Network Working Group
Internet-Draft
Intended status: Experimental
Expires: December 19, 2020

H. Liu
R. Miao
Alibaba Group
R. Pan
JK. Lee
C. Kim
Intel Corporation
June 17, 2020

HPCC++: Enhanced High Precision Congestion Control
draft-pan-tsvwg-hpccplus-00

Abstract

Congestion control (CC) is the key to achieving ultra-low latency, high bandwidth and network stability in high-speed networks. However, the existing high-speed CC schemes have inherent limitations for reaching these goals.

In this document, we describe HPCC++ (High Precision Congestion Control), a new high-speed CC mechanism which achieves the three goals simultaneously. HPCC++ leverages in-network telemetry (INT) to obtain precise link load information and controls traffic precisely. By addressing challenges such as delayed INT information during congestion and overreaction to INT information, HPCC++ can quickly converge to utilize free bandwidth while avoiding congestion, and can maintain near-zero in-network queues for ultra-low latency. HPCC++ is also fair and easy to deploy in hardware, implementable with commodity programmable NICs and switches.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on December 19, 2020.

Table of Contents

1. Introduction

The link speed in data center networks has grown from 1Gbps to

100Gbps in the past decade, and this growth is continuing. Ultralow latency and high bandwidth, which are demanded by more and more applications, are two critical requirements in today's and future high-speed networks.

Given that traditional software-based network stacks in hosts can no longer sustain the critical latency and bandwidth requirements [Zhu-SIGCOMM2015], offloading network stacks into hardware is an inevitable direction in high-speed networks. large-scale networks with RDMA (remote direct memory access) over Converged Ethernet Version 2 (RoCEv2) often becomes hardware-offloading solution. Unfortunately, the RDMA networks still face fundamental challenges to reconcile low latency, high bandwidth utilization, and high stability.

This document describes a new CC mechanism, HPCC++ (Enhanced High Precision Congestion Control), for large-scale, high-speed networks. The key idea behind HPCC++ is to leverage the precise link load information from INT to compute accurate flow rate updates. Unlike existing approaches that often require a large number of iterations to find the proper flow rates, HPCC++ requires only one rate update step in most cases. Using precise information from INT enables HPCC++ to address the three limitations in current CC schemes. First, HPCC++ senders can quickly ramp up flow rates for high utilization or ramp down flow rates for congestion avoidance. Second, HPCC++ senders can quickly adjust the flow rates to keep each link's input rate slightly lower than the link's capacity, preventing queues from being built-up as well as preserving high link utilization. Finally, since sending rates are computed precisely based on direct measurements at switches, HPCC++ requires merely three independent parameters that are used to tune fairness and efficiency.

The base form of HPCC++ is the original HPCC algorithm and its full description can be found in [SIGCOMM-HPCC]. While the original design lays the foundation for INT-based precision congestion control, HPCC++ is an enhanced version which takes into account system constraints and aims to reduce the design overhead and further improves the performance. Section 6 describes these detailed proposed design enhancements and guidelines.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. System Overview

Figure 1 shows the end-to-end system that HPCC++ operates in. During the propagation of the packet from the sender to the receiver, each switch along the path leverages the INT feature of its switching ASIC to insert some meta-data that reports the current load of the packet's egress port, including timestamp (ts), queue length (qLen), transmitted bytes (txBytes), and the link bandwidth capacity (B). When the receiver gets the packet, it can copies all the meta-data recorded by the switches to the ACK message it sends back to the sender, and then the sender decides how to adjust its flow rate each time it receives an ACK with network load information. Alternatively, the receiver can calculate the flow rate based on the INT information and feedback the calculated rate back to the sender. The nofication packets would include delayed ack information as well.

Note that there also exist network nodes along the reverse (potentially uncongested) path that the RTCP feedback reports traverse. Those network nodes are not shown in the figure for sake of brevity.

