A Cloud-Optimized Transport Protocol for Elastic and Scalable HPC

Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag
Annapurna Labs, Amazon Web Services

***Abstract*—Amazon Web Services (AWS) took a fresh look at the network to provide consistently low latency required for supercomputing applications, while keeping the benefits of public cloud: scalability, elastic on-demand capacity, cost effectiveness, and fast adoption of newer CPUs and GPUs. We built a new network transport protocol, scalable reliable datagram (SRD), designed to utilize modern commodity multitenant datacenter networks (with a large number of network paths) while overcoming their limitations (load imbalance and inconsistent latency when unrelated flows collide). Instead of preserving packets order, SRD sends the packets over as many network paths as possible, while avoiding overloaded paths. To minimize jitter and to ensure the fastest response to network congestion fluctuations, SRD is implemented in the AWS custom Nitro networking card. SRD is used by HPC/ML frameworks on EC2 hosts via AWS elastic fabric adapter kernel-bypass interface.**

ONE OF THE major benefits of cloud computing is the ability to instantaneously provision and deprovision resources as needed. This is strikingly different from traditional supercomputing, where physical supercomputers are custom-built (taking months or years) and not easy to get access to, because of their high cost and limited capacity. One of the main reasons for using custom-built systems for supercomputing is the challenges of building a high-performance network and sharing it between applications. In the context of cloud computing, using either specialized hardware such as InfiniBand or commodity hardware dedicated exclusively to HPC workloads is prohibitively expensive, hard to scale, and hard to evolve fast.

Digital Object Identifier 10.1109/MM.2020.3016891

Amazon Web Services (AWS) opted to provide customers access to affordable supercomputing using the existing AWS network (starting from 100 Gbps) and added a new HPC-optimized network interface as an extension of the network functionality offered by AWS Nitro cards.

As expected, running HPC traffic on a shared network comes with its own set of challenges. AWS uses commodity Ethernet switches to build high-radix Folded Clos topology with equal-cost multipath (ECMP) routing. ECMP is commonly used to statically stripe flows across available paths using flow hashing. This static mapping of flows to paths is beneficial for keeping the per-flow order for TCP, but it does not account for current network utilization or flow rate. Hash collisions result in "hotspots" on some of the links, causing nonuniform load distribution across paths, packet drops, decreased throughput, and high tail latency (as studied extensively, e.g., in the articles by Al-Fares *et al.*,[1] Ghorbani *et al.*,[2] Handley *et al.*,[3] Hopps *et al.*,[4] and Vanini *et al.*).[5] A heavy flow can potentially affect unrelated applications even in an over-provisioned network.

Packet delays and packet drops interfere with the low-latency requirements of HPC/ML applications, resulting in reduced scaling efficiency. Latency outliers have a profound impact on these applications, as they typically follow bulk synchronous parallel programming model, with epochs of computation followed by bulk synchronization across the whole cluster. A single outlier would keep the entire cluster waiting, limiting scalability due to Amdahl's law.

Why Not TCP

TCP is the primary means of reliable data transfer in IP networks, which has served the Internet well since its inception and continues to be the optimal protocol for the majority of the communication. However, it is ill-suited for latency-sensitive processing. For TCP in a data center, while best-case round-trip latency could be as good as 25 μs, the latency outliers under congestion (or link faults) can be anywhere between 50 ms and several seconds, even when alternative noncongested network paths are available. One of the main reasons for these outliers is a retransmission of lost TCP packets: TCP implementations are forced to keep retransmission timeout high, to account for OS delays.

