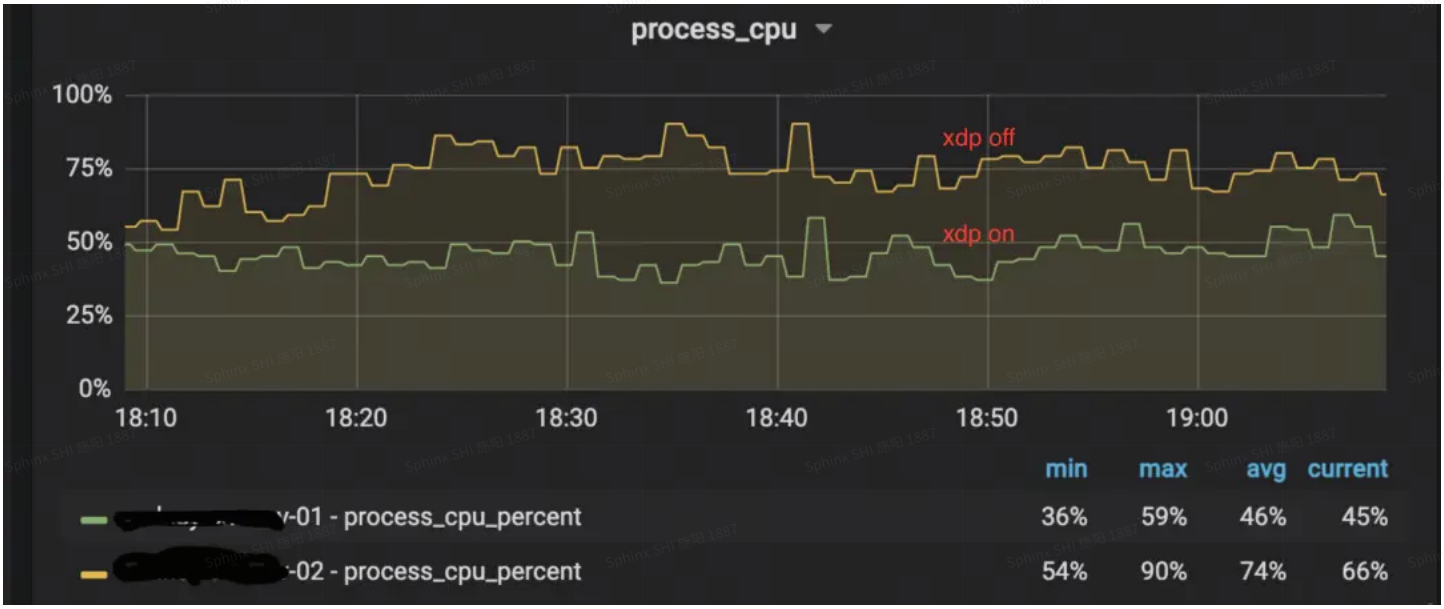


表3 相似平均带宽，开启xdp和不开启xdp，cpu的消耗对比

结论：

1. 压测结果表明xdp模式下的最大带宽近9Gbps；非xdp模式下的最大带宽接近7Gbps（表1，表2）。
2. xdp模式下的每带宽CPU比非xdp模式下提升25%～30%左右（表3）

04 未来展望

目前AF_XDP已在自建nCDN的QUIC网关上全量上线，且获得了较大的收益。除了进一步优化AF_XDP的性能之外，我们未来的计划是：

1. 开源AF_XDP的模块代码到bilibili/quiche仓库中。
2. 开发基于AF_XDP的性能分析工具。目前，我们在AF_XDP使能QUIC网关之后，通常是在网关程序中统计程序的运行状态，而通用的相关工具如tcpdump、iftop等都已失效。因此，需要完善相关周边工具，帮助我们更好地了解程序运行状态并针对性地对程序进行优化。
3. 赋能AF_XDP到更多业务场景中，如RTP, datachannel等基于UDP协议的传输协议。

参考链接：

[1] https://www.kernel.org/doc/html/latest/networking/af_xdp.html

[2] https://github.com/libbpf

[3] https://github.com/bilibili/quiche