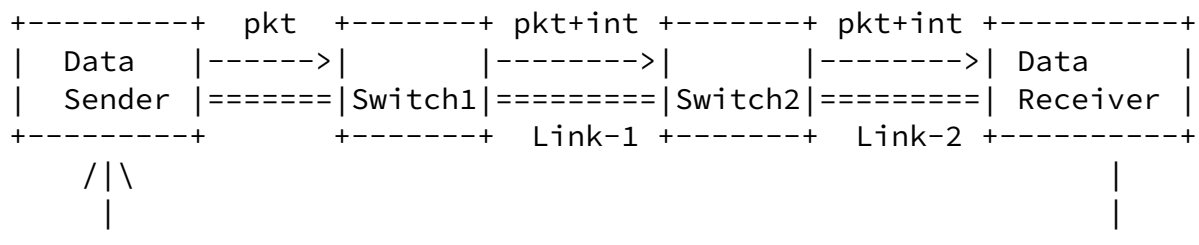

Figure 1: System Overview

- Data sender: responsible for controlling inflight bytes. HPCC++ is a window-based CC scheme that controls the number of inflight bytes. The inflight bytes mean the amount of data that have been sent, but not acknowledged at the sender yet. Controlling inflight bytes has an important advantage compared to controlling rates. In the absence of congestion, the inflight bytes and rate are interchangeable with equation inflight = rate * T where T is the base propagation RTT. The rate can be calculated locally or obtained from the notification packet.

- Network nodes: responsible of inserting the INT information to the data packet. The INT information reports the current load of the packet's egress port, including timestamp (ts), queue length (qLen), transmitted bytes (txBytes), and the link bandwidth

capacity (B). Note that the INT information can be nested with each network node adds its own information or more capable network nodes may compare its INT information against the one in the packet header. If its congestion is more severe, the node may replace the packet's INT information with its own.

- Data receiver: responsible for either reflecting back the INT information in the data packet using ACKs or calculating the proper flow rate based on network congestion information in INT and sending notification packets back to the sender.

4. HPCC++ Algorithm

HPCC++ is a window-based congestion control algorithm. The key design choice of HPCC++ is to rely on network nodes to provide fine-grained load information, such as queue size and accumulated tx/rx traffic to compute precise flow rates. This has two major benefits: (i) HPCC++ can quickly converge to proper flow rates to highly utilize bandwidth while avoiding congestion; and (ii) HPCC++ can consistently maintain a close-to-zero queue for low latency.

This section introduces the list of notations and describes the core

congestion control algorithm.

4.1. Notations

This section summarizes the list of variables and parameters used in the HPCC++ algorithm. Figure 3 also includes the default values for choosing the algorithm parameters either to represent a typical setting in practical applications or based on theoretical and simulation studies.

Notation	Variable Name
W_i	Window for flow i
Wc_i	Reference window for flow i
B_j	Bandwidth for Link j
I_j	Estimated inflight bytes for Link j
U_j	Normalized inflight bytes for Link j
qlen	INT information: link j queue length
txRate	INT information: link j output rate
ts	INT information: timestamp
txBytes	INT information: link j total transmitted bytes associated with timestamp ts

Figure 2: List of variables.

Notation	Parameter Name	Default Value
T	Known baseline RTT	5us
eta	Target link utilization	95%
maxStage	Maximum stages for additive increases	5
N	Maximum number of flows	...
W_ai	Additive increase amount	...

Figure 3: List of algorithm parameters and their default values.

4.2. Design Functions and Procedures

The HPCC++ algorithm can be outlined as below:

```
1:    Function MeasureInflight(ack)
2:    u = 0;
3:    for each link i on the path do
4:              ack.L[i].txBytes-L[i].txBytes
      txRate =  ---------------------------- ;
                   ack.L[i].ts-L[i].ts
5:          min(ack.L[i].qlen,L[i].qlen)      txRate
      u' = ---------------------------- +  ---------- ;
                ack.L[i].B*T                ack.L[i].B
6:    if u' > u then
7:       u = u'; tau = ack.L[i].ts -  L[i].ts;
8:    tau = min(tau, T);
9:    U = (1 - tau/T)*U + tau/T*u;
10:   return U;

11:   Function ComputeWind(U, updateWc)
12:   if U >= eta or incStage >= maxStagee then
13:            Wc
         W = ----- + W_ai;
             U/eta
14:   if updateWc then
15:      incStagee = 0; Wc = W ;
16:   else
17:      W = Wc + W_ai ;
18:      if updateWc then
19:         incStage++; Wc = W ;
20:   return W
```



```
21:   Procedure NewAck(ack)
22:   if ack.seq > lastUpdateSeq then
23:      W = ComputeWind(MeasureInflight(ack), True);
24:      lastUpdateSeq = snd_nxt;
25:   else
26:      W = ComputeWind(MeasureInflight(ack), False);
27:   R = W/T; L = ack.L;
```

