This paper was presented as part of the main technical program at IEEE INFOCOM 2011

Mahout: Low-Overhead Datacenter Traffic Management using End-Host-Based Elephant Detection

Andrew R. Curtis*
University of Waterloo
Waterloo, Ontario, Canada

Wonho Kim*
Princeton University
Princeton, NJ, USA

Praveen Yalagandula
HP Labs
Palo Alto, CA, USA

***Abstract*—Datacenters need high-bandwidth interconnection fabrics. Several researchers have proposed highly-redundant topologies with multiple paths between pairs of end hosts for datacenter networks. However, traffic management is necessary to effectively utilize the bisection bandwidth provided by these topologies. This requires timely detection of *elephant* flows—flows that carry large amount of data—and managing those flows. Previously proposed approaches incur high monitoring overheads, consume significant switch resources, and/or have long detection times.**

We propose, instead, to detect elephant flows at the end hosts. We do this by observing the end hosts's socket buffers, which provide better, more efficient visibility of flow behavior. We present Mahout, a low-overhead yet effective traffic management system that follows OpenFlow-like central controller approach for network management but augments the design with our novel end host mechanism. Once an elephant flow is detected, an end host signals the network controller using in-band signaling with low overheads. Through analytical evaluation and experiments, we demonstrate the benefits of Mahout over previous solutions.

I. Introduction

Datacenter switching fabrics have enormous bandwidth demands due to the recent uptick in bandwidth-intensive applications used by enterprises to manage their exploding data. These applications transfer huge quantities of data between thousands of servers. For example, Hadoop [18] performs an all-to-all transfer of up to petabytes of files during the shuffle phase of a MapReduce job [15]. Further, to better consolidate employee desktop and other computation needs, enterprises are leveraging virtualized datacenter frameworks (e.g., using VMWare [29] and Xen [10], [30]), where timely migration of virtual machines requires high throughput network.

Designing datacenter networks using redundant topologies such as Fat-tree [6], [12], HyperX [5], and Flattened Butterfly [22] solves the high-bandwidth requirement. However, traffic management is necessary to extract the best bisection bandwidth from such topologies [7]. A key challenge is that the flows come and go too quickly in a data center to compute a route for each individually; e.g., Kandula *et al.* report 100K flow arrivals a second in a 1,500 server cluster [21].

For effective utilization of the datacenter fabric, we need to detect *elephant flows*—flows that transfer significant amount of data—and dynamically orchestrate their paths. Datacenter measurements [17], [21] show that a large fraction of datacenter traffic is carried in a small fraction of flows. The authors report that 90% of the flows carry less than 1MB of data and more than 90% of bytes transferred are in flows greater than 100MB. Hash-based flow forwarding techniques such as Equal-Cost Multi-Path (ECMP) routing [19] works well only for large numbers of small (or *mice*) flows and no elephant flows. For example, Al-Fares *et al.*'s Hedera [7] shows that managing elephant flows effectively can yield as much as 113% higher aggregate throughput compared to ECMP.

Existing elephant flow detection methods have several limitations that make them unsuitable for datacenter networks. These proposals use one of three techniques to identify elephants: (1) periodic polling of statistics from switches, (2) streaming techniques like sampling or window-based algorithms, or (3) application-level modifications (full details of each approach are given in Section II). We have not seen support for Quality of Service (QoS) solutions take hold, which implies that modifying applications is probably an unacceptable solution. We will show that the other two approaches fall short in the datacenter setting due to high monitoring overheads, significant switch resource consumption, and/or long detection times.

We assert that the right place for elephant flow detection is at the end hosts. In this paper, we describe Mahout, a low-overhead yet effective traffic management system using end-host-based elephant detection. We subscribe to the increasingly popular simple-switch/smart-controller model (as in OpenFlow [4]), and so our system is similar to NOX [28] and Hedera [7].

Mahout augments this basic design. It has low overhead, as it monitors and detects elephant flows at the end host via a shim layer in the OS, rather than monitoring at the switches in the network. Mahout does timely management of elephant flows through an in-band signaling mechanism between the shim layer at the end hosts and the network controller. At the switches, any flow not signaled as an elephant is routed using a static load-balancing scheme (e.g., ECMP). Only elephant flows are monitored and managed by the central controller. The combination of end host elephant detection and in-band signaling eliminates the need for per-flow monitoring in the switches, and hence incurs low overhead and requires few switch resources.

We demonstrate the benefits of Mahout using analytical evaluation and simulations and through experiments on a small testbed. We have built a Linux prototype for our end host elephant flow detection algorithm and tested its effectiveness.

*This work was performed while Andrew and Wonho were interns at HP Labs—Palo Alto.

978-1-4244-9921-2/11/$26.00 ©2011 IEEE

We have also built a Mahout controller, for setting up switches with default entries and for processing the tagged packets from the end hosts. Our analytical evaluation shows that Mahout offers one to two orders of magnitude of reduction in the number of flows processed by the controller and in switch resource requirements, compared to Hedera-like approaches. Our simulations show that Mahout can achieve considerable throughput improvements compared to static load balancing techniques while incurring an order of magnitude lower overhead than Hedera. Our prototype experiments show that the Mahout approach can detect elephant flows at least an order of magnitude sooner than statistics-polling based approaches.

The key contributions of our work are: 1) a novel end host based mechanism for detecting elephant flows, 2) design of a centralized datacenter traffic management system that has low overhead yet high effectiveness, and 3) simulation and prototype experiments demonstrating the benefits of the proposed design.

II. Background & Related Work

A. Datacenter networks and traffic

The heterogeneous mix of applications running in datacenters produces flows that are generally sensitive to either latency or throughput. Latency-sensitive flows are usually generated by network protocols (such as ARP and DNS) and interactive applications. They typically transfer up to a few kilobytes. On the other hand, throughput-sensitive flows, created by, e.g., MapReduce, scientific computing, and virtual machine migration, transfer up to gigabytes. This traffic mix implies that a datacenter network needs to deliver high bisection bandwidth for throughput-sensitive flows without introducing setup delay on latency-sensitive flows.

