STRIDE: STT-Based Reliable Data Transport Protocol for reconfigurable Cloud

[1]Author 1, [2]author, and [3]author
[1]Dept. Computer Science and Engineering, University A,
[2]Dept. Computer Science, University B,
[3]Dept. Engineering, University C.
{author 1, author 2, and author 3}@email.edu

***Abstract*— In a recent development, reconfigurable clouds become a viable solution to overcome practical problems in clouds, such as scalability, delay, etc., by offloading computation tasks to hardware like FPGA. Several existing techniques, such as TCP/IP Offload Engine (TOE) and Lightweight Transport Layer (LTL), are still not implementable in real-world deployment due to large overhead or stringent dependency of the underlying network. In this paper, we propose STRIDE, a novel delay based inter-FPGA data communication protocol to provide reliable end-to-end communication which addresses practical problems in deployment. In our design, STRIDE leverages FPGA's strength, such as precise timestamping, to make more accurate measurement on end-to-end queuing delay and deliver more precise control in managing traffic in cloud. We implement STRIDE on a FPGA-based network experimental platform and demonstrate that STRIDE reduces various hardware resources consumption by 36% to 49% compared to TOE. Additionally, it also improves flow completion time in comparison to TOE by 55%; and improve average bandwidth utilization rate by over 90% while achieving stability.**

I. INTRODUCTION

Semiconductor industries have been struggling to keep up with the rapidly growing demand for computing power to support big data applications [1, 2], and have reached their limit in overcoming practical problems such as scalability, delay, etc. Reconfiguration computing offers a great potential to resolve these problems through utilizing reconfigurable chips such as Field-Programmable Gate Arrays (FPGA) [4]. This technology provides capabilities to achieve high throughput, more predictable latency, programmability, low power consumption, and precise timestamping, allowing servers in clouds to accelerate on demand which results in higher system utilization (e.g. bandwidth, CPU, etc.). Another advantage of FPGA is that it can be quickly inserted into existing (legacy) system to enhance system performance without altering the system ("bump-in-the-wire") [4,5,6]. Nevertheless, despite the benefits of FPGA, there is little study to effectively exploit these benefits in implementation.

One way to exploit FPGA is to offload computation task of the transport layer protocol (e.g. TCP, etc.) to FPGA. *TCP/IP Offload Engine* (TOE) [11], a pioneering work on offloading protocol to FPGA, allows applications to offload data without copying and performing interrupt handling in the kernel. This reduces processing delay in TCP/IP stack [11]. Additionally, TOE is compatible to legacy TCP/IP and does not require switch support. However, due to implementation complexity and resource consumption, TOE does not fully offload TCP into FPGA, i.e. some parts of processing are still done in kernel [12]. Later in this paper, we provide a discussion on implementation challenges in full offloading TCP (and any existing transport layer protocols) to FPGA. Another work of the same theme, *Lightweight Transport Layer* (LTL) [4] is proposed to provide low latency connection while mitigating packet drop in inter-FPGA end-to-end connection settings. To address reliability, the authors propose a conceptual level congestion control scheme similar to DC-QCN [13]. The mechanism utilizes priority flow-control (PFC) to slow down incoming traffic to a switch when congestion is detected. LTL also employs ECN to signal end-hosts about a congestion. However, the authors do not provide detailed descriptions on how their congestion control scheme is implemented and offloaded into FPGA.

We uncover there is design incompatibility between existing CC schemes (TCP and TIMELY) and hardware design. This incompatibility not only limits the full utilization of FPGA, it may also result in system performance degradation caused by these following phenomenon occurring at sender's end: (i) Multiplication and division operation required for computing congestion window (*cwnd*) takes up the majority of the processing resource in FPGA. (ii) Mechanism used to keep track of packet state (e.g. which packets has been ACK and transmitted, or not ACK and not transmitted, or retransmitted, etc.) occupies a large portion of memory in FPGA. Moreover, memory allocated for multiplication and division operations further increases the memory cost. The detail on these observations are discussed in Section II.

