View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Edinburgh Research Explorer

THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

TCPRand: Randomizing TCP Payload Size for TCP Fairness in Data Center Networks

Citation for published version:
Lee, S, Lee, M, Lee, D, Jung, H & Lee, B-S 2015, TCPRand: Randomizing TCP Payload Size for TCP Fairness in Data Center Networks. in Proceedings of IEEE Infocom. IEEE, Kowloon, Hong Kong, pp. 1697-1705, 2015 IEEE Conference on Computer Communications , Kowloon, Hong Kong, 26/04/15. DOI: 10.1109/INFOCOM.2015.7218550

Digital Object Identifier (DOI):
10.1109/INFOCOM.2015.7218550

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of IEEE Infocom

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. Apr. 2019

TCPRand: Randomizing TCP Payload Size for TCP Fairness in Data Center Networks

Soojeon Lee*‡, Myungjin Lee†, Dongman Lee‡, Hyungsoo Jung§, Byoung-Sun Lee*
*Electronics and Telecommunications Research Institute (ETRI)
†University of Edinburgh
‡Korea Advanced Institute of Science and Technology (KAIST)
§Seoul National University

***Abstract*—As many-to-one traffic patterns prevail in data center networks, TCP flows often suffer from severe unfairness in sharing bottleneck bandwidth, which is known as the TCP outcast problem. The cause of the TCP outcast problem is the bursty packet losses by a drop-tail queue that triggers TCP timeouts and leads to decreasing the congestion window. This paper proposes TCPRand, a transport layer solution to TCP outcast. The main idea of TCPRand is the randomization of TCP payload size, which breaks synchronized packet arrivals between flows from different input ports. We investigate how TCPRand reduces consecutive packet drops and demonstrate various benefits of TCPRand with extensive experiments and ns-3 simulation. Our evaluation results show that TCPRand guarantees the superior enhancement of TCP fairness with negligible overheads in all of our test cases.**

***Index Terms*—Data center networks, TCP outcast, Fairness**

I. INTRODUCTION

In recent years, the proliferation of data center applications with many-to-one traffic pattern has brought a body of new network research issues such as TCP incast [9,14,22], deadline-awareness [15,16,17,18] and TCP outcast [1]. Among these issues, this paper focuses on the TCP outcast problem for which practical solutions have not successfully been proposed yet. The TCP outcast problem is observed easily in data center networks, where routers or switches are usually connected through a multi-rooted and hierarchical topology such as fat-tree [5] and senders and receivers are leaves of the topology. As many-to-one traffic patterns emerge in such an environment, multiple flows arrive at different input ports of a receiver's ingress switch and compete for the same outgoing queue. With excessive traffic flows, drop-tail queueing may drop a series of consecutive packets at each input port, and this is called port blackout [1]. Suppose that there are two input ports, A and B, and many flows arrive at A while a few flows do at B. If all these flows are destined to the same output port, the outcast flows (i.e., the flows arriving at B) lose the goodput substantially because TCP timeouts are triggered more easily. This is the essence of the TCP outcast problem [1] that has negative impacts on the TCP fairness among competing flows. It even leads to much higher goodput decrease in flows with a short RTT than in those with a long RTT in a fat-tree topology.

Several solutions have been suggested for the TCP outcast problem. They can be categorized into link layer and network layer solutions. To our knowledge, *none of them can be readily rolled out to the existing data center networks*. The link layer solutions require a modification to the current switching architecture [20] or are not widely supported in today's switches [7]. Equal-length routing [1], one of network layer solutions, only works in non-oversubscribed networks. To overcome the shortcomings of these two approaches, a transport layer solution can be viable since it neither relies on any specific link layer supports nor assumes any particular network topology. However, existing rate-based transport layer approaches are not applicable to TCP outcast in data centers because they require the precise control of inter-packet spacing time [3, 21] which operating systems hardly guarantee and are inappropriate [21] for a multi-hop environment.

In this paper, we propose a transport layer solution called *TCPRand*. To prevent the port blackout, we randomize each TCP packet's payload size for arbitrating the arrival times of back-to-back packets. This can reduce the chance of burst packet drops per input port. At the sender side, the proposed solution makes the TCP payload size uniformly distributed between [*rMin*, MSS]. However, it may increase the packet header overhead due to the smaller payload size and curtail the total goodput. To achieve high fairness without loss of total goodput, the proposed solution calculates *rMin* by adapting to the changes of congestion window (*cwnd*). The method is based on the observation that for many-to-one applications (e.g., especially with a barrier synchronization property [22]) as *cwnd* of a flow is growing, the network is more congested and the port blackout happens more frequently. Hence, if *cwnd* of a flow increases, the scheme decreases *rMin* for the flow.

We implement TCPRand by modifying the sender side execution path of TCP protocol stack in the Linux kernel and perform extensive experiments in our testbed. We demonstrate that TCPRand reduces consecutive packet drops and TCP timeouts significantly, and as a result, it improves TCP fairness substantially with a small loss of the overall goodput and negligible additional retransmission overheads. We also show that TCPRand always guarantees the superior enhancement of TCP fairness to the regular TCP in various test cases. In addition, we use ns-3 [11] to evaluate TCPRand with a larger and more realistic topology (i.e., fat-tree [5]) and workloads of data center networks, and show that TCPRand substantially improves TCP fairness and rarely sacrifices flow completion times of flows, especially those of small flows.

The remainder of this paper is organized as follows. In Section II, we discuss the limitations of existing solutions. In

Section III, we briefly explain the port black out problem and why payload size randomization is its key solution. Section IV provides the details of the proposed solution. We outline our evaluation setup in Section V. Evaluation results are presented in Sections VI and VII before we conclude in Section VIII.

II. Limitations of Related Work

Link layer solutions: Random early detection (RED) [6] and stochastic fair queueing (SFQ) [7] have been tested to solve the TCP outcast problem. Prakash et al. [1] point out that RED shows RTT bias while SFQ makes flows have throughput fairly and achieves RTT fairness but uncommon in commodity switches. More importantly, both solutions cannot be easily deployed for ToR switches in data centers for cost reasons [1]. Zhang et al. [20] propose a cross-layer protocol that supports bandwidth sharing by allocating switch buffer; the switch determines the size of the congestion window of its passing flow. However, all the switches in data centers must be modified for supporting such a feature to make use of this solution. Alizadeh et al. [25] propose DCTCP which may be useful to solve the outcast problem by controlling a congested port's queue length properly. However, DCTCP must leverage Explicit Congestion Notification (ECN) capability, which is not yet widely supported by most commodity ToR switches especially in small and medium data centers to our knowledge.

Network layer solutions: Equal-length routing [1] makes all flows from senders routed up to the core switch regardless of the senders' locations. Then, all the flows take the same downward path from the core to the destination which leads to RTT fairness. It uses a detour path to increase the path similarity instead of the shortest path. However, this approach causes performance degradation if data center networks are oversubscribed. Furthermore, it significantly lacks flexibility.

