RepNet: Cutting Latency with Flow Replication in Data Center Networks

Shuhao Liu, Hong Xu, Libin Liu, Wei Bai, Kai Chen, and Zhiping Cai

Abstract—Data center networks need to provide low latency, especially at the tail, as demanded by many interactive applications. To improve tail latency, existing approaches require modifications to switch hardware and/or end-host operating systems, making them difficult to be deployed. We present the design, implementation, and evaluation of RepNet, an application layer transport that can be deployed today. RepNet exploits the fact that only a few paths among many are congested at any moment in the network, and applies simple flow replication to mice flows to opportunistically use the less congested path. RepNet has two designs for flow replication: (1) RepSYN, which only replicates SYN packets and uses the first connection that finishes TCP handshaking for data transmission, and (2) RepFlow which replicates the entire mice flow. We implement RepNet on `node.js`, one of the most commonly used platforms for networked interactive applications. `node`'s single threaded event-loop and non-blocking I/O make flow replication highly efficient. Performance evaluation on a real network testbed and in Mininet reveals that RepNet is able to reduce the tail latency of mice flows, as well as application completion times, by more than 50%.

Index Terms—Data center networks, latency, flow replication.

1 Introduction

As modern Web services become data-driven and interactive (*e.g.,* web search and social networks), their Quality-of-Service tends to have a higher demand in computation capacity and a more strict requirement on response times. Such applications are usually housed in data centers, where abundant distributed computing and networking resources are readily available.

Data center networks, in particular, are tasked to provide very low latency for many interactive applications [8], [10], [53]. Low *tail* latency (e.g. 99%ile or 99.9%ile) is especially important, since completing a request depends on all (or most) of the responses from many worker machines [19]. Unfortunately current data center networks are not up to this task: Many report that the tail latency of short TCP flows can be more than 10x worse than the average in production networks, even when the network is only lightly loaded [10], [52], [53]. The main reason for long tail latency is that elephant and mice flows co-exist in data center networks. While most flows are mice with less than say 100 KB, most bytes are in fact from elephant flows much fewer in number [8], [22], [32]. Thus mice flows are often queued behind bursts of packets from elephants in switches, resulting in long queueing delay and flow completion time (FCT).

The problem has attracted much attention recently in our community. Loosely speaking, existing work reduces the tail latency by: (1) reducing the queue length, [8], [9], [36]; (2) prioritizing mice flows, [10], [13], [28], [50]; and (3) engineering better multi-path schemes, [26], [49], [53]. While effective, they require changes to switches and/or end-hosts, and face deployment challenges. Thus there is a growing need for an application layer solution that provides immediate latency gains without an infrastructure overhaul.

To this end, we introduce RepNet, a low latency transport at the application layer that can be readily deployed in current infrastructures. RepNet is based on the simple idea of flow replication. The key insight is the observation that multi-path diversity, which is readily available with high bisection bandwidth topologies such as fat-tree [6], is an effective means to combat performance degradation that happens in a random fashion. Flash congestion due to bursty traffic and imperfect load balancing happen randomly in any part of the network at any time. As a result, congestion levels on different paths are statistically independent. In RepNet, the replicated and original flow are highly likely to traverse different paths, and the probability that both experience long queueing delay is much smaller. RepNet targets general clusters running mixed workloads, where short flows typically represent a very small fraction ($<$ 5%) of overall traffic according to measurements [8], [22]. Additionally, flow replication is *orthogonal* to all TCP-friendly proposals in the literature. Thus it can be used together with schemes such as DCTCP [8] and pFabric [10], providing even more benefit in reducing latency.

In this paper we make three contributions in designing, implementing, and evaluating RepNet based on flow replication.

First, we design RepNet with two schemes of flow repli-

- *The work was supported in part by the Hong Kong RGC ECS-21201714, GRF-11202315, and CRF-C7036-15G. Part of this work was presented at IEEE INFOCOM 2014. The corresponding author is Hong Xu.*

- *S. Liu is with University of Toronto, Toronto, ON, M5S 3G4 Canada (email: shuhao@ece.utoronto.ca). The work was done when he was with City University of Hong Kong, Kowloon, Hong Kong.*
H. Xu and L. Liu are with City University of Hong Kong, Kowloon, Hong Kong (email: henry.xu@cityu.edu.hk, libinliu-c@my.cityu.edu.hk).
W. Bai and K. Chen are with Hong Kong University of Science and Technology, Kowloon, Hong Kong (email: wbaiab@cse.ust.hk, kaichen@cse.ust.hk).
Z. Cai is with National University of Defence Technology, China (email: zpcai@nudt.edu.cn).

Manuscript received xxx xx, 2017; revised xxx xx, 2017.

cation, RepFlow and RepSYN, achieving different trade-off points for different use cases. Both directly use existing TCP protocols deployed in the network. RepFlow replicates each short TCP flow by creating another TCP connection to the receiver, and sending identical packets for both flows. The application uses the first flow that finishes the transfer. RepFlow fully reaps the benefits of replication at the cost of a small amount of redundancy, and works in most cases. Yet an astute reader might be concerned about the use of RepFlow in incast scenarios where many senders transmit at the same time to a common destination causing throughput collapse [47]. RepFlow potentially aggravates the incast problem. To address this, we design RepSYN which only replicates the SYN packet on the second TCP connection, and uses the connection that finishes handshaking first for data transmission.

Second, we implement RepNet with both RepFlow and RepSYN on `node.js` [1] as a transport module that can be directly used by existing applications running in data centers. `node.js` (or simply `node`) is a server-side JavaScript platform that uses a single-threaded event-loop with a non-blocking I/O model, which makes it ideal for replicating TCP flows with minimal performance overhead. Moreover, `node` is widely used for developing the back-end of large-scale interactive applications in production systems at LinkedIn [4], Microsoft, Alibaba, etc.[1] RepNet is implemented as an asynchronized socket programming library for `node`. For compatibility and ease of deployment, it exposes the same set of APIs as the standard network library (`Net`), masking the required flow replication and redundancy removal mechanisms behind the scene. Thus, RepNet on `node` potentially provides immediate latency benefit for a large number of these applications with minimal code change.

Our third contribution is a comprehensive performance evaluation of RepNet using queueing analysis (Sec. 3), testbed experiments (Sec. 5.2-5.3), and Mininet emulation (Sec. 5.4). We develop a simple M/G/1 queueing model to analyze mean and tail FCT in data center networks. Our model shows that the diversity gain of replication can be understood as a reduction in the effective traffic load seen by short flows, which leads to improved queueing delay and FCT. We perform testbed evaluation on a small scale leaf-spine network with five Pronto 3295 switches, and a larger scale Mininet emulation with a 6-pod fat-tree [25], using an empirical flow size distribution from a production network [8]. Our evaluation shows that, both RepFlow and RepSYN reduce the tail latency of mice flows, especially under high loads, by more than 50%. RepSYN is less effective compared with RepFlow in most cases, but it remains beneficial in incast scenarios where RepFlow suffers from performance degradation. We further implement a bucket sort application using RepNet, and observe that both RepFlow and RepSYN improves the application level completion times by around 50%. The implementation code [3], and scripts used for performance evaluation, are available online [2]. We are in the process of making RepNet available as an NPM (Node Package Manager) module for the `node` user community.

1. https://github.com/nodejs/node/wiki/Projects,-Applications,-and-Companies-Using-Node

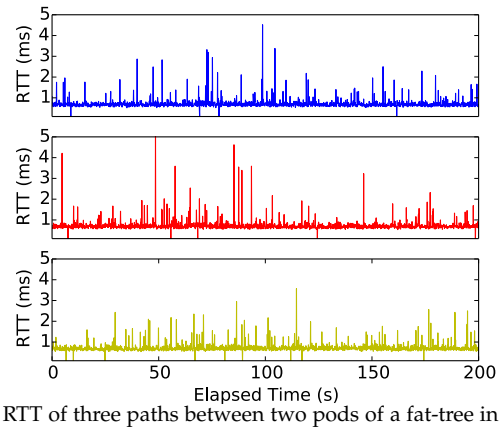


Fig. 1: RTT of three paths between two pods of a fat-tree in Mininet.

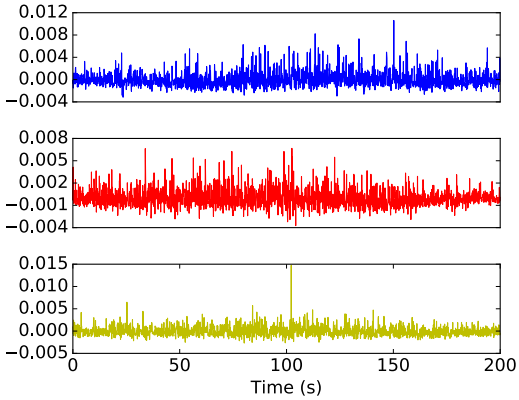


Fig. 2: Cross-covariance of each pair of RTT time series measured in Fig. 1. The absolute value of cross-covariance is always less than 0.015, implying that the time series are uncorrelated.

