DARD: Distributed Adaptive Routing for Datacenter Networks

Xin Wu
Dept. of Computer Science, Duke University
Durham, USA
xinwu@cs.duke.edu

Xiaowei Yang
Dept. of Computer Science, Duke University
Durham, USA
xwy@cs.duke.edu

***Abstract*—Datacenter networks typically have many paths connecting each host pair to achieve high bisection bandwidth for arbitrary communication patterns. Fully utilizing the bisection bandwidth may require flows between the same source and destination pair to take different paths. However, existing routing protocols have little support for load-sensitive adaptive routing. We propose DARD, a Distributed Adaptive Routing architecture for Datacenter networks. DARD allows each end host to move traffic from overloaded paths to underloaded paths without central coordination. We use an OpenFlow implementation and simulations to show that DARD can effectively use a datacenter network's bisection bandwidth under both static and dynamic traffic patterns. It outperforms previous solutions based on random path selection by 10%, and performs similarly to previous work that assigns flows to paths using a centralized controller. We use competitive game theory to show that DARD's path selection algorithm makes progress in every step and converges to a Nash equilibrium in finite steps. Our evaluation results suggest that DARD can achieve a close-to-optimal solution in practice.**

***Keywords*-Distributed Adaptive Routing; Datacenter.**

I. Introduction

Datacenter network applications, *e.g.*, MapReduce and network storage, often demand high intra-cluster bandwidth [1]. The components of an application cannot always be placed on machines close to each other for two main reasons. First, applications may share common services, *e.g.*, DNS, search, and storage. These services are not necessarily placed in nearby machines. Second, the auto-scaling feature offered by a datacenter network [2], [3] allows an application to create dynamic instances when its workload increases. Where those instances will be placed depends on machine availability, and is not guaranteed to be close to the application's other instances.

Therefore, it is important for a datacenter network to have high bisection bandwidth to avoid hot spots between any pair of hosts. To achieve this goal, today's datacenter networks often use commodity Ethernet switches to form multi-rooted tree topologies [4] (*e.g.*, fat-tree [5] or Clos topology [6]) that have multiple equal-cost paths connecting any host pair. A flow (a *flow* refers to a TCP connection in this paper) can use an alternative path if one path is overloaded.

However, legacy transport protocols such as TCP lack the ability to dynamically select paths based on traffic load. At a high-level view, there are two types of mechanisms to the explore path diversities in datacenters: centralized dynamic path selection, and distributed traffic-oblivious load balancing. A representative example of centralized path selection is Hedera [1], which uses a central controller to compute an optimal flow-to-path assignment based on dynamic traffic load. Equal-Cost-Multi-Path forwarding (ECMP) [7] and VL2 [6] are examples of traffic-oblivious load balancing. With ECMP, routers hash flows based on flow identifiers to multiple equal-cost next hops. VL2 [6] uses edge switches to forward a flow to a randomly selected core switch to achieve valiant load balancing.

Both of these two design paradigms improve the available bisection bandwidth. Yet each has its limitations. A centralized path selection approach can reach near-optimal flow allocation. However, it introduces a potential scaling bottleneck and a centralized point of failure. When a datacenter scales to a large size, the control traffic sent to and from the controller may congest the link connecting the controller and the rest of the network. Distributed traffic-oblivious load balancing scales well to large datacenter networks, but may still create hot spots, as their flow assignment algorithms do not consider dynamic traffic load.

In this paper, we aim to combine the best of both worlds, designing a datacenter routing system that is both scalable and able to balance dynamic traffic among multiple paths. To this end, we propose DARD, a lightweight and distributed routing system that enables end hosts to select paths based on dynamic traffic patterns. This design paradigm has two main advantages. First, a distributed design can be more robust and scale better than a centralized approach. Second, placing the path selection logic at an end system facilitates deployment, as it does not require special hardware to replace commodity switches.

A key design challenge DARD faces is how to achieve dynamic distributed load balancing with low overhead. In DARD, no end system or router has a global view of the network. Each end system can only select a path based on its local knowledge, thereby making it difficult to achieve close-to-optimal load balancing. To address this challenge, DARD uses a selfish path selection algorithm that provably converges to a Nash equilibrium in finite steps (Appendix B). Our experimental evaluation shows that the equilibrium's gap to the optimal solution is small. To facilitate path

selection, DARD uses hierarchical addressing to represent an end-to-end path with a pair of source and destination addresses.

We have implemented a DARD prototype on DeterLab [8] and ns-2 simulator. We use static traffic patterns to show DARD converges to a stable state in two to three control intervals. We use dynamic traffic patterns to show the bisection bandwidth DARD achieves is larger than ECMP, VL2 and TeXCP and is slightly smaller than the bisection bandwidth achieved by centralized scheduling.

The rest of this paper is organized as follows. Section II introduces background knowledge and discusses related work. Section III describes DARD's design goals and system components. In Section IV, we introduce the system implementation details. We evaluate DARD in Section V. Section VI concludes our work.

II. Background and Related Work

A. Datacenter Topologies

Recent proposals [5], [6], [9] suggest to use multi-rooted tree topologies to build datacenter networks. Figure 1 shows a 3-stage multi-rooted tree topology. The topology has three vertical layers: *Top-of-Rack* (*ToR*), *aggregation*, and *core*. A *pod* is a management unit. It represents a replicable building block consisting of a number of servers and switches that share the same power and management infrastructure.

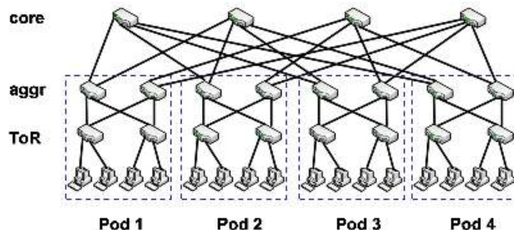

Figure 1. **A multi-rooted tree topology for a datacenter network.**

We design DARD to work for arbitrary multi-rooted tree topologies. But for ease of exposition, we mostly use the fat-tree topology [5] to illustrate DARD's design.