Designing datacenter networks using redundant topologies such as Fat-tree [6], [12], HyperX [5], or Flattened Butterfly [22] solves the high-bandwidth requirement. However, these networks use multiple end-to-end paths to provide this high-bandwidth, so they need to load balance traffic across them. Load balancing can be performed with no overhead using *oblivious routing*, where the path a flow from node i to node j is routed on is randomly selected from a probability distribution over all i to j paths, but it has been shown to achieve less than half the optimal throughput when the traffic mix contains many elephant flows [7]. The other extreme is to perform online scheduling by selecting the path for all new flows using a load balancing algorithm, e.g., greedily adding a flow along the path with least congestion. This approach doesn't scale well—flows arrive too quickly for a single scheduler to keep up—and it adds too much setup time to latency-sensitive flows. For example, flow installation using NOX can take up to 10ms [28]. Partition-aggregate applications (such as search and other web applications) partition work across multiple machines and then aggregate the responses. Jobs have a deadline of 10–100ms [8], so a 10ms flow setup delay can consume the entire time budget. Therefore, online scheduling is not suitable for latency-sensitive flows.

B. Identifying elephant flows

The mix of latency- and throughput-sensitive flows in the data centers means that effective flow scheduling needs to balance visibility and overhead—a one size fits all approach is not sufficient in this setting. To achieve this balance, elephant flows must be identified so that they are the only flows touched by the controller. The following are the previously considered mechanisms for identifying elephants:

- *Applications identify their flows as elephants:* This solution accurately and immediately identifies elephant flows. This is a common assumption for a plethora of research work in network QoS where focus is to give higher priority to latency and throughput-sensitive flows such as voice and video applications (see, e.g., [11]). However, this solution is impractical for traffic management in datacenters as each and every application must be modified to support it. If all applications are not modified, an alternative technique will still be needed to identify elephant flows initiated by unmodified applications.
  A related approach is to classify flows based on which application is initiating them. This classifies flows using stochastic machine learning techniques [27], or using simple matching based on the packet header fields (such as TCP port numbers). While this approach might be suitable for enterprise network management, it is unsuitable for datacenter network management because of the enormous amount of traffic in the datacenter and the difficulty in obtaining flow traces to train the classification algorithms.
- *Maintain per-flow statistics:* In this approach, each flow is monitored at the first switch that the flow goes through. These statistics are pulled from switches by the controller at regular intervals and used to classify elephant flows. Hedera [7] and Helios [16] are examples of systems proposing to use such a mechanism. However, this approach does not scale to large networks. First, this consumes significant switch resources: a flow table entry for each flow monitored at a switch. We'll show in Section IV that this requires considerable number of flow table entries. Second, bandwidth between switches and the controller is limited, so much so that transferring statistics becomes the bottleneck in traffic management in datacenter network. As a result, the flow statistics cannot be quickly transferred to the controller, resulting in prolonged sub-par routings.
- *Sampling:* Instead of monitoring each flow in the network, in this approach, a controller samples packets from all ports of the switches using switch sampling features such as sFlow [3]. Only a small fraction of packets are sampled (typically, 1 in 1000) at the switches and only headers of the packets are transferred to the controller. The controller analyzes the samples and identifies a flow as an elephant after it has seen sufficient number of samples from the flow. However, such an approach can not reliably detect an elephant flow before it has carried more

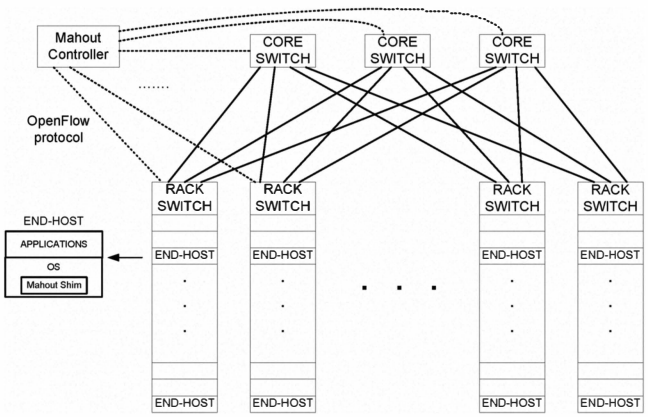


Fig. 1: Mahout architecture.

than 10K packets, or roughly 15MB [25]. Additionally, sampling has high overhead, since the controller must process each sampled packet.

C. OpenFlow

OpenFlow [23] aims to open up traditionally closed designs of commercial switches to enable network innovation. OpenFlow switches maintain a flow-table where each entry contains a pattern to match and the actions to perform on a packet that matches that entry. OpenFlow defines a protocol for communication between a controller and an OpenFlow switch to add and remove entries from the flow table of the switch and to query statistics of the flows.

Upon receiving a packet, if an OpenFlow switch does not have an entry in the flow table or TCAM that matches the packet, the switch encapsulates and forwards the packet to the controller over a secure connection. The controller responds back with a flow table entry and the original packet. The switch then installs the entry into its flow table and forwards the packet according to the actions specified in the entry. The flow table entries expire after a set amount of time, typically 60 seconds. OpenFlow switches maintain statistics for each entry in their flow table. These statistics include a packet counter, byte counter, and duration.

The OpenFlow 1.0 specification [2] defines matching over 12 fields of packet header (see the top line in Figure 3). The specification defines several actions including forwarding on a single physical port, forwarding on multiple ports, forwarding to the controller, drop, queue (to a specified queue), and defaulting to traditional switching. To support such flexibility, current commercial switch implementations of OpenFlow use TCAMs for flow table.