Why Not RoCE

InfiniBand is a popular high-throughput low-latency interconnect for high-performance computing, which supports kernel bypass and transport offload. RDMA over Converged Ethernet (RoCE), also known as InfiniBand over Ethernet, allows running InfiniBand transport over Ethernet and could in theory provide an alternative to TCP in AWS datacenters. We considered the RoCEv2 support, and elastic fabric adapter (EFA) host interface closely resembles the InfiniBand/RoCE interface. However, we found InfiniBand transport to be unsuitable for AWS scalability requirements. One of the reasons was that RoCE [v2] required priority flow control (PFC), which is not feasible on large-scale networks, because it creates head-of-the-line blocking, congestion spreading, and occasional deadlocks. One solution to PFC problems at scale was described in the article by Guo,[6] but it explicitly relied on datacenter size significantly smaller than that of AWS datacenters. Moreover, even with PFC, RoCE would still suffer from ECMP collisions under congestion, similar to TCP, and suboptimal congestion control.[7]

Our Approach

Since neither TCP nor other transport protocols provide the level of performance we need, in the network we use, we chose to design our own network transport. Scalable reliable datagram (SRD) is optimized for hyper-scale datacenters: it provides load balancing across multiple paths and fast recovery from packet drops or link failures. It utilizes standard ECMP functionality on the commodity Ethernet switches and works around its limitations: the sender controls the ECMP path selection by manipulating packet encapsulation. SRD employs a specialized congestion control algorithm that helps further decrease the chance of packet drops and minimize retransmit times, by keeping queuing to a minimum.

We made a somewhat unusual choice of protocol guarantees: SRD provides reliable but out-of-order delivery and leaves order restoration to the layers above it. We found that strict in-order delivery is often not necessary and enforcing it would just create head-of-line blocking, increase latency, and reduce bandwidth. For example, message passing interface (MPI) tagged messages only have to be delivered in-order if the same message tag is used. Therefore, when parallelism in the network causes packet arrival out-of-order, we leave the message order restoration to the upper layer, because it has a better understanding of the required ordering semantics.

We choose to implement the SRD reliability layer in the AWS Nitro card. Our goal was to have SRD as close as possible to the physical network layer and to avoid performance noise injected by the host OS and hypervisor. This allows fast adaptation to network behavior: fast retransmission and prompt slowdown in response to queue build-up.

SRD is exposed to the host as an EFA PCIe device. EFA is a network interface for Amazon EC2 instances (i.e., virtual and bare-metal servers) that enables customers to run tightly coupled applications at scale on AWS. In particular, EFA enables running HPC applications and ML distributed training, currently supported in several MPI implementations: OpenMPI, Intel MPI, and MVAPICH, as well as NVIDIA Collective Communications Library. As shown in Figure 1, EFA offers a "user-space driver" that utilizes the operating system (OS) bypass hardware interface to enhance the performance of inter-instance communication (reducing latency, jitter, avoiding OS system calls, and reducing memory copies), which is key to scaling these applications.

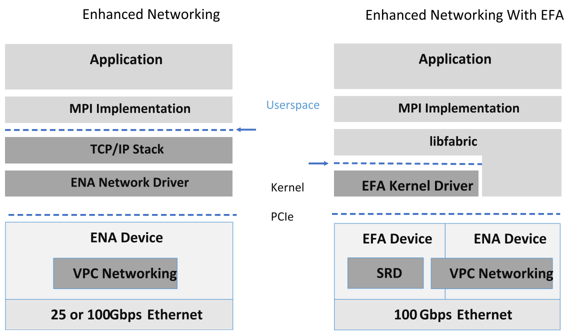

Figure 1. HPC Stack with and without EFA.

SCALABLE RELIABLE DATAGRAM DESIGN

Multipath Load Balancing

To decrease the chance of packet drops, the traffic should be distributed uniformly across available paths. The SRD sender needs to spray packets over multiple paths even for a single application flow, especially for a heavy flow, to minimize the chance of hotspots and also to detect suboptimal paths. We designed SRD to share the network with legacy traffic, which is not multipath-enabled, therefore it is not enough to just spray the traffic randomly. To minimize the impact of heavy legacy flows, SRD avoids overloaded paths using round-trip time (RTT) information collected for each path.

At scale, occasional hardware failures are unavoidable; to allow fast recovery from network link failures, SRD is able to reroute a retransmitted packet in case the path used for original transmission became unavailable, without waiting for network-wide routing updates convergence which takes 2–3 orders of magnitude longer. This route change is done by SRD without costly connection re-establishment.

Out of Order Delivery

Balancing the traffic across the multiple available paths helps to reduce queuing latency and to prevent packet drops, however, it inevitably leads to out-of-order packet arrival in large networks. It is notoriously expensive to restore packet ordering in network cards, which typically have limited resources (memory bandwidth, reordering buffer capacity, or number of open ordering contexts).

