

Cloudflare architecture and how BPF eats the world

Marek Majkowski

Recently at [Netdev 0x13](#), the Conference on Linux Networking in Prague, I gave [a short talk titled "Linux at Cloudflare"](#). The [talk](#) ended up being mostly about BPF. It seems, no matter the question - BPF is the answer.

Here is a transcript of a slightly adjusted version of that talk.

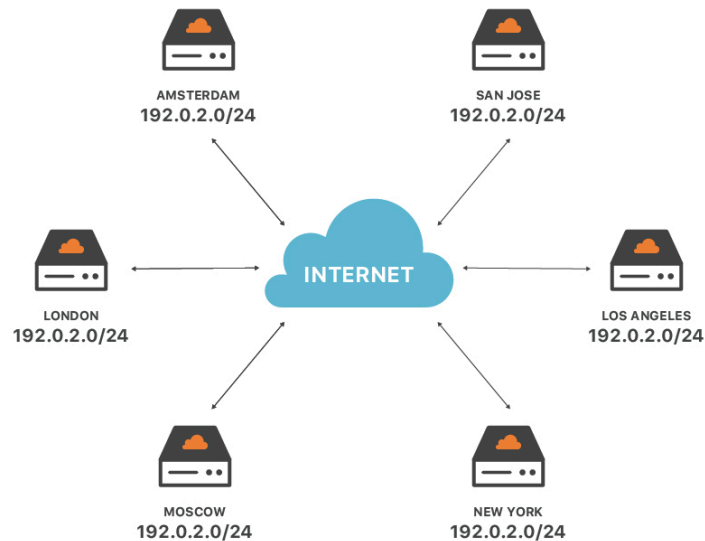
Edge Network – locations



At Cloudflare we run Linux on our servers. We operate two categories of data centers: large "Core" data centers, processing logs, analyzing attacks, computing analytics, and the "Edge" server fleet, delivering customer content from 180 locations across the world.

In this talk, we will focus on the "Edge" servers. It's here where we use the newest Linux features, optimize for performance and care deeply about DoS resilience.

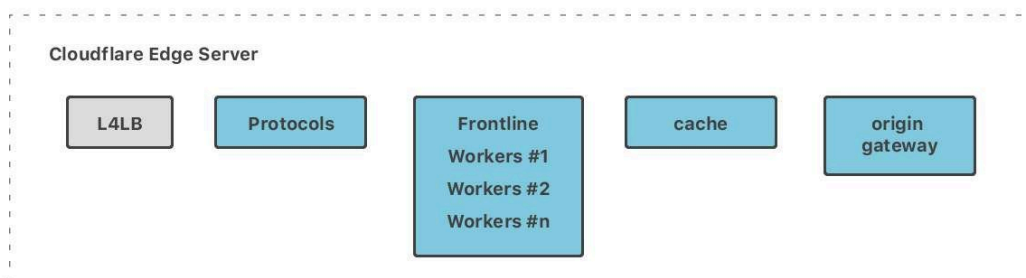
Edge Network – anycast



Our edge service is special due to our network configuration - we are extensively using anycast routing. Anycast means that the same set of IP addresses are announced by all our data centers.

This design has great advantages. First, it guarantees the optimal speed for end users. No matter where you are located, you will always reach the closest data center. Then, anycast helps us to spread out DoS traffic. During attacks each of the locations receives a small fraction of the total traffic, making it easier to ingest and filter out unwanted traffic.

Edge Network – uniform software



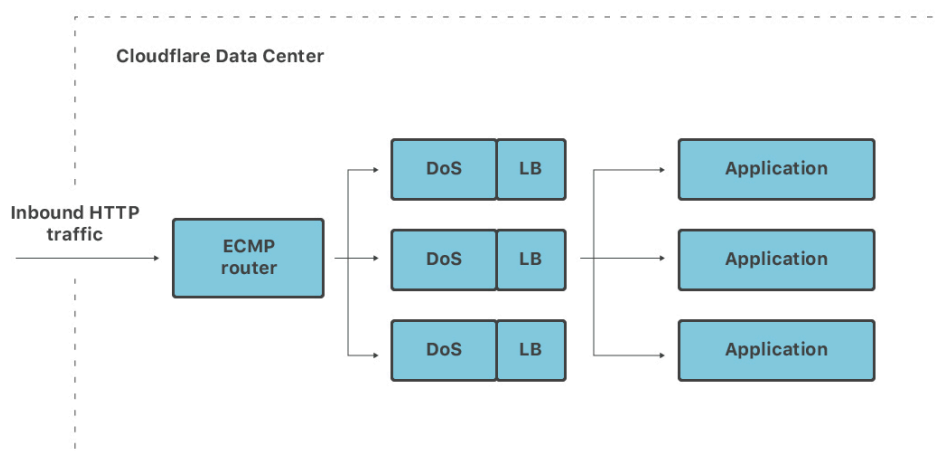
Anycast allows us to keep the networking setup uniform across all edge data centers. We applied the same design inside our data centers - our

software stack is uniform across the edge servers. All software pieces are running on all the servers.

