LPC 2025

Methodology and Practice in Observing Kernel Networking

Jason Xing

Tencent

I. Background

Background - Current Status

1. Building a private platform in production to enhance the visibility is an emerging tendency in decades.
2. We are prone to build an efficient and functional platform, but how?

Google Technical Report dapper-2010-1, April 2010

Dapper, a Large-Scale Distributed Systems Tracing Infrastructure

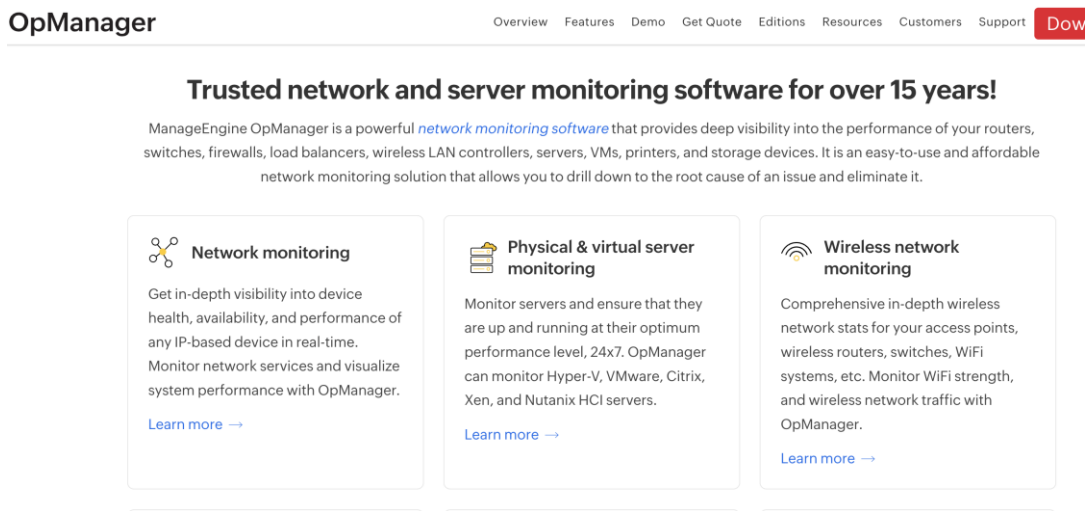
Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson,
Manoj Plakal, Donald Beaver, Saul Jaspan, Chandan Shanbhag

Abstract

Modern Internet services are often implemented as complex, large-scale distributed systems. These applications are constructed from collections of software modules that may be developed by different teams, perhaps in different programming languages, and could span many thousands of machines across multiple physical facilities. Tools that aid in understanding system behavior and reasoning about performance issues are invaluable in such an environment.

because large collections of small servers are a particularly cost-efficient platform for Internet services workloads [4]. Understanding system behavior in this context requires observing related activities across many different programs and machines.

A web-search example will illustrate some of the challenges such a system needs to address. A front-end service may distribute a web query to many hundreds of query servers, each searching within its own piece of the index. The query may also be sent to a number



