

# XDP: 1.5 years in production. Evolution and lessons learned.

Author: Nikita V. Shirokov

## ABSTRACT

BPF and XDP have started a new era in kernel's programability for networking subsystem. Before the only way to do fast networking in Linux was to use kernel bypass technologies, such as DPDK[1] or NetMap[2]. There are lots of examples[3] of how XDP could be used, or reports of some small scale test deployments[4], but there is a lack of good examples of how XDP is being used in a big scale production network. This paper/presentation is trying to address this, by showing how some of BPF helpers could be used in network related code as well as explains what limitations could arise during operation of XDP and how these limitations could be overcome.

## INTRODUCTION

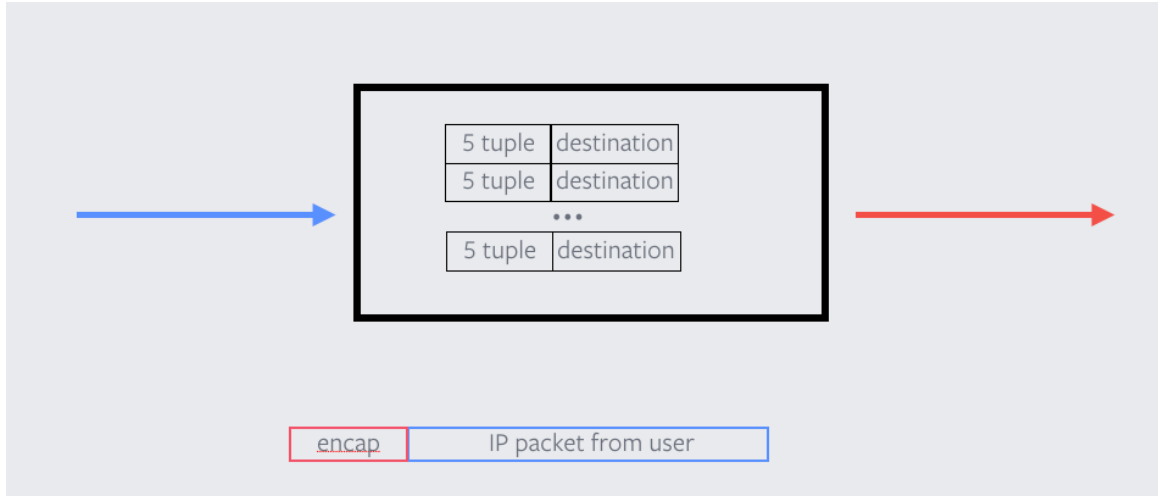
Every packet toward [facebook.com](https://www.facebook.com) has been processed by XDP enabled application since May of 2017. That gave us an unique experience of running XDP on large scale production network and prompted us to implement and improve existing BPF infrastructure. During the operation we also saw shortcomings and lack of tooling around XDP, which was not obvious/not thought about when we started our journey. Running XDP in production allowed us to measure and profile most of the BPF helpers and gave us insights of how they behave under real load/traffic.

## OVERVIEW

The XDP enabled application which we have started to run in May of 2017 is a L4 load balancer([5] on why we need this and how it's used. it's an open source solution[6]). In a nutshell it's a networking node which:

1. receives a packet from the external user
2. doing stateful lookup in connection's table, to check if it saw a packet from the same session (described by "5 tuple": (source ip, destination ip, source port, destination port, protocol)). the result of this lookup is either a MISS or a HIT.
3. in case of HIT matched value is an ip address of end node, where this packet is redirected.
4. in case of MISS there is additional logic to determine where the packet needs to be sent.
5. last step is to encapsulate original packet from the user into another IP header (that allow us to preserve original packet and implement DSR[7] schema for load balancing)