Therefore, the question is: *How to design a CC scheme that is compatible with hardware design such as FPGA*? To address this question, we first must consider a plug-and-play solution that can be fully offloaded to FPGA. This is because in order for a solution to be deployable ready, firstly the solution must be compatible with legacy commodity switches that are commonly used in cloud (or datacenter network). Secondly, datacenter network is typically made up of tens of thousands of switches and servers [38]. For this reason, from network management and maintenance perspective, the solution must require minimal network reconfiguration requires only minimal switch support. Driven by these two practical motivations, we consider delay-based scheme instead of ECN and PFC based scheme. Moreover, in order to design an effective scheme that can be offloaded to FPGA, our CC scheme must be able to achieve low CPU, memory, and power consumption. At last, our design must be able to

achieve basic CC mechanisms, such as end-to-end reliability, congestion window adaptation, etc.

In this paper, we present our CC scheme, STRIDE (Single-TRIp-time-based reliable Data transport protocol for the rEconfigurable cloud) that is specifically designed for reconfigurable cloud. STRIDE is a lightweight reliable data transport protocol for inter-FPGA communication that imposes lower overhead on resource utilization and bandwidth. To further lower computation cost and memory usage in hardware, we propose bit-shifting techniques to compute and adjust transmission rate, instead of multiplication and division operations. This allows the entire computation to be done in FPGA, resulting in quicker responses to congestion.

Additionally, STRIDE also takes advantage of functionalities available in FPGA, such as packet timestamping in hardware, FPGA time precision, etc., which enable STRIDE to accurately measure end-to-end delay. By doing so, STRIDE can measure precise single trip time (STT) delay between two end-hosts as a congestion signal. This strategy is different from most of the existing delay-based CC schemes (e.g. TCP-Vegas [37], TIMELY [15], DX [27], etc.) where they employ a round trip time (RTT) delay as congestion signal. The advantage of STT over RTT is that STT may provide a more precise delay end-to-end measurement, while RTT becomes imprecise when the reverse delay is distorted. Thus, in utilizing STT as congestion signal, STRIDE employs gradient based scheme to provide better control on transmission rate. By taking advantage of functionalities available in FPGA, STRIDE avoids using packet drop as congestion signal, which results in lower packets dropped. Importantly, this makes STRIDE a plug-and-play solution without requiring additional reconfiguration in network (e.g. setting up ECN). Finally, STRIDE also provides basic functionalities for end-to-end reliability, such as ACK scheme, retransmission, three way handshaking initialization, termination connection, etc.

To evaluate STRIDE performance, we build packet procession modules and implement an actual prototype using FPGA Accelerated Switches platform [33, 34], which we then deploy in our testbed. In our testbed experiments, we compare STRIDE to the state of the art (TOE) and demonstrate that STRIDE lowers the flow completion time by at least 55%. This improvement is an outcome from STRIDE experiencing lower packet lost (including ACK packets), delivering more accurate congestion feedbacks from receiver end. The detailed description is provided in the evaluation section. Additionally, STRIDE also reduces computational and memory cost by up to 49.15% and 37.82% respectively when compared to TOE.

Note: More evaluation result…. Here.

This paper is organized as follows. We begin by presenting previous related works and background in section II and III, followed by the details of STRIDE's design in section IV. Following these, section V describes STRIDE detailed implementation. Experiment results are presented in section VI, and concluding remarks in section VII.

II. Related Work

One of the early developments on TCP offloading technology for high-speed network is TCP offload engine (TOE). It allows NIC card to offload the entire TCP/IP stack network controller, reducing the processing overhead in the CPU. Such overheads include connection establishment and termination, TCP checksum, and sliding window for congestion control and reliability [?]. By doing so, TOE reduces CPU and server I/O system overhead and thereby speeds up the processing speed. However, at implementation level, CPU is still required for TCP connection management function to process received and sent packets. For example, every socket created in user space application corresponds to socket and sock structure in kernel space. Thus, every system call eventually has to invoke a function in the kernel. For these reasons, offloading TCP protocol from Linux system core function to a dedicated processing unit (hardware) can be very challenging and complex because TCP connection management depends on functionalities residing in kernel space. There are number of FPGA cluster projects that focus on physical and data link layer [18, 19], but do not consider transport layer such as congestion control scheme due to implementation challenges described above.