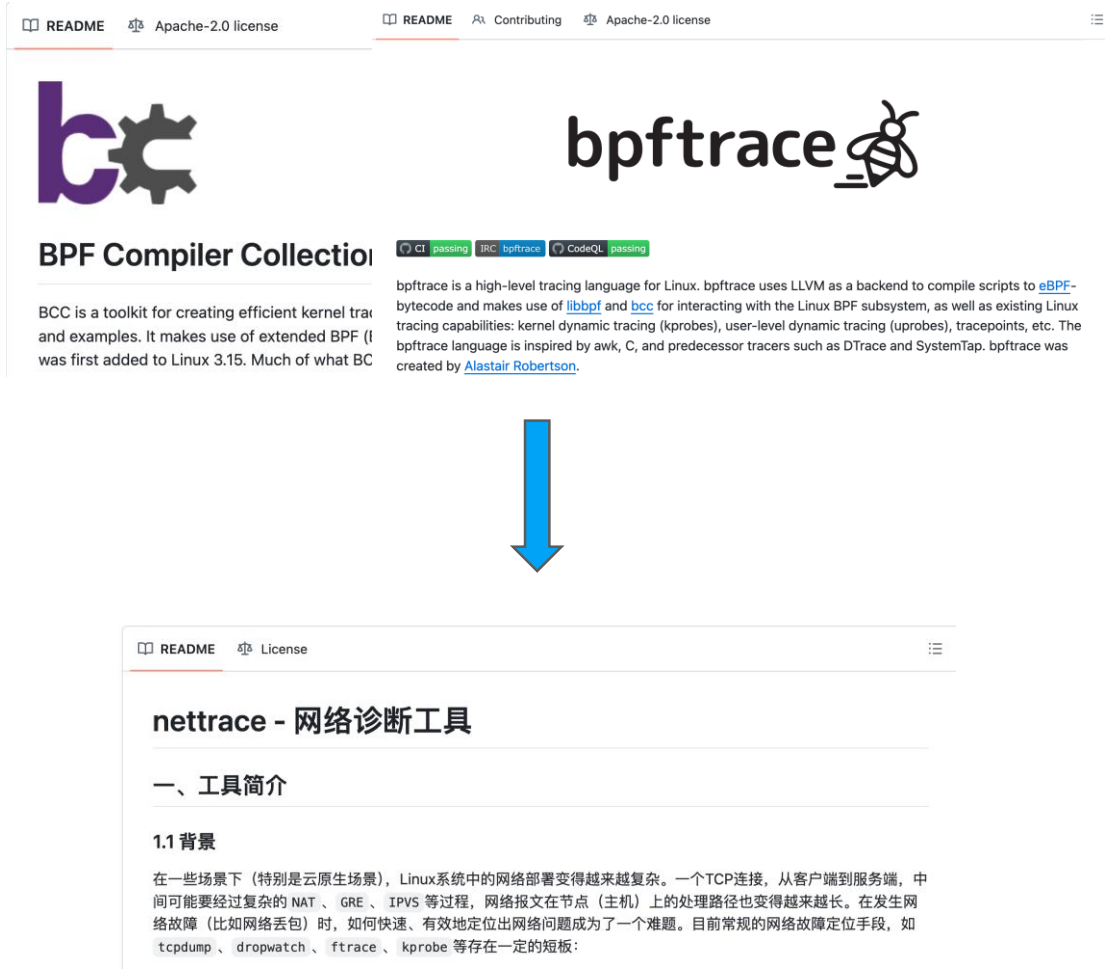
Examples

Background - Open Source Tools

Two questions:

1) How many types of tracing tools are currently available so far?
    1) bcc-based tools
    2) bpftrace-based tools
    3) libbpf-based tools
    4) kernel module tools, like diagnose-tools
    5) More complex tools, like nettrace

2) Do they meet our needs in production?
    1) requires deep understanding of kernel knowledge
    2) potential huge performance affect
    3) unavoidable interference / no isolation



