

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: 10 June 2022

R. Miao
H. Liu
Alibaba Group
R. Pan
J. Lee
C. Kim
Intel Corporation
B. Gafni
Y. Shpigelman
Mellanox Technologies, Inc.
J. Tantsura
Microsoft Corporation
7 December 2021

HPCC++: Enhanced High Precision Congestion Control
draft-miao-iccr-g-hpccplus-01

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 inband telemetry to obtain precise link load information and controls traffic precisely. By addressing challenges such as delayed signaling during congestion and overreaction to the congestion signaling using inband and granular telemetry, HPCC++ can quickly converge to utilize all the available 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 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-Draft

HPCC++

December 2021

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 10 June 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the [Trust Legal Provisions](#) and are provided without warranty as described in the Revised BSD License.

Table of Contents

1.	Introduction	3
2.	Terminology	4
3.	System Overview	4
4.	HPCC++ Algorithm	5
4.1.	Notations	5
4.2.	Design Functions and Procedures	6
5.	Configuration Parameters	8
6.	Design Enhancement and Implementation	8
6.1.	HPCC++ Guidelines	9
6.2.	Receiver-based HPCC	9
7.	Reference Implementations	10
7.1.	Inband telemetry padding at the network elements	10
7.2.	Congestion control at NICs	10
8.	IANA Considerations	12
9.	Discussion	12
9.1.	Internet Deployment	12
9.2.	Switch-assisted congestion control	12
9.3.	Work with transport protocols	13
9.4.	Work with QoS queuing	13
10.	Acknowledgments	14

11. Contributors	14
12. References	14
12.1. Normative References	14
12.2. Informative References	14
Authors' Addresses	15

[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 as described in [[Zhu-SIGCOMM2015](#)], offloading network stacks into hardware is an inevitable direction in high-speed networks. As an example, large-scale networks with RDMA (remote direct memory access) often uses hardware-offloading solutions. In some cases, the RDMA networks still face fundamental challenges to reconcile low latency, high bandwidth utilization, and high stability.

This document describes a new congestion control 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 signaled through inband telemetry 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 inband telemetry enables HPCC++ to address the limitations in current congestion control schemes. First, HPCC++ senders can quickly ramp up flow rates for high utilization and ramp down flow rates for congestion avoidance. Second, HPCC++ senders can quickly adjust the flow rates to keep each link's output 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 inband telemetry 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.

This document describes the architecture changes in switches and end-hosts to support the needed transmission of inband telemetry and its consumption, that improves the efficiency in handling network congestion.

[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 traverse of the packet from the sender to the receiver, each switch along the path inserts inband telemetry that reports the current state of the packet's egress port, including timestamp (ts), queue length (qLen), transmitted bytes (txBytes), and the link bandwidth capacity (B), together with switch_ID and port_ID. When the receiver gets the packet, it may copy all the inband telemetry recorded from the network 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 may calculate the flow rate based on the inband telemetry information and feedback the calculated rate back to the sender. The notification 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.

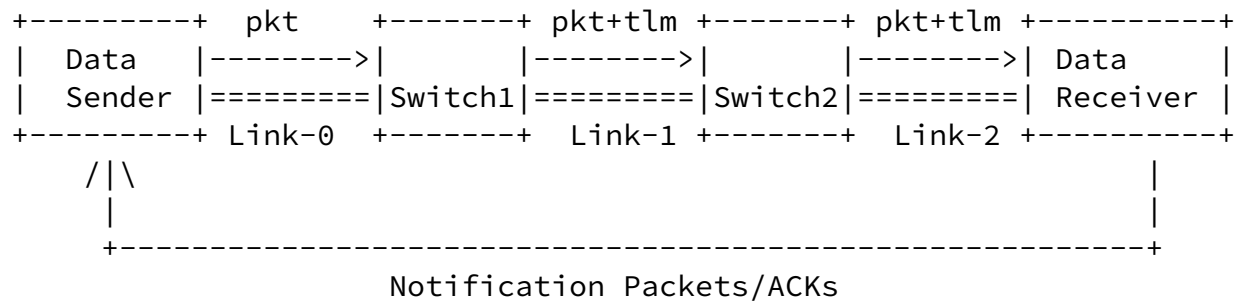


Figure 1: System Overview (tlm=inband telemetry)

- * Data sender: responsible for controlling inflight bytes. HPCC++ is a window-based congestion control scheme that controls the number of inflight bytes. The inflight bytes mean the amount of data that have been sent, but not acknowledged by 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 $\text{inflight} = \text{rate} * T$ where T is the base propagation RTT. The rate can be calculated locally or obtained from the notification packet. The sender may further use the data pacing mechanism, potentially implemented in hardware, to limit the rate accordingly.