In this paper, we use the term *elephant flow* to refer to a continuous TCP connection larger than a threshold defined as the number of transferred bytes. We discuss how to choose this threshold in Section III.

B. Related work

Related work falls into three broad categories: adaptive path selection in datacenters, end host based multipath transmission, and traffic engineering protocols.

Adaptive path selection: Adaptive path selection mechanisms [1], [6], [7] can be further divided into centralized and distributed approaches. Hedera [1] adopts a centralized approach in the granularity of a flow. In Hedera, edge switches detect and report elephant flows to a centralized controller. The controller calculates a path arrangement and periodically updates switches' routing tables. Hedera can almost fully utilize a network's bisection bandwidth, but a recent data center traffic measurement suggests that this centralized approach needs parallelism route computation to support dynamic traffic patterns [10].

An ECMP-enabled switch [7] is configured with multiple next hops for a given destination and forwards a packet according to a hash of selected fields of the packet header. Since packets of the same flow share the same hash value, they take the same path and maintain the packet order. However, multiple large flows can collide on their hash values and congest an output port [1].

VL2 [6], on the other hand, places the path selection logic at edge switches. An edge switch first forwards a flow to a randomly selected core switch, which then forwards the flow to the destination. As a result, multiple elephant flows can still get collided on the same core switch.

DARD differs from ECMP and VL2 in two key aspects. First, its path selection algorithm is load sensitive, in which flows are shifted from overloaded paths to underloaded paths. Second, it places the path selection logic at end systems to facilitate deployment. A datacenter network can deploy DARD by upgrading its end system's software stack instead of updating switches.

Multi-path Transmission: Multipath TCP (MPTCP) [11] enables an end host to simultaneously use multiple paths to improve its throughput. Different from MPTCP, DARD is transparent to applications since it is implemented as a path selection module under the transport layer (Section III).

Traffic Engineering Protocols: Traffic engineering protocols such as TeXCP [12] are originally designed for ISP networks, and has the potential to be adopted by datacenter networks. However, TeXCP forward traffic along different paths in the granularity of a packet rather than a flow, which can cause TCP reordering problem.

III. DARD Design

In this section, we first highlight the system design goals. Then we present the design overview and details in the following sub-sections.

A. Design Goals

DARD's essential goal is to effectively utilize a datacenter's bisection bandwidth with practically deployable mechanisms and limited control overhead.

1. Efficiently utilizing the bisection bandwidth: We aim to take advantage of the multiple paths to fully utilize the bisection bandwidth. Meanwhile we desire to prevent any systematic risk that can cause packet reordering

2. Fairness among elephant flows: We aim to provide fairness among elephant flows so that concurrent elephant flows can evenly share the available bisection bandwidth. We focus our work on elephant flows mainly because elephant flows occupy a significant fraction of the total bandwidth (more than 90% of bytes are in the 1% of the largest flows [6]).

3. Lightweight and scalable: We aim to design a lightweight and scalable system. We desire to avoid a centralized scaling bottleneck and minimize the amount of control traffic and computation

4. Practically deployable: We aim to make DARD compatible with existing datacenter infrastructures so that it can be deployed without significant modifications or upgrade of existing infrastructures.

B. Overview

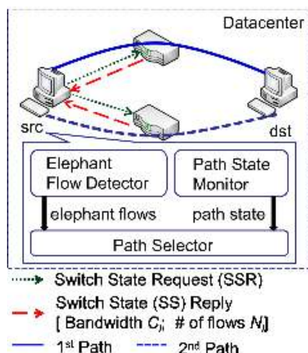

Figure 2. DARD has three components. The *Elephant Flow Detector* detects elephant flows. The *Path State Monitor* periodically queries the traffic load on each path. The *Path Selector* moves flows from overloaded paths to underloaded paths.

Figure 2 shows DARD's system components. A switch in DARD has only two functions: (1) it forwards packets to the next hop according to a pre-configured routing table; (2) it tracks the *Switch State* (*SS*, Section III-D) and replies to end systems' *Switch State Request* (*SSR*).

An end system has three components as shown in Figure 2. The *Elephant Flow Detector* monitors all the output flows and treats one flow as an elephant once its size grows beyond a threshold. We use 100KB as the threshold because more than 85% of flows in a datacenter are less than 100 KB [6]. The *Path State Monitor* sends SSR to the switches and assembles the SS replies into *Path State* (*PS*, Section III-D), which indicates the load on each path. Based on the path states and the elephant flow traffic demand, the *Path Selector* periodically assigns flows from overloaded paths to underloaded paths.

The rest of this section presents DARD's design details, including enabling end hosts to select paths using hierarchical addressing (Section III-C), Scalably monitoring path states (Section III-D) and adaptively assign flows to different paths (Section III-E).

C. Addressing and Routing

To fully utilize the bisection bandwidth and, at the same time, to prevent retransmissions caused by packet reordering, we allow a flow to take different paths in its life cycle. However, one flow can use only one path at any given time.

A datacenter network is usually constructed as a multi-rooted tree. In Figure 3, all the devices highlighted by the solid circles form a tree with its root $core_1$. This strictly hierarchical structure facilitates adaptive routing via some customized addressing rules [5]. We borrow the idea from NIRA [13] to split an end-to-end path into *uphill* and *downhill* segments and to encode a path in the source and destination addresses. In DARD, each of the core switches obtains a unique prefix and recursively allocates non-overlapping subdivisions of the prefix to its sub-trees. By this hierarchical prefix allocation, each network device receives multiple IP addresses, each of which represents the device's position in one of the trees.