open source tools

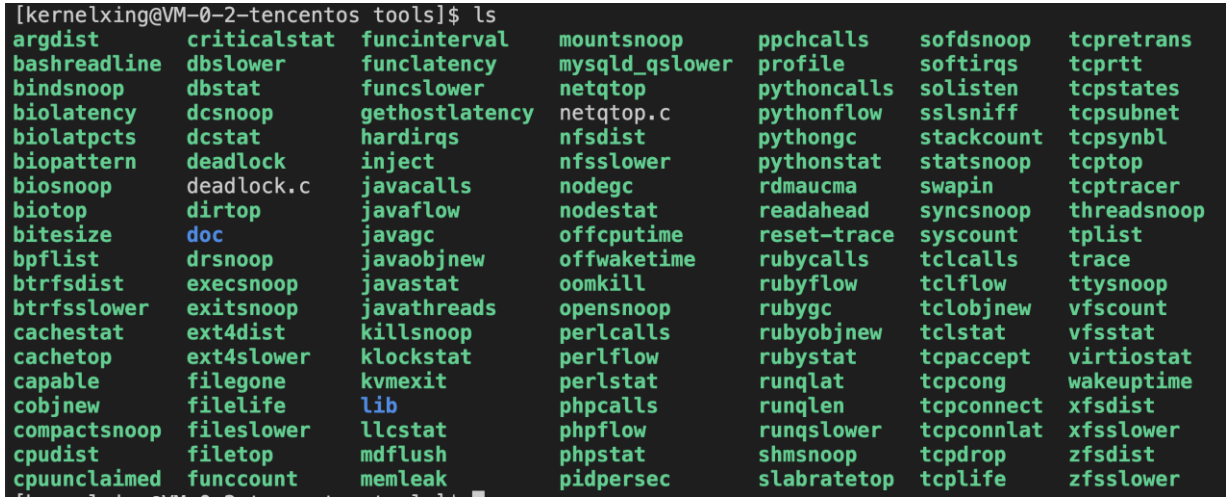
II. Theory

Theory - Triage of Issues

1. Bug or intented behavior causing problems?
2. Bug
    1. It is defined as an unintended issue caused by human error or developer oversight.
    2. It can appear unexpectedly anywhere.
    3. Numerours cases like accessing NULL pointer...
3. Intended behavior
    1. It definitely cannot be categorized as bug.
    2. Its behavior is set as developers design initially.
    3. Case 1: packets can be dropped because of nf_contrack but people who tries to find the root cause of drop instance are not awared.
    4. Case 2: TCP rst is sent due to receiving one packet that doesn't belong to any existing socket in the hashmap.

Theory - Testbed or Production Env?

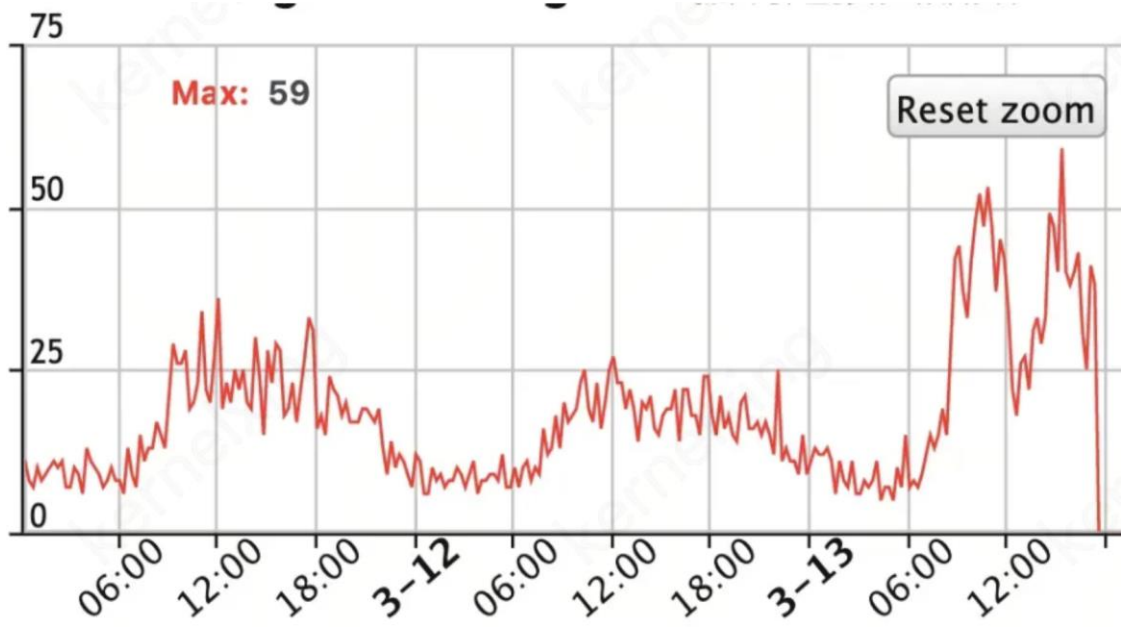
1. For testbed scenario, we can do anything we want, like hacking the kernel, writing various BPF program, using long-forgotten systemtap as long as the issues are resolved. Nearly all the open-source tracing tools are categorized into this.



BCC tools

Theory - Testbed or Production Env?

1. For testbed scenario, we can do anything we want, like hacking the kernel, writing various BPF program, using long-forgotten systemtap as long as the issues are resolved. Nearly all the open-source tracing tools are categorized into this unfortunately.
2. In production, we're faced with challenging tasks, e.g. adverse performance effects. Effects like cpu overhead, memory comsumption, performance impact, kernel panic are brought by the tools themselves.



Latency metric from our customers

Conclusion - Ultimate Goal

1. Widely and continuously running in the real workloads.
2. Suitable for all the internal kernels (even including 3.x, 4.x, 5.x old kernels).
3. Less additional overhead.
4. Zero code intrusion.
5. Effective and useful.
6. Detecting intended behavior rather than bugs is our first priority.
7. Oriented to customers, admins and network operators.
8. Address real-world issues and optimize service performance.