In principle, every machine can handle every task - and we run many diverse and demanding tasks. We have a full HTTP stack, the magical [Cloudflare Workers](#), two sets of DNS servers - authoritative and resolver, and many other publicly facing applications like [Spectrum](#) and [Warp](#).

Even though every server has all the software running, requests typically cross many machines on their journey through the stack. For example, an HTTP request might be handled by a different machine during each of the 5 stages of the processing.

Life of a request



Let me walk you through the early stages of inbound packet processing:

(1) First, the packets hit our router. The router does ECMP, and forwards packets onto our Linux servers. We use ECMP to spread each target IP across many, at least 16, machines. This is used as a rudimentary load balancing technique.

(2) On the servers we ingest packets with XDP eBPF. In XDP we perform two stages. First, we run volumetric [DoS mitigations](#), dropping packets belonging to very large layer 3 attacks.

(3) Then, still in XDP, we perform layer 4 [load balancing](#). All the non-attack packets are redirected across the machines. This is used to work around

the ECMP problems, gives us fine-granularity load balancing and allows us to gracefully take servers out of service.

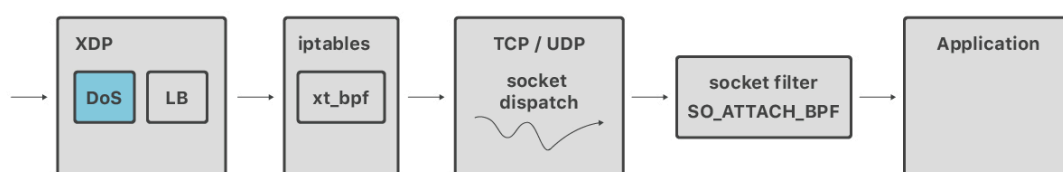
(4) Following the redirection the packets reach a designated machine. At this point they are ingested by the normal Linux networking stack, go through the usual iptables firewall, and are dispatched to an appropriate network socket.

(5) Finally packets are received by an application. For example HTTP connections are handled by a "protocol" server, responsible for performing TLS encryption and processing HTTP, HTTP/2 and QUIC protocols.

It's in these early phases of request processing where we use the coolest new Linux features. We can group useful modern functionalities into three categories:

- DoS handling
 - Load balancing
 - Socket dispatch
-

DDoS mitigation – XDP



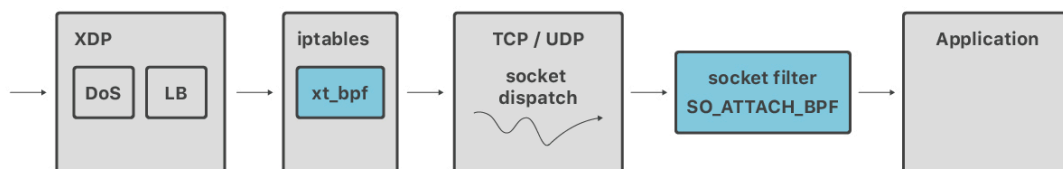
Let's discuss DoS handling in more detail. As mentioned earlier, the first step after ECMP routing is Linux's XDP stack where, among other things, we run DoS mitigations.

Historically our mitigations for volumetric attacks were expressed in classic BPF and iptables-style grammar. Recently we adapted them to execute in the XDP eBPF context, which turned out to be surprisingly hard. Read on about our adventures:

- [L4Drop: XDP DDoS Mitigations](#)
- [xdpcap: XDP Packet Capture](#)
- [XDP based DoS mitigation](#) talk by Arthur Fabre
- [XDP in practice: integrating XDP into our DDoS mitigation pipeline](#) (PDF)

During this project we encountered a number of eBPF/XDP limitations. One of them was the lack of concurrency primitives. It was very hard to implement things like race-free token buckets. Later we found that [Facebook engineer Julia Kartseva](#) had the same issues. In February this problem has been addressed with the introduction of `bpf_spin_lock` helper.

