## 问题背景

据某些 TKE 业务方反馈，延迟敏感型应用迁移至 TKE 生产环境之后，开始出现偶发性请求超时现象。针对这一现象，我们借助[云驭](https://yunyu.cloud.tencent.com/) （性能评测平台）对 TKE 网络展开 ping flood 性能测评，观察是否能够复现网络抖动现象。ping flood 具体场景如下图所示。其中，Pod A、Pod B 处于同一 K8s 集群，Pod A 向 Pod B 发起 ping flood，共计 5 千万次 ping 请求。



云驭评测报告的统计结果表明，ping 请求平均延迟 0.058 ms，共计 10 ~ 20 次大于 1ms，Debug 信息如下所示。

# ping -D -f 10.50.105.144 PING 10.50.105.144 (10.50.105.144) 56(84) bytes of data. [1633177499.852333] icmp\_seq=58063 time=1.30 ms [1633177506.879423] icmp\_seq=21613 time=1.27 ms [1633177522.629392] icmp\_seq=36949 time=1.29 ms [1633177580.466357] icmp\_seq=28338 time=2.37 ms [1633177680.083396] icmp\_seq=55992 time=1.27 ms ... rtt min/avg/max/mdev = 0.034/0.058/2.376/0.006 ms, ipg/ewma 0.073/0.046 ms

反复执行若干次表现基本一致，相当于稳定复现了 TKE 生产环境的网络抖动问题。

接下来，本文将针对 ping flood 偶发性网络抖动问题进一步分析。首先编写新的 eBPF 追踪工具用来分析 ping 请求在网络协议栈的处理情况、发现关键线索，然后使用现有 eBPF 追踪工具定位根因，最终提出折中的解决方案消除 TKE 生产环境的网络抖动问题。

## 问题分析

刚开始我们先使用传统性能工具（top、free、sar、iostat 等）进行定性分析，发现 CPU 使用率、平均负载、内存、网络流量和丢包数量、磁盘 I/O 等指标均无明显异常。分析网络抖动问题的难点在于偶发性，当抖动真正发生的时候缺少比较详细的现场状态信息，传统性能工具应对这类问题显得有些捉襟见肘，而基于 eBPF 的追踪工具能够起到很好的弥补这一缺陷。eBPF 强大的系统级观测能力，更深更细的辅助性能分析与问题定位工作，尤其适用于解决偶发性的网络抖动问题。由此，我们开始尝试基于 eBPF 分析该问题。

## eBPF 选用原因

* 传统网络性能工具可以展示内核中的各种统计信息，包括包速率、吞吐量、socket 状态等。但内核状态是这些工具的盲区，它们看不到具体哪个进程发送了哪些数据包，也看不到对应的调用栈信息以及 socket、TCP 的内核状态。这些信息只能借助 eBPF 追踪工具或较底层的 Tracepoints、kprobes 获得。
* 多数传统网络性能工具的最小统计间隔以秒为单位，难以识别最大延迟仅几个 ms 的 ping 请求。
* 常见的网络抓包工具 tcpdump 除了增加每个网络包的处理开销，还增加了将数据包写入文件系统时的 CPU、内存和存储的开销，还需要在后期进行读取处理时的开销。相比起来，使用 eBPF 追踪每个网络包本身就将效率提升了很多。因为 eBPF 直接在内核内存中保存统计信息，而不需要抓包文件。eBPF 使用 Tracepoints、kprobes 等探索网络软件栈的内部情况，扩展传统网络性能工具提供的网络可观测能力，为细粒度分析网络问题提供了更多新思路。

## eBPF 工具设计思路

目前，还没有现成的 eBPF 工具可以提供细粒度的 ping 数据包追踪功能，不过我们可以使用 eBPF 前端 bcc 很方便的编写自定义的 eBPF 工具。

针对 ping 数据包发送与接收流程，我们编写了 eBPF 工具 pingsnd、pingrcv 追踪工具，具体设计思路如下：

* 利用 kprobes 记录 ping 数据包在发送端、接收端网络软件栈关键函数（napi\_gro\_receive、\_\_netif\_receive\_skb\_core、virtqueue\_xxx 等）的状态信息、耗时情况以及关键函数之间的时间差，便于分析网络软件栈处理流程的延迟情况。关键函数如下图所示，其中实线箭头标识 ping request 处理流程，而虚线箭头标识 ping reply 处理流程，合在一起构成一次完整的 ping 请求。