Theory - Metrics

1. 'nstat -s' already has so many metrics.
2. Various bpf-base tools can display anything we want.

Key Point: not all the metrics are suitable for monitoring.

1. Why - why are they not appropriate?
2. What - what type of metrics are applicable?
3. How - how to acquire them?

```
[kernelxing@VM-0-2-tencentos tools]$ nstat -s
#kernel
IpInReceives                    9281701            0.0
IpInAddrErrors                  1                  0.0
IpInDelivers                    9281700            0.0
IpOutRequests                   6114717            0.0
IpOutNoRoutes                   66                 0.0
IpOutTransmits                  6114717            0.0
IcmpInMsgs                      624793             0.0
IcmpInTimeExcds                 332                0.0
IcmpInEchos                     624457             0.0
IcmpInEchoReps                  4                  0.0
IcmpOutMsgs                     628485             0.0
IcmpOutRateLimitHost            6                  0.0
IcmpOutDestUnreachs             4023               0.0
IcmpOutEchos                    5                  0.0
IcmpOutEchoReps                 624457             0.0
IcmpMsgInType0                  4                  0.0
IcmpMsgInType8                  624457             0.0
IcmpMsgInType11                 332                0.0
IcmpMsgOutType0                 624457             0.0
IcmpMsgOutType3                 4023               0.0
IcmpMsgOutType8                 5                  0.0
TcpActiveOpens                  426278             0.0
TcpPassiveOpens                 513                0.0
TcpAttemptFails                 1                  0.0
TcpEstabResets                  10851              0.0
```

nstat -s

Theory - Ineffective Metrics

1. It does not reflect the actual anomaly.
    1. It have nothing to do with the error or exception in networking.
    2. e.g. how many bytes IP layer sends (IpExtOutOctets)

```
[kernelxing@VM-120-121-tencentos net-next]$ nstat -s | grep TcpOutSegs
TcpOutSegs                      2007999849         0.0
```

```
[kernelxing@VM-120-121-tencentos net-next]$ nstat -s | grep Ip6OutOctets
Ip6OutOctets                    329890             0.0
```

Counters for egress

Theory - Ineffective Metrics

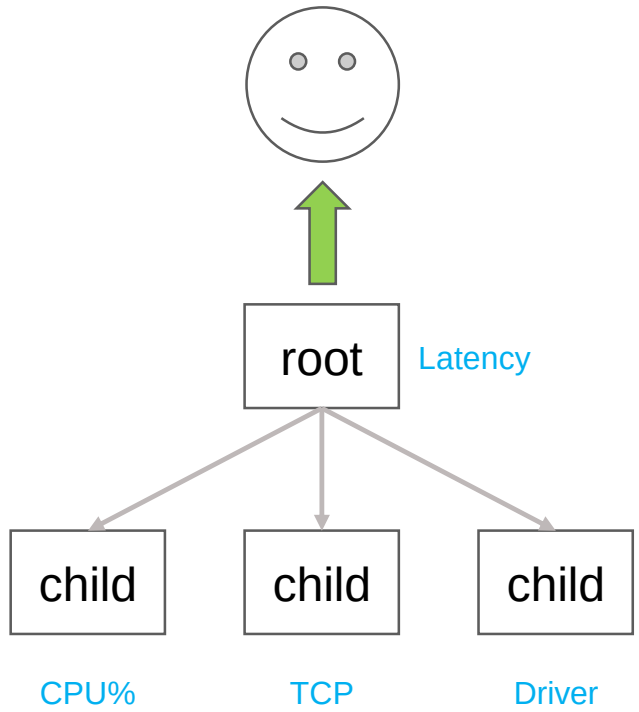
1. It does not reflect the actual anomaly.
    1. It have nothing to do with the error or exception in networking.
    2. e.g. how many bytes IP layer sends (IpExtOutOctets)
2. It does not _directly_ reflect the actual anomaly.
    1. They are so opaque that require users to dig deeper in analysis before reaching conclusion. Hence, if something wrong happens, it will likely go unnoticed.
    2. e.g. TcpExtTCPWantZeroWindowAdv, it might not a problem.

