



Listen backlog and the growth of kernel decision points

Linux Plumbers 2026

Alan Maguire

Linux Kernel Networking, Oracle

October 2026



Topic

- I'd like to discuss adding a new decision point into the kernel for sock_ops; but also like to raise the issue of decision points in Linux in general
- We are adding more and more decision points into the kernel
- We've always had some flavours of these
 - CONFIG options at compile time
 - Boot options
 - Sysctls at runtime
- BPF is adding more and more all the time
- Are there mechanisms/best practices we could use?

The problem

- TCP connections are managed in the kernel via two queues:
 - the SYN queue: where SYN state awaiting the final ACK in the three-way handshake are represented
 - the accept queue: where connections that have completed the three-way handshake are represented prior to the application calling `accept()`
- The accept queue length is specified by the `listen()` system call, and the eventual length is the minimum of the value specified and the `sysctl` value `net.core.somaxconn` (default 4096).
- If the acceptq is full, further SYNs are silently dropped to allow it to drain.



The problem

There are two problems with respect to this setting at any given time:

- If the `listen()` backlog is set too low, the accept queue will fill up and new incoming SYNs will be dropped, reducing the ability for the service to handle requests
- If the `listen()` backlog is set too high, the service may not be able to keep up with the number of connections that need to be `accept()`ed. Pending connections may time out, but on the order of minutes since as far as they are concerned the connection is established.

The general advice to deal with the first problem is to set the listen backlog sufficiently high in order to ensure that the service can deal with maximum connection rate spike. However the danger in setting this value too high is that we make the second situation worse.



Real-world example

After power outage, “thundering herd” of booting clients was observed in data center

- Placed pressure on iSCSI server
- With so many expensive CPU/IO-bound operations (iSCSI connection setup, disk access latency) service was limited in number of connections it could handle.
- With a long `acceptq` (which at times was the right answer) many connections were established at the TCP layer, but waiting for `app accept()`s.
- But to clients, the connections seemed established, so they took multiple minutes to time out.



The goal

- At times, we want to drop SYNs silently to allow services to recover
 - SYN retransmit will occur using exponential backoff at 1, 2, 4, 8, ... sec
 - 6 retries
- Gives service breathing space to recover
- This is much better than giving clients the appearance of connection establishment
- In that case, as we saw, it can take multiple minutes for connections to time out
- If a service is maxed out, it would be good to be able to dynamically “pretend” the queue is full and drop further SYNs



One approach

- An initial approach we explored was using an `sk_lookup` program in tandem with `sock_ops`
- `sock_ops` records metrics like acceptq length, connection rate etc
- When we exceed limits, `SK_DROP` the connection
- Good: it is per-connection, not per-packet like a `tc-bpf` program
- Bad:
 - an `SK_DROP` results in a RST to the client; we would prefer a silent SYN drop
 - `sock_ops` are bound to cgroup, `sk_lookup` to network namespace
- Hack: instead of `SK_DROP`, redirect to a blackhole socket (create a dummy device with no connectivity and use `setsockopt(.., SO_BINDTODEVICE)` to ensure our blackhole socket will not send SYN|ACK



Another possibility

- Add a new sock ops hook called if flag is enabled at points `sk_acceptq_full()` is called
 - `BPF_SOCK_OPS_QUEUE_FULL`
 - Parameters: `sk`
 - Returns: 1 to simulate queue full, 0 otherwise
 - Does not override “real” queue full



Zooming out

- The number of decision points we are adding to Linux keeps growing
- BPF solutions now can customize most aspects of Linux now
 - Networking
 - Scheduling
 - Memory management
 - I/O
 - ...
- Are there ways we should approach these to ensure
 - They signal intent clearly such that the system can be understood
 - They play nicely together



Growth of BPF-based decision points

- Most systems have dozens/hundreds of BPF programs running on them
- How do we reason about these highly customized systems?
 - Observability plays a critical role – what happened?
 - But so does declarative logging – I did X for reason Y
 - At least then people and AI systems have some transparency into both the what and the why
 - Zooming out that allows wider across-systems observations, push down intent to local services
 - Or understand what your laptop is doing
 - Can we distill some best practices?



bpftune approach

- Auto-tunes (mostly network) tunables, congestion control, etc via BPF.
- BPF is used
 - As a means to observe the system at a fine grain
 - As a means to alter the system at a fine grain
 - As a means to *observe the effect* of altering system at a fine grain
 - Philosophy: “bend the curve” of trade-offs, in part by monitoring effects
 - e.g. when do throughput-related changes (socket buffer size increases) start to correlate with latencies?
- Key design decision: Log every change with a clear explanation
 - What changed?
 - Why?
 - Allow rollback



bpftune approaches: simple machine learning

- Examples:
 - Congestion control algorithm choice via reinforcement learning
 - Aggregates cost metrics (based around BDP estimation) to evaluate algorithms across connections
 - Choose among algorithms based on this
 - All done in-kernel
 - Algorithm selection on connection setup
 - Cost update on connection close
 - Per netns TCP buffer tuning via sysctl
 - If we see “approaching full”, conditionally increase buffer size
 - While monitoring correlation between it and RTT
 - Kernel collects most stats for correlation, userspace does the final computation/sysctl setting



What bpftune got right

- Logging of decisions (via syslog)
- Plays nicely with others
 - Watches for sysctl setting in the domains where it auto-tunes
 - If not originated by bpftune, disables its auto-tuning in that domain
- Flexible: supports multiple policies for a domain, switching between them based on policy evaluation



What is still difficult

- System is a moving target based on active auto-tuning
- Checking sysctls at any one time just gives a snapshot
- No sense of system dynamics given policies in place



The broader problem

- As more policy decisions move into BPF programs, can we think of ways of making them less opaque?
- Something as simple as a “ospolicy” tracepoint (inspired by “osnoise”) might help
 - Structured to provide
 - A domain area (net, mm etc)
 - A subdomain
 - A what (the event)
 - And a why
 - Could be triggered via bpf progs that make policy changes via a bpf kfunc
 - Unified across key kernel policy choices, bpf-driven policy choices



Summary

- We are seeing BPF applied to more and more kernel subsystems
- This can make understanding system behaviour even more challenging
- Can we think of scalable approaches that better facilitate this?



References

- <https://netdevconf.info/2.2/papers/brakmo-tcpbpf-talk.pdf>
- https://docs.kernel.org/bpf/prog_sk_lookup.html
- <https://blog.cloudflare.com/syn-packet-handling-in-the-wild/>
- https://www.alibabacloud.com/blog/tcp-syn-queue-and-accept-queue-overflow-explained_599203
- <https://lore.kernel.org/bpf/20261001184551.314c7a0e@kernel.org/>
- <https://blogs.oracle.com/linux/bpftune-using-reinforcement-learning-in-bpf>