The Future of SO_TIMESTAMPING

Jason Xing
<kernelxing@tencent.com>

Content

1. Brief Introduction
2. History
3. Current Status
4. BPF Timestamping Feature
5. Extending BPF Timestamping Feature

Socket Timestamping: Brief Introduction

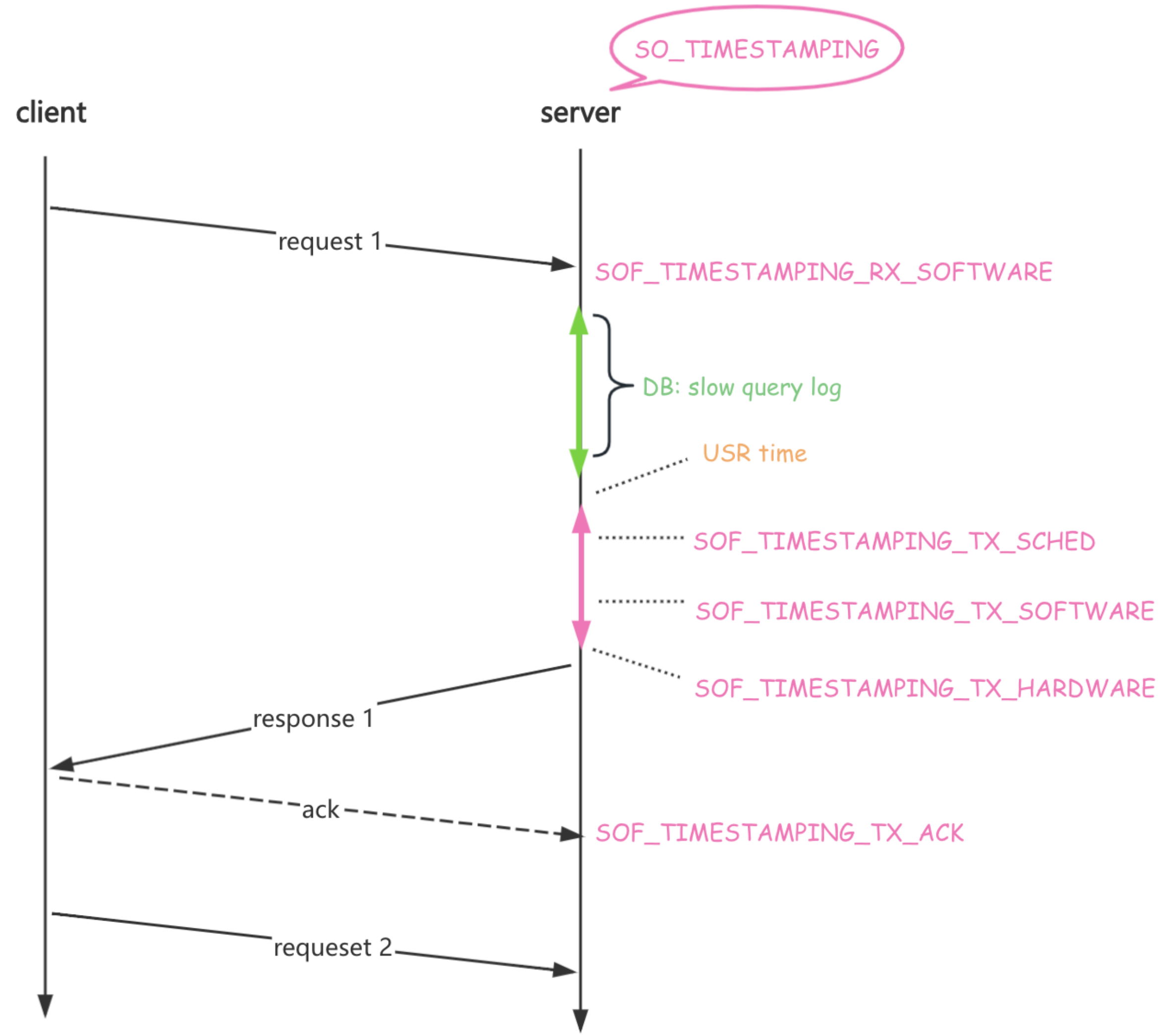
Brief Introduction

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. [1]

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.

[1]:
https://netdevconf.info/0x17/sessions/talk/so_timestamping-powering-fleetwide-rpc-monitoring.html



Socket Timestamping: Past and Present

History – 2009 (1)

Patrick Ohly <patrick.ohly@intel.com> implemented the first edition of socket timestamping in 2009.

commit cb9eff097831 (“net: new user space API for time stamping of incoming and outgoing packets”) defines uAPI in and brings the idea of report flags and generation flags(4 report flags VS 3 generation flags)

Commits like commit 51f31cabe3ce5 (“ip: support for TX timestamps on UDP and RAW sockets”) supports UDP/RAW sockets.

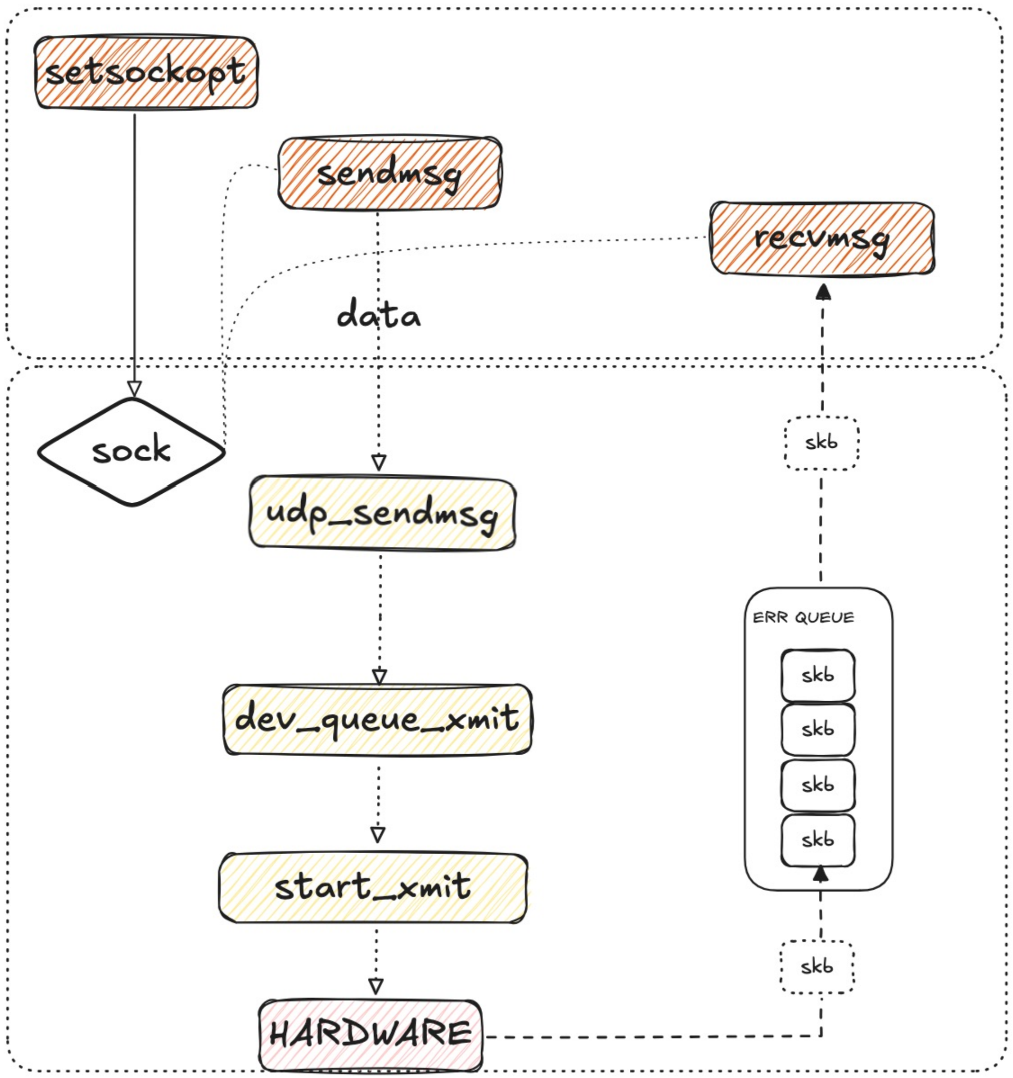
commit 20d4947353b (“net: socket infrastructure for SO_TIMESTAMPING”) provides explicit flag SO_TIMESTAMPING to allow application to enable the feature through setsockopt().

```
+SO_TIMESTAMPING:
+
+Instructs the socket layer which kind of information is wanted. The
+parameter is an integer with some of the following bits set. Setting
+other bits is an error and doesn't change the current state.
+
+SOF_TIMESTAMPING_TX_HARDWARE:  try to obtain send time stamp in hardware
+SOF_TIMESTAMPING_TX_SOFTWARE:  if SOF_TIMESTAMPING_TX_HARDWARE is off or
+                               fails, then do it in software
+SOF_TIMESTAMPING_RX_HARDWARE:  return the original, unmodified time stamp
+                               as generated by the hardware
+SOF_TIMESTAMPING_RX_SOFTWARE:  if SOF_TIMESTAMPING_RX_HARDWARE is off or
+                               fails, then do it in software
+SOF_TIMESTAMPING_RAW_HARDWARE: return original raw hardware time stamp
+SOF_TIMESTAMPING_SYS_HARDWARE: return hardware time stamp transformed to
+                               the system time base
+SOF_TIMESTAMPING_SOFTWARE:     return system time stamp generated in
+                               software
+
+SOF_TIMESTAMPING_TX/RX determine how time stamps are generated.
+SOF_TIMESTAMPING_RAW/SYS determine how they are reported in the
+following control message:
+    struct scm_timestamping {
+           struct timespec systime;
+           struct timespec hwtimetrans;
+           struct timespec hwtimeraw;
+    };
+
```

History – 2009 (2)

Patrick Ohly <patrick.ohly@intel.com> implemented the first edition of socket timestamping in 2009.

commit ac45f602ee3d (“net: infrastructure for hardware time stamping”) implements the communication framework between kernel and userspace. After this, many patches add more generation flags by the virtue of this mechanism.