TcpExtTCPDSACKRecv	183	0.0
TcpExtTCPAbortOnData	7	0.0
TcpExtTCPAbortOnClose	10854	0.0
TcpExtTCPAbortOnTimeout	3	0.0
TcpExtTCPDSACKIgnoredNoUndo	90	0.0
TcpExtTCPSackShifted	111	0.0
TcpExtTCPSackMerged	405	0.0
TcpExtTCPSackShiftFallback	25146	0.0
TcpExtTCPRcvCoalesce	793387	0.0
TcpExtTCPOFOQueue	4181	0.0

Counters for details in TCP layer

Theory - Effective Metrics

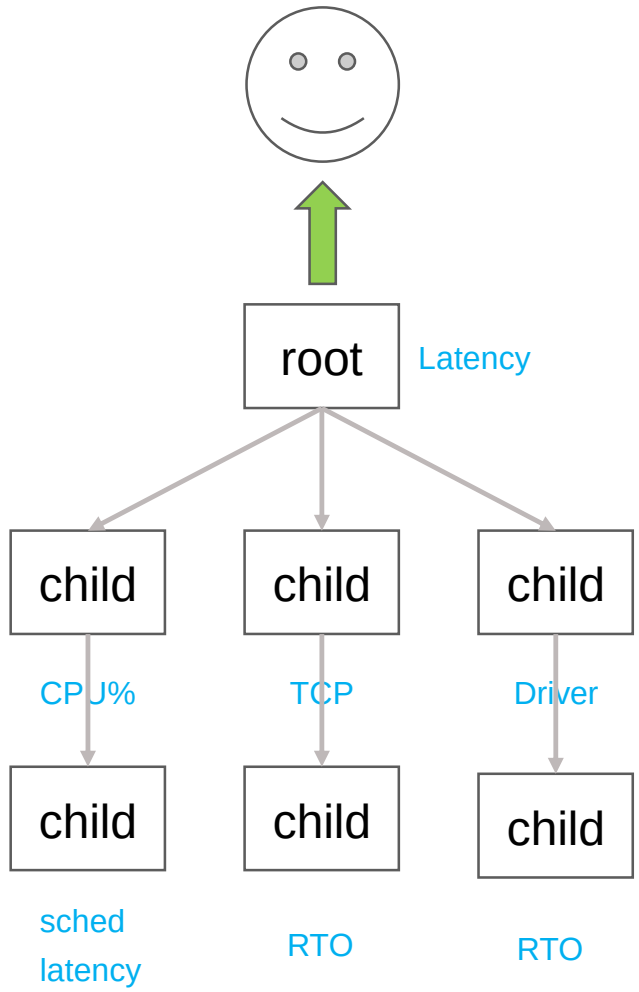
1. A N-ary tree hierarchy.
    1. Each root node (namely, effective root metric) represents unique catagory and is presented to users, showing the status of kernel networking is good or not.
    2. Children nodes reflects the details if people are willing to acquire more information.



Hierarchy of metrics

Theory - Effective Metrics

1. A N-ary tree hierarchy.
    1. Each root node (namely, effective root metric) represents unique catagory and is presented to users, showing the status of kernel networking is good or not.
    2. Children nodes reflects the details if people are willing to acquire more information.
2. It can be acquired by reading the source code and collecting issue reports.
    1. Being easily understood and straightforward.
    2. It is composed of multiple effective metrics.
        1. e.g. RTO is only one component of stall (high latency) metric, so it doesn't count as a standalone catagory. It's only the child of N-ary tree.
    3. It minimizes the gap, to the utmost extent, between the common user-space metrics and kernel abnormal behavior. It provides a possible approach that allows anyone to easily understand the kernel.



Hierarchy of metrics

Theory - Effective Children Metrics

1. Being clear-cut.
    1. Knowing it, we can dig into the exact case and then find the final solution.
    2. e.g. TcpExtTCPLostRetransmit that strongly manifests at least 200ms latency occurs.
2. It can have pre-set threshold.
    1. If the current value exceeds the pre-set threshold, it is unacceptable and it will signal upward through parent nodes until reaching the root node. I would highly recommend that let users adjust the threshold.
    2. e.g. RTO exceeds 2 times.
3. It can be composed of one or multiple (in)effective metrics
    1. Ineffective metrics can be the bottommost leaf.
    2. Multiple metrics help users guess and conclude.
    3. e.g. TcpExtTCPSpuriousRTOs doesn't show networking becomes worse, but it does show if with other metrics like RTO backoff times.
4. One ineffective metric can be turned into an effective child metric only when compared to its history behavior.
    1. By default, it is not put into the high priority list unless users/admins expects to see it.
    2. Users should be aware of what he is doing.
    3. e.g. numerous flows launches connections all of a sudden (TcpActiveOpens). It can be a question because it normally depends on different applications.

