Network Observability BoF

Netdev 0x1A Roma

Jason Xing <kernelxing@tencent.com>
Jamal Hadi Salim <jhs@mojatatu.com>

Content

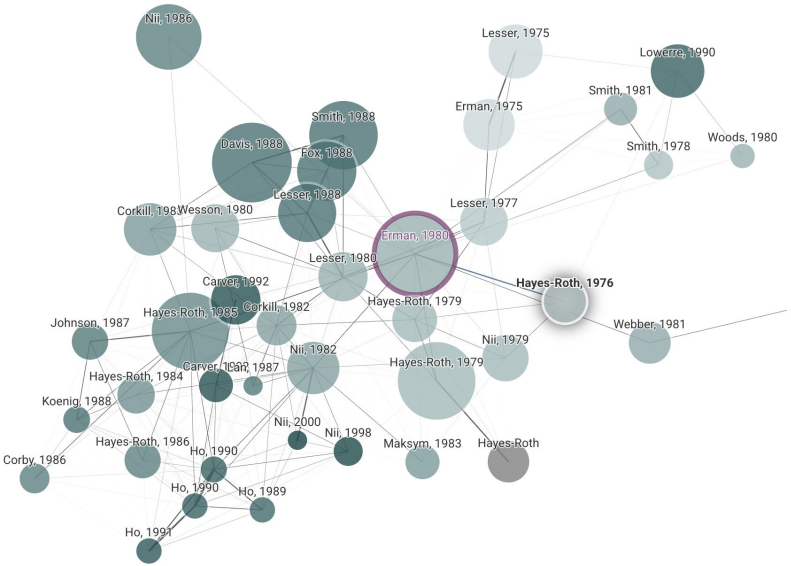
1. Motivation && Kick off
2. Long History
3. Current Status
4. Subtopics
    a. BPF Timestamping 2.0: A Novel kernel Latency Monitoring Framework
    b. More?

1. Motivation && Kick Off

1. **Long history**: ‘Observatility’ has been a very hot/well-discussed topic especially for various customers, admins, kernel developers and researchers since the last decade.
2. **An aha moment**: I once tried to talk with Jakub about the BPF timestamping and get some feedbacks on this. Jakub suggested why not at a broader conference? Yes, I think, why not then make it bigger and call for more proposals on Network Observability instead? So I reached out to Jamal, then the story has begun…
3. **LPC Linux System Monitoring and Observability MC**: We (Breno Leitao, Usama Arif, me) will hold the 2nd edition at LPC 2026. Last year, we received many interesting topics not only in the kernel community.
   a. LPC 2025: https://lpc.events/event/19/contributions/2010/
   b. LPC 2026: https://lpc.events/event/20/sessions/262/
4. **Goal**: bring BoF people together, find the real requirements, discuss the direction, contribute to the networking community!

2. Long History (1)

1. ‘Blackboard’ since ~1980
   a. “ KS is an independent condition-action module, KSs communicate through a global database called the blackboard.”
      i. KS: Knowledge Source
   b. 1980: Integrating Knowledge to Resolve Uncertainty
   c. 1988: Knowledge-based monitoring and control: an approach to understanding the behavior of tcp/ip network protocols



2. Long History (2)

1. Components of Blackboard
   a. Meta-Plan Plane - “problem definition, problem-solving model, policies, and evaluation criteria.”
   b. Knowledge Base Plane - “the levels of the knowledge base contain problem-specific information, we have given them problem-specific names”
   c. [A Cognitive Model of Planning]

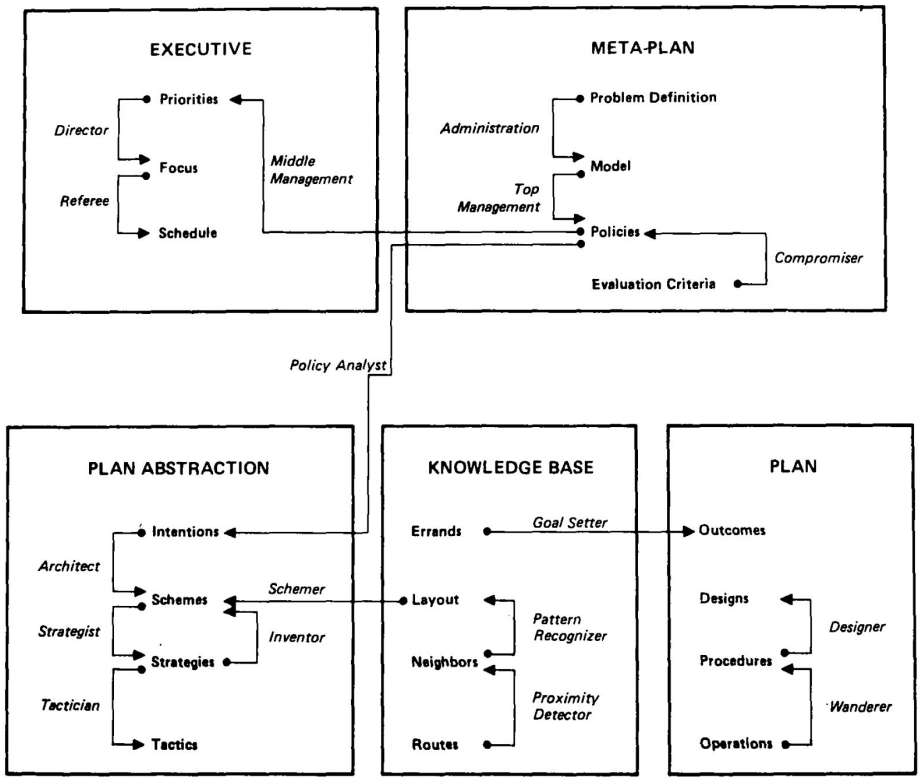
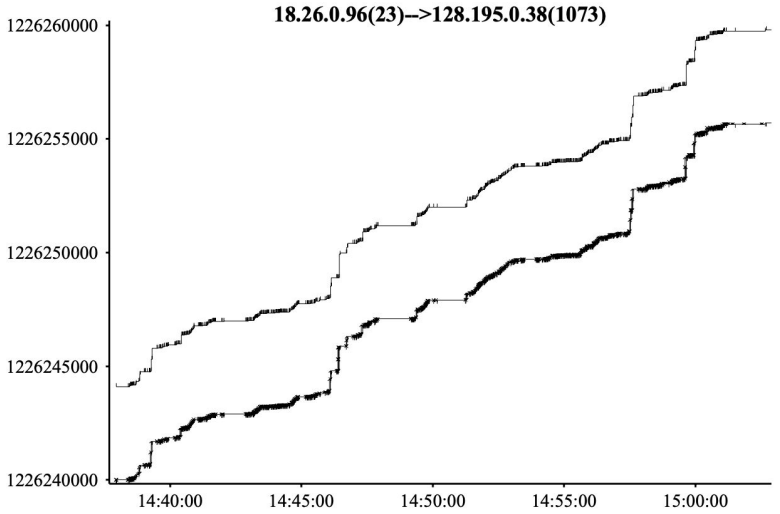


Figure 3. The planning blackboard and the actions of illustrative specialists.

2. Long History (3)

1. TCP analyzer (MIT)
   a. Observation of TCP
   b. “to diagnose performance problems of the transport protocol TCP”
   c. TCP Packet Trace Analysis



This is a time-sequence plot of the host-to-user half of a typical TELNET connection from a user on a distant host. Over twenty minutes of the connection is shown here. Most of the TCP data segments are short and probably correspond to remote echoes of typed characters. At this scale, these short segments are all blurred together on the ack line. Occasionally larger bursts of traffic can be seen. These are probably output from a program.

Figure 3.3: Telnet connection

2. Long History (4)

1. Tcpdump since 1988
    a. It has become a widely adopted traffic monitoring tool

Year	Event
1987	*The Packet Filter: An Efficient Mechanism for User-level Network Code*
1988	Tcpdump Release
1992	The BSD Packet Filter: A New Architecture for User-level Packet Capture
2008	Wireshark Release
2011	libpcap: An Architecture and Optimization Methodology for Packet Capture
2020 - untill now	PCAP Now Generic (pcapng) Capture File Format

2. Long History (5)

1. Directions since last century
    a. Blackboard system
    b. Protocol analyzer

3. Current Status

1. Blackboard System
    a. Application-level monitoring
        i. distributed system is one of cases.
        ii. The monitoring also includes the kernel latency/network latency.
    b. System-level monitoring
        i. The goal is to get a better insight of the kernel behavior
    c. Switch-level monitoring
        i. Active prober sending probe packets at set intervals.
        ii. It can be deployed in either switch side or host side

3. Current Status

1. Blackboard System
    a. Google:
        i. Dapper, a large-scale distributed systems tracing infrastructure
        ii. Fathom: Understanding Datacenter Application Network Performance
    b. Facebook/Meta
        i. Inside the Social Network's (Datacenter) Network
        ii. NetNORAD: Troubleshooting networks via end-to-end probing
        iii. Canopy: An end-to-end performance tracing and analysis system
    c. Microsoft
        i. Pingmesh: A large-scale system for data center network latency measurement and analysis
    d. VMware
        i. How to diagnose nanosecond network latencies in rich end-host stacks
    e. Bytedance
        i. Bytetracker: An agentless and real-time path-aware network probing system
        ii. R-pingmesh: A service-aware roce network monitoring and diagnostic system
    f. Alibaba
        i. Tcprt: Instrument and diagnostic analysis system for service quality of cloud databases at massive scale in real-time
    g. deepflow
        i. Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code

3. Current Status

1. Kernel Community
    a. Types of existing approaches
        i. MIB
        ii. tracepoint
        iii. BPF-based tools
        iv. protocol specific tools
            1. Packetdrill
                a. github repo
                b. 2025: Tutorial: Using Packetdrill to Write Automated Tests for the Linux Networking Stack
                c. 2013: packetdrill: Scriptable Network Stack Testing, from Sockets to Packets

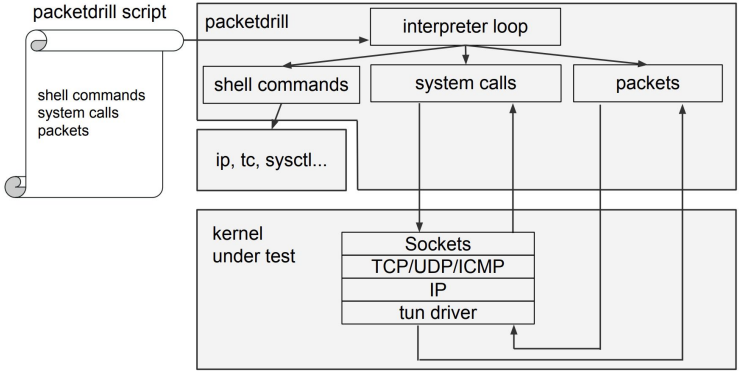


Figure 1: design and implementation

3. Current Status

1. Kernel Community
    a. Types of existing approaches
    b. Types of issues
        i. misconfigurations
            1. iptables rules
            2. wrong use of syscall
            3. …
            4. It’s simple to pinpoint the issues with our ‘KS’ (Knowledge Source)

3. Current Status

1. Kernel Community
    a. Types of existing approaches
    b. Types of issues
        i. misconfigurations
        ii. reference count leak
            1. We've already had lots of reports showing the reference count that leads to crashs/memory leak…
            2. Eric proposed a detect framework in 2021: commit 9ba74e6c9e9d0 ("net: add networking namespace refcount tracker")
            3. lwn: A reference-count tracking infrastructure

2. lib: add reference counting tracking infrastructure: https://git.kernel.org/pub/scm 6934eaadc83b27ada8d42b60894018f3bfabf
3. lib: add tests for reference tracker: https://git.kernel.org/pub/scm/linux/kernel/git 87e0efa52f27c4b7ea75b893
4. net: add net device refcount tracker infrastructure: https://git.kernel.org/pub/scm 2b95ff2f95f13df9bad0b5a25a9f60e72758d
5. net: add net device refcount tracker to struct netdev_rx_queue: https://git.kerne mmit/?id=80e8921b2b72c300ca56a01729004d30bedb82cd
6. net: add net device refcount tracker to struct netdev_queue: https://git.kernel.or t/?id=0b688f24b7d611db3a02f3d4ab562d049c78a17d
7. drop_monitor: add net device refcount tracker: https://git.kernel.org/pub/scm/lin f65c60259ce5d1563433ecaf5fab693c83
8. net: dst: add net device refcount tracking to dst_entry: https://git.kernel.org/pub 9038c320001dd07f60736018edf608ac5baca0ab
9. net: add networking namespace refcount tracker: https://git.kernel.org/pub/scm/ 4e6c9e9d0c5c1e5792a7111fc7d1a0589cb8
10. net: add netns refcount tracker to struct sock: https://git.kernel.org/pub/scm/lir 5ffb37a957d6062385112ab1069f760de6
11. net: add netns refcount tracker to struct seq_net_private: https://git.kernel.org/ t/?id=04a931e58d1944ab3d1e11fdfde1947fbe5b6a37