An approach providing congestion management to avoid early packet drop, authors of [4] incorporate ECN [20] based DC-QCN scheme [20] and Priority Flow Control (PFC) [?] in their implementation of inter-FPGA network protocol. However, ECN and PFC based scheme requires switch support. Additionally, even with switches that support ECN and PFC, the scheme requires network operator to configure the switches in datacenter, which may not be efficient as a typical datacenter is made up of thousands of switches.

III. Background

Despite benefits that FPGA offers, we discover challenges of implementing a complete offloading congestion control scheme to FPGA. In our experiment, we first attempted to implement TCP and TIMELY in FPGA, and then performed simple experiments in our testbed with two servers connected through two serial switches. Through our endeavors, we learn that it is not only difficult to fully offload a protocol to FPGA, our experiments also show that the system performs poorly. The lessons we learned from implementing a fully offloaded FPGA presented in this section provides the insights to the design of our CC scheme design for inter-FPGA end-to-end connection.

Lesson 1: ***Computational cost***. Offloading the entire TCP stack to FPGA is computationally expensive, caused by the complex mechanism in calculating congestion window size in hardware. Generally, the high computation cost comes from multiplication and division operations. In multiplication operation, for every single partial product generated by multiplying two bits, it requires multiple rounds of bits shifting and addition operation. Moreover, since division involves a number of multiplication operations, the computation cost becomes even more expensive because the computation involves multiplying several factors by the devisor to obtain approximately 1, followed by multiplying

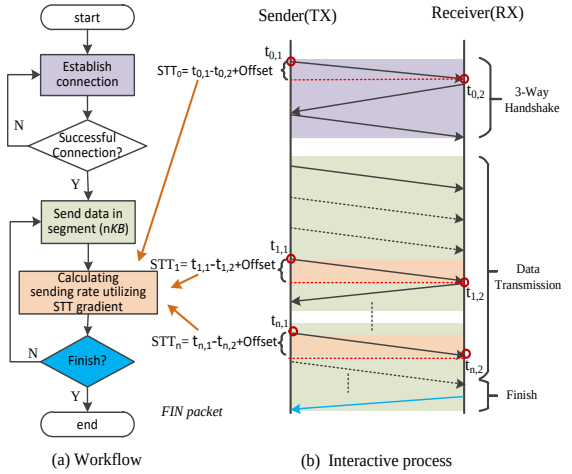
the outcome from previous multiplication operations by the dividend. Therefore, calculating congestion window size becomes computationally expensive, especially for a scheme that requires heavy multiplication and division operations like TIMELY does. The cost worsens when there are a large number of flows transmitting from the same FPGA card.

Lesson 2: ***Memory cost***. Typically a CC scheme needs to keep track of packet status such as ACK, inflight packets, packets that are ready to be transmitted, packets that are not ready for transmission, etc.; these information is stored in the memory inside the FPGA card. Moreover, the scheme also needs to allocate additional memory space to keep track of the partial product from multiplication and division operations. The demands of these two tasks may result in exhausting the memory space.

Therefore, based on lessons learned, we ought to rethink how congestion control scheme should be designed to achieve the objective of complete offloading to FPGA.

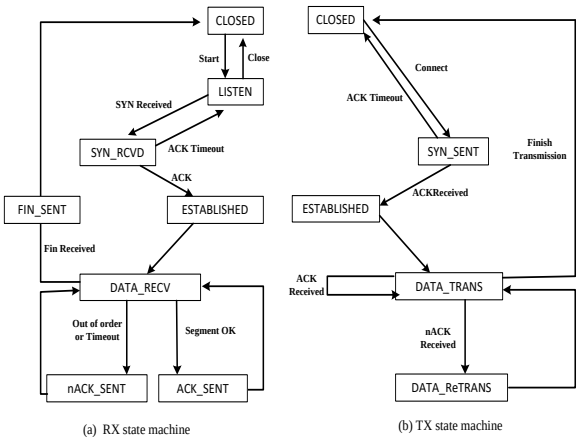
IV. STRIDE Overall Design

In this section, we present an overview of STRIDE by describing its workflow.



