AF_XDP

Overview

AF_XDP is an address family that is optimized for high performance packet processing.

This document assumes that the reader is familiar with BPF and XDP. If not, the Cilium project has an excellent reference guide at http://cilium.readthedocs.io/en/latest/bpf/.

Using the XDP_REDIRECT action from an XDP program, the program can redirect ingress frames to other XDP enabled netdevs, using the bpf_redirect_map() function. AF_XDP sockets enable the possibility for XDP programs to redirect frames to a memory buffer in a user-space application.

An AF_XDP socket (XSK) is created with the normal socket() syscall. Associated with each XSK are two rings: the RX ring and the TX ring. A socket can receive packets on the RX ring and it can send packets on the TX ring. These rings are registered and sized with the setsockopts XDP_RX_RING and XDP_TX_RING, respectively. It is mandatory to have at least one of these rings for each socket. An RX or TX descriptor ring points to a data buffer in a memory area called a UMEM. RX and TX can share the same UMEM so that a packet does not have to be copied between RX and TX. Moreover, if a packet needs to be kept for a while due to a possible retransmit, the descriptor that points to that packet can be changed to point to another and reused right away. This again avoids copying data.

The UMEM consists of a number of equally sized chunks. A descriptor in one of the rings references a frame by referencing its addr. The addr is simply an offset within the entire UMEM region. The user space allocates memory for this UMEM using whatever means it feels is most appropriate (malloc, mmap, huge pages, etc). This memory area is then registered with the kernel using the new setsockopt XDP_UMEM_REG. The UMEM also has two rings: the FILL ring and the COMPLETION ring. The FILL ring is used by the application to send down addr for the kernel to fill in with RX packet data. References to these frames will then appear in the RX ring once each packet has been received. The COMPLETION ring, on the other hand, contains frame addr that the kernel has transmitted completely and can now be used again by user space, for either TX or RX. Thus, the frame addrs appearing in the COMPLETION ring are addrs that were previously transmitted using the TX ring. In summary, the RX and FILL rings are used for the RX path and the TX and COMPLETION rings are used for the TX path.

The socket is then finally bound with a bind() call to a device and a specific queue id on that device, and it is not until bind is completed that traffic starts to flow.

The UMEM can be shared between processes, if desired. If a process wants to do this, it simply skips the registration of the UMEM and its corresponding two rings, sets the XDP_SHARED_UMEM flag in the bind call and submits the XSK of the process it would like to share UMEM with as well as its own newly created XSK socket. The new process will then receive frame addr references in its own RX ring that point to this shared UMEM. Note that since the ring structures are single-consumer / single-producer (for performance reasons), the new process has to create its own socket with associated RX and TX rings, since it cannot share this with the other process. This is also the reason that there is only one set of FILL and COMPLETION rings per UMEM. It is the responsibility of a single process to handle the UMEM.

How is then packets distributed from an XDP program to the XSKs? There is a BPF map called XSKMAP (or BPF_MAP_TYPE_XSKMAP in full). The user-space application can place an XSK at an arbitrary place in this map. The XDP program can then redirect a packet to a specific index in this map and at this point XDP validates that the XSK in that map was indeed bound to that device and ring number. If not, the packet is dropped. If the map is empty at that index, the packet is also dropped. This also means that it is currently mandatory to have an XDP program loaded (and one XSK in the XSKMAP) to be able to get any traffic to user space through the XSK.

AF_XDP can operate in two different modes: XDP_SKB and XDP_DRV. If the driver does not have support for XDP, or XDP_SKB is explicitly chosen when loading the XDP program, XDP_SKB mode is employed that uses SKBs together with the generic XDP support and copies out the data to user space. A fallback mode that works for any network device. On the other hand, if the driver has support for XDP, it will be used by the AF_XDP code to provide better performance, but there is still a copy of the data into user space.

Concepts

In order to use an AF_XDP socket, a number of associated objects need to be setup. These objects and their options are explained in the following sections.

For an overview on how AF_XDP works, you can also take a look at the Linux Plumbers paper from 2018 on the subject: http://vger.kernel.org/lpc_net2018_talks/lpc18_paper_af_xdp_perf-v2.pdf. Do NOT consult the paper from 2017 on "AF_PACKET v4", the first attempt at AF_XDP. Nearly everything changed since then. Jonathan Corbet has also written an excellent article on LWN, "Accelerating networking with AF_XDP". It can be found at https://lwn.net/Articles/750845/.

UMEM

UMEM is a region of virtual contiguous memory, divided into equal-sized frames. An UMEM is associated to a netdev and a specific

queue id of that netdev. It is created and configured (chunk size, headroom, start address and size) by using the XDP_UMEM_REG setsockopt system call. A UMEM is bound to a netdev and queue id, via the bind() system call.