2 Motivation and Design

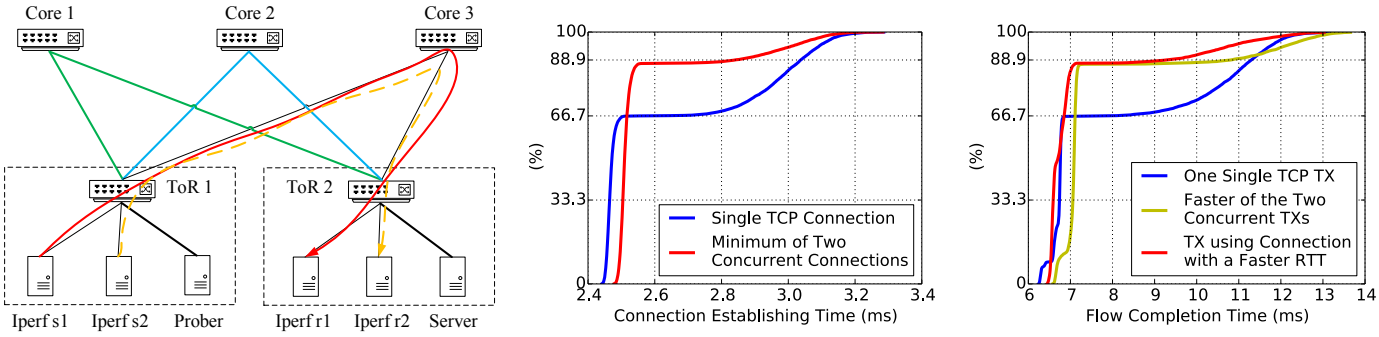
Let us start by motivating the idea of flow replication to reduce latency in data center networks, followed by the high-level design of RepNet including both RepFlow [51] and RepSYN.

2.1 Motivation

Today's data center networks are usually constructed with Clos topologies [11]. In these networks, many paths of equal distance exist between a pair of hosts. Equal-cost multi-path routing, or ECMP, is used to perform flow-level load balancing [29] that routes packets based on the hash value of the five-tuple in the packet header. Due to the randomness of traffic and ECMP, congestion happens randomly in some paths of the network, while many others are not congested at all.

We experimentally validate this observation using Mininet [25] with real traffic traces from data centers. We construct a 6-pod fat-tree without oversubscription, with 3 hosts per rack. Traffic traces from a web search cluster [8] are used to generate flows with average link load of 0.3, representing the common utilization figure in production networks [43]. To measure RTT as indicator of congestion, we configure 3 hosts in one rack to ping 3 hosts of another rack in a different pod, respectively. A POX controller is configured to route the 3 ICMP sequences to 3 distinct paths between the two ToR switches. The interval of ping is 100 ms and the measurement lasts for 200 seconds. The RTT results

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2793250, IEEE Transactions on Services Computing



(a) Experiment leaf-spine topology. (b) CDF of the measured RTTs. (c) CDF of 100KB mice flow FCTs.

Fig. 3: Experimental evaluation results to verify our motivation for flow replication.

are shown in Fig. 1. It highlights two key characteristics of data center traffic: (1) RTT on a single path is low most of the time, indicating no congestion; (2) the occurrences of flash congestion, which results in occasional peaks in the RTT, are *uncorrelated* on different paths according to the cross-covariance analysis shown in Fig. 2; and (3) it is rare that all paths are congested at the same time.

This form of path diversity motivates the idea of flow replication [51]. By trading a small amount of traffic redundancy for a higher degree of connectivity, replication considerably lowers the probability of transmission experiencing long latency. Theoretically speaking, if the proportion of congested paths between two end hosts is p, then the probability of a flow being delayed due to congestion is lowered from p to p^2 after replication. Since the hot spots in data center networks are typically scarce, we have $p \ll 1$, such that $p^2 \ll p$.

This simple intuition is verified in our testbed (more details about the testbed in Sec. 5.1). We establish a small leaf-spine topology with three paths between two racks as shown in Fig. 3(a). We generate long-live flows using `iperf` that congest one of the paths as illustrated in Fig. 3(a). Two senders, s1 and s2 in the left rack, are communicating with r1 and r2 in the right rack, respectively. We are able to confirm that two TCP flows are routed to the same path and they are sending at half the link rate (∼500Mbps) each. Meanwhile, the other two paths are idle.

We then measure RTT between the *prober* in the left rack and the *server* in the right rack, which is shown in Fig. 3(b). The RTT is measured at the application layer during TCP handshaking. Specifically, the *prober* opens a TCP connection by sending a SYN packet to the *server* and starts timing. The timing stops as soon as the connection is established successfully (when the ACK to the SYN-ACK is sent by the *prober*). We collect 10K RTT samples. As seen from Fig. 3(b), the RTT distribution in our real testbed matches our probability analysis in the motivation example well. That is, with ECMP, a redundant TCP connection can lower the probability of choosing a congested path from p ($\frac{1}{3}$ in this case) to p^2 ($\frac{1}{9}$).

We also collect FCTs of 100 KB mice flows, whose CDFs are illustrated in Fig. 3(c), using three methods: (1) Send the flow with one TCP. (2) Send the same flow using two concurrent TCP connections, and record the FCT of the first one that finishes. (3) Start two TCP connections at the same time first, then send the payload through the connection that finishes handshaking first. Clearly, the CDF of FCT in Fig. 3(c) show a similar trend to that of RTT in Fig. 3(b), suggesting that the RTT of SYN packets can reasonably reflect the congestion of the chosen path. These observations motivate the design of RepFlow and RepSYN.

2.2 RepNet Design

RepNet comprises of two mechanisms: RepFlow [51] and RepSYN. We heuristically mandate that flows less than or equal to 100KB are considered short flows, and are replicated to achieve better latency. This threshold value is chosen in accordance with many existing papers [8], [10], [28], [39]. Thus in both mechanisms, only mice flows less than 100 KB are replicated. This can be easily changed for different networks.

RepFlow uses flow replication to exploit multi-path diversity. It does not modify the transport protocol, and thus works on top of TCP as well as any other TCP variants, such as DCTCP [8] and D2TCP [46]. RepFlow realizes flow replication by simply creating two TCP sockets for transmitting identical data for the same flow. Though conceptually simple, RepFlow doubles the number of bytes sent for the flow. Further, it may aggravate throughput collapse in incast scenarios, when flows sending concurrently to the same host [47].

We thus design RepSYN to overcome RepFlow's shortcomings. The idea is simple: we establish two TCP connections as in RepFlow. However data is only transmitted using the first established connection, and the other is ended immediately. Essentially SYN is used to probe the network and find a better path. The delay experienced by the SYN reflects the latest congestion condition of the corresponding path. RepSYN only replicates SYN packets and clearly does not aggravate incast compared to TCP.

Since RepSYN replicates SYN only and incurs ignorable traffic overheads, it may be more beneficial to establish even more TCP connections. The the replication factor can vary based on the path diversity in the data center network, and it should also take into account the additional system resource consumption.

2.3 Discussion

One possible concern is that, in the application layer, the observable latency of establishing a TCP connection does

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2793250, IEEE Transactions on Services Computing

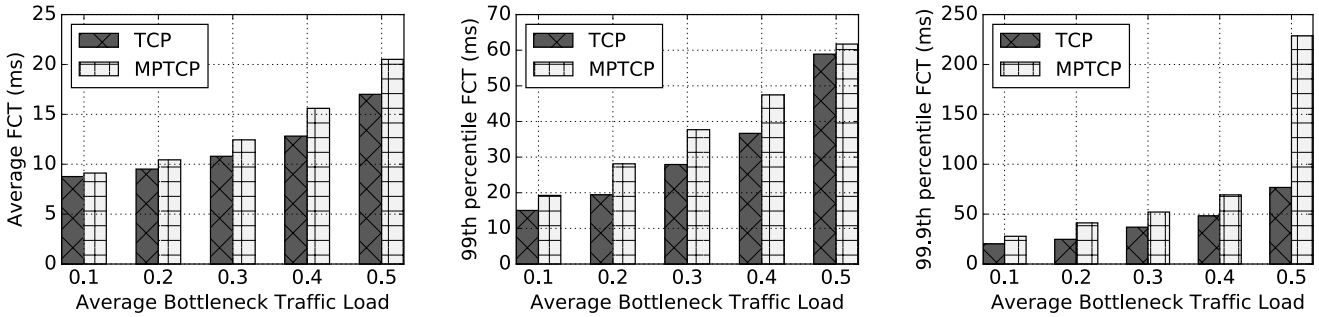


Fig. 4: FCT comparison between TCP and MPTCP for mice flows (<100KB) when network oversubscription is 2:1.

not completely reflect the RTT experienced by the SYN. However, it provides sufficient information for the application to *validate* the relative latency along the two paths, and determine which path is relatively better.

One may suggest MPTCP [24] as a better choice of transport, as it is also designed to exploit multiple available paths in data center networks. is a common transport protocol to exploit multi-path diversity in networks. As compared to MPTCP, RepNet differs in several important aspects. First, RepNet is a pure application-layer solution, while MPTCP requires kernel upgrades. Second, they have different objectives. MPTCP aims to increase the throughput for large flows, while RepNet focuses on minimizing the latency experienced by critical mice flows. Third, they have different designs because of different objectives. MPTCP strips a single copy of data and sends it via multiple paths, while RepNet sends a redundant replica. Completing a small flow in MPTCP requires each part to be received successfully, which is often slower compared to TCP.

It is reported that average and tail FCT in MPTCP is over 20% worse than TCP for mice flows less than 100 KB [7]. We also conduct experiments on our testbed that validate this observation. The results are shown in Fig. 4, which compares the FCT of mice flows achieved by TCP and MPTCP, respectively. In terms of 99.9%ile FCT, MPTCP is over 40% worse than TCP regardless of the traffic load.