III. Our Solution: Mahout

Mahout's architecture is shown in Figure 1. In Mahout, a shim layer on each end host monitors the flows originating from that host. When this layer detects an elephant flow, it marks subsequent packets of that flow using an in-band signaling mechanism. The switches in the network are configured to forward these marked packets to the Mahout controller. This

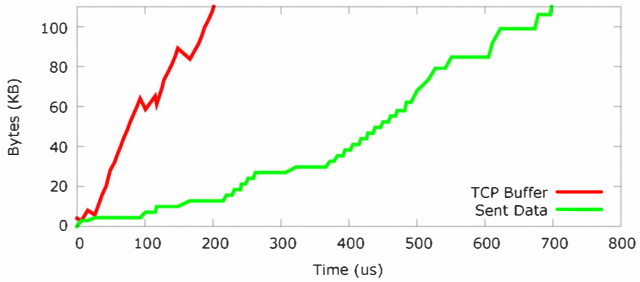


Fig. 2: Amount of data observed in the TCP buffers vs. data observed at the network layer for a flow.

simple approach allows the controller to detect elephant flows without any switch CPU- and bandwidth-intensive monitoring. The Mahout controller then manages only the elephant flows, to maintain a globally optimal arrangement of them.

In the following, we describe Mahout's end host shim layer for detecting elephant flows, our in-band signaling method for informing controller about the elephant flows, and the Mahout network controller.

A. Detecting Elephant Flows

An end host based implementation for detecting elephant flows is better than in-network monitoring/sampling based methods, particularly in datacenters, because: (1) The network behavior of a flow is affected by how rapidly the end-point applications are generating data for the flow, and this is not biased by congestion in the network. In contrast to in-network monitors, the end host OS has better visibility into the behavior of applications. (2) In datacenters, it is possible to augment the end host OS; this is enabled by the single administrative domain and software uniformity typical of modern datacenters. (3) Mahout's elephant detection mechanism has very little overhead (it is implemented with two if statements) on commodity servers. In contrast, using an in-network mechanism to do fine-grained flow monitoring (e.g., using exact matching on OpenFlow's 12-tuple) can be infeasible, even on an edge switch, and even more so on a core switch, especially on commodity hardware. For example, assume that 32 servers are connected, as is typical, to a rack switch. If each server generates 20 new flows per second, with a default flow timeout period of 60 seconds, an edge-switch needs to maintain and monitor 38400 flow entries. This number is infeasible in any of the real switch implementations of OpenFlow that we are aware of.

A key idea of the Mahout system is to monitor end host socket buffers, and thus determine elephant flows before and with lower overheads than with in-network monitoring systems. We demonstrate the rationale for this approach with a micro-benchmark: an *ftp* transfer of a 50MB file from a host 1 to host 2, connected via two switches all with 1 Gbps links.

In Figure 2, we show the cumulative amount of data observed on the network, and in the TCP buffer, as time progresses. The time axis starts when the application first provides data to the kernel. From the graph, one can observe that the application fills the TCP buffers at a rate much higher than the observed network rate. If the threshold for considering

Algorithm 1 Pseudocode for end host shim layer

```
1: When sending a packet
2: if number of bytes in buffer ≥ threshold_elephant then
3:    / * Elephant flow */
4:    if last-tagged-time - now() ≥ T_tagperiod then
5:       set DS = 00001100
6:       last-tagged-time = now()
7:    end if
8: end if
```

a flow as an elephant is 100KB (Figure 2. of [17] shows that more than 85% of flows are less than 100KB), we can see that Mahout's end host shim layer can detect a flow to be an elephant 3x sooner than in-network monitoring. In this experiment there were no other active flows on the network. In further experimental results, presented in Section V, we observe an order of magnitude faster detection when there are other flows.

Mahout uses a shim layer in the end hosts to monitor the socket buffers. When a socket buffer crosses a chosen threshold, the shim layer determines that the flow is an elephant. This simple approach ensures that flows that are bottlenecked at the application layer and not in the network layer, irrespective of how long-lived they are or how many bytes they have transferred, will not be determined as the elephant flows. Such flows need no special management in the network. In contrast, if an application is generating data for a flow faster than the flow's achieved network throughput, the socket buffer will fill up, and hence Mahout will detect this an an elephant flow that needs management.

B. In-band Signaling

Once Mahout's shim layer has detected an elephant flow, it needs to signal this to the network controller. We do this indirectly, by marking the packets in a way that is easily and efficiently detected by OpenFlow switches, and then the switches divert the marked packets to the network controller. To avoid inundating the controller with too many packets of the same flow, the end host shim layer marks the packets of an elephant flow only once every $T_{tagperiod}$ seconds (we use 1 second in our prototype).

To mark a packet, we repurpose the Differentiated Services Field (DS Field) [26] in the IPv4 header. This field was originally called the IP Type-of-Service (IPToS) byte. The first 6 bits of the DS Field, called Differentiated Services Code Point (DSCP), define the per-hop behavior of a packet. The current OpenFlow specification [2] allows matching on DSCP bits, and most commercial switch implementations of OpenFlow support this feature in hardware; hence, we use the DS Field for signaling between the end host shim layer and the network controller. Currently the code point space corresponding to $xxxx11$ (x denotes a wild-card bit) is reserved for experimental or local usage [20], and we leverage this space. When an end host detects an elephant flow, it sets the DSCP bits to 000011 in the packets belonging to that flow.

Algorithm 1 shows pseudocode for the end host shim layer function that is executed when a TCP packet is being sent.

C. Mahout Controller

At each rack switch, the Mahout controller initially configures two default OpenFlow flow table entries: (i) an entry to send a copy of packets with the DSCP bits set to 000011 to the controller and (ii) the lowest-priority entry to switch packets using NORMAL forwarding action. We set up switches to perform ECMP forwarding by default in the NORMAL operation mode. Figure 3 shows the two default entries at the bottom. In this figure, an entry has a higher priority over (is matched before) entries below that entry.