We considered having the Nitro network card deliver in-order receive messages, similar to common reliable transports like TCP or infiniband reliable connections (RC). However, that would either limit scalability or increase average latency in the presence of drops. If we postpone delivery of out-of-order packets to the host software, we would need a large intermediate buffer, and we would greatly increase average latency, as many packets are delayed until the missing one is resent. Most or all of these packets are likely to be unrelated to the lost packet, so such delay is unnecessary. Dropping out-of-order packets "solves" the buffering problem, but not the latency problem, and increases network bandwidth consumption. Therefore, we decided to deliver packets to the host even when they might be out-of-order.

Handling out-of-order packets by an application is untenable with a byte streaming protocol such as TCP where message boundaries are opaque to the transport layer but is easy when using message-based semantics. The per-flow ordering or other kind of dependency tracking is done by the messaging layer above SRD; the messaging-layer sequencing information is transferred with the packet to the other side, opaque to SRD.

Congestion Control

Multipath spraying reduces the load on intermediate switches in the network, but by itself does nothing to alleviate *incast* congestion problem. Incast is a traffic pattern in which many flows converge on the same interface of a switch, exhausting the buffer space for that interface, resulting in packet drops. It is common in the last-hop switch connected to the receiver in many-to-one communication patterns, but it may happen at other layers as well.[2]

Spraying can actually make incast problem worse, as micro-bursts from the same sender, even though originally limited by link bandwidth of the sender, may arrive on different paths simultaneously. Therefore, it is critical that congestion control for a multipath transport keeps aggregate queueing on all paths to a minimum.

The objective of SRD congestion control is to get a fair share of the bandwidth with minimum in-flight bytes, preventing queue buildup and preventing packet drops (rather than relying on them for congestion detection). SRD congestion control is somewhat similar to BBR,[8] with additional datacenter multipath considerations. It is based on a per-connection dynamic rate limit, combined with an inflight limit. The sender adjusts its per-connection transmission rate according to rate estimation as indicated by the timing of incoming acknowledge packets, taking into account also the recent transmit rate and RTT changes. Congestion is detected if the RTT goes up on the majority of paths, or if the estimated rate becomes lower than the transmit rate. This method allows detection of connection-wide congestion affecting all paths, e.g., in case of incast. Congestion on an individual path is handled independently by rerouting.

USER INTERFACE: EFA

SRD transport on the Nitro card is exposed to AWS customers over EFA. EFA interface resembles InfiniBand verbs.[9] However, its SRD semantics are drastically different from standard InfiniBand transport types. EFA user-space software comes in two flavors: the basic "user-space driver" software exposes reliable out-of-order delivery as provided natively by the Nitro card EFA hardware device, while libfabric[10] "provider" layered above it implements packet reordering as a part of message segmentation and MPI tag matching support.

EFA as an Extension of Elastic Network Adapter

The Nitro cards are a family of cards that offloads and accelerates network, storage, security, and virtualization functions on AWS EC2 servers. In particular, Nitro Card for VPC includes the elastic network adapter (ENA) PCIe Controller that presents classic network devices to the host, while also implementing the data plane for AWS VPC. Enhanced Networking uses PCIe single root I/O virtualization (SR-IOV) to provide high-performance networking capabilities without hypervisor involvement; it exposes dedicated PCIe devices to EC2 instances running on AWS host, resulting in higher I/O performance, lower latency and lower CPU utilization when compared to traditional para-virtualized network interfaces. EFA is an additional optional service provided by Nitro VPC cards on AWS high-performance servers suited for HPC and ML.