RepNet lends itself to many implementation choices. Regardless of the detail, it is crucial to ensure path diversity is utilized, i.e. the five-tuples of the original and replicated flow have to be different (assuming ECMP is used). In our implementation we use different destination port numbers for this purpose.

3 Queueing Analysis

The ideas of RepFlow and RepSYN presented in Sec. 2.1 are simple and intuitive. In this section we first present a queueing analysis of flow completion times in data centers to theoretically understand the benefits and overhead of replication. We only present analysis for RepFlow. Analyzing RepSYN requires modeling the conditional expectation and tail queueing delay which is significantly more challenging, and we leave it to future work.

3.1 Queueing Model

A rich literature exists on TCP steady-state throughput models for both long-lived flows [37], [40] and short flows [27]. There are also efforts in characterizing the completion times of TCP flows [15], [35]. See [15] and references therein for a more complete literature review. These models are developed for wide-area TCP flows, where RTTs and loss probabilities are assumed to be constants. Essentially, these are open-loop models. The data center environment, with extremely low fabric latency, is distinct from the wide-area Internet. RTTs are largely due to switch queueing delay caused by TCP packets, the sending rate of which in turn are controlled by TCP congestion control reacting to RTTs and packet losses. This closed-loop nature makes the analysis more intriguing [42].

Our objective is to develop a simple FCT model for TCP flows that accounts for the impact of queueing delay due to long flows, and demonstrates the potential of RepNet in data center networks. We do not attempt to build a fine-grained model that accurately predicts the mean and tail FCT, which is left as future work. Such a task is potentially challenging because of not only the reasons above, but also the complications of timeouts and retransmissions [41], [47], switch buffer sizes [12], [35], etc. in data centers.

We construct our model based on some simplifying assumptions. We abstract one path of a data center network as a M/G/1 first-come-first-serve (FCFS) queue with infinite buffer. Thus we do not consider timeouts and retransmissions. Flows arrive following a Poisson process and have size $X \sim F(\cdot)$. Since TCP uses various window sizes to control the number of in-flight packets, we can think of a flow as a stream of bursts arriving to the network. We assume the arrival process of the bursts is also Poisson. One might argue that the arrivals are not Poisson as a burst is followed by another burst one RTT later (implying that interarrival times are not even i.i.d). However queueing models with general interarrival time distributions are difficult to analyze and fewer results are available [21]. For tractability, we rely on the commonly accepted M/G/1-FCFS model [10], [12]. We summarize some key notations in the table below. Throughout this paper we consider (normalized) FCT defined as the flow's completion time normalized by its best possible completion time without contention.

For short flows, they mostly stay in the slow-start phase for their life time [12], [15], [20], [35]. Their burst sizes depend on the initial window size k. In slow-start, each flow first sends out k packets, then $2k$, $4k$, $8k$, etc. Thus, a short flow with X packets will be completed in $\log_2(X/k+1)$ RTTs, and its normalized completion time can be expressed

TABLE 1: Key notations.

M	maximum window size (64KB, 44 packets)
S_L	threshold for long flows (100KB, 68 packets)
$F(\cdot), f(\cdot)$	flow size CDF and PDF
$\rho \in [0, 1)$	overall traffic load
W	queueing delay of the M/G/1-FCFS queue
k	initial window size in slow-start

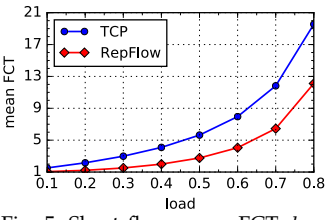


Fig. 5: Short flow mean FCT. $k = 3$ packets, flow size distribution from the web search workload [8].

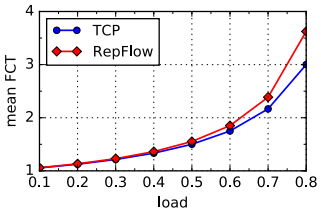


Fig. 6: Large flow mean FCT. $k = 3$ packets, flow size distribution from the web search workload [8].

as

$$FCT_X = \sum_{i=1}^{\log_2(X/k+1)} W_i/X + 1, \tag{1}$$

assuming link capacity is 1 packet per second.

For long flows larger than S_L, we assume that they enter the congestion avoidance phase immediately after it arrives [37], [40]. They continuously send bursts of a fixed size equal to the maximum window size M (64KB by default in Linux). A large flow's FCT is then

$$FCT_X^L = \sum_{i=1}^{X/M} W_i/X + 1, X \geq S_L. \tag{2}$$

3.2 RepFlow: Quantitative Analysis

We now present a quantitative analysis of FCT for RepFlow.

3.2.1 Mean FCT Analysis

Proposition 1. The mean FCT of short TCP flows can be expressed by

$$E[FCT] = \frac{\rho M}{2(1-\rho)} \int_0^{S_L} \frac{\log_2(x/k+1)}{x} \frac{f(x)}{F(S_L)} \mathrm{d}x + 1. \tag{3}$$

If RepFlow is used, their mean FCT becomes

$$E[FCT_{rep}] = \frac{(1+\epsilon)^2\rho^2 M}{2(1-(1+\epsilon)^2\rho^2)} \int_0^{S_L} \frac{\log_2(x/k+1)}{x} \frac{f(x)}{F(S_L)} \mathrm{d}x + 1. \tag{4}$$

Proof: See Appendix A for a detailed derivation. □

Apparently, the mean FCT for short TCP flows calculated by (3) depends on the load of the network and the flow size distribution. We use $k = 3$ packets as the TCP initial window size, which is consistent with our testbed setting in Sec. 5.1. Using the same flow size distribution as in Sec. 2.1 from a web search data center [8], Fig. 5 plots the FCT with varying load.

For FCT of Repflows (4), given small $\epsilon \leq 0.1$, $(1+\epsilon)^2\rho^2$ is much smaller than ρ. As ρ increases the difference is smaller. However the factor $\rho/(1-\rho)$ that largely determines the queueing delay $E[W]$ and FCT is very sensitive to ρ in high loads, and a small decrease of load leads to significant decrease in FCT. In the same Fig. 5, we plot FCT for RepFlow with the same web search workload [8], where 95% of bytes are from long flows, i.e. $\epsilon = 0.05$. Observe that RepFlow is able to reduce mean FCT by a substantial margin compared to TCP in all loads.

Our analysis reveals that intuitively, the benefit of RepFlow is due to a significant decrease of effective load experienced by the short flows. Such a load reduction can be understood as a form of multi-path diversity discussed earlier as a result of multi-path network topologies and randomized load balancing.

At this point one may be interested in understanding the drawback of RepFlow, especially the effect of increased load on long flows. We now perform a similar FCT analysis for long flows. For a large flow with $X > S_L$ packets,we thus have

$$E[FCT^L] = \frac{\rho M}{2(1-\rho)} \frac{X}{M \cdot X} + 1 = \frac{\rho}{2(1-\rho)} + 1. \tag{5}$$

The mean FCT for long flows only depends on the traffic load. With RepFlow, load increases to $(1+\epsilon)\rho$, and FCT becomes

$$E[FCT_{rep}^L] = \frac{(1+\epsilon)\rho}{2(1-(1+\epsilon)\rho)} + 1, \tag{6}$$

For long flows, load only increases by ϵ, whereas small flows see a load decrease of $1-(1+\epsilon)^2\rho$. long flows are only mildly affected by the overhead of replication. Fig. 6 plots the mean FCT comparison for long flows.

3.2.2 99%ile FCT Analysis

To determine the latency performance at the extreme cases, such as the 99%ile FCT [8], [9], [28], [53], we need the probability distribution of the queueing delay, not just its average. This is more difficult as no closed form result exists for a general M/G/1 queueing delay distribution. Instead, we approximate its tail using the effective bandwidth model [34], which leads to the following proposition:

Proposition 2. With and without RepFlow being applied, the tail FCT for long flows can be expressed as follows, respectively:

$$\tilde{FCT}^L = E[FCT^L] + (2\ln 10 - 1)E[W] \cdot P, \tag{7}$$

$$\tilde{FCT}_{rep}^L = E[FCT_{rep}^L] + (2\ln 10 - 1)E[W_{rep}^L] \cdot P, \tag{8}$$

$$\text{where } P = \int_{S_L}^{\infty} \frac{1}{x} \frac{f(x)}{1-F(S_L)} \mathrm{d}x,$$

$$E[W_{rep}^L] = \frac{(1+\epsilon)\rho M}{2(1-(1+\epsilon)\rho)}.$$

Proof: See Appendix B for a detailed derivation. □

Fig. 8 shows the numerical results. Long flows enjoy better tail FCT compared to short flows, since their transmission lasts for a long time and is not sensitive to long-tailed queueing delay. Again observe that RepFlow does not penalize long flows.

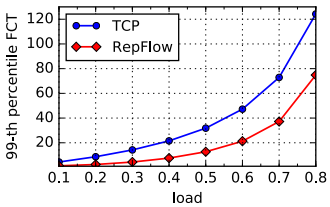


Fig. 7: Short flow tail FCT. $k = 3$ packets, flow size distribution from the web search workload [8].