When a flow starts, it normally will match the lowest-priority (NORMAL) rule, so its packet will follow ECMP forwarding. When an end host detects a flow as an elephant and marks a packet of that flow. That packet marked with DSCP 000011 matches the other default rule, and the rack switch forwards it to the Mahout controller. The controller then computes the best path for this elephant, and installs a flow-specific entry in the rack switch.

In Figure 3, we show a few example entries for the elephant flows. Note that these entries are installed with higher priority than Mahout's two default rules; hence, the packets corresponding to these elephant flows are switched using the actions of these flow-specific entries rather than the actions of the default entries. Also, the DS field is set to wildcard for these elephant flow entries, so that once the flow-specific rule is installed, any tagged packets from the end hosts are not forwarded to the controller.

Once an elephant flow is reported to the Mahout controller, it needs to be placed on the best available path. We define the best path for a flow from s to t as the least congested of all paths from s to t. The least congested s-t path is found by enumerating over all such paths.

To manage the elephant flows, Mahout regularly pulls statistics on the elephant flows and link utilizations from the switches, and uses these statistics to optimize the elephant flows' routes. This is done with the increasing first fit algorithm given in Algorithm 2. Correa and Goemans introduced this algorithm and proved that it finds routings that have at most a 10% higher link utilization than the optimal routing [13]. While we cannot guarantee this bound because we reroute only the elephant flows, we expect this algorithm to perform as well as any other heuristic.

D. Discussion

a) DSCP bits: In Mahout, the end host shim layer uses the DSCP bits of the DS field in IP header for signaling elephant flows. However, there may be some datacenters where DSCP may be needed for other uses, such as for prioritization among different types of flows (voice, video, and data) or for prioritization among different customers. In such scenarios, we plan to use VLAN Priority Code Point (PCP) [1] bits. OpenFlow supports matching on these bits too. We can leverage the fact that it is very unlikely for both these code point fields (PCP and DSCP) to be in use simultaneously.

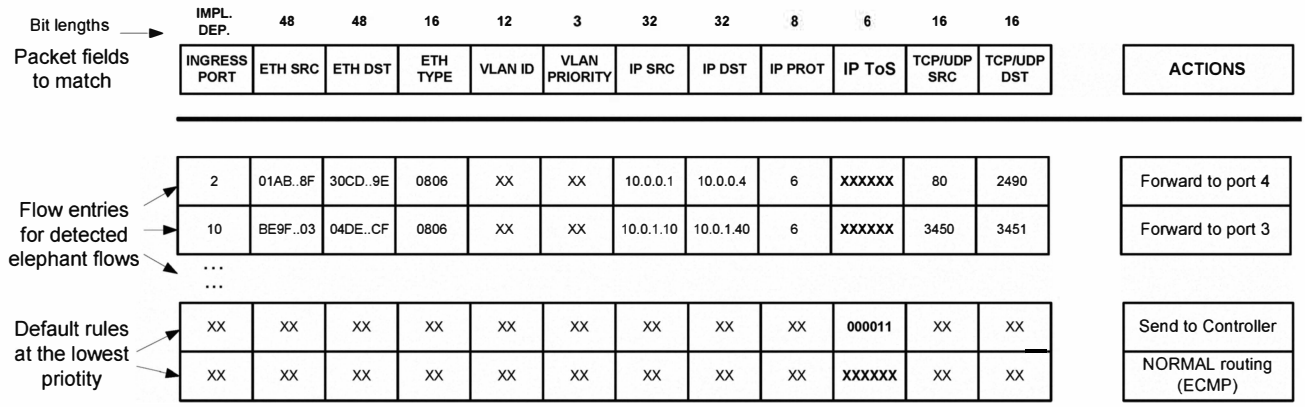


Fig. 3: An example flow table setup at a switch by the Mahout controller.

Algorithm 2 Offline increasing first fit

```
1: sort(F); reverse(F) /* F: set of elephant flows */
2: for f ∈ F do
3:   for l ∈ f.path do
4:     l.load = l.load - f.rate
5:   end for
6: end for
7: for f ∈ F do
8:   best_paths[f].congest = ∞
9:   /* P_st: set of all s-t paths */
10:  for path ∈ P_st do
11:    congest = (f.rate + path.load) / path.bandwidth
12:    if congest < best_path.congest then
13:      best_paths[f] = path
14:      best_paths[f].congest = congest
15:    end if
16:  end for
17: end for
18: return best_paths
```

b) Virtualized Datacenter: In a virtualized datacenter, a single server will host multiple guest virtual machines, each possibly running a different operating system. In such a scenario, the Mahout shim layer needs to be deployed in each of the guest virtual machines. Note that the host operating system will not have visibility into the socket buffers of a guest virtual machine. However, in cloud computing infrastructures such as Amazon EC2 [9], typically the infrastructure provider makes available a few preconfigured OS versions, which include the paravirtualization drivers to work with the provider's hypervisor. Thus, we believe that it is feasible to deploy the Mahout shim layer in virtualized datacenters, too.

c) Elephant flow threshold: Choosing too low a value for $\texttt{threshold}_{elephant}$ in Algorithm 1 can cause many flows to be recognized as elephants, and hence cause the rack switches to forward too many packets to the controller. When there are many elephant flows, to avoid the controller overload, we could provide a means for the controller to signal the end hosts to increase the threshold value. However, this would require a out-of-band control mechanism. An alternative is to use multiple DSCP values to denote different levels of thresholds. For example, $xxxx11$ can be designated to denote that a flow has more than 100KB data, $xxx111$ to denote more than 1MB, $xx1111$ to denote more than 10MB, and so on. The controller can then change the default entry corresponding to the tagged packets (second from bottom in the Figure 3) to select higher thresholds, based on the load at the controller. Further study is needed to explore these approaches.

