DISTRIBUTED

Jump the Queue to Lower Latency

MATTHEW P. GROSVENOR, MALTE SCHWARZKOPF, IONEL GOG, AND ANDREW MOORE

Matthew P. Grosvenor is a PhD student at the University of Cambridge Computer Laboratory. His interests lie in cross-layer optimizations of networks, with a particular focus on network latency. He has completed research internships at NICTA (Sydney), Microsoft Research Silicon Valley, and Microsoft Research Cambridge, and he maintains strong ties to the high-speed networking vendor Exablaze.
matthew.grosvenor@cl.cam.ac.uk

Malte Schwarzkopf is currently finishing his PhD at the University of Cambridge Computer Laboratory. His research is primarily on operating systems and scheduling for datacenters, but he dallies in many a trade. He completed a research internship in Google's cluster management group and will join the PDOS group at MIT after graduating.
malte.schwarzkopf@cl.cam.ac.uk

Ionel Gog is a PhD student in the University of Cambridge Computer Laboratory. His research interests include distributed systems, data processing systems, and scheduling. He received his MEng in computing from Imperial College London and has done internships at Google, Facebook, and Microsoft Research.
ionel.gog@cl.cam.ac.uk

In this article, we show that it is possible and practical to achieve bounded latency in datacenter networks using QJUMP, an open-source tool that we've been building at the University of Cambridge. Furthermore, we show how QJUMP can concurrently support a range of network service levels, from strictly bounded latency through to line-rate throughput using the prioritization features found in any datacenter switch.

Bringing Back Determinism

In a statistically multiplexed network, packets share network resources in a first come, first served manner. A packet arriving at a statistically multiplexed ("stat-mux") switch (or router) is either forwarded immediately or forced to wait until the link is free. This makes it hard to determine how long the packet will take to cross the network. In other words, stat-mux networks do not provide *latency determinism*.

The desire to retrofit latency determinism onto Internet Protocol (IP) stat-mux networks sparked a glut of research in the mid-90s on "Quality of Service" (QoS) schemes. QoS technologies like DiffServ demonstrated that coarse-grained *classification* and *rate-limiting* could be used to control Internet network latencies. However, these schemes were complex to deploy and often required cooperation between multiple competing entities. For these reasons (and many others) Internet QoS struggled for widespread deployment, and hence provided limited benefits [1].

Today, the muscle behind the Internet is found in datacenters, with tens of thousands of networked compute nodes in each. Datacenter networks are constructed using the same fundamental building blocks as the Internet. Like the Internet, they use statistical multiplexing and Internet Protocol (IP) communication. Also like the Internet, datacenter networks suffer from lack of latency determinism, or "tail latency" problems. Worse still, the close coupling of applications in datacenters magnifies tail-latency effects. Barroso and Dean showed that, if as few as one machine in 10,000 is a straggler, up to 18% of user requests can experience long tail latencies [2].

Unsurprisingly, the culprit for these tail latencies is once again statistical multiplexing. More precisely, congestion from some applications causes queueing that delays traffic from other applications. We call the ability of networked applications to affect each others' latencies *network interference*. For example, Hadoop MapReduce can cause queueing that interferes with memcached request latencies, causing latency increases of up to 85x.

The good news is that datacenters are also unlike the Internet. They have well-known network structures, and the bulk of the network is under the control of a single authority. The differences between datacenters and the Internet allow us to apply QoS schemes in new ways, different and simpler than the Internet does. In datacenters, we can enforce a system-wide policy, and, using known host counts and link rates, we can calculate specific rate limits that allow us to provide a guaranteed bound on network latency.