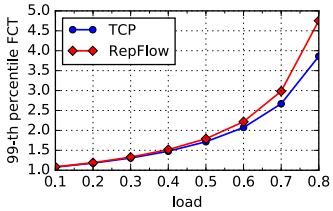


Fig. 8: Large flow tail FCT. $k = 3$ packets, flow size distribution from the web search workload [8].

3.3 Summary

We summarize our analytical findings. Short flow mean and tail FCT depend critically on queueing delay, and the factor $\frac{\rho}{1-\rho}$ assuming a M/G/1-FCFS queue.

Using replication, they have much less probability of entering a busy queue, and the effective load they experience is greatly reduced. This confirms the intuition that RepNet provides path diversity gains in data center networks.

4 Implementation

We now describe our implementation of RepNet with `node`. The source code is available online [3].

4.1 Why `node`?

On a high level, `node` is a highly scalable platform for real-time server-side networked applications. It combines single-threaded, non-blocking socket with the event-driven philosophy of JavaScript. It runs on Google V8 engine with core libraries optimized for performance and scalability [1].

The first reason for choosing `node` is *efficiency*. Replication introduces the overhead of launching additional TCP connections. To provide maximal latency improvements, we need to minimize this overhead. This rules out a multi-threaded implementation using for example `Tornado` or `Thrift` [44]. For one thing, replicating mice flows nearly doubles the number of concurrent connections a server needs to handle. For the other, the necessary status synchronization between the original connection and its replica demands communication or shared memory across threads. For applications with I/O from a large number of concurrent connections, a multi-threaded RepFlow will be burdened by frequent thread switching and synchronization [45] with poor performance and scalability. In fact, we tried to implement RepNet on `Thrift` based on python, and found that the performance is unacceptable.

`node` satisfies our requirement for high efficiency. Specifically, its non-blocking I/O model in a single thread greatly alleviates the CPU overhead. Asynchronous sockets in `node` also avoid the expensive synchronization between the two connections of RepFlow. For example, it is complex to choose a quicker completion between two `socket.read` operations using blocking sockets: three threads and their status sharing will be needed. Instead, `node` relies on callback of the `'data'` event to handle multiple connections in one thread, which greatly reduces complexity. The thread stack memory footprint (typically 2MB per thread) is also reduced.

The second reason we choose `node` is that it is widely deployed in production systems for companies such as LinkedIn, Microsoft, etc. [4]. Besides deployment in front-end web servers to handle user queries, a large number of companies and open source projects rely on `node` at the back-end for compatibility.[2] `node` integrates smoothly with NoSQL data stores, e.g. MongoDB,[3] and caches, e.g. memcached,[4] and enables a full JavaScript stack for the ease of application development and maintenance. For these reasons, `node` is commonly used in data centers to fetch data across server machines. Thus implementing RepNet on it is likely to benefit a large audience and generate immediate impact to the industry.

4.2 Overview

Before we evaluate RepNet on a real-world infrastructure to verify the promising theoretical analysis, we first present its implementation. RepNet is implemented based upon the `Net`[5] module, `node`'s standard library for non-blocking socket programming. Similar to `Net`, RepNet exposes some socket functions, and wraps useful asynchronous network methods to create even-driven servers and clients, with additional low latency support by flow replication.

We implement RepNet with the following objectives:

Transparency. RepNet should provide the same set of APIs as `Net`, making it transparent to applications. That is, to enable RepFlow, one only needs to include `require('repnet')` instead of `require('net')`, without changing anything else in the existing code. By default, RepNet uses RepFlow for small flows (with a threshold size $\leq$ 100 KB). Users can customize the parameters to switch to RepSYN or tune the threshold of mice flows.

Though RepNet offers complete transparency, developers can still enjoy the flexibility to use it. In most cases, developers are fully aware of the particular applications that generate small flows that are latency-critical. They can explicitly apply RepNet on these flows, while leaving other TCP connections unchanged.

In cases where developers are not aware, there is an option to blindly automate this selection, with some predictable overheads. In RepSYN, the strategy can be applied to all flows. In RepFlow, each flow can be initially replicated after the connection is established. Then, as soon as enough data (*e.g.*, 100KB) has been sent out, RepNet can close the slower connection and stay with a single TCP connection.

Compatibility. A RepNet server should be able to handle regular TCP connections at the same time. This is required as elephant flows are not replicated.

RepNet consists of two classes: `RepNet.Socket` and `RepNet.Server`. `RepNet.Socket` implements a replication capable asynchronous socket at both ends of a connection. It maintains a single socket abstraction for applications while performing I/O over two TCP sockets. `RepNet.Server` provides functions for listening for and managing both replicated and regular TCP connections. Note that `RepNet.Server` does not have any application logic. Instead, it creates a connection listener at the server

2. https://github.com/nodejs/node/wiki/Projects,-Applications,-and-Companies-Using-Node
3. www.mongolab.com/node-js-platform
4. https://nodejsmodules.org/pkg/memcached
5. http://nodejs.org/api/net.html

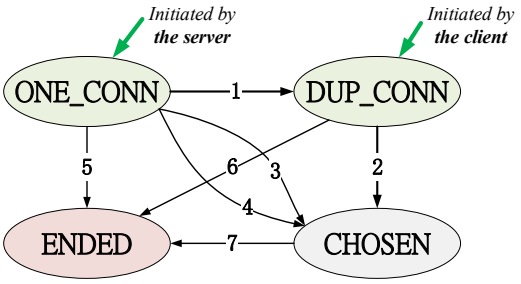


Fig. 9: The FSM of `RepNet.Socket`.

side, which responds to SYN packets by establishing a connection and emitting a connected `RepNet.Socket` object in a corresponding callback for applications to use.

We now explain the high-level design and working of RepNet by examining the lifetime of a RepFlow transmission. The case of RepSYN is similar. First, the server runs a `RepNet.Server` that listens on two distinct ports. This is to make sure that the original and replicated flows have different five-tuples and traverse different paths with ECMP. When the client starts a RepFlow connection, a `RepNet.Socket` object is instantiated. Two `Net.Socket` objects, being two members of the `RepNet.Socket` object, will send SYN packets to the two ports on the receiver, respectively. They share the same source port number though, so the server can correctly recognize them among potentially many concurrent connections it has.

Now our server may not get the two SYN packets at the same time. To minimize delay, upon the arrival of the first SYN, the server responds immediately by emitting a new `RepNet.Socket`, using one member `Net.Socket` to process handshaking while creating another null `Net.Socket`. The first TCP connection is then established and ready for applications to use right away.

The server now waits for the other connection. Its `RepNet.Server` maintains a waiting list of connections — represented by `<ip_addr:port>` tuples — whose replicas has yet to arrive. When the second SYN arrives, the server matches it against the waiting list, removes the connection from the list, and has the corresponding `RepNet.Socket` instantiate the other member `Net.Socket`. This second TCP connection will then proceed. At this point, both sides can send data using RepFlow, as two complete `RepNet.Socket` objects. Note that the server also handles standard TCP connection. In this case a second SYN will never arrive and can be detected by timeout.

Our implementation is based on `node 0.11.13`. We introduce more details of our implementation in the following.

4.3 Class: `RepNet.Socket`

The key difference between `RepNet.Socket` and `Net.Socket` is the I/O implementation. Since a `RepNet.Socket` has two TCP sockets, a Finite State Machine (FSM) model is used to handle the asynchronous I/O across them. For brevity, all four states of the FSM are listed in Table 2. Figure 9 shows the possible state transitions with more explanation in Table 3.

The client, who initiates the connection, always starts in `DUP_CONN`, and `socket.write()` in RepNet is done by calling `socket.write()` of both member `Net.Socket` objects to send data out. The server always starts in `ONE_CONN` waiting for the other SYN to arrive, and when it does enters `DUP_CONN`. In both states read operations are handled in the callback of a `'data'` event. A counter is added for each connection to coordinate the detection of new data. As soon as new chunks of buffer are received, `RepNet.Socket` emits its `'data'` event to the application.

For the server, if there are writes in `ONE_CONN`, they are performed on the active connection immediately and archived for the other connection with the associated data. If the archived data exceeds a threshold, the server enters `CHOSEN` and disregards the other connection. The server may also enter `CHOSEN` after timeout on waiting for the other connection, which corresponds to the standard TCP.

4.4 Class: `RepNet.Server`

`RepNet.Server` has two `Net.Server` objects which listen on two distinct ports. The key component we add is the waiting list which we explain now.

The waiting list is a frequently updated queue. Each flow in the waiting list has three fields: `TTL`, `flowID` (the client's `<ip_addr:port>` tuple), and `handle` (a pointer to the corresponding `RepNet.Socket` instance).

There are three ways to update the list:

Push. If a new SYN arrives and finds no match in the list, a new `RepNet.Socket` object is emitted and its corresponding flow will be pushed to the list.

Delete. If a new SYN arrives and it matches with an existing flow, the corresponding `RepNet.Socket` object is then completed and this flow is removed from the list.

Timeout. If the flow stays on the list for too long to be matched, it is timed out and removed. This timeout can be adjusted by setting the `WL_TIMEOUT` option. The default is equal to RTO of the network. A higher value of `WL_TIMEOUT` may decrease the probability of matching failures, at the cost of increasing computation and memory.

Note that to achieve transparency by exposing the same APIs as `Net.Server`, the constructor of `RepNet.Server` accepts only one port number parameter. It simply advances the number by one for the second port. An `error` event will be emitted if either of the port is already in use.