The above illustrates the overall process of CC at the sender side for a single flow. Each newly received ACK message triggers the procedure NewACK at Line 21. At Line 22, the variable lastUpdateSeq is used to remember the first packet sent with a new W c , and the sequence number in the incoming ACK should be larger than lastUpdateSeq to trigger a new sync betweenW c andW (Line 14-15 and 18-19). The sender also remembers the pacing rate and current INT information at Line 27. The sender computes a new window size W at Line 23 or Line 26, depending on whether to update W c , with function MeasureInflight and ComputeWind. Function MeasureInflight estimates normalized inflight bytes with Eqn (2) at Line 5. First, it computes txRate of each link from the current and last accumulated transferred bytes txBytes and timestamp ts (Line 4). It also uses the minimum of the current and last qlen to filter out noises in qlen (Line 5). The loop from Line 3 to 7 selects maxi(Ui) in Eqn. (3). Instead of directly using maxi(Ui), we use an EWMA (Exponentially Weighted Moving Average) to filter the noises from timer inaccuracy and transient queues. (Line 9). Function ComputeWind combines multiplicative increase/ decrease (MI/MD) and additive increase (AI) to balance the reaction speed and fairness. If a sender finds it should increase the window size, it first tries AI for maxStage times with the stepWAI (Line 17). If it still finds room to increase after maxStage times of AI or the normalized inflight bytes is above, it calls Eqn (4) once to quickly ramp up or ramp down the window size (Line 12-13).

5. Configuration Parameters

HPCC++ has three easy-to-set parameters: eta, maxStagee, and W_ai. eta controls a simple tradeoff between utilization and transient queue length (due to the temporary collision of packets caused by their random arrivals, so we set it to 95% by default, which only loses 5% bandwidth but achieves almost zero queue. maxStage controls a simple tradeoff between steady state stability and the speed to reclaim free bandwidth. We find maxStage = 5 is conservatively large for stability, while the speed of reclaiming free bandwidth is still much faster than traditional additive increase, especially in high bandwidth networks. W_ai controls the tradeoff between the maximum number of concurrent flows on a link that can sustain near-zero

queues and the speed of convergence to fairness. Note that none of

the three parameters are reliability-critical.

HPCC++'s design intentionally brings advantages to short-lived flows, by allowing flows starting at line-rate and the separation of utilization convergence and fairness convergence. HPCC++ achieves fast utilization convergence to mitigate congestion in almost one round-trip time, while allows flows to gradually converge to fairness. This design choice is especially helpful for the workload of datacenter applications, where the majority of flows are short and latency-sensitive. Normally we set a very small W_ai to support a large number of concurrent flows on a link, because slower fairness is not critical. A rule of thumb is to set W_ai = W_init*(1-eta) / N where N is the expected or receiver reported maximum number of concurrent flows on a link. The intuition is that the total additive increase every round (N*W_ai) should not exceed the bandwidth headroom, and thus no queue forms. Even if the actual number of concurrent flows on a link exceeds N, the CC is still stable and achieves full utilization, but just cannot maintain zero queues.

6. Design Enhancement and Implementation

The basic design of HPCC++, i.e. HPCC, as described above is to add INT information into every date packet to response congestion as soon as the very first packet observing the network congestion. This is especially helpful to reduce the risk of severe congestion in incast scenario at the first round-trip time. In addition, original HPCC's algorithm of introducing Wc is for the purpose of solving the over-reaction issue from using this per-packet response.

Alternatively, the INT information needs not to be added to every data packet to reduce the overhead. Switches can generate INT less frequently, e.g., once per RTT or upon congestion happening.

6.1. HPCC++ Guidelines

To ensure network stability, HPCC++ establishes a few guidelines for different implementations:

- The algorithm should commit the window/rate update at most once per round-trip time, similar to the procedure of updating Wc.

- To support different workloads and to properly set W_ai, HPCC++ allows the option to incorporate mechanisms to speed up the fairness convergence.

- The switch should capture INT information that includes link load (txBytes, ts, qlen) and link spec (switch ID, egress port ID, port

speed) at the egress port. Note, each switch should record all those information at the single snapshot to achieve a precise link load estimate.