We have implemented these ideas in QJUMP. QJUMP is a simple and immediately deployable approach to controlling network interference in datacenter networks. QJUMP is open source

Andrew W. Moore is a Senior Lecturer at the University of Cambridge Computer Laboratory in England, where he is part of the Systems Research Group working on issues of network and systems architecture. His research interests include enabling open-network research and education using the NetFPGA platform. Other research pursuits include low-power energy-aware networking and novel network and systems datacenter architectures. andrew.moore@cl.cam.ac.uk

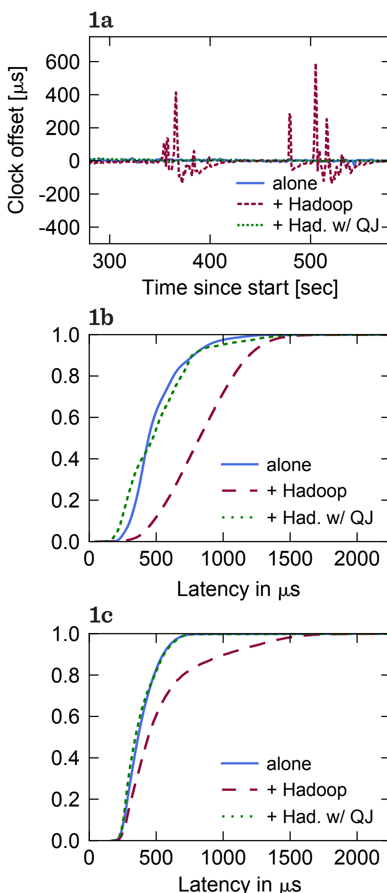

Figure 1a shows a timeline of PTPd synchronization offset. **Figure 1b** has a CDF of memcached request latency, and **Figure 1c** has a CDF of Naiad synchronization time.

and runs on unmodified hardware and software. A full paper describing QJump will appear in the 12th USENIX Symposium on Networked System Design and Implementation (NSDI '15) [3]. Additional information including source code and data is available from our accompanying Web site: http://www.cl.cam.ac.uk/research/srg/netos/qjump.

QJump in Action

To illustrate how bad network interference can get and how well QJump fixes it, we show the results from a collection of experiments with latency-sensitive datacenter applications (see Figure 1). In each experiment, the application: (1) runs alone on the network, (2) shares the network with Hadoop MapReduce, and (3) shares the network with Hadoop, but has QJump enabled. A complete evaluation of QJump, including full details of these experiments (and many others), is available in the full paper.

1. Clock Synchronization. Precise clock synchronization is important to distributed systems such as Google's Spanner. PTPd offers microsecond-granularity time synchronization from a time server to machines on a local network. However, it assumes roughly constant network delay. In Figure 1a, we show a timeline of PTPd synchronizing a host clock on both an idle network and when sharing the network with Hadoop. In the shared case, Hadoop causes queueing which delays PTPd's synchronization packets. This causes PTPd to temporarily fall 200–500 µs out of synchronization, 50x worse than on an idle network. With QJump enabled, the PTPd synchronization remains unaffected by Hadoop's traffic.

2. Key-Value Stores. Memcached is a popular in-memory key-value store used by Facebook and others to store small objects for quick retrieval. We benchmark memcached using the memaslap load generator and measure the request latency. Figure 1b shows the distribution of request latencies on an idle network and a network shared with Hadoop. With Hadoop running, the 99th percentile request latency degrades by 1.5x from 779 µs to 1196 µs. Furthermore, around 1 in 6,000 requests takes over 200 ms to complete, over 85x worse than the maximum latency on an idle network. With QJump enabled, these effects are mitigated.

3. Big Data Computation. Naiad [4] is a framework for big data computation. In some computations, Naiad's performance depends on low-latency synchronization between worker nodes. To test Naiad's sensitivity to network interference, we execute a synchronization benchmark (provided by the Naiad authors) with and without Hadoop running. Figure 1c shows the distribution of Naiad synchronization latencies in both situations. On an idle network, Naiad takes around 500 µs at the 99th percentile to perform a four-way synchronization. With interference, this grows to 1.1–1.5 ms, a 2–3x performance degradation. With QJump running, the performance nearly exactly conforms to the interference-free situation.

These experiments cover just a small set of applications, but there are many others that can also benefit from using QJump. Examples include coordination traffic for Software Defined Networking (SDN), distributed locking/consensus services, and fast failure detectors.