4.5 RepSYN

As explained in Sec. 2.2, we propose RepSYN to alleviate RepFlow's drawbacks in incast scenarios. A RepSYN client can work compatibly with a `RepNet.Server`. Specifically, once the second connection is established and the server-side socket enters `DUP_CONN`, it would be reset immediately by the client to trigger the transition to `CHOSEN` in Table 3. RepSYN can be activated by setting the `Flag_RepSYN` flag of the `RepNet.Socket` object.

5 Evaluation

We evaluate RepNet using both testbed experiments and Mininet emulation. Our evaluation focuses on four key questions:

- **How does RepNet perform in practice?** With a real-world flow size distribution [8], we show that for

State	Description	On Waiting List	Performing I/O on
ONE_CONN	Only one `Net.Socket` is open. The other one is pending.	Yes	The only connection.
DUP_CONN	Both member `Net.Socket` objects are open.	No	Both connections.
CHOSEN	One of `Net.Socket` objects is no longer valid.	Depend on State	The chosen connection.
ENDED	The `RepNet.Socket` is ended.	No	N/A

TABLE 2: All states in the FSM.

Transition	Trigger	Additional Consequence
1	The slower connection is detected at the server.	The corresponding flow is removed from the waiting list. The replicated connection is binded with the matching one.
2	One connection raises an exception, or emits an `'error'` event.	The abnormal connection is abandoned by calling the `destroy()` function and resetting the other end.
3	The corresponding flow in the waiting list is timed out.	The item is deleted from the waiting list.
4	The archived data for writes exceeds the threshold.	The corresponding flow will NOT be removed from the waiting list until the second SYN arrives for correctness.
5, 6, 7	Both connections are destroyed or ended.	

TABLE 3: Trigger of the state transitions

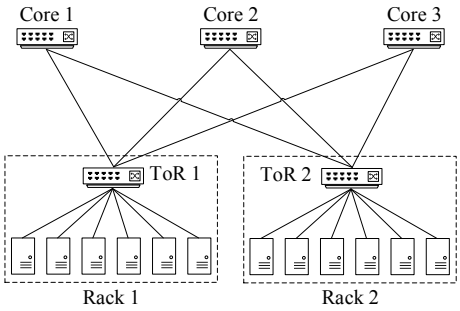


Fig. 10: The leaf-spine topology of the testbed.

mice flows, RepFlow and RepSYN provide up to ∼69% and ∼57% 99.9%ile latency reduction, respectively, over linux stack TCP. RepFlow and RepSYN also reduce the 99%ile and median latency. The small replication overhead does not impact the FCT of elephant flows.

- **How does RepNet perform under incast scenarios?** We show that in our testbed with 11-to-1 incast (22-to-1 for RepFlow), RepFlow still provides lower 99.9%ile latency, but suffer from higher FCT at the 99%ile due to the aggravation of incast. RepSYN, on the other hand, is indeed effective in reducing both 99%ile and 99.9%ile FCT in incast scenarios.
- **How does applications benefit from RepNet?** We implement a distributed bucket sort application with partition-aggregate workflows using RepNet. The testbed results show that its 99.9%ile job completion time is reduced by ∼45%, and 99%ile job completion time by ∼50% with both RepFlow and RepSYN.
- **Does RepNet work well in a large scale?** Using Mininet emulation with a 6-pod fat-tree and the web search workload [8], we show that RepNet still provides lower median and tail FCT for mice flows when the network load is larger than 0.4.

5.1 Testbed Setup

Our testbed uses Pronto 3295 48-port Gigabit Ethernet switches with 4MB shared buffer. The switch OS is PicOS 2.04 with ECMP enabled. Our testbed server has an Intel E5-1410 2.8GHz CPU (8-thread quad-core), 8GB memory, and a Broadcom BCM5719 NetXtreme Gigabit Ethernet NIC.

The servers run Debian 6.0 64-bit Linux, kernel version 2.6.38.3. We change RTO_{min} to 10 ms in order to remedy the impact of incast and packet retransmission [47]. We found that setting it to a value lower than 10 ms leads to system instability in our testbed. The initial window size is 3, i.e. about 4.5 KB payload. The initial RTO is 3 seconds by default in the kernel, which influences our experiments in cases where TCP connections fail to establish at the first time. We tried to set it to a smaller value, but found that kernel panics occur frequently because of fatal errors experienced by the TCP keep-alive timer.

Topology. The testbed uses a leaf-spine topology depicted in Fig. 10 which is widely used in production [7], [10], [26]. There are 12 servers organized in 2 racks, and 3 spine switches which provide three equal-cost paths between two hosts under different ToRs. The ping RTT is ∼178 μs across racks. The topology is oversubscribed at 2:1 when all hosts are used. We also conduct experiments without oversubscription, by shutting down half of the servers in each rack.

Although our testbed is small in scale, it gives us a glimpse of how RepFlow and RepSYN are beneficial in production environments. The benefits of RepFlow and RepSYN are more salient in actual datacenters. The reason is that large-scale datacenters provide more paths, and more diversity gains between a pair of nodes. Also the probability of two TCP flows being routed to the same path is lower, meaning that more mice flows can actually benefit from replication. In the current configuration, this probability is $\frac{1}{9}$, when the redundant TCP connection fails to find an alternate path. In this case, RepFlow and RepSYN are no longer useful.

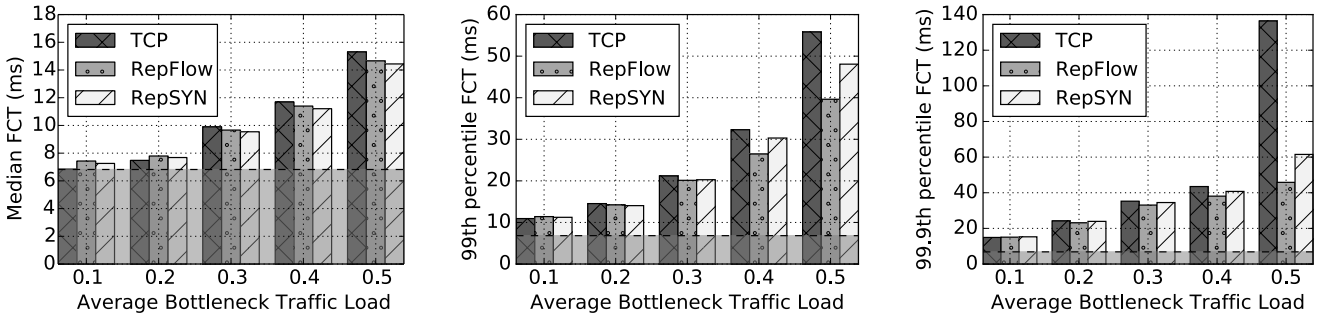


Fig. 11: FCT/NFCT comparison when network oversubscription is 1:1. The shadowed area indicates the estimated baseline software overhead.

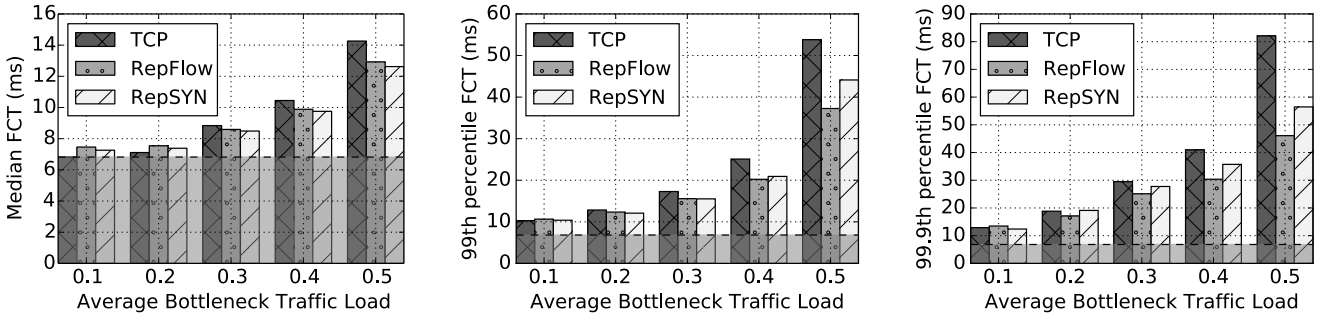


Fig. 12: FCT/NFCT comparison when network oversubscription is 2:1. The shadowed area indicates the estimated baseline software overhead.

5.2 Performance under Empirical Traffic Workload

5.2.1 Workload Generation

We use the flow size distribution from a web search workload [8] to drive our experiments. Most flows (∼60%) in this workload are mice flows smaller than 100KB, though over 95% of the bytes are from 30% of flows larger than 1MB.

Flows are generated between random pairs of servers in different racks following a Poisson process, with bottleneck traffic load varying from 0.1 to 0.5 for both the oversubscribed and non-oversubscribed settings. We notice that when the bottleneck load is higher than 0.5, packet drops and retransmissions become too frequent to conduct meaningful experiments. At each run, we collect and analyze flow size and completion time information from at least 200,000 flows for each scheme, and each experiment lasts for at least 6 machine hours.