IV. Analytical Evaluation

In this section, we analyze the expected overhead of detecting elephant flows with Mahout, with flow sampling, and by maintaining per-flow statistics (e.g., the approach used by Hedera). We set up an analytical framework to evaluate the number of switch table entries and control messages used by each method. We evaluate each method using an example datacenter, and show that Mahout is the only solution that can scale to support large datacenters.

Flow sampling identifies elephants by sampling an expected 1 out of k packets. Once it has seen enough packets from the same flow, then the flow is classified as an elephant. The number of packets needed to classify an elephant does not affect our analysis in this section, so we ignore it for now. Hedera [7] uses periodic polling for elephant flow detection. Every t seconds, the Hedera controller pulls the per-flow statistics from each switch. In order to estimate the true rate of a flow (i.e., the rate of the flow if its rate is only constrained by its endpoints' NICs and not by any link in the network), the statistics for every flow in the network must be collected. Pulling statistics for all flows using OpenFlow requires setting up a flow table entry for every flow, so each flow must be sent to the controller before it can be started, so we include this cost in our analysis.

We consider a million server network for the following analysis. Our notation and and the assumed values are shown in the Table I.

Hedera [7]: As table entries need to be maintained for all flows, the number of flow table entries needed at each rack switch is $T \cdot F \cdot D$. In our example, this translates to $32 \cdot 20 \cdot 60 = 38,400$ entries at each rack switch. We are not aware of any existing switch with OpenFlow support that can support this many entries in the flow table in the hardware—for example, HP ProCurve 5400zl switches support up to 1.7K OpenFlow

Parameter	Description	Value
N	Num. of end hosts	2^{20} (1M)
T	Num. of end hosts per rack switch	32
S	Num. of rack switches	2^{15} (32K)
F	Avg. new flows per second per end host	20 [28]
D	Avg. duration of a flow in the flow table	60 seconds
c	Size of counters in bytes	24 [2]
r_{stat}	Rate of gathering statistics	1-per-second
p	Num. of bytes in a packet	1500
f_m	Fraction of mice	0.99
f_e	Fraction of elephants	0.01
r_{sample}	Rate of sampling	1-in-1000
h_{sample}	Size of packet sample (bytes)	60

TABLE I: Parameters and typical values for the analytical evaluation

entries per linecard. It is unlikely that any switch in the near future will support so many table entries given the expense of high-speed memory.

The Hedera controller needs to handle $N \cdot F$ flow setups per second, or more than 20 million requests per second in our example. A single NOX controller can handle only 30,000 requests per second; hence one needs 667 controllers to just handle the flow setup load [28], assuming that the load can be perfectly distributed. Flow scheduling, however, does not seem to be a simple task to distribute.

The rate at which the controller needs to process the statistics packets is

$$= \frac{c \cdot T \cdot F \cdot D}{p} \cdot S \cdot r_{stat}$$

In our example, this implies $(24 \cdot 38400)/1500 \cdot 2^{15} \cdot 1 \approx 20.1M$ control packets per second. Assuming that NOX controller can handle these packets at the rate it can handle the flow setup requests (30,000 per second), this translates to needing 670 controllers just to process these packets. Or, if we consider only one controller, then the statistics can be gathered only once every 670 seconds (≈ 11 minutes).

Sampling: Sampling incurs the messaging overhead of taking samples, and then installs flow table entries when an elephant is detected. The rate at which the controller needs to process the sampled packets is

$$= \text{throughput} \cdot r_{sample} \cdot \frac{\text{bytes per sample}}{p}$$

We assume that each sample contains only a 60 byte header and that headers can be combined into 1500 byte packets, so there are 25 samples per message to the controller. The aggregate throughput of a datacenter network changes frequently, but if 10% of the hosts are sending traffic, the aggregate throughput (in Gbps) is $0.10 \cdot N$. We then find the messaging overhead of sampling to be around 550K messages per second, or if we bundle samples into packets (i.e., 25 samples fit in a 1500 byte packet), then this drops to 22K messages per second.

At first blush, this messaging overhead does not seem like too much overhead; however, as the network utilization increases, the messaging overhead can reach 3.75 million (or 150K if there are 25 samples per packet) packets per second. Therefore, sampling incurs the highest overhead when load balancing is most needed. Decreasing the sampling rate reduces this overhead but adversely impacts the effects of flow scheduling since not all elephants are detected.

We expect the number of elephants identified by sampling to be similar to Mahout, so we do not analyze the flow table entry overhead of sampling separately.

Mahout: Because elephant flow detection is done at the end-host, switches contain flow table entries for elephant flows only. Also, statistics are only gathered for the elephant flows. So, the number of flow entries per rack switch in Mahout is $T \cdot F \cdot D \cdot f_e = 384$ entries. The number of flow setups that the Mahout controller needs to handle is $N \cdot F \cdot f_e$, which is about 200K requests per second, which needs 7 controllers. Also, the number of packets per second that need to be processed for gathering statistics is a f_e fraction of the same in case of Hedera. Thus 7 controllers are needed for gathering statistics at the rate of once per second, or the statistics can be gathered by a single controller at the rate of once every 7 seconds.

V. Experiments

A. Simulations

Our goal is to compare the performance and overheads of Mahout against the competing approaches described in the previous section. To do so, we implemented a flow-level, event-based simulator that can scale to a few thousand end hosts connected using Clos topology [12]. We now describe this simulator and our evaluation of Mahout with it.