- * Network nodes: responsible of inserting the inband telemetry information to the data packet. The inband telemetry information reports the current load of the packet's egress port, including timestamp (ts), queue length (qLen), transmitted bytes (txBytes), and link bandwidth capacity (B). Besides, the inband telemetry contains switch_ID and port_ID to identify a link.
- * Data receiver: responsible for either reflecting back the inband telemetry information in the data packet or calculating the proper flow rate based on network congestion information in inband telemetry 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
$qlen$	Telemetry info: link j queue length
$txRate$	Telemetry info: link j output rate
ts	Telemetry info: timestamp
$txBytes$	Telemetry info: 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:     txRate =  $\frac{\text{ack.L}[i].\text{txBytes} - L[i].\text{txBytes}}{\text{ack.L}[i].\text{ts} - L[i].\text{ts}}$  ;
5:     u' =  $\frac{\min(\text{ack.L}[i].\text{qlen}, L[i].\text{qlen})}{\text{ack.L}[i].B * T} + \frac{\text{txRate}}{\text{ack.L}[i].B}$  ;
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 >= maxStage then
13:     Wc
14:     W =  $\frac{U}{\text{eta}}$  + W_ai;
15:     if updateWc then
16:       incStage = 0; Wc = W ;
17:   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 between W_c and W (Line 14-15 and 18-19). The sender also remembers the pacing rate and current inband telemetry 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 $\max_i(U_i)$ in Eqn. (3). Instead of directly using $\max_i(U_i)$, 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).

HPCC++ has three easy-to-set parameters: η , maxStage , and W_{ai} . η 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 $\text{maxStage} = 5$ 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 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 feature of HPCC++ is especially helpful for the workload of datacenter applications, where flows are usually 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 $W_{ai} = W_{init} \cdot (1 - \eta) / N$ 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 ($N \cdot W_{ai}$) 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 inband telemetry information into every data packet to react to 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 scenarios at the first round-trip time. In addition, original HPCC's algorithm introduction of W_c is for the purpose of solving the over-reaction issue from using this per-packet response.

Alternatively, the inband telemetry information needs not to be added to every data packet to reduce the overhead. Switches can attach inband telemetry less frequently, e.g., once per RTT or upon congestion occurrence.

[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 W_c .
- * 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 inband telemetry information that includes link load (txBytes, qlen, ts) and link spec (switch_ID, port_ID, B) 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 use a probe packet to query the inband telemetry 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 inband telemetry header format or encapsulation, which are orthogonal to the HPCC++ algorithm described in this document. The algorithm can be implemented with a choice of inband telemetry protocols, such as in-band network telemetry [[P4-INT](#)], IOAM [[I-D.ietf-ippm-ioam-data](#)], IFA [[I-D.ietf-kumar-ippm-ifa](#)] and others. In fact, the emerging inband telemetry protocols can inform the evolution for a broader range of protocols and network functions, where this document leverages the trend to propose the architecture change to support HPCC++ algorithm.

[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 inband telemetry information can be echoed back to the sender via ACK self-clocking. Not all ACK packets need to carry the inband telemetry information. To reduce the Packet Per Second (PPS) overhead, the receiver may examine the inband telemetry 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.

Internet-Draft

HPCC++

December 2021

The receiver-based algorithm, Rx-HPCC, is based on `int.L`, which is the inband telemetry 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 recalculate `Wc` as in Line 30 and send out the ACK packet which would include `W` information. 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 `Wai` to achieve better fairness. The improvement would allow flows to quickly converge to fairness without causing large swings under heavy load.

[7.](#) Reference Implementations

A prototype of HPCC++ is implemented in NICs to realize the congestion control algorithm and in switches to realize the inband telemetry feature.

[7.1.](#) Inband telemetry padding at the network elements

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

padding for HPCC++ is considered to be low.

7.2. Congestion control at NICs

Figure 4 shows HPCC++ implementation on a NIC. The NIC provides an HPCC++ module that resides on the data path of the NIC, HPCC++ modules realize both sender and receiver roles.