Figure 1: a lot of patches are coming in since 2021

3. Current Status

1. Kernel Community
    a. Types of existing approaches
    b. Types of issues
        i. misconfigurations
        ii. reference count leak
        iii. memory leak
            1. BPF-based tools that we often use, like /usr/share/bcc/tools/memleak

```
EXAMPLES:

./memleak -p $(pidof allocs)
        Trace allocations and display a summary of "leaked" (outstanding)
        allocations every 5 seconds
./memleak -p $(pidof allocs) -t
        Trace allocations and display each individual allocator function call
./memleak -ap $(pidof allocs) 10
        Trace allocations and display allocated addresses, sizes, and stacks
        every 10 seconds for outstanding allocations
./memleak -c "./allocs"
        Run the specified command and trace its allocations
./memleak
        Trace allocations in kernel mode and display a summary of outstanding
        allocations every 5 seconds
./memleak -o 60000
        Trace allocations in kernel mode and display a summary of outstanding
        allocations that are at least one minute (60 seconds) old
./memleak -s 5
        Trace roughly every 5th allocation, to reduce overhead
./memleak --sort count
        Trace allocations in kernel mode and display a summary of outstanding
        allocations that are sorted in count order
```

Figure 1: bcc tools - memleak

3. Current Status

1. Kernel Community
    a. Types of existing approaches
    b. Types of issues
        i. misconfigurations
        ii. reference count leak
        iii. memory leak
        iv. crash
            1. Nowadays, AI+crash analyzer which is already a production-ready tool help developers pinpoint the issue efficiently.
            2. crash-utility/crash github repo
            3. drgn , drgn-tools

3. Current Status

1. Kernel Community
    a. Types of existing approaches
    b. Types of issues
        i. misconfigurations
        ii. reference count leak
        iii. memory leak
        iv. crash
        v. latency
            1. socket timetamping
            2. BPF timestamping 1.0
            3. BPF-based tools like [nettrace]

```
$ ./nettrace --tiny-show -p tcp --port 9999
begin trace...
***************** 63618400,636184e8 ***************
[11485.683405] [__tcp_transmit_skb  ] TCP: 127.0.0.1:39642 -> 127.
[11485.683413] [skb_clone           ]
[11485.683416] [__ip_queue_xmit     ]
[11485.683418] [nf_hook_slow        ]
[11485.683423] [nft_do_chain        ]
[11485.683425] [ip_output           ]
[11485.683427] [nf_hook_slow        ]
[11485.683428] [nft_do_chain        ]
[11485.683432] [ip_finish_output    ]
[11485.683434] [ip_finish_output2   ]
[11485.683435] [__dev_queue_xmit    ]
[11485.683437] [dev_hard_start_xmit ]
[11485.683438] [enqueue_to_backlog  ]
[11485.683441] [__netif_receive_skb_core.constprop.0]
[11485.683442] [ip_rcv              ]
[11485.683444] [ip_rcv_core         ]
[11485.683445] [nf_hook_slow        ]
[11485.683446] [ip_local_deliver    ]
```

Figure 1: nettrace

3. Current Status

1. Kernel Community
    a. Types of existing approaches
    b. Types of issues
        i. misconfigurations
        ii. reference count leak
        iii. memory leak
        iv. crash
        v. latency
        vi. skb drop
            1. David Ahern suggested introducing skb drop reason feature.
            2. Menglong Dong completed the first edition: Merge branch 'net-skb-introduce-kfree_skb_with_reason'

```
+/* The reason of skb drop, which is used in kfree_skb_reason().
+ * en...maybe they should be splited by group?
+ *
+ * Each item here should also be in 'TRACE_SKB_DROP_REASON', which is
+ * used to translate the reason to string.
+ */
+enum skb_drop_reason {
+       SKB_DROP_REASON_NOT_SPECIFIED,
+       SKB_DROP_REASON_NO_SOCKET,
+       SKB_DROP_REASON_PKT_TOO_SMALL,
+       SKB_DROP_REASON_TCP_CSUM,
+       SKB_DROP_REASON_TCP_FILTER,
+       SKB_DROP_REASON_UDP_CSUM,
+       SKB_DROP_REASON_MAX,
+};
```

Figure 1: drop reasons

3. Current Status

1. Kernel Community
    a. Types of existing approaches
    b. Types of issues
        i. misconfigurations
        ii. reference count leak
        iii. memory leak
        iv. crash
        v. latency
        vi. skb drop
        vii. protocol specific
            1. e.g. [TCP RST Diagnostic Payload: draft-ietf-tcpm-rst-diagnostic-payload-02]

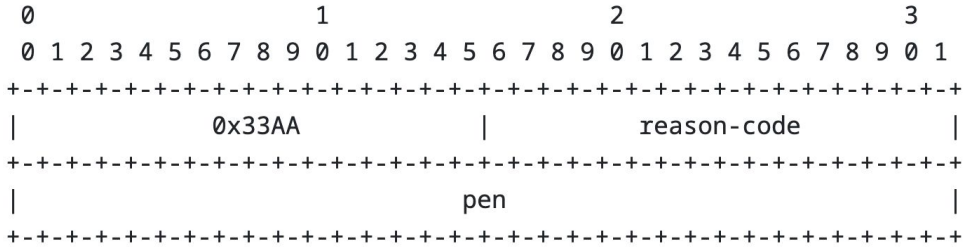


Figure 1: Structure of the RST Diagnostic Payload

A Novel Kernel Latency Monitoring Framework

<BPF Timestamping 2.0>

Jason Xing <kernelxing@tencent.com>

Content

1. Methodology (before/after)
2. Socket Timestamping
3. BPF Timestamping 1.0
4. BPF Timestamping 2.0
5. The Future

1. Methodology (before/after)

1. Before (1.0)
    a. Knowledge Source
        i. metrics <----------|
        ii. tracepoints |
        iii. kernel module/bpf |
        iv. reading source code | (again)
        v. The pain of the traditional approach
            1. Existing metrics cover only a fraction of possible failure events
            2. BPF investigation requires deep kernel knowledge & is time-consuming
            3. The gap between Phase 1 and Phase 2 is where most time is wasted
            4. **<u>It's really hard to make everything run automatically.</u>**

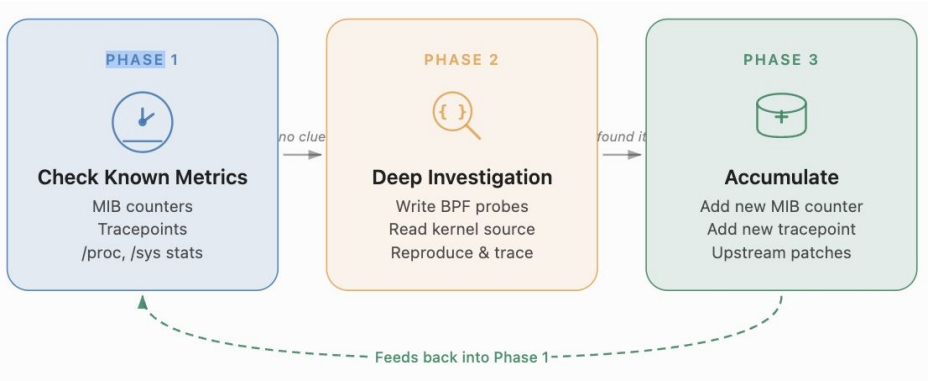


Figure 1: the process of root-causing issues

1. Methodology (before/after)

1. Before (1.0)
   a. Knowledge Source
      i. metrics <----------|
      ii. tracepoints |
      iii. kernel module/bpf |
      iv. reading source code | (again)
   b. why is it not effective any more?
      i. Slowing trend
         1. the growing speed of the number in these two decades becomes very slow.
         2. We cannot afford adding all the potential points.

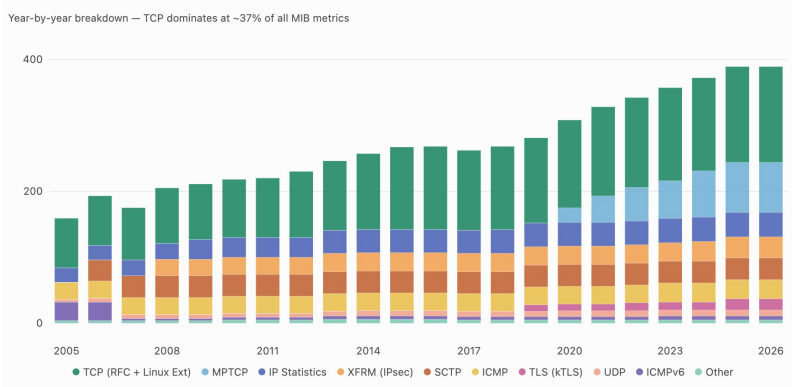


Figure 1: Metrics Statistics

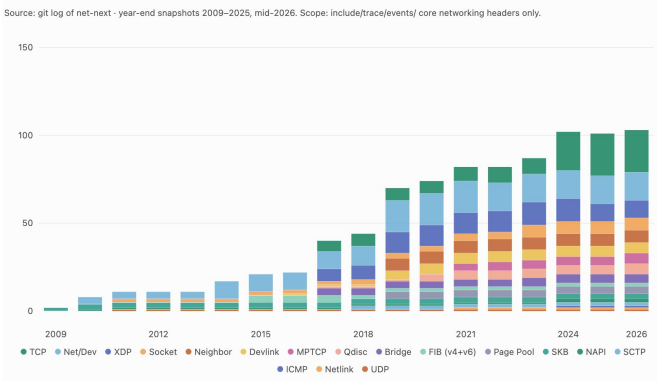


Figure 2: Tracepoint Statistics

1. Methodology (before/after)

1. Before (1.0)
    a. Knowledge Source
        i. metrics <----------|
        ii. tracepoints |
        iii. kernel module/bpf |
        iv. reading source code | (again)
    b. why is it not effective any more?
        i. Slowing trend
        ii. Coverage Blindspot
            1. it’s not possible to cover 100% problems via metrics/tracepoints
            2. metrics only reflect those classic and known issues
            3. we're not encouraged to add more tracepoints
            4. RTO metric is an good example that doesn’t help us understand which condition leads to the issue (without the help of Figure 2).

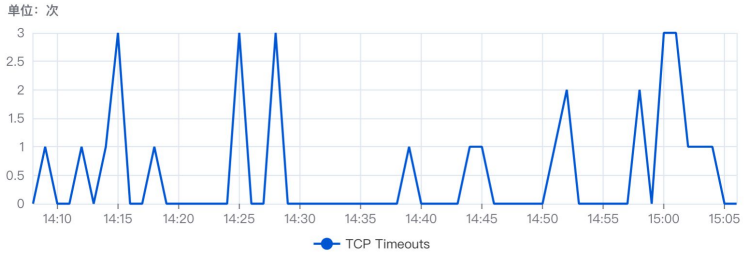


Figure 1: TCP RTO Events

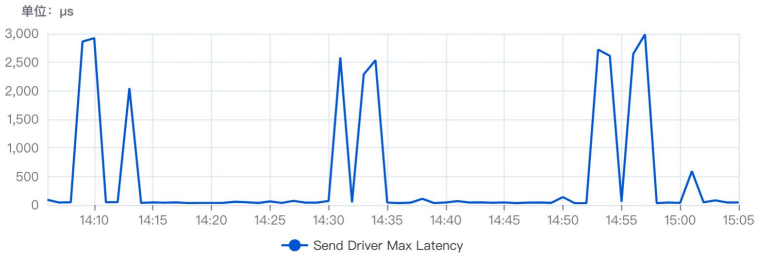


Figure 2: Latency between send syscall and driver xmit

1. Methodology (before/after)

1. Before (1.0)
    a. Knowledge Source
        i. metrics <----------|
        ii. tracepoints |
        iii. kernel module/bpf |
        iv. reading source code | (again)
    b. why is it not effective any more?
        i. Slowing trend
        ii. Coverage Blindspot
        iii. Storage Problem
            1. It's not affordable for large scale deployment
                a. 0.5 KB/metric × 2M nodes × 100 metrics x (24 hours x 6) = 14.4 PB/day
            2. 95% collected metrics are USELESS!

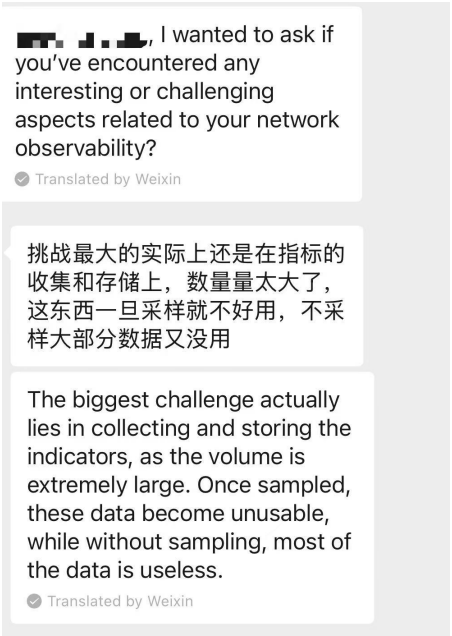


Figure 1: One short conversation with my friend when I'm writing this slide

1. Methodology (before/after)

1. Before (1.0)
2. After (2.0)
    a. Effective Root Metrics Theory
        i. What is the root metric?
        ii. Why ‘latency’?
        iii. How to trace ‘latency’?
        iv. LPC 2025 - Methodology and Practice in Observing Kernel Networking

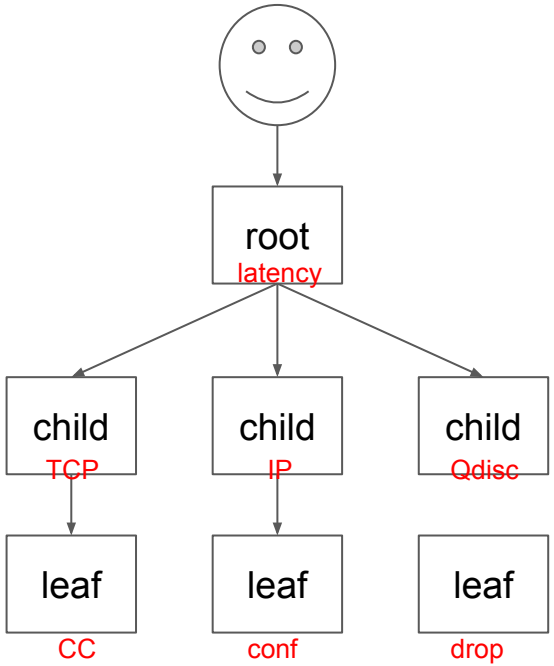


Figure 1: what does a metrics tree look like?

