The Case for Moving Congestion Control Out of the Datapath

HotNets-XVI Dialogue

Thomas Anderson and P. Brighten Godfrey

BG: This paper argues for moving CC into userspace, with the kernel or hardware responsible for the datapath and userspace responsible for policy.

TA: One of the reviewers pointed out that Linux has a kernel module that allows for customization of the TCP CC policy, so is this adding something essential?

BG: Google's QUIC framework may provide a useful case study. Evolving new congestion control algorithms was a big part of the motivation of QUIC; if it had been convenient to use an existing interface to configure congestion control, then that would have been easier rather than building a new user-space, performance-limited stack.

TA: There's also the problem of portability. QUIC needed to run everywhere, not just on Linux.

BG: So there's reason to believe this paper is tapping into a real need for more flexibility. But will CCP's approach help?

TA: I was reminded of Hydra, a system built back when I was in middle school. The idea was to structure the OS so that mechanism was in the kernel but policy was at user level. Much of the rationale given in this paper would be familiar to the Hydra designers. You can see the principle being applied in various domains—Mach user-level pagers being a prime example.

BG: How did that approach pan out over the years and decades?

TA: It is an idea that resurfaces from time to time. In the networking arena, we increasingly have the separation of control and data planes, but mainly that's to make configuration easier. In OSes, mechanism/policy separation has been less successful. That doesn't mean it can't work here—but we should understand the barriers. One of the hardest is getting the interface right: the mechanism defines the policy knobs. Another is the performance of the control loop.

BG: In the congestion control domain, is there an API into the congestion control mechanism (implemented in the kernel) that's sufficiently general?

TA: There's a claim in the paper that the proposed API is generic—and they give a set of example algorithms that would work with their system. But then I thought about my one foray into TCP design, PCP, which sends packets in a burst and measures the interarrival times between packets. It is based on packet pair—what we would all do if the network implemented fair queueing. The paper doesn't handle that.

BG: Here's another example to test generality of the API: there are a couple proposals to send packets multiple times to mask loss or utilize additional bandwidth, and in particular to send duplicate packets in reverse order, so that packets that are most likely to have been lost are resent first (before the sender knows explicitly which packets are lost). That can be useful when even one RTT is too long to wait.

TA: In their design, CCP's policy decisions are asynchronous. For example, you get an RTT worth of measurements and then make a decision. But while you are thinking, you are collecting another RTT—the decision and the next round are happening simultaneously. What happens to those measurements? If the RTT is long enough, ok, but APIs that can't be implemented efficiently don't have a long shelf life. Inside

the data center, with 10 or 100 μsec RTTs, I think you'd almost need to dedicate a core to running the CC loop.

BG: Certainly the space of possible congestion control mechanisms is pretty large and can step outside the scope of CCP's currently-proposed API. That should be an interesting area to explore as work on CCP continues in the future: How much flexibility do you give up by using CCP's API instead of a ground-up design? But you're also raising the question of another limitation—performance. What's the CPU burned for CCP vs. Linux kernel CUBIC? How does that compare for a simple controller, or for a more complex machine-learning based controller, for example?

TA: And not just how much CPU but also how much remaining useful CPU you would have left? Of course it's not all limitations; let's get back to those potential benefits. Is it easier to program at user level or easier to do that in the kernel? We probably both agree that Linux is hard to modify! But that's a problem with Linux, not with the user/kernel split.

BG: Userspace programming could bring easier deployment, easier use of external libraries (like ML packages as the paper mentions), and easier floating point operations. Put all that together, and this is a package that I'd find valuable to use myself!

TA: This is perhaps a nit, but floating point is possible to use in the kernel! You just need to save and restore the floating point state. In practice, we don't do that because the overhead would be too high. At user level, the programming model is easier, but that's because you aren't counting the performance hit—the kernel still has to save and restore the fp registers on every process context switch.

BG: So you're saying it might be more convenient in user space, but there's no free lunch in performance.

TA: Right. Ages ago, POSIX added a way to catch and redirect system calls to user level, e.g., to implement a process jail. That turned out to be very useful—it is basically what is used for containers. But for that it is way too slow! So Linux lets you download system call interposition code into the kernel, using a form of BPF. If CCP proves useful, we'll need to figure out how to move it back into the kernel.

BG: There's another capability we haven't touched on yet: CCP lets you write once and run on many data paths (kernel, QUIC, hardware offload, etc.).

TA: In a world with kernel bypass, multiple protocols (QUIC, RDMA, TCP, UDP), and hardware NIC accelerators, how should the responsibilities for congestion control be partitioned among hardware bypass, kernel, user? Full disclosure: my own work on FlexNIC grapples with this, putting rate control in the NIC and computing CC policy offline in the kernel. I believe that the principal concern for the data path will be speed, and as a result, anything we can move offline—like CC policy—should be. In a data center setting, you can't afford to do useless work.

BG: ...which means one of the key tradeoffs that CCP grapples with (flexibility of the API and especially the flexibility vs. performance tradeoff) has a more general variant—what are the right tradeoffs for the interfaces between each of the data path components you mentioned? Even for the give-and-take between just the kernel and user space, the question really isn't settled yet.

TA: That's right, we'll have to see how it plays out. Should be a good discussion at the workshop!