Transport layer solutions: The rate-based delivery (e.g., TCP pacing [3] and sending time randomization [21]) has also been considered as a solution to the TCP outcast problem. TCP pacing, combined with the window based congestion control, avoids burst delivery by giving some interval between the transmission times of two consecutive packets and shows inverse RTT bias. However, the TCP outcast problem still remains considerably in TCP pacing [1]. Chandrayana et al. propose a scheme randomizing the sending times by adjusting the inter-packet gap [21]. This, however, cannot retain the initial randomness created by the sender throughout the routing path mainly due to the bursty departure process at the first bottleneck queue. This makes the approach ineffective in a multi-hop environment. Moreover, the rate-based delivery has a severe practical limitation because it is practically infeasible to do (sub-)microsecond level packet spacing [2] (e.g., in 1/10Gbps link), quite strictly required to get better randomness effects in data center networks (where RTT<1ms [14]). Even though a high resolution timer (e.g., hrtimer in Linux) is available, operating systems hardly guarantee the precise control of inter-packet spacing time. Furthermore, frequent timer interrupts lead to a large interrupt handling overhead [14].

Viewed in this light, practicality and easy deployment of a solution do matter. *The proposed approach—payload size randomization—has two practical advantages compared to these rate-based solutions*. First, the shuffle effect is preserved even by the departure process of the bottleneck queue. Thus, our approach guarantees the random arrival times of back-to-back packets for a multi-hop environment. Second, it does not require packet spacing at a (sub-)microsecond level, which is difficult to achieve in practice.

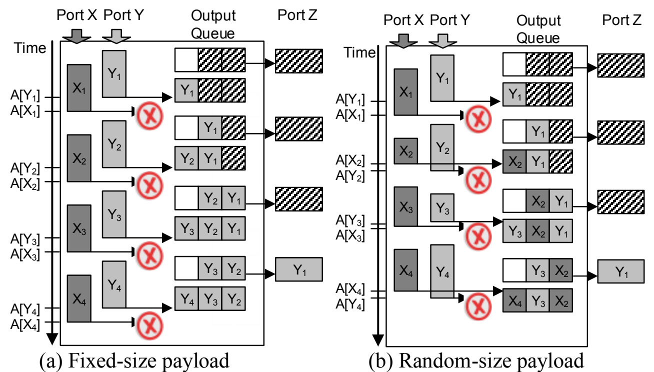


Fig. 1. Port blackout at a switch and effect of payload size randomization.

III. Effect of Randomization

In this section, we first explain why port blackout occurs in detail. Next, we discuss the payload size randomization idea as a solution to the phenomenon. Finally, through an experiment, we quantitatively show that the randomization method substantially mitigates the degree of the port blackout.

A. Port Blackout Problem

The port blackout phenomenon in data center networks is well studied in [1]. Figure 1(a) illustrates how the port blackout occurs at a bottleneck switch where a drop-tail queue management policy is applied and there exist two input ports (i.e., X and Y) and one output port (i.e., Z). Further, we assume that TCP-based bulk data transfer application traffic arrives at the switch through ports X and Y and leaves it via port Z.

In this setup, packets are almost of the same size (i.e., the size of TCP/IP headers + MSS). Traffic is bursty and the inter-frame gap between packets is constant (e.g., 0.096μs for a gigabit Ethernet) following the IEEE 802.3 specification. This condition can create a situation where packets from port Y are always stored in the output queue while packets from port X are always discarded. This occurs because packets from port Y always arrive ahead of competing packets from port X. For instance, as shown in Figure 1(a), the arrival time of packet Y_1 (denoted as $A[Y_1]$) is ahead of that of packet X_1 (i.e., $A[X_1]$), $A[Y_2] < A[X_2]$, and so forth. Even though a series of packet drops happen fairly on ports X and Y by turns, they damage more seriously to the throughput of the incoming stream from port X if the stream consists of less number of TCP flows. This is the port blackout problem [1].

B. Avoiding Concurrent Packet Arrivals

The port blackout problem can be ameliorated by reducing concurrent packet arrivals at two input ports. At the transport layer, this can be achieved by the rate-based approach but it is less practical as discussed in Section II. Our approach to the

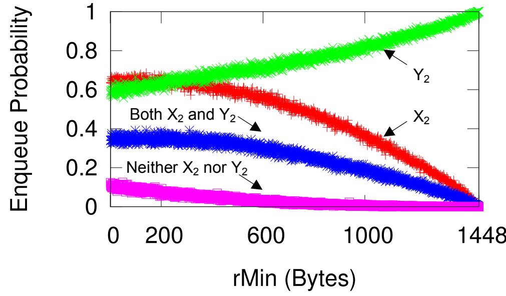


Fig. 2. Enqueue probability of X_2 and Y_2 at congestion.

problem is rather to randomize the size of each TCP payload. The intuition behind this is, randomizing the size of TCP payload can induce randomness in the arrival times of packets and it finally breaks the synchronized arrival times of back-to-back packets at each input port. This can reduce the chance of having port blackout, and the initial randomness can be preserved all the way down to the receiver in multi-hop environments. For example, in Figure 1(b), X_1 is dropped since Y_1 arrives slightly before X_1. However, in the next phase, X_2 is inserted to the output queue since X_2's TCP payload size is reduced after the randomization so that the arrival time of X_2 is ahead that of Y_2, having Y_2 dropped. After that, the arrival time of Y_3 is ahead that of X_3 due to the randomization of the TCP payload size. Thus, Y_3 is enqueued while X_3 is dropped. Eventually, this procedure lets us have $A[Y_1] < A[X_1]$, $A[X_2] < A[Y_2]$, $A[Y_3] < A[X_3]$, and $A[X_4] < A[Y_4]$. Thus, Y_1, X_2, Y_3 and X_4 are inserted to the output queue while the rest are discarded. Packet drops occur rather alternately in each port; thus the frequency of the port blackout phenomenon decreases.

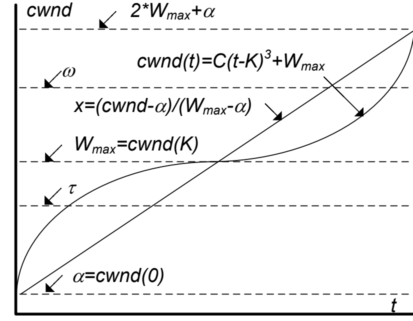
C. Understanding the Effect of Payload Size Randomization

To take a closer look at the port blackout phenomenon, we investigate how much a series of packet drops from each input port can be alleviated with the payload size randomization at a switch under congestion. Let Q ($0 \leq Q \leq Q_{max}$) be the output queue length. A packet drop occurs at a drop-tail queue if a packet arrives when $Q=Q_{max}$. To quantitatively measure the effect of the payload size randomization, we focus on the enqueue probability of two packets X_2 and Y_2 after Y_1 is enqueued and X_1 is dropped (see Figure 1(b)).