(a) Workflow (b) Interactive process

Figure 1 . need to remove offset



(a) RX state machine (b) TX state machine

Fig. 2: STRIDE state machines

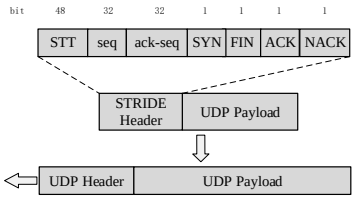


Fig. 3: STRIDE packet structure

As showed in the Figure 1, the data transport of STRIDE consists of 3 stages, which includes connection establishment

A. Overall Design

Packet format. In our design, STRIDE packet information (Figure 3) is encapsulate in UDP packet, allowing STRIDE to be compatible with legacy in network middle box protocol (e.g. NAT, packet filter, etc.).

Establishing and terminating a connection. STRIDE performs TCP's three-way handshake like scheme to establish a connection between TX and RX. Similarly, STRIDE also mimics TCP's connection termination scheme to end a connection.

Congestion Control. Despite the advantageous of FPGA, offloading and implementing existing protocol in FPGA can be very challenging as described above. Thus, we consider FPGA limitation in our solution and design a new congestion control that is implementable and readily deployable. In our design, we design delay based scheme with these following considerations: (i) STRIDE is a plug-and-play solution without requiring switch hardware modification. Unlike ECN based solutions (e.g. DCTCP [?], ECN*[ ], etc.), they require configuring switches, which may not be scalable in the large scale setting. (ii) STRIDE takes advantage of hardware technology such as timestamping [?] which allows us to measure queueing delay accurately. This allows STRIDE to perform a more fine-grained control loop for datacenter congestion control.

B. Single Trip Time and Timestamping

In our implementation, STRIDE takes advantage of hardware timestamping function available in FPGA card to measure STT. Thus, this allow us to calculate STT by estimating the gap between timestamping at TX and RX. Let t_{TX}^{i} and t_{RX}^{i} denote timestamp at TX and RX for packet i, we have $STT(t) = t_i^{TX} - t_i^{RX}$, for $t_i^{TX} > t_i^{RX}$. Traditional, measuring STT requires clock synchronization between two hosts to compensate clock drift, which is not possible with regular NICs. Precision Time Protocol (PTP) enables clock synchronization with sub-microseconds [?]. However, it requires hardware support and is difficult to be deployed in practice [36]. However, measuring with STT with FPGA card is possible because of these following reasons:

(1) FPGA clock precision minimizes occurrences of clock drift in datacenter. In our testbed experiment of two servers connected by two serial 10GB switches, we measure time difference Δt between two observed STT at time t_{i+1} and t_i is $\Delta t = \mathrm{STT}(t+1) - \mathrm{STT}(t)$ and observe throughout 5 minutes experiments that $\Delta t < 1\ nanoseconds$ for every 1

millisecond interval. The error Δt is sufficiently small to provide accurate STT approximation for flows in datacenter, which typically complete in microseconds level [?]. This is because FPGA has a crystal error where the error range is within $\frac{20}{1000000}$, which leads to less than 1 nanosecond error margin.

(2) FPGA clock synchronization is accomplished by assuring the cards are operating at the same frequency, which can be achieved through programing FPGA such that cards have the same cycle [?]

Timestamping is performed in FPGA card by utilizing a counter that counts number of bits belonging to a same packet. This allows STRIDE to perform packet timestamping as soon as the last segment enters buffer. After that, the timestamp value is inserted into the packet by assigning bit in the STT field (Figure 3) in STRIDE header. The stamping is performed at both TX and RX end. To compute STT, RX extracts timestamp value in STT field received from TX and use it to compute STT.

C. Congestion Control

We present a congestion control for inter-FPGA in datacenters that targets low queueing delay, without any form of in-network support. Latency feedback signals the length of time of inflight packets in network. In this section, we present the basic mechanism and design of our new congestion control for inter-FPGA in datacenter, STRIDE.

