RepFlow on node.js: Cutting Tail Latency in Data Center Networks at the Applications Layer

Shuhao Liu*, Wei Bai†, Hong Xu*, Kai Chen†, Zhiping Cai‡
*Department of Computer Science, City University of Hong Kong
†Department of Computer Science and Engineering, HKUST
‡College of Computer, National University of Defense Technology, China

ABSTRACT

Low latency, especially at the tail, is increasingly demanded by interactive applications in data center networks. To improve tail latency, existing approaches require modifications to switch hardware and/or end-host stacks, making them difficult to be deployed. We present the design, implementation, and evaluation of RepFlow, an application layer transport based on node.js that can be deployed today. RepFlow replicates mice flows to cut tail latency in Clos topologies. node.js’s single threaded event-loop and non-blocking I/O makes flow replication highly efficient. We further implement RepSYN to alleviate RepFlow’s negative impact in incast scenarios by only using the first connection that finishes TCP handshaking. Performance evaluation on a leaf-spine network testbed reveals that RepFlow is able to reduce the tail latency of small flows by more than 50%. Also, RepSYN offers similar benefits in scenarios with and without incast.

1. INTRODUCTION

Data center networks are increasingly tasked to provide low latency for many interactive applications they support [6,8,26]. Low tail latency (e.g. 99th or 99.9th percentile) is especially important for these applications, since a request’s completion depends on all (or most) of the responses from many worker machines [11]. 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 10 times worse than the average in production networks, even when the network is lightly loaded [8,25,26].

The main reason for long tail latency is the co-existence of elephant and mice flows in data center networks. While most flows are mice with less than say 100 KB, most bytes are in fact from a few elephants [6,12,14]. Thus mice flows are often queued behind bursts of packets from elephants in switches, resulting in long queuing delay and flow completion time (FCT). The problem has attracted much attention recently in our community [16]. Loosely speaking, existing work reduces the tail latency by (1) reducing the queue length, such as DCTCP [6] and HULL [7], (2) prioritizing mice flows, such as D³ [23], PDQ [13], pFabric [8], and (3) engineering better multi-path schemes, such as DeTail [26] and DRB [9]. While effective, they require changes to switches and/or end-hosts, and face significant deployment challenges. Thus there is a growing need for an application layer solution that can provide immediate latency gains without an overhaul of the infrastructure.

To this end, we introduce RepFlow, a low latency transport that can be readily deployed in current infrastructures. RepFlow is based on the simple idea of flow replication proposed in our previous work [24]. In RepFlow, a mice flow is replicated by creating another TCP connection to capture the multipath diversity in Clos based topologies [6,12]. The intuition is that, as flash congestion happens randomly in the network due to bursty traffic and imperfect hash based load balancing, it is unlikely that both the original and replicated flows experience congestion and long queuing delay. The benefit of flow replication was evaluated using simple queueing analyses and NS-3 simulations in [24].

In this paper, we take one step further, and implement RepFlow on node.js [2] as an application layer 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 without much performance overhead compared to Tornado and other frameworks. Moreover, node is widely used for developing the back-end of large-scale interactive applications in production systems at LinkedIn, Yahoo!, Microsoft, etc. RepFlow on node potentially provides immediate latency benefit for a large number of these applications with minimal code change.

An astute reader might be concerned about using RepFlow in incast scenarios, where many senders transmit at the same time to a common destination [21], causing throughput collapse. To alleviate this problem, we design and implement RepSYN, which uses SYN packets to choose a less congested path. Instead of replicating a complete flow, RepSYN only uses the first connection that finishes handshaking for data transmission.

We evaluate our RepFlow and RepSYN on a 12-server leaf-spine testbed, using an empirical flow size distribution

URLs:
http://www.tornadoweb.org/
http://nodejs.org/industry/
from a production network. Our evaluation shows that, though with some overhead, RepFlow is still able to significantly reduce the tail latency of mice flows, especially when the traffic load is high enough to cause frequent packet losses. As an alternative, RepSYN is less effective compared with RepFlow in most cases, but it remains beneficial in incast scenarios where RepFlow suffers from performance degradation. The implementation code, with scripts used for performance evaluation, is available online [3]. We are in the process of making it available as a NPM (Node Package Manager) module for the node user community.

2. WHY node?

Let us start by introducing node and the main reasons why we choose to implement RepFlow on it. On a high level, node is a highly scalable platform for real-time server-side networking applications. It combines single-threaded, non-blocking socket with the even-driven philosophy of JavaScript. It runs on Google V8 engine with core libraries optimized for performance and scalability. For more details see [2].

2.1 Efficiency

The first reason for choosing node is efficiency. RepFlow introduces the overhead of launching additional TCP connections. To provide maximal latency improvements, we need to minimize this overhead. This quickly rules out a multi-threaded implementation using for example Tornado or Thrift [18]. 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 [19] with poor performance and scalability. In fact, we tried to implement RepFlow on Thrift based on python, and found that the performance is simply 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.