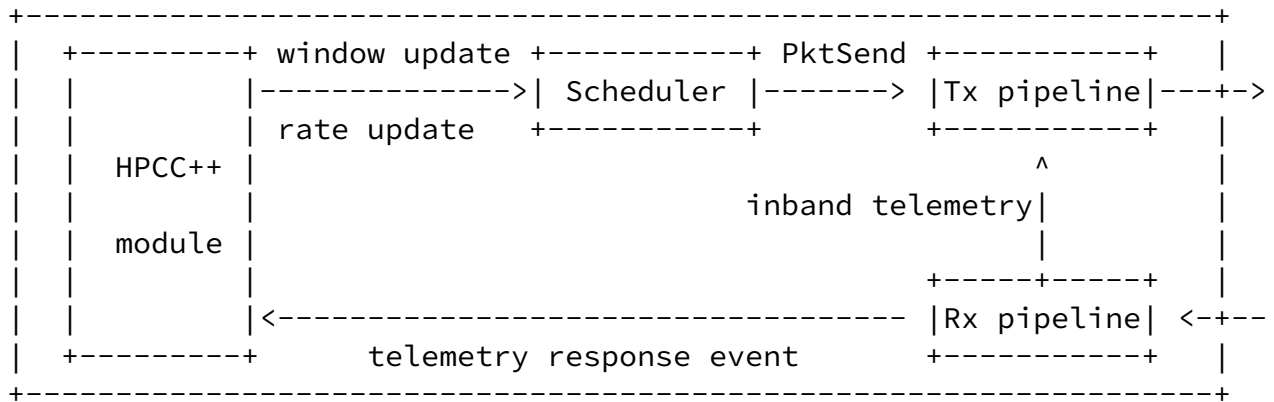


Figure 4: Overview of NIC Implementation

1. Sender side flow

The HPCC++ module running the HPCC CC algorithm in the sender side for every flow in the NIC. Flow can be defined by some transport parameters including 5-tuples, destination QP (queue pair), etc. It receives inband telemetry response events per flow which are generated from the RX pipeline, adjusts the sending window and rate, and update the scheduler on the rate and window of the flow.

The scheduler contains a pacing mechanism that determine the flow rate by the value it got from the algorithm. It also maintains the current sending window size for active flows. If the pacing mechanism and the flow's sending window permits, the scheduler invokes for the flow a PktSend command to TX pipeline.

The TX pipeline implements packet processing. Once it receives the PktSend event with flow ID from the scheduler, it generates the

corresponding packet and delivers to the Network. If a sent packet should collect telemetry on its way the TX pipeline may add indications/headers that triggers the network elements to add telemetry data according to the inband telemetry protocol in use. The telemetry can be collected by the data packet or by dedicated prob packets generated in the TX pipeline.

The RX pipe parses the incoming packets from the network and identifies whether telemetry is embedded in the parsed packet. On receiving a telemetry response packet, the RX pipeline extracts the network status from the packet and passes it to the HPCC++ module for processing. A telemetry response packet can be an ACK containing inband telemetry, or a dedicated telemetry response prob packet.

2. Receiver side flow

On receiving a packet containing inband telemetry, the RX pipeline extracts the network status, and the flow parameters from the packet and passes it to the TX pipeline. The packet can be a data packet containing inband telemetry, or a dedicated telemetry request prob packet. The Tx pipeline may process and edit the telemetry data, and then sends back to the sender the data using either an ACK packet of the flow or a dedicated telemetry response packet.

[8.](#) IANA Considerations

This document makes no request of IANA.

[9.](#) Discussion

[9.1.](#) Internet Deployment

Although the discussion above mainly focuses on the data center environment, HPCC++ can be adopted at Internet at large. There are several security considerations one should be aware of.

There may rise privacy concern when the telemetry information is conveyed across Autonomous Systems (ASes) and back to end-users. The link load information captured in telemetry can potentially reveal the provider's network capacity, route utilization, scheduling policy, etc. Those usually are considered to be sensitive data of

the network providers. Hence, certain action may take to anonymize the telemetry data and only convey the relative ratio in rate adaptation across ASes without revealing the actual network load.

Another consideration is the security of receiving telemetry information. The rate adaptation mechanism in HPCC++ relies on feedback from the network. 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-avtc core-cc-feedback-message](#)] discusses the potential risk of a receiver providing misleading congestion feedback information and the mechanisms for mitigating such risks.

[9.2](#). Switch-assisted congestion control

HPCC++ falls in the general category of switch-assisted congestion control. However, HPCC++ includes a few unique design choices that are different from other switch-assisted approaches.