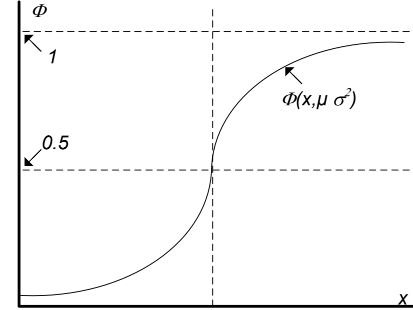
More formally, the probability of packet *pkt* to be enqueued at A[*pkt*], is acquired by:

$$Pq(pkt)=1-P(Q=Q_{max} \text{ at } A[pkt]). \quad (1)$$

Based on the notion of Eq. 1, we experimentally measure the enqueue probabilities of i) X_2 ($Pq(X_2)$), ii) Y_2 ($Pq(Y_2)$), iii) both X_2 and Y_2 ($Pq(X_2 \cap Y_2)$), and iv) neither X_2 nor Y_2 ($Pq(\sim X_2 \cap \sim Y_2)$) while randomizing payload sizes. To do so, we write an offline test code generating two virtual back-to-back flows (from X and Y). We randomly select a payload size of each packet within the range of [*rMin*, MSS]. We vary the degree of randomness by changing *rMin* from 1B to 1448B at the interval of 1B. We construct a simple experimental setup as follows: First, nodes are connected with 1Gbps links. Second, there are two input ports X and Y, and one output port Z. Third, back-to-back packets arrive continuously at each input port and



(a) CUBIC's *cwnd*(t). In CUBIC, when a packet drop is detected, the *cwnd* decreases by a factor of β (=717/1024 in Linux kernels). Then, a new CUBIC epoch begins at t=0, and the initial *cwnd* of the epoch α is set to *cwnd*(0). W_{max} (called the current maximum or the origin point) is the *cwnd* where packet losses occurred previously. Refer to [13] for more details on C and *K*.



(b) Φ: Normal Distribution CDF

Fig. 3. Adaptive selection of Φ based on CUBIC's *cwnd*.

the inter-frame gap is 0.096μs (i.e., 8B in a gigabit Ethernet). Last, Y_1 is enqueued to the output queue while X_1 is dropped.

By tracing all the packet arrivals and departures since $A[Y_1]$, we measure $Pq(X_2)$ and $Pq(Y_2)$. We conduct this test 1000 times per each *rMin*. Figure 2 shows the four types of probabilities of interest. If the regular TCP (i.e., the payload size is not randomized at all and *rMin*=1448B) is used, X_2 never be enqueued. Of course, this simple experimental result may not hold in real network environments since the packet arrival time can be distorted due to some random factors (e.g., variations in sending patterns or other unpredictable random behaviors) [1, 4] and TCP does not generate endless bursty traffic unlike we did for the test. However, Figure 2 clearly indicates why the port blackout is hard to be prevented with the regular TCP at a drop-tail queueing switch.

As *rMin* decreases (i.e., from the right of the *x*-axis to the left in Figure 2), $Pq(X_2)$ increases and $Pq(Y_2)$ decreases. $Pq(X_2)$ and $Pq(Y_2)$ approach to 0.63 and 0.58, respectively when *rMin*=1B. One interesting observation is that $Pq(X_2 \cap Y_2)$ also increases by decreasing *rMin*. However, the payload size randomization can make both X_2 and Y_2 dropped (e.g., with the probability of 0.11 when *rMin*=1B). Nevertheless, the advantages far outweigh this disadvantage since the probability of consecutive packet drops reduces significantly by the randomization mechanism.

Another implication from the above result is that it is unnecessary to reduce *rMin* overmuch. There are two reasons. First, the enqueue probability of X_2 grows up more slowly as

rMin approaches to 1B. Second, the lower *rMin*, the larger the header overhead. It results in bandwidth waste.

IV. Proposed Scheme

In this section, we focus on the design of our proposed scheme that we call *TCPRand*. As mentioned in Section III.C, for each packet, we determine its payload size via generating a uniform random number in the range of [*rMin*, MSS]. Since *rMin* is a configurable variable (1 ≤ *rMin* ≤ MSS), we can diversify randomly generated payload sizes by selecting one *rMin* value. However, it is unclear what value to set. Moreover, the degree of port blackout can vary depending on several factors such as background traffic, changes in traffic patterns, etc. Due to these reasons, we consider a scheme that can adaptively select *rMin* value and effectively react to changes in such factors. Thus, *our design choice for the adaptation method lies not only in maximizing the fairness, but also in minimizing the loss of total goodput in any circumstances*. We design our adaptation method on top of TCP CUBIC [13], the default congestion control algorithm in Linux.

A. *Modeling Adaptive Selection of rMin* in CUBIC

We focus on CUBIC's *cwnd* growth function for designing an adaptive *rMin* selection method as variation in *cwnd* value can be indicative of the probability of packet loss, which is a necessary condition of the port blackout.

Let us first take a look at how the CUBIC's window growth function (i.e., *cwnd*(t)), depicted in Figure 3(a), works. We classify a CUBIC epoch into 4 stages and present our adaptive *rMin* selection strategy for each stage based on its functional characteristics.

Stage 1) Fast growth of cwnd (when cwnd < W_{max}): At the initial phase of a CUBIC epoch, the *cwnd* grows very fast. The rationale here is that the fast *cwnd* growth is unlikely to cause a packet drop since the *cwnd* is already reduced by a factor of β just before the start of this epoch. Therefore, as **Strategy 1**, we propose to not reduce *rMin* aggressively.

Stage 2) Slow growth of cwnd (when cwnd < W_{max}): CUBIC slows down the growth of *cwnd* as approaching to $W_{\max}$ since packet losses occurred at $W_{\max}$ previously. The CUBIC's heuristic indicates that the probability of packet loss is increasing fast at this stage. To counter the port blackout actively, **Strategy 2** is to reduce *rMin* aggressively.

Stage 3) Slow growth of cwnd (when cwnd ≥ W_{max}): If the *cwnd* grows past $W_{\max}$, CUBIC enters a max probing phase [13]. At the beginning of the max probing phase, the *cwnd* grows slowly to find out a new maximum point nearby as the CUBIC's heuristic expects that the probability of packet loss becomes higher when *cwnd* ≥ W_{max}. Thus, as **Strategy 3**, *rMin* must decrease aggressively again to prevent the port blackout.

Stage 4) Fast growth of cwnd (when cwnd ≥ W_{max}): If no packet loss is detected for some period of time after stage 3, CUBIC performs a fast increase of *cwnd* since it guesses the new maximum is far away. Thus, **Strategy 4** is to not reduce *rMin* actively at this stage.