STRIDE is a rate-based congestion control, and its congestion avoidance follows the gradient-based approach and algorithm, where rate is adjusted according to the rate of STT variation or the gradient.

Algorithm 1: *STRIDE Congestion Control*

1. $STT_{new} = 0$ // Initilization
2. $STT_{min} = min(\{S\})$ // S is a set of observed STTs
3. **Procedure** STRIDE ( )
4. $\Delta STT = (1 - \alpha)\ .\ \Delta STT + \alpha\ (STT_{new} - STT_{old})$
5. $STT_{old} = STT_{new}$
6. **if** $STT_{new} < T_{low}$
7. rate = rate + β // increment with step size $\beta > 0$
8. *return*
9. **if** $STT_{new} \geq T_{high}$
10. Solve eq. (1) with + $k = 3$
11. *return*
12. **if** $T_{high} > new_{stt} > T_{low}$
13. Δ= ΔSTT
14. *normalized_gradient* = 0
15. **while** $\Delta > STT_{min}$ // solving eq. (2)
16. *normalized_gradient*++
17. $\Delta = \Delta - STT_{min}$
18. **end while**
19. **if** *normalized_gradient* < 0
20. rate = rate + $\omega.\beta$ // increase with weighted β
21. **if** *normalized_gradient* > 0
22. rate = rate (1 - *normalized_gradient*)
23. *return*
24. **end procedure**

For each flow, STRIDE maintains a single rate $r(t)$ at time t. Since performing multiplication and division may lead to poor performance, STRIDE relies on shifting to update the transmission rate using an equation below:

$$r(t+1) = r(t)\sum_{i=1}^{k}\frac{1}{2^i}, \quad (1)$$

where k is number of shift required. In our experiment, we set $k = 3$. Eq. (1) is performed when $r(t)$ exceeds predetermined threshold described in algorithm 1.

$$normalized_{gradient} = \frac{\Delta STT}{STT_{min}} \quad (2)$$

ACK response. ACK response performed by RX is similar to TCP acknowledgement scheme. RX sends an ACK packet when the last segment of a packet is received.

How does STRIDE reduce memory usage?

STRIDE Reliability. When RX detects missing packets, RX will response with NACK with a sequence number of the missing packet. Upon receiving NACK message, TX resends the missing packet and the subsequence packets. The idea is similar to GO-back-N protocol [ ]. The advantageous of single NACK packet over TCP duplicate packets (typically 3 duplicate packets) is that handling single packet provides lower computational and memory space cost in FPGA. Furthermore, single notification also speeds up the resend request process than waiting for multiple notifications.

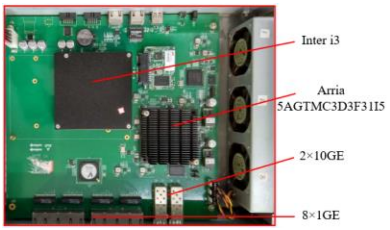


Fig. 4: iRouter board

V. IMPLEMENTATION

In our implementation, we build our solution in an open source FPGA Accelerated switches (FAS) platform [33] [34]. It is an open source project. This platform provides multiple modular interfaces for FPGA developer. The FPGA board (iRouter board depicted in Figure 4) is equipped with Arria 5AGTMC3D3F31I5 and a piece of flash storing the configuration file of the FPGA. It consist of multi-core CPUs

allowing software to communicate with FPGA through the PCIe bus. The line-card board provides eight 1-GigE Ethernet ports and two 10-GigE Ethernet ports.

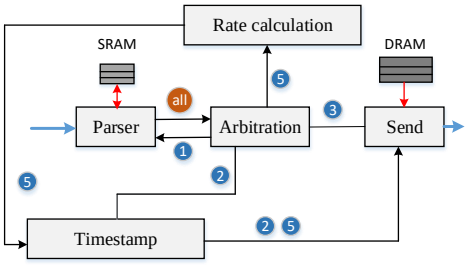


Fig. 5: A variety of packet processing module (remove number?)

Hardware module design. Packet processing operations varies are described in figure 5. When a packet arrive in FPGA card (at TX or RX) and place in SRAM, packet is processed first in the Parser module, then Arbitration module, Rate Calculation module, Timestamp module, and then place the packet in DRAM to be processed at Send module.