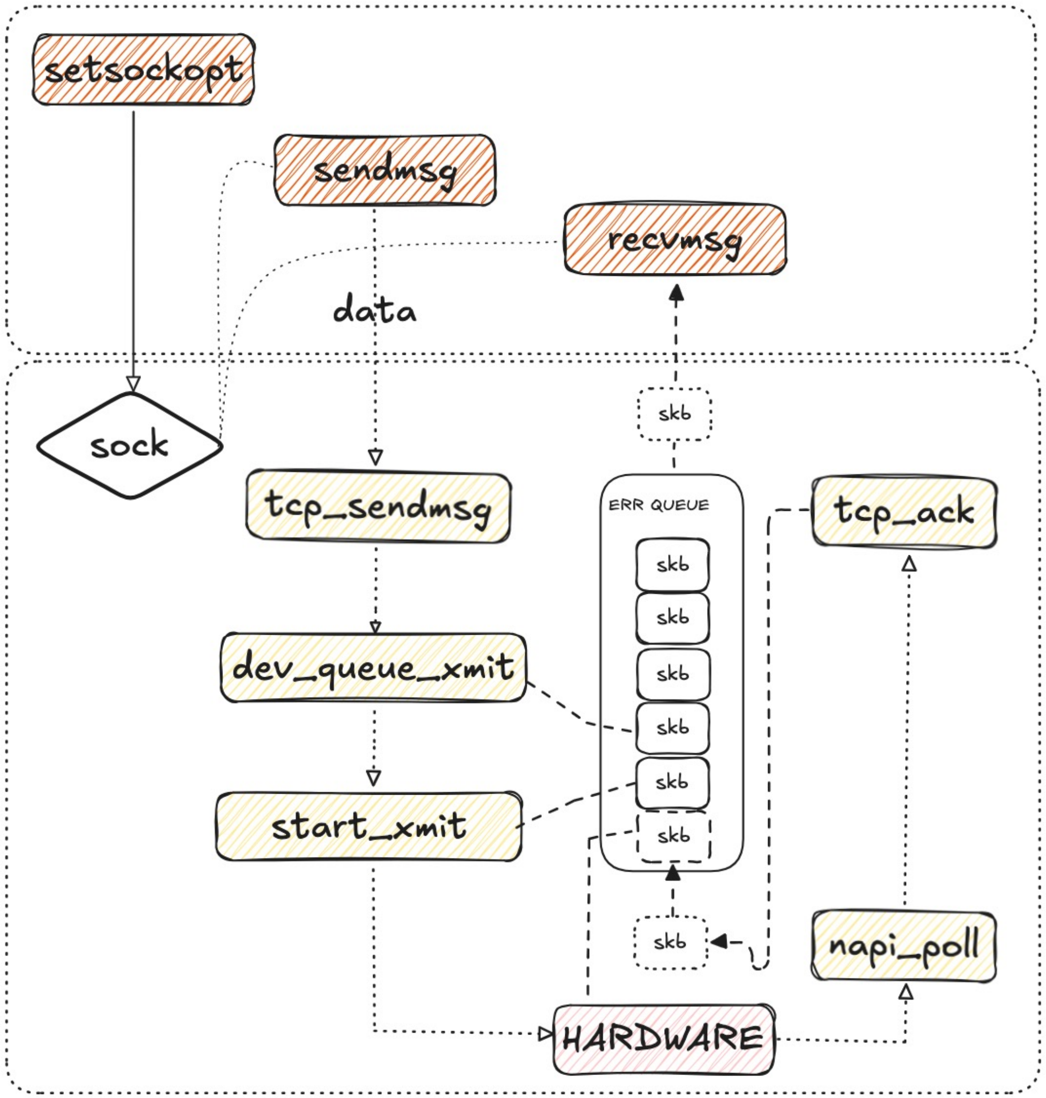
History – 2014 (1)

Willem de Bruijn <willemb@google.com>
fulfilled and enhanced socket timestamping in every aspect in 2009.

- Support TCP and UDP
- Add tskey to correlate each timestamped skb with corresponding sendmsg
- Add SCHED timestamp on entering packet scheduler

// One slide don't have enough room to list all the important commits, so sorry that I gave up.

Actually more details will be revisited later :)



Current Status

1. Academic studies (a few year ago): like Dapper and Fathom at Google
2. More and more hardwares have already supported timestamping feature.
3. New usage of tskey for UDP like OPT ID CMSG is done.
4. TX Completion feature is coming soon.
5. BPF timestamping is halfway done :)

BPF Timestamping Introduction

Optimization Idea

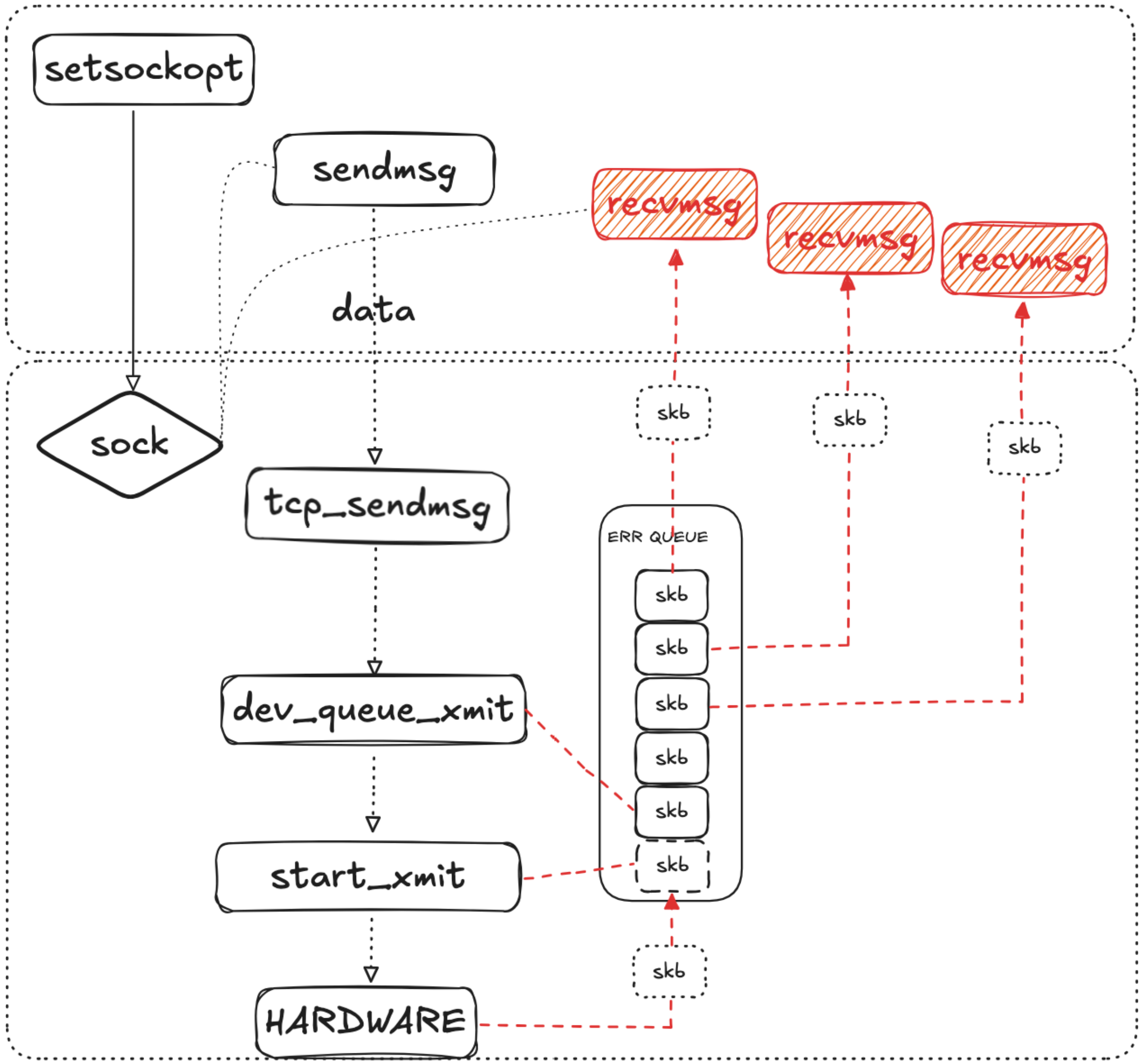
Let’s optimize it.

1. Applications modification required.

2. System overhead.
   - Extra X times calling recvmsg() per send
   - “20% degradation” mentioned in previous netdev

3. uAPI compatability
   - It’s not possible to change previous behavior

4. Inflexibility
   - Limited information to output unless we add more fields in kernel and then upgrade the kernel.



Background of BPF Timestamping feature

Before 2024, I had been haunted by numerous complicated issues reported from customers internally. At that time, I totally had no idea and had not come up with a good approach to have a clear insight of what happened in history in one of hundreds and thousands machines. Then I noticed SO_TIMESTAMPING that clearly and accurately helps us know where the latency issues come from, application, kernel, driver, physical link... In order to quickly use the feature in production, I planned to extend SO_TIMESTMAMPING feature by writing a kernel module so that we are able to transparently equip applications with this feature and require no modification in user side. In September 2024, we discussed at netconf and agreed that bpf is good way to fulfill it, which was mainly suggested by John Fastabend and Willem de Bruijn. Since upstreaming the first edition in October 2024, we've been through 13 revisions during nearly 5 months. Martin supported a significant BPF idea in this. So now I'm grateful…

Many thanks to Martin KaFai Lau, Willem de Bruijn, John Fastabend, Jakub Kicinski, Vadim Fedorenko for reviewing and testing this big patchset.

Many thanks to my colleagues, Yushan Zhou and Qian Huang, for cooperating to develop the robust kernel module internally.

Implementation – bpf_setsockopt()

- Add sk_bpf_cb_flags in struct sock.
  - Not only for TCP, but more protocols
- Add SK_BPF_CB_FLAGS
  - bpf_setsockopt() works becasuse of this flag
- Add SK_BPF_CB_TX_TIMESTAMPING
  - Used in transmit path to see if the flow needs to be monitored

https://web.git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/?id=24e82b7c045ba

```
diff --git a/include/net/sock.h b/include/net/sock.h
index 60ebf3c7b229..a95eedacae76 100644
--- a/include/net/sock.h
+++ b/include/net/sock.h
@@ -303,6 +303,7 @@ struct sk_filter;
   *    @sk_stamp: time stamp of last packet received
   *    @sk_stamp_seq: lock for accessing sk_stamp on 32 bit architectures only
   *    @sk_tsflags: SO_TIMESTAMPING flags
+  *    @sk_bpf_cb_flags: used in bpf_setsockopt()
   *    @sk_use_task_frag: allow sk_page_frag() to use current->task_frag.
   *                       Sockets that can be used under memory reclaim should
   *                       set this to false.
@@ -525,6 +526,8 @@ struct sock {
        u8                      sk_txtime_deadline_mode : 1,
                                sk_txtime_report_errors : 1,
                                sk_txtime_unused : 6;
+#define SK_BPF_CB_FLAG_TEST(SK, FLAG) ((SK)->sk_bpf_cb_flags & (FLAG))
+       u8                      sk_bpf_cb_flags;
```

```
 static int sol_socket_sockopt(struct sock *sk, int optname,
                               char *optval, int *optlen,
                               bool getopt)
@@ -5238,6 +5257,7 @@ static int sol_socket_sockopt(struct sock *sk, int optname,
        case SO_MAX_PACING_RATE:
        case SO_BINDTOIFINDEX:
        case SO_TXREHASH:
+       case SK_BPF_CB_FLAGS:
                if (*optlen != sizeof(int))
                        return -EINVAL;
                break;
@@ -5247,6 +5267,9 @@ static int sol_socket_sockopt(struct sock *sk, int optname,
                return -EINVAL;
        }

+       if (optname == SK_BPF_CB_FLAGS)
+               return sk_bpf_set_get_cb_flags(sk, optval, getopt);
+
```