Algorithm1: Adaptation Method to Select *rMin*

1: **Input:** $\omega, \tau,$ *cwnd* $, \mu, \sigma^2, \theta$
2: **if** $\tau \leq$ *cwnd* **and** *cwnd* $\leq \omega$ **then**
3: $\quad x = \dfrac{cwnd - \alpha}{W_{\max} - \alpha}$; /* normalized distance from α */
4: $\quad \Phi(x,\mu,\sigma^2) = \dfrac{1}{2}\left[1 + \dfrac{1}{\sqrt{\pi}} \int_{-(\frac{x-\mu}{\sigma\sqrt{2}})}^{\frac{x-\mu}{\sigma\sqrt{2}}} e^{-t^2}\, dt\right]$;
5: $\quad rMin = \max(MSS \times (1 - \Phi(x,\mu,\sigma^2) \times \nu), \theta)$;
6: **else**
7: $\quad$ *rMin*=MSS;
8: **end if**

B. *Adaptive Algorithm to Calculate rMin*

We adopt the proposed strategies discussed in Section IV.A and propose the TCPRand's adaptation method (**Algorithm 1**) to calculate *rMin* before sending a packet.

1) *How to decide rMin?*

rMin is calculated based on $\Phi(x,\mu,\sigma^2)$, which is the normal distribution CDF[1] shown in Figure 3(b). As the first parameter of Φ, x is a normalized distance between *cwnd* and α as shown at the line 3 of **Algorithm 1**. For instance, if *cwnd*=$W_{\max}$, x=1. The second and third parameters of Φ, μ and σ^2 are the mean and the variance, respectively and they are configurable. *rMin* is determined by the line 5 of **Algorithm 1** based on Φ and the other two parameters[2], ν and θ. ν is a scale factor adjusting the effect of Φ. The lower bound of *rMin* is set by a parameter θ, to prevent *rMin* from decreasing overmuch.

The normal distribution CDF supports our strategy for each of the 4 stages well as follows. Assume that μ=1. At stage 1, Φ increases very slowly and it leads to the gradual reduction of *rMin* as **Strategy 1**. At stage 2, Φ increases fast and finally converged to 0.5; it causes the fast reduction of *rMin* as **Strategy 2**. At stage 3, Φ grows quickly so that the reduction of *rMin* is still fast as **Strategy 3**. At stage 4, Φ grows leisurely and leads to the slow reduction of *rMin* as **Strategy 4**.

2) *When to turn TCPRand on/off?*

Trigger point: Based on **Strategy 1**, we activate TCPRand only when the $\tau \leq$ *cwnd*. The trigger point τ shown in Figure 3(a) is acquired by:

$$\tau = W_{\max} - \frac{W_{\max} - \alpha}{\nu_\tau}, \tag{2}$$

where ν_τ is a scale factor tuning τ. If ν_τ=1, τ=α. If $\nu_\tau \to \infty$, τ=$W_{\max}$.

End point: With **Strategy 4**, TCPRand can also set the end point ω, as shown in Figure 3(a). TCPRand is deactivated if *cwnd* grows above ω, which is set by:

---

[1] To reduce the Φ calculation overhead (not trivial) at kernel, we pre-calculated Φ for various input parameters and stored the result in a table. Thus, Φ is acquired by a simple table lookup.

[2] We set ν=1 and θ=200B.

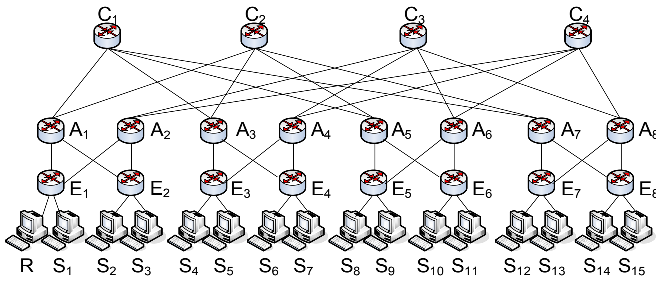


Fig. 4. Fat-tree topology composed of switches (C_n: Core, A_n: Aggregation, and E_n: Edge) and end-nodes (R: Receiver and S_n: Sender).

$$\omega = W_{\max} + \frac{W_{\max} - \alpha}{v_\omega}, \tag{3}$$

where v_ω is a scale factor tuning ω. If $v_\omega \to 0$, $\omega \to \infty$. If $v_\omega=1$, $\omega = 2\times W_{\max} - \alpha$. Preventing ω from growing too much is useful to avoid unnecessary payload size randomization in case of large *cwnd* (e.g., when the competing flows finish). Note that Eq. 2 and Eq. 3 are implemented at the line 2 of **Algorithm 1**.

V. Evaluation Setup

We evaluate the proposed solution in two ways: ns-3 simulator and real testbed. We first describe our evaluation environments, enumerate parameters for TCP and TCPRand, and finally outline evaluation metrics and test scenarios before presenting our results in Sections VI and VII.

A. NS-3 simulation environment

We incorporate TCPRand with the packet-level simulator ns-3 to experiment it in a full-blown topology (i.e., fat-tree [5]) of a data center network. We choose ns-3 because it enables high performance simulation. We adopt most of the configuration parameters suggested in [1] (including link capacity (=1Gbps), TCP minRTO value (=2ms), MSS value (=1460B), routing policy, etc.). The processing delay of each switch is set to 25 microseconds as suggested in [8]. We integrate TCPRand to both NewReno and CUBIC. We use CUBIC source code for ns-3 obtained from [10].

B. Testbed environment

To make our testbed realistically reflect a fat-tree topology shown in Figure 4, we use a topology illustrated in Figure 5. All the machines, on which TCPRand is running, are equipped with an Intel Core i7-3770K CPU @3.50GHz, 32GB of main memory and Intel 82579 Gigabit Ethernet NIC. For the switches, we use Cisco catalyst 2970 which adopts the drop-tail queue management policy. We implement TCPRand by modifying the TCP output engine in the Linux kernel 3.2.39. All the offload options including TCP segmentation offload (TSO), generic segmentation offload (GSO) and generic receive offload (GRO) are disabled because they use the offload engine in NIC and make TCPRand not work as expected.