Picture 1 shows in a schematic view of L4 load balancer operation

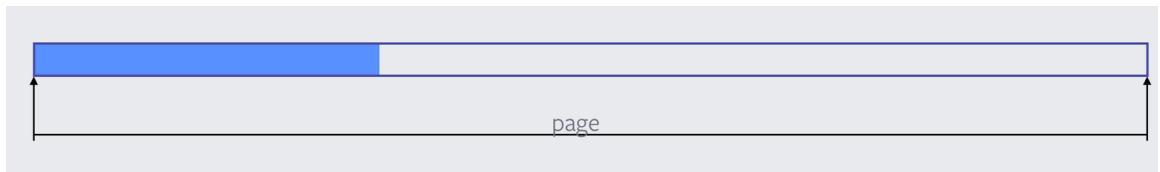


Picture 1: L4 load balancer at a glance

As for any stateful service, the worse case scenario is when it is under flood (every arrived packet belongs to a new connection, 0% HIT in connection's table ), and this is what we were optimizing our code for (because if it works good under such extreme conditions, it would also work great under day to day traffic load, which, in our environment, has connection's table HIT ratio close to 100%).

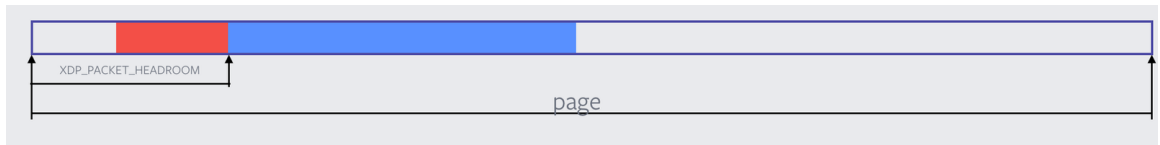
**IMPLEMENTATION**

To be able to do encapsulation, new `bpf_xdp_adjust_head[8]` helper was introduced. Before having this helper, XDP was working in a way, that NIC's driver were allocating page per packet and packet was written in the beginning of the page (Picture 2). Unfortunately such layout didn't allow any room for the encapsulation



Picture 2: initial XDP packet's layout/allocation

With introduction of `bpf_xdp_adjust_head` drivers starts to write packet on specific offset (driver's specific. for most of them it is equal to 256 bytes) and this helper allows manipulating pointer to the beginning of packet's data and therefore create a headroom for encapsulation (Picture 3). example of the usage could be found in Codeblock 1.



Picture 3: XDP packet layout/allocation after `bpf_xdp_adjust_head` was introduced

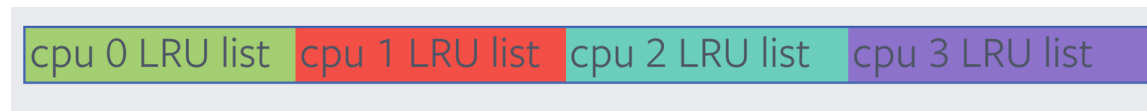
```

// ip(6)ip6 encap. move xdp->data to allow additional ipv6 header.
if (bpf_xdp_adjust_head(xdp, 0 - (int)sizeof(struct ipv6hdr))) {
    return false;
}
data = (void *) (long) xdp->data;
data_end = (void *) (long) xdp->data_end;

```

Codeblock 1: example of `bpf_xdp_adjust_head` usage

To store states of connections new BPF's map type was introduced: LRU[9]. It allowed to replace stale entries (old/dead connections) in a map w/ a new ones w/o any separate logic (such as timer + callback used, for example, for the same purpose in IPVS). Together with a new map's type special flag was introduced - `BPF_F_NO_COMMON_LRU`. It allows for contention free updates (each CPU do inserts/ LRU housekeeping only in dedicated zone of the map) w/ "single key - single value" model (compare to per-cpu maps, where each cpu could have dedicated value (and hence it's is lock free for value updates as well), but still share the same key). Schematic example of how `BPF_F_NO_COMMON_LRU` works could be seen on Picture 4. Each CPU updates (write) only dedicated area (color coded) but can do lookup (read) across whole map.



Picture 4: example of LRU map w/ `BPF_F_NO_COMMON_LRU` flag specified

## OPERATIONAL EXPERIENCE AND LESSONS LEARNED

A big benefit for us was that when we were writing our XDP application we were aware of environment where it is going to be used. For example there was no BPF helper for routing lookups[10], so instead, as in our environment servers are connected with a single interface to the L3 capable top of the rack switch, while sending packet from the load balancer, instead of doing full route lookup on a server side, we were rewriting destination mac to be a mac address of the L3 switch. That allowed us to do "offloading" of route lookups to the switch. For intrarack traffic this model would work as fast as L2 switching, because most of the modern L3 switches has the same speed of L2/L3 lookups.