Implementation – Isolation

- Allow BPF timestamping and socket timestamping work nearly at the same time. They works respectively without any confliction.

```
diff --git a/net/core/skbuff.c b/net/core/skbuff.c
index a441613a1e6c..341a3290e898 100644
--- a/net/core/skbuff.c
+++ b/net/core/skbuff.c
@@ -5539,6 +5539,23 @@ void skb_complete_tx_timestamp(struct sk_buff *skb,
 }
 EXPORT_SYMBOL_GPL(skb_complete_tx_timestamp);

+static bool skb_tstamp_tx_report_so_timestamping(struct sk_buff *skb,
+                                                 struct skb_shared_hwtstamps *hwtstamps,
+                                                 int tstype)
+{
+       switch (tstype) {
+       case SCM_TSTAMP_SCHED:
+               return skb_shinfo(skb)->tx_flags & SKBTX_SCHED_TSTAMP;
+       case SCM_TSTAMP_SND:
+               return skb_shinfo(skb)->tx_flags & (hwtstamps ? SKBTX_HW_TSTAMP :
+                                                    SKBTX_SW_TSTAMP);
+       case SCM_TSTAMP_ACK:
+               return TCP_SKB_CB(skb)->txstamp_ack;
+       }
+
+       return false;
+}
+
 void __skb_tstamp_tx(struct sk_buff *orig_skb,
                      const struct sk_buff *ack_skb,
                      struct skb_shared_hwtstamps *hwtstamps,
@@ -5551,6 +5568,9 @@ void __skb_tstamp_tx(struct sk_buff *orig_skb,
        if (!sk)
                return;

+       if (!skb_tstamp_tx_report_so_timestamping(orig_skb, hwtstamps, tstype))
+               return;
+
        tsflags = READ_ONCE(sk->sk_tsflags);
        if (!hwtstamps && !(tsflags & SOF_TIMESTAMPING_OPT_TX_SWHW) &&
            skb_shinfo(orig_skb)->tx_flags & SKBTX_IN_PROGRESS)
```

https://web.git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/?id=aa290f93a4a

Implementation – Correlation

How can we identify the matched skb with its sendmsg? How can we correlate sendmsg timestamp with its skb timestamps in every phase(SCHED/SOFTWARE/ACK)?

- Under the same socket lock protection, BPF program uses fentry to hook tcp_sendmsg_locked() and record current timestamp A.

- In tcp_tx_timestamp(), search skb's socket and then call bpf_sock_ops_enable_tx_tstamp() to tag the corresponding skb with SKBTX if any.

```
bpf_sock_ops_enable_tx_tstamp()
{
     skb_shinfo(skb)->tx_flags |= SKBTX_BPF;
     TCP_SKB_CB(skb)->txstamp_ack |=
                                TSTAMP_ACK_BPF;
     skb_shinfo(skb)->tskey = TCP_SKB_CB(skb)-
                            >seq + skb->len - 1;
}
```

```
diff --git a/net/ipv4/tcp.c b/net/ipv4/tcp.c
index 2171e2f045bb..298d1da05bee 100644
--- a/net/ipv4/tcp.c
+++ b/net/ipv4/tcp.c
@@ -496,6 +496,10 @@ static void tcp_tx_timestamp(struct sock *sk, struct sockcm_coo
                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);
 }
```

https://web.git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/?id=c9525d240c811

Implementation – SKBTX_BPF

- Introduce SKBTX_BPF
- We have exact four generation flags in BPF timestamping, SCHED, SW SND, HW SND, ACK
- __dev_queue_xmit() for SCHED timestamp
- Driver xmit (e.g. start_xmit() in virtio_net)
- Hardware PTP timestamp
- __skb_tstamp_tx() for ACK timestamp

https://web.git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/?id=6b98ec7e882a

https://web.git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/?id=ecebb17ad818

https://web.git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/?id=2deaf7f42b8c

https://web.git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/?id=b3b81e6b009

```
static void skb_tstamp_tx_report_bpf_timestamping(struct sk_buff *skb,
                                                  struct skb_shared_hwtstamp
                                                  struct sock *sk,
                                                  int tstype)
{
        int op;

        switch (tstype) {
        case SCM_TSTAMP_SCHED:
                op = BPF_SOCK_OPS_TSTAMP_SCHED_CB;
                break;
        case SCM_TSTAMP_SND:
                if (hwtstamps) {
                        op = BPF_SOCK_OPS_TSTAMP_SND_HW_CB;
                        *skb_hwtstamps(skb) = *hwtstamps;
                } else {
                        op = BPF_SOCK_OPS_TSTAMP_SND_SW_CB;
                }
                break;
        case SCM_TSTAMP_ACK:
                op = BPF_SOCK_OPS_TSTAMP_ACK_CB;
                break;
        default:
                return;
        }

        bpf_skops_tx_timestamping(sk, skb, op);
}
```

Implementation – Selective Sampling

- It's not realistic to store every timestamp generated. The limited storage is the problem.
- Add selective sampling function to allow BPF program to control the frequency of sampling in real workload.
- Google already adopts this method in production a few years ago.

https://web.git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/?id=59422464266f

```
diff --git a/net/core/filter.c b/net/core/filter.c
index 90a8fbc2e096..a0867c5b32b3 100644
--- a/net/core/filter.c
+++ b/net/core/filter.c
@@ -12103,6 +12103,25 @@ __bpf_kfunc int bpf_sk_assign_tcp_reqsk(struct __sk_buff *s,
 #endif
 }

+__bpf_kfunc int bpf_sock_ops_enable_tx_tstamp(struct bpf_sock_ops_kern *skops,
+                                              u64 flags)
+{
+        struct sk_buff *skb;
+
+        if (skops->op != BPF_SOCK_OPS_TSTAMP_SENDMSG_CB)
+                return -EOPNOTSUPP;
+
+        if (flags)
+                return -EINVAL;
+
+        skb = skops->skb;
+        skb_shinfo(skb)->tx_flags |= SKBTX_BPF;
+        TCP_SKB_CB(skb)->txstamp_ack |= TSTAMP_ACK_BPF;
+        skb_shinfo(skb)->tskey = TCP_SKB_CB(skb)->seq + skb->len - 1;
+
+        return 0;
+}
+
 __bpf_kfunc_end_defs();

 int bpf_dynptr_from_skb_rdonly(struct __sk_buff *skb, u64 flags,
@@ -12136,6 +12155,10 @@ BTF_KFUNCS_START(bpf_kfunc_check_set_tcp_reqsk)
 BTF_ID_FLAGS(func, bpf_sk_assign_tcp_reqsk, KF_TRUSTED_ARGS)
 BTF_KFUNCS_END(bpf_kfunc_check_set_tcp_reqsk)

+BTF_KFUNCS_START(bpf_kfunc_check_set_sock_ops)
+BTF_ID_FLAGS(func, bpf_sock_ops_enable_tx_tstamp, KF_TRUSTED_ARGS)
+BTF_KFUNCS_END(bpf_kfunc_check_set_sock_ops)
```

Tutorial

Step by step:

1. bpf_setsockopt in the init phase
2. Record socket and its timestamp in tcp_sendmsg()
3. In each stage, BPF timestamping callback will be triggered
4. In each callback, BPF program can generate current timestamp and calculate the latency.

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

```
SEC("fentry/tcp_sendmsg_locked")
int BPF_PROG(trace_tcp_sendmsg_locked, struct sock *sk, struct msghdr *msg,
             size_t size)
{
        __u32 pid = bpf_get_current_pid_tgid() >> 32;
        u64 timestamp = bpf_ktime_get_ns();
        u32 flag = sk->sk_bpf_cb_flags;
        struct sk_stg *stg;

        if (pid != monitored_pid || !flag)
                return 0;

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

        stg->sendmsg_ns = timestamp;
        nr_snd += 1;
        return 0;
}

SEC("sockops")
int skops_sockopt(struct bpf_sock_ops *skops)
{

        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;
        }
}
```

Noteworthy Points

tx_flags in skb

- We’re running out of precious skb_shinfo(skb)->tx_flags
- Willem has already reclaim one by removing SKBTX_HW_TSTAMP_USE_CYCLES.
- If the Bluetooth series adds the TX COMPLETION flag, no more available bit
- We’re going to add SKBRX_BPF that works in the receive path for BPF timestamping. Will we continue to free up one bit?

Discussion can be seen at the following link: https://lore.kernel.org/netdev/67b7b88c60ea0_292289294bb@willemb.c.googlers.com.notmuch/

No lock protection

- Without any socket lock protection, it might be not that accurate to acquire the fields from struct sock and friends.

Eric Dumazet once mentioned this at 2024 netconf

What are left to complete in BPF timestamping?

- UDP support
- RX support

Don’t worry. I’m working on it. Hopefully the remaining part will be finished in the first half of year.

uAPI compatability problem still exists…

- It seems not possible to solve this issue

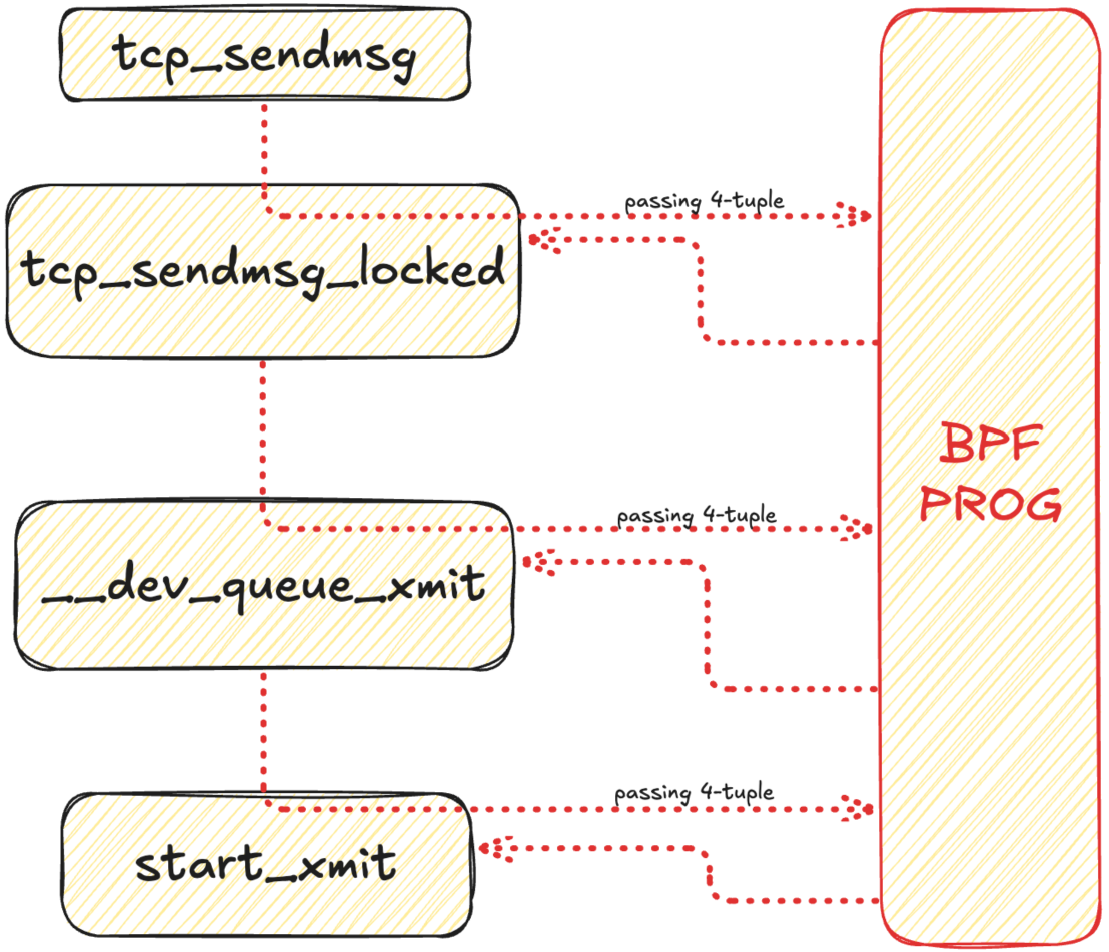
How to solve Interference Impact Issue?