5.2.2 Performance Metrics

flow completion time (FCT) is commonly used as the performance metric in the literature [7], [10], [31]. Yet here we adopt a slightly different metric called *Normalized Flow Completion Time* (NFCT) as our main performance metric for RepFlow and RepSYN. NFCT is defined as the measured FCT minus the estimated *baseline software overhead*, which is the software networking overhead to use a *single* TCP connection. Software networking overhead includes for example code interpretation, transition between user space and kernel space, socket creation, binding, context switching, etc., and varies depending on the OS and the networking stack. It is also possible to almost completely avoid this overhead using a low-level, compiled language and various kernel bypassing techniques [33], [38], [54]. By removing its impact, *NFCT is in fact a better metric to reflect the actual in-network latency*.

The estimated baseline software overhead on our testbed with our `node` implementation is 6.82 ms. It is measured by averaging the FCTs of 100K flows of 1KB sent to localhost, using our implementation without network latency. More discussion on overhead is deferred to Sec. 5.2.6. Note that although RepFlow and RepSYN incur more software networking overhead than TCP (because of the use of an extra TCP socket), this slight difference is already included in their NFCT statistics by definition.

We compare RepFlow and RepSYN against standard Linux TCP Cubic. Since both RepFlow and RepSYN are completely working in the application layer, whose functionality is orthogonal to lower layer schemes, we do not compare against these schemes.

5.2.3 NFCT of Mice Flows

First, we study the NFCT of mice flows. We compare three statistics, median, 99%ile and 99.9%ile NFCT, to show RepFlow and RepSYN's impact on both the median and tail latency.

Fig. 11 shows the results without oversubscription in the network. Neither RepFlow nor RepSYN makes much difference when the load is low ($\leq$ 0.2). As the load increases, RepFlow and RepSYN yield greater benefits in both median and tail latency. When the load is 0.5, RepFlow provides 15.3%, 33.0% and even 69.9% reduction in median, 99%ile, and 99.9%ile NFCT, respectively. RepSYN also achieves 10.0%, 15.8% and 57.8% reduction in median, 99%ile, and 99.9%ile NFCT, compared with TCP.

An interesting observation is that when the load is high, RepFlow achieves much lower tail latency, while RepSYN becomes less beneficial. This is because RepFlow with duplicated transmissions has a lower probability of experiencing packet losses which constitutes a great deal in tail latency.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2793250, IEEE Transactions on Services Computing

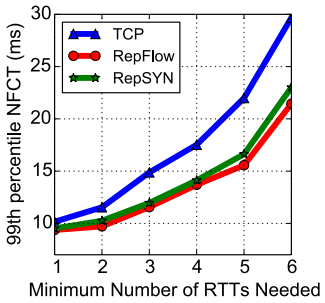


Fig. 13: 99%ile NFCT comparison of flows with different sizes.

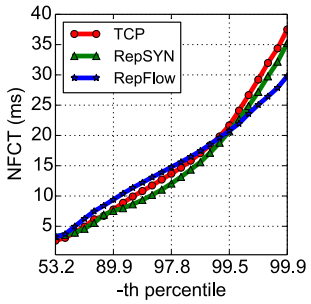


Fig. 14: NFCT of mice flows in incast. Average bottleneck load is 0.2.

When the network is oversubscribed at 2:1, the results are similar in general as shown in Fig. 12. RepFlow and RepSYN are in fact more beneficial in this case, because bursty traffic is more likely to appear at the second or third hop now, which can be avoided by choosing another available path. Therefore, in a production data center network where the topology is typically oversubscribed with many paths available, RepFlow and RepSYN are able to greatly reduce the tail latency and provide better performance.

We also study the impact of flow size on performance improvement. We divide all mice flows into 6 groups based on the minimum number of round trips needed to transmit by TCP. Fig. 13 illustrates the 99%ile NFCT of these groups, when the load is 0.4. We can see that RepFlow and RepSYN are equally beneficial for mice flows of different sizes. We observe the same result for different loads and oversubscription settings and omit the figures here.

5.2.4 Performance under Incast

We carefully study RepFlow and RepSYN's performance in incast scenrios here. In this experiment, whenever we generate a mice flow, we create another 10 flows of the same size with the same destination in parallel, resulting in a 11-to-1 incast pattern. For RepFlow it becomes 22-to-1 incast. Note the flow size distribution still follows the web search workload with both mice and elephants.

The performance is illustrated in Fig. 14. Note that the x-axis is in log scale, which shows more details about the tail latency. Though RepFlow is still able to cut the 99.9%ile NFCT by 20.5%, it is no longer beneficial in the 99%ile, which is $\sim$400μs longer than TCP. Most flows experience longer delay using RepFlow. The benefit in the 99.9%ile is because hash collision with elephants still contributes to the worst-case FCTs in our testbed. However, the benefit may be smaller if the concurrency of small flows was extremely high in incast. In those cases RepFlow could become a real burden.

Fig. 14 shows that RepSYN, on the other hand, has 8.7% and 6.0% NFCT reductions in the 99%ile and 99.9%ile, respectively. The slowest half of all flows are accelerated. Therefore, our suggestion for applications which incorporate serious many-to-one traffic patterns is to use RepSYN instead. Without aggravating the last hop congestion, RepSYN is still beneficial for reducing in-network latency.

5.2.5 Impact on Large Flows

We now assess the impact of RepFlow on elephant flows due to the additional traffic it introduces. We plot throughput of elephants in both low and high loads in Fig. 15a and Fig. 15b, respectively. It is clear that throughput is not affected by RepFlow or RepSYN. The reason is simple: for data centers mice flows only account for a fraction of the total traffic [8], [22], and replicating them thus cause little impact on elephants.

5.2.6 Overhead of Replication

We look at the additional software networking overhead of RepFlow and RepSYN due to the extra TCP connections and state management as mentioned in Sec. 4. We use the same method of obtaining the baseline software overhead — measuring the average FCT of 100K flows of 1KB sent to localhost — for RepFlow and RepSYN. These 100K flows are sent sequentially, making them independent of each other. The result is shown in Fig. 16 with error bars representing one standard deviation. Observe that on average, RepFlow incurs an extra 0.49ms of software overhead, while RepSYN's overhead is 0.32ms in our current implementation. Note that the software networking overhead differs with different system settings.

Another source of overhead is the extra data sent into the network. This is the reason why we see a relatively longer median NFCT in RepFlow and RepSYN when the traffic load is low (*e.g.*, 0.1). This overhead does not directly impact application performance because it is the tail latency, rather than the average, of mice flows that critically affects the performance of applications with a partition-aggregation workflow in datacenters [19].

5.2.7 Discussion

Finally, we comment that the testbed scale is small with limited multipath diversity. Both the kernel configuration and our implementation can be further optimized. Thus the results obtained shall be viewed as a conservative estimate of RepFlow and RepSYN's practical benefits in a production scale network with a large number of paths.

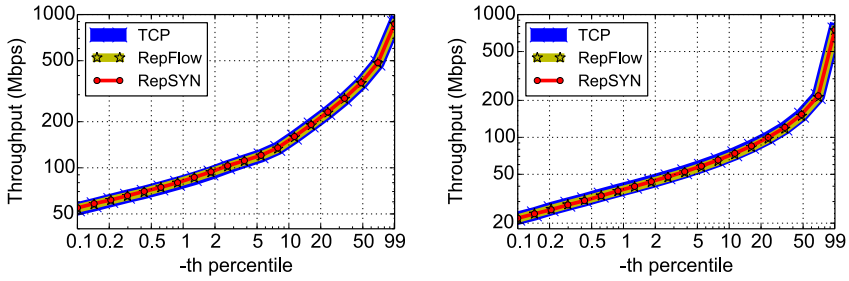
5.3 Application-Level Performance

After the flow-level performance evaluation, one question remains unclear: *how much performance enhancement can we get by using RepNet for applications*? We answer this question by implementing a distributed bucket sort application in node on our testbed, and evaluating the job completion times with different transport mechanisms.

5.3.1 A Sorting Application

We choose to implement bucket sort [17], a classical distributed sorting algorithm, as an example application with a partition-aggregation workflow. In bucket sort, there exists a master which coordinates the sorting process and several slave nodes which complete the sub-processes individually. Whenever the master has a large array of numbers (usually stored in a large text file) to sort, it divides the range of these values into a given number of non-overlapping groups, i.e. buckets. The master then scans the array, disseminates the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2793250, IEEE Transactions on Services Computing



(a) Low bottleneck traffic load of 0.2. (b) High bottleneck traffic load of 0.4.

Fig. 15: Throughput distribution of large flows.

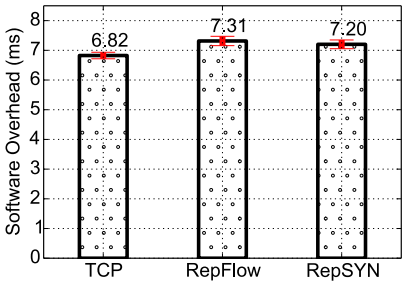


Fig. 16: Estimated software networking overhead comparison.

elements to their corresponding buckets using TCP connections. Typically, each slave node holds a bucket, taking care of unsorted numbers in this bucket. In this case, the slaves are doing counting sort as the unsorted data arrive sequentially. A slave returns the sorted bucket to the master, who simply concatenates the results from all slaves together as a sorted array.

In our experiment, the unsorted array comprises one million integers, which are randomly distributed between 0 and 65535. We have all 12 hosts in our testbed working at the same time, with 1 master and 11 slaves for an individual sorting job.