1. Methodology (before/after)

1. Before (1.0)
2. After (2.0)
    a. Effective Root Metrics Theory
    b. Troubleshooter VS Designer Theory
        i. [1st] Attribution
            1. kernel Level
            2. Layer Level
            3. Function Level
        ii. [2nd] Knowledge Source
            1. metrics
            2. tracepoint
            3. kernel module/bpf
            4. reading source code

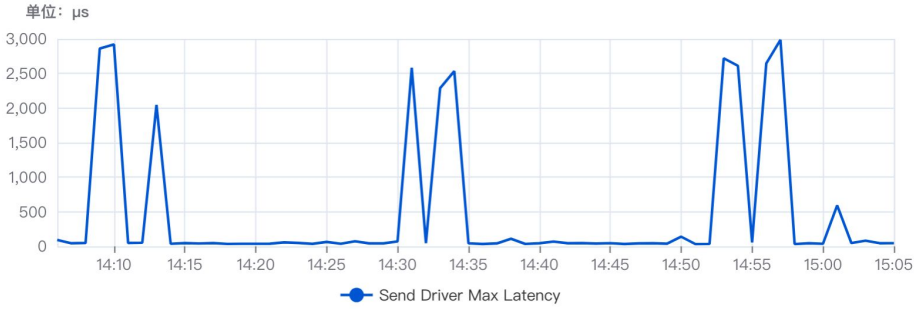


Figure 1: Latency from ‘send’ to ‘driver’

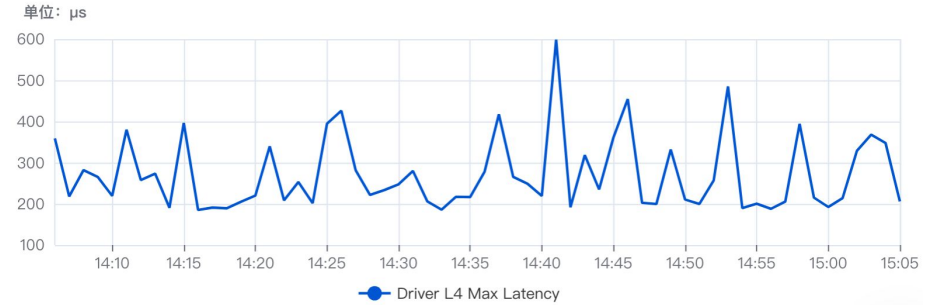


Figure 2: Latency from ‘driver’ to ‘tcp rcv’

1. Methodology (before/after)

1. Before (1.0)
2. After (2.0)
   a. Effective Root Metrics Theory
   b. Troubleshooter VS Designer Theory
   c. Contract
      i. a new contract akin to tracepoint that we tell people “just trace those functions then” should be explored.
      ii. How about fentry + kfunc (to print the current ‘latency’)?
      iii. example 1: [trace: tcp: Add tracepoint for tcp_cwnd_reduction()]
      iv. example 2: [trace: tcp: Add tracepoint for tcp_cwnd_reduction()]

```
>
> I meant, in the position where you insert a one-line tracepoint, which
> should be easily replaced with a bpf program (kprobe
> tcp_cwnd_reduction with two checks like in the earlier if-statement).
> It doesn't mean that I object to this new tracepoint, just curious if
> you have other motivations.

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.
```

Figure 1: Breno’s reply in the example 1

```
Add a lightweight tracepoint to monitor TCP congestion window
adjustments via tcp_cwnd_reduction(). This tracepoint enables tracking
of:
- TCP window size fluctuations
- Active socket behavior
- Congestion window reduction events

Meta has been using BPF programs to monitor this function for years.
Adding a proper tracepoint provides a stable API for all users who need
to monitor TCP congestion window behavior.

Use DECLARE_TRACE instead of TRACE_EVENT to avoid creating trace event
infrastructure and exporting to tracefs, keeping the implementation
minimal. (Thanks Steven Rostedt)

Given that this patch creates a rawtracepoint, you could hook into it
using regular tooling, like bpftrace, using regular rawtracepoint
infrastructure, such as:
```

Figure 2: Breno’s reply in the example 2

2. Socket Timestamping

1. History
   a. Keyword: latency monitoring
   b. **Patrick Ohly** implemented the first edition of socket timestamping in 2009.
   c. **Willem de Bruijn** fulfilled and enhanced socket timestamping in every aspect in 2014.
   d. References:
      i. Timestamping kernel Doc
      ii. 2026: RPC Latency Breaker: Where Did My RPC Time Go?
      iii. 2025: The future of SO_TIMESTAMPING
      iv. 2024: Extending SO_TIMESTAMPING feature
      v. 2023: SO_TIMESTAMPING: powering fleetwide RPC monitoring
      vi. 2019: TCP SO_TIMESTAMPING with OPT_STATS for performance analytics
      vii. 2019: TCP Analytics

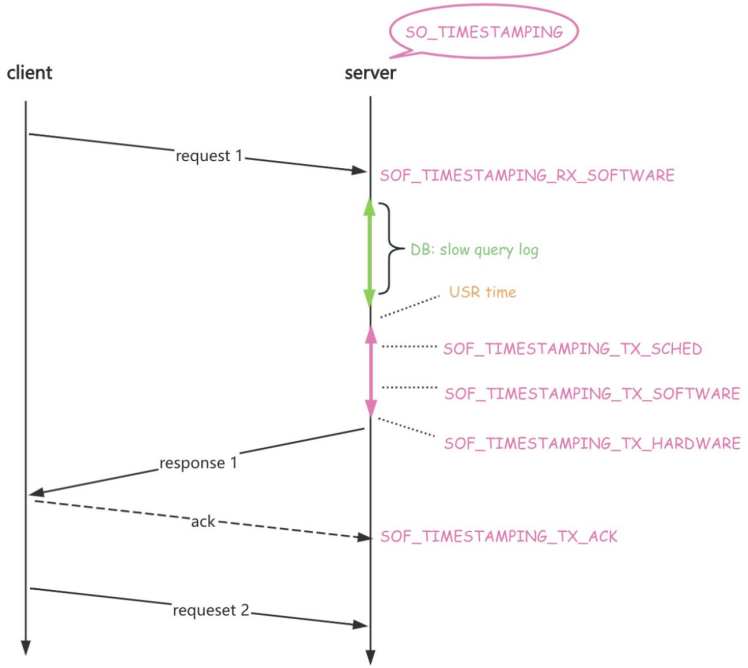


Figure 1: one example of RPC using socket timestamping

2. Socket Timestamping

1. History
2. How to use?
   a. What is SO_TIMESTAMPING feature? From Willem: Timestamping is key to debugging network stack latency. With SO_TIMESTAMPING, bugs that are otherwise incorrectly assumed to be network issues can be attributed to the kernel. It can isolate transmission, reception and even scheduling sources.
   b. Applications have the ability to use this feature through setsockopt, expecting to study and analyze closely in kernel behavior. Then the jitter issue can be effortlessly traced down to which layer is the cause.

```c
// 1. Enable timestamping
int flags = SOF_TIMESTAMPING_RX_SOFTWARE
          | SOF_TIMESTAMPING_TX_SOFTWARE
          | SOF_TIMESTAMPING_SOFTWARE;
setsockopt(fd, SOL_SOCKET, SO_TIMESTAMPING, &flags, sizeof(flags));

// 2. Send (for TX timestamp)
send(fd, data, len, 0);

// 3. Receive timestamp via recvmsg + cmsg
struct msghdr msg = { .msg_control = cbuf, .msg_controllen = sizeof(cbuf), ... };
recvmsg(fd, &msg, MSG_ERRQUEUE);  // TX timestamps come from error queue
                                  // RX timestamps come from normal recvmsg()

for (struct cmsghdr *cm = CMSG_FIRSTHDR(&msg); cm; cm = CMSG_NXTHDR(&msg, cm)) {
    if (cm->cmsg_level == SOL_SOCKET && cm->cmsg_type == SO_TIMESTAMPING) {
        struct timespec *ts = (struct timespec *)CMSG_DATA(cm);
        // ts[0] = software timestamp
        // ts[1] = hw timestamp (transformed)
        // ts[2] = hw timestamp (raw)
    }
}
```

Figure 1: example in selftests

2. Socket Timestamping

1. History
2. How to use?
3. Limitations
   a. Applications need changes
      i. Modify the params in setsockopt/sendmsg/recvmsg
      ii. for(...) loops to parse the cmsgs
   b. Additional overhead per recv call
      i. Syscalls: 0-2 times setsockopt() + X times recvmsg() (X is the number of tx generation flags)
      ii. Extra analysis in apps
   c. Inflexibility
      i. Limited information to output unless we add more fields in kernel and then upgrade the kernel.
   d. Uapi limits the future
      i. If we have flaws in the original design
      ii. If we have more useful data to output

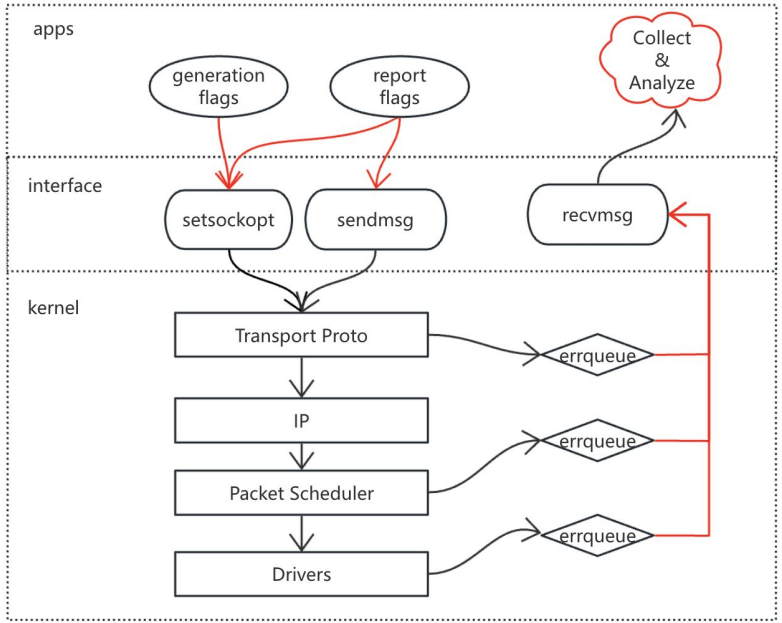


Figure 1: flaws of socket timestamping

2. BPF Timestamping 1.0

1. History
    a. the mirror of socket timestamping that was re-implemented with BPF
        i. support selectively sampling
        ii. only support TCP protocol
        iii. only support TX (RX is on the way)
    b. It works as a third-party centralized BPF service
    c. Report path was replaced with BPF skops
    d. More information about sk/skb other than SOF_TIMESTAMPING_OPT_STATS (in socket timestamping) can be retrieved.

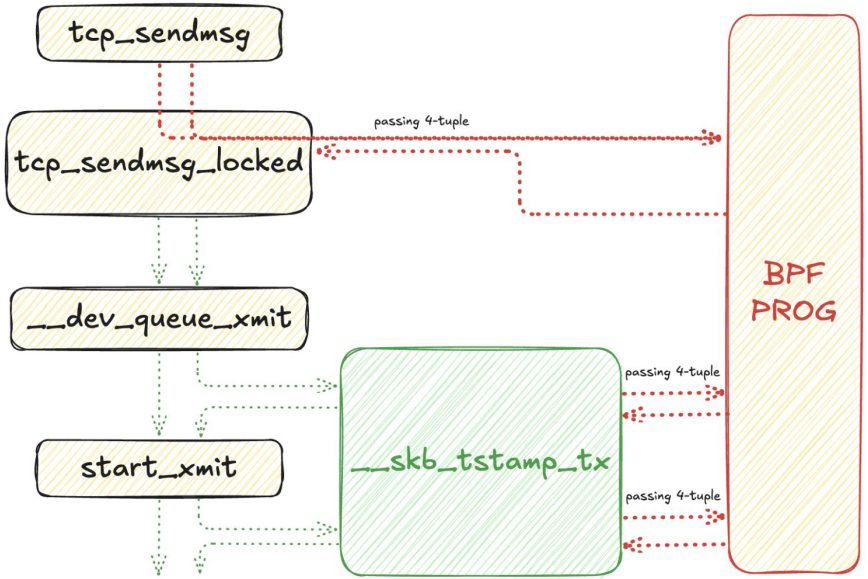


Figure 1: components of BPF timestamping 1.0

2. BPF Timestamping 1.0

1. History
2. How to use?
   a. **fentry tcp_sendmsg_locked**: get the accurate timestamp of triggering send syscall.
   b. **BPF_SOCK_OPS_TSTAMP_SENDMSG_CB**: BPF program calls a kfunc named bpf_sock_ops_enable_tx_tstamp() to decide if we need to trace the transmission. If so, then the kfunc will tags the last skb. And BPF prog finishes the correlation of skb and its sendmsg.
   c. **BPF_SOCK_OPS_TSTAMP_{SCHED/SND/ACK}_CB**: they are the timestamping report points which allows BPF program to report the current timestamp and calculate the latency compared to the start time of send syscall.