1) Methodology: We simulate a datacenter network by modeling the behavior of flows. The network topology is modeled as a capacitated, directed graph and forms a three-level Clos topology. All simulations here are of a 1,600 server datacenter network, and they use a network with a rack to aggregation and aggregation to core links that are 1:5 oversubscribed, i.e., the network has 320Gb bisection bandwidth. All servers have 1Gbps NICs and links have 1Gbps capacity. Our simulation is event-based, so there is no discrete clock—instead, the timing of events is accurate to floating-point precision. Input to the simulator is a file listing the start time, bytes, and endpoints of a set of flows (our workloads are described below). When a flow starts or completes, the rate of each flow is recomputed.

We model the OpenFlow protocol only by accounting for the delay when a switch sets up a flow table entry for a flow. When this occurs, the switch sends the flow to the OpenFlow controller by placing it in its OpenFlow queues. This queue has 10Mbps of bandwidth (this number was measured from an OpenFlow switch [24]). This queue has infinite capacity, so our model optimistically estimates the delay between a switch and the OpenFlow controller since a real system drops arriving packets if one of these queues is full, resulting in TCP timeouts. Moreover, we assume that there is no other overhead when setting up a flow, so the OpenFlow controller deals with the flow and installs flow table entries instantly.

We simulate three different schedulers: (1) an offline scheduler that periodically pulls flow statistics from the switches, (2) a scheduler that behaves like the Mahout scheduler, but

uses sampling to detect elephant flows, and (3) the Mahout scheduler as described in Sec. III-C.

The stat-pulling controller behaves like Hedera [7] and Helios [16]. Here, the controller pulls flow statistics from each switch at regular intervals. The statistics from a flow table entry are 24 bytes, so the amount of time to transfer the statistics from a switch to the controller is proportional to the number of flow table entries at the switch. When transferring statistics, we assume that the CPU-to-controller rate is the bottleneck, not the network or OpenFlow controller itself. Once the controller has statistics for all flows, it computes a new routing for elephant flows and reassigns paths instantly. In practice, computing this routing and inserting updated flow table entries into the switches will take up to hundreds of milliseconds. We allow this to be done instantaneously to find the theoretical best achievable results using an offline approach. The global re-routing of flows is computed using the increasing best fit algorithm described in Algorithm 2. This algorithm is simpler than the simulated annealing employed by Hedera; however, we expect the results to be similar, since this heuristic is likely to be as good as any other (as discussed in Sec. III-C)

As we are doing flow-level simulations, sampling packets is not straightforward since there are no packets to sample from. Instead, we sample from flows by determining the amount of time it will take for k packets to traverse a link, given its rate, and then sample from the flows on the link by weighting each flow by its rate. Full details are in [14].

a) Workloads: We simulate background traffic modeled on recent measurements [21] and add traffic modeled on MapReduce traffic to stress the network. We assume that the MapReduce job has just gone into its shuffle phase. In this phase, each end host transfers 128MB to each other host. Each end host opens a connection to at most five other end hosts simultaneously (as done by default in Hadoop's implementation of MapReduce). Once one of these connections completes, the host opens a connection to another end host, repeating this until it has transferred its 128MB file to each other end host. The order of these outgoing connections is randomized for each end host. For all experiments here, we used 250 randomly selected end hosts in the shuffle load. The reduce phase shuffle begins three minutes after the background traffic is started to allow the background traffic to reach a steady state, and measurements shown here are taken for five minutes after the reduce phase began.

We added background traffic following the macroscopic flow measurements collected by Kandula *et al.* [17], [21] to the traffic mix because datacenters run a heterogeneous mix of services simultaneously. They give the fraction of correspondents a server has within its rack and outside of its rack over a ten second interval. We follow this distribution to decide how many inter- and intra-rack flows a server starts over ten seconds; however, they do not give a more detailed breakdown of flow destinations than this, so we assume that the selection of a destination host is uniformly random across the source server's rack or the remaining racks for an intra- or inter-rack flow respectively. We select the number of bytes in a flow following the distribution of flow sizes in their measurements as well. Before starting the shuffle job, we simulate this background traffic for three minutes. The simulation ends whenever the last shuffle job flow completes.

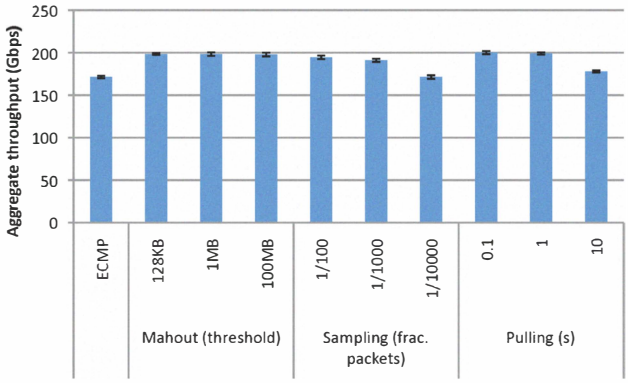


Fig. 4: Throughput results for the schedulers with various parameters. Error bars on all charts show 95% confidence intervals.

b) Metrics: To measure the performance of each scheduler, we tracked the aggregate throughput of all flows; this is the amount of bisection bandwidth the scheduler is able to extract from the network. We measure overhead as before in Section IV, i.e., by counting the number of control messages and the number of flow table entries at each switch. All numbers shown here are averaged from ten runs.

2) Results: The per-second aggregate throughput for the various scheduling methods is shown in Figure 4. We compare these schedulers to static load balancing with equal-cost multipath (ECMP), which uniformly randomizes the outgoing flows across a set of ports [7]. We used three different elephant thresholds for Mahout: 128KB, 1MB, and 100MB, and flows carrying at least this threshold of bytes were classified as an elephant after sending 2, 20, or 2000 packets respectively. As expected, controlling elephant flows extracts more bisection bandwidth from the network—Mahout extracts 16% more bisection bandwidth from the network than ECMP and the other schedulers obtain similar results depending on their parameters.