* ping 发送端、接收端分开进行追踪，最终形成相应的 pingsnd、pingrcv 追踪工具。pingsnd、pingrcv 追踪工具在每次事件发生时，以 ICMP 编号为主键，记录 CPU 编号、进程名称、系统时间等上下文信息。
* pingsnd 仅追踪指定目的 IP 地址相关的数据包，且过滤掉 ping 发送端处理时长小于指定最小阈值的数据包（默认最小阈值为 0 ms，不执行过滤）。
* pingrcv 仅追踪指定源 IP 地址相关的数据包，且过滤掉 ping 接收端处理时长小于指定最小阈值的数据包（默认最小阈值为 0 ms，不执行过滤）。

pingsnd、pingrcv 源码文件见附件。

## eBPF 工具小试牛刀

1. ping 发送端（Pod A）执行 pingsnd，追踪目的 IP 为 10.50.105.144（Pod B）所有 ping 请求。

(Pod A)# ./pingsnd -i 10.50.105.144 > /run/pingsnd.log

追踪结果写入 tmpfs 文件，ext4fs 可能来不及写入而导致一部分追踪事件丢失。

1. ping 接收端（Pod B）执行 pingrcv，追踪所有 ping 请求：

(Pod B)# ./pingrcv -i 10.50.105.144 > /run/pingrcv.log

追踪结果写入 tmpfs 文件，ext4fs 可能来不及写入而导致一部分追踪事件丢失。

1. 触发云驭针对 TKE 制定的性能评测流水线，相当于执行以下 ping flood 命令：

(Pod A)# ping -f -D 10.50.105.144

1. 运行一段时间，共计出现 5 次大于 1ms 的 ping 请求（已涵盖了待分析的两类情况），接下来将以这些请求作为分析样本。为了便于表述，分别编号 pr1、pr2、pr3、pr4、pr5。

PING 10.50.105.144 (10.50.105.144) 56(84) bytes of data. [1633177499.852333] icmp\_seq=58063 time=1.30 ms [1633177506.879423] icmp\_seq=21613 time=1.27 ms [1633177522.629392] icmp\_seq=36949 time=1.29 ms [1633177580.466357] icmp\_seq=28338 time=1.29 ms [1633177680.083396] icmp\_seq=55992 time=1.27 ms --- 10.50.105.144 ping statistics --- 3329313 packets transmitted, 3329083 received, 0% packet loss, time 243393ms rtt min/avg/max/mdev = 0.034/0.058/1.306/0.006 ms, ipg/ewma 0.073/0.046 ms

1. 首先分析 pr1（icmp\_seq: 58063），根据时间戳、icmp\_seq，搜索步骤 1 追踪结果文件 /run/pingsnd.log，pr1 在发送端的追踪结果如下。

2021-10-02 20:24:59.736836 603944.736836 ping send timeval: 1633177499.851011 delta(ms):115.485 1. send ping at ip\_output seq:58063 cpu:7 pid:579577 delta(ms):0.000 2. send ping at \_\_dev\_queue\_xmit seq:58063 cpu:7 pid:579577 delta(ms):0.001 3. send ping at virtqueue\_add\_outbuf seq:58063 cpu:7 pid:579577 delta(ms):0.002 4. recv ping at napi\_gro\_receive seq:58063 cpu:10 pid:0 delta(ms):1.304 5. recv ping at \_\_netif\_receive\_skb\_core seq:58063 cpu:7 pid:0 delta(ms):1.309 6. recv ping at \_\_sock\_queue\_rcv\_skb seq:58063 cpu:7 pid:0 delta(ms):1.310

1 ~ 3 将 ping request 数据包加入 vring 队列。   
3 ~ 4 时间差大于 1ms。   
4 ~ 6 网络协议栈中接收 ping reply 数据包，并投递至 socket 接收队列。网络协议栈处理耗时占比很小。   
4 在 CPU 10 上执行，而 5 在 CPU 7 上执行，背后实际上发生了 RPS 负载均衡。（下面的追踪结果也有这种情况，不再一一赘述。）