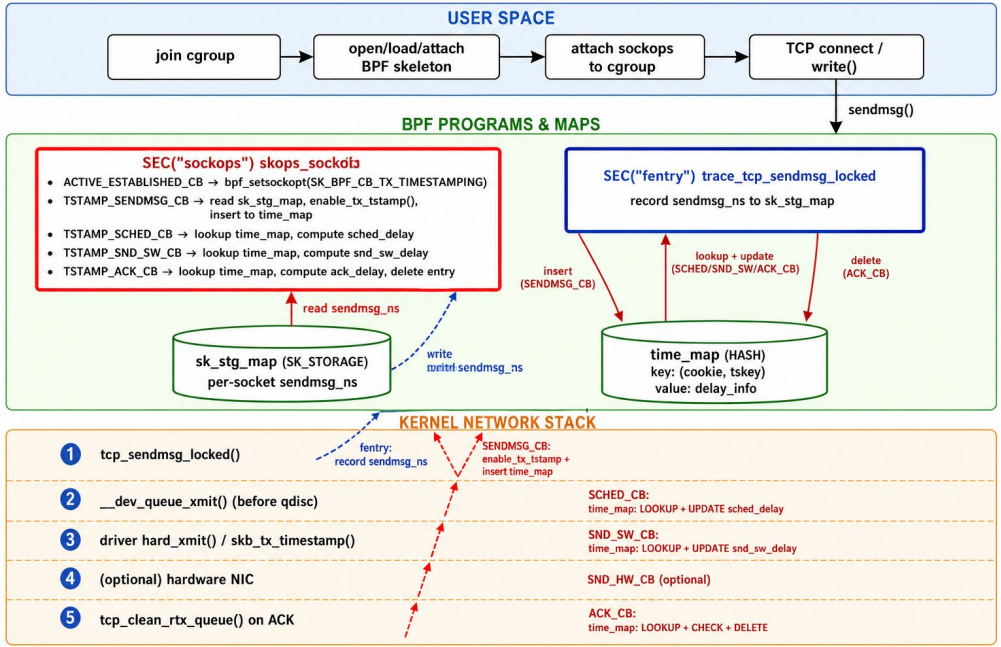


Figure 1: the arch of BPF timestamping 1.0

2. BPF Timestamping 1.0

1. History
2. How to use?
3. Limitations (inheritted from socket timestamping)
    a. Miss SYN/FIN/RST/probe cases
        i. They are not traced, but they all belong to network observation.
        ii. Implementation
            1. hook the start point
                a. tcp_v4_connect()/BPF_SOCK_OPS_TCP_CONNECT_CB
                b. tcp_v4_send_reset()/tcp_send_active_reset()
                c. …
            2. Correlate the start time with the skb in a BPF map, like BPF_SOCK_OPS_TSTAMP_SENDMSG_CB
                a. tcp_init_nondata_skb() fits for all
                b. Or add corresponding callbacks respectively
            3. Enable timestamping feature on each skb, like kfunc bpf_sock_ops_enable_tx_tstamp()

```
static void tcp_tx_timestamp(struct sock *sk, struct sockcm_cookie *sockc)
{
        struct sk_buff *skb = tcp_write_queue_tail(sk);
        u32 tsflags = sockc->tsflags;

        if (unlikely(!skb))
                skb = skb_rb_last(&sk->tcp_rtx_queue);

        if (tsflags && skb) {
                struct skb_shared_info *shinfo = skb_shinfo(skb);
                struct tcp_skb_cb *tcb = TCP_SKB_CB(skb);

                sock_tx_timestamp(sk, sockc, &shinfo->tx_flags);
                if (tsflags & SOF_TIMESTAMPING_TX_ACK)
                        tcb->txstamp_ack |= TSTAMP_ACK_SK;
                if (tsflags & SOF_TIMESTAMPING_TX_RECORD_MASK)
                        shinfo->tskey = TCP_SKB_CB(skb)->seq + skb->len - 1;
        }

        if (cgroup_bpf_enabled(CGROUP_SOCK_OPS) &&
            SK_BPF_CB_FLAG_TEST(sk, SK_BPF_CB_TX_TIMESTAMPING) && skb)
                bpf_skops_tx_timestamping(sk, skb, BPF_SOCK_OPS_TSTAMP_SENDMSG_CB);
}
```

Figure 1: correlation process in the kernel

```
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;
}
```

Figure 2: correlation process in the BPF prog

2. BPF Timestamping 1.0

1. History
2. How to use?
3. Limitations (inheritted from socket timestamping)
   a. miss 3-way handshake cases
   b. miss tagging the tail skb
      i. without tagging, we cannot keep track of this send syscall.
      ii. it can happen due to shortage of memory in [wait_for_space scenario]
      iii. it's not feasible to successfully identify the last skb before push functions get called due to the original design
      iv. Please see patch 3&&4 in the series: [bpf-timestamp: convert to push-level granularity]

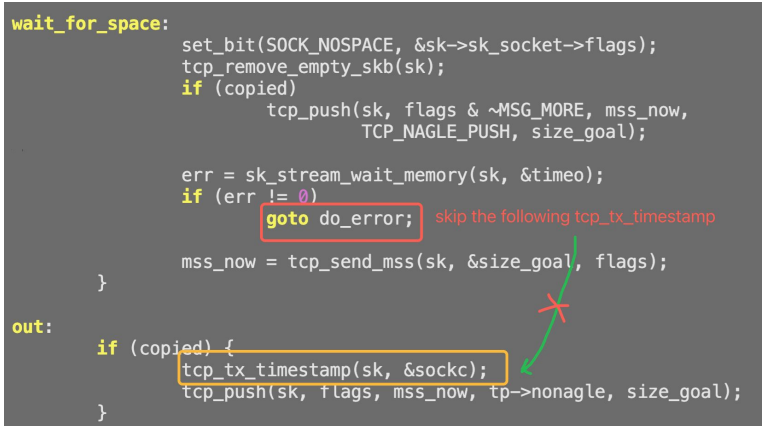


Figure 1: how to reproduce this in tcp_sendmsg_locked()

2. BPF Timestamping 1.0

1. History
2. How to use?
3. Limitations (inheritted from socket timestamping)
    a. miss 3-way handshake cases
    b. miss tagging the tail skb
    c. delayed ack issue (1)
        i. For RPC, it only happens on the tail packet because no more new packet will be pushed out by the kernel.
        ii. High delta between ‘send’ and ‘ack’ hides the fluctuation of network rtt.
        iii. An ack packet can be delayed within 40 ms.
        iv. Kevin Yang said he will add this and define ‘ack_delay’ and ‘network_rtt’ in the process of upstreaming SWIFT.

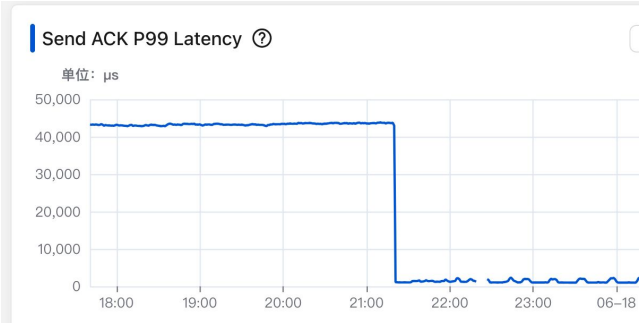


Figure 1: delayed ack makes the latency measurement spuriously high

2. BPF Timestamping 1.0

1. History
2. How to use?
3. Limitations (inheritted from socket timestamping)
   a. miss 3-way handshake cases
   b. miss tagging the tail skb
   c. delayed ack issue (2)
      i. TCP ETS: Extensible Timestamp Options
         1. negotiate during 3-way handshake
         2. we define NetworkRTT as the time from when the data segment, which has TSval=X, is sent until when it arrives at the receiver, plus the time from when the corresponding ACK is sent by the (data) receiver until the ACK arrives at the data sender.
         3. NetworkRTT = ACK.ArrivalTime - ACK.TSecr - ACK.AckDelay
            a. 11 - 1 - (10 - 2) = 2
      ii. With that formula, ‘send - ack’ - ‘send - driver’ is equal to the real network rtt (that doesn’t include host delay anymore)

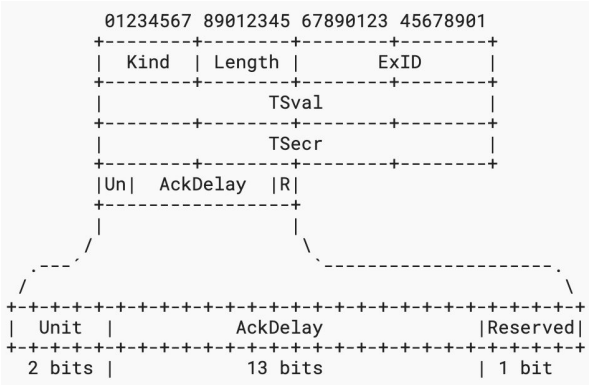


Figure 1: ETS option header format

Figure 2: how NetworkRTT is computed

2. BPF Timestamping 1.0

1. History
2. How to use?
3. Limitations (inheritted from socket timestamping)
   a. miss 3-way handshake cases
   b. miss tagging the tail skb
   c. delay ack issue
   d. big packet issue
      i. Big means the packet will be splitted into multiple smaller-sized packets due to gso/tso, etc.
      ii. If the packet is small, this packet is the tail packet of this send syscall. That's fine.
      iii. If the packet is big, what's the meaning of reporting SCHED/SOFTWARE? Only the latency between 'send' and 'ack' makes sense.
      iv. Big packet may include a few rounds of network RTT latency + host latency which means we cannot perform the correct attribution

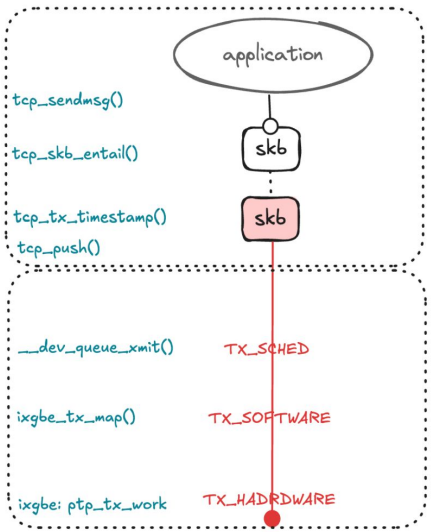


Figure 1: small packet case

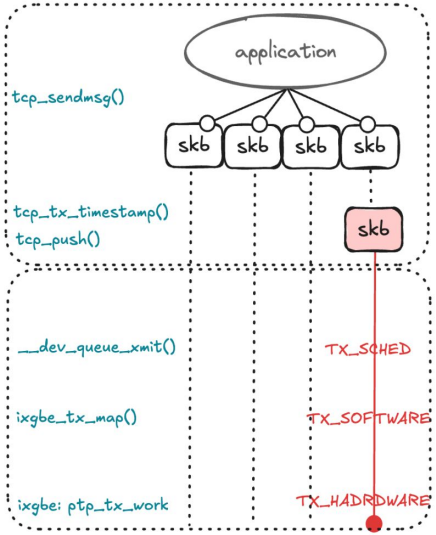


Figure 2: big packet case

2. BPF Timestamping 1.0

1. History
2. How to use?
3. Limitations (inheritted from socket timestamping)
    a. miss 3-way handshake cases
    b. miss tagging the tail skb
    c. delay ack issue
    d. big packet issue
    e. It cannot be applied to the historical flows
        i. BPF prog sets the option during 3-way handshake
        ii. Can we set to the historical flows? Yes, scan the hashinfo of TCP, filter the matched flows, and tag them!

```
SEC("sockops")
int skops_sockopt(struct bpf_sock_ops *skops)
{
        struct bpf_sock *bpf_sk = skops->sk;
        const struct sock *sk;

        if (!bpf_sk)
                return 1;

        sk = (struct sock *)bpf_skc_to_tcp_sock(bpf_sk);
        if (!sk)
                return 1;

        switch (skops->op) {
        case BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB:
                nr_active += !bpf_test_sockopt(skops, sk, 0);
                break;
        case BPF_SOCK_OPS_TSTAMP_SENDMSG_CB:
                if (bpf_test_delay(skops, sk))
                        nr_snd += 1;
                break;
        case BPF_SOCK_OPS_TSTAMP_SCHED_CB:
                if (bpf_test_delay(skops, sk))
                        nr_sched += 1;
                break;
```

Figure 1: sockops - 3-way handshake

2. BPF Timestamping 1.0

1. History
2. How to use?
3. Limitations (inheritted from socket timestamping)
    a. miss 3-way handshake cases
    b. miss tagging the tail skb
    c. delay ack issue
    d. big packet issue
    e. It cannot be applied to the historical flows
    f. TraceID
        i. This is a very open question!
        ii. AFAIK, some developers adopting BPF timestamping as the base of monitoring platform implements the TraceID to correlate the behavior of kernel and RPC to provide a full vision of one transmission.