Theory - Metrics in Networking

Latency is everything!

Theory - Networking Metrics

TCP

1. memory allocation
2. delayed ack
3. RTO
4. window/buffer limitation
5. TSQ limitation

...

IP

1. customized hook points that opens the gate to allow users decides the fate of each skb

...

qdisc + driver

1. rate limiter - HTB qdisc / EDT algorithm etc
2. qdisc delay
3. BQL limitation
4. tx-usecs, tx-frames, tx-usecs-irq, tx-frames-irq
5. ...

Ingress path

1. softirq latency
2. driver
    1. coalesce: rx-usecs, rx-frames, rx-usecs-irq, rx-frames-irq
    2. budget limitation
3. IP
    1. customized hook points that opens the gate to allow users decideds the fate of each skb
4. TCP
    1. buffer limitation
    2. lock+backlog
5. epoll delay
6. wrong usage in application
7. wakeup late
8. recv syscall delay
9. sched delay

...

Theory - Networking Root Metrics at Tencent

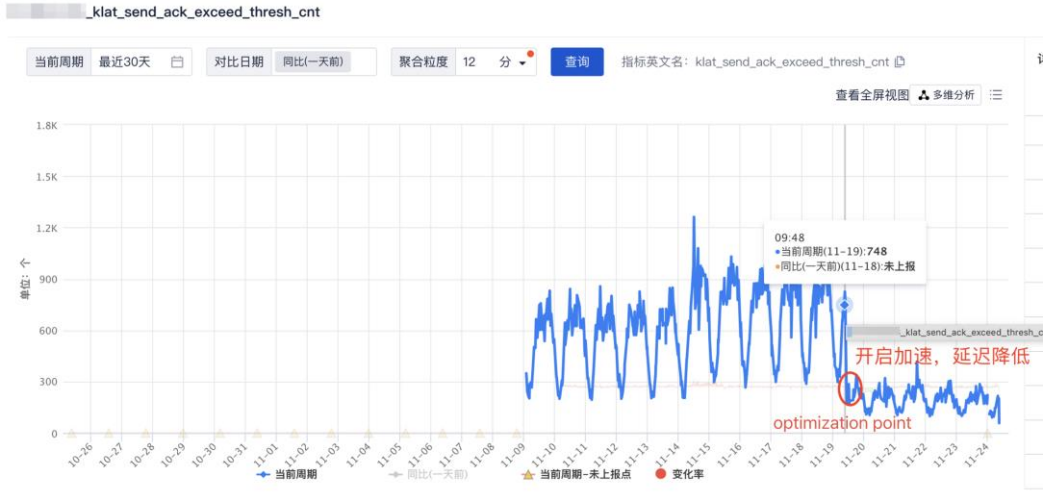
High frequency issues or reports in production can also be considered as root metrics.
At Tencent, three extended root metrics were added...

1. Performance root metric
    1. To compare different kernel/machine behaviours
        1. e.g. tuned ksoftirqd mechanism brings a gain in performance
    2. To study the performance and then optimise it
        1. e.g. qdisc latency caused by disabled TSO in VM.
        2. e.g. ADV: without consistently monitoring, we are unable to know if the feature or adjustment takes effect
2. TCP reset root metric
    1. tracing TCP reset reason, list the features and IETF stuff:
    2. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d5115a55ffb52
    3. https://datatracker.ietf.org/doc/draft-boucadair-tcpm-rst-diagnostic-payload/
3. PING delay root metric
    1. even though the most likely problem is bufferbloat which happens either in Qdisc or the link in between.
    2. Our customers often expect to see the solid/conclusive result
    3. Without BPF timestamping, it's only used for testbeds or highly tolerant production loads.

III. Practice

Practice - Ultimate Goal (recap)

1. Widely and continuously running in the real workloads.
2. Suitable for all the internal kernels (even including 3.x, 4.x, 5.x old kernels).
3. Less additional overhead.
4. Zero code intrusion.
5. Effective and useful.
6. Detecting intended behavior rather than bugs is our first priority.
7. Oriented to customers, admins and network operators.
8. Address real-world issues and optimize service performance.



NBS: networking blackboard system at Tencent

Practice - BPF Timestamping (1)

1. BPF timestamping feature is designed to equip with those functions.
2. It was derived from an internal kernel module that also considers compatability.