What Is Interference Impact?

Every skb sent from every send syscall will go into the BPF program many times.

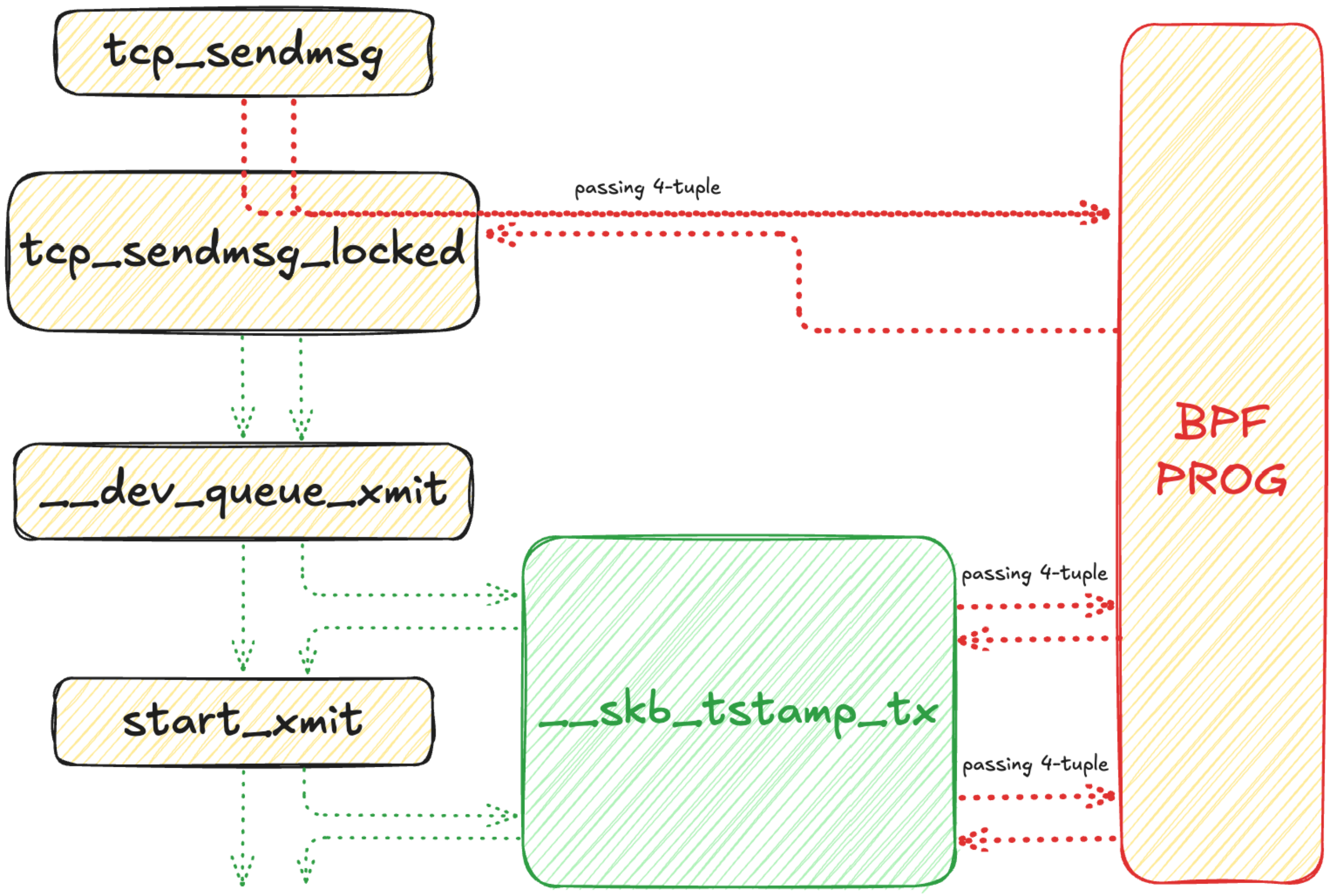
It will affect unmatched flows, which is against our expectations, causing inevitable performance degradation in real workload. We’ve seen that many times!

The right graph is how normal BPF based programs work. More complicated the prog is, more performance degradation it brings.



BPF Timestamping Has The Same Issue?

```
BPF timestmaping mitigates most of
the impacts of interference.

Good news is we only have one place
to handle in tcp_sendmsg_locked().

Bad news is we still have one place
to handle in tcp_sendmsg_locked().
```



Revisit How We Use Fentry

Before getting deeply into this chapter, let’s revisit a little bit on how we use in BPF timestamping case first.

The left code snippet shows the only place where we try fentry to hook tcp_sendmsg_locked().

Q: So the question is does it really matter?
A: Yes, it does matter!

```
SEC("fentry/tcp_sendmsg_locked")
int BPF_PROG(trace_tcp_sendmsg_locked, struct sock *sk, struct msghdr *msg,
             size_t size)
{
        __u32 pid = bpf_get_current_pid_tgid() >> 32;
        u64 timestamp = bpf_ktime_get_ns();
        u32 flag = sk->sk_bpf_cb_flags;
        struct sk_stg *stg;

        if (pid != monitored_pid || !flag)
                return 0;

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

        stg->sendmsg_ns = timestamp;
        nr_snd += 1;
        return 0;
}
```

Light-weighted GCC –mfentry Feature

fentry function is light-weighted and outperforms than bpftrace but it still bring ungelectable impact. We expect real ZERO impact on unmatched flow in the transmit path.

The reason behind it is that we indeed see a performance decrease after loading kernel module working similarly to BPF program in selftests.

Based on that I assume fentry has the same issue as ftrace.

How does fentry work?

```
crash> dis -l tcp_sendmsg_locked 10
/data/home/kernelxing/source_code/net-next.compileonly/net/ipv4/tcp.c: 1061
0xffffffff81fa1120 <tcp_sendmsg_locked>:      nopl   0x0(%rax,%rax,1) [FTRACE NOP]
0xffffffff81fa1125 <tcp_sendmsg_locked+5>:    push   %rbp
0xffffffff81fa1126 <tcp_sendmsg_locked+6>:    mov    %rsp,%rbp
0xffffffff81fa1129 <tcp_sendmsg_locked+9>:    push   %r15
0xffffffff81fa112b <tcp_sendmsg_locked+11>:   push   %r14
0xffffffff81fa112d <tcp_sendmsg_locked+13>:   mov    %rsi,%r14
0xffffffff81fa1130 <tcp_sendmsg_locked+16>:   push   %r13
0xffffffff81fa1132 <tcp_sendmsg_locked+18>:   mov    %rdi,%r13
0xffffffff81fa1135 <tcp_sendmsg_locked+21>:   push   %r12
0xffffffff81fa1137 <tcp_sendmsg_locked+23>:   push   %rbx
```

```
crash> dis -l tcp_sendmsg_locked 10
/data/home/kernelxing/source_code/net-next.compileonly/net/ipv4/tcp.c: 1061
0xffffffff81fa1120 <tcp_sendmsg_locked>:      call   0xffffffffa000d0c0
0xffffffff81fa1125 <tcp_sendmsg_locked+5>:    push   %rbp
0xffffffff81fa1126 <tcp_sendmsg_locked+6>:    mov    %rsp,%rbp
0xffffffff81fa1129 <tcp_sendmsg_locked+9>:    push   %r15
0xffffffff81fa112b <tcp_sendmsg_locked+11>:   push   %r14
0xffffffff81fa112d <tcp_sendmsg_locked+13>:   mov    %rsi,%r14
0xffffffff81fa1130 <tcp_sendmsg_locked+16>:   push   %r13
0xffffffff81fa1132 <tcp_sendmsg_locked+18>:   mov    %rdi,%r13
0xffffffff81fa1135 <tcp_sendmsg_locked+21>:   push   %r12
0xffffffff81fa1137 <tcp_sendmsg_locked+23>:   push   %rbx
```

Fentry Impact (1)

TEST 1: *do nothing* in libbpf program

taskset -c 1 netperf -H 127.0.0.1 -t TCP_STREAM

pps: 151935 pkts/sec
thr: 3432003 KB/sec

With fentry:
26145.01 10^6bits/sec

Without fentry:
27413.65 10^6bits/sec

The number decreases by **4.6%**!!

Good new is that I'm unable to see degradation in other tests.

```
SEC("fentry/tcp_sendmsg")
int BPF_PROG(tcp_sendmsg, struct sock *sk, struct msghdr *msg, size_t size)
{
        // do nothing
        return 0;
}

char LICENSE[] SEC("license") = "GPL";
```

Fentry Impact (2)

TEST 2: *read addr/port/pid only* in libbpf program

taskset -c 1 netperf -H 127.0.0.1 -t TCP_STREAM

pps: 151935 pkts/sec
thr: 3432003 KB/sec

With fentry:
25465.29 10^6bits/sec

Without fentry:
27413.65 10^6bits/sec

The number decreases by **7.1%**!!

```
static __always_inline int
filter_tcp_sendmsg(struct sock *sk)
{
        __u64 pid_tgid = bpf_get_current_pid_tgid();
        struct ipv4_flow_key key = {};

        BPF_CORE_READ_INTO(&key.saddr, sk, __sk_common.skc_rcv_saddr);
        BPF_CORE_READ_INTO(&key.daddr, sk, __sk_common.skc_daddr);
        BPF_CORE_READ_INTO(&key.dport, sk, __sk_common.skc_dport);

        return 0;
}


SEC("fentry/tcp_sendmsg")
int BPF_PROG(tcp_sendmsg, struct sock *sk, struct msghdr *msg, size_t size)
{
        filter_tcp_sendmsg(sk);

        return 0;
}

char LICENSE[] SEC("license") = "GPL";
```

How to Avoid tcp_sendmg Fentry Impact?