From TX's perspective, when packet arriving at FPGA is placed in SRAM to be processed in parser module allowing STRIDE to parse STDE, UDP, IP, and MAC header. Then, the Arbitration module determines packet type (e.g. ACK, NACK, FIN, or SYN packet) and process the packet accordingly. For example, a termination packet arrives at RX's FPGA card and placed in SRAM, the parser module parses the packet header for Arbitration module to extract information on packet type. Then, Arbitration module identifies which of the four fields (SYN, FIN, ACK, NACK) is set to 1. Since it is a FIN packet, Arbitration module detects FIN is set to 1 and proceed with connection termination operation.

Other Arbitration module functions includes: keeping track packet length so packet can be timestamped when the last bits of the packet is processed at TX, detecting packets out of order in RX and then responding it with sending a NACK to TX, retransmitting packet when timeout is detected in TX's end, detecting packet error at RX by checking whether received bits of a packet reach the unit length, creating STRIDE header at TX, inserting timestamp into STRIDE header after timestamping is performed at Timestamp module, fetching transmission rate determined by Rate Calculation module and relaying to Send module.

Timestamping is performed at timestamping modules at both TX and RX when the last packet of the segment of a packet is detected by Arbitration module. FPGA technology allowing STRIDE to obtain STT at 10 nanoseconds accuracy. For each new flow, STT is initialized through STRIDE's SYN and ACK packet.

For the rest of the modules, Rate Calculation module determines transmission rate according to algorithm 1. Send module transmits the number of packet transmitted according to the rate decided in Rate Calculation module. Send module will serve packet stored at DRAM after timestamping in FIFO order.

VI. Evaluation

In this section, we evaluate the performance of STRIDE in our testbed.

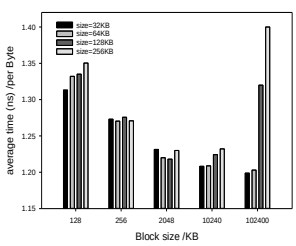


Fig. 11: Average time (per Byte) varying with segment size

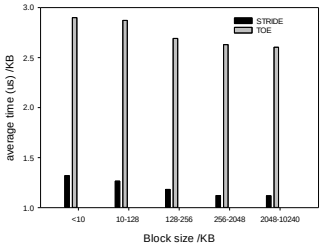


Fig. 12: Average time (per KByte) varying with block size

Simulation setting: We develop a flow-level simulator and

Simulation results. Here we discuss Gurita's average performance in trace.

.

VII. Conclusion

Gurita is a multi-stage coflow scheduler that leverages coflow's characteristic at different stages to minimize average CCT without relying on a central controller. The outcomes from our experiments demonstrate that Gurita is an effective and practical

Acknowledgement. This research was supported in part by a research grant from Chinese National Programs for Key Tech. of SDN, Supporting Resource Elasticity Scheduling, and Equipment Development (863 Programs) (No.2015AA016103), and National Natural Science Foundation: Synergy Research on CPU / FPGA Heterogeneous Network Processing System for Complex Network Applications (No.61702538). We thank Dr. Weichao Li for discussion and feedback.

Reference

[1] M. Malik and H. Homayoun, "Big data on low power cores are low power embedded processors a good fit for the big data workloads," 33rd IEEE International Conference on Computer Design, 2015.

[2] M. Malik, et. al., "System and architecture level characterization of big data applications on big and little core server architectures," IEEE Int. Conference on Big Data. IEEE BigData, 2015.

[3] V. Kontorinis, et. al., "Managing distributed ups energy for effective power capping in data centers", ISCA 2012.

[4] A. Caulfield, et. al., "A Cloud-Scale Acceleration Architecture". Annual IEEE/ACM Int. symp. on Microarchitecture (MICRO), 2016.

[5] Jian Ouyang, et. al.. "SDA: Software-Defined Accelerator for Large Scale DNN Systems. IEEE Hot Chips 26 Symposium (HCS), 2014.