2.2 Immediate Impact to a Broad Community

The second reason we choose node is that it is widely deployed in production systems for companies such as LinkedIn, Microsoft, etc. [2]. node supports a full JavaScript architecture and integrates well with NoSQL data stores at backend. Thus implementing RepFlow on it is likely to benefit a bigger audience and generate more impact to the industry. Further, node is largely used for data-intensive real-time applications such as online chatting and data streaming, which calls for low-latency transports such as RepFlow.

3. IMPLEMENTATION

We now describe our implementation of RepFlow as a new module in node, called RepNet. RepNet is based upon the Net module, which is 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 RepFlow.

3.1 Objectives

RepNet is designed 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.

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

3.2 Overview

RepNet consists of two classes: RepNet.Socket and RepNet.Server. RepNet.Socket implements a RepFlow 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 RepFlow and regular TCP connections. Note that RepNet.Server does not have any application logic. Instead, it creates a connection listener at the server side, which responds to the SYN packet 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. 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

http://nodejs.org/api/net.html
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 right away.

The server now waits for the replicated 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 code is based on node version 0.11.13. All code is available online [3]. We introduce more details of our implementation in the following.

3.3 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 1. Figure 1 shows the possible state transitions with more explanation in Table 2. 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.

3.4 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 for RepFlow 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.

3.5 RepSYN to Alleviate Incast

RepFlow takes advantage of congestion diversity of different paths to cut tail latency. However, if the traffic creates incast with multiple flows sending concurrently to the same destination host [21], replication will significantly aggravate the throughput collapse.

We propose and implement RepSYN, an extension in RepNet to alleviate RepFlow’s limitation in incast. The idea is simple: Before transmitting data, 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.

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 2. RepSYN can be activated by setting the Flag.RepSYN flag of the RepNet.Socket object.

4. EXPERIMENTAL EVALUATION

We present our testbed evaluation of both RepFlow and RepSYN in this section.

4.1 Testbed Setup
The slower connection is detected at the server. The archived data for writes exceeds the threshold. Both member Net.Socket objects are open. One of Net.Socket objects is no longer valid. The RepNet.Socket is ended.

| 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 1: 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 2: Trigger of the state transitions

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

The server OS is Debian 6.0 64-bit Linux, kernel version 2.6.38.3. We change RTO_{min} to 10 ms in order to remedy the negative impact of incast and packet retransmission [21]. 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 4.5KB of 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 as depicted in Fig.2 which is widely used in production data centers [8]. There are 12 servers organized in 2 racks, and 3 spine switches which provide up to 3 equal-cost paths between two hosts under different ToRs. The ping RTT is \( \sim 178 \mu s \) across racks. The topology is oversubscribed at 2:1 when all hosts are used. We also conduct experiments without oversubscription, by simply shutting down half of the servers in each rack.

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

The server OS is Debian 6.0 64-bit Linux, kernel version 2.6.38.3. We change RTO_{min} to 10 ms in order to remedy the negative impact of incast and packet retransmission [21]. 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 4.5KB of 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 as depicted in Fig.2 which is widely used in production data centers [8]. There are 12 servers organized in 2 racks, and 3 spine switches which provide up to 3 equal-cost paths between two hosts under different ToRs. The ping RTT is \( \sim 178 \mu s \) across racks. The topology is oversubscribed at 2:1 when all hosts are used. We also conduct experiments without oversubscription, by simply shutting down half of the servers in each rack.

**Empirical Flow Size.** We use the flow size distribution from a web search workload [6] 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.

**Performance Metrics.** We compare RepFlow and RepSYN against standard linux TCP cubic. We use Normalized Flow Completion Time (NFCT), defined as the measured FCT minus the kernel networking overhead for TCP as the performance metric for all the three schemes. Kernel overhead includes for example 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 kernel bypass and other techniques [15]. Thus we remove its impact in NFCT. Note that RepFlow and RepSYN incur more kernel overhead than TCP, which is included in their NFCT statistics by definition. We measure the kernel overhead of TCP as the average FCT of 100K flows of 1KB sent to localhost using our implementation without network latency, which is 6.82ms. More discussion on overhead is deferred to Sec. 4.5.

### 4.2 NFCT of Mice Flows

First, we study the NFCT of mice flows. We compare three statistics, the average, the 99th percentile and the 99.9th percentile NFCTs, to show RepFlow and RepSYN’s impact on both the average and tail latency.

Fig. 3 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 average and tail latency. When the load is 0.5, RepFlow provides 15.3%, 33.0% and even 69.9% reduction in average, 99th percentile, and 99.9th percentile NFCT, respectively. Rep-
SYN also achieves 10.0%, 15.8% and 57.8% reduction in average, 99th percentile, and 99.9th percentile NFCT, compared with TCP.

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

When the network is oversubscribed at 2:1, the results are similar in general as shown in Fig. 4. 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 RTT rounds needed to transmit by TCP. Fig. 5 illustrates the 99th percentile NFCT of these groups, when the load is 0.4. We can clearly 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.