- HPCC++ can optionally use a probe packet to query the INT information. Thereby, the probe packets should take the same routing path and QoS queueing with the data packets.

As long the above guidelines are met, this document does not mandate a particular INT header format or encapsulation, which are orthogonal to the HPCC++ algorithms described in this document. The algorithm can be implemented with a choice of in-band network telemetry [P4-INT], [I-D.ietf-ippm-ioam-data], [I-D.ietf-kumar-ippm-ifa].

6.2. Receiver-based HPCC

Note that the window/rate calculation can be implemented at either the data sender or the data receiver. If the ACK packets already exist for reliability purpose, the INT information can be echoed back to the sender via ACK self-clocking. Not all ACK packets need to carry the INT information. To reduce the Packet Per Second (PPS) overhead, the receiver may examine the INT information and adopt the technique of delayed ACKs that only sends out an ACK for a few of received packets. In order to reduce PPS even further, one may implement the algorithm at the receiver and feedback the calculated window in the ACK packet once every RTT.

The receiver-based algorithm, Rx-HPCC, is based on int.L, which is the INT information in the packet header. The receiver performs the same functions except using int.L instead of ack.L. The new function NewINT(int.L) is to replace NewACK(int.L)

```
28:   Procedure NewINT(int.L)
29:   if now > (lastUpdateTime + T) then
30:      W = ComputeWind(MeasureInflight(int), True);
31:      send_ack(W)
32:      lastUpdateTime = now;
33:   else
34:      W = ComputeWind(MeasureInflight(int), False);
```

Here, since the receiver does not know the starting sequence number of a burst, it simply records the lastUpdateTime. If time T has passed since lastUpdateTime, the algorithm would recalcuate Wc as in Line 30 and send out the ACK packet which would include W informtion. Otherwise, it would just update W information locally. This would reduce the amount of traffic that needs to be feedback to the data

sender.

Note that the receiver can also measure the number of outstanding flows, N, if the last hop is the congestion point and use this information to dynamically adjust W_ai to achieve better fairness. The improvement would allow flows to quickly converge to fairness without causing large swings under heavy load.

6.3. Switch-side Optimizations

Switches can potentially generate and send separate packets containing INT information (aka INT response packets) directly back to the data senders so that they can slow down as soon as possible. This fast feedback and reaction can further reduce buffer size consumption upon heavy incast. Switches can consider the level of congestion to decide when to trigger direct INT responses. A simple bloom-filter and timer can be used at switches to avoid sending a burst of INT responses to the same sender. An INT response packet must carry the sequence number of the original data packet, so that the sender can correctly correlate the INT response with the data packet triggered the INT response.

One may optimize the INT header overhead by implementing a simple subscription-based INT. The data senders may simply use a different DSCP codepoint or a flag bit in the INT instruction header to indicate INT subscription. (We expect future INT specs to support such a subscription service.) The senders can selectively subscribe to INT on a per-packet basis to control the INT data overhead. While forwarding INT-subscribed data packets, the switches can monitor the level of congestion and conditionally generate separate INT responses as described above. The INT responses can be directly sent back to the senders or to the receivers depending on which version of HPCC++ algorithm (sender-based or receiver-based) is used in the network.

7. Reference Implementations

A prototype of HPCC++ in commodity NICs with FPGA programmability is implemented to realize the CC algorithm and commodity switching ASICs with P4 programmability to realize a standard INT feature.