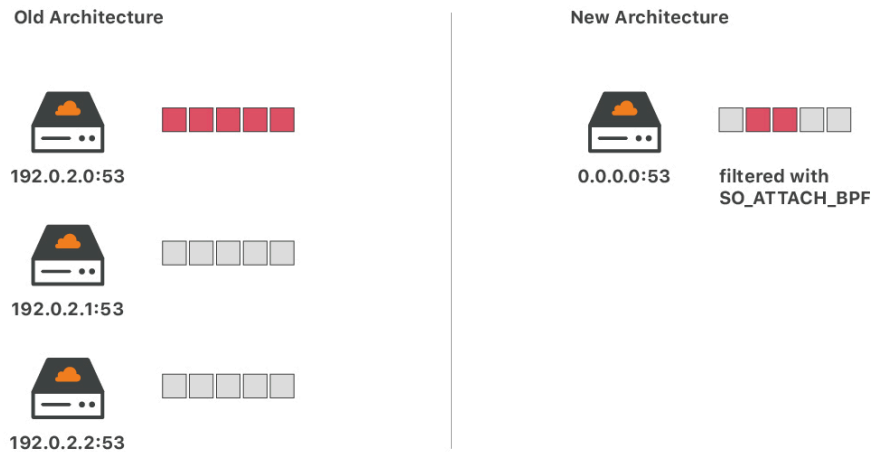
DDoS mitigation – xt_bpf, SO_ATTACH_BPF



While our modern volumetric DoS defenses are done in XDP layer, we still rely on iptables for application layer 7 mitigations. Here, a higher level firewall's features are useful: connlimit, hashlimits and ipsets. We also use the `xt_bpf` iptables module to run cBPF in iptables to match on packet payloads. We talked about this in the past:

- [Lessons from defending the indefensible](#) (PPT)
 - [Introducing the BPF tools](#)
-

SO_ATTACH_BPF on sockets



After XDP and iptables, we have one final kernel side DoS defense layer.

Consider a situation when our UDP mitigations fail. In such case we might be left with a flood of packets hitting our application UDP socket. This might overflow the socket causing packet loss. This is problematic - both good and bad packets will be dropped indiscriminately. For applications like DNS it's catastrophic. In the past to reduce the harm, we ran one UDP socket per IP address. An unmitigated flood was bad, but at least it didn't affect the traffic to other server IP addresses.

Nowadays that architecture is no longer suitable. We are running more than 30,000 DNS IP's and running that number of UDP sockets is not optimal. Our modern solution is to run a single UDP socket with a complex eBPF socket filter on it - using the `SO_ATTACH_BPF` socket option. We talked about running eBPF on network sockets in past blog posts:

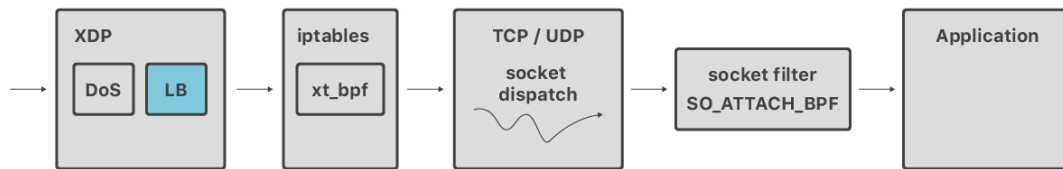
- [eBPF, Sockets, Hop Distance and manually writing eBPF assembly](#)
- [SOCKMAP - TCP splicing of the future](#)

The mentioned eBPF rate limits the packets. It keeps the state - packet counts - in an eBPF map. We can be sure that a single flooded IP won't affect other traffic. This works well, though during work on this project we found a rather worrying bug in the eBPF verifier:

- [eBPF can't count?!](#)

I guess running eBPF on a UDP socket is not a common thing to do.

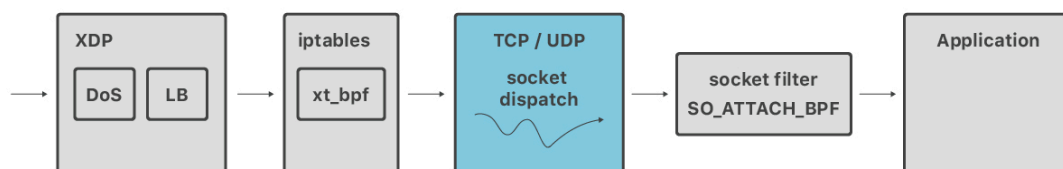
Load balancing



Apart from the DoS, in XDP we also run a layer 4 load balancer layer. This is a new project, and we haven't talked much about it yet. Without getting into many details: in certain situations we need to perform a socket lookup from XDP.