As shown in Figure 3, we use $core_i$ to refer to the ith core, $aggr_{ij}$ to refer to the jth aggregation switch in the ith pod. We follow the same rule to interpret ToR_{ij} for the top of rack switches and E_{ij} for the end hosts. We use the device names prefixed with letter P and delimited by colons to illustrate how prefixes are allocated along the hierarchies. The first core is allocated with prefix $Pcore_1$. It then allocates non-overlapping prefixes $Pcore_1.Paggr_{11}$ and $Pcore_1.Paggr_{21}$ to two of its sub-trees. The sub-tree rooted from $aggr_{11}$ will further allocate four prefixes to lower hierarchies.

For a general multi-rooted tree topology, the datacenter operators can generate a similar addressing schema and allocate the prefixes along the topology hierarchies. In case more IP addresses are assigned to one network cards, we propose to use IP alias to configure multiple IP addresses to one network interface. The latest operating systems support a large number of IP alias to associate with one network interface, *e.g.*, Linux kernel 2.6 sets the limit to be $256K$ IP alias per interface [14]. Windows NT 4.0 has no limitation on the number of IP addresses per interface [15].

One nice property of this hierarchical addressing is that one address uniquely encodes the sequence of upper-level switches that allocate that address, *e.g.*, in Figure 3, E_{11}'s address $Pcore_1.Paggr_{11}.PToR_{11}.PE_{11}$ uniquely encodes the address allocation sequence $core_1 \rightarrow aggr_{11} \rightarrow ToR_{11}$. A source and destination address pair can further uniquely identify a path, *e.g.*, in Figure 3, we can use the source and destination pair highlighted by dotted circles to uniquely encode the dotted path from E_{11} to E_{21} through $core_1$. We call the partial path encoded by source address the *uphill path* and the partial path encoded by destination address the *downhill path*. To move a flow to a different path,

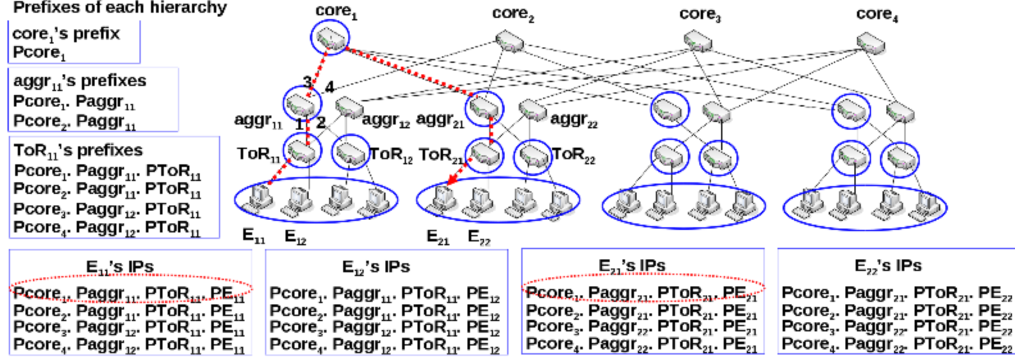

Figure 3. DARD's addressing and routing. E_{11}'s address $Pcore_1.Paggr_{11}.PToR_{11}.PE_{11}$ encodes the uphill path ToR_{11}-$aggr_{11}$-$core_1$. E_{21}'s address $Pcore_1.Paggr_{21}.PToR_{21}.PE_{21}$ encodes the downhill path $core_1$-$aggr_{21}$-ToR_{21}.

we can simply use different source and destination address combinations without dynamically reconfiguring the routing tables.

To forward a packet, each switch stores a *downhill table* and an *uphill table*. The uphill table keeps the entries for the upstream switch prefixes and the downhill table keeps the entries for the downstream switch prefixes. Table I shows the switch $aggr_{11}$'s downhill and uphill table. The port indexes are marked in Figure 3. When a packet arrives, a switch first looks up the destination address in the downhill table using the longest prefix matching algorithm. If a match is found, the packet will be forwarded to the corresponding downstream switch. Otherwise, the switch will look up the source address in the uphill table to forward the packet to the corresponding upstream switch. A core switch only has the downhill table. In fact, the downhill-uphill-looking-up is not necessary for a fat-tree topology, since a core switch in a fat-tree uniquely determines the entire path. However, not all the multi-rooted trees share the same property, *e.g.*, a Clos network.

downhill table	
Prefixes	Port
$Pcore_1.Paggr_{11}.PToR_{11}$	1
$Pcore_1.Paggr_{11}.PToR_{12}$	2
$Pcore_2.Paggr_{11}.PToR_{11}$	1
$Pcore_2.Paggr_{11}.PToR_{12}$	2
uphill table	
Prefixes	Port
$Pcore_1$	3
$Pcore_2$	4

Table I
$aggr_{11}$'S DOWNHILL AND UPHILL ROUTING TABLES.

Each network component in DARD is also assigned a location independent IP address, *ID*, which uniquely identifies the component and is used for making TCP connections. The mapping from IDs to underlying IP addresses is maintained by a DNS system and cached locally. To deliver a packet, the source encapsulates the packet with a proper source and destination address pair. Switches in the middle will forward the packet according to the encapsulated packet header. When the packet arrives at the destination, it will be decapsulated and passed to upper layer protocols.

D. Path State Monitoring

To achieve load-sensitive path selection at an end host, DARD informs every end host with the link loads. Each end host will accordingly select paths for its outbound elephant flows. Because on-demand active probing has the capability of limiting the control traffic, we choose to let DARD's *Path State Monitor* to actively probe for link loads rather than to passively receive link load updates. This section first describes a straw man design of the *Path State Monitor*. Then we discuss how to improve it.

We use C_l to note the output link l's capacity. N_l denotes the number of elephant flows on the link. We define *link l's Fair Share* $S_l = C_l/N_l$ for the bandwidth each elephant flow will get if they fairly share that link ($S_l = 0$, if $N_l = 0$). Output link l's *Link State* (LS_l) is defined as a triple [C_l, N_l, S_l]. A switch r's *Switch State* (SS_r) is defined as $\{LS_l$, l is r's output link$\}$. A path p refers to a set of links that connect a source and destination ToR switch pair. If link l has the smallest S_l among all the links on path p, we use LS_l to represent p's *Path State* (PS_p).