An AF_XDP is socket linked to a single UMEM, but one UMEM can have multiple AF_XDP sockets. To share an UMEM created via one socket A, the next socket B can do this by setting the XDP_SHARED_UMEM flag in struct sockaddr_xdp member sxdp_flags, and passing the file descriptor of A to struct sockaddr_xdp member sxdp_shared_umem_fd.

The UMEM has two single-producer/single-consumer rings that are used to transfer ownership of UMEM frames between the kernel and the user-space application.

Rings

There are a four different kind of rings: FILL, COMPLETION, RX and TX. All rings are single-producer/single-consumer, so the user-space application need explicit synchronization of multiple processes/threads are reading/writing to them.

The UMEM uses two rings: FILL and COMPLETION. Each socket associated with the UMEM must have an RX queue, TX queue or both. Say, that there is a setup with four sockets (all doing TX and RX). Then there will be one FILL ring, one COMPLETION ring, four TX rings and four RX rings.

The rings are head(producer)/tail(consumer) based rings. A producer writes the data ring at the index pointed out by struct xdp_ring producer member, and increasing the producer index. A consumer reads the data ring at the index pointed out by struct xdp_ring consumer member, and increasing the consumer index.

The rings are configured and created via the _RING setsockopt system calls and mmapped to user-space using the appropriate offset to mmap() (XDP_PGOFF_RX_RING, XDP_PGOFF_TX_RING, XDP_UMEM_PGOFF_FILL_RING and XDP_UMEM_PGOFF_COMPLETION_RING).

The size of the rings need to be of size power of two.

UMEM Fill Ring

The FILL ring is used to transfer ownership of UMEM frames from user-space to kernel-space. The UMEM addrs are passed in the ring. As an example, if the UMEM is 64k and each chunk is 4k, then the UMEM has 16 chunks and can pass addrs between 0 and 64k.

Frames passed to the kernel are used for the ingress path (RX rings).

The user application produces UMEM addrs to this ring. Note that, if running the application with aligned chunk mode, the kernel will mask the incoming addr. E.g. for a chunk size of 2k, the log2(2048) LSB of the addr will be masked off, meaning that 2048, 2050 and 3000 refers to the same chunk. If the user application is run in the unaligned chunks mode, then the incoming addr will be left untouched.

UMEM Completion Ring

The COMPLETION Ring is used transfer ownership of UMEM frames from kernel-space to user-space. Just like the FILL ring, UMEM indices are used.

Frames passed from the kernel to user-space are frames that has been sent (TX ring) and can be used by user-space again.

The user application consumes UMEM addrs from this ring.

RX Ring

The RX ring is the receiving side of a socket. Each entry in the ring is a struct xdp_desc descriptor. The descriptor contains UMEM offset (addr) and the length of the data (len).

If no frames have been passed to kernel via the FILL ring, no descriptors will (or can) appear on the RX ring.

The user application consumes struct xdp_desc descriptors from this ring.

TX Ring

The TX ring is used to send frames. The struct xdp_desc descriptor is filled (index, length and offset) and passed into the ring.

To start the transfer a sendmsg() system call is required. This might be relaxed in the future.

The user application produces struct xdp_desc descriptors to this ring.

Libbpf

Libbpf is a helper library for eBPF and XDP that makes using these technologies a lot simpler. It also contains specific helper functions in tools/lib/bpf/xsk.h for facilitating the use of AF_XDP. It contains two types of functions: those that can be used to make the setup of AF_XDP socket easier and ones that can be used in the data plane to access the rings safely and quickly. To see an example on how to use this API, please take a look at the sample application in samples/bpf/xdpsock_usr.c which uses libbpf for both setup and data plane operations.

We recommend that you use this library unless you have become a power user. It will make your program a lot simpler.

XSKMAP / BPF_MAP_TYPE_XSKMAP

On XDP side there is a BPF map type BPF_MAP_TYPE_XSKMAP (XSKMAP) that is used in conjunction with bpf_redirect_map() to pass the ingress frame to a socket.

The user application inserts the socket into the map, via the bpf() system call.

Note that if an XDP program tries to redirect to a socket that does not match the queue configuration and netdev, the frame will be dropped. E.g. an AF_XDP socket is bound to netdev eth0 and queue 17. Only the XDP program executing for eth0 and queue 17 will successfully pass data to the socket. Please refer to the sample application (samples/bpf/) in for an example.

Configuration Flags and Socket Options

These are the various configuration flags that can be used to control and monitor the behavior of AF_XDP sockets.