The problem is relatively simple - our code needs to look up the "socket" kernel structure for a 5-tuple extracted from a packet. This is generally easy - there is a `bpf_sk_lookup` helper available for this. Unsurprisingly, there were some complications. One problem was the inability to verify if a received ACK packet was a valid part of a three-way handshake when SYN-cookies are enabled. My colleague Lorenz Bauer is working on adding support for this corner case.

TCP / UDP Socket dispatch



After DoS and the load balancing layers, the packets are passed onto the usual Linux TCP / UDP stack. Here we do a socket dispatch - for example packets going to port 53 are passed onto a socket belonging to our DNS server.

We do our best to use vanilla Linux features, but things get complex when you use thousands of IP addresses on the servers.

Convincing Linux to route packets correctly is relatively easy with [the "AnyIP" trick](#). Ensuring packets are dispatched to the right application is another matter. Unfortunately, standard Linux socket dispatch logic is not flexible enough for our needs. For popular ports like TCP/80 we want to share the port between multiple applications, each handling it on a different IP range. Linux doesn't support this out of the box. You can call `bind()` either on a specific IP address or all IP's (with 0.0.0.0).

Socket dispatch

	Single IP	All IP's	Selected subnets
Single port	<code>bind (192.0.2.1)</code>	<code>bind (0.0.0.0)</code>	<code>bind (0.0.0.0) + SO_BINDTOPREFIX</code>
Many ports	TPROXY	TPROXY	TPROXY

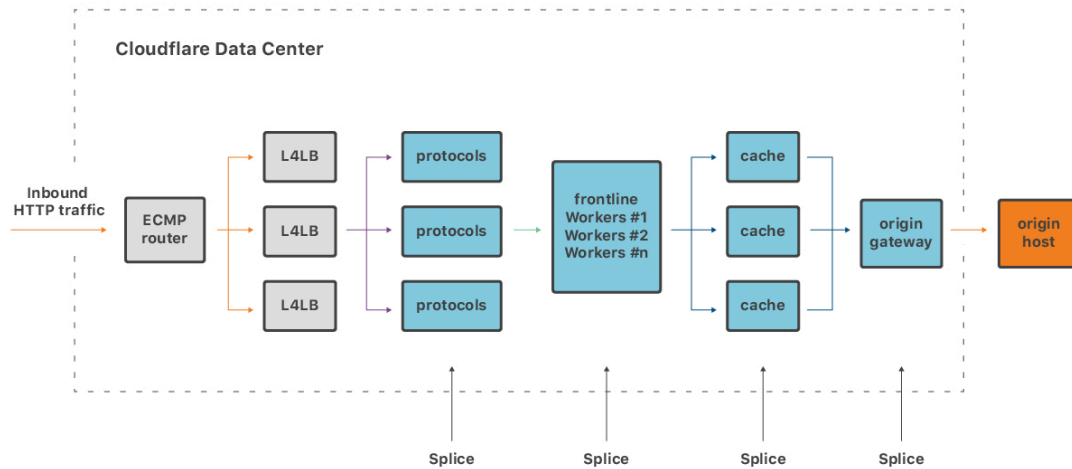
In order to fix this, we developed a custom kernel patch which adds [a SO_BINDTOPREFIX socket option](#). As the name suggests - it allows us to call `bind()` on a selected IP prefix. This solves the problem of multiple applications sharing popular ports like 53 or 80.

Then we run into another problem. For our Spectrum product we need to listen on all 65535 ports. Running so many listen sockets is not a good idea (see [our old war story blog](#)), so we had to find another way. After some experiments we learned to utilize an obscure iptables module - TPROXY - for this purpose. Read about it here:

- [Abusing Linux's firewall: the hack that allowed us to build Spectrum](#)

This setup is working, but we don't like the extra firewall rules. We are working on solving this problem correctly - actually extending the socket dispatch logic. You guessed it - we want to extend socket dispatch logic by utilizing eBPF. Expect some patches from us.