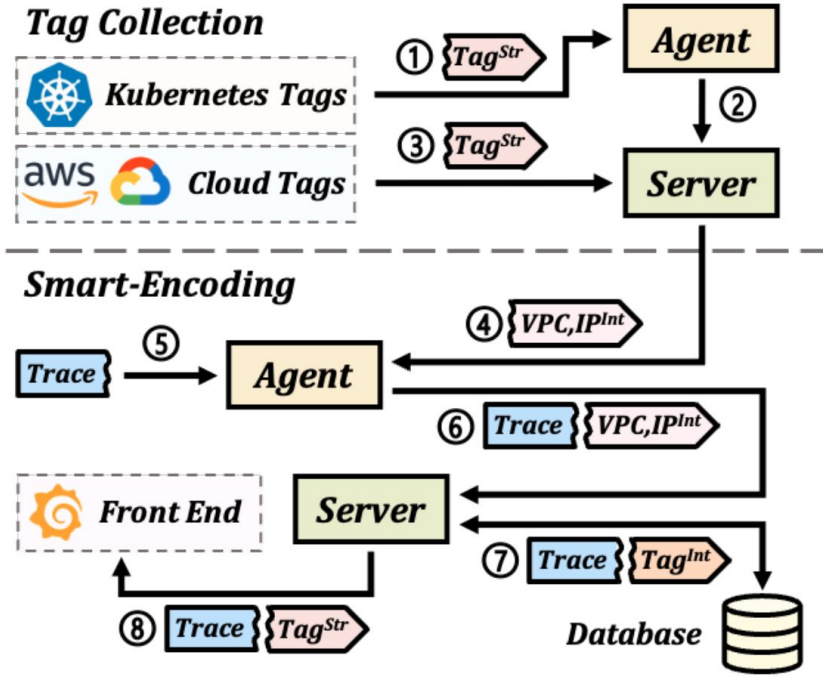


Figure 1: tag-based correlation in deepflow

3. BPF Timestamping 2.0

1. Ultimate Goal
    a. address 1.0 issues
        i. miss 3-way handshake cases √
        ii. miss tagging the tail skb √
        iii. delay ack issue √
        iv. big packet issue √
        v. It cannot be applied to the historical flows √
        vi. TraceID ？

3. BPF Timestamping 2.0

1. Ultimate Goal
    a. address 1.0 issues
    b. support per packet basis (1)
        i. slides starting from page 29 in [The Future of SO_TIMESTAMPING]

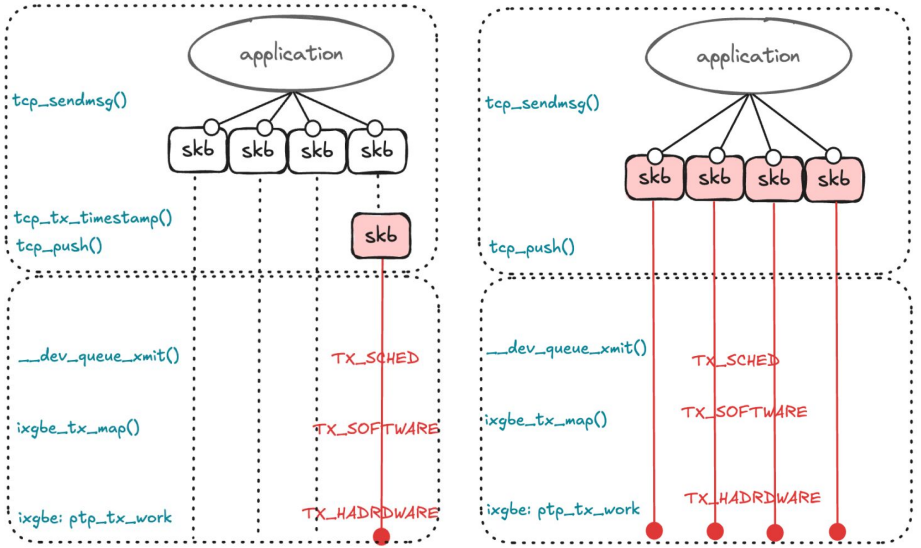


Figure 1: per packet basis in TCP

3. BPF Timestamping 2.0

1. Ultimate Goal
    a. address 1.0 issues
    b. support per packet basis (2)
        i. entry
            1. tcp_skb_entail() makes sure every newly generated skb is tagged

```
tcp_skb_entail()
  -> __sock_tx_timestamp(tsflags, tx_flags);
     // for example
     -> flags |= SKBTX_SCHED_TSTAMP;
     // or
     -> bpf_skops_tx_timestamping()
```

Figure 2: pseudo code

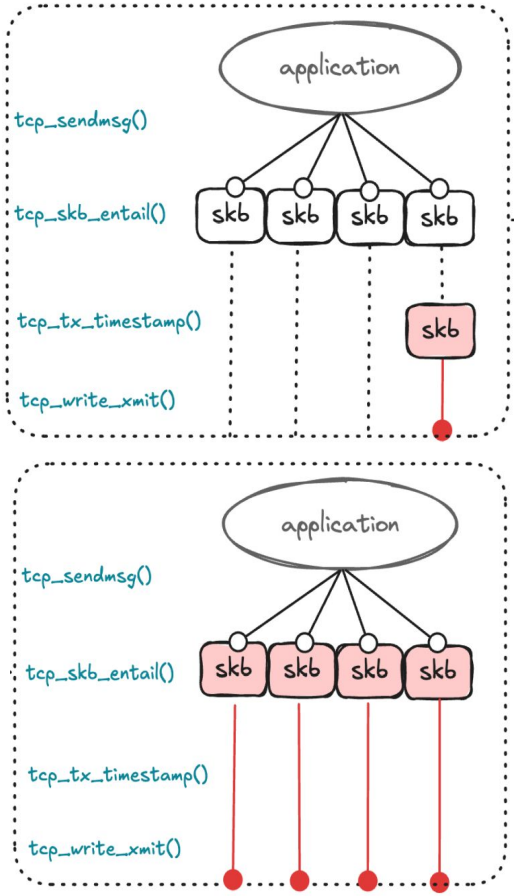


Figure 1: per packet basis in TCP

3. BPF Timestamping 2.0

1. Ultimate Goal
    a. address 1.0 issues
    b. support per packet basis (3)
        i. entry
            1. tcp_skb_entail() makes sure every newly generated skb is tagged
        ii. TCP layer
            1. tso_fragment() splits the big packet for various reasons
            2. tcp_skb_collapse_tstamp()
            3. …

```
tso_fragment()
  -> tcp_fragment_tstamp()
    // tag both skbs
    -> shinfo->tx_flags |= tsflags;
    -> shinfo2->tx_flags |= tsflags;
```

Figure 2: pseudo code

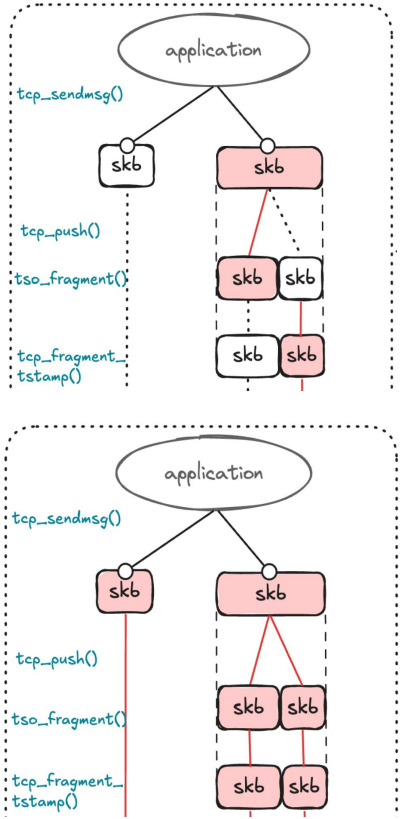


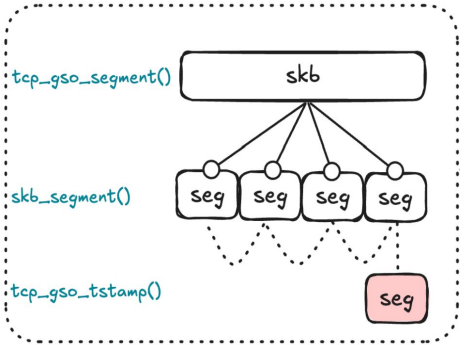
Figure 1: per packet basis in TCP

3. BPF Timestamping 2.0

1. Ultimate Goal
    a. address 1.0 issues
    b. support per packet basis (4)
        i. entry
            1. tcp_skb_entail() makes sure every newly generated skb is tagged
        ii. TCP layer
            1. tso_fragment() splits the big packet for various reasons
        iii. GSO layer (before Qdisc)
            1. tcp_gso_tstamp() does the same thing

```
tcp_gso_tstamp()
  -> while (skb) {
        skb_shinfo(skb)->tx_flags |=
                          SKBTX_SW_TSTAMP;
```

Figure 2: pseudo code



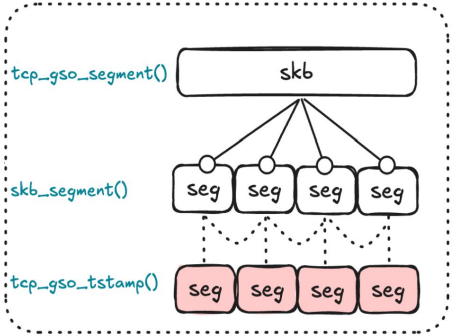


Figure 1: per packet basis in TCP

3. BPF Timestamping 2.0

1. Ultimate Goal
    a. address 1.0 issues
    b. support per packet basis
    c. support the container scenario (1)
        i. It's crucial since the timestamping feature deployed in the host should have the ability to monitor both container and host.
        ii. Container networking scenario is way too complex since we have more than 10 combinations with the use of vlan, vxlan, macvlan, veth, bridge...etc.

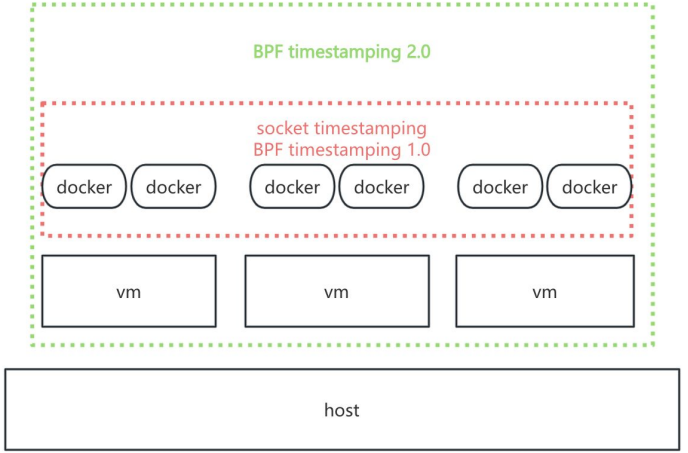


Figure 1: what are we missing in container?

3. BPF Timestamping 2.0

1. Ultimate Goal
    a. address 1.0 issues
    b. support per packet basis
    c. support the container scenario (2)
        i. The path is so different either in tx path or in rx path.
        ii. What is the problem here?
            1. veth
            2. bridge
            3. vxlan
            4. …

```
[POD netns]
  sendmsg()
    → tcp_sendmsg() → tcp_write_xmit() → tcp_transmit_skb()
      → ip_queue_xmit() → ip_local_out()             // NF_LOCAL_OUT
        → ip_output()                                 // NF_POST_ROUTING
          → dev_queue_xmit()                          // qdisc: noqueue
            → veth_xmit()

  ---- veth pair / skb_scrub_packet(): skb->sk=NULL, tstamp=0 ----

[NODE netns]
  netif_receive_skb()
    → ip_rcv()                                        // NF_PRE_ROUTING
      → ip_forward()                                  // NF_FORWARD
        → ip_output()                                 // NF_POST_ROUTING (SNAT)
          → dev_queue_xmit()                          // qdisc: fq_codel
            → NIC_driver_xmit()                       // → wire
```

Figure 1: tx path in veth case

```
[NODE netns]
  NIC_driver_poll() → napi_gro_receive()
    → ip_rcv()                                        // NF_PRE_ROUTING (DNAT)
      → ip_forward()                                  // NF_FORWARD
        → ip_output()                                 // NF_POST_ROUTING
          → dev_queue_xmit()
            → veth_xmit()

  ---- veth pair / skb_scrub_packet(): tstamp=0 ----

[POD netns]
  netif_receive_skb()
    → ip_rcv()                                        // NF_PRE_ROUTING
      → ip_local_deliver()                            // NF_LOCAL_IN
        → tcp_v4_rcv() → tcp_rcv_established()
          → tcp_ack() → tcp_clean_rtx_queue()
              → tcp_tstamp_tx()                       // ACK timestamp ✓ (uses sk directly)
```

Figure 2: rx path in veth case

3. BPF Timestamping 2.0

1. Ultimate Goal
    a. address 1.0 issues
    b. support per packet basis
    c. support the container scenario (3)
        i. tx problem (1)
            1. ip_rcv_core() clears skb->sk
            2. commit 71f9dacd2e4d (“inet: Call skb_orphan before tproxy activates”) says: “As transparent proxying looks up the socket early and assigns it to the skb for later processing, we must drop any existing socket ownership prior to that in order to distinguish between the case where tproxy is active and where it is not.”