The future of SO_TIMESTAMPING

Speakers

Jason Xing

Label

Nuts and Bolts

Session Type

Talk

Contents

- slides
- video

Description

The future of SO_TIMESTAMPING

SO_TIMESTAMPING[1] is key to debugging network in kernel and end–to–end app latency. [2] states that by using SO_TIMESTAMPING, bugs that are otherwise incorrectly assumed to be network issues can be attributed to the kernel.

There are a few areas that need optimization for usability and performance[3]. These include: uAPI compatibility, extra system call overhead, and the need for application modification. Our initial solution to solve these issues constituted writing a kernel module that hooks various key functions. However, this approach is not suitable to land in the kernel officially, even though it has been deployed successfully in production. For this reason and based on feedback we took an approach of going with am eBPF extension approach.

https://netdevconf.info/0x19/sessions/talk/the-future-of-so_timestamping.html

Practice - BPF Timestamping (2)

Hook

1. Find a lightweight approach.
2. We uses ftrace to hook because it's a kernel module.
3. The best choice should be fentry but we have older kernels.

```
$ cd tools/testing/selftests/bpf
$ ./benchs/run_bench_trigger.sh
usermode-count :  890.171 ± 1.522M/s
kernel-count   :  409.184 ± 0.330M/s
syscall-count  :   26.792 ± 0.010M/s
fentry         :  171.242 ± 0.322M/s
fexit          :   80.544 ± 0.045M/s
fmodret        :   78.301 ± 0.065M/s
rawtp          :  192.906 ± 0.900M/s
tp             :   81.883 ± 0.209M/s
kprobe         :   52.029 ± 0.113M/s
kprobe-multi   :   62.237 ± 0.060M/s
kprobe-multi-all:    4.761 ± 0.014M/s
kretprobe      :   23.779 ± 0.046M/s
kretprobe-multi:   29.134 ± 0.012M/s
kretprobe-multi-all:    3.822 ± 0.003M/
```

https://lore.kernel.org/bpf/20251118123639.688444-1-dongml2@chinatelecom.cn/

Practice - BPF Timestamping (3)

Hook

1. We uses ftrace to hook because it's a kernel module.
2. The best choice should be fentry but we have older kernels.

Process

1. We set the special indicator at the very beginning so that it will not affect the hot path.
2. We need to filter some flows and process massive data and calculate some results.
3. In the hot paths, we make sure the code is simple as much as we can, or else it will causes performance degradation.

```
51
52 static int bpf_test_sockopt(void *ctx, const struct sock *sk, int expected)
53 {
54         int tmp, new = SK_BPF_CB_TX_TIMESTAMPING;
55         int opt = SK_BPF_CB_FLAGS;
56         int level = SOL_SOCKET;
57
58         if (bpf_setsockopt(ctx, level, opt, &new, sizeof(new)) != expected)
59                 return 1;
60
61         if (bpf_getsockopt(ctx, level, opt, &tmp, sizeof(tmp)) != expected ||
62             (!expected && tmp != new))
63                 return 1;
64
65         return 0;
66 }
67
```

tools/testing/selftests/bpf/progs/net_timestamping.c

Practice - BPF Timestamping (4)

Hook

1. We uses ftrace to hook because it's a kernel module.
2. The best choice should be fentry but we have older kernels.

Process

1. We set the special indicator at the very beginning so that it will not affect the hot path.
2. We need to filter some flows and process massive data and calculate some results.
3. In the hot paths, we make sure the code is simple as much as we can, or else it will causes performance degradation.