Scheduling and Queueing Latency

To understand how QJump works, we first need to understand the two main sources of latency nondeterminism in statistically multiplexed (stat-mux) networks: scheduling latency and queueing latency. In Figure 2a, a collection of packets (P) arrive at an idle switch S0. At

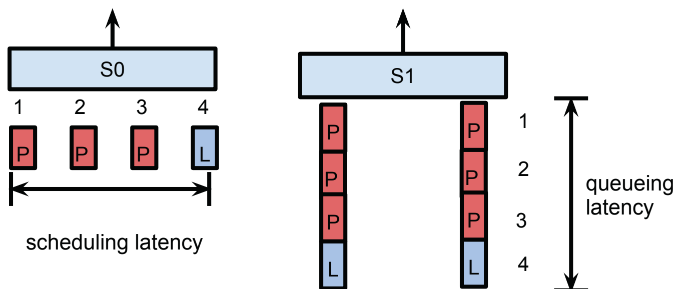

Figure 2: Latency causes (a) fan-in, packets waiting to be serviced by the switch scheduler, or (b) queueing, packets waiting behind many other packets.

the same time, a *latency sensitive* packet (L) also arrives. The L packet experiences *scheduling latency* as it waits for other P packets to be serviced by the switch scheduler. Scheduling latency is a consequence of *fan-in,* which happens when multiple packets contend for the same output port on the switch. If the switch takes too long to output packets, then new packets can queue behind existing ones. Figure 2b shows two latency sensitive packets (L) queued behind many other waiting packets (P). This is a kind of head-of-line blocking that we call *queueing latency*. Queuing latency is caused by excessive scheduling latency. We cannot eliminate scheduling latency in a stat-mux network. However, using some simple math, we can put a bound on it. By doing so, we can ensure that packets are issued into the network at a rate that prevents them from queueing up behind each other, thus also control queueing latency.

Bounded Queues—Bounded Latency

Considering Figure 2a, in the worst case the L packet will need to wait for the switch scheduler to service all preceding P packets before it is serviced. For a switch with n ports, the worst-case waiting time is $n - 1$ (approximately n) packets. As the number of ports on the switch grows, the worst-case latency grows with it.

We can easily expand this understanding to cover multi-hop networks by treating the whole network as a single "big switch" (this is an application of the "hose-constraint" [4] model). Hence we can apply the same calculation as above. Knowing that a packet of size P will take P/R seconds to transmit at link-rate R, we can therefore bound the maximum interference delay at:

$$\textit{worst case end-to-end delay} \leq n \times \frac{P}{R} \qquad (1)$$

where n is the number of hosts, P is the maximum packet size (in bits), and R is the rate of the slowest link in bits per second. Equation 1 assumes that hosts have only one (active) link to the network and that the speed at the core of the network is never slower than the speed at the edge. We think that these are both safe assumptions for any reasonable datacenter network.

We refer to the worst-case delay as a *network epoch*. A network epoch is the maximum time that an initially idle network will take to service one packet from every sending host, regardless of the source, destination, or timing of those packets. Intuitively, if we imagine the network as a funnel, the network epoch represents the time that the funnel will take to drain when it is filled to the top. If all hosts are rate-limited so that they cannot issue more than one packet per epoch, no permanent queues can build up, and the end-to-end network delay bound will be maintained forever. That is, we rate-limit hosts so that the funnel will never overflow.

The problem with a network epoch is that it is a global concept. To maintain it, all hosts need to agree on when an epoch begins and when it ends. It would seem that this requires all hosts in the network to have tightly synchronized clocks. In fact, network epochs can work even without clock synchronization. If we assume that network epochs occur at the same frequency, but not necessarily in the same phase, the network becomes *mesochronous*. This requires us to double the latency bound, but all other properties hold (see [3] for further details). The network epoch thus becomes:

$$\textit{network epoch} = 2n \times \frac{P}{R} \qquad (2)$$

Equation 2 is the basis for QJump. QJump is based on the principle that, if we rate-limit all hosts so that they can only issue one packet every network epoch, then no packet will take more than one network epoch to be delivered to the destination even in the worst case.