Also we would recommend to do proper performance testing of XDP software in your setup. XDP can bring a huge improvements for packet processing, but software/CPU is not always a bottleneck. Unfortunately network interface cards still acts as black boxes (hardware/firmware part of it) even in Linux. Some example which we have found during our tests, where bottlenecks in NIC were way before software/CPU limits of the host:

1. Vendor/model of the NIC dictates packet processing rate (e.g. small packet vs large packet). Unfortunately there is serious lack of counters, reported by NIC itself. e.g. if NIC is bottlenecked by doing packet parsing for RSS - there is no counter which would show you that
2. speaking of RSS - noticeable differences in RSS performance were observed between TCP and UDP traffic
3. IPv6 packet processing (most likely RSS) is usually around 10-20% slower.

While deploying XDP you also need to think about how you are going to monitor network usage of the host. As XDP acts before the kernel, if packet was dropped (`XDPA_DROP`) or transmitted (`XDPA_TX`), kernel counters wont reflect this. This could lead to situation when host, doing millions of packets per seconds, tens of gigabits of traffic, would report almost 0 usage to the monitoring system, just because all monitoring gear was relying on the counters, provided by

kernel. Solution here could be either implement counters in XDP program itself and expose them to the monitoring, or starting to use counters, reported by hardware (e.g. from ethtool output)

Requirements of multiple XDP programs and lack of native support for it (on a single interface), was a big concern as well: for example we need to run tool for network debugging, firewall and load balancer on a same host at the same time in XDP context. Fortunately this limitation could be overcome with a use of combination of BPF's program's array and `bpf_tail_call`. Idea is pretty simple: instead of attaching full-fledged BPF program to the interface directly, we are attaching simple BPF program (we started to call it a "root" BPF/XDP program), the only purpose of which is to try to call other BPF's programs from BPF's program array (see Codeblock 2 for the example). Idea is to share this program's array across multiple BPF programs (e.g. through pinning). Under normal circumstances, a `bpf_tail_call` control flow does not go back to calling program (compare to the regular function call), so this limitations needs to be addresses in chained BPF program by:

1. each BPF program need to know on which position it's attached (so there wont be any recursion in `bpf_tail_calls`)
2. it needs to implement same logic for `bpf_tail_call` in the end of it's run (see Codeblock 3 for the example)

```
#include <uapi/linux/bpf.h>
#include "bpf_helpers.h"

#define ROOT_ARRAY_SIZE 3

struct bpf_map_def SEC("maps") root_array = {
    .type = BPF_MAP_TYPE_PROG_ARRAY,
    .key_size = sizeof(__u32),
    .value_size = sizeof(__u32),
    .max_entries = ROOT_ARRAY_SIZE,
};

SEC("xdp-root")
int xdp_root(struct xdp_md *ctx) {
    __u32 *fd;
    #pragma clang loop unroll(full)
    for (__u32 i = 0; i < ROOT_ARRAY_SIZE; i++) {
        bpf_tail_call(ctx, &root_array, i);
    }
    return XDP_PASS;
}

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

Codeblock 2: example of "root" BPF program.