Log

1. We need a high performance way to tranfer data from kernel to userspace, like relayfs, BPF_MAP_TYPE_ARRAY_OF_MAPS + BPF_MAP_TYPE_RINGBUF.
2. net_timestamping will be hopefully replaced by the more effcient map.

```c
static bool bpf_test_delay(struct bpf_sock_ops *skops, const struct sock *sk)
{
        struct bpf_sock_ops_kern *skops_kern;
        u64 timestamp = bpf_ktime_get_ns();
        struct skb_shared_info *shinfo;
        struct delay_info dinfo = {0};
        struct sk_tskey key = {0};
        struct delay_info *val;
        struct sk_buff *skb;
        struct sk_stg *stg;
        u64 prior_ts, delay;

        if (bpf_test_access_bpf_calls(skops, sk))
                return false;

        skops_kern = bpf_cast_to_kern_ctx(skops);
        skb = skops_kern->skb;
        shinfo = bpf_core_cast(skb->head + skb->end, struct skb_shared_info);

        key.cookie = bpf_get_socket_cookie(skops);
        if (!key.cookie)
                return false;

        if (skops->op == BPF_SOCK_OPS_TSTAMP_SENDMSG_CB) {
                stg = bpf_sk_storage_get(&sk_stg_map, (void *)sk, 0, 0);
                if (!stg)
                        return false;
                dinfo.sendmsg_ns = stg->sendmsg_ns;
                bpf_sock_ops_enable_tx_tstamp(skops_kern, 0);
                key.tskey = shinfo->tskey;
                if (!key.tskey)
                        return false;
                bpf_map_update_elem(&time_map, &key, &dinfo, BPF_ANY);
                return true;
        }

        key.tskey = shinfo->tskey;
        if (!key.tskey)
                return false;

        val = bpf_map_lookup_elem(&time_map, &key);
        if (!val)
                return false;
```

tools/testing/selftests/bpf/progs/net_timestamping.c

Practice - BPF Programs

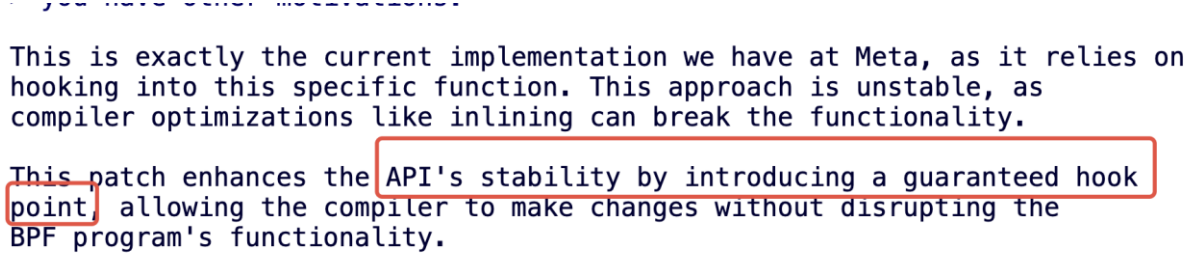
1. BPF programs are used to cover those specific issues, like connection delay.
2. They are used to being the effective children metrics, directing us on how to move forward.
3. Classic issues like high scheduler latency:
    1. Revert "softirq: Let ksoftirqd do its job" https://git.kernel.org/pub/scm/linux/kernel/git/tip/tip.git/commit/?id=d15121be7485655129101f3960ae6add40204463

```
#!/usr/bin/bpftrace
#include <linux/types.h>
/*
 * sudo bpftrace ptype_all.bt 1
 */
BEGIN
{
        $head = (struct list_head*)*kaddr("ptype_all");
        $cur = $head->next;
        $i = 1;
        printf("addr: %p, %p\n", $head, $cur);
        while ($cur !=0 && $cur != $head && $i < 1000) {
                $cur = $cur->next;
                $i++;
        }
        if ($i < 200) {
                printf("[INFO] ptype_all length: %d\n", $i);
        } else if ($i < 1000) {
                printf("[WARNING] ptype_all length: %d\n", $i);
        } else {
                printf("[WARNING] ptype_all longer than 1000\n")
        }
        @threshold = (uint64)200;
        printf("Tracing dec_queue_xmit_nit longer than %dus. Hit Ctrl-C to end.\n", @threshold);
}
```

ptype_all issue

Key Concerns for the Future

1. Many people have different goals to monitor different functions. Sometimes functions get inlined or renamed which leads to the change in the monitoring system.
2. Adding stable tracepoints that have been used at Meta for many years: https://lore.kernel.org/all/20250214-cwnd_tracepoint-v2-1-ef8d15162d95@debian.org/
3. Do we need to have KABI-like rules? Please see at https://lore.kernel.org/all/20250120-daring-outstanding-jaguarundi-c8aaed@leitao/



```
This is exactly the current implementation we have at Meta, as it relies on
hooking into this specific function. This approach is unstable, as
compiler optimizations like inlining can break the functionality.

This patch enhances the API's stability by introducing a guaranteed hook
point, allowing the compiler to make changes without disrupting the
BPF program's functionality.
```

One of replies from Breno

Thank you!

Any questions?