Hedera's results found that flow scheduling gives a much larger improvement over ECMP than our results (up to 113% on some workloads) [7]. This is due to the differences in workloads. Our workload is based on measurements [21], whereas their workloads are synthetic. We have repeated our simulations using some of their workloads and find similar results: the schedulers improve throughput by more than 100% compared to ECMP on their workloads.

We examine the overhead versus performance tradeoff by counting the maximum number of flow table entries per rack switch and the number of messages to the controller. These results are shown in Figures 5 and 6

Mahout has the least overhead of any scheduling approach considered. Pulling statistics requires too many flow table entries per switch and sends too many packets to the controller to scale to large datacenters; here, the stat-pulling scheduler used nearly 800 flow table entries per rack switch on average

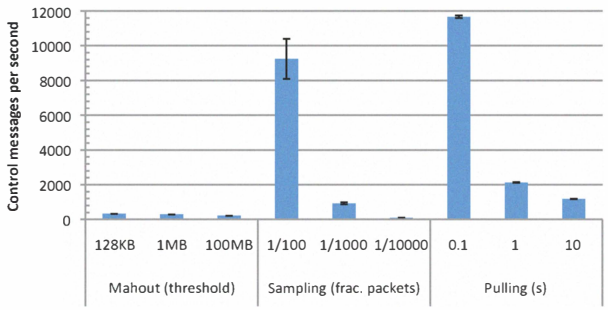


Fig. 5: Number of packets sent to controller by various schedulers. Here, we bundled samples together into a single packet (there are 25 samples per packet)—each bundle of samples counts as a single controller message.

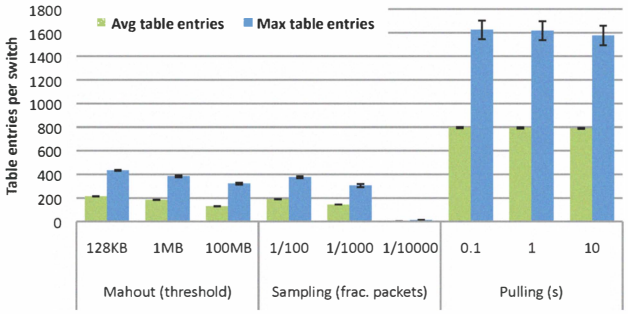


Fig. 6: Average and maximum number of flow table entries at each switch used by the schedulers.

no matter how frequently the statistics were pulled. This is more than seven times the number of entries used by the sampling and Mahout controllers, and makes the offline scheduler infeasible in larger datacenters because the flow tables will not be able to support such a large number of entries. Also, when pulling stats every 1 sec., the controller receives 10x more messages than when using Mahout with an elephant threshold of 100MB.

These simulations indicate that, for our workload, the value of $\texttt{threshold}_{elephant}$ affects the overhead of the Mahout controller, but does not have much of an impact on performance (up to a point: when we set this threshold to 1GB (not shown on the charts), the Mahout scheduler performed no better than ECMP). The number of packets to the Mahout controller goes from 328 per sec. when the elephant threshold is 128KB to 214 per sec. when the threshold is 100MB, indicating that tuning it can reduce controller overhead by more than 50% without affecting the scheduler's performance. Even so, we suggest making this threshold as small as possible to save memory at the end hosts and for quicker elephant flow detection (see the experiments on our prototype in the next section). We believe a threshold of 200–500KB is best for most workloads.

B. Prototype & Microbenchmarks

We have implemented a prototype of the Mahout system. The shim layer is implemented as a kernel module inserted between the TCP/IP stack and device driver, and the controller is built upon NOX [28], an open-source OpenFlow controller written in Python language. For the shim layer, we created a function for the pseudocode shown in Algorithm 1 and invoke it for the outgoing packets, after the IP header creation in

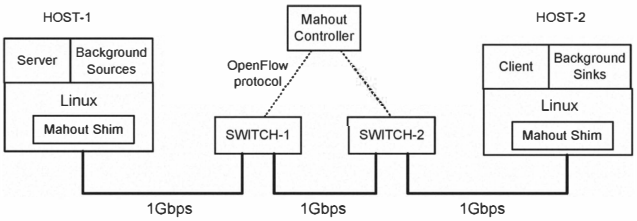


Fig. 8: Testbed for prototype experiments.

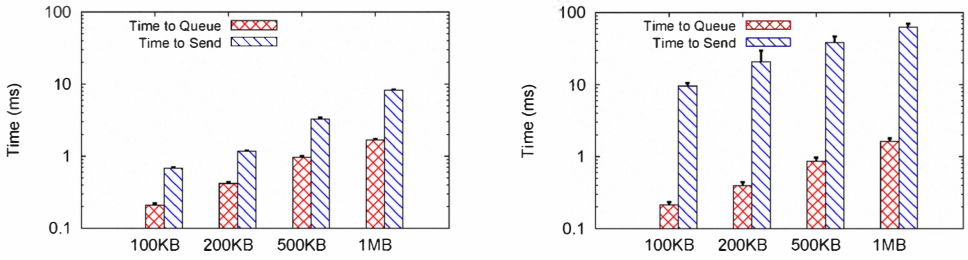
the networking stack. Implementing it as a separate kernel module improves deployability because it can be installed without modifying or upgrading the linux kernel. Our controller leverages the NOX platform to learn topology and configure switches with the default entries. It also processes the packets marked by the shim layer and installs entries for the elephant flows.

Our testbed for experimenting different components of the prototype is shown in the Figure 8. Switches 1 and 2 in the Figure are HP ProCurve 5400zl switches running firmware with OpenFlow support. We have two end hosts in the testbed, one acting as the server for the flows and another acting as the client. For some experiments, we also run some background flows to emulate the other network traffic.