```
#pragma clang loop unroll(full)
for (int i = 1; i < 16; i++) {
    jmp.call((void *)ctx, i);
}
return XDP_PASS;
```

Codeblock 3: example of how chained BPF program should behave on exit. in this example BPF program was registered on position 0 in prog array and thats why it starts to scan it from position 1 (everything behind itself)

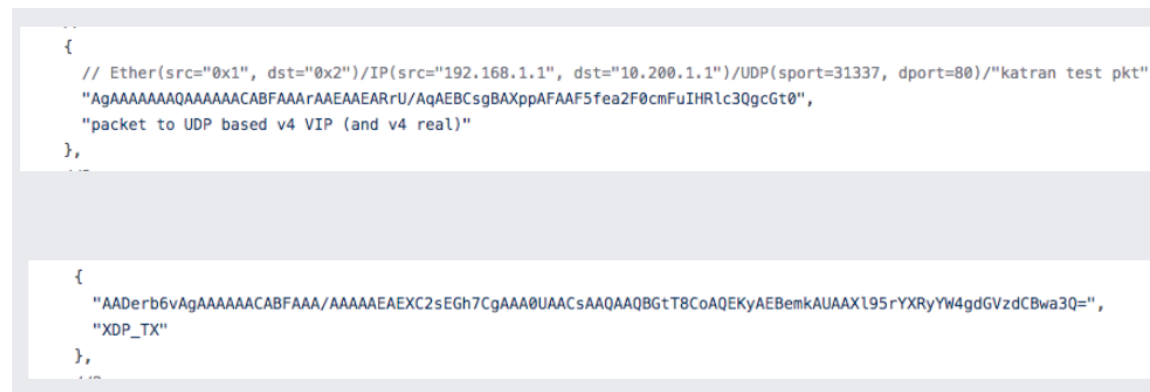
Speaking of debugging: the biggest operation pain point while we have started to deploy XDP was that debug/troubleshooting tools stop to work. Namely tcpdump, because it operates within kernel's TCP/IP stack and XDP runs before it. To be able to do proper debugging and with a help of XDP chaining xdpdump [10] tool was introduced. It did allow us to capture packets before any other XDP/BPF program would run and gave us an ability to save this output in pcap format for further offline processing.

Overall from operational point of view XDP was pretty stable and there wasn't any major outage because of this or errors, which wasn't been found in our testing environment.

## TESTING

Initially testing of a new code were require full topology to be set up in controlled/lab environment. Every test run was consuming a lot of time and it was hard to automate this process. So BPF\_PROG\_TEST\_RUN[11] feature was introduced. Idea was that you can load your BPF program, specify what to use as an input (pointer of the memory location with input packet), and where the result of the run should be written (pointer to where modified packet is going to be written). Also BPF\_PROG\_TEST\_RUN would inform caller of XDP's exit code (for example XDP\_TX, XDP\_PASS etc). In our test case we were comparing resulting packet with expected one and if there was a match, test was considered as successful.

Picture 5 shows example of how input and output data for tests could looks like. e.g. we are using base64 encoded packet as an input and output.



```
...
{
  // Ether(src="0x1", dst="0x2")/IP(src="192.168.1.1", dst="10.200.1.1")/UDP(sport=31337, dport=80)/"katran test pkt"
  "AgAAAAAAAAQAAAAACABFAAArAAEAAEARrU/AqAEBCsgBAXppFAAF5fea2F0cmFuIHRlc3QgcGt0",
  "packet to UDP based v4 VIP (and v4 real)"
},
...

{
  "AADerb6vAgAAAAACABFAAA/AAAAEAEXC2sEGh7CgAAA0UAACsAAQAAQBGtT8CoAQEKyAEBemKAUAAXl95rYXRyYW4gdGVzdCBwa3Q=",
  "XDP_TX"
},
...
```

Picture 5: example of input and output data for BPF\_PROG\_TEST\_RUN

As a result we have built a collection of unittests for every codepath in our load balancer's code and it became really easy to test any new feature, to make sure that nothing is broken and to do it in automatic way. Picture 6 shows an example of how output of unittests could looks like.