SOCKMAP for TCP splicing



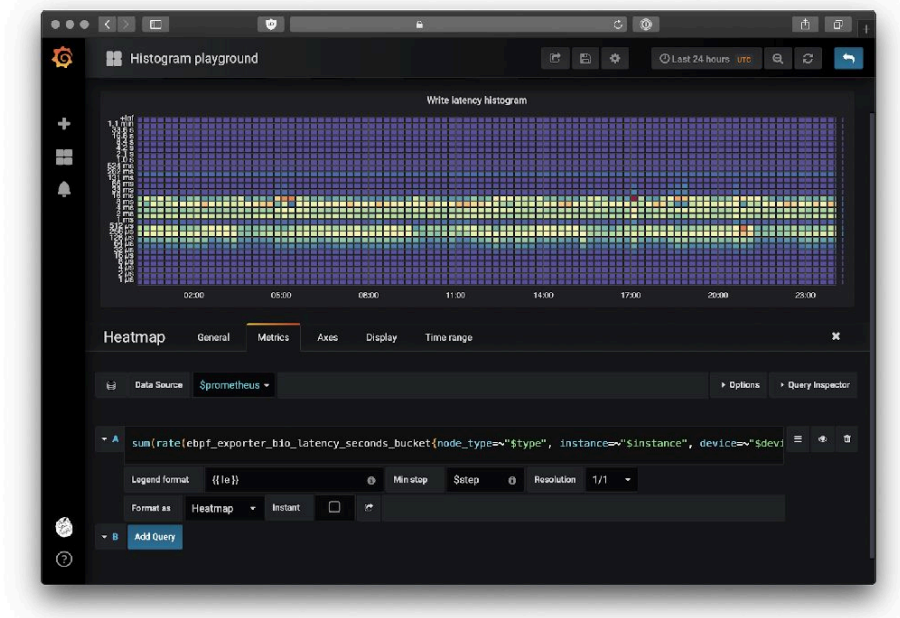
Then there is a way to use eBPF to improve applications. Recently we got excited about doing TCP splicing with SOCKMAP:

- [SOCKMAP - TCP splicing of the future](#)

This technique has a great potential for improving tail latency across many pieces of our software stack. The current SOCKMAP implementation is not quite ready for prime time yet, but the potential is vast.

Similarly, the new [TCP-BPF aka BPF_SOCK_OPS](#) hooks provide a great way of inspecting performance parameters of TCP flows. This functionality is super useful for our performance team.

Prometheus – ebpf_exporter



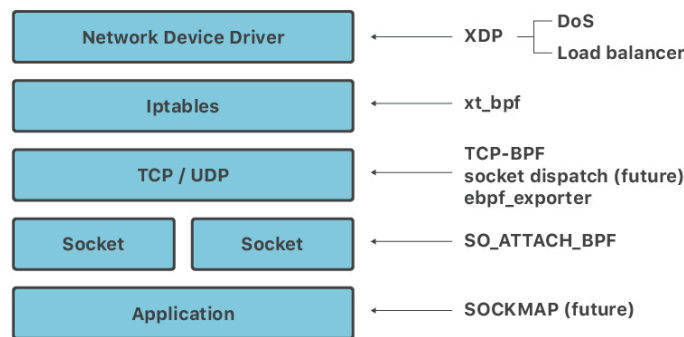
Some Linux features didn't age well and we need to work around them. For example, we are hitting limitations of networking metrics. Don't get me wrong - the networking metrics are awesome, but sadly they are not granular enough. Things like `TcpExtListenDrops` and `TcpExtListenOverflows` are reported as global counters, while we need to know it on a per-application basis.

Our solution is to use eBPF probes to extract the numbers directly from the kernel. My colleague Ivan Babrou wrote a Prometheus metrics exporter called "ebpf_exporter" to facilitate this. Read on:

- [Introducing ebpf_exporter](#)
- https://github.com/cloudflare/ebpf_exporter

With "ebpf_exporter" we can generate all manner of detailed metrics. It is very powerful and saved us on many occasions.

BPF is everywhere



In this talk we discussed 6 layers of BPFs running on our edge servers:

- Volumetric DoS mitigations are running on XDP eBPF
- Iptables xt_bpf cBPF for application-layer attacks
- SO_ATTACH_BPF for rate limits on UDP sockets
- Load balancer, running on XDP
- eBPFs running application helpers like SOCKMAP for TCP socket splicing, and TCP-BPF for TCP measurements
- "ebpf_exporter" for granular metrics

And we're just getting started! Soon we will be doing more with eBPF based socket dispatch, eBPF running on [Linux TC \(Traffic Control\)](#) layer and more integration with cgroup eBPF hooks. Then, our SRE team is maintaining ever-growing list of [BCC scripts](#) useful for debugging.

It feels like Linux stopped developing new API's and all the new features are implemented as eBPF hooks and helpers. This is fine and it has strong advantages. It's easier and safer to upgrade eBPF program than having to recompile a kernel module. Some things like TCP-BPF, exposing high-volume performance tracing data, would probably be impossible without eBPF.

Some say "software is eating the world", I would say that: "BPF is eating the software".