XDP_COPY and XDP_ZEROCOPY bind flags

When you bind to a socket, the kernel will first try to use zero-copy copy. If zero-copy is not supported, it will fall back on using copy mode, i.e. copying all packets out to user space. But if you would like to force a certain mode, you can use the following flags. If you pass the XDP_COPY flag to the bind call, the kernel will force the socket into copy mode. If it cannot use copy mode, the bind call will fail with an error. Conversely, the XDP_ZEROCOPY flag will force the socket into zero-copy mode or fail.

XDP_SHARED_UMEM bind flag

This flag enables you to bind multiple sockets to the same UMEM. It works on the same queue id, between queue ids and between netdevs/devices. In this mode, each socket has their own RX and TX rings as usual, but you are going to have one or more FILL and COMPLETION ring pairs. You have to create one of these pairs per unique netdev and queue id tuple that you bind to.

Starting with the case were we would like to share a UMEM between sockets bound to the same netdev and queue id. The UMEM (tied to the fist socket created) will only have a single FILL ring and a single COMPLETION ring as there is only on unique netdev,queue_id tuple that we have bound to. To use this mode, create the first socket and bind it in the normal way. Create a second socket and create an RX and a TX ring, or at least one of them, but no FILL or COMPLETION rings as the ones from the first socket will be used. In the bind call, set he XDP_SHARED_UMEM option and provide the initial socket's fd in the sxdp_shared_umem_fd field. You can attach an arbitrary number of extra sockets this way.

What socket will then a packet arrive on? This is decided by the XDP program. Put all the sockets in the XSK_MAP and just indicate which index in the array you would like to send each packet to. A simple round-robin example of distributing packets is shown below:

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

#define MAX_SOCKS 16

struct {
    __uint(type, BPF_MAP_TYPE_XSKMAP);
    __uint(max_entries, MAX_SOCKS);
    __uint(key_size, sizeof(int));
    __uint(value_size, sizeof(int));
} xsks_map SEC(".maps");

static unsigned int rr;

SEC("xdp_sock") int xdp_sock_prog(struct xdp_md *ctx)
{
    rr = (rr + 1) & (MAX_SOCKS - 1);

    return bpf_redirect_map(&xsks_map, rr, XDP_DROP);
}
```

Note, that since there is only a single set of FILL and COMPLETION rings, and they are single producer, single consumer rings, you need to make sure that multiple processes or threads do not use these rings concurrently. There are no synchronization primitives in the libbpf code that protects multiple users at this point in time.

Libbpf uses this mode if you create more than one socket tied to the same UMEM. However, note that you need to supply the XSK_LIBBPF_FLAGS__INHIBIT_PROG_LOAD libbpf_flag with the xsk_socket__create calls and load your own XDP program as there is no built in one in libbpf that will route the traffic for you.

The second case is when you share a UMEM between sockets that are bound to different queue ids and/or netdevs. In this case you have to create one FILL ring and one COMPLETION ring for each unique netdev,queue_id pair. Let us say you want to create two sockets bound to two different queue ids on the same netdev. Create the first socket and bind it in the normal way. Create a second socket and create an RX and a TX ring, or at least one of them, and then one FILL and COMPLETION ring for this socket. Then in the bind call, set he XDP_SHARED_UMEM option and provide the initial socket's fd in the sxdp_shared_umem_fd field as you

registered the UMEM on that socket. These two sockets will now share one and the same UMEM.

There is no need to supply an XDP program like the one in the previous case where sockets were bound to the same queue id and device. Instead, use the NIC's packet steering capabilities to steer the packets to the right queue. In the previous example, there is only one queue shared among sockets, so the NIC cannot do this steering. It can only steer between queues.

In libbpf, you need to use the xsk_socket__create_shared() API as it takes a reference to a FILL ring and a COMPLETION ring that will be created for you and bound to the shared UMEM. You can use this function for all the sockets you create, or you can use it for the second and following ones and use xsk_socket__create() for the first one. Both methods yield the same result.

Note that a UMEM can be shared between sockets on the same queue id and device, as well as between queues on the same device and between devices at the same time.

XDP_USE_NEED_WAKEUP bind flag

This option adds support for a new flag called need_wakeup that is present in the FILL ring and the TX ring, the rings for which user space is a producer. When this option is set in the bind call, the need_wakeup flag will be set if the kernel needs to be explicitly woken up by a syscall to continue processing packets. If the flag is zero, no syscall is needed.

If the flag is set on the FILL ring, the application needs to call poll() to be able to continue to receive packets on the RX ring. This can happen, for example, when the kernel has detected that there are no more buffers on the FILL ring and no buffers left on the RX HW ring of the NIC. In this case, interrupts are turned off as the NIC cannot receive any packets (as there are no buffers to put them in), and the need_wakeup flag is set so that user space can put buffers on the FILL ring and then call poll() so that the kernel driver can put these buffers on the HW ring and start to receive packets.

If the flag is set for the TX ring, it means that the application needs to explicitly notify the kernel to send any packets put on the TX ring. This can be accomplished either by a poll() call, as in the RX path, or by calling sendto().

An example of how to use this flag can be found in samples/bpf/xdpsock_user.c. An example with the use of libbpf helpers would look like this for the TX path:

```
if (xsk_ring_prod__needs_wakeup(&my_tx_ring))
    sendto(xsk_socket__fd(xsk_handle), NULL, 0, MSG_DONTWAIT, NULL, 0);
```

I.e., only use the syscall if the flag is set.

We recommend that you always enable this mode as it usually leads to better performance especially if you run the application and the driver on the same core, but also if you use different cores for the application and the kernel driver, as it reduces the number of syscalls needed for the TX path.

XDP_{RX|TX|UMEM_FILL|UMEM_COMPLETION}_RING setsockopts

These setsockopts sets the number of descriptors that the RX, TX, FILL, and COMPLETION rings respectively should have. It is mandatory to set the size of at least one of the RX and TX rings. If you set both, you will be able to both receive and send traffic from your application, but if you only want to do one of them, you can save resources by only setting up one of them. Both the FILL ring and the COMPLETION ring are mandatory as you need to have a UMEM tied to your socket. But if the XDP_SHARED_UMEM flag is used, any socket after the first one does not have a UMEM and should in that case not have any FILL or COMPLETION rings created as the ones from the shared UMEM will be used. Note, that the rings are single-producer single-consumer, so do not try to access them from multiple processes at the same time. See the XDP_SHARED_UMEM section.

In libbpf, you can create Rx-only and Tx-only sockets by supplying NULL to the rx and tx arguments, respectively, to the xsk_socket__create function.

If you create a Tx-only socket, we recommend that you do not put any packets on the fill ring. If you do this, drivers might think you are going to receive something when you in fact will not, and this can negatively impact performance.

XDP_UMEM_REG setsockopt

This setsockopt registers a UMEM to a socket. This is the area that contain all the buffers that packet can recide in. The call takes a pointer to the beginning of this area and the size of it. Moreover, it also has parameter called chunk_size that is the size that the UMEM is divided into. It can only be 2K or 4K at the moment. If you have an UMEM area that is 128K and a chunk size of 2K, this means that you will be able to hold a maximum of 128K / 2K = 64 packets in your UMEM area and that your largest packet size can be 2K.

There is also an option to set the headroom of each single buffer in the UMEM. If you set this to N bytes, it means that the packet will start N bytes into the buffer leaving the first N bytes for the application to use. The final option is the flags field, but it will be dealt with in separate sections for each UMEM flag.

XDP_STATISTICS getsockopt

Gets drop statistics of a socket that can be useful for debug purposes. The supported statistics are shown below:

```
struct xdp_statistics {
    __u64 rx_dropped; /* Dropped for reasons other than invalid desc */
```

```
    __u64 rx_invalid_descs; /* Dropped due to invalid descriptor */
    __u64 tx_invalid_descs; /* Dropped due to invalid descriptor */
};
```

XDP_OPTIONS getsockopt

Gets options from an XDP socket. The only one supported so far is XDP_OPTIONS_ZEROCOPY which tells you if zero-copy is on or not.

Usage

In order to use AF_XDP sockets two parts are needed. The user-space application and the XDP program. For a complete setup and usage example, please refer to the sample application. The user-space side is xdpsock_user.c and the XDP side is part of libbpf.

The XDP code sample included in tools/lib/bpf/xsk.c is the following:

```
SEC("xdp_sock") int xdp_sock_prog(struct xdp_md *ctx)
{
    int index = ctx->rx_queue_index;

    // A set entry here means that the corresponding queue_id
    // has an active AF_XDP socket bound to it.
    if (bpf_map_lookup_elem(&xsks_map, &index))
        return bpf_redirect_map(&xsks_map, index, 0);

    return XDP_PASS;
}
```

A simple but not so performance ring dequeue and enqueue could look like this:

```
// struct xdp_rxtx_ring {
//     __u32 *producer;
//     __u32 *consumer;
//     struct xdp_desc *desc;
// };

// struct xdp_umem_ring {
//     __u32 *producer;
//     __u32 *consumer;
//     __u64 *desc;
// };

// typedef struct xdp_rxtx_ring RING;
// typedef struct xdp_umem_ring RING;

// typedef struct xdp_desc RING_TYPE;
// typedef __u64 RING_TYPE;

int dequeue_one(RING *ring, RING_TYPE *item)
{
    __u32 entries = *ring->producer - *ring->consumer;

    if (entries == 0)
        return -1;

    // read-barrier!