搜索步骤 2 追踪结果文件 /run/pingrcv.log，pr1（icmp\_seq: 58063）在接收端的追踪结果如下。

2021-10-02 20:24:58.895722 603967.895722 ping send timeval: 1633177499.851011 delta(ms):955.303000 1. recv ping at napi\_gro\_receive seq:58063 cpu:15 pid:100 delta(ms):0.000 2. recv ping at \_\_netif\_receive\_skb\_core seq:58063 cpu:15 pid:100 delta(ms):0.001 3. recv ping at icmp\_recv seq:58063 cpu:9 pid:0 delta(ms):0.011 4. send ping at ip\_output seq:58063 cpu:9 pid:0 delta(ms):0.012 5. send ping at \_\_dev\_queue\_xmit seq:58063 cpu:9 pid:0 delta(ms):0.013 6. send ping at virtqueue\_add\_outbuf seq:58063 cpu:9 pid:0 delta(ms):0.014

1 ~ 3 网络协议栈中接收 ping request 数据包，由运行在 CPU 15 的内核线程 ksoftirqd/15 (pid: 100) 处理。网络协议栈处理耗时较小（0.014 ms）。   
4 ~ 6 将 ping request 数据包加入 vring 队列。

1. 同理，pr2（icmp\_seq: 21613）在发送端的追踪结果如下。

2021-10-02 20:25:06.763924 603951.763924 ping send timeval: 1633177506.878124 delta(ms):115.485 1. send ping at ip\_output seq:21613 cpu:7 pid:579577 delta(ms):0.000 2. send ping at \_\_dev\_queue\_xmit seq:21613 cpu:7 pid:579577 delta(ms):0.001 3. send ping at virtqueue\_add\_outbuf seq:21613 cpu:7 pid:579577 delta(ms):0.002 4. recv ping at napi\_gro\_receive seq:21613 cpu:10 pid:70 delta(ms):1.271 5. recv ping at \_\_netif\_receive\_skb\_core seq:21613 cpu:7 pid:0 delta(ms):1.275 6. recv ping at \_\_sock\_queue\_rcv\_skb seq:21613 cpu:7 pid:0 delta(ms):1.285

1 ~ 3 将 ping request 数据包加入 vring 队列。   
3 ~ 4 时间差大于 1ms。其中，napi\_gro\_receive 由运行在 CPU 10 的内核线程 ksoftirqd/10 (pid: 70) 触发。   
4 ~ 6 网络协议栈中接收 ping reply 数据包，并投递至 socket 接收队列。网络协议栈处理耗时较小（0.016ms）。

pr2（icmp\_seq: 21613）在接收端的追踪结果如下。

2021-10-02 20:25:05.921582 603974.921582 ping send timeval: 1633177506.878124 delta(ms):956.550 1. recv ping at napi\_gro\_receive seq:21613 cpu:15 pid:0 delta(ms):0.000 2. recv ping at \_\_netif\_receive\_skb\_core seq:21613 cpu:15 pid:0 delta(ms):0.005 3. recv ping at icmp\_recv seq:21613 cpu:9 pid:0 delta(ms):0.006 4. send ping at ip\_output seq:21613 cpu:9 pid:0 delta(ms):0.007 5. send ping at \_\_dev\_queue\_xmit seq:21613 cpu:9 pid:0 delta(ms):0.007 6. send ping at virtqueue\_add\_outbuf seq:21613 cpu:9 pid:0 delta(ms):0.008

1 ~ 3 网络协议栈中接收 ping request 数据包。   
4 ~ 6 将 ping request 数据包加入 vring 队列。   
网络协议栈处理耗时较小（0.008 ms）。

1. 继续依次分析 pr3 ~ pr5。**结果发现，ping 请求发生延迟抖动时，网络协议栈处理耗时占比很小。通过 pingsnd、pindrcv 等 eBPF 工具并未明确定位较大比重的延迟发生的具体位置。不过提供了一条有价值的线索：ping 数据包接收的关键函数 napi\_gro\_receive 由内核线程 ksoftirqd/xx 触发。**pr1 ~ pr5 统计情况如下表所示。因此，初步怀疑网络抖动是由于软中断线程 ksoftirqd/xx 调度延迟导致。

| **请求编号** | **位置** | **cpu** | **pid** | **comm** | **原始请求** |
| --- | --- | --- | --- | --- | --- |
| pr1 | 接收端 | 15 | 100 | ksoftirqd/15 | [1633177499.852333] icmp\_seq=58063 time=1.30 ms |
| pr2 | 发送端 | 10 | 70 | ksoftirqd/10 | [1633177506.879423] icmp\_seq=21613 time=1.27 ms |
| pr3 | 发送端 | 10 | 70 | ksoftirqd/10 | [1633177522.629392] icmp\_seq=36949 time=1.29 ms |
| pr4 | 接收端 | 14 | 94 | ksoftirqd/14 | [1633177580.466357] icmp\_seq=28338 time=1.29 ms |
| pr5 | 接收端 | 15 | 100 | ksoftirqd/15 | [1633177680.083396] icmp\_seq=55992 time=1.27 ms |