EFA SRD Transport Type

As in InfiniBand verbs, all EFA data communication is done via queue pairs (QPs), which are addressable communication endpoints containing a send queue and a receive queue, used to submit requests to asynchronously send and receive messages, directly from/to user-space. QPs are expensive resources, and traditionally a large number of QPs were necessary to establish all-to-all process connectivity in large clusters (where a large number of processes typically run on each server). EFA SRD transport allows significant savings in the required number of QPs as described in https://github.com/amzn/amzn-drivers/blob/master/kernel/linux/efa/SRD.txt.[11] EFA SRD semantics resemble InfiniBand reliable datagram (RD) model, but eliminate the RD limitations (caused by untenable complexity of handling interleaved segmented messages from different senders to the same destination QP, while providing in-order delivery). Unlike RD, SRD QPs deliver data out-of-order and limit message size to avoid segmentation. This allows support for multiple outstanding messages without creating head-of-line blocking so that separate application flows can be multiplexed without interfering with each other.

Out of Order Packet Handling Challenges

EFA SRD QP semantics introduce an unfamiliar ordering requirement for EFA upper layer processing, which we call "Messaging Layer," typically used by HPC applications to abstract away network specifics. This new functionality is lightweight comparing to full-blown transport implementation (such as TCP), as the reliability layer is offloaded.

Ideally, the buffer management and flow control done by Messaging Layer should be tightly coupled with the application, which is feasible since our primary focus is HPC-like applications, which already support and actually prefer user-space networking with the ability to manage user buffers.

With Message semantics, out-of-order arrival of message segments for a large transfer may necessitate data copy, if the application messaging layer expects to receive the data into a virtually contiguous buffer rather than a gather list. This is not worse than TCP, which requires a copy from kernel buffers to user buffers. This copy can be avoided in EFA using RDMA capability (out-of-scope of this article).

SRD PERFORMANCE EVALUATION

We compared EFA SRD performance to TCP (with default configuration) on the AWS cloud, on the same set of servers. We do not analyze the differences due to OS kernel bypass, because a) its impact in EFA is not substantially different from well-studied InfiniBand case and b) it is

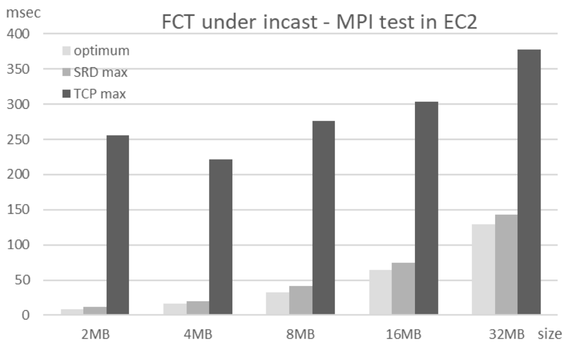

Figure 2. Maximum FCT, bursty 48 flows incast.

minor compared to transport behavior differences under congestion.

The MPI implementation is another factor that has a profound impact on HPC application performance, in particular, for MPI on early versions of EFA as was shown in the article by Chakraborty *et al.*[12] Since our goal is to evaluate the transport protocol, and MPI implementation is out of the scope of this article, we only used basic MPI primitives (including reordering logic) in OpenMPI, or micro-benchmarks bypassing the MPI layer.

Incast FCT and Fairness

We evaluated 48 independent flows sent from 4 servers running 12 processes each, to a single destination server, creating a bottleneck at the last network hop. We measure flow completion time (FCT) for SRD and TCP, and compare it to optimum FCT, i.e., the ideal FCT in case of 100% utilization of the bottleneck link divided equally between the flows.

"Bursty" Incast FCT We ran an MPI bandwidth benchmark over EFA/SRD or TCP, when the senders use a barrier to start each transfer at approximately the same time. Figure 2 shows the ideal and maximum FCT for different transfer sizes. SRD FCT is close to the optimum with very low jitter, while TCP FCT is noisy, when maximal time is 3–20 times higher than the ideal.

Figure 3 shows a CDF of FCT for 2 MB transfers. TCP tail latency above 50 ms reflects retransmits, as minimal retransmission timeout is 50 ms. Even when looking only at the samples below 50 ms (i.e., when delays are not attributable to timeouts), a large number of samples are 3 times higher than ideal.