    *item = ring->desc[*ring->consumer & (RING_SIZE - 1)];
    (*ring->consumer)++;
    return 0;
}

int enqueue_one(RING *ring, const RING_TYPE *item)
{
    u32 free_entries = RING_SIZE - (*ring->producer - *ring->consumer);

    if (free_entries == 0)
        return -1;

    ring->desc[*ring->producer & (RING_SIZE - 1)] = *item;

    // write-barrier!

    (*ring->producer)++;
    return 0;
}
```

But please use the libbpf functions as they are optimized and ready to use. Will make your life easier.

Sample application

There is a xdpsock benchmarking/test application included that demonstrates how to use AF_XDP sockets with private UMEMs. Say that you would like your UDP traffic from port 4242 to end up in queue 16, that we will enable AF_XDP on. Here, we use ethtool for this:

```
ethtool -N p3p2 rx-flow-hash udp4 fn
ethtool -N p3p2 flow-type udp4 src-port 4242 dst-port 4242 \
    action 16
```

Running the rxdrop benchmark in XDP_DRV mode can then be done using:

```
samples/bpf/xdpsock -i p3p2 -q 16 -r -N
```

For XDP_SKB mode, use the switch "-S" instead of "-N" and all options can be displayed with "-h", as usual.

This sample application uses libbpf to make the setup and usage of AF_XDP simpler. If you want to know how the raw uapi of AF_XDP is really used to make something more advanced, take a look at the libbpf code in tools/lib/bpf/xsk.[ch].

FAQ

Q: I am not seeing any traffic on the socket. What am I doing wrong?

A: When a netdev of a physical NIC is initialized, Linux usually

> allocates one RX and TX queue pair per core. So on a 8 core system, queue ids 0 to 7 will be allocated, one per core. In the AF_XDP bind call or the xsk_socket__create libbpf function call, you specify a specific queue id to bind to and it is only the traffic towards that queue you are going to get on you socket. So in the example above, if you bind to queue 0, you are NOT going to get any traffic that is distributed to queues 1 through 7. If you are lucky, you will see the traffic, but usually it will end up on one of the queues you have not bound to.
>
> There are a number of ways to solve the problem of getting the traffic you want to the queue id you bound to. If you want to see all the traffic, you can force the netdev to only have 1 queue, queue id 0, and then bind to queue 0. You can use ethtool to do this:
>
> ```
> sudo ethtool -L <interface> combined 1
> ```
>
> If you want to only see part of the traffic, you can program the NIC through ethtool to filter out your traffic to a single queue id that you can bind your XDP socket to. Here is one example in which UDP traffic to and from port 4242 are sent to queue 2:
>
> ```
> sudo ethtool -N <interface> rx-flow-hash udp4 fn
> sudo ethtool -N <interface> flow-type udp4 src-port 4242 dst-port \
> 4242 action 2
> ```
>
> A number of other ways are possible all up to the capabilities of the NIC you have.

Q: Can I use the XSKMAP to implement a switch betwen different umems

> in copy mode?

A: The short answer is no, that is not supported at the moment. The

> XSKMAP can only be used to switch traffic coming in on queue id X to sockets bound to the same queue id X. The XSKMAP can contain sockets bound to different queue ids, for example X and Y, but only traffic goming in from queue id Y can be directed to sockets bound to the same queue id Y. In zero-copy mode, you should use the switch, or other distribution mechanism, in your NIC to direct traffic to the correct queue id and socket.

Q: My packets are sometimes corrupted. What is wrong?

A: Care has to be taken not to feed the same buffer in the UMEM into

> more than one ring at the same time. If you for example feed the same buffer into the FILL ring and the TX ring at the same time, the NIC might receive data into the buffer at the same time it is sending it. This will cause some packets to become corrupted. Same thing goes for feeding the same buffer into the FILL rings belonging to different queue ids or netdevs bound with the XDP_SHARED_UMEM flag.

Credits

- BjÃ¶rn TÃ¶pel (AF_XDP core)
- Magnus Karlsson (AF_XDP core)
- Alexander Duyck
- Alexei Starovoitov
- Daniel Borkmann
- Jesper Dangaard Brouer

- John Fastabend
- Jonathan Corbet (LWN coverage)
- Michael S. Tsirkin
- Qi Z Zhang
- Willem de Bruijn