C. TCP Parameters

TCPRand randomizes the payload size, which in most cases becomes smaller than MSS, and as a result it may generate more packets compared to the regular TCP. Due to its unique

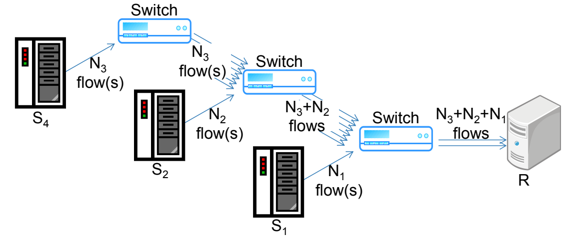


Fig. 5. Abstracted subset topology of fat-tree in Fig.4.

characteristics, we consider the following factors that can affect the performance of TCPRand as follows:

Appropriate Byte Counting (ABC): Even though TCP output engine in Linux increases *cwnd* based on the “number” of acks (which works well with the MSS-sized payload), by enabling ABC [23] option, *cwnd* can be increased based on the “bytes” acked. In Linux kernels, ABC is implemented only in Reno but we also implement it in CUBIC to observe its effects. However, for the scenarios where TCP outcast happens (e.g., many flows and a few flows are arriving at two input ports and destined to the same output port), the use of ABC did not change the overall test result. It is because the effect of ABC is far smaller than that of the port blackout in the TCP outcast scenarios. Thus, in this paper, we only show the results experimented without ABC.

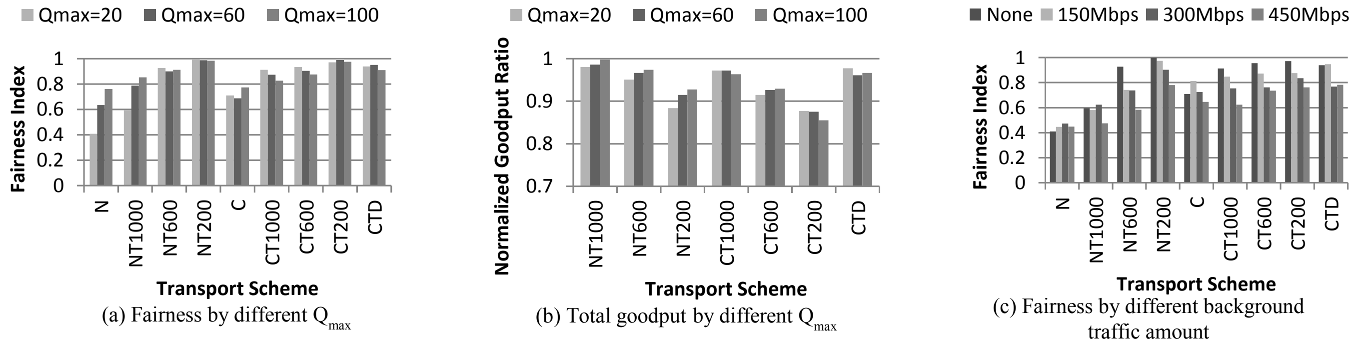
Nagle’s Algorithm and Congestion Control: To observe how TCPRand cooperates with different congestion control mechanisms, we choose Reno, BIC and CUBIC [13] and test them with or without the Nagle’s algorithm [12]. However, for the TCP outcast scenarios, there is no noticeable difference among the six combinations since the port blackout overwhelms their effect. Thus, we only address the case with CUBIC and the Nagle’s algorithm since CUBIC is the default congestion management protocol in Linux today and most bulk transfer applications enable the Nagle’s algorithm.

SACK: By default, SACK[3] is enabled for the fast recovery from multiple packet losses in today’s Linux. However we also conduct experiments without SACK to see its role in TCP outcast scenarios when combined with TCPRand.

D. TCPRand Parameters

TCPRand has four parameters (i.e., σ^2, μ, v_τ, v_ω), thus allowing many possible combinations of these parameters. For instance, we can vary parameter values as follows: σ^2={0.2,1,5}, μ={1,0}, v_τ={∞,1}, v_ω={0,1}. A larger σ^2 causes faster growth of Φ when $x<\mu$ but Φ grows slowly when $x\geq\mu$. With a smaller μ, more aggressive increase of Φ can be observed. $\tau=\alpha$ if $v_\tau=1$, while $\tau=W_{\max}$ if $v_\tau\to\infty$. $\omega = 2\times W_{\max} - \alpha$ if $v_\omega=1$, while $\omega\to\infty$ if $v_\omega=0$. Out of many configurations possible, we conduct evaluation with the three sets of configurations denoted in the form of (σ^2, μ, v_τ, v_ω). One configured as (1, 1, 1, 1) represents a moderate setting, which is our default setting. The other is set as (1, 1, ∞, 1) which represents the most conservative setting. The third is the most aggressive setting that is configured as (1, 0, 1, 0). Unless

[3] SACK is not supported by the current version of ns-3.



(a) Fairness by different Q_{max} (b) Total goodput by different Q_{max} (c) Fairness by different background traffic amount

Fig. 6. Effect of Q_{max} and background traffic. N: NewReno, NT*x*: NewReno+TCPRand(*rMin*=*x* bytes), C: CUBIC, CT*x*: CUBIC+TCPRand(*rMin*=*x* bytes), and CTD: CUBIC+ Adaptive TCPRand with $(\sigma^2, \mu, \nu_\tau, \nu_\omega)$=(1, 1, 1, 1).

otherwise mentioned, we use the default setting while we mix and match the configurations when necessary.

E. Evaluation metrics

We are primarily interested in evaluating TCPRand with two key metrics: fairness and goodput across both real testbed and simulation cases. We shortly define each of them next.

Fairness: We use Jain's fairness index [24] defined as follows:

$$Fariness(g_1, g_2, \ldots, g_n) = \frac{(\sum_{i=1}^{n} g_i)^2}{n \times \sum_{i=1}^{n} g_i^2}, \quad (4)$$

where g_i is the average goodput of flows sent by S_i .

Goodput: As typically defined, we obtain goodput by dividing the amount of application-level data by the total time taken until the completion of its delivery.

F. Test Scenarios

i) The typical TCP outcast scenario: A total of 15 senders (S_1-S_{15}) generate one TCP flow per sender to receiver R in the fat-tree topology in Figure 4. We check how TCPRand mitigates the TCP outcast problem. In doing so, we analyze how TCPRand interacts with varying the maximum length of the drop-tail queue (Q_{max}) and background traffic values. Specifically, all 15 senders (S_1-S_{15}) simultaneously generate only one flow per sender for 10 seconds. Each flow sent from sender S_n is denoted by F_n. Thus, in the fat-tree, E_1 is the most bottlenecked switch and F_1 is the most outcast flow since it competes with $F_{2:15}$ for the output queue at E_1.

