XDP Inside and Out

David S. Miller

Overview

XDP vs. userland frameworks

The checklists

The good, the bad, and the ugly

Using XDP ideas elsewhere

Userland Protocol Stacks

Library, SDK, or other kind of networking framework in userspace

Sits on top of a device access facility such as DPDK

Completely bypasses the kernel

Sits in it’s own universe

Completely segregated from the kernel

Why bother?

Performance, but much of this argument is marginal

“Safety”

Developing kernel code is “cray cray”, one typo crashes entire system

There are lots of userland programmers

Rebooting the kernel is disruptive

Rebooting the kernel causes traffic loss

On a Sunny Day at Lawrence Berkeley Labs…

The first step of a very long journey was taken

Van Jacobson and Steven McCane saw that at least one part of the kernel should be fully programmable

And this, of course, led to the Berkeley Packet Filter or BPF

Limited in scope to sockets, and mainly used for packet sniffing applications, this incredible virus sat dormant for 24 years before spreading further

Programmable Policy?

This is not about changing the kernel

This is about policy driven decision making

What does the user want (filter out these packets)

What does the admin want (drop packets from X)

This is all opt-in

If you want it great, and if you don’t, that’s fine too

Traditional Policy Support

Long lead times until deployment. Lots of cruft…

Without programmable policies, the kernel is in limbo forever

New ABIs constantly being added to the kernel

Because we cannot predict future policy needs

Old ABIs fall into disuse, and can’t be removed

Programmable policy ends this cycle for good

BPF Was Not Ready to Save the World

Not powerful enough yet

Simple 32-bit byte code engine

2 registers, A and X

Small stack area for local storage

Harnessing the full power of programmable policy requires something closer to a real CPU

Hence, eBPF...

Brainchild of Alexei Starovoitov, landing in the 3.18 kernel

Full 64-bit engine, a dozen or so registers

Comprehensive instruction set with atomic operations, etc.

Normal C code can be compiled to generate eBPF programs

But wait, there’s more...

eBPF Has Real Data Structures

We call these eBPF “MAPS”

Several types of MAPS exist, more can be added:

- Array
- Hash Table
- LPM Trie

Maps are really friggin’ important because....

eBPF Programs Must Be Simple

Must execute in short finite amount of time

No back branches allowed, and none needed due to MAPS

Memory accesses must be strictly controlled (f.e. Socket filters can only access packet data and metadata)

Remember: This is about implementing policy and only very simple operations

Enter XDP

eXpress Data Path
Run eBPF programs at the earliest place possible in the stack
Exactly when the device driver takes the packet from the RX ring
XDP eBPF program returns a verdict:

    DROP, PASS, TX, ABORT

XDP datapath lives in full harmony with rest of kernel networking stack

XDP Applications

DDoS protection using “bad IPs” list in the form of an eBPF MAP

- If XDP sees a packet with IP in this list, return XDP_DROP

More sophisticated DDoS protection

- eBPF program looks for “patterns” perhaps using ancillary data in a MAP

Load balancing via XDP_TX verdict

Switching, Routing, Tunnel termination...

Why XDP?

XDP can co-operate with the socket layer
It can filter packets which otherwise would hit the applications
It can therefore protect applications
XDP can perform well in east-west VM to VM server use cases
XDP can be pushed out from the VM itself to the bare metal host
This avoids the overhead of pushing harmful traffic into the VM itself

The Checklist, XDP vs. Userspace Stacks

Performance

“Safety”

Typos take the entire system down

Developer pervasiveness

Kernel reboot is disruptive

Traffic loss

Arduino as a BPF Metaphor

The development process for both are similar

Special development environment and tools

Simple programs with well defined entry points

Programs are “pushed” to the execution environment

Crashing Arduino doesn’t crash the laptop

Arduino is well confined “black box”

Pushing Metadata’less SKBs

Early parts of the stack don’t need a full sk_buff

The question is “how much” of the early parts of the stack

If it’s deep enough, performance gain might be worth it

Prefetching sk_buff alloc?

The issue of parallel code paths, which are inevitable

Long term maintenance

Thank You

Alexei Starovoitov and Tom Herbert

Linus Torvalds

Van Jacobson

Jesper Dangaard Brouer

Thomas Graf

Daniel Borkmann