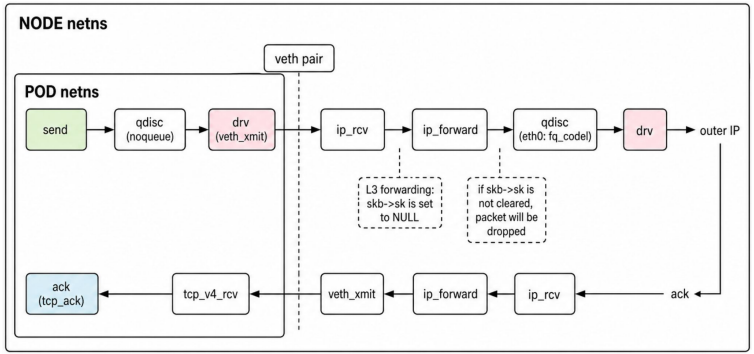


Figure 1: the big view of the netns case

```
static __always_inline void skb_orphan(struct
{
        if (skb->destructor) {
                skb->destructor(skb);
                skb->destructor = NULL;
                skb->sk         = NULL;
        } else {
                BUG_ON(skb->sk);
        }
}
```

Figure 2: skb->sk is cleared in ip_rcv_core()

3. BPF Timestamping 2.0

1. Ultimate Goal
    a. address 1.0 issues
    b. support per packet basis
    c. support the container scenario (4)
        i. tx problem (2)
            1. ip_rcv_core() clears skb->sk
            2. __skb_tstamp_tx() does nothing.
                a. skip it because of no sk
                b. socket timestamping doesn't support it because one of reasons is it needs to put the skb carrying the timestamp into the error queue.
                c. BPF timestamping doesn't support it because sockops needs the sk.

```
void __skb_tstamp_tx(struct sk_buff *orig_skb,
                     const struct sk_buff *ack_skb,
                     struct skb_shared_hwtstamps *hwtstamps,
                     struct sock *sk, int tstype)
{
        struct sk_buff *skb;
        bool tsonly, opt_stats = false;
        u32 tsflags;

        if (!sk)
                return;
```

Figure 1: timestamping doesn't support it

```
err = sock_queue_err_skb(sk, skb);

if (err)
        kfree_skb(skb);
```

Figure 2: socket timestamping needs it

```
void bpf_skops_tx_timestamping(struct sock *sk, struct sk_buff *skb, int op)
{
        struct bpf_sock_ops_kern sock_ops;

        memset(&sock_ops, 0, offsetof(struct bpf_sock_ops_kern, temp));
        sock_ops.op = op;
        sock_ops.is_fullsock = 1;
        sock_ops.sk = sk;
        bpf_skops_init_skb(&sock_ops, skb, 0);
        __cgroup_bpf_run_filter_sock_ops(sk, &sock_ops, CGROUP_SOCK_OPS);
}
#endif
```

Figure 3: socket timestamping needs it

3. BPF Timestamping 2.0

1. Ultimate Goal
    a. address 1.0 issues
    b. support per packet basis
    c. support the container scenario (5)
        i. tx problem (3)
            1. ip_rcv_core() clears skb->sk
            2. __skb_tstamp_tx() does nothing.
            3. how to workaround it for BPF timestamping?
                a. SEC("fentry/__skb_tstamp_tx”)
                b. when !sk condition is met, print out 4-tuple information from orig_skb with the current timestamp.
                c. Calculate the delta.
            4. My take here is we rely too much on the socket. Can we avoid that?

3. BPF Timestamping 2.0

1. Ultimate Goal
    a. address 1.0 issues
    b. support per packet basis
    c. support the container scenario (6)
        i. rx problem (1)
            1. skb_scrub_packet() clears skb->tstamp.
            2. commit c47d8c2f38f8 (“net: Clear skb->tstamp only on the forwarding path”) says: “Now that skb->tstamp will be used to hold packet's txtime, we must ensure that it is being cleared when traversing namespaces. Also, doing that from skb_scrub_packet() before the early return would break our feature when tunnels are used.”

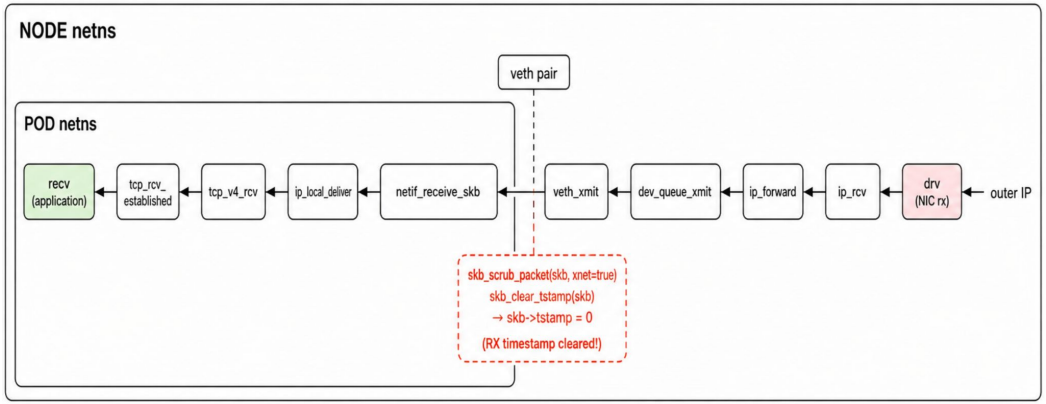


Figure 1: the big view of the netns case

```
        skb->mark = 0;
        skb_clear_tstamp(skb);
}
EXPORT_SYMBOL_GPL(skb_scrub_packet);
```

Figure 2: socket timestamping skip netns case

3. BPF Timestamping 2.0

1. Ultimate Goal
    a. address 1.0 issues
    b. support per packet basis
    c. support the container scenario (7)
        i. rx problem (2)
            1. The ‘tstamp’ of skb is cleared, so in the tcp_recvmsg_locked() the event will not be reported.
            2. It seems no workaround method.
            3. My take here is the tstamp can be used in both sides for absolutely different reasons and also used for timestamping feature! Timetamping feature should be a standalone module which means we can have a standalone field to record the time?

```
#define net_timestamp_check(COND, SKB)
        if (static_branch_unlikely(&netstamp_needed_key)) {
                if ((COND) && !(SKB)->tstamp)
                        (SKB)->tstamp = ktime_get_real();
        }
```

Figure 1: tag the skb at the beginning of rcv path

```
if (TCP_SKB_CB(skb)->has_rxtstamp) {
        tcp_update_recv_tstamps(skb, tss);
        *cmsg_flags |= TCP_CMSG_TS;
}
```

Figure 2: tcp_recvmsg_locked()

3. BPF Timestamping 2.0

1. Ultimate Goal
    a. address 1.0 issues
    b. support per packet basis
    c. support the container scenario
    d. promising directions
        i. **simple interfaces** to plug in
        ii. achieve the **extremely low overhead**
        iii. **always-on** in production
        iv. **transparently** observe
        v. **easy-to-use**
        vi. **flexible**

Extend 2.0 as the latency framework which is worth it but definitely challenging!!!

3. BPF Timestamping 2.0

1. Design
    a. Where to put the global-wide timestamp?
        i. new member in shared info?
            1. Can we afford the global impact of adding u32/u64 space?
            2. Pros:
                a. Independence
                    i. no longer rely on BPF timestamping report points
                b. Flexibility
                    i. we can calculate the latency as long as we can write a bpf-based tool that is able to hook the function and get the skb parameter.
            3. Cons:
                a. each skb across protocols suffers from the overhead of an extra u64 storage slot.
            4. side note: also need to cover the netfilter case since the 4-tuple of skb is changed in this case.

3. BPF Timestamping 2.0

1. Design
    a. Where to put the global-wide timestamp?
        i. skb->tstamp
            1. tstamp can be used in dual direction.
            2. TX
                a. When the skb carrying this value is forwarded, that probably means the skb will be sent out in the tx path, so this value can interfere with the pacing logic.
            3. RX
                a. socket timesstamping

```
static int fq_enqueue(struct sk_buff *skb, struct Qdisc *sch,
                      struct sk_buff **to_free)
{
        struct fq_sched_data *q = qdisc_priv(sch);
        struct fq_flow *f;
        u64 now;
        u8 band;

        band = fq_prio2band(q->prio2band, skb->priority & TC_PRIO_M
        if (unlikely(q->band_pkt_count[band] >= sch->limit)) {
                q->stat_band_drops[band]++;
                return qdisc_drop_reason(skb, sch, to_free, QDISC_D
        }

        now = ktime_get_ns();
        if (!skb->tstamp) {
                fq_skb_cb(skb)->time_to_send = now;
        } else {
                ktime_t time_to_send = fq_skb_tstamp_to_mono(skb);

                /* Check if packet timestamp is too far in the futu
                if (fq_packet_beyond_horizon(time_to_send, q, now))
```

Figure 1: the use of tstamp in tx path

3. BPF Timestamping 2.0

1. Design
    a. Where to put the global-wide timestamp?
        i. sk_cookie
            1. it is bound to socket itself rather than packet
            2. Can we have something like an unique cookie for each skb?
                a. No, because the allocation of cookie is not cheap since it needs to deal with the huge contention if the pps of flow is high. (please see figure 2)
            3. We need to find a field belonging to the skb and friends.

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

Figure 1: inspiration from BPF prog

```
static __always_inline u64 gen_cookie_next(struct gen_cookie *gc)
{
        struct pcpu_gen_cookie *local = this_cpu_ptr(gc->local);
        u64 val;

        if (likely(local_inc_return(&local->nesting) == 1)) {
                val = local->last;
                if (__is_defined(CONFIG_SMP) &&
                    unlikely((val & (COOKIE_LOCAL_BATCH - 1)) == 0)) {
                        s64 next = atomic64_add_return(COOKIE_LOCAL_BATCH,
                                                       &gc->forward_last);
                        val = next - COOKIE_LOCAL_BATCH;
                }
                local->last = ++val;
        } else {
                val = atomic64_dec_return(&gc->reverse_last);
        }
        local_dec(&local->nesting);
        return val;
}
```

Figure 2: generation of cookie

3. BPF Timestamping 2.0

1. Design
   a. Where to put the global-wide timestamp?
      i. skb->cb
         1. TCP_SKB_CB is used for original skb rather than the cloned one.
         2. TCP_SKB_CB is no longer valid until the clone skb traverses IP layer
         3. different layers has the privilege to use this private member

```
#define TCP_SKB_CB(__skb)       ((struct tcp_skb_cb *)&((__skb)->cb[0]))
```

```
static void tcp_tx_timestamp(struct sock *sk, struct sockcm_cookie *sockc)
{
        struct sk_buff *skb = tcp_write_queue_tail(sk);
        u32 tsflags = sockc->tsflags;

        if (unlikely(!skb))
                skb = skb_rb_last(&sk->tcp_rtx_queue);

        if (tsflags && skb) {
                struct skb_shared_info *shinfo = skb_shinfo(skb);
                struct tcp_skb_cb *tcb = TCP_SKB_CB(skb);

                sock_tx_timestamp(sk, sockc, &shinfo->tx_flags);
                if (tsflags & SOF_TIMESTAMPING_TX_ACK)
                        tcb->txstamp_ack |= TSTAMP_ACK_SK;
                if (tsflags & SOF_TIMESTAMPING_TX_RECORD_MASK)
                        shinfo->tskey = TCP_SKB_CB(skb)->seq + skb->len - 1;
        }
```

Figure 1: CB in TCP layer

```
#define IPCB(skb) ((struct inet_skb_parm*)((skb)->cb))
#define IP6CB(skb)      ((struct inet6_skb_parm*)((skb)->cb))
```

Figure 2: CB in IP layer

3. BPF Timestamping 2.0

1. Design
    a. Where to put the global-wide timestamp?
        i. enum skb_ext_id
            1. it doesn't support rx side well since skb->slow_gro = 1; will slow down the transmission speed.
            2. skb_scrub_packet() also clears the content
                a. see the comment: “skb_scrub_packet can also be used to clean a skb before injecting it in another namespace (@xnet == true). We have to clear all information in the skb that could impact namespace isolation.”

```
#ifdef CONFIG_SKB_EXTENSIONS
enum skb_ext_id {
#if IS_ENABLED(CONFIG_BRIDGE_NETFILTER)
        SKB_EXT_BRIDGE_NF,
#endif
#ifdef CONFIG_XFRM
        SKB_EXT_SEC_PATH,
#endif
#if IS_ENABLED(CONFIG_NET_TC_SKB_EXT)
        TC_SKB_EXT,
#endif
#if IS_ENABLED(CONFIG_MPTCP)
        SKB_EXT_MPTCP,
#endif
#if IS_ENABLED(CONFIG_MCTP_FLOWS)
        SKB_EXT_MCTP,
#endif
#if IS_ENABLED(CONFIG_INET_PSP)
        SKB_EXT_PSP,
#endif
#if IS_ENABLED(CONFIG_CAN)
        SKB_EXT_CAN,
#endif
        SKB_EXT_NUM, /* must be last */
};
```

Figure 1: the structure of existing type

3. BPF Timestamping 2.0

1. Design
    a. Where to put the global-wide timestamp?
        i. tskey
            1. as the unique ID?
                a. the possibility of conflict is too high
            2. union tskey?
                a. skb_scrub_packet() doesn't touch it
            3. Tskey is an ‘u32’ field but timestamp is u64, so it’s not a good candidate.

```
struct skb_shared_info {
        __u8            flags;
        __u8            meta_len;
        __u8            nr_frags;
        __u8            tx_flags;
        unsigned short  gso_size;
        /* Warning: this field is not always filled in
        unsigned short  gso_segs;
        struct sk_buff  *frag_list;
        union {
                struct skb_shared_hwtstamps hwtstamps;
                struct xsk_tx_metadata_compl xsk_meta;
        };
        unsigned int    gso_type;
        u32             tskey;
```