All network flows are originally generated through the socket API provided by the `Net` module. In order to test RepFlow and RepSYN provided by our RepNet module, all we need to do is to change the module `require` statements at the very beginning of the `node` script.

Mice Flows. The unsorted data distribution process from the master involves a large number of mice flows sending out to multiple destination slaves, because the unsorted numbers are scanned sequentially by the master. A buffering mechanism is used to reduce the flow fragmentation — a chunk of unsorted numbers will not be sent out until a set of 20 numbers to the same destination slave is buffered. With buffering, these flows are still small in size (<1 KB).

Elephant Flows. When a slave completes its share of work, it returns the sorted results in the form of a large file to the master. We take these flows as elephants which will not be replicated by RepNet.

Performance Metrics. In our experiment, each server is working as both master and slave at the same time. As a master node, it continuously generates new random unsorted data sets after the completion of the last job it coordinates. At the same time, it is working as a slave node for each one of the other 11 servers. In this case, the network traffic is a mixture of mice and elephant flows, whose communication pattern is similar to that of a production cluster. Note that the starting time of each sorting master is delayed for several milliseconds randomly, in order to reduce flow concurrency at the beginning of our experiment.

We examine the CDF of the job completion times with different transport mechanisms, i.e. stack TCP, RepFlow and RepSYN. The timing of the job starts when the sorting master begins, i.e. starts reading the unsorted array from the input file, and stops as soon as the sorted array are successfully written to a text file.

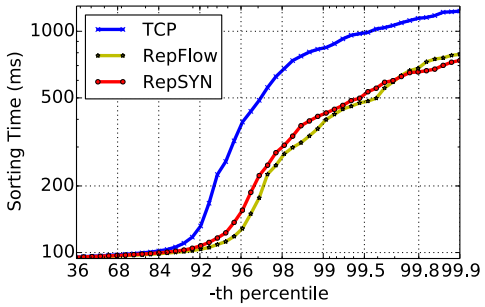


Fig. 17: Job completion time CDF of the bucket sort application.

5.3.2 Job Completion Time Analysis

We run the bucket sort application over 1,000 times on each machine with each transport mechanism, respectively. As a result, over 12,000 job completion times of similar sorting tasks are collected. The CDFs are plotted in Fig. 17. Note that the y-axis is in log scale to emphasize the tail distribution.

The job completion time (JCT) of bucket sort is determined by the last slave node that completes its work. The long FCT of even just a single flow greatly degrade the application-level performance. Therefore, the impact of the "long tail" of FCTs is magnified. We observe that most jobs (∼85%) can finish between 95 to 100 ms, since the paths are idle most of the time. However, due to flash congestions in the network, some jobs experience extremely long delay. With stack TCP, the 99.9%ile job completion time can be as long as 1.2s, which is over 11x more than a job without congestion. RepNet improves JCT here: both RepFlow and RepSYN reduce the 99.9%ile JCT by ∼45%, to 700–800 ms, and the 99%ile JCT by ∼50%.

Comparing RepFlow and RepSYN, their distributions are similar with small differences. RepFlow turns out to be marginally better in most (99.7%) jobs, but has a longer tail (nearly 100 ms) at the 99.9%ile. The reason is that gathering sorting results may result in an incast pattern, with multiple-to-one elephant flow transmission. In most cases, these flows are not concurrent — slaves typically do not finish at the same time, and RepFlow works fine. However when elephant flows happen to have a high concurrency and incast happens, RepSYN is able to better survive the extreme cases.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2793250, IEEE Transactions on Services Computing

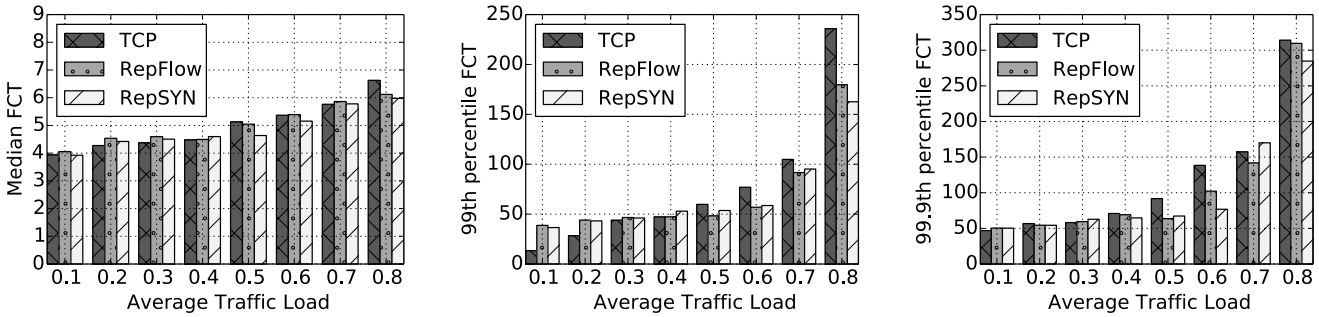


Fig. 18: FCT comparison in Mininet with a fat-tree.

5.4 Mininet Emulation

To verify the performance of RepNet in a larger scale network with higher path diversity, we conduct experiments using Mininet [25]. Mininet is a high fidelity network emulator for software-defined networks on a single Linux machine. All the scripts used for evaluation here is available online [2].

5.4.1 Mininet Configuration

To guarantee high fidelity of the emulation results, we use a Mininet 2.2.0 virtual machine (official VM distributed with Ubuntu 14.04 LTS 64-bit) running on an Amazon EC2 `c3.4xlarge` instance, which has 16 vCPUs and 30GB memory.

We create a 6-pod fat-tree without oversubscription. This is a 3-tier topology with 6 core switches, 18 aggregation switches, and 18 ToR switches. Each rack holds 3 hosts. As a result, it supports 54 hosts with 6 equal cost paths between hosts for different pods. Note that all links in the topology are set to 50Mbps because of the limited switching ability on a single machine. The buffer size at each switch output port is configured to 100 packets. To enable ECMP, an open-source POX controller module[6] is used. The controller implements the ECMP five-tuple hash routing as in RFC 2992. Flows are generated in exactly the same way in our testbed experiments using the empirical web search workload.

5.4.2 Emulation Results

We plot the average, 99%ile and 99.9%ile FCT under various traffic loads in Fig. 18.

Salient Benefit at Tail or High Load. Not surprisingly, both RepFlow and RepSYN show benefits for tail latency or under high load (≥ 0.4), and the figures show similar trends to Fig. 11 and Fig. 12. However, one significant difference is that RepSYN is able to approximate or even outperform RepFlow. The reason is that with more paths available, congestion level on a single path is less fluctuating. Therefore, RTTs of the SYN packets can accurately estimate the congestion level throughout the transmission process of a single mice flow, with less overhead of redundant bytes.

Low Traffic Load (≤ 0.4). However, under low loads, we cannot see much benefit from using RepNet. In some cases, they are even worse than the stack TCP. This is due to the controller overhead in Mininet which we explain now.

6. https://bitbucket.org/msharif/hedera/src

5.4.3 Discussion

Since Mininet is originally designed to emulate a software-defined network, traffic is tightly controlled by a centralized controller, i.e. a POX controller process. This makes Mininet an imperfect tool for traditional network emulation.

When a flow starts in Mininet, its SYN is identified as an unknown packet by the first switch it encounters, and is forwarded to the controller immediately. Then the controller runs ECMP for this packet, and installs new forwarding rules on all switches along the corresponding path. This process usually takes $\sim$1 ms (as the ping result suggests) independent of the network state. With a large number of flows starting around the same time the controller is easily congested. Flow replication aggravates the controller overload. This results in the distortion of flow latency, which does not exist in real data center networks. Nevertheless, in most cases, we can still benefit from using RepNet despite the controller overhead.

6 Related Work

Motivated by the drawbacks of TCP, many new data center transport designs have been proposed. We briefly review the most relevant prior work here. We also introduce some additional work that uses replication in wide-area Internet, MapReduce, and distributed storage systems for latency gains.

Data center transport. DCTCP [8] and HULL [9] use ECN-based adaptive congestion control and appropriate throttling of long flows to keep the switch queue occupancy low in order to reduce short flows' FCT. D^3 [50], D2TCP [46], and PDQ [28] use explicit deadline information to drive the rate allocation, congestion control, and preemptive scheduling decisions. DeTail [53] and pFabric [10] present clean-slate designs of the entire network fabric that prioritize latency sensitive short flows to reduce the tail FCT. FUSO [16] attempts to strip a flow and send it via multiple paths, and utilize the lightly-loaded paths to expedite loss recovery on the heavily-loaded paths. All of these proposals require modifications to switches and operating systems. Our design objective is different: we strive for a simple way to reduce FCT without any change to TCP and switches, and can be readily implemented at layers above the transport layer. RepNet presents such a design with simple flow replication that works with any existing transport protocol.

Replication for latency. Though seemingly naive, the general idea of using replication to improve latency has

gained increasing attention in both academia and industry for its simplicity and effectiveness. [5] proposes to employ duplicated SYN packets for path selection in a multi-path system, which is similar to RepSYN. The proposed implementation is in the transport layer. It requires both the SYNs and the SYN-ACKs to include unique identifiers, such that the inbound and outbound paths can be selected separately. RepSYN is implemented in the application layer on top of legacy TCP. Google reportedly uses request replications to rein in the tail response times in their distributed systems [18]. Vulimiri et al. [48] argue for the use of redundant operations as a general method to improve latency in various systems, such as DNS, databases, and networks. RANS [30] makes the network stack be aware of duplicated application-layer requests, such that flows can be well scheduled for shorter latencies. RepNet, if implemented on top of its APIs, can benefit from such scheduling technique.