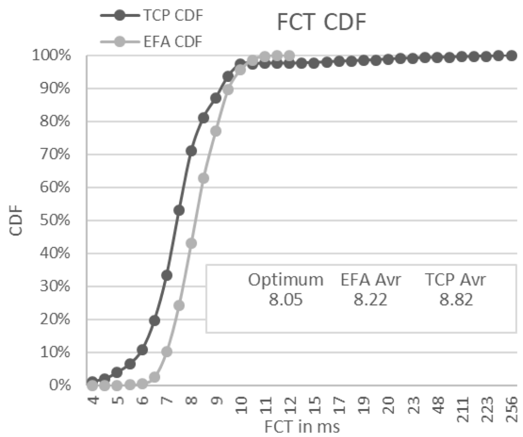

Figure 3. CDF of FCT for 2 MB transfers, bursty 48 flows incast.

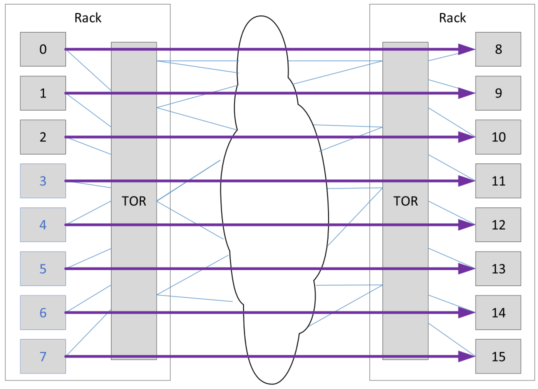

Figure 5. Independent flows sharing inter-switch links.

Flow Throughput Under Persistent Congestion Incast To understand the high FCT variance for TCP (even when ignoring long tail due to timeouts), we examined individual flow throughput under incast. We used low-level benchmarks bypassing MPI, to measure throughput when sending data continuously. We sampled the throughput of each flow every second. At a combined rate of 100 Gb/s, the expected fair share of each flow is approximately 2 Gb/s.

Figure 4 shows the TCP and SRD throughput for two representative senders each. SRD flows throughput is consistent and close to ideal for all flows, while TCP throughput of each flow is oscillating wildly, and some flows have much lower average throughput than expected, which explains FCT jitter.

Multipath Load Balancing

We evaluated also a less demanding case, without correlated load. As depicted in Figure 5, we ran multiple flows from 8 servers located in the same rack to 8 servers located in another rack, in a full-bisection network. Each machine ran 16 MPI ranks (processes), all sending or receiving data on separate flows to/from the same remote machine. The TOR switch uplinks are utilized at 50%, and downlinks are not expected to be congested as only one sender sends to any receiver.

Figure 6 shows the FCT for all flows of one of the 8 senders for TCP and EFA (other senders look similar). Even though with ideal load balancing there would be no congestion at all, TCP clearly experienced congestion and even packet drops, because of nonuniform ECMP balancing for inter-switch links. TCP median latency is highly variable and the average is 50% higher than expected, while tail latency is 1–2 orders of magnitude higher than expected. Median SRD FCT is just 15% higher than ideal, and maximal SRD FCT is lower than average TCP FCT.

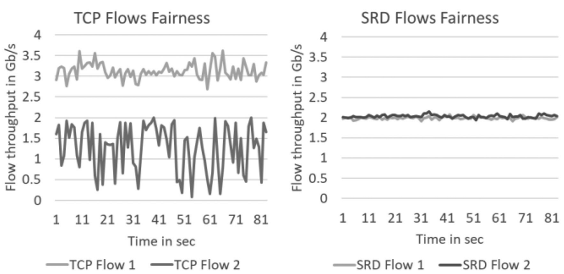

Figure 4. Throughput sampled each second, 48-way incast, representative flows.

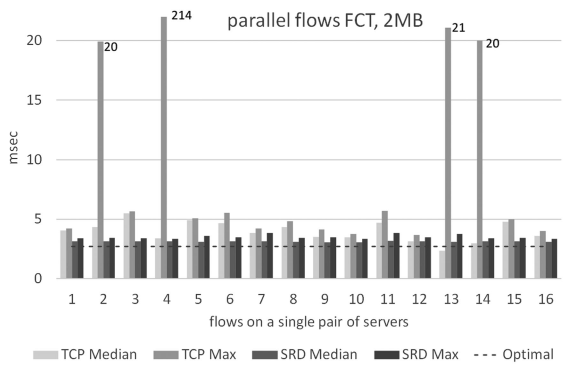

Figure 6. Impact of ECMP imbalance.