Latency Variance vs. Throughput

Although the equation derived above provides an absolute upper bound on in-network delay, it also aggressively restricts throughput. Formulating Equation 2 for throughput, we obtain:

$$\textit{throughput} = \frac{P}{\textit{network epoch}} = \frac{R}{2n} \qquad (3)$$

For example, with 1,000 hosts and a 10 Gb/s edge, we obtain an effective throughput of 5 Mb/s per host. Clearly, this is not ideal. We can improve this situation by making two observations. First, Equation 2 is pessimistic: it assumes that all hosts transmit to one destination at the worst time, which is unlikely given a realistic network and traffic distribution. Second, some applications, like PTPd, are more sensitive to interference than others—for example, memcached and Naiad—whereas still other applications, like Hadoop, are more sensitive to throughput restrictions. From the first observation, we can relax the throughput constraints in Equation 2 by assuming that fewer than n hosts send to a single destination at the worst time. For example, if we guess that only 500 of the 1,000 hosts concur-

rently send to a single destination, then those 500 hosts can send at twice the rate and maintain the same network delay if our assumption holds. More generally, we define a scaling factor *f* so that the assumed number of senders n' is given by:

$$n' = \frac{n}{f} \quad \text{where} \quad 1 \leq f \leq n. \qquad (4)$$

Intuitively, *f* is a "throughput factor": as the value of *f* grows, so does the available bandwidth.

From the second observation, some (but not all) applications can tolerate some degree of latency variance. Instead, for these applications we aim for a statistical reduction in latency variance. This reintroduces a degree of statistical multiplexing to the network, albeit one that is more tightly controlled. When the guess for *f* is too optimistic (the actual number of senders is greater than n'), some queueing occurs, causing interference.

The probability that interference occurs increases with increasing values of *f*. At the upper bound ($f = n$), latency variance is similar to existing networks and full network throughput is available. At the lower bound ($f = 1$), latency is guaranteed, albeit with reduced throughput. In essence, *f* quantifies the latency variance vs. throughput tradeoff.

Jump the Queue with Prioritization

We would like to use multiple values of *f* concurrently, so that different applications can benefit from the latency variance vs. throughput tradeoff that suits them best. To achieve this, we partition the network so that traffic from latency-sensitive applications, like PTPd, memcached, and Naiad can "jump-the-queue" over traffic from throughput-intensive applications like Hadoop. Ethernet switches support the IEEE 802.1Q standard, which provides eight (0–7) hardware enforced "service classes" or "priorities."

The problem with using priorities is that they can become a "race to the top." For example, memcached developers may assume that memcached traffic is the most important and should receive the highest priority to minimize latency. Meanwhile, Hadoop developers may assume that Hadoop traffic is the most important and should similarly receive the highest priority to maximize throughput. Since there are a limited number of priorities, neither can achieve an advantage and prioritization loses its value. QJump is different: it intentionally binds priority values to rate-limits. High priorities are given aggressive rate limits (small *f* values), and priorities thus become useful because they are no longer "free." QJump users must choose between low latency variance at low throughput (high priority) and high latency variance at high throughput (low priority). We call the assignment of an *f* value to a priority a "QJump level." The latency variance of a given QJump level depends on the number of QJump levels above it and their traffic patterns.

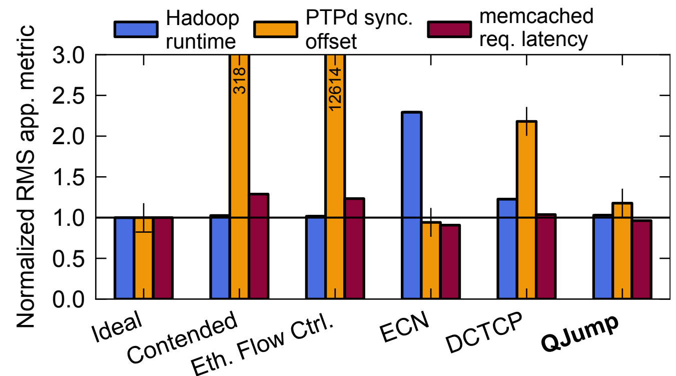

Figure 3: Comparison of QJump to several available congestion control alternatives

Implementation

QJump has two components: a rate-limiter to provide admission control to the network, and an application utility to configure unmodified applications to use QJump levels. Our full paper describes the rate limiter and application utility in detail, and the source code for both is available from our Web site.

In our prototype, we use our own high-performance rate limiter built upon the queueing discipline (qdisc) mechanism offered by the Linux kernel traffic control (TC). TC modules do not require kernel modifications and can be inserted and removed at runtime, making them flexible and easy to deploy.