### 4.3 Incast

We carefully study RepFlow and RepSYN’s performance in incast scenarios 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 flows the web search workload with both mice and elephants.

The performance is illustrated in Fig. 6. 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 99.9th percentile NFCT by 20.5%, it is no longer beneficial in the 99th percentile, which is ~400μs longer than TCP. Most flows experience longer delay using RepFlow. The benefit in the 99.9th percentile 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. 6 shows that RepSYN, on the other hand, has 8.7% and 6.0% NFCT reductions in the 99th and 99.9th percentile, 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.

4.4 Impact on Large Flows

Another possible concern is that RepFlow may degrade throughput for elephant flows due to the additional traffic it introduces. We plot throughput of elephants in both low and high loads in Fig. 7a and Fig. 7b, 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 [6,12], and replicating them thus cause little impact on elephants.

4.5 Overhead of Replication

We look at the additional kernel overhead of RepFlow and RepSYN due to the extra TCP connections and state management as in Sec. 3. We use the same method of obtaining kernel overhead of TCP — measuring the FCT of 100K flows of 1KB sent to localhost — for RepFlow and RepSYN. The result is shown in Fig. 8 with error bars representing one standard deviation. Observe that on average, RepFlow incurs an extra 0.49ms of overhead, while RepSYN’s overhead is only 0.32ms in our current implementation. Compared with tail NFCT which is more than 20ms, this overhead is negligible. Optimization such as kernel bypass [1,15] can further reduce this overhead though it is beyond the scope.

4.6 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. RELATED WORK

Low latency data center networking has been an active research area over the recent years. DCTCP [6] and HULL [7] use ECN-based congestion control to keep the switch queue occupancy low. D3 [23], D2TCP [20], and PDQ [13] use explicit deadline information for rate allocation, congestion control, and preemptive scheduling. DeTail [26] and pFabric [8] present clean-slate designs of the entire network fabric that prioritize latency sensitive short flows. All of them require modifications to switches and end-hosts. Note priority queueing is widely supported in commodity switches and in principle can be used to expedite mice flows. However this will interact negatively with its existing use in traffic differentiation based on applications and purposes [4] which is fairly common in production networks.

We also comment that the general idea of using replication to improve latency has gained increasing attention. Mitzenmacher’s seminal work on “power of two choices” [17] proposes for a request to randomly sample two servers, and queue at the one with less queueing to achieve good load balancing. Google reportedly uses request replication to reduce in the tail response times [10]. Vulimiri et al. [22] argue for the use of redundant operations to improve latency in various systems. To our best knowledge, our work is among the first to provide a readily deployable implementation and testbed evaluation of flow replication in data centers.

6. CONCLUDING REMARKS

We presented the design and implementation of RepNet, a node module which provides socket APIs to enable RepFlow at the application layer. Experimental evaluation on a testbed demonstrates its effectiveness on both mean and tail latency for mice flows. We also proposed RepSYN to alleviate its performance degradation in incast scenarios.

An interesting observation inspired by this work is that efficient multipath routing in data center networks is worth further digging. Because randomly choosing another path indeed makes a difference in RepFlow, a congestion-aware routing algorithm that aims to choose the best path may provide even more latency reduction, though this imposes daunting challenges of collecting global and timely congestion information.

Acknowledgments

We thank Fred Baker from Cisco and Shuang Yang from Google for helpful feedback and suggestions.

1 Packets from control protocols typically have higher priority. Also, traffic of production systems have higher priority than traffic for experimental and development purposes.
7. REFERENCES

[1] https://ramcloud.stanford.edu/wiki/display/ramcloud/Low+latency+RPCs
[2] node.js official website, https://nodejs.org
[3] RepNet source code repository. https://bitbucket.org/liuempire/repnet
[4] Node at LinkedIn: the pursuit of thinner, lighter, faster. ACM Queue, 11(12):40:40–40:48, December 2013.
[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In Proc. ACM SIGCOMM, 2008.
[6] 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.
[7] 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.
[8] 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.
[9] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz. Per-packet load-balanced, low-latency routing for Clos-based data center networks. In Proc. ACM CoNEXT, 2013.
[10] J. Dean. Achieving rapid response times in large online services. Berkeley AMPLab Cloud Seminar, http://research.google.com/people/jeff/latency.html March 2012.
[11] J. Dean and L. A. Barroso. The tail at scale. Commun. ACM, 56(2):74–80, February 2013.
[12] A. Greenberg, J. R. Hamilton, N. Jain, S. Kendula, C. Kim, P. Lahiri, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In Proc. ACM SIGCOMM, 2009.
[13] C.-Y. Hong, M. Caeser, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In Proc. ACM SIGCOMM, 2012.
[14] S. Kendula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of datacenter traffic: Measurements & analysis. In Proc. IMC, 2009.
[15] 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.
[16] S. Liu, H. Xu, and Z. Cai. Low latency datacenter networking: A short survey. http://arxiv.org/abs/1312.3455 2013.
[17] M. D. Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. PhD thesis, UC Berkeley, 1996.
[18] M. Sles, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 2007.