Figure 1: skb_shared_info structure

3. BPF Timestamping 2.0

1. Design
    a. Where to put the global-wide timestamp?
        i. skb metadata?
            1. Martin suggested me to apply skb metadata, but I reckon it's definitely a no-go. Performance impact of network observation is extremely important!!!
            2. "when you need to attach metadata only to the first packet of a TCP/QUIC connection or sample packets **at very low rates** for tracing."
            3. References:
                a. [PATCH RFC bpf-next 0/5] skb extension for BPF local storage
                b. 2026: Thrice the charm: an skb extension for BPF metadata
                c. 2025: Packet Metadata - Where Are Thee?
                d. 2025: Traits: Rich Packet Metadata
                e. 2024: Marking Packets With Rich Metadata

```
Our initial rough benchmarks on a VM with kernel.bpf_stats_enabled=1 [3]
show that running a tc/ingress prog that creates skb local storage and
writes to it amounts to ~330 nsec of per-packet overhead.  Retrieving skb
local storage and reading from it in a cgroup_skb/ingress hook contributes
an additional ~115 nsec.

Rounding up to ~500 nsec per packet:
- at 100k pps, that's 5% of the 10 usec per-packet budget, but
- at 1 Mpps, that's already 50% of the budget, which is not acceptable.
```

Figure 1: current status of skb metadata

3. BPF Timestamping 2.0

1. Design
    a. Where to put the global-wide timestamp?
        i. hwtstamp (1)
            1. rename and repurpose the use of hwtstamp
            2. the orig and cloned skb share the same hwtstamp
            3. tx
                a. find/create a ktime filed to store the timestamp
                b. handle gso/tso/collapse… logic
            4. rx
                a. add the same initialization logic in net_timestamp_check()
                b. handle the functions that use ‘has_rxtstamp’
                c. …

```
void tcp_skb_collapse_tstamp(struct sk_buff *skb,
                             const struct sk_buff *next_skb)
{
        if (unlikely(tcp_has_tx_tstamp(next_skb))) {
                const struct skb_shared_info *next_shinfo =
                        skb_shinfo(next_skb);
                struct skb_shared_info *shinfo = skb_shinfo(skb);

                shinfo->tx_flags |= next_shinfo->tx_flags & SKBTX_ANY_TSTAMP;
                shinfo->tskey = next_shinfo->tskey;
                TCP_SKB_CB(skb)->txstamp_ack |=
                        TCP_SKB_CB(next_skb)->txstamp_ack;
        }
}
```

Figure 1: collapse example

```
#define net_timestamp_check(COND, SKB)
        if (static_branch_unlikely(&netstamp_needed_key)) {
                if ((COND) && !(SKB)->tstamp)
                        (SKB)->tstamp = ktime_get_real();
        }
```

Figure 2: skb->tstamp initialization

3. BPF Timestamping 2.0

1. Design
    a. Where to put the global-wide timestamp?
        i. hwtstamp (2)
            1. advantage:
                a. Independence
                    i. no longer rely on BPF timestamping report points
                b. Flexibility
                    i. we can calculate the latency as long as we can write a bpf-based tool that is able to hook the function and get the skb parameter.
            2. side note: also need to cover the netfilter case since the 4-tuple of skb is changed in this case.

3. BPF Timestamping 2.0

1. Implementation for any protocol (3-step)
    a. **Record/reset start time**
        i. Find where to generate the start time
        ii. Propagate it to the target skb
        iii. Call a kfunc to reset it
    b. **Handle splitted skbs**
        i. If new skbs are generated, make sure that the skb can be passed to the corresponding start time.
    c. **Report the event**
        i. It can happen at any hook point
        ii. It can happen at the pre-embeded points like the callbacks of BPF timestamping 1.0

3. BPF Timestamping 2.0

1. Blind Spots
    a. irq/softirq latency
        i. the start time in rx path is net_timestamp_check() which is pretty close to the driver side, but what if the napi poll is triggered too late?
        ii. example: commit d15121be74856(“Revert "softirq: Let ksoftirqd do its job"”)
    b. sched latency
        i. cpu utilization reaches 100%
        ii. in-house scheduler makes the mistake
        iii. too many heavy tasks in one running queue of CPU
    c. anything else?

3. BPF Timestamping 2.0

1. Agent direction
    a. socket timestamping (1)
        i. report timestamps along with some fields in tcp_info, and application needs to take care of the rest logic: match, calculate, upload.
            1. tskey is used to match timestamp in each report point with the start time of its real send syscall.
            2. ‘printpacket()’ is the basic example on how to parse the timestamp after invoking an additional recvmsg() syscall, which impacts the performance
            3. Please refer to tools/testing/selftests/net/timestamping.c

```
static void recvpacket(int sock, int recvmsg_flags,
                       int siocgstamp, int siocgstampns, int ptpv
{
        char data[256];
        struct msghdr msg;
        struct iovec entry;
        struct sockaddr_in from_addr;
        struct {
                struct cmsghdr cm;
                char control[512];
        } control;
        int res;

        memset(&msg, 0, sizeof(msg));
        msg.msg_iov = &entry;
        msg.msg_iovlen = 1;
        entry.iov_base = data;
        entry.iov_len = sizeof(data);
        msg.msg_name = (caddr_t)&from_addr;
        msg.msg_namelen = sizeof(from_addr);
        msg.msg_control = &control;
        msg.msg_controllen = sizeof(control);

        res = recvmsg(sock, &msg, recvmsg_flags|MSG_DONTWAIT);
        if (res < 0) {
                printf("%s %s: %s\n",
                       "recvmsg",
                       (recvmsg_flags & MSG_ERRQUEUE) ? "error" :
                       strerror(errno));
        } else {
                printpacket(&msg, res, data,
                            sock, recvmsg_flags,
                            siocgstamp, siocgstampns, ptpv2);
        }
}
```

Figure 1: selftests example in tools/testing/selftests/net/timestamping.c

3. BPF Timestamping 2.0

1. Agent direction
    a. socket timestamping (2)
        i. Sync mode
            1. application: get time A -> send data -> recvmsg from error queue and get time B -> calculate the delta (B-A) -> upload to the remote centralized servers
            2. Pros:
                a. simple/fixed pattern in application: get the timestamps and compute the latency
            3. Cons:
                a. terrible performance
            4. Apparently, it's a not production solution

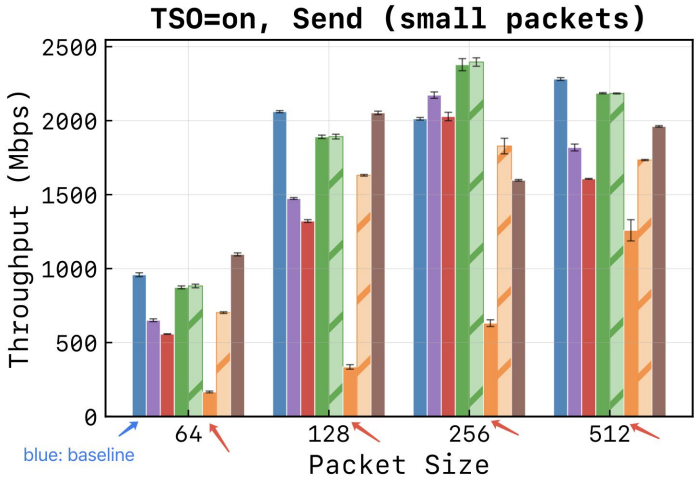


Figure 1: socket timstamping (sync) + netperf

3. BPF Timestamping 2.0

1. Agent direction
    a. socket timestamping (3)
        i. Async mode
            1. application: get time A -> send data -> continue with the rest && (start a async thread to poll -> recvmsg from error queue and get time B -> **match** and find A -> calculate the delta (B-A) -> upload to the remote centralized servers
            2. Pros:
                a. better performance
                b. computing is simple:
                    i. the timeline is ordered chronologically
                    ii. matching process might consume a little cpu%/mem resources
            3. Cons:
                a. the number of events is limited by sk->sk_rcvbuf (in sock_queue_err_skb())
            4. Yes, it's a production solution!

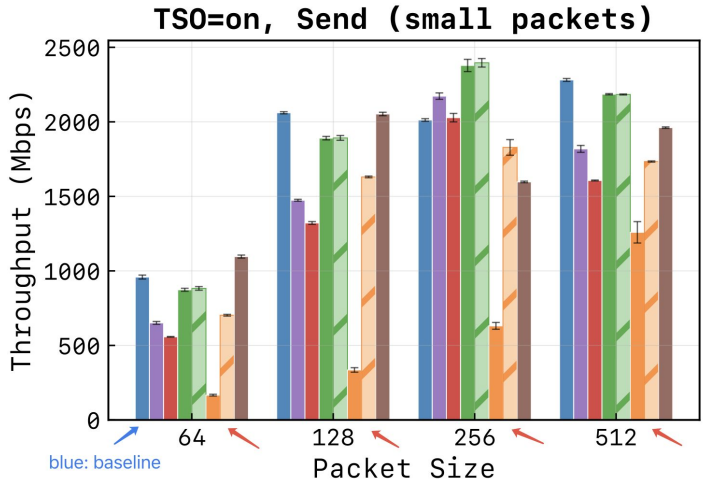


Figure 1: socket timstamping (async) + netperf

3. BPF Timestamping 2.0

1. Agent direction
   a. socket timestamping
   b. bpf timestamping 1.0 (1)
      i. sockops points are triggered, and BPF prog needs to take care of the whole logic: generate the current timestamp, match, calculate, upload.
         1. Application doesn't get involved anymore.
         2. Tskey is used to match timestamp in each report point with the start time of its real send syscall.
         3. Please refer to 1) tools/testing/selftests/bpf/progs/net_timestamping.c , 2) tools/testing/selftests/bpf/prog_tests/net_timestamping.c

```
SEC("sockops")
int skops_sockopt(struct bpf_sock_ops *skops)
{
        struct bpf_sock *bpf_sk = skops->sk;
        const struct sock *sk;

        if (!bpf_sk)
                return 1;

        sk = (struct sock *)bpf_skc_to_tcp_sock(bpf_sk);
        if (!sk)
                return 1;

        switch (skops->op) {
        case BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB:
                nr_active += !bpf_test_sockopt(skops, sk, 0);
                break;
        case BPF_SOCK_OPS_TSTAMP_SENDMSG_CB:
                if (bpf_test_delay(skops, sk))
                        nr_snd += 1;
                break;
        case BPF_SOCK_OPS_TSTAMP_SCHED_CB:
                if (bpf_test_delay(skops, sk))
                        nr_sched += 1;
                break;
        case BPF_SOCK_OPS_TSTAMP_SND_SW_CB:
                if (bpf_test_delay(skops, sk))
                        nr_txsw += 1;
                break;
        case BPF_SOCK_OPS_TSTAMP_ACK_CB:
                if (bpf_test_delay(skops, sk))
                        nr_ack += 1;
                break;
        }

        return 1;
}
```

Figure 1: sockops in BPF prog to do the computation

3. BPF Timestamping 2.0

1. Agent direction
   a. socket timestamping
   b. bpf timestamping 1.0 (2)
      i. Sync mode (1)
         1. tcp_sendmsg_locked(): store the start time in ‘sk_stg_map’
         2. SENDMSG_CB: look up the sk in ‘sk_stg_map’, correlate the start time with the current skb, store the info of skb in ‘time_map’
         3. Report points/other CB: look up the tskey in ‘time_map’, correlate the current time with its start time, compute the latency.
         4. AFAIK, this method has been deployed at a small production scale.

```
stg = bpf_sk_storage_get(&sk_stg_map, sk, 0,
                         BPF_SK_STORAGE_GET_F_CREATE);
if (!stg)
        return 0;

stg->sendmsg_ns = timestamp;
```

Figure 1: fentry - tcp_sendmsg_locked()

```
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;
}
```

Figure 2: sockops - SENDMSG_CB

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

switch (skops->op) {
case BPF_SOCK_OPS_TSTAMP_SCHED_CB:
        val->sched_delay = timestamp - val->sendmsg_ns;
        delay = val->sched_delay;
        break;
```

Figure 3: sockops - other points

3. BPF Timestamping 2.0

1. Agent direction
    a. socket timestamping
    b. bpf timestamping 1.0 (3)
        i. Sync mode (2)
            1. Pros
                a. computing is simple
                    i. there is no need to do matching.
            2. Cons
                a. a certain performance impact
                    i. two times of lookup process in the hot paths per sendmsg.
                    ii. storage map - sk_storage_map_ops
                    iii. hashtab map - htab_map_ops

```
stg = bpf_sk_storage_get(&sk_stg_map, sk, 0,
                         BPF_SK_STORAGE_GET_F_CREATE);
if (!stg)
        return 0;

stg->sendmsg_ns = timestamp;
```

Figure 1: fentry - tcp_sendmsg_locked()

```
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;
}
```

Figure 2: sockops - SENDMSG_CB

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

switch (skops->op) {
case BPF_SOCK_OPS_TSTAMP_SCHED_CB:
        val->sched_delay = timestamp - val->sendmsg_ns;
        delay = val->sched_delay;
        break;
```