To support unmodified applications, we implemented a utility that dynamically intercepts socket setup system calls and alters their options. We inject the utility into unmodified executables via the Linux dynamic linker's `LD_PRELOAD` support.

Performance Comparison

We have already demonstrated that QJump can resolve network interference, but how does it compare to existing congestion control mechanisms? To find out, we have tested QJump against several readily deployable congestion control schemes. In these experiments, PTPd, memcached, and Hadoop are configured to run on the same network for a 10-minute period. Since interference is transient in these experiments, we measure the degree to which it affects applications using the root mean square (RMS) of each application-specific metric. For Hadoop, the metric of interest is the job runtime, for PTPd it is the time synchronization offset, and for memcached it is the request latency. Figure 3 shows six cases: an ideal case, a contended case, and one for each of the four comparison schemes. All cases are normalized to the ideal case, which has each application running alone on an idle network.

Ethernet Flow Control

Like QJump, Ethernet Flow Control is a data link layer congestion control mechanism. Hosts and switches issue special *pause* messages when their queues are nearly full, alerting senders to slow down. Figure 3 shows that Ethernet Flow Control (Pause frames) has a limited positive impact on memcached but increases the RMS offset for PTPd. Hadoop's performance remains unaffected.

Early Congestion Notification (ECN)

ECN is a network-layer mechanism in which switches indicate queueing to end hosts by marking TCP packets. Our Arista 7050 switches implement ECN with weighted random early detection (WRED). The effectiveness of WRED depends on an administrator correctly configuring upper and lower marking thresholds. We investigated 10 different marking threshold pairs, ranging between [5, 10] and [2560, 5120], in packets. None of these settings achieved ideal performance for all three applications, but the best compromise was [40, 80]. With this configuration, ECN very effectively resolves the interference experienced by PTPd and memcached. However, this comes at the expense of increased Hadoop job runtimes.

Datacenter TCP (DCTCP)

DCTCP uses the rate at which ECN markings are received to build an estimate of network congestion. It applies this to a new TCP congestion avoidance algorithm to achieve lower queueing delays. We configured DCTCP with the recommended ECN marking thresholds of [65, 65]. Figure 3 shows that DCTCP reduces the variance in PTPd synchronization and memcached latency compared to the contended case. However, this comes at an increase in Hadoop job runtimes, as Hadoop's bulk data transfers are affected by DCTCP's congestion avoidance.

QJump

Figure 3 shows that QJump achieves the best results. The variance in Hadoop, PTPd, and memcached performance is close to the uncontended ideal case.

Conclusion

QJump applies QoS-inspired concepts to datacenter applications to mitigate network interference. It offers multiple QJump levels with different latency variance vs. throughput tradeoffs, including bounded latency (at low rate) and full utilization (at high latency variance). QJump is readily deployable, open source, and requires no hardware, protocol, or application changes.

Our source code and all experimental data sets are available at http://www.cl.cam.ac.uk/research/srg/netos/qjump.

Acknowledgments

The full paper version of this work includes contributions from Jon Crowcroft, Steven Hand, and Robert N. M. Watson. This work was jointly supported by a Google Fellowship, EPSRC INTERNET Project EP/H040536/1, the Defense Advanced Research Projects Agency (DARPA), and the Air Force Research Laboratory (AFRL), under contract FA8750-11-C-0249. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

References

[1] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield, "QoS's Downfall: At the Bottom, or Not at All!" in *Proceedings of the ACM SIGCOMM Workshop on Revisiting IP QoS,* 2003, pp. 109–114.

[2] J. Dean and L. A. Barroso, "The Tail at Scale: Managing Latency Variability in Large-Scale Online Services," *Communications of the ACM*, vol. 56, no. 2 (Feb. 2013), pp. 74–80.

[3] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues Don't Matter if You Can JUMP Them!" forthcoming in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15),* May 2015.

[4] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, and P. Barham, "Naiad: A Timely Dataflow System," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP),* 2013, pp. 439–455.

[5] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive, "A Flexible Model for Resource Management in Virtual Private Networks," in *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM),* Aug. 1999, pp. 95–108.