1. 为了验证这一想法，在 ping 发送端、接收端分别执行 BCC 现有工具 runqslower，追踪大于 1000 ms 的调度延迟事件，这里仅关注 ksoftirqd 内核线程，具体命令如下：

# ./runqslower 1000 | grep ksoftirqd

perf sched 也可以用于分析任务调度延迟，不过相比 eBPF 现有工具 runqlat、runqslower 开销非常大（包括事件数据拷贝、磁盘 I/O、后续分析统计开销等），尤其是很繁忙的生产系统。因此，eBPF 工具可用情况下，不推荐使用 perf sched。

然后 ping 发送端执行 ping flood，一段时间后，如下大于 1ms 的 ping 请求（一共出现 5 次）。

[1634029324.545499] icmp\_seq=16704 time=1.52 ms [1634029414.366726] icmp\_seq=55064 time=1.40 ms [1634029414.780387] icmp\_seq=61242 time=1.44 ms [1634029469.778275] icmp\_seq=35284 time=1.37 ms [1634029714.109554] icmp\_seq=34807 time=2.48 ms

查看 ping 发送端追踪结果，一共 2 次，与上述 ping flood 输出结果的其中 2 个请求的时间点吻合。

# ./runqslower 1000 | grep ksoftirqd Tracing run queue latency higher than 1000 us TIME COMM PID LAT(us) 17:04:29 ksoftirqd/15 100 1372 17:08:34 ksoftirqd/15 100 2482

查看 ping 接收端追踪结果，一共 3 次，与上述 ping flood 输出结果的其中 3 个请求的时间点吻合。

# ./runqslower 1000 | grep ksoftirqd Tracing run queue latency higher than 1000 us TIME COMM PID LAT(us) 17:02:04 ksoftirqd/6 46 1509 17:03:34 ksoftirqd/7 52 1346 17:03:34 ksoftirqd/7 52 1372

ping 发送端、接收端一共追踪到 5 次大于 1ms 的 ksoftirqd/xx 软中断调度延迟，与上述 ping flood 输出结果中各个 ping 请求发生时间点吻合。

BCC 现有工具 runqlat 方便查看整体或指定进程的调度延迟分布直方图，同样也可以发现上述 ksoftirqd/xx 软中断调度延迟问题，感兴趣的读者可以做下尝试。

1. （选读）借助 pingsnd、pingrcv 追踪工具并未直接发现延迟抖动发生的具体位置，我们可以继续扩展其边界，把 ping 请求的关键链路进一步细化。整个过程相对复杂，这里不再详细展开，大致讲述一下思路。我为此编写了一个 BCC 工具 vring\_interrupt，用于追踪接收数据包事件（napi\_gro\_receive）的上一个紧邻事件（vring\_interrupt）发生的时间点，尝试分析 vring\_interrupt -> napi\_gro\_receive 之间是否存在较大的时间差。vring\_interrupt 参数中不包含 skb，因此不能与 pingsnd、pingrcv 相关事件的时间点直接对上。不过，Pod 网卡在没有进行 ping flood 时基本没有数据包进来（偶尔有极少量的 IGMP 数据包），因此提供了计算 vring\_interrupt -> napi\_gro\_receive 时间差的可能性。当 vring\_interrupt 事件发生时，根据 irq 编号过滤（比如 Pod B 内部的 eth0 网卡队列的 irq 范围为 45 ~ 52），然后打印出该事件发生时刻对应的 CPU、进程名称、进程 ID、时间戳。最终可以结合 pingsnd、pingrcv 追踪工具的输出，间接分析出 ksoftirqd 内核线程存在调度延迟。

利用新编写的 vring\_interrupt 工具，结合 pingsnd、pingrcv，分析出延迟抖动发生的具体位置在 vring\_interrupt -> napi\_gro\_receive 之间。

vring\_interrupt 源码文件见附件。

## 优化方案

针对 ksoftirqd 软中断调度延迟，比较常见的一种折中解决方案是更改 ksoftirqd 内核线程的调度策略与优化级，例如将调度策略修改实时调度 RR 或 FIFO。该方案同样有副作用，不具有普适性，但对于延迟敏感型应用而言在某种程度上却又是适用的。其实，Linux 软中断处理自身就充斥了不少折中逻辑，仍然在不断改进中。