```

I0904 19:25:18.770014 20857 XdpTester.cpp:167] Test: packet to UDP based v4 VIP (and v4 real) result: Passed
I0904 19:25:18.770071 20857 XdpTester.cpp:167] Test: packet to TCP based v4 VIP (and v4 real) result: Passed
I0904 19:25:18.770124 20857 XdpTester.cpp:167] Test: packet to TCP based v4 VIP (and v4 real; any dst ports). result: Passed
I0904 19:25:18.770166 20857 XdpTester.cpp:167] Test: packet to TCP based v4 VIP (and v6 real) result: Passed
I0904 19:25:18.770216 20857 XdpTester.cpp:167] Test: packet to TCP based v6 VIP (and v6 real) result: Passed
I0904 19:25:18.770243 20857 XdpTester.cpp:167] Test: v4 ICMP echo-request result: Passed
I0904 19:25:18.770279 20857 XdpTester.cpp:167] Test: v6 ICMP echo-request result: Passed
I0904 19:25:18.770328 20857 XdpTester.cpp:167] Test: v4 ICMP dest-unreachabe fragmentation-needed result: Passed
I0904 19:25:18.770391 20857 XdpTester.cpp:167] Test: v6 ICMP packet-too-big result: Passed
I0904 19:25:18.770426 20857 XdpTester.cpp:167] Test: drop of IPv4 packet w/ options result: Passed
I0904 19:25:18.770458 20857 XdpTester.cpp:167] Test: drop of IPv4 fragmented packet result: Passed
I0904 19:25:18.770498 20857 XdpTester.cpp:167] Test: drop of IPv6 fragmented packet result: Passed
I0904 19:25:18.770531 20857 XdpTester.cpp:167] Test: pass of v4 packet with dst not equal to any configured VIP result: Passed
I0904 19:25:18.770571 20857 XdpTester.cpp:167] Test: pass of v6 packet with dst not equal to any configured VIP result: Passed
I0904 19:25:18.770597 20857 XdpTester.cpp:167] Test: pass of arp packet result: Passed
I0904 19:25:18.770635 20857 XdpTester.cpp:167] Test: LRU hit result: Passed
I0904 19:25:18.770673 20857 XdpTester.cpp:167] Test: packet #1 dst port hashing only result: Passed
I0904 19:25:18.770711 20857 XdpTester.cpp:167] Test: packet #2 dst port hashing only result: Passed
I0904 19:25:18.770748 20857 XdpTester.cpp:167] Test: ipinip packet result: Passed
I0904 19:25:18.770799 20857 XdpTester.cpp:167] Test: ipv6inipv6 packet result: Passed
I0904 19:25:18.770843 20857 XdpTester.cpp:167] Test: ipv4inipv6 packet result: Passed
I0904 19:25:18.770880 20857 XdpTester.cpp:167] Test: QUIC: long header. Client Initial type. LRU miss result: Passed
I0904 19:25:18.770918 20857 XdpTester.cpp:167] Test: QUIC: long header. 0-RTT Protected. CH. LRU hit. result: Passed
I0904 19:25:18.770956 20857 XdpTester.cpp:167] Test: QUIC: long header. Handshake. v4 vip v6 real. Conn Id based. result: Passed
I0904 19:25:18.771003 20857 XdpTester.cpp:167] Test: QUIC: long header. client initial. v6 vip v6 real. LRU miss result: Passed
I0904 19:25:18.771033 20857 XdpTester.cpp:167] Test: QUIC: short header. No connection id. CH. LRU hit result: Passed
I0904 19:25:18.771068 20857 XdpTester.cpp:167] Test: QUIC: short header w/ connection id result: Passed
I0904 19:25:18.771100 20857 XdpTester.cpp:167] Test: QUIC: short header w/ connection id but non-existing mapping result: Passed
I0904 19:25:18.771131 20857 XdpTester.cpp:167] Test: QUIC: short header w/ conn id. host id = 0. CH. LRU hit result: Passed