Figure 3: sockops - other points

3. BPF Timestamping 2.0

1. Agent direction
    a. socket timestamping
    b. bpf timestamping 1.0 (4)
        i. Async mode
            1. ring buffer
                a. replace the heavy perf buffer
                    i. memory overhead;
                    ii. data ordering;
                    iii. wasted work and extra data copying.
                b. Andrii Nakryiko's Blog - BPF ring buffer
                c. anakryiko/bpf-ringbuf-examples
                d. [PATCH v4 bpf-next 0/5] BPF ring buffer

```
BPF_CALL_3(bpf_ringbuf_reserve, struct bpf_map *, map, u64, size, u64,
{
        struct bpf_ringbuf_map *rb_map;

        if (unlikely(flags))
                return 0;

        rb_map = container_of(map, struct bpf_ringbuf_map, map);
        return (unsigned long)__bpf_ringbuf_reserve(rb_map->rb, size);
}
```

Figure 1. implementation

```
All the below scenarios are implemented in a script in
benchs/run_bench_ringbufs.sh. Tests were performed on 28-core/56-thread
Intel Xeon CPU E5-2680 v4 @ 2.40GHz CPU.

Single-producer, parallel producer
==================================
rb-libbpf            12.054 ± 0.320M/s (drops 0.000 ± 0.000M/s)
rb-custom            8.158 ± 0.118M/s (drops 0.001 ± 0.003M/s)
pb-libbpf            0.931 ± 0.007M/s (drops 0.000 ± 0.000M/s)
pb-custom            0.965 ± 0.003M/s (drops 0.000 ± 0.000M/s)

Single-producer, parallel producer, sampled notification
========================================================
rb-libbpf            11.563 ± 0.067M/s (drops 0.000 ± 0.000M/s)
rb-custom            15.895 ± 0.076M/s (drops 0.000 ± 0.000M/s)
pb-libbpf            9.889 ± 0.032M/s (drops 0.000 ± 0.000M/s)
pb-custom            9.866 ± 0.028M/s (drops 0.000 ± 0.000M/s)
```

Figure 2: performance comparison

3. BPF Timestamping 2.0

1. Agent direction
    a. socket timestamping
    b. bpf timestamping 1.0 (4)
        i. Async mode - global
            1. At each timestamping point, the only thing for BPF prog to do is 1) get the current timestamp, 2) report it to an agent in a global shared buffer
            2. Pros:
                a. performance impact
                    i. no more lookup
                    ii. lock contention exists!
            3. Cons:
                a. Agent needs to do the complex matching process

```
struct {
        __uint(type, BPF_MAP_TYPE_RINGBUF);
        __uint(max_entries, 256 * 1024);  /* 256KB */
} rb SEC(".maps");

SEC("fentry/tcp_sendmsg_locked")
int BPF_PROG(example, struct sk_buff *skb)
{
        struct event *e;

        e = bpf_ringbuf_reserve(&rb, sizeof(*e), 0);
        if (!e)
                return 0;

        e->ts = bpf_ktime_get_ns();
        bpf_ringbuf_submit(e, 0);
        return 0;
}
```

Figure 1: single buffer example

3. BPF Timestamping 2.0

1. Agent direction
    a. socket timestamping
    b. bpf timestamping 1.0 (5)
        i. Async mode - per cpu
            1. At each timestamping point, the only thing for BPF prog to do is 1) get the current timestamp, 2) report it to an agent in the current cpu's shared buffer
            2. Pros:
                a. performance impact
                    i. no more lookup
                    ii. no more lock contention
            3. Cons:
                a. Agent needs to do the **extremely** complex matching process.
                b. Our experience is **terrible** because we have to consider how TCP works before reaching a stable status. It's because everything events are grouped into one map/file. And the timeline is not ordered chronologically.

```
/* BPF */
struct {
        __uint(type, BPF_MAP_TYPE_ARRAY_OF_MAPS);
        __uint(max_entries, MAX_CPUS);
        __type(key, __u32);
        __uint(values_fd, 0);
} rb_per_cpu SEC(".maps");

/* Userspace */
int nr_cpus = libbpf_num_possible_cpus();
int outer_fd = bpf_map__fd(skel->maps.rb_per_cpu);

for (int i = 0; i < nr_cpus; i++) {
        LIBBPF_OPTS(bpf_map_create_opts, opts);
        int inner_fd = bpf_map_create(BPF_MAP_TYPE_RINGBUF,
                                      "rb_inner", 0, 0, 256 * 1024, &opts);
        bpf_map_update_elem(outer_fd, &i, &inner_fd, BPF_ANY);
        close(inner_fd);
}
```

Figure 1: per-cpu buffer example

3. BPF Timestamping 2.0

1. Agent direction
    a. socket timestamping
    b. bpf timestamping 1.0
    c. bpf timestamping 2.0
        i. Sync mode (only)
            1. Pros
                a. Good performance
                    i. no more lookup
                    ii. per cpu shared buffer
                b. Easy computation
                    i. kernel reports the final result
                c. Flexiblity
                    i. People can write BPF-based tools hooking other functions (that must have the skb parameter) to use this feature
            2. Cons
                a. (Sorry, I don't know ;p)

```
now = ts_get_ktime();
ts_print_timestamp_delta(sk, TS_SCM_TSTAMP_RECV_TCP,
                         start_time, now - start_time,
                         tskey, NULL, true);
```

Figure 1: only report delta at each timestamping point

3. BPF Timestamping 2.0

1. Agent direction

Feature	User App	Performance	Simplicity	Key Factors
socket timestamping	sync	1	5	extra syscalls
	async	2	3	extra syscalls + match events of each socket
BPF timestamping 1.0	sync	3	5	2 times lookup
	async - per cpu	5	1	match events of all the sockets + out-of-order
	async - global	4	3	match events of all the sockets + in order
BPF timestamping 2.0	sync	5	5	(NULL)
	Async	(2.0 doesn't require unneeded async mode due to the kernel design)		

Table 1: comparison

Bonus: answers to ‘The Future of SO_TIMESTAMPING’

1. The Future of SO_TIMESTAMPING
    a. Q1: Can we thoroughly settle down the latency boundary issue?
        i. A1: Yes, contract is the first thing we need to discuss. After that, we completes the full attribution (kernel-level, layer-level).
    b. Q2: Will we add more hooks like in tcp_write_xmit() to see why the skb is not sent to the Qdisc by using kfunc?
        i. A1: No, unless it’s one part of layer-level attribution.
    c. Q3: Is It Possible to Replace tcpdump?
        i. A1: why not? It’s just one of our goals in the modern world.
    d. Q4: Explore to leverage numerous real data?
        i. A1: Yes, we’re on the way right now.
            1. it shows the continuous performance behavior (Figure 1)
            2. It captures the counterintuitive events (Figure 2)

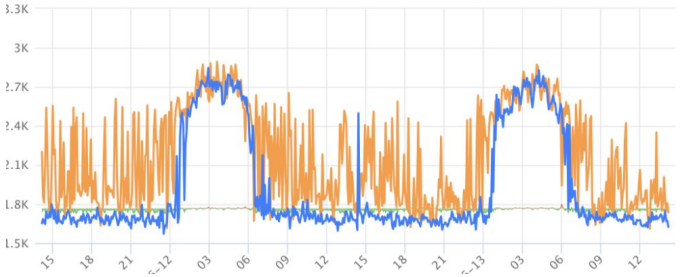


Figure 1: the adjustment of flow control (before/after)

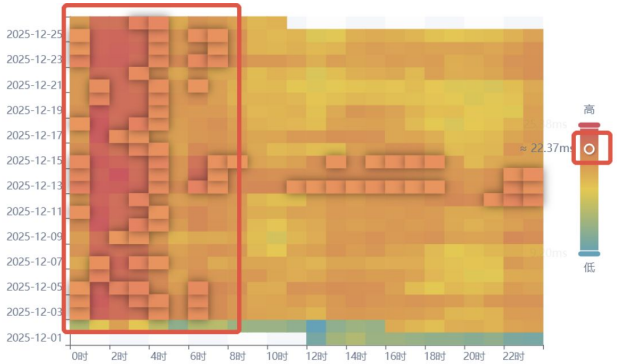


Figure 2: the heap map of more than thousands of machines

Bonus: new paradigm of trouble shooting (before)

Old-fashioned Blackboard

1. **All of them are counters**
   a. basic counters: cpu/mem/io/fs/net
   b. advanced counters: via kernel modules/BPF/customized tracepoints/MIB
2. **All of them are line charts**
   a. spikes indicate issues
   b. trends represent the prediction of potential issues
3. **Newbie VS Expert Knowledge**
   a. Only <10% expert knowledge can be effectively integrated into the auto monitoring system.

The problem is if without experts how far can we use ths blackboard to solve the real-world issues?

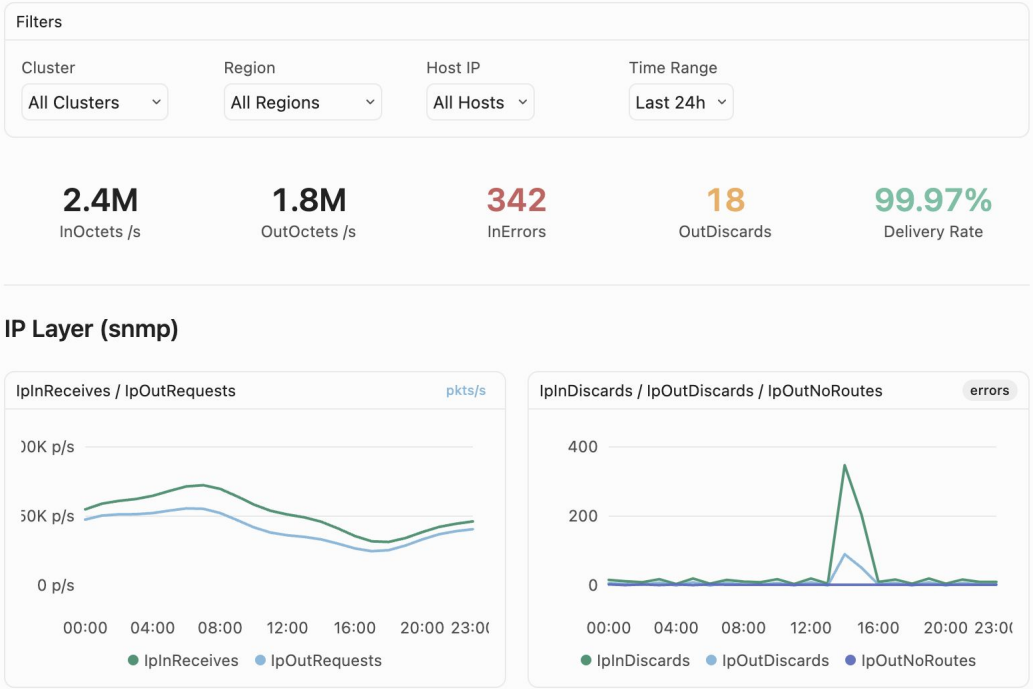


Figure 1: Old-fashioned Blackboard (generated by AI)

Bonus: new paradigm of trouble shooting (after)

Agentic Blackboard

1. **AI + Harness**
   a. Automatically/step by step root-cause the issue
2. **BLUF is the headline**
   a. no more line charts because they only serve for admins/system developers rather than anyone who may be not expertised in the system.
   b. Reasons and instructions are what we care
3. **Snapshot**
   a. If there is a jitter, just do the snapshot of everything (including cpu/mem/io/net…) like what vmcore looks like.
   b. The precondition is using BPF timestamping 2.0

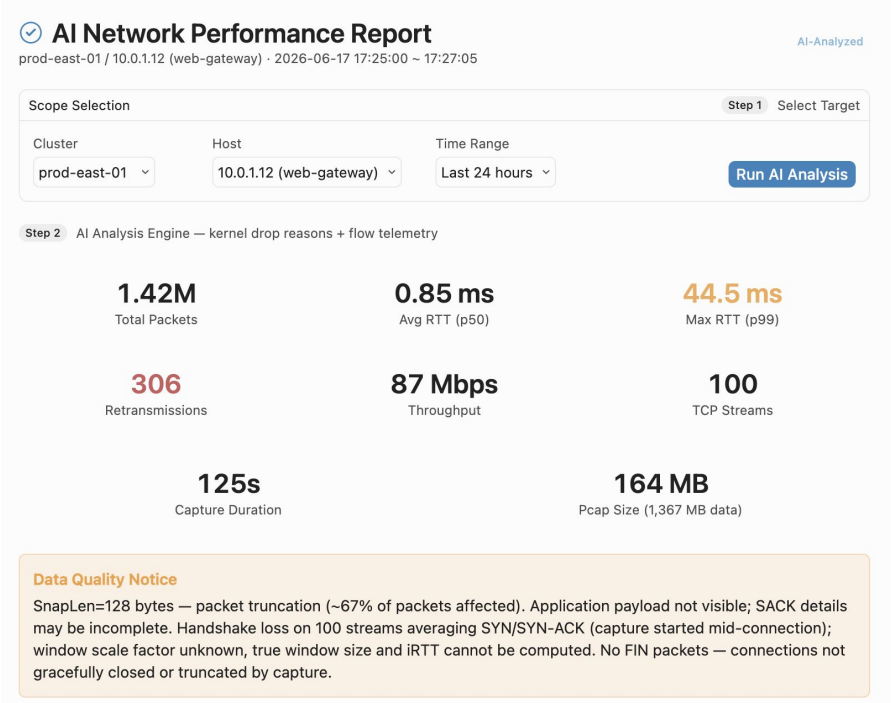


Figure 1: Agentic Blackboard (generated by AI)

Thank you!