由于 ping 发送端、接收端均可能存在 ksoftirqd 软中断调度延迟，因此两端均需要修改 ksoftirqd 内核线程的调度策略与优化级。

pgrep ksoftirqd | xargs -t -n1 chrt -r -p 99

建议获取 Pod 网卡队列 irq 绑定 CPU 列表，仅修改该列表对应 ksoftirqd 内核线程的调度策略与优化级。

实施该折中方案，然后执行 ping flood，不再出现大于 1ms 的 ping 请求了。

更优的解决方案有待进一步探索。。。

## 总结思考

针对 TKE 网络抖动问题，我们编写了 eBPF 工具 pingsnd、pingrcv，并结合 eBPF 现有工具 runqlat、runqslower，定位出 ksoftirqd 软中断调度延迟是导致网络抖动问题的罪魁祸首，最后提出一种优化方案解决该低概率的网络抖动问题。

* 相比传统工具，使用 BCC 工具极大的提升了系统观测能力，适合分析网络抖动、调度器延迟等问题。但是，不要太相信工具自身，结果不符合预期时，保持怀疑态度。在这个过程中，也踩了不少 BCC 工具的坑，包括 softirqs、runqlat、runqlower 在现有环境上都有小问题。
* eBPF 现有工具不够用的情况下，尝试创造工具，提供更细粒度的下钻分析能力。

## eBPF 常用工具附录

排查该问题过程中使用到的 eBPF 工具汇总如下，供读者参考。

# 查看硬中断触发频率、累计耗时等 /usr/share/bcc/tools/hardirqs -C 1 /usr/share/bcc/tools/hardirqs -d -NT 1 # 查看软中断延迟分布、累计耗时等 /usr/share/bcc/tools/softirqs 1 10 /usr/share/bcc/tools/softirqs -dT 1 # 指定 CPU、频率采样（相比 perf record 更高效）： /usr/share/bcc/tools/profile -f -C 14 -F 999 # 网络包接收、RPS 等内核函数计数 /usr/share/bcc/tools/funccount -i 1 napi\_gro\_receive /usr/share/bcc/tools/funccount -i 1 get\_rps\_cpu # 查看内核函数延迟分布： /usr/share/bcc/tools/funclatency -uTi 5 \_\_do\_softirq /usr/share/bcc/tools/funclatency -uTi 5 net\_rx\_action # 查看内核函数参数分布、调用次数： /usr/share/bcc/tools/argdist -H 'r::vfs\_read()' /usr/share/bcc/tools/argdist -C 'r::vfs\_read()' /usr/share/bcc/tools/argdist -H 't:net:net\_dev\_queue():u32:args->len' # 追踪事件每次发生时的状态信息 /usr/share/bcc/tools/trace -C 'r::vfs\_read "%d", retval' # 查看指定线程的调度延迟： /usr/share/bcc/tools/runqlat 1 10 /usr/share/bcc/tools/runqlat -p 94 -mT 2 # 查看大于指定时长的调度延迟： /usr/share/bcc/tools/runqslower 1000 /usr/share/bcc/tools/runqslower -p 94 1000 # 查看堆栈+计数： /usr/share/bcc/tools/stackcount -Ti 5 icmp\_rcv /usr/share/bcc/tools/stackcount -Ti 5 vring\_interrupt /usr/share/bcc/tools/stackcount -c 12 -i 1 \_\_raise\_softirq\_irqoff

本文提及 eBPF 工具的运行要求

1. Required kernel version: 4.9+ 2. BCC pre-installed: yum -y install bcc

想了解更多关于性能优化的专题，请参考腾讯云服务性能优化工程季报(2021第四季度)：<https://km.woa.com/articles/show/528347?ts=1637311474>

## 往期 eBPF 系列文章

[【性能分析】eBPF 工具快速定位腾讯会议雪崩根因](https://km.woa.com/group/cloudlei/articles/show/544800)

[TKEx 容器中运行 eBPF 工具的填坑指南](https://km.woa.com/group/38034/articles/show/548226)

[基于 eBPF 的内存泄漏（增长）通用分析方法探索](https://km.woa.com/group/38034/articles/show/528025?ts=1686660707)

[ClickHouse 一次从 P 到 I 的性能优化实践](https://km.woa.com/articles/show/563499)