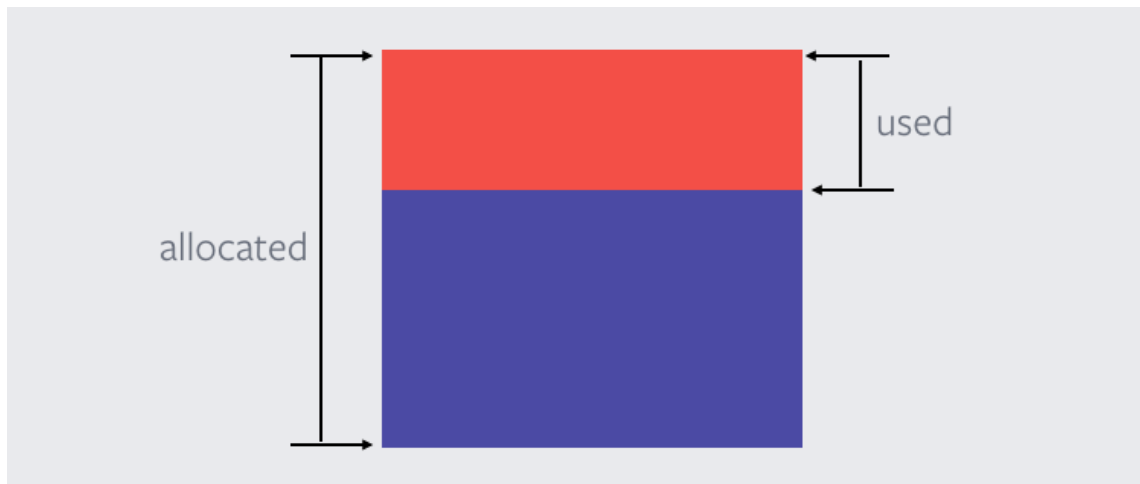
```

Picture 6: example of output of unittesting framework, based on BPF\_PROG\_TEST\_RUN.

## EVOLUTION

After initial version of XDP load balancer was introduced, few major performance related changes and additional features were introduced.

LRU map with BPF\_F\_NO\_COMMON\_LRU flag works good in scenario when there could be multiple writers, but it could be significantly inefficient in terms of memory consumption in environment when only subset of CPUs are responsible for IRQ handling (e.g. number of RX queues in a NIC is less then number of CPUs on the host). This is because LRU w/ BPF\_F\_NO\_COMMON\_LRU flag is divided by multiple zones, and each CPU updates only one zone, however because only subset of CPU are doing IRQ handling, most of the map's zones are not used at all. See Picture 7. for visual representation of the issue. In our environment we have seen up to 70% of allocated memory was unused because of this.



Picture 7: LRU's working are vs total allocated

To fight this inefficiency BPF's "map-in-map" was introduced[12]. Map in map is a special BPF's map (either "hash of maps" or "array of maps") where value is a descriptor of another BPF's map. In our environment we are using one to one mapping between NIC's IRQ and CPU, so we know in advance, before starting BPF's program, which CPUs are

going to be responsible for traffic handling. This knowledge allow us to create “map-in-map” where key is a CPU id and value is a descriptor of LRU map for this CPU. see Codeblock 4: for the usage example.

```
__u32 cpu_num = bpf_get_smp_processor_id();
void *lru_map = bpf_map_lookup_elem(&lru_maps_mapping, &cpu_num);
```

Codeblock 4: example of “map in map” usage.

With this approach we stopped to waste memory and as a side effect this also increased performance of our load balancer, as for the lookups the size of the memory became smaller (CPU started to do lookups only in dedicated map, which is smaller compare to what we have used before) and more cache friendly.

BPF's map allocation had a specific limitation, that on the host with multiple CPUs/multiple NUMA domains, it would allocate maps on the same NUMA node as userspace program, which is creating them, is running. In our environment we have hosts with CPUs and IRQ affinity configured in a way, that each CPU has same amount of IRQs mapped to it. That lead to situation where CPUs, which are “remote” (from NUMA point of view) are seeing more stall cycles, when trying to access BPF's maps (because to access em, they need to cross QPI[13] link). To fight this, special NUMA hint was introduced during BPF's map creating [14]. This allow to specify on which NUMA domain specific map should be allocated, and did allow us to allocate LRUs (most frequently updated type of maps in our L4 load balancer) on the local NUMA node. We have saw huge performance boost (tens of percents) on CPUs, which were forced to do QPI traversal before.

Another interesting problem which we were facing is that XDP lacks support for fragmentation. The problem because of this limitation is that after adding additional header resulting packet could be bigger than interface MTU. One solution could be to increase MTU on every server to the value, bigger than default MTU size of the internet (1500 bytes), to make sure that we have additional head room for encapsulation even after we have received packet of maximum size, supported by the internet. Changing MTU could be feasible on our edge network (small clusters which are located all over the world. where the size of the cluster is multiple racks). However it would be much harder/takes more time in our datacenters, with millions of servers, where we do run our load balancer as well. To be able to deploy XDP there, we have introduced additional helper - bpf\_xdp\_adjust\_tail [15]. This helper allows to “shrink” the packet and is used to generate ICMP “packet too big, fragmentation needed”. This ICMP message would contain, as a payload, first N bytes of the packet, which triggered generation of it. To be able to do it from XDP context, you would need:

1. shrink the packet to the “first N bytes” by using bpf\_xdp\_adjust\_tail
2. add enough headroom for ICMP + additional IP headers

Example of the usage of this helper could be found in Codeblock 5.