Add a new BPF timestamping callback in the very beginning of tcp_sendmsg() to replace the fentry usage that can be seen in selftest.

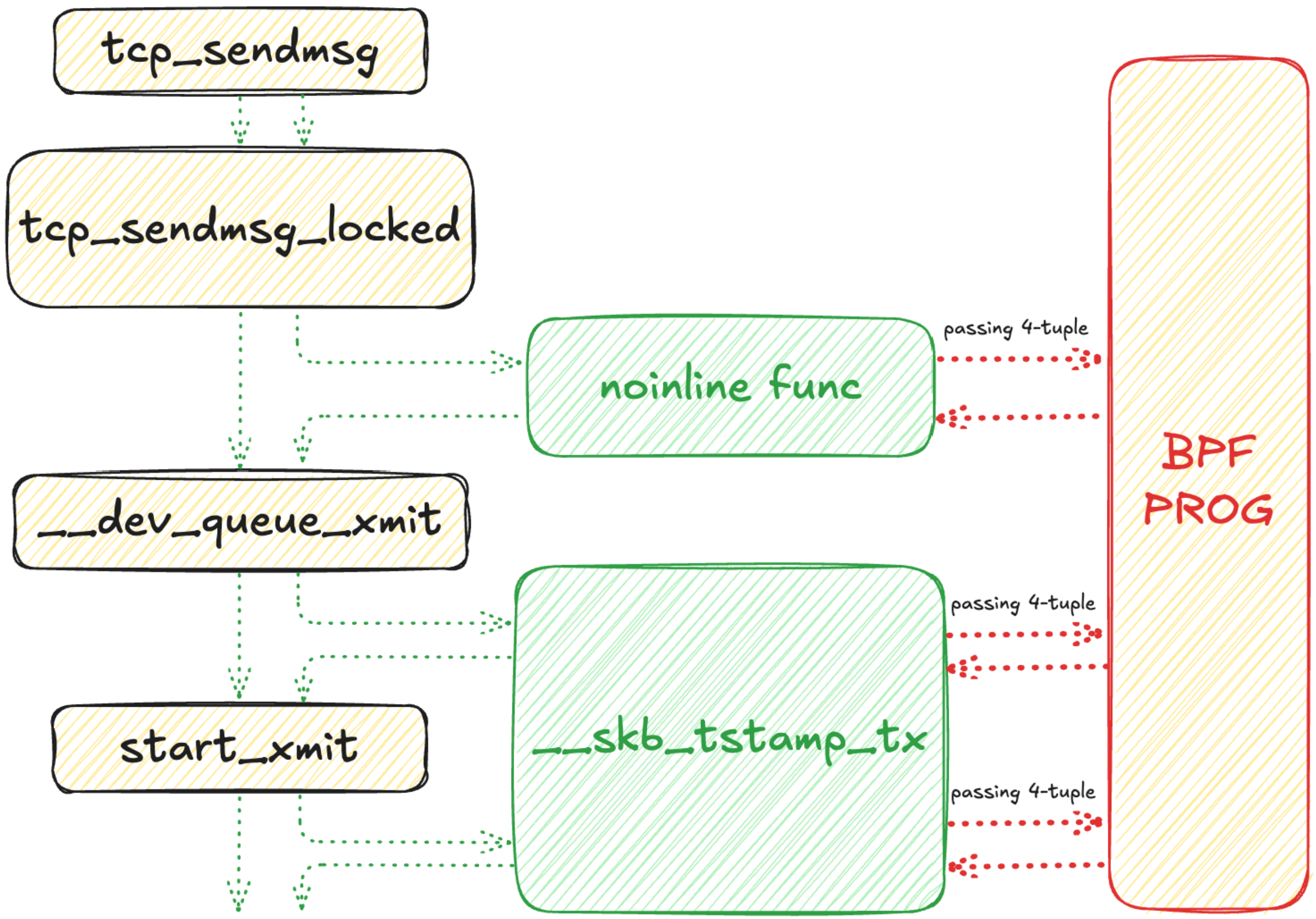
It’s done by the virtue of SK_BPF_CB_TX_TIMESTAMPING flag and noinline function.

Finally libbpf program would not have any impact on the normal flows that are not expected to be traced.

```
diff --git a/net/ipv4/tcp.c b/net/ipv4/tcp.c
index 118486692213..af662a6620a6 100644
--- a/net/ipv4/tcp.c
+++ b/net/ipv4/tcp.c
@@ -1057,6 +1057,11 @@ int tcp_sendmsg_fastopen(struct sock *sk, struct msghdr *msg, int *copied,
        return err;
 }

+static noinine bpf_tcp_sendmsg_callback(struct sock *sk)
+{
+       bpf_skops_tx_timestamping(sk, NULL, BPF_SOCK_OPS_TSTAMP_TCPSEND_CB);
+}
+
 int tcp_sendmsg_locked(struct sock *sk, struct msghdr *msg, size_t size)
 {
        struct tcp_sock *tp = tcp_sk(sk);
@@ -1069,6 +1074,10 @@ int tcp_sendmsg_locked(struct sock *sk, struct msghdr *msg, size_t size)
        int zc = 0;
        long timeo;

+       if (cgroup_bpf_enabled(CGROUP_SOCK_OPS) &&
+           SK_BPF_CB_FLAG_TEST(sk, SK_BPF_CB_TX_TIMESTAMPING))
+           bpf_tcp_sendmsg_callback(sk);
+
        flags = msg->msg_flags;

        if ((flags & MSG_ZEROCOPY) && size) {
```

Zero Interference Impact

With that new callback introduced,
then the issue will be solved.



Tracing every skb for BPF timestamping?

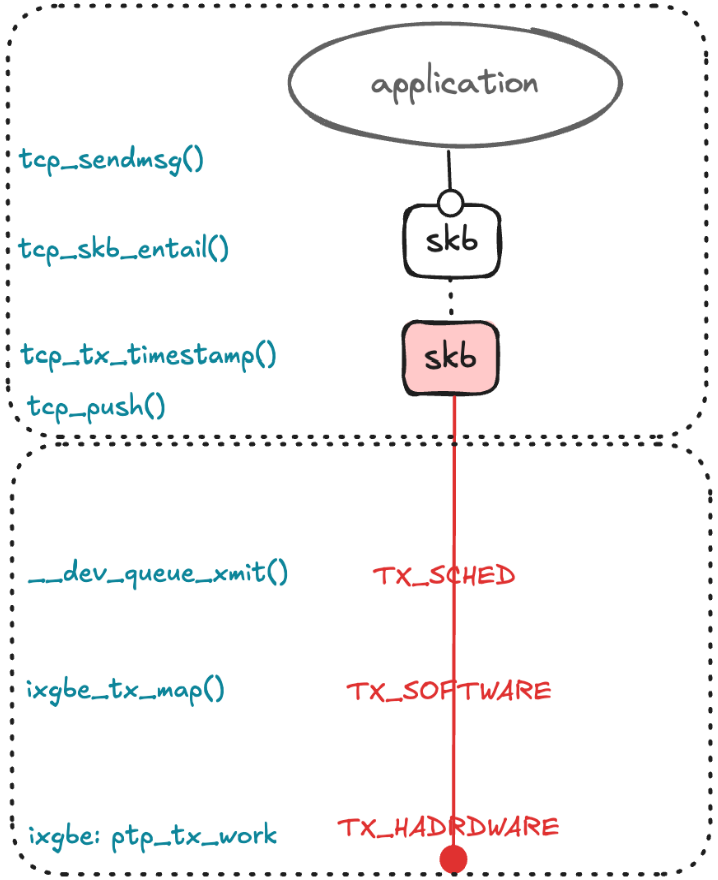
Current Implementation of TCP Flow

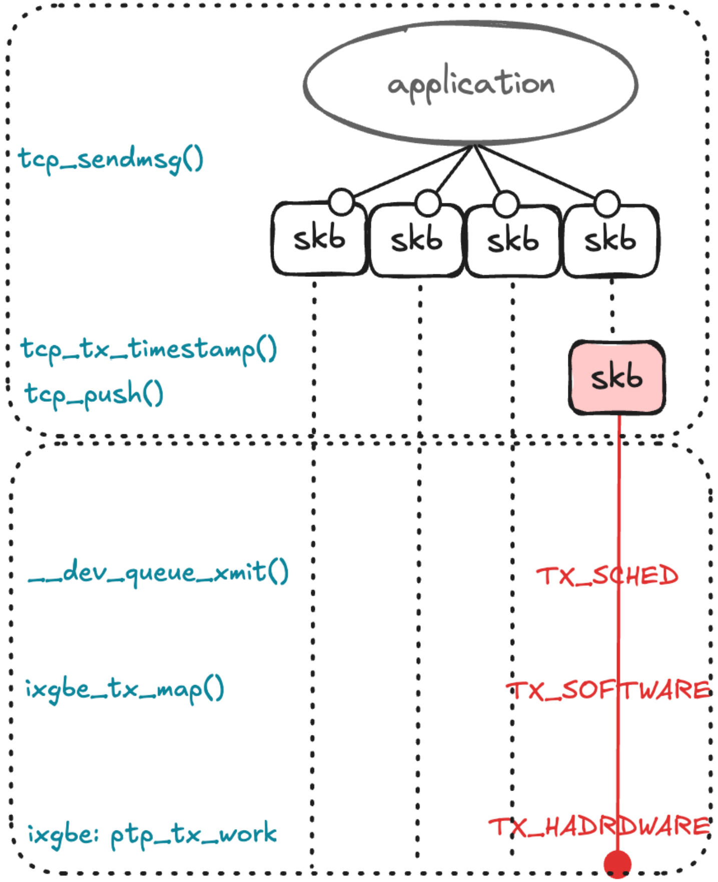
TCP

The feature interprets a send call on a bytestream as a request for a timestamp for the last byte in that send() buffer.

```
tcp_tx_timestamp()
{
    sock_tx_timestamp(sk, sockc,
                      &shinfo->tx_flags);
}
```