To demonstrate that TCPRand works well in the real world, we construct a testbed which simplifies the fat-tree topology in Figure 4 but still preserves its essential nature for creating TCP outcast. The testbed topology is shown in Figure 5. Using this topology, one can create many TCP outcast cases with different combinations of (N_1, N_2, N_3)[4]. In fact, we tested TCPRand in many TCP outcast events and found in all cases TCPRand achieves similar fairness and goodput. Thus, out of them, we choose two combinations: i) (2, 4, 26) for mimicking the observation [1] that more flows come from distant senders while less flows come from close senders in the fat-tree and ii) (26, 4, 2) as an opposite case of the above to show that TCPRand can solve the TCP outcast problem even in unusual situations.

ii) Realistic data center workload scenario: Since TCPRand tends to reduce the payload size less than MSS, one may wonder whether it increases flow completion time (FCT), in particular that of short flows which in general originate from latency-sensitive applications. To answer that question, we trace the effect of TCPRand to FCT using two realistic data center workloads (i.e., web search and data mining) [19] that consist of a mix of short and long flows. Flow arrivals follow a Poisson process and the sender and receiver for each flow are chosen randomly among all the 16 end-nodes (i.e., R, S_1, …, S_{15}). The flow arrival rate (i.e., load in the fabric) is varied from 0.2 to 0.8 as suggested in [19].

iii) Microscopic view on TCPRand: To further understand what effects TCPRand brings to TCP flows in detail, we conduct a microscopic analysis with a simplest topology exhibiting the TCP outcast. We do this in our testbed instead of ns-3 simulator. It is because the testbed environment can best reflect the microscopic behaviors caused by the temporal port blackout that happens at an output queue of commodity hardware switches.

VI. NS-3 SIMULATION RESULTS

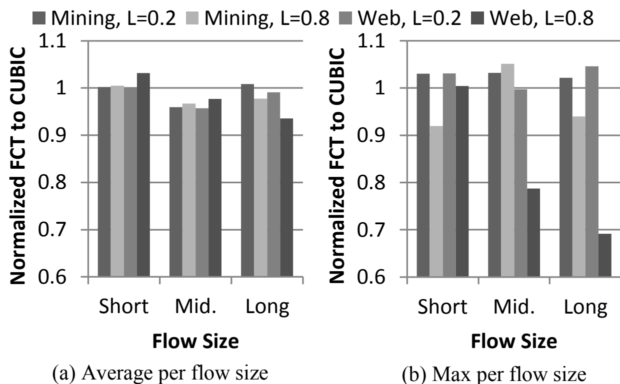
We first evaluate TCPRand in an ns-3 environment. The evaluation focus lies on the two metrics—fairness and goodput—while we vary network conditions such as switch queue size (Q_{max}) and the amount of background traffic. In addition to that, we conduct simulation with data center workloads [19] to show that TCPRand in general supports flows with different sizes well. More details on test scenarios are found in Section V.F.

A. Fairness and Goodput Analysis

In this analysis, we additionally plot the results of TCPRand with static settings (i.e., fixed *rMin*) alongside TCPRand (denoted as CTD in Figure 6) to demonstrate why the adaptive *rMin* selection method is better than configuring *rMin* statically.

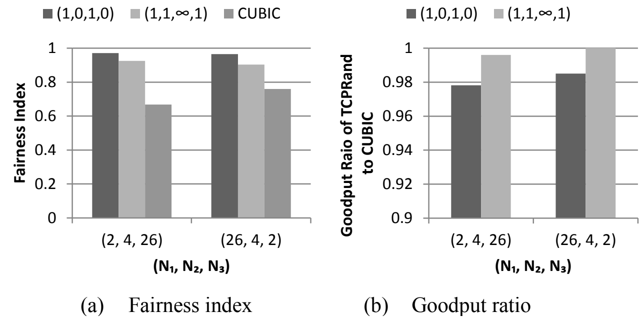
Impact of Q_{max} on fairness and goodput: To see the effect of Q_{max} to TCPRand, we set Q_{max}={20, 60, 100} packets. Notations for transport schemes are given in the caption of Figure 6. As shown in Figure 6(a), the regular TCP (i.e., N and C) suffers from the unfairness caused by the TCP outcast. As

[4] N_1, N_2 and N_3 are the number of flows generated by S_1, S_2 and S_4, respectively in Figure 5.



(a) Average per flow size (b) Max per flow size

Fig. 7. Normalized FCT of TCPRand to CUBIC for different workloads and traffic loads. Sizes of short, mid and long flows are [0,100KB), [100KB,10MB), and [10MB,∞), respectively.



(a) Fairness index (b) Goodput ratio

Fig. 8. Fairness and goodput of TCPRand and CUBIC under testbed with the topology in Fig. 5., respectively. The 4-tuple in legend corresponds to (σ^2, μ, ν_τ, ν_ω) of TCPRand. (1, 0, 1, 0) is the most aggressive setting while (1, 1, ∞, 1) is the most conservative configuration.

decreasing *rMin* statically, the outcast flows recover quickly and the fairness index approaches to 1 regardless of Q_{max}. However, more aggressive reduction of *rMin* triggers more loss of total goodput as shown in Figure 6(b) (goodput ratio normalized to that of N or C). For instance, when *rMin*=200B, CT200 loses 14.5% of total goodput compared to C. In contrast, CTD efficiently strikes a balance between the fairness and total goodput. For example, it always keeps both the fairness and total goodput higher than CT600, a seemingly best static setting.

Impact of background traffic on fairness: For this simulation, given 15 senders, we make each sender additionally generate, to the receiver, 10, 20 and 30Mbps UDP CBR traffic, accounting for 150, 300 and 450Mbps aggregate background traffic, respectively. Figure 6(c) shows the effect of background traffic to the fairness where Q_{max}=20. We clearly observe that TCPRand always achieves higher fairness than the regular TCP. However, the larger the background flows, the smaller the additional fairness gain of TCPRand to the regular TCP. Note that the payload size of the background flows is not randomized at all. Thus the effect of the payload size randomization to the port blackout is restricted more as the amount of background traffic increases. However, even with the largest background traffic (i.e., 450Mbps), TCPRand still achieves a noticeable fairness improvement.

B. Analysis on real data center workloads with TCPRand

Figure 7 shows the normalized FCT of TCPRand to CUBIC per flow size. Two trends are observed. First, TCPRand does not increase the average FCT of short flows noticeably (see Figure 7(a)). It is because many short flows are extremely small in real (especially in data mining) workloads and many of them finish before TCPRand performs the aggressive reduction of *rMin*. Second, the CUBIC (especially long) flows often experience timeout due to TCP outcast under high traffic load but TCPRand successfully curtails the outcast of the flows (see Figure 7(b)).

VII. EXPERIMENTAL RESULTS

Now we evaluate TCPRand in a real testbed. The main purpose of this evaluation in the testbed is to truly confirm that TCPRand in practice improves fairness without compromising goodput in the presence of TCP outcast. Next, we conduct microscopic analysis to shed light on how several aspects (packet drops, timeouts, and retransmissions) in TCP congestion control are affected by TCPRand. Finally, we discuss CPU overhead caused by TCPRand.

A. Fairness and Goodput Analysis

As explained in Section V.F, we test TCPRand using an abstracted subset topology of a fat-tree in Figure 5. We tested tens of different TCP outcast events and found that regardless of the test cases, the performance of TCPRand is quite similar. Thus, we only show the most interesting results obtained from two configurations (N_1=2, N_2=4, N_3=26) and (26, 4, 2).