In the straw man design, each switch tracks its *Switch State* locally. An end host's *Path State Monitor* periodically sends the *Switch State Requests* (*SSR*) to every switch and assembles the *Switch State Replies* into the *Path States*, which indicate the load on each path. This straw man design requires every switch to have a customized flow counter and the capability of replying to *SSR*. These two functions are already supported by OpenFlow enabled switches [16].

In the straw man design, The control traffic in every control interval can be estimated using formula (1), where pkt_size refers to the sum of request and response packet

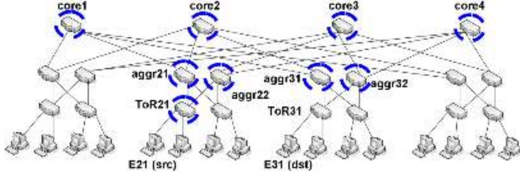

Figure 4. The set of switches a source sends *SSRs* to.

Algorithm 1 Selfish Path Selection

1: max_S = maximum S_l in $P.PV$;
2: max_i = max_S's index in $P.PV$;
3: min_S = minimum S_l in $P.PV$,
whose corresponding element in $P.FV > 0$;
4: min_i = min_S's index in $P.PV$;
5: **if** $max_i \neq min_i$ **then**
6: $estimation = \frac{P.PV[max_i].C_l}{P.PV[max_i].N_l+1}$
7: **if** $estimation - min_S > \delta$ **then**
8: move one elephant flow from path min_i to path max_i.
9: **end if**
10: **end if**

sizes. The amount of control traffic is bounded by the topology size.

$$num_of_servers \times num_of_switches \times pkt_size \quad (1)$$

We further improve the straw man design by decreasing the control traffic. There are two intuitions for the optimizations. First, if an end host is not sending any elephant flow, it is not necessary to monitor the path state. Second, an end host does not have to query every switch for the path states. As shown in Figure 4, E_{21} is sending elephant flows to E_{31}. The switches highlighted by the dotted circles are the ones E_{21} needs to send *SSR* to, since the rest switches are not on any path. We do not highlight ToR_{31}, because it is the last hop switch to the destination. Based on these two observations, each end host only needs to send *SSRs* to the switches on the paths to the elephant flows' destinations.

E. Path Selection

As shown in Figure 2, a *Path Selector* running on an end host takes the detected elephant flows and *Path States* as the input and periodically moves flows from overloaded paths to underloaded paths.

Given an elephant flow's elastic traffic demand and small delays in datacenters, elephant TCP flows tend to fully and fairly utilize their bottlenecks. As a result, moving one flow from a path with small *Fair Share* (defined in Section III-D) to a path with large *Fair Share* will push both the small and large *Fair Shares* toward the middle and thus improve fairness. Based on this observation, we propose DARD's path selection algorithm, whose high level idea is to enable every end host to selfishly increase the minimum *Fair Share* they observe. The Algorithm *Selfish Path Selection* illustrates one iteration of the path selection process.

In DARD, every source and destination pair maintains two vectors: the *Path State Vector* (PV), whose ith item is the *Path State* of the ith path (the triple [C_l, N_l, S_l] defined in Section III-D), and the *Flow Vector* (FV), whose ith item is the number of elephant flows the source is sending along the ith path. Line 6 estimates the *Fair Share* of the max_ith path if another elephant flow is moved to it. The δ in line 7 is a positive threshold to decide whether to move a flow. If we set δ to 0, line 7 makes sure this algorithm will not decrease the global minimum *Fair Share*. If we set δ to be larger than 0, the algorithm will converge as soon as the estimation being close enough to the current minimum *Fair Share*. In general, a small δ will evenly split elephant flows among different paths and a large δ will accelerate the algorithm's convergence.

Load-sensitive routing protocols can lead to oscillations and instability [17]. Figure 5 shows an example. There are three source and destination pairs, (E_1, E_2), (E_3, E_4) and (E_5, E_6). Each of the pairs has two paths and two elephant flows. The source in each pair will run DARD independently without knowing the other two's behaviors. In the beginning, the shared path, (link $switch_1$-$switch_2$), has no elephant flows on it. As a result, the three sources will move flows to it and increase the number of elephant flows on the shared path to three. This will further cause the three sources to move flows off from the shared paths. The three sources repeat this process and cause permanent oscillation and bandwidth under-utilization.

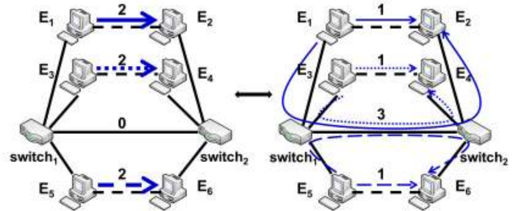

Figure 5. Path oscillation example.

The reason for path oscillation is different sources synchronously move flows to under-utilized paths. To prevent this oscillation, DARD adds a random time slot to the interval between two adjacent flow movements. The evaluation in Section V-C3 shows this simple solution can prevents path oscillation very well.

IV. Implementation

To test DARD's performance and to show DARD is practically deployable, we implemented a prototype and deployed it in the fat-tree topology in DeterLab [8], as shown in Figure 3. We also implemented a simulator on ns-2 to evaluate DARD's performance on different topologies.

A. Test Bed

We set up a fat-tree topology using 4-port PCs acting as the switches and configure IP addresses according to Section III-C. All PCs run the Ubuntu 10.04 LTS standard image. All switches run OpenFlow 1.0. An OpenFlow enabled switch maintains per flow and per port statistics and allows us to customize the forwarding table

We implement a NOX [18] component to configure all switches' flow tables during their initialization. This component allocates the downhill table to flow table 0 and the uphill table to flow table 1 to enforce a higher priority for the downhill table. All entries are permanent. Each link's bandwidth is $100Mbps$.