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4ed2d765dfacc





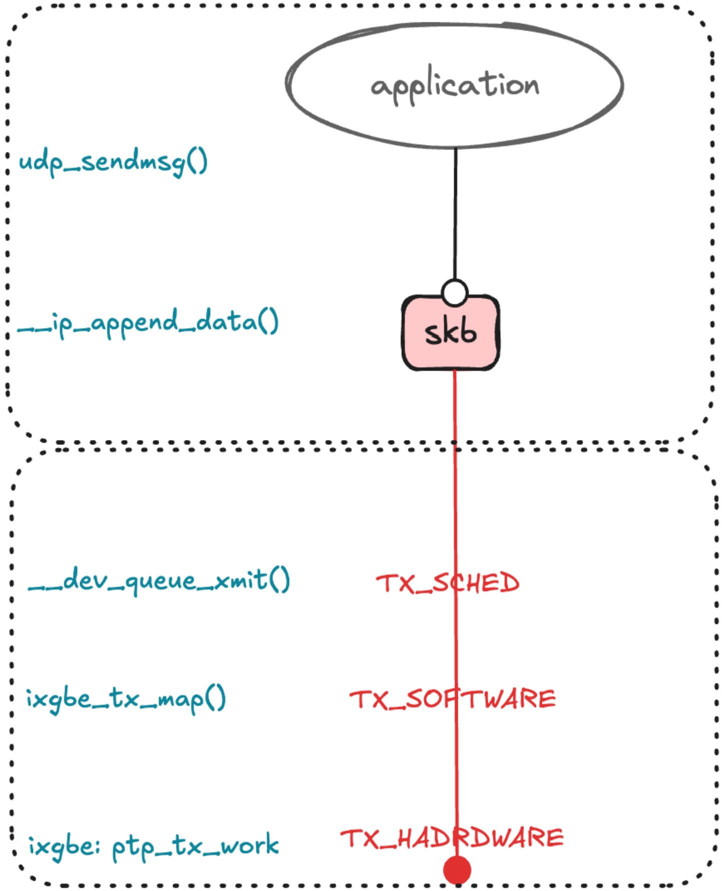
Current Implementation of UDP Flow

UDP

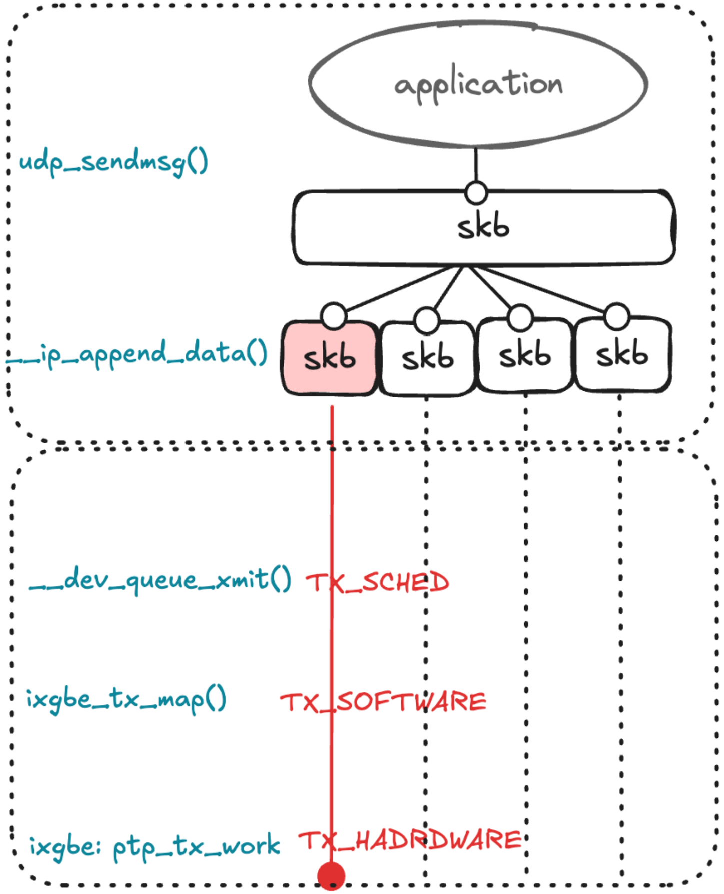
```
If the big outgoing packet is
fragmented or cumulative packets
are sent, then only the first
fragment/packet is timestamped.
However, it rarely happens in
pratice. Now we assume all the
UDP skbs are timestamped due to
protocol nature.
```

```
__ip_append_data()
{
    skb_shinfo(skb)->tx_flags =
                        cork->tx_flags;
    cork->tx_flags = 0;
}
```

```
https://git.kernel.org/pub/scm/l
inux/kernel/git/torvalds/linux.g
it/commit/?id=51f31cabe3ce5
```



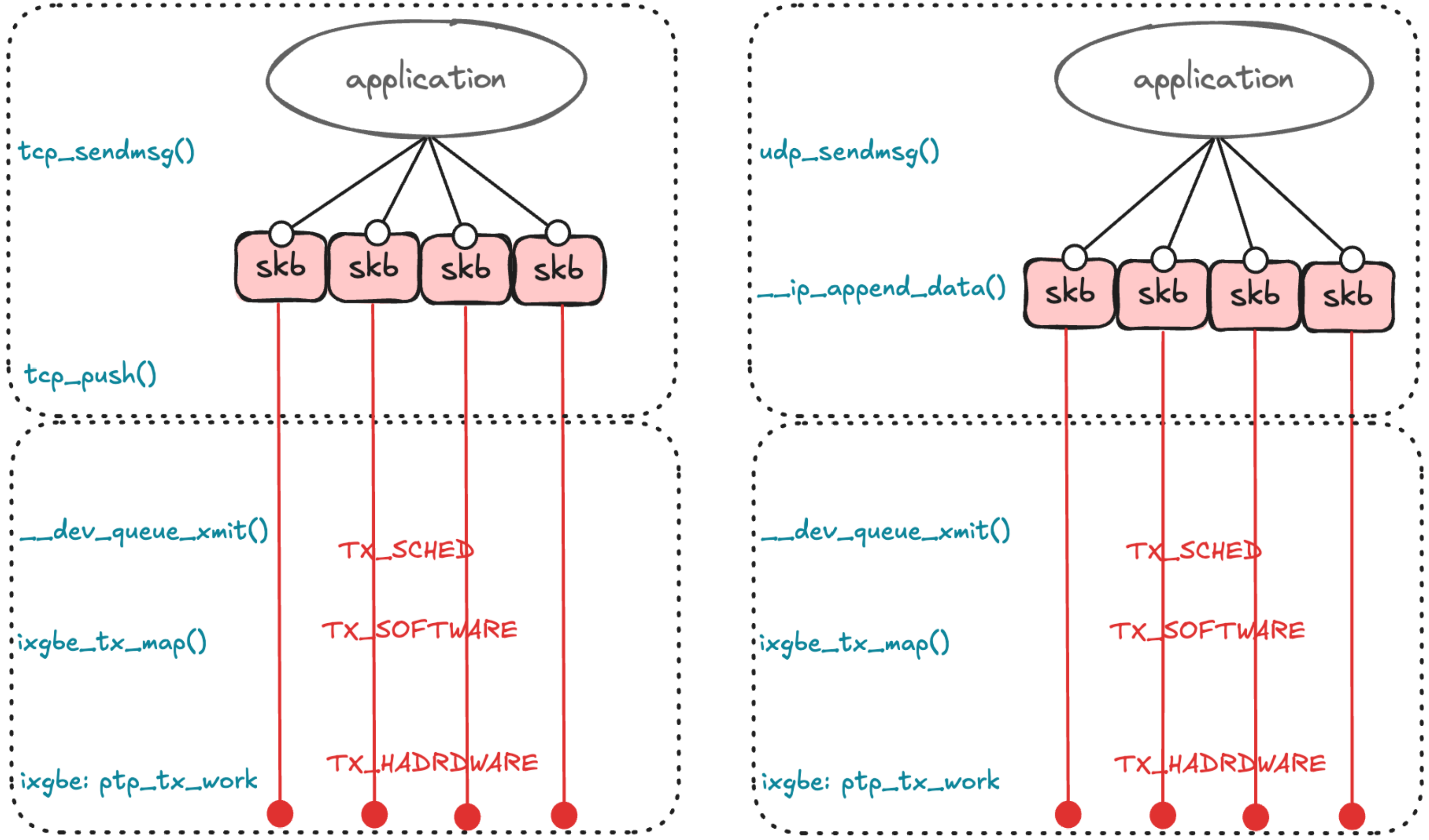
len < mtu case



IP fragmentation case

Outline of Tracing Every SKB

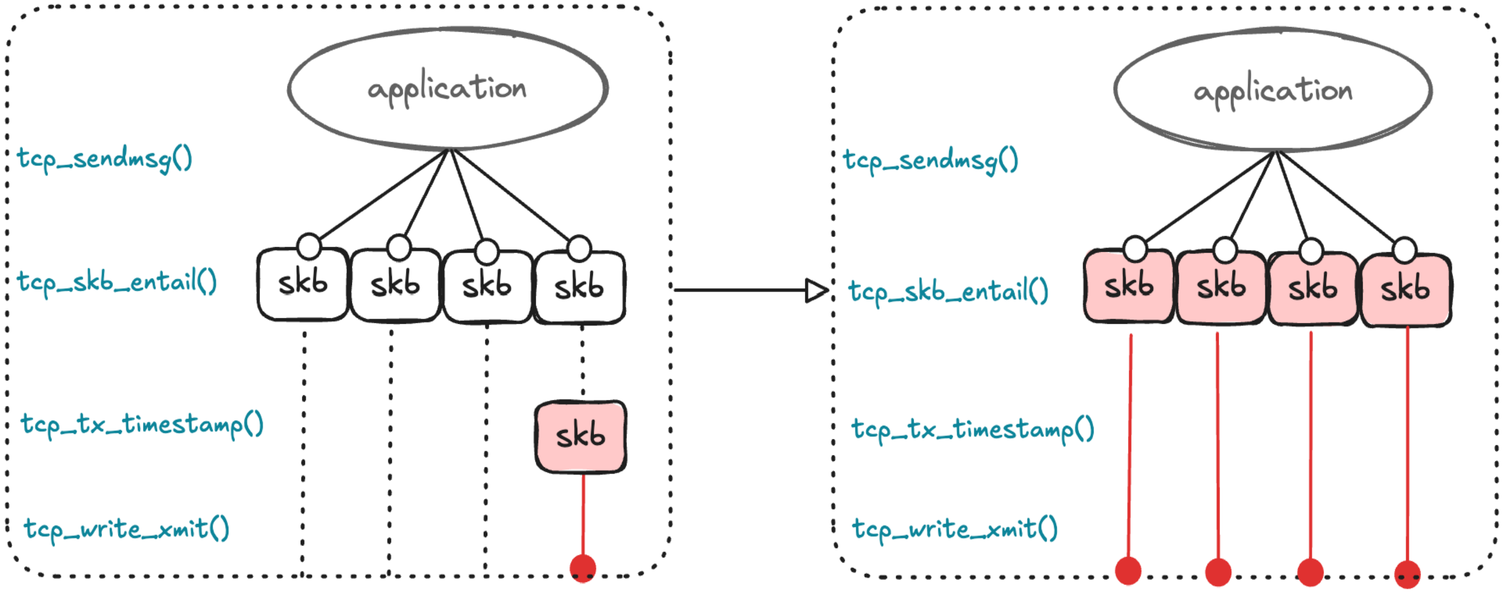
```
The right side graphs illustrate
everything - make sure each skb is
time stamped.

But how?
```