Fairness: Figure 8(a) shows that regardless of the parameter (σ^2,μ,ν_τ,ν_ω) configurations, TCPRand always achieves a higher fairness index than CUBIC. We observe that higher fairness is achieved as configurations become more aggressive (i.e., with smaller μ, smaller τ or larger ω) in randomizing the payload size. For instance, the largest increase of TCP fairness is accomplished with (σ^2=1, μ=0, ν_τ=1, ν_ω=0), which is the most aggressive setting and guarantees fairness index superior to 0.9 in all the scenarios we experimented. However, even with the most conservative setting (i.e., σ^2=1, μ=1, $\nu_\tau \rightarrow \infty$, ν_ω=1), the fairness is enhanced significantly compared to CUBIC.

Loss of total goodput: If μ=1, TCPRand always keeps the additional loss of total goodput to CUBIC low (mostly < 1%) as shown in Figure 8(b). Although we do not show the exact picture for brevity, even for the case where the TCP outcast does not happen (i.e., the same number of flows compete) and the total number of competing flow is small (i.e., 3), TCPRand minimizes the total goodput loss (~1%) effectively. This indicates that even though TCPRand is mainly designed to pursue more fairness for TCP outcast scenarios, it causes only a trivial amount of additional goodput loss for non-outcast scenarios; this is possible since the proposed adaptive randomization scheme in **Algorithm 1** avoids or minimizes unnecessary payload size randomization as much as possible. Of course, as expected, the most aggressive setting (i.e., with σ^2=1, μ=0, ν_τ=1, ν_ω=0) leads to the largest (i.e.,~2.3%) decrease of total goodput compared to CUBIC. However, even for this worst case, we believe the additional goodput loss caused by TCPRand is low and reasonable (depending on

TABLE I. Distribution of F_1's consecutive packet drops.

	M	*rMin* (Bytes)	# consecutive packet drops 1	2	3	4	5	6	7	8	Total drops
SACK enabled	5	1448	341	144	53	26	16	5	2	1	1024
		1000	401	115	39	11	5	2	1	1	844
		600	394	86	31	7	1	0	0	0	692
		200	430	94	26	4	0	1	0	0	718
	15	1448	1983	791	230	40	13	5	1	0	4527
		1000	1702	409	106	31	9	2	0	0	3019
		600	1684	321	59	7	4	0	2	0	2565
		200	1601	212	17	2	0	0	0	0	2084
	25	1448	2549	1231	197	10	0	1	1	0	5666
		1000	2468	627	163	39	11	3	0	0	4449
		600	2635	456	99	11	2	0	0	0	3898
		200	2618	303	27	9	1	0	0	0	3346
SACK disabled	5	1448	311	155	56	27	11	4	6	1	1135
		1000	345	134	40	14	5	3	0	0	832
		600	376	103	29	6	7	1	0	0	758
		200	408	100	22	6	2	0	0	0	708
	15	1448	880	430	207	58	76	35	43	29	5515
		1000	1174	402	243	81	80	31	35	20	6252
		600	1693	480	225	48	44	18	10	10	5035
		200	1610	339	85	13	7	11	2	8	3085
	25	1448	1492	839	211	43	49	15	24	10	5082
		1000	1112	406	303	65	88	32	37	29	5440
		600	1198	406	368	68	90	51	38	31	6960
		200	1586	484	300	56	63	61	15	35	6782

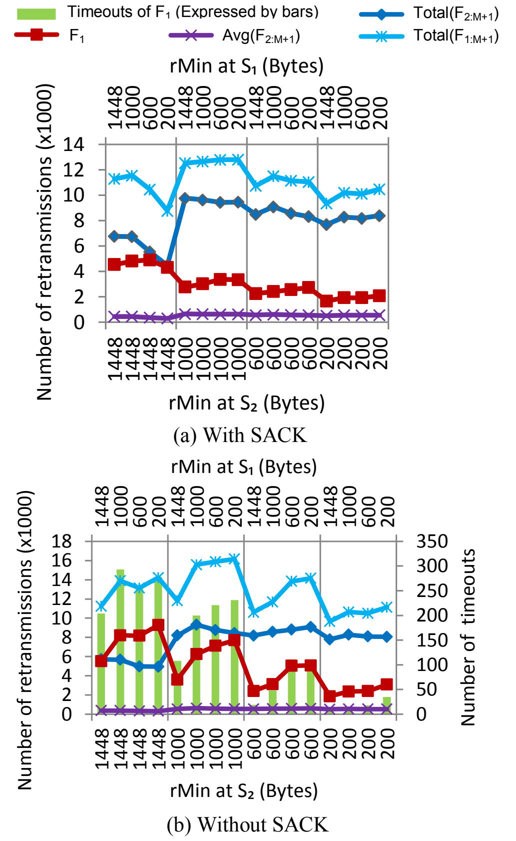
applications' characteristics). It is because most many-to-one applications are barrier synchronized [22] thus enhancing the goodput of the slowest TCP connection is more important than maximizing the total goodput.

B. Microscopic Analysis on TCPRand

We use the same testbed shown in Figure 5 where we only use two senders (S_1 and S_2) and one receiver (R). S_1 creates one flow (denoted as F_1) to R and S_2 does M flows (from F_2 to F_{M+1}, denoted as $F_{2:M+1}$) to the same R. We vary M where M={5, 10, 15, 20, 25}. Out of these five cases, we only present the most prominent results that are observed when M={5, 15, 25}. We disable the adaptive *rMin* selection method and statically vary *rMin* values. *rMin* of each flow is set to 1448, 1000, 600 or 200 bytes to make our analysis more tractable. For the measurements, we use *iperf* and run it for 100 seconds per each case. All flows (i.e., $F_{1:M+1}$) start transmission simultaneously[5].

Basically, SACK is enabled in our experiments as most modern Linux distributions support SACK by default, but for a broader analysis, we also present results while disabling SACK as well. We examine consecutive packet drops, TCP timeouts, and packet retransmission for the analysis.

Consecutive Packet Drops: Table I shows the distribution of consecutive packet drops measured with or without SACK. Both S1 and S2 use the same *rMin*. When SACK is enabled, the number of consecutive packet drops decreases significantly as *rMin* decreases. The largest reduction is observed with the smallest *rMin* (i.e., 200B) across all M's. More importantly, the reduction of more than one consecutive packet drops drives the reduction of the total packet drops. When SACK is off, the number of consecutive packet drops does decrease with TCPRand up to M = 15, but the number does not decrease much as M further increases.



(a) With SACK

(b) Without SACK

Fig. 9. The number of TCP timeouts and retransmissions when M=15.