The *Elephant Flow Detector* leverages the TCPTrack [19] at each end host to reports an elephant flow if a TCP connection grows beyond 100KB [6]. The *Path State Monitor* tracks the *Fair Share* of all the equal-cost paths connecting the source and destination ToR switches. It queries switches for their states using the *aggregate flow statistics* interfaces provided by OpenFlow infrastructure [20]. The query interval is set to 1 second. This interval causes acceptable amount of control traffic as shown in Section V-C4. We leave exploring the impact of varying this interval to our future work. The δ in the *Selfish Path Selection* algorithm is set to $10Mbps$. This number is a tradeoff between maximizing the minimum flow throughput and fast convergence. The flow movement interval is 5 seconds plus a random time from $[0s, 5s]$. On the one hand, this conservative interval setting limits the frequency of flow movement, and on the other hand, it also prevents an elephant flow from being delivered even without the chance to be moved to a less congested path, because a significant amount of elephant flows last for more than 10s [4]. We use the Linux IP-in-IP tunneling as the encapsulation/decapsulation module. All the mappings from IDs to underlying IP addresses are kept at end hosts.

B. Simulator

To evaluate DARD's performance in larger topologies, we build a DARD simulator on ns-2, which captures the system's packet level behavior. A link's bandwidth is $1Gbps$ and its delay is $0.01ms$. The queue size is set to be the delay-bandwidth product. TCP New Reno is used as the transport protocol. We use the same settings as the test bed for the rest of the parameters.

V. Evaluation

This section describes the evaluation of DARD using DeterLab test bed and ns-2 simulation. We focus this evaluation on four aspects. (1) Can DARD fully utilize the bisection bandwidth? (2) How fast can DARD converge to a stable state? (3) Will DARD's distributed algorithm cause any path oscillation? (4) How much is DARD's control overhead?

A. Traffic Patterns

Due to the absence of commercial datacenter network traces, we use the three traffic patterns introduced in [5] for both our test bed and simulation evaluations. (1) *Stride*, where an end host with index E_{ij} sends elephant flows to the end host with index E_{kj}, $i \neq k$. This traffic pattern emulates the extreme case where the traffic stresses out the links between the core and the aggregation layers. (2) *Staggered*(P_{ToR},P_{Pod}), where an end host sends elephant flows to another end host connecting to the same ToR switch with probability P_{ToR}, to any other end host in the same pod with probability P_{Pod} and to the end hosts in different pods with probability $1 - P_{ToR} - P_{Pod}$. In our evaluation P_{ToR} is 0.5 and P_{Pod} is 0.3. This traffic pattern emulates the case where an application's instances are close to each other and the most traffic is in the same pod or even under the same ToR switch. (3) *Random*, where an end host sends elephant flows to any other end host in the topology with a uniform probability.

The above three traffic patterns can be either $static$ or $dynamic$. The static traffic refers to a number of permanent elephant flows. The dynamic traffic means the elephant flows start at different times and transfer large files of different sizes. According to [4], we set the elephant flow inter-arrival time to be 75ms and the file size to be uniformly distributed between 100MB and 1GB.

B. Test Bed Results

The purpose of the test bed evaluation is to prove DARD is readily deployable and can fully utilize the bisection bandwidth in practice. We use the static traffic patterns and constantly measure the incoming bandwidth at every end host. The experiment lasts for one minute. We use the results from the middle 40 seconds to calculate the average bisection bandwidth.

We also implement a static hash-based ECMP and a modified version of flow-level VLB in the test bed. In the ECMP implementation, a flow is forwarded according to a hash of the source and destination's IP addresses and TCP ports. In the flow-level VLB implementation, we randomly picks a core every 10 seconds for an elephant flow to prevent long term collisions. This 10s interval is set roughly the same as DARD's control interval for a fair comparison. We note this implementation as *periodical VLB* (*pVLB*). We will compare DARD and other mechanisms besides ECMP and pVLB in the simulation.

Figure 6 shows the comparison of DARD, ECMP and pVLB's bisection bandwidths under different static traffic patterns. DARD outperforms both ECMP and pVLB. We also observe that the bisection bandwidth gap between DARD and the other two approaches increases in the order of staggered, random and stride. This is because flows through the core have more path diversities than the flows inside a pod. Compared with ECMP and pVLB, DARD's

strategic path selection reaches a better flow allocation than simply relying on randomness.

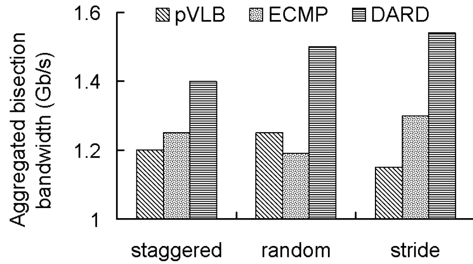

Figure 6. **DARD, ECMP and pVLB's bisection bandwidths under different static traffic patterns. Measured on test bed.**

We also compare DARD and ECMP's large file transfer times under dynamic traffic patterns. We vary each source-destination pair's flow generating rate from 1 to 10 per second. Each elephant flow is a TCP connection transferring a 128MB file. We use a fixed file length because we need to differentiate whether finishing a flow earlier is caused by a better path selection algorithm or a smaller file. The experiment lasts for five minutes. We track the start and the end time of every elephant flow and calculate the average file transfer times for both DARD and ECMP.

Figure 7 shows DARD's average file transfer time improvement over ECMP $vs.$ the flow generating rate under different traffic patterns. For the stride traffic, DARD outperforms ECMP because DARD moves flows from overloaded paths to underloaded ones and increases the minimum flow throughput in every step. We find random and staggered traffic share an interesting pattern. When the flow generating rate is low, ECMP and DARD have almost the same performance because the bandwidth is over-provided. As the flow generating rate increases, cross-pod flows congest the switch-to-switch links, in which case DARD can reallocate the flows sharing the same bottleneck and improves the average file transfer time. When flow generating rate becomes even higher, the host-switch links are occupied by flows within the same pod and thus become the bottlenecks, in which case no path selection algorithm can bypass them. The comparison of DARD and pVLB follows a similar pattern.