Since this is a not a large scale testbed, we perform microbenchmark experiments focusing on the timeliness of elephant flow detection and compare Mahout against Hedera-like polling approach. We first present measurements of the time it takes to detect an elephant flow at end host and then present the overall time it takes for the controller to detect an elephant flow. Our workload consists of a file transfer using *ftp*. For experiments with presence of background flows, we run 10 simultaneous *iperf* connections.

a) End host elephant flow detection time: In this experiment, we *ftp* a 50MB file from Host-1 to Host-2. We track the number of bytes in the socket buffer for that flow and the number of bytes transferred on the network along with the timestamps. We did 30 trials of this experiment. Figure 2 shows a single run. In Figure 7, we show the time it takes before a flow can be classified as an elephant based on information from the buffer utilization versus based on the number of bytes sent on the network. Here we consider different thresholds for considering a flow as an elephant. We present both cases of with and without background flows. It is clear from these results that Mahout's approach of monitoring the TCP buffers can significantly quicken the elephant flow detection (more than an order of magnitude sooner in some cases) at the end hosts and is also not affected by the congestion in the network.

b) Elephant flow detection time at the controller: In this experiment, we measure how long it takes for an elephant flow to be detected at the controller using the Mahout approach versus using Hedera-like periodic polling approach. To be fair to the polling approach, we have done the periodic polling at the fastest rate possible (poll in a loop without any wait periods in between). As can be seen from the Table II, Mahout controller can detect an elephant flow in few milliseconds.



(a) No background flows (b) 10 background flows

Fig. 7: For each threshold bytes, time taken for the TCP buffer is filled versus the time taken for those many bytes to appear on the network with (a) no background flows and (b) 10 background TCP flows. Error bars show 95% confidence intervals.

Threshold	100KB	200KB	500KB	1MB
Mahout	1.531ms	1.712ms	3.820ms	5.479ms
Hedera	189.83ms	189.83ms	189.83ms	189.83ms

TABLE II: Time it takes to detect an elephant flow at the Mahout controller vs. the Hedera controller, with no other active flows.

In contrast, Hedera takes 189.83ms before the flow can be detected as an elephant irrespective of the threshold. All times are same due to switch's overheads in collecting statistics and relaying it to the central controller.

Overall, our working prototype demonstrates the deployment feasibility of Mahout. The experiments show an order of magnitude difference in the elephant flow detection times at the controller in Mahout vs. a competing approach.

VI. Conclusion

Previous research in datacenter network management has shown that elephant flows—flows that carry large amount of data—need to be detected and managed for better utilization of the multi-path topologies. However, the previous approaches for elephant flow detection are based on monitoring the behavior of flows in the network and hence incur long detection times, high switch resource usage, and/or high control bandwidth and processing overhead. In contrast, we propose a novel end host based solution that monitors the socket buffers to detect elephant flows and signals the network controller using an in-band mechanism. We present Mahout, a low-overhead yet effective traffic management system based on this idea. Our experimental results show that our system can detect elephant flows an order of magnitude sooner than polling based approaches while incurring an order of magnitude lower controller overhead than other approaches.

Acknowledgements

We sincerely thank Sujata Banerjee, Jeff Mogul, Puneet Sharma, and Jean Tourrilhes for several beneficial discussions and comments on earlier drafts of this paper.

References

[1] IEEE Std. 802.1Q-2005, Virtual Bridged Local Area Networks.
[2] OpenFlow Switch Specification, Version 1.0.0. http://www.openflowswitch.org/documents/openflow-spec-v1.0.0.pdf.
[3] sFlow. http://www.sflow.org/.
[4] The OpenFlow Switch Consortium. http://www.openflowswitch.org/.
[5] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. Hyperx: topology, routing, and packaging of efficient large-scale networks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, 2009.
[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.
[7] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.
[8] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Dctcp: Efficient packet transport for the commoditized data center. In *SIGCOMM*, 2010.
[9] `http://aws.amazon.com/ec2/`.
[10] P. Barham, B. Dragovic, K. Fraser, S. H, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.
[11] R. Braden, D. Clark, and S. Shenker. Integrated service in the internet architecture: an overview. Technical report, IETF, Network Working Group, June 1994.
[12] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(5):406–424, 1953.
[13] J. R. Correa and M. X. Goemans. Improved bounds on nonblocking 3-stage clos networks. *SIAM J. Comput.*, 37(3):870–894, 2007.
[14] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-Overhead Datacenter Traffic Management using End-Host-Based Elephant Detection. Technical Report HPL-2010-91, HP Labs, 2010.
[15] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters . In *OSDI*, 2004.
[16] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *SIGCOMM*, 2010.
[17] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *SIGCOMM*, 2009.
[18] Hadoop MapReduce. http://hadoop.apache.org/mapreduce/.
[19] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992 (Informational), Nov. 2000.
[20] IANA DSCP registry. http://www.iana.org/assignments/dscp-registry.
[21] S. Kandula, S. Sengupta, A. Greenberg, and P. Patel. The nature of datacenter traffic: Measurements & analysis. In *IMC*, 2009.
[22] J. Kim and W. J. Dally. Flattened butterfly: A cost-efficient topology for high-radix networks. In *ISCA*, 2007.
[23] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM CCR*, 2008.
[24] J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. R. Curtis, and S. Banerjee. Devoflow: Cost-effective flow management for high performance enterprise networks. In *HotNets*, 2010.
[25] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto. Identifying elephant flows through periodically sampled packets. In *Proc. IMC*, pages 115–120, Taormina, Oct. 2004.
[26] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474 (Proposed Standard), Dec. 1998.
[27] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield. Class-of-service mapping for qos: A statistical signature-based approach to ip traffic classification. In *In IMC04*, pages 135–148, 2004.
[28] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the datacenter. In *HotNets-VIII*, 2009.
[29] VMware. http://www.vmware.com.
[30] Xen. http://www.xen.org.