CONCLUSION

EFA allows HPC/ML applications to run on AWS public cloud at scale. It provides consistently low latency, with tail latency orders of magnitude lower than that of TCP. This is achieved by the novel network transport semantics provided by SRD, combined with an unorthodox split of functionality between the network interface card and different layers of host software. By running SRD multipath load balancing and congestion control on Nitro card, we both decrease the chance of packet drops in the network and enable faster recovery from drops.

ACKNOWLEDGMENTS

The authors would like to thank E. Izenberg, Z. Machulsky, S. Bshara, M. Wilson, P. DeSantis, A. Judge, T. Scholl, R. Galliher, M. Olson, B. Barrett, and A. Liguori for their help with distilling SRD and EFA requirements and for reviewing the design. The authors would also like to thank AWS Nitro chip team, EFA, SRD, and LibFabric teams for building the hardware and software that implements SRD and EFA.

REFERENCES

1. M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, “Hedera: Dynamic flow scheduling for data center networks,” in *Proc. Netw. Syst. Des. Implementation Symp.*, 2010, pp. 281–295.
2. S. Ghorbani, Z. Yang, B. Godfrey, Y. Ganjali, and A. Firoozshahian, “DRILL: Micro load balancing for low-latency data center networks,” in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2017, pp. 225–238.
3. M. Handley *et al.*, “Re-architecting datacenter networks and stacks for low latency and high performance,” in *Proc. Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 29–42.
4. C. Hopps, “Analysis of an equal-cost multi-path algorithm,” RFC 2992, 2000.
5. E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, “Let it flow: Resilient asymmetric load balancing with flowlet switching,” in *Proc. 14th USENIX Conf. Networked Syst. Des. Implementation*, 2017.
6. C. Guo *et al.*, “RDMA over commodity ethernet at scale,” in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 202–215.
7. R. Mittal *et al.*, “Revisiting network support for RDMA,” in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2018, pp. 313–326.
8. N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, “BBR: Congestion-based congestion control,” *ACM Queue*, vol. 14, pp. 20–53, Jan. 2017.
9. InfiniBand Architecture Specification. vol. 1, Release 1.3.
10. OpenFabricsWorkingGroup.Libfabric. [Online]. Available: https://ofiwg.github.io/libfabric/
11. [Online]. Available: https://github.com/amzn/amzn-drivers/blob/master/kernel/linux/efa/SRD.txt
12. S. Chakraborty, S. Xu, H. Subramoni, and D. Panda, “Designing scalable and high-performance MPI libraries on Amazon elastic fabric adapter,” in *Proc. IEEE Symp. High-Perform. Interconnects*, 2019, pp. 40–44.

Leah Shalev is a Senior Principal Engineer with Amazon, working on high-speed networking for HPC and storage in AWS. Her interests include cloud system architecture, datacenter networking, high-performance server I/O, hardware acceleration, and performance modeling. Shalev received the M.Sc. degree in computer science from the Technion in 1998. Contact her at shalevl@amazon.com.

Hani Ayoub is a Staff Software Engineer with Amazon, taking part in the AWS Nitro system development. His focus is leading the network transport protocol development, and other areas including: networking, storage, HPC, HW offloads and SW/HW interactions. Ayoub received the B.Sc. degree in computer science from the Technion in 2012. Contact him at ayoubh@amazon.com.

Nafea Bshara is a VP/Distinguished Engineer with Amazon, working on system architectures for AWS infrastructure. His roles involve leading projects and architecture spanning hypervisor, machine learning, Server, networking, storage, and optics. Bshara received the M.Sc. degree in computer engineering from the Technion in 2002. Contact him at nafea@amazon.com.

Erez Sabbag is a Principal Engineer with Amazon Web Service (AWS), working on system architecture of AWS cloud infrastructure, including computer networks, machine learning, and storage systems. Sabbag received the Ph.D. degree in electrical engineering from the Technion in 2011. Contact him at esabbag@amazon.com.