- * First, HPCC++ implements a primal-mode algorithm that requires only the ``write-to-packet'' operation from switches, which has already been supported by telemetry protocols like INT [[P4-INT](#)] or IOAM [[I-D.ietf-ippm-ioam-data](#)]. Please note that this is very different from dual-mode algorithms such as XCP [[Katabi-SIGCOMM2002](#)] and RCP [[Dukkipati-RCP](#)], where switches take an actively role in determining flows' rates.
- * Second, HPCC++ achieves a fast utilization convergence by decoupling it from fairness convergence, which is inspired by XCP.
- * Third, HPCC++ enables the switch-guided multiplicative increase (MI) by defining the ``inflight byte'' to quantify the link load. The inflight byte tells both the underload and overload of the link precisely and thus it allows the flow to increase/decrease the rate multiplicatively and safely. By contrast, traditional approaches of using the queue length or RTT as the feedback cannot guide the rate increase and instead have to rely on additive

increase (AI) with heuristics. As the link speed continues to grow, this becomes increasingly slow in reclaiming the unused bandwidth. Besides, queue-based feedback mechanisms subject to latency inflation.

- * Last, HPCC++ uses TX rate instead of RX rate used by XCP and RCP. As detailed in [[SIGCOMM-HPCC](#)], we view the TX rate is more precise because RX rate and queue length are overlapped and thus it causes oscillation.

[9.3.](#) Work with transport protocols

HPCC++ can be adopted as the CC algorithm by a wide range of transport protocols such as TCP and UDP, as well as others that may run on top of them, such as iWARP, RoCE etc. It requires to have the window limit and congestion feedback through ACK self-clocking, which naturally conforms to the paradigm of TCP design. With that, HPCC++ introduces a scheme to measure the total inflight bytes for more precise congestion control. To run in UDP, some modifications need to be done to enforce the window limit and collect congestion feedback via probing packets, which is incremental.

[9.4.](#) Work with QoS queuing

Under the use of QoS (Quality of service) priority queuing in switches, the length of flow's own queue cannot tell the actual queuing time and the exact extent of congestion. Although general approaches for running congestion control with QoS queuing are out of the scope of this document, we provide a few hints for HPCC++ running friendly with QoS queuing. In this case, HPCC++ can leverage the

packet sojourn time (the egress timestamp minus the ingress timestamp) instead of the queue length to quantify the packet's actual queuing delay. In addition, the operators typically use the Deficit Weighted Round Robin (DWRR) instead of the strict priority (SP) as their QoS scheduling to prevent traffic starvation. DWRR provides a minimum bandwidth guarantee for each queue so that HPCC++ can leverage it for precise rate update to avoid congestion.

[10.](#) Acknowledgments

The authors would like to thank ICCRG members 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. A. Ramalho, "RTP Control Protocol (RTCP) Feedback for Congestion Control", Work in Progress, Internet-Draft, [draft-ietf-avtcore-cc-feedback-message-09](#), 2 November 2020, <<https://www.ietf.org/archive/id/draft-ietf-avtcore-cc-feedback-message-09.txt>>.
- [Katabi-SIGCOMM2002]
Katabi, D., Handley, M., and C. Rohrs, "Congestion Control for High Bandwidth-Delay Product Networks", ACM SIGCOMM Pittsburgh, Pennsylvania, USA, October 2002.

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

Deployments", ACM SIGCOMM London, United Kingdom, August 2015.

[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>.

[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>>.

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

[Dukkipati-RCP]
Dukkipati, N., "Rate Control Protocol (RCP): Congestion control to make flows complete quickly.", Stanford University , 2008.

Authors' Addresses

Rui Miao
Alibaba Group
525 Almanor Ave, 4th Floor
Sunnyvale, CA 94085
United States of America

Email: miao.rui@alibaba-inc.com

Hongqiang H. Liu
Alibaba Group
108th Ave NE, Suite 800
Bellevue, WA 98004
United States of America

Email: hongqiang.liu@alibaba-inc.com

Rong Pan
Intel, Corp.
2200 Mission College Blvd.
Santa Clara, CA 95054
United States of America

Email: rong.pan@intel.com

Jeongkeun Lee
Intel, Corp.
4750 Patrick Henry Dr.
Santa Clara, CA 95054
United States of America

Email: jk.lee@intel.com

Changhoon Kim
Intel Corporation
4750 Patrick Henry Dr.
Santa Clara, CA 95054
United States of America

Email: chang.kim@intel.com

Barak Gafni
Mellanox Technologies, Inc.
350 Oakmead Parkway, Suite 100
Sunnyvale, CA 94085
United States of America

Email: gbarak@mellanox.com

Yuval Shpigelman
Mellanox Technologies, Inc.
Haim Hazaz 3A
Netanya 4247417
Israel

Email: yuvals@nvidia.com

Internet-Draft

HPCC++

December 2021

Jeff Tantsura
Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399
United States of America

Email: jefftantsura@microsoft.com