Tracing Every SKB for TCP (1)

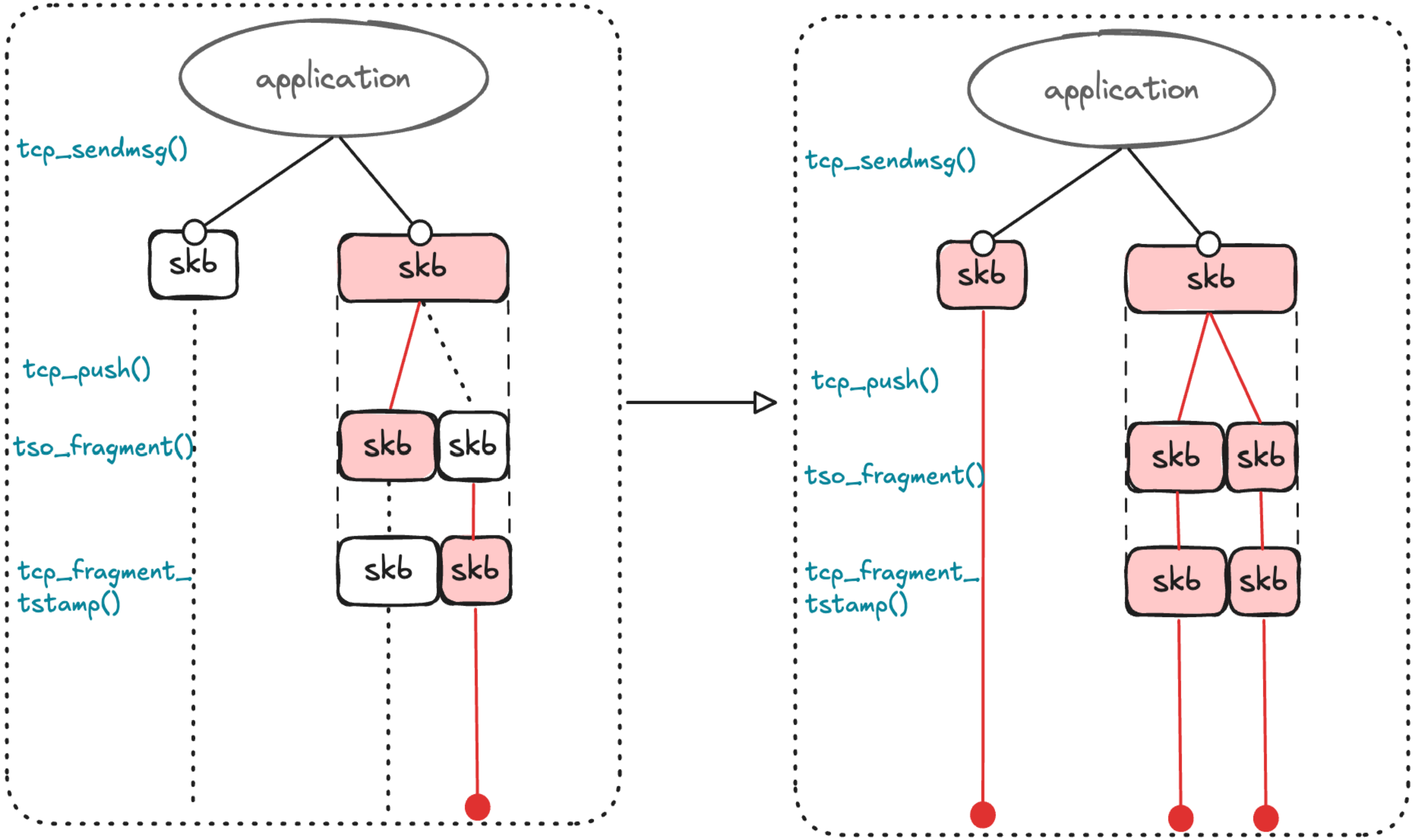
```
Keep in mind that:
1. GSO is on as a default setting
   for TCP.

We will do:
1. Tag the skb in allocation period,
   say, tcp_skb_entail().
2. In skb creation phase, call
   bpf_skops_tx_timestamping() to
   let BPF program selectively
   sample.
```

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



Tracing Every SKB for TCP (2)

```
We will do:
1. Handle tso_fragment() which might
   split the big packet into two
   skbs and only tag the last one.

Before:
tso_fragment()
  -> tcp_fragment_tstamp()
    // swap the old and new skb
    -> shinfo->tx_flags &= ~tsflags;
    -> shinfo2->tx_flags |= tsflags;

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



Tracing Every SKB for TCP (3)

We will do:

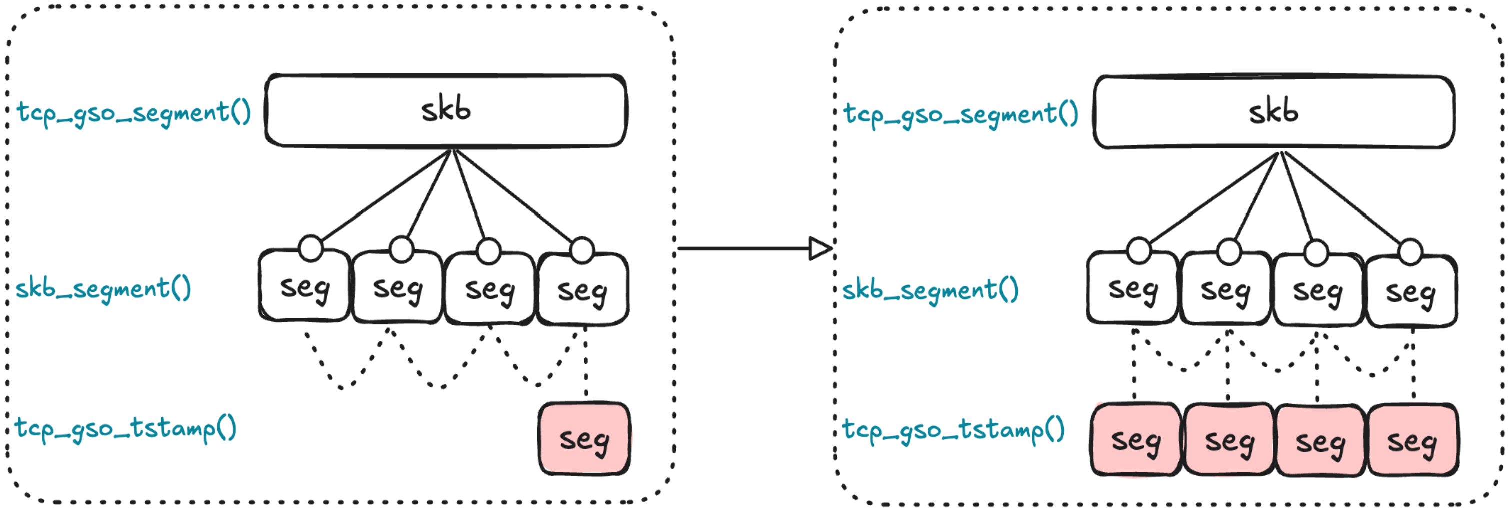
1. Handle tcp_gso_tstamp() case to make sure each skb is time stamped.

Before:

```
tcp_gso_tstamp()
  -> while (skb) {
         if (before(ts_seq, seq + mss)) {
             skb_shinfo(skb)->tx_flags |=
                          SKBTX_SW_TSTAMP;
```

After:

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



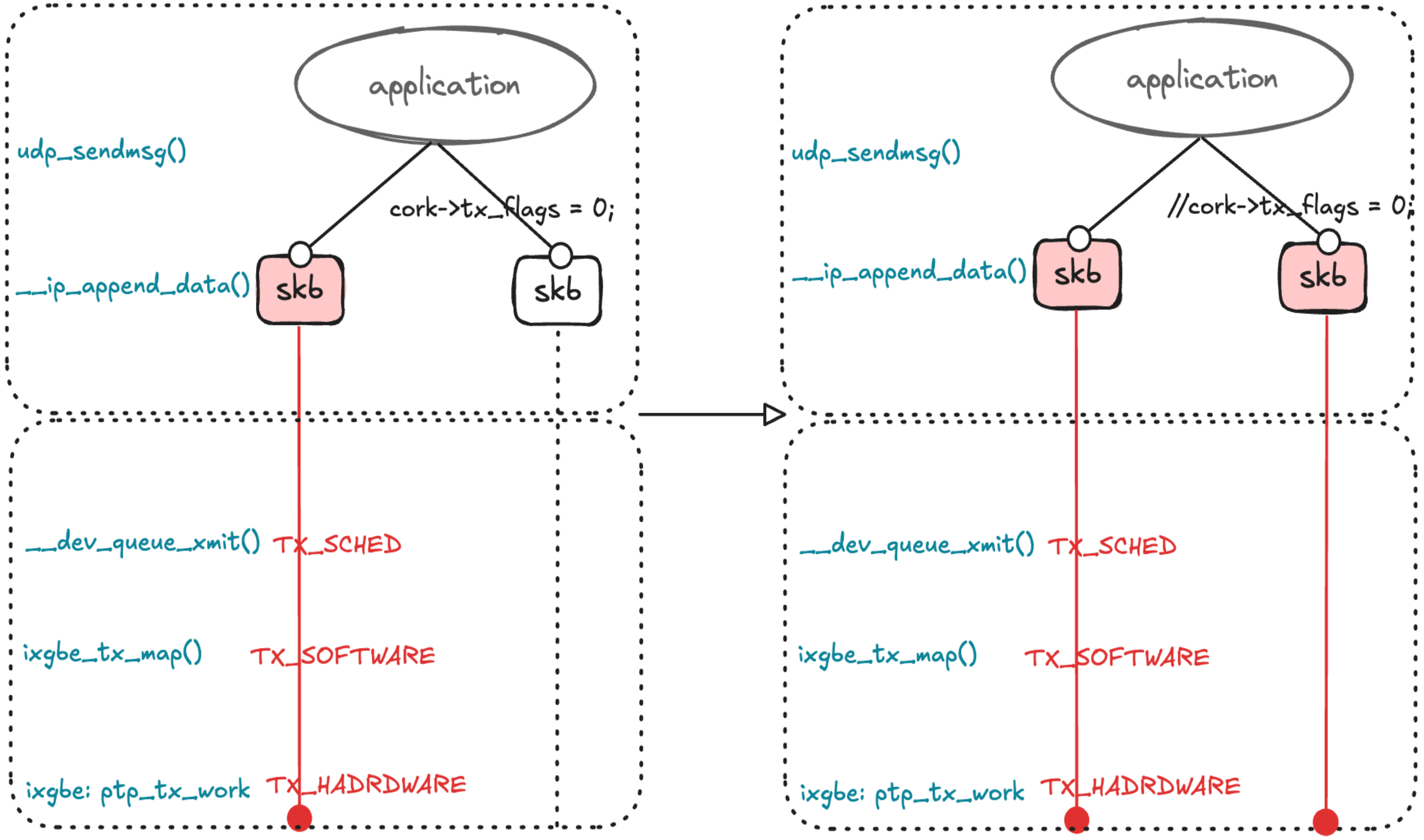
Tracing Every SKB for UDP (1)

Keep in mind that:
1. In almost all the cases, application sends a small packet, so there will no more fragments.
2. udp_cork or MSG_MORE option is seldomly used.

We will do:
1. Handle __ip_append_data() case.

Before:
```
__ip_append_data()
  -> tcp_fragment_tstamp()
    // the initial fragment is time stamped
    -> skb_shinfo(skb)->tx_flags = cork-
>tx_flags;
    -> cork->tx_flags = 0;
```

After:
```
__ip_append_data()
  -> tcp_fragment_tstamp()
    // each fragment is time stamped
    -> skb_shinfo(skb)->tx_flags = cork-
>tx_flags;
    -> // cork->tx_flags = 0;
```



Tracing Every SKB for UDP (2)

We will do:

1. No need to handle __udp_gso_segment() case? Since 1) hardware can only have one outstanding TS request at a time, 2) udp gso is not widely used. Otherwise, skb has already been fragmented to MSS sized seg in IP layer.

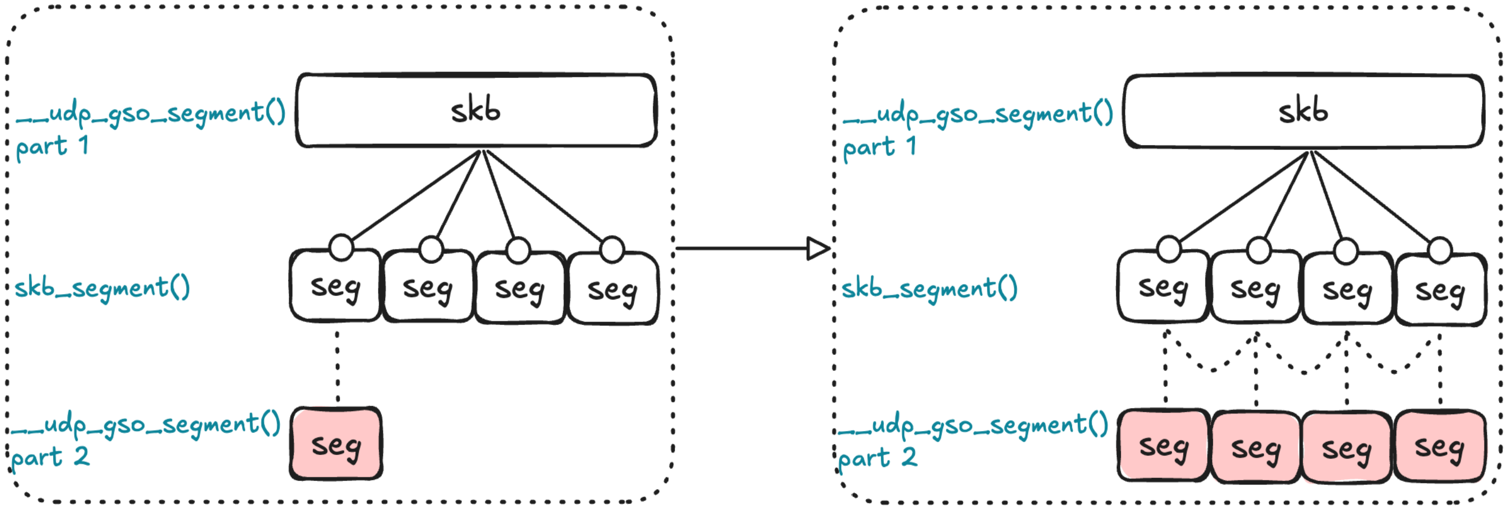
Before:

```
__udp_gso_segment()
  // only the first one keeps the tag
  -> skb_shinfo(seg)->tx_flags |=
          (skb_shinfo(gso_skb)->tx_flags &
           SKBTX_ANY_TSTAMP);
```

After:

```
__udp_gso_segment()
  -> // iterate the linked list from
     // the first seg, then set each one
```

https://web.git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=76e21533a48b



Merits vs Demerits

Merits:

- It helps us know the explicit latency between each skb, which approaches to some open sources that are developed based BPF, like Retis.

- It can be a good temporary tool to debug the kernel locally.

- It focus more on the kernel/stack itself instead of previous isolation function.

Demerits:

- It’s not that realistic to deploy normally in production. The shortage of storage remains a problem.

The Future of BPF Timestamping

What Is The End?

Can we thoroughly settle down the latency boundary issue like previously mentioned Retis? A more fine-grained solution is still appealing to me…

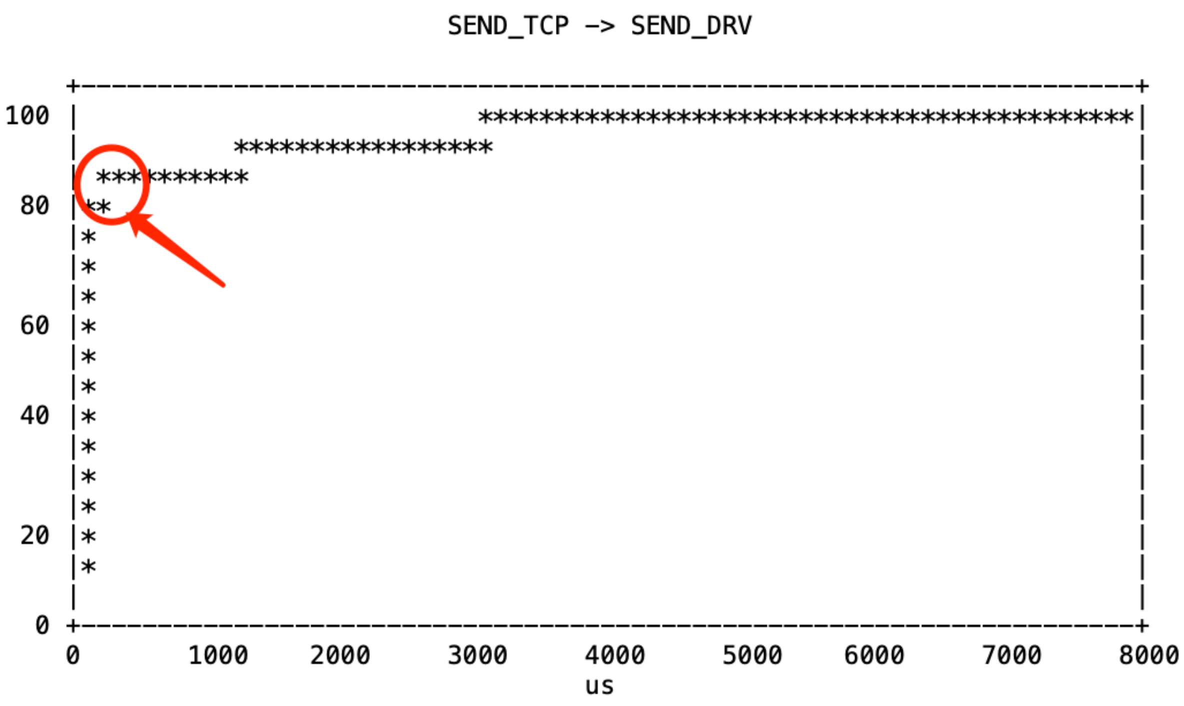
Will we add more hooks like in tcp_write_xmit() to see why the skb is not sending to the Qdisc by using kfunc?

Is It Possible to Replace tcpdump?

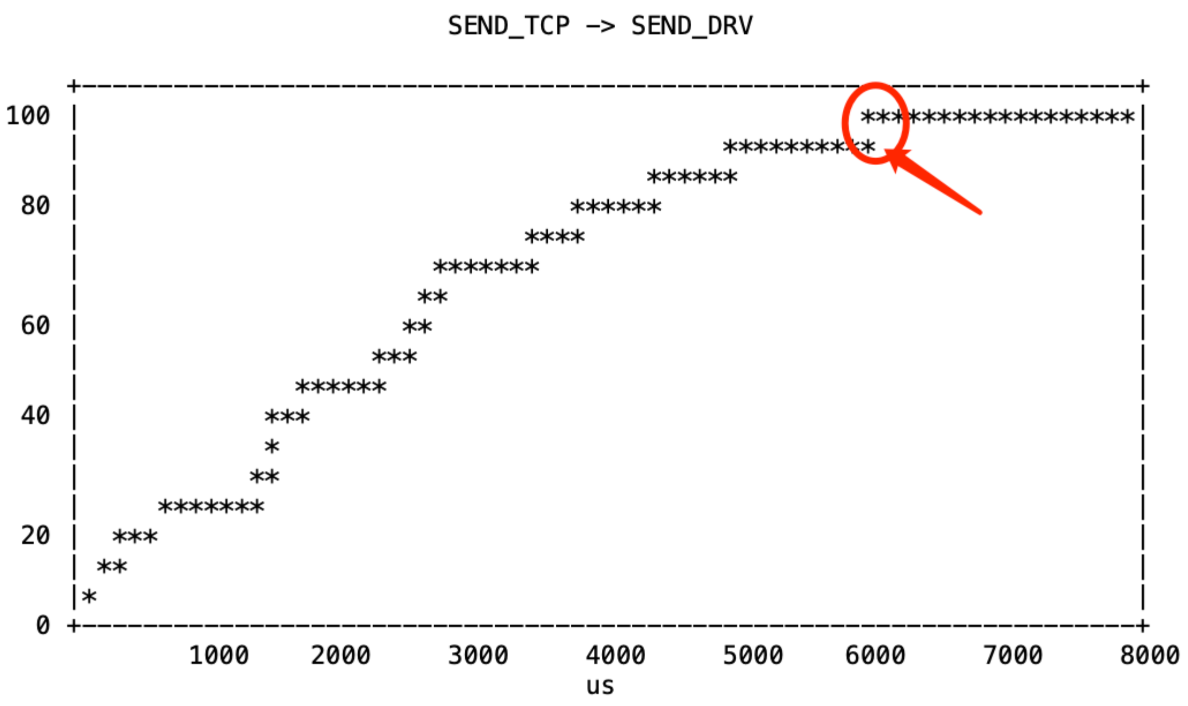
Now we have a better view of kernel behavior…

“Once the lifetimes of messages are constructed, they can be compared to identify anomalous processing that led to their latency deviations.” ——— from **How to diagnose nanosecond network latencies in rich end-host stacks NSDI’22**

Left CDF graph shows 90% flows complete transmitting every skb less than **1 ms** while the right one shows less than **6ms**.



high-performance VM



slow VM

Explore to leverage numerous real data?

Now we have so much useful information sent by the kernel, can we have a good approach to analyzing them? Or else, what a huge waste!

Thank you all !!!