[6] Jaspers, "Acceleration of read alignment with coherent attached FPGA coprocessor", Faculty of Elect., Math. and Computer Scie., 2015.

[7] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” in OSDI, 2004.
[8] M. Alizadeh, et. al.,“Data center TCP (DCTCP),” in SIGCOMM 2010.
[9] R. Nishtala, et. al., “Scaling Memcache at Facebook,” in NSDI 2013.
[10] D. Sidler, et al., “Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware,” in FCCM, 15.
[11] C.Kachris and D. Soudris. A survey on reconfigurable accelerators for cloud computing. Int. Conf. on Field Prog.Logic and App. Aug., 2016.
[12] D. Sidler, et. al.. Low-Latency TCP/IP Stack for Data Center Applications. Int. Conf. on Field Prog.Logic and App., 2016.
[13] Y. Zhu, et. al.. Congestion Control for Large-Scale RDMA Deployments. In SIGCOMM, 2015.
[14] C. Guo, et. al.. RDMA over Commodity Ethernet at Scale. In SIGCOMM, 2016.
[15] R. Mittal, et. al.. TIMELY: RTT-based Congestion Control for the Datacenter. In SIGCOMM, 2015.
[16] W. Bai, et. al.. Enabling ECN in Multi-Service Multi-Queue Data Centers. In NSDI, 2016.
[17] A. Putnam, et. al., “A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services,” in International Symposium on Computer Architecture (ISCA), 2014.
[18] M. Blott, et. al., “Achieving 10gbps line-rate key-value stores with fpgas,”, USENIX Workshop on Hot Topics in Cloud Computing. 2013.
[19] S. Jun, et. al.. “A Transport-Layer Network for Distributed FPGA Platforms”. Int. Conf. on Field Prog.Logic and App., 2015.
[20] Y. Zhu et. al.. “ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY”. CoNEXT, 2016.
[21] H. Saleh, et. al.. “Packet communication within a Go-Back-N ARQ system using Simulink” Crtl Eng. & Information Tech. , 2016.
[22] A. Bergkvist, et. al., “WebRTC 1.0: Real-time communication between browsers,” World Wide Web Consortium, W3C Editor’s Draft, Mar. 2013. http://dev.w3.org/2011/webrtc/editor/ webrtc.html.
[23] QUIC – IETF Working Group, Retrieved 2016.
[24] A. LangLey, et. al., “The QUIC Transport Protocol: Design and Internet-Scale Deplyoment”, ACM SIGCOMM, 2017.
[25] R. E. Kalman, “A new Approach to Linear Filtering and Prediction Problems”, in Trans. Of the ASME, 1960.
[26] C. Lee, et. al., “Accurate Latency-based Congestion Feedback for Datacenters”, in NSDI, 2015
[27] C. Lee, et. al.. “DX: Latency-Based Congestion Control for Datacenters”. In ACM Trans. On Networking, 2016.
[28] Chelsio T5 Packet Rate Performance Report. http://goo.gl/3jJL6p .
[29] The NetFPGA Project. http://netfpga.org/.
[30] Who (Really) Needs Sub-microsecond Packet Timestamps? http://goo.gl/TI3r1u, 2013.
[31] M. Alizadeh, et. al.. “pfabric: Minimal near-optimal datacenter transport”. in SIGCOMM 2013.
[32] M. Alizadeh, et. al.. Less is more: trading a little bandwidth for ultra-low latency in the data center, USENIX NSDI 2012.
[33] FAST project. https://github.com/FAST-Switch/fast.
[34] FAST Official website: http://www.fastswitch.org/
[35] J.. Lockwood, et. al.. NetFPGA-An Open Platform for Gigabit-Rate Network Switching and Routing. Microelectronic Systems Edu., 2007.
[36] P. Estrela, et. al., Challenges Deplyoing PTPv2 in a Global Financial Company, in ISPCS, 2012.
[37] J.Kurosa and k. Ross, “Computer Networking: A Top-Down Approach”, Pearson Education, 2017.
[38] A. Roy, et al, “Inside the Social Network’s (Datacenter) Network,” in ACM SIGCOMM 2015.
[39]