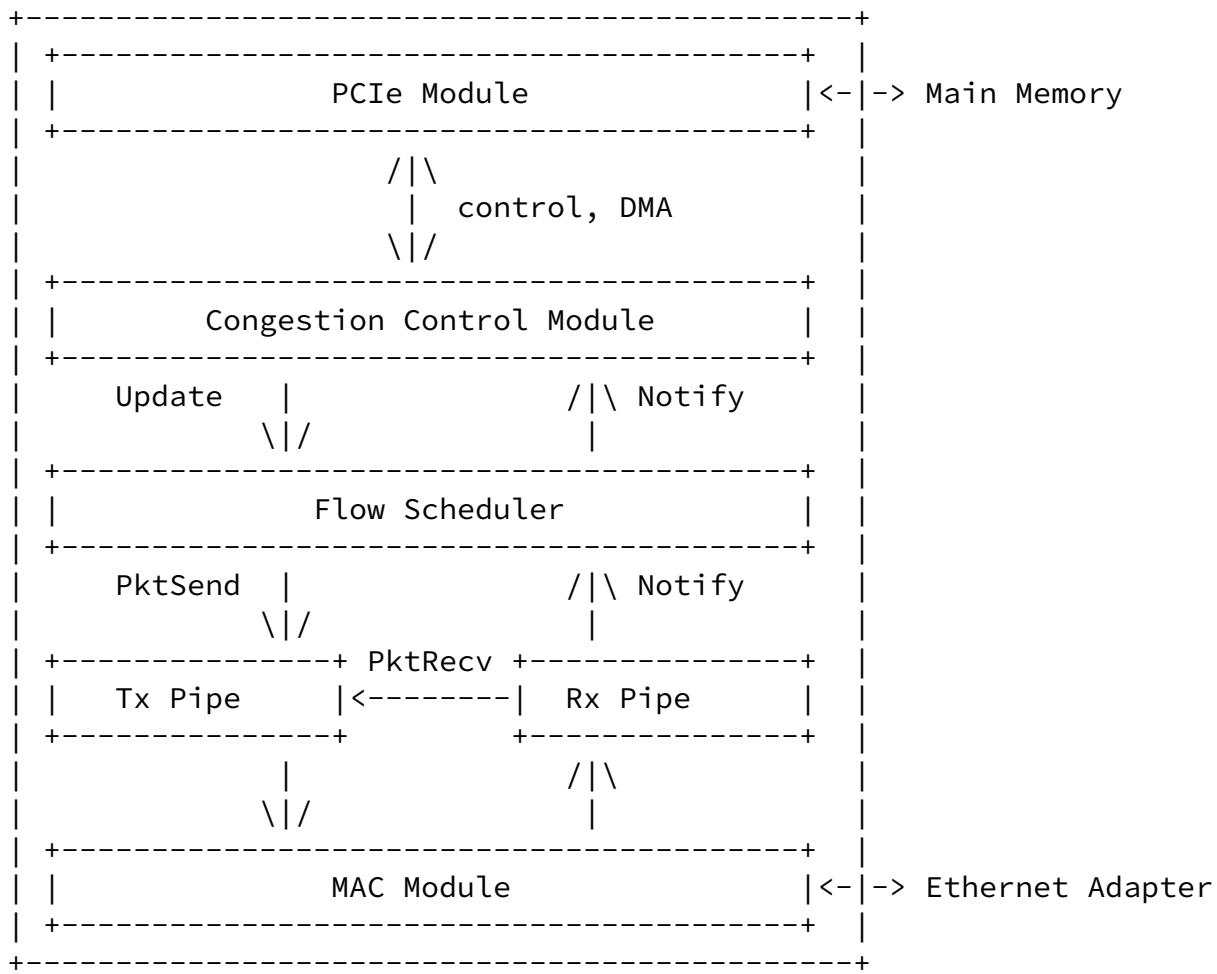

Figure 4: Overview of NIC Implementation

7.1. INT padding at switches

HPCC++ only relies on packets to share information across senders, receivers, and switches. HPCC++ is open to a variety of INT format standards. Inside a data center, the path length is often no more than 5 hops. The overhead of the INT padding for HPCC++ is low.

7.2. Congestion control at NICs

Figure 4 shows HPCC++ implementation on a commodity programmable NIC. The NIC provides an FPGA chip which is connected to the main memory with a vendor-provided PCIe module and the Ethernet adapter with a vendor-provided MAC module. Sitting between the PCIe and MAC modules, HPCC++ modules realize both sender and receiver roles.

The Congestion Control (CC) module implements the sender side CC algorithm. It receives ACK events which are generated from the RX pipeline, adjusts the sending window and rate, and stores the new

sending window and rate for the flow of the current ACK in the flow scheduler via an Update event.

The flow scheduler paces flow rates with a credit-based mechanism. Specifically, it scans through all the flows in a round-robin manner and assigns credit to each flow proportional to its current pacing rate. It also maintains the current sending window size and unacknowledged packets for active flows. If a flow has accumulated sufficient credits to send one packet and the flow's sending window permits, the flow scheduler invokes a PktSend event to TX pipe.