TCP Timeouts: Figure 9(a) shows that TCPRand+SACK prevents the outcast flow from experiencing any TCP timeout (represented by the right *y*-axis of graphs with bars); although omitted for brevity, when M=25, only one configuration caused at most 4 timeouts. On the other hand, disabling SACK shows two intriguing patterns in Figure 9(b). i) TCPRand reduces the number of TCP timeouts enormously with smaller *rMin* values; when M=15, TCP timeouts decrease from 204 (*rMin*=1448, regular TCP) to 9 times. ii) However, when M grows to 25, TCPRand fails to reduce TCP timeouts noticeably (not shown for brevity). Even for the regular TCP, enabling SACK option greatly helped in reducing the number of TCP timeouts (e.g., only one timeout when M=25).

Packet Retransmissions: Figure 9 shows the number of packet retransmissions of flows (represented by the left *y*-axis in each graph). In all cases, the outcast flow causes more packet retransmissions than the non-outcast flows do as expected. When SACK is on, TCPRand generates more packets (smaller than MSS) than the regular TCP as decreasing *rMin*. Thus, the number of packet retransmissions of the outcast flow becomes larger as *rMin* of the flow decreases. However if *rMin* of the non-outcast flows decreases, the number of packet retransmissions in the outcast flow tends to decrease while those in the non-outcast flows increase as shown in Figure 9(a). In comparing Figures 9(a) and 9(b), we see TCPRand without SACK makes the outcast flow generate

[5] Note that we also conducted experiments with delaying the start of some flows and found that the arrival time difference of flows changes the result little.

more unnecessary retransmissions than that with SACK due to the lack of selective acknowledgement mechanism. However, when M < 15, TCPRand without SACK shows the similar pattern to that with SACK (the graph is omitted).

Throughout the analysis, we find out that TCPRand in general decreases the number of consecutive drops, TCP timeouts and packet retransmissions of the outcast flow. Another interesting finding is that TCPRand alongside SACK option is most effective in alleviating several adversary events to TCP performance. However, even with SACK, statically changing *rMin* value is insufficient to completely address the TCP outcast problem, reassuring that our adaptive payload size randomization method that we proposed is absolutely necessary.

C. CPU Overhead

By default in Linux, offload options such as TSO are enabled to reduce CPU overhead if its NICs support them. In our testbed, 4.6% of the resource of a CPU core is used in sending a CUBIC flow when TSO is enabled whereas if TSO is disabled, 12.5% of the resource is consumed. In addition, with TSO disabled, TCPRand consumes more CPU cycles than CUBIC since it generates more number of packets than CUBIC. In our test, TCPRand uses at most 37.5% of the resource of one core when *rMin*=200B and the number of flows, say *n*, is 25. In an extreme case (e.g., *rMin*=200B and *n*=1,000), TCPRand consumes 53% of the CPU core resource. However, this amount of CPU clock consumption may be acceptable since even commodity servers are equipped with multicore CPUs.

VIII. Conclusion

To address the TCP outcast problem in data center networks, we proposed a payload size randomization scheme called TCPRand which guarantees the superior enhancement of TCP fairness while neither sacrificing the total goodput nor incurring any noticeable network overhead. We believe that it is the first practical, cheap, lightweight and efficient solution that solves the TCP outcast problem. While we showcase the efficacy of TCPRand in this work, we also see several avenues for future work. One of such directions is to reduce the CPU overhead of TCPRand. We envision that integrating TCPRand into the TSO engine in NICs has a lot of promise for that.

Acknowledgements

This work was supported by the Space Core Technology Development Program of NRF [NRF-2014M1A3A3A03034729, Development of core S/W standard platform for GEO satellite ground control system], ICT R&D program of MSIP/IITP [B0101-14-0334, Development of IoT-based Trustworthy and Smart Home Community Framework], and a grant from the British Council.

References

[1] P. Prakash, A. Dixit, Y. Hu, and R. Kompella, "The TCP outcast problem: exposing unfairness in data center networks," in UNENIX *NSDI*, 2012.

[2] C. Lee, K. Jang, and S. Moon, "Reviving Delay-based TCP for Data Centers," in ACM *SIGCOMM*, 2012 (poster).

[3] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the Performance of TCP Pacing," in IEEE *INFOCOM*, 2000.

[4] R. Kapoor, A. Snoeren, G. Voelker, and G. Porter, "Bullet trains: a study of NIC burst behavior at microsecond timescales," in *ACM CoNEXT*, 2013.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable commodity datacenter network architecture," in ACM *SIGCOMM*, 2008.

[6] S. Floyd, and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM* ToN, August 1993.

[7] P. E. McKenney, "Stochastic fairness queueing," in IEEE *INFOCOM*, pages733–740, 1990.

[8] Cisco, "Design Best Practices for Latency Optimization, Financial Services Technical Decision Maker White Paper," http://www.cisco.com/application/pdf/en/us/guest/netsol/ns407/c654/ccmigration_09186a008091d542.pdf.

[9] Y. Chen, R. Griffith, D. Zats, A. D. Joseph, and R. Katz, "Understanding TCP Incast and Its Implications for Big Data Workloads," USENIX ;login: Magazine. Vol.37. No.3. pp.24-38. June 2012.

[10] http://www.nsnam.org/wiki/Current_Development.

[11] http://www.nsnam.org/.

[12] J. Nagle, "Congestion control in IP/TCP internetworks," RFC896, Internet Engineering Task Force, Jan. 1984.

[13] I. Rhee and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant," in Proc. of the PFLDNet Workshop, Feb. 2005.

[14] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G.Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and effective fine-grained TCP retransmissions for data center communication," in ACM *SIGCOMM*, 2009.

[15] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks," in ACM *SIGCOMM*, 2012.

[16] B. Vamanan, J. Hasan, and T. N. Vijaykumar, "Deadline-Aware Datacenter TCP (D2TCP)," in ACM *SIGCOMM*, 2012.

[17] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron, "Better never than late: Meeting deadlines in datacenter networks," in ACM *SIGCOMM*, 2011.

[18] C. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in ACM *SIGCOMM*, 2012.

[19] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal Near-Optimal Datacenter Transport," in ACM *SIGCOMM*, 2013.

[20] J. Zhang, F. Ren, X. Yue, R. Shu, and C. Lin, "Sharing Bandwidth by Allocating Switch Buffer in Data Center Networks," IEEE JSAC, Vol. 32, No. 1, 2014.

[21] K. Chandrayana, S. Ramakrishnan, B. Sikdar, and S. Kalyanaraman, "On randomizing the sending times in TCP and other window based algorithms," Computer Networks. 50(3). pp.422-447. Feb. 2006.

[22] H. Wu, Z. Feng, Ch. Guo, and Y. Zhang, "ICTCP: Incast Congestion Control for TCP in Data Center Networks," in ACM *CoNEXT*, 2010.

[23] http://tools.ietf.org/html/rfc3465.

[24] R. Jain, A. Durresi, and G. Babic, "Throughput Fairness Index: An Explanation," ATM Forum/99-0045, February 1999.

[25] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in ACM *SIGCOMM*, 2010.