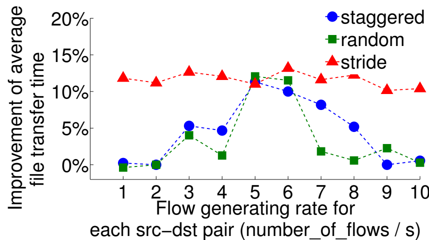

Figure 7. **File transfer improvement. Measured on test bed.**

C. Simulation Results

We also compare DARD with Hedera, TeXCP and MPTCP in our simulation. We implement both the demand-estimation and the simulated annealing algorithm described in Hedera and set its scheduling interval to 5 seconds [1]. In the TeXCP implementation each ToR switch pair maintains the utilizations for all the available paths by periodically probing (We decrease the default probe interval to 10ms given datacenter's small RTT). We do not implement the flowlet [21] mechanisms in the simulator. As a result, each ToR switch schedules traffic at packet level. We use MPTCP's ns-2 implementation [22] to compare with DARD. Each MPTCP connection uses all the simple paths connecting the source and destination pair.

1) Performance Improvement: We use static traffic pattern on a fat-tree with 1024 hosts to evaluate whether DARD can fully utilize the bisection bandwidth in a larger topology. Figure 8 shows the result. DARD achieves higher bisection bandwidth than both ECMP and pVLB under all the three traffic patterns. As a centralized method, Hedera slightly outperforms DARD under stride and random traffic patterns. However, Hedera achieves less bisection bandwidth than DARD under staggered traffic pattern. This is because current Hedera only schedules the flows going through the cores. When intra-pod traffic is dominant, Hedera degrades to ECMP.

Even though TeXCP reaches similar bisection bandwidth as DARD, we observe TeXCP's medium retransmission rate is 3% while DARD's this number is less than 1%. TeXCP's high retransmission rate shows packet level scheduling in datacenter network does cause reordering problems.

MPTCP has comparable bisection bandwidth as DARD. However, it achieves less bisection bandwidth than Hedera. We suspect this is because the current MPTCP's ns-2 implementation does not support flow level retransmission. Thus, lost packets are always retransmitted along the same path regardless how congested the path is. We leave a comprehensive comparison between DARD and MPTCP as our future work.

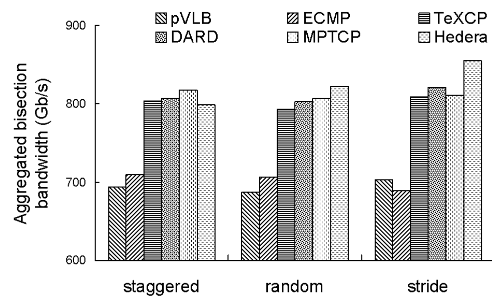

Figure 8. **Bisection bandwidth under different static traffic patterns. Simulated on fat-tree with** 1024 **end hosts.**

2) Convergence Speed: DARD is provable to converge to a Nash equilibrium (Appendix B). However, if the convergence takes a significant amount of time, the network will be underutilized during the convergence process. As a result, we measure how fast can DARD converge to a stable state, in which every flows stops changing paths. We use the static traffic patterns on a fat-tree with 1024 hosts. For each source and destination pair, we vary the number of elephant flows from 1 to 64. We start these elephant flows simultaneously and track the time when all the flows stop changing paths. Figure 9 show the CDF of DARD's convergence time. DARD converges in less than 25s for more than 80% of the cases. Given DARD's control interval at each end host is roughly 10s, the entire system converges in less than three control intervals. DARD converges faster under staggered traffic pattern, because its dominant intra-pod flows have less path diversity.

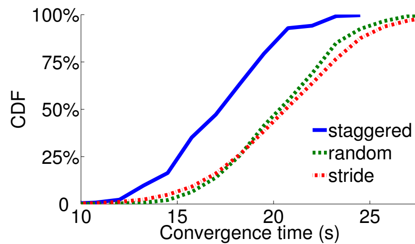

Figure 9. **DARD converges to a stable state in 2 or 3 control intervals given static traffic patterns.**

3) Stability: DARD adds a random span of time to the control interval to prevent path oscillation. This section evaluates the effects of this simple mechanism.

We use dynamic random traffic pattern on a fat-tree with 128 end hosts. Because a core's output link is usually the bottleneck for a inter-pod elephant flow [1], we track this bottleneck link utilization at the core. Figure 10 shows the link utilizations on the 8 output ports of the first core switch. As we can see, after the initial oscillation the link utilizations stabilize afterward.

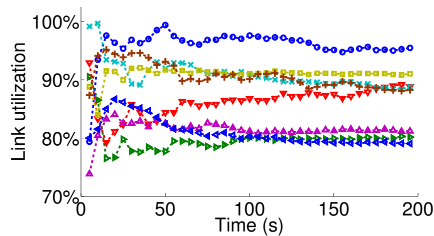

Figure 10. **The first core switch's output port utilizations under dynamic random traffic pattern.**

However we cannot simply conclude that DARD does not cause path oscillations, because the link utilization misses single flow's behavior. We first disable the random time added to the control interval and log every single flow's path selection history. We find even though the link utilizations are stable, certain flows are constantly moved between two paths, *e.g.*, one 512MB elephant flow are moved between two paths 23 times in its life cycle. This indicates path oscillation exists in load-sensitive adaptive routing.

After many attempts, we choose to add a random span of time to the control interval to address this path oscillation problem. Figure 11 shows the CDF of how many times flows change their paths in their life cycles. For the staggered traffic, around 90% of the flows stick to their original paths. This indicates when most of the flows are within the same pod or even the same ToR switch, the bottleneck is most likely located at the host-switch links, in which case few path diversities exist. On the other hand, for the stride traffic, where all flows are inter-pod, around 50% of the flows do not change their paths. Another 50% change their paths for less than 4 times. This small number of path changing times indicates that DARD is stable.