The TX pipe implements IB/UDP/IP stacks for running in RoCEv2. It maintains the flow context for each of concurrent flows, including 5-tuples, the packet sequence number (PSN), destination QP (queue pair), etc. Once it receives the PktSend event with QP ID from the flow scheduler, it generates the corresponding packet and delivers to the MAC module.

The RX pipe parses the incoming packets from the MAC module and generates multiple events to other HPCC++ modules. (1) On receiving a data packet, the RX pipe extracts its flow context and invokes a PktRecv event to the TX pipe to formulate a corresponding ACK packet. If the packet is out-of-sequence (OOS), the TX pipe sends a NAK instead. (2) On receiving an ACK packet, the RX pipe extracts the network status from the packet and passes it to the CC module via the

flow scheduler. (3) On receiving a NAK, the RX pipe notifies the TX pipe to start go-back-to-N retransmission. (4) On receiving a control packet with an RDMA operation, the RX pipe notifies the flow scheduler to create a flow with a new QP ID, or remove an existing flow. Currently, HPCC++ supports two operations: RDMA WRITE and RDMA READ. We leave the full support of IB verbs as future work.

8. IANA Considerations

This document makes no request of IANA.

9. Security Considerations

The rate adaptation mechanism in HPCC++ relies on feedback from the data receiver. As such, it is vulnerable to attacks where feedback messages are hijacked, replaced, or intentionally injected with misleading information resulting in denial of service, similar to those that can affect TCP. It is therefore RECOMMENDED that the notification feedback message is at least integrity checked. In addition, [I-D.ietf-avtcore-cc-feedback-message] discusses the potential risk of a receiver providing misleading congestion feedback information and the mechanisms for mitigating such risks.

10. Acknowledgments

The authors would like to thank ... for their valuable review comments and helpful input to this specification.

11. Contributors

The following individuals have contributed to the implementation and evaluation of the proposed scheme, and therefore have helped to validate and substantially improve this specification: Pedro Y. Segura, Roberto P. Cebrian, Robert Southworth and Malek Musleh.

12. References

12.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119,

DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

12.2. Informative References

[I-D.ietf-avtcore-cc-feedback-message]
Sarker, Z., Perkins, C., Singh, V., and M. Ramalho, "RTP Control Protocol (RTCP) Feedback for Congestion Control", draft-ietf-avtcore-cc-feedback-message-06 (work in progress), March 2020.

[I-D.ietf-ippm-ioam-data]
"Data Fields for In-situ OAM", March 2020, <https://tools.ietf.org/html/draft-ietf-ippm-ioam-data-09>.

[I-D.ietf-kumar-ippm-ifa]
"Inband Flow Analyzer", February 2019, <https://tools.ietf.org/html/draft-kumar-ippm-ifa-01>.

[P4-INT] "In-band Network Telemetry (INT) Dataplane Specification, v2.0", February 2020, <https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_0.pdf>.

[SIGCOMM-HPCC]
Li, Y., Miao, R., Liu, H., Zhuang, Y., Fei Feng, F., Tang, L., Cao, Z., and M. Zhang, "HPCC: High Precision Congestion Control", ACM SIGCOMM Beijing, China, August 2019.

[Zhu-SIGCOMM2015]
Zhu, Y., Eran, H., Firestone, D., Guo, C., Lipshteyn, M., Liron, Y., Padhye, J., Raindel, S., Yahia, M., and M. Zhang, "Congestion Control for Large-Scale RDMA Deployments", ACM SIGCOMM London, United Kingdom, August

2015.

Authors' Addresses

Hongqiang H. Liu
Alibaba Group
108th Ave NE, Suite 800
Bellevue, WA 98004
USA

Email: hongqiang.liu@alibaba-inc.com

Rui Miao
Alibaba Group
525 Almanor Ave, 4th Floor
Sunnyvale, CA 94085
USA

Email: miao.rui@alibaba-inc.com

Rong Pan
Intel, Corp.
2200 Mission College Blvd.
Santa Clara, CA 95054
USA

Email: rong.pan@intel.com

Jeongkeun Lee
Intel, Corp.
4750 Patrick Henry Dr.
Santa Clara, CA 95054

USA

Email: jk.lee@intel.com

Changhoon Kim
Intel Corporation
4750 Patrick Henry Dr.
Santa Clara, CA 95054
USA

Email: chang.kim@intel.com