7 Concluding Remarks

We presented the design, analysis, implementation, and evaluation of `RepNet`, a low-latency application layer transport module based on `node` which provides socket APIs to enable flow replication. Experimental evaluation on a real testbed and in Mininet demonstrates its effectiveness on both mean and tail latency for mice flows. We also proposed RepSYN to alleviate its performance degradation in incast scenarios.

References

[1] `node.js` official website. https://nodejs.org

[2] RepNet experiment code. https://bitbucket.org/shuhaoliu/repnet_experiment

[3] RepNet source code. https://bitbucket.org/shuhaoliu/repnet

[4] Node at LinkedIn: the pursuit of thinner, lighter, faster. *ACM Queue*, 11(12):40:40–40:48, December 2013.

[5] K. Agarwal. Path Selection Using TCP Handshake in a Multipath Environment. US Patent 14 164 422, July 30, 2015.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. ACM SIGCOMM*, 2008.

[7] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proc. ACM SIGCOMM*, 2014.

[8] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. ACM SIGCOMM*, 2010.

[9] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proc. USENIX NSDI*, 2012.

[10] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. M. B. Prabhakar, and S. Shenker. pFabric: Minimal near-optimal datacenter transport. In *Proc. ACM SIGCOMM*, 2013.

[11] A. Andreyev. Introducing data center fabric, the next-generation Facebook data center network. https://code.facebook.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/. November 2014.

[12] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. In *Proc. ACM SIGCOMM*, 2004.

[13] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and W. Sun. PIAS: Practical information-agnostic flow scheduling for data center networks. In *Proc. ACM HotNets*, 2014.

[14] O. Boxma and B. Zwart. Tails in scheduling. *SIGMETRICS Perform. Eval. Rev.*, 34(4):13–20, March 2007.

[15] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP latency. In *Proc. IEEE INFOCOM*, 2000.

[16] G. Chen, Y. Lu, Y. Meng, B. Li, K. Tan, D. Pei, P. Cheng, L. L. Luo, Y. Xiong, X. Wang, and Y. Zhao. Fast and cautious: Leveraging multi-path diversity for transport loss recovery in data centers. In *Proc. USENIX Annual Technical Conference (ATC)*, 2016.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[18] J. Dean. Achieving rapid response times in large online services. Berkeley AMPLab Cloud Seminar, http://research.google.com/people/jeff/latency.html, March 2012.

[19] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[20] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An argument for increasing TCP's initial congestion window. *ACM SIGCOMM Comput. Commun. Rev.*, 40(3):26–33, June 2010.

[21] S. Foss. *Wiley Encyclopedia of Operations Research and Management Science*, chapter The G/G/1 Queue. 2011.

[22] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. ACM SIGCOMM*, 2009.

[23] D. Gross, J. F. Shortle, J. M. Thompson, and C. M. Harris. *Fundamentals of Queueing Theory*. Wiley-Interscience, 2008.

[24] H. Han, S. Shakkottai, C. V. Hollot, R. Srikant, and D. Towsley. Multi-path tcp: A joint congestion control and routing scheme to exploit path diversity in the Internet. *IEEE/ACM Trans. Netw.*, 14(6):1260–1271, December 2006.

[25] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proc. ACM CoNEXT*, 2012.

[26] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *Proc. ACM SIGCOMM*, 2015.

[27] J. Heidemann, K. Obraczka, and J. Touch. Modeling the performance of HTTP over several transport protocols. *IEEE/ACM Trans. Netw.*, 5(5):616–630, October 1997.

[28] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *Proc. ACM SIGCOMM*, 2012.

[29] C. Hopps. Analysis of an Equal-Cost Multi-Path algorithm. http://tools.ietf.org/html/rfc2992, November 2000.

[30] A. M. Iftikhar, F. Dogar, and I. A. Qazi. Towards a redundancy-aware network stack for data centers. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, 2016.

[31] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg. Eyeq: Practical network performance isolation at the edge. In *Proc. USENIX NSDI*, 2013.

[32] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of datacenter traffic: Measurements & analysis. In *Proc. IMC*, 2009.

[33] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *Proc. ACM SoCC*, 2012.

[34] F. P. Kelly. Notes on effective bandwidths. In *Stochastic networks: Theory and applications*, pages 141–168. Oxford University Press, 1996.

[35] A. Lakshmikantha, C. Beck, and R. Srikant. Impact of file arrivals and departures on buffer sizing in core routers. *IEEE/ACM Trans. Netw.*, 19(2):347–358, April 2011.

[36] C. Lee, C. Park, K. Jang, S. Moon, and D. Han. Accurate Latency-based Congestion Feedback for Datacenters. In *Proc. USENIX ATC*, 2015.

[37] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM Comput. Commun. Rev.*, 27(3):67–82, July 1997.

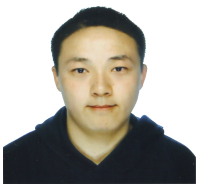
[38] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proc. ACM SIGCOMM*, 2015.

[39] A. Munir, I. A. Qazi, Z. A. Uzmi, A. Mushtaq, S. N. Ismail, M. S. Iqbal, and B. Khan. Minimizing flow completion times in data centers. In *Proc. IEEE INFOCOM*, 2013.

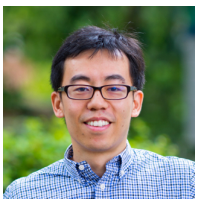
[40] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *Proc. ACM SIGCOMM*, 1998.

[41] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *Proc. USENIX FAST*, 2008.

[42] R. S. Prasad and C. Dovrolis. Beyond the model of persistent TCP flows: Open-loop vs closed-loop arrivals of non-persistent flows. In *Proc. IEEE ANSS*, 2008.
[43] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proc. ACM SIGCOMM*, 2015.
[44] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 2007.
[45] S. Tilkov and S. Vinoski. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6), 2010.
[46] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware datacenter TCP (D2TCP). In *Proc. ACM SIGCOMM*, 2012.
[47] 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 datacenter communication. In *Proc. ACM SIGCOMM*, 2009.
[48] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. In *Proc. ACM CoNEXT*, 2013.
[49] P. Wang, H. Xu, Z. Niu, D. Han, and Y. Xiong. Expeditus: Distributed Congestion-Aware Load Balancing in Clos Data Center Networks. In *Proc. ACM SoCC*, 2016.
[50] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better never than late: Meeting deadlines in datacenter networks. In *Proc. ACM SIGCOMM*, 2011.
[51] H. Xu and B. Li. RepFlow: Minimizing flow completion times with replicated flows in data centers. In *Proc. IEEE INFOCOM*, 2014.
[52] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Proc. USENIX NSDI*, 2013.
[53] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the flow completion time tail in datacenter networks. In *Proc. ACM SIGCOMM*, 2012.
[54] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale rdma deployments. In *Proc. ACM SIGCOMM*, 2015.



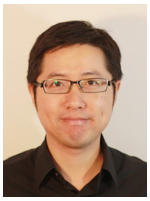
Shuhao Liu is currently a Ph.D. student in the Department of Electrical and Computer Engineering, University of Toronto. He received his B.Eng. degree from Tsinghua University in 2012. His current research interests include software-defined networking and big data analytics.



Hong Xu received the B.Eng. degree from the Department of Information Engineering, The Chinese University of Hong Kong, in 2007, and the M.A.Sc. and Ph.D. degrees from the Department of Electrical and Computer Engineering, University of Toronto. He joined the Department of Computer Science, City University of Hong Kong in 2013, where he is currently an assistant professor. His research interests include data center networking, SDN, NFV, and cloud computing. He was the recipient of an Early Career Scheme Grant from the Research Grants Council of the Hong Kong SAR, 2014. He also received the best paper awards from IEEE ICNP 2015 and ACM CoNEXT Student Workshop 2014. He is a member of ACM and IEEE.

Wei Bai received the B.E. degree in information security from Shanghai Jiao Tong University, China in 2013 and the Ph.D. degree in computer science in Hong Kong University of Science and Technology in 2017. He is now a researcher in Microsoft Research Asia, Beijing, China. His current research interests are in the area of data center networks.

Libin Liu is currently a PhD student in the Department of Computer Science, City University of Hong Kong. He received his B.E. degree in software engineering from Shandong University, in 2015. His current research interests include network function virtualization and data center networks.



Kai Chen is an Associate Professor with Department of Computer Science and Engineering, Hong Kong University of Science and Technology. He received his PhD in Computer Science from Northwestern University, Evanston IL in 2012. His research interests include networked systems design and implementation, data center networks, data centric networking, and cloud and big data systems. He is interested in finding simple yet effective solutions to real-world networking systems problems.

Zhiping Cai received his B.S., M.S. and Ph.D. degrees in computer science and technology with honor all from National University of Defense Technology (NUDT), Changsha, Hunan, in July 1996, April 2002 and December 2005, respectively. He is currently a Professor in Department of Networking Engineering, College of Computer, NUDT, Changsha, Hunan, China. His doctoral dissertation won the Outstanding Dissertation Award of the China PLA. His research interests include network security, network measurement and network virtualization. He is a member of ACM and IEEE.