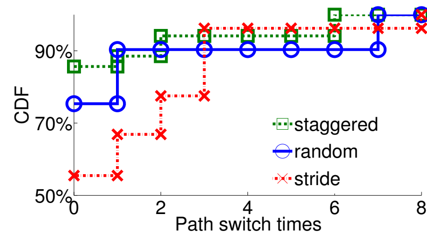

Figure 11. **CDF of the times that flows change their paths under dynamic traffic patterns.**

4) Control Overhead: To evaluate DARD's communication overhead, we trace the control messages for both DARD and Hedera on a fat-tree with 128 hosts under static random traffic pattern. DARD's communication overhead is mainly caused by the periodical probes, including both queries from hosts and replies from switches. This communication overhead is bounded by the size of the topology, because all pair probing is the worst case. On the other hand, Hedera's communication overhead is proportional to the number of elephant flows, since the ToR switches report every detected elephant flow to the centralized controller.

Figure 12 shows how much of the bandwidth is taken by control messages given different number of elephant flows. With the increase of the number of elephant flows, there are three stages. In the first stage, DARD's control messages take less bandwidth than Hedera's. The reason is mainly because DARD has a smaller control message size (Hedera's control message payload is either 80 or 72 Bytes, while DARD's is either 48 or 32 bytes). In the second stage, DARD's control messages take slightly more bandwidth. That is because one new elephant flow introduces more control messages in DARD than that in Hedera. In the third stage, DARD's probe traffic is eventually bounded by the

topology size. However, Hedera's communication overhead increases proportionally to the number of elephant flows.

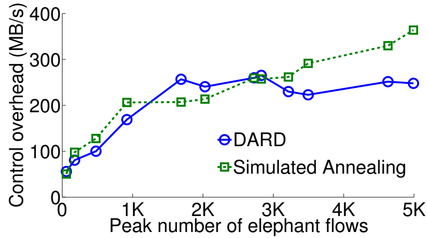

Figure 12. **DARD and Hedera's communication overhead.**

VI. Conclusion

This paper proposes DARD, a readily deployable, lightweight and distributed adaptive routing system for datacenter networks. DARD allows each end host to selfishly move elephant flows from overloaded paths to underloaded paths. Our analysis shows that DARD converges to a Nash equilibrium in finite steps. Test bed emulation and ns-2 simulation show the bisection bandwidth DARD achieves is larger than the distributed traffic-oblivious load balancing and comparable to the centralized scheduling.

Appendix

A. Explanation of the objective

We assume TCP is the dominant transport protocol in datacenter, which tries to achieve max-min fairness if combined with fair queuing. Each end host moves flows from overloaded paths to underloaded ones to increase its observed minimum *Fair Share*. This section explains given max-min fair bandwidth allocation, the global minimum *Fair Share* is the lower bound of the global minimum flow throughput, thus increasing the minimum *Fair Share* actually increases the global minimum flow throughput.

Theorem 1. Given max-min fair bandwidth allocation, the global minimum *Fair Share* is the lower bound of global minimum flow throughput.

First we define a bottleneck link according to [23]. A link l is a bottleneck for a flow f if and only if (a) link l is fully utilized, and (b) flow f has the maximum throughput among all the flows using link l.

A link l_i's *Fair Share* S_i is defined as C_i/N_i, where C_i is the link capacity and N_i is the number of elephant flows via the link. We assume link l_0 has the minimum *Fair Share* S_0. Flow f has the minimum throughput, min_tput. Link l_f is flow f's bottleneck. Theorem 1 claims $min_tput \geq S_0$. We prove this theorem using contradiction.

According to the bottleneck definition, min_tput is the maximum flow rate on link l_f, and thus $C_f/N_f \leq min_tput$. Suppose $min_tput < S_0$, we get

$$C_f/N_f < S_0 \tag{A1}$$

which is conflict with S_0 being the minimum *Fair Share*. As a result, the minimum *Fair Share* is the lower bound of the global minimum flow throughput.

In DARD, every end host tries to increase its observed minimum *Fair Share* in each round, thus the global minimum *Fair Share* keeps increasing, so does the global minimum flow throughput.

B. Convergence proof

We now formalize DARD's selfish path selection algorithm as a *congestion game* [24] and prove it converges to a Nash equilibrium in finite steps.

We use p^f to represent a set of paths that can deliver flow f. A $Strategy$ $s = [p^{f_1}_{i_1}, p^{f_2}_{i_2}, \ldots, p^{f_{|F|}}_{i_{|F|}}]$ is a collection of paths, in which $p^{f_k}_{i_k}$, the i_kth path in p^{f_k}, is currently delivering flow f_k.

Given a strategy s, a link l_j's $Link\ State\ LS_j(s)$ is a triple $(C_j,\ N_j,\ S_j)$. A path p's *Path State* $PS_p(s)$ is represented by the *Link State*, which has the smallest *Fair Share* along that path. The $System\ State\ SysS(s)$ is the *Link State* with the smallest *Fair Share* in the network. A $Flow\ State\ FS_f(s)$ is the corresponding path state, *i.e.*, $FS_f(s) = PS_p(s)$, flow f is running on path p.