```
static inline int send_icmp_too_big(struct xdp_md *xdp,
                                   bool is_ipv6, int pkt_size) {

    int offset = pkt_size;
    if (is_ipv6) {
        offset -= ICMP6_TOO_BIG_SIZE;
    } else {
        offset -= ICMP_TOO_BIG_SIZE;
    }
    if(bpf_xdp_adjust_tail(xdp, 0 - offset)) {
        return XDP_DROP;
    }
    if (is_ipv6) {
```

```
    return send_icmp6_too_big(xdp);
} else {
    return send_icmp4_too_big(xdp);
}
}
```

Codeblock 5: example of usage `bpf_xdp_adjust_tail` helper. packet is shrunk to `ICMP(6)_TOOBIG_SIZE` bytes

## NEXT STEPS

Today we have good amount of helpers and features in XDP layer or BPF in general, however lots of things are still missing. To name the few, it would be nice to have an API between XDP and NIC to allow checksum offloading. Today it's close to impossible to do any encapsulation with XDP which requires full packet checksumming (e.g. VXLAN or GUE). Other missing feature is to have a crypto helper, which would allow to run encryption/decryption routines, as today lots of new protocols built with privacy in mind, and for network middle box became crucial to be able to decrypt some small partition of packet for further processing or routing (e.g. QUIC's connection-id field could be encrypted in near future). The last but not least is to be able to do bounded loops inside BPF program. The need for them arise from the need to parse TLV based protocols, where offsets are not well defined (e.g. TCP options)

## REFERENCES

- [1] - DPDK: <https://www.dpdk.org/>
- [2] - netmap: <http://info.iet.unipi.it/~luigi/netmap/>
- [3] - Open Source Days 2017 XDP presentation: [https://people.netfilter.org/hawk/presentations/OpenSourceDays2017/XDP\\_DDoS\\_protecting\\_osd2017.pdf](https://people.netfilter.org/hawk/presentations/OpenSourceDays2017/XDP_DDoS_protecting_osd2017.pdf)
- [4] - "XDP in practice: integrating XDP in our DDoS mitigation pipeline" by Cloudflare: <https://netdevconf.org/2.1/session.html?bertin>
- [5] - "Layer 4 Load Balancing at Facebook": <https://atscaleconference.com/videos/networking-scale-2018-layer-4-load-balancing-at-facebook/>
- [6] - Katran: <https://github.com/facebookincubator/katran>
- [7] - L3 DSR w/ IP Tunneling: <http://www.linuxvirtualserver.org/VS-IPTunneling.html>
- [8] - `bpf_xdp_adjust_head` patch series: <https://patchwork.ozlabs.org/patch/702198/>
- [9] - `bpf_LRU_map` support patch series: <https://lwn.net/Articles/706318/>
- [10] - `xdpdump`: <https://github.com/facebookincubator/katran/tree/master/tools/xdpdump>
- [11] - `BPF_PROG_TEST_RUN` patch series: <https://patchwork.ozlabs.org/patch/745468/>
- [12] - `BPF_map-in-map` support patch series: <https://lwn.net/Articles/718182/>
- [13] - QPI: [https://en.wikipedia.org/wiki/Intel\\_QuickPath\\_Interconnect](https://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect)
- [14] - NUMA hints support during BPF maps creation: <https://patchwork.ozlabs.org/patch/803369/>
- [15] - `bpf_xdp_adjust_tail` patch series: <https://patchwork.ozlabs.org/cover/900109/>