Notation s_{-k} refers to the strategy s without $p^{f_k}_{i_k}$, $i.e.$ $[p^{f_1}_{i_1}, \ldots, p^{f_{k-1}}_{i_{k-1}}, p^{f_{k+1}}_{i_{k+1}}, \ldots, p^{f_{|F|}}_{i_{|F|}}]$. $(s_{-k}, p^{f_k}_{i_{k'}})$ refers to the strategy $[p^{f_1}_{i_1}, \ldots, p^{f_{k-1}}_{i_{k-1}}, p^{f_k}_{i_{k'}}, p^{f_{k+1}}_{i_{k+1}}, \ldots, p^{f_{|F|}}_{i_{|F|}}]$. Flow f_k is $locally\ optimal$ in strategy s if

$$FS_{f_k}(s).S \geq FS_{f_k}(s_{-k}, p^{f_k}_{i_{k'}}).S \tag{B1}$$

for all $p^{f_k}_{i_{k'}} \in p^{f_k}$. A $Nash\ equilibrium$ is a state where all flows are locally optimal. A strategy s^* is $global\ optimal$ if for any strategy s, $SysS(s^*).S \geq SysS(s).S$.

Theorem 2. If there is no synchronized flow scheduling, Algorithm *Selfish Path Selection* increases the minimum *Fair Share* in every step and converges to a Nash equilibrium in finite steps. The global optimal strategy is also a Nash equilibrium strategy.

A strategy s's *Strategy Vector* $SV(s)$ is in the form of $[v_0(s),\ v_1(s),\ v_2(s),\ldots]$, where $v_k(s)$ is the number of links whose *Fair Share* is in $[k\delta,\ (k+1)\delta)$. The δ is the positive parameter in the selfish path selection algorithm. As a result, $\sum_k v_k(s)$ is the total number of links in the network. A small δ groups the links in a fine granularity and increases the minimum *Fair Share*. A large δ improves the convergence speed. Suppose s and s' are two strategies, $SV(s) = [v_0(s),\ v_1(s),\ v_2(s),\ldots]$ and $SV(s') = [v_0(s'),$ $v_1(s'), v_2(s'), \ldots]$. We define $s = s'$ if $v_k(s) = v_k(s')$ for all $k \geq 0$. $s < s'$ if there exists some j such that $v_j(s) < v_j(s')$ and $\forall k < j, v_k(s) \leq v_k(s')$. It is easy to show that given three strategies s, s' and s'', if $s \leq s'$ and $s' \leq s''$, then $s \leq s''$.

Given a specific network, traffic pattern and δ, there are only finite number of *Strategy Vectors*. According to the definition of " $=$ " and " $<$ ", we can find at least one strategy $\widetilde{s}$ that is the $smallest$, *i.e.*, for any strategy s, $\widetilde{s} \leq s$. This $\widetilde{s}$ has the largest minimum *Fair Share* or has the least number of links with the minimum *Fair Share* and thus, is the global optimal.

If one flow f selfishly changes its path and makes the strategy change from s to s', this action decreases the number of links with small *Fair Shares* and increases the number of links with larger *Fair Shares*. In other words, $s' < s$. This indicates asynchronous and selfish path selection actually increase global minimum *Fair Share* in every step until all flows reach their locally optimal state. Since the number of *Strategy Vectors* is finite, the steps to converge to a Nash equilibrium is finite. Because $\widetilde{s}$ is the smallest strategy, no flow can have a further movement to decrease $\widetilde{s}$, $i.e$ every flow is in its locally optimal state. As a result, this global optimal state $\widetilde{s}$ is also a Nash equilibrium.

References

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks," in *Proceedings of the 7th ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, Apr. 2010.

[2] "Amazon elastic compute cloud," http://aws.amazon.com/ec2.

[3] "Microsoft Windows Azure," http://www.microsoft.com/windowsazure.

[4] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *IMC '09: Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. New York, NY, USA: ACM, 2009, pp. 202–208.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, 2008.

[6] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 51–62, 2009.

[7] C. Hopps, "Analysis of an Equal-Cost Multi-Path Algorithm," RFC 2992, 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2992.txt

[8] "Deter Lab," http://www.isi.deterlab.net/.

[9] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "PortLand: a scalable fault-tolerant layer 2 data center network fabric," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 39–50, 2009.

[10] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th annual conference on Internet measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 267–280. [Online]. Available: http://doi.acm.org/10.1145/1879141.1879175

[11] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, "Design, implementation and evaluation of congestion control for multipath tcp," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 8–8. [Online]. Available: http://portal.acm.org/citation.cfm?id=1972457.1972468

[12] S. Kandula, D. Katabi, B. Davie, and A. Charny, "Walking the tightrope: Responsive yet stable traffic engineering," in *In Proc. ACM SIGCOMM*, 2005.

[13] X. Yang, D. Clark, and A. W. Berger, "NIRA: A New Inter-Domain Routing Architecture," *IEEE/ACM TRANSACTIONS ON NETWORKING*, vol. 15, 2007.

[14] "IP alias limitation in linux kernel 2.6," http://lxr.free-electrons.com/source/net/core/dev.c#L935.

[15] "IP alias limitation in windows NT 4.0," http://support.microsoft.com/kb/149426.

[16] T. Greene, "Researchers show off advanced network control technology," http://www.networkworld.com/news/2008/102908-openflow.html.

[17] A. Khanna and J. Zinky, "The revised ARPANET routing metric," in *Symposium proceedings on Communications architectures & protocols*, ser. SIGCOMM '89. New York, NY, USA: ACM, 1989, pp. 45–56. [Online]. Available: http://doi.acm.org/10.1145/75246.75252

[18] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, 2008.

[19] "TCPTrack," http://www.rhythm.cx/~steve/devel/tcptrack/.

[20] "OpenFlow switch specification, version 1.0.0," http://www.openflowswitch.org/documents/openflow-spec-v1.0.0.pdf.

[21] S. Sinha, S. Kandula, and D. Katabi, "Harnessing TCP's Burstiness using Flowlet Switching," in *3rd ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, San Diego, CA, November 2004.

[22] "ns-2 implementation of mptcp," http://www.jp.nishida.org/mptcp/.

[23] J.-Y. Boudec, "Rate adaptation, congestion control and fairness: A tutorial," 2000.

[24] C. Busch and M. Magdon-Ismail, "Atomic routing games on maximum congestion," *Theor. Comput. Sci.*, vol. 410, no. 36, pp. 3337–3347, 2009.