OpenFlow Based Load Balancing

Hardeep Uppal and Dane Brandon
University of Washington
CSE561: Networking
Project Report

Abstract: In today's high-traffic internet, it is often desirable to have multiple servers representing a single logical destination server to share load. A typical configuration consists of multiple servers behind a load-balancer which would determine which server would service a client's request. Such hardware is expensive, has a rigid policy set, and is a single point of failure. In this paper we implement and evaluate an alternative load-balancing architecture using an OpenFlow switch connected to a NOX controller, which gains flexibility in policy, costs less, and has the potential to be more robust to failure with future generations of switches

1 Introduction

There are many scenarios in today's increasingly cloud-service based internet where a client sends a request to a URL, or logical server, and receives a response from one of potentially many servers acting as the logical server at the address. One example would be a Google web-server: after a client resolves the IP address from the URL, the request is sent to a server farm. Since Google receives billions of requests a day, it has many servers which may all serve the same data. When the client's request arrives at the data center, a specialized load-balancer looks at the request and determines, based on some predetermined policy, which server will handle the request.

1.1 Motivation

Load-balancers are expensive (prices are easily upwards of $50k), highly specialized machines that typically "know" things about the servers they are forwarding to. They may route based on the servers' current loads, location of content relative to the request, or some naïve policy such as round-robin. Since load-balancers are not commodity hardware and run custom software, the policies are rigid in their offered choices. Special administrators are required and it is not possible to implement arbitrary policies. Since policy implementer and switch are coupled we are reduced to a single point of failure.

We implement a load-balancer architecture with an OpenFlow switch and a commodity server to control it which costs an order of magnitude less than a commercial load-balancer and provides the flexibility of writing modules for the controller that allow arbitrary policies to be applied.

We further hypothesize that with the next-generation of OpenFlow switches being capable of connection to multiple controllers, it will be possible to make the system robust to failure by allowing any server behind the switch able to act as the controller.

2 Background

2.1 Load Balancing

Load balancing in computer networks is a technique used to spread workload across multiple network links or computers. This helps improve performance by optimally using available resources and helps in minimizing latency and response time and maximizing throughput. Load balancing is achieved by using multiple resources i.e. multiple servers that are able to fulfill a request or by having multiple paths to a

resource. Having multiple servers with load spread out evenly across them avoids congestion at a server and improves response time of a request.

Load-balancing can be achieved by using dedicated hardware devices like load-balancers or by having smart DNS servers. A DNS server can redirect traffic from datacenters with heavy load or redirect requests made by clients to a datacenter that is least network hops away from the clients. Many datacenters use expensive load-balancer hardware devices that help distribute network traffic across multiple machines to avoid congestion on a server.

A DNS server resolves a hostname to a single IP address where the client sends its request. To the outside world there is one logical address that a hostname resolves to. This IP address is not associated with a single machine, but represents the kind of service a client is requesting. The DNS could resolve a hostname to a load-balancer inside a datacenter. But this might be avoided for security reasons and to avoid attacks on the device. When a client's request arrives to the load-balancer, the request gets redirected depending on policy.

2.2 OpenFlow Switch

OpenFlow switches are like a standard hardware switch with a flow table performing packet lookup and forwarding. The difference lies in how flow rules are inserted and updated inside the switch's flow table. A standard switch can have static rules inserted into the switch or can be a learning switch where the switch inserts rules into its flow table as it learns on which interface (switch port) a machine is. The OpenFlow switch on the other hand uses an external controller called NOX to add rules into its flow table.

2.2.1 NOX Controller

NOX is an external controller that is responsible for adding or removing new routing rules into the OpenFlow switch's flow table. The OpenFlow switch is connected to the NOX controller and communicates over a secure channel using the OpenFlow protocol. The current design of OpenFlow only allows one NOX controller per switch. The NOX controller decides how packets of a new flow should be handled by the switch. When new flows arrive at the switch, the packet gets redirected to the NOX controller which then decides whether the switch should drop the packet or forward it to a machine connected to the switch. The NOX controller can also delete or modify existing flow entries in the switch.

The NOX controller can execute modules that describe how a new flow should be handled. This provides us an interface to write C++ modules that dynamically add or delete routing rules into the switch and can use different policies for handling flows.

2.2.2 Flow Table

A flow table entry of an OpenFlow switch consists of a header fields, counters and actions. Each flow table entry stores Ethernet, IP and TCP/UDP header information. This information includes destination/source MAC and IP address and source/destination TCP/UDP port numbers. Each flow table entry also maintains a counter of number of packets, and bytes arrived per flow. A flow table entry can also have one or more action fields that describe how the switch will handle packets that match the flow entry. Some of the actions include sending the packet on all output ports, forwarding the packet on an output port of a particular machine and modifying packet headers (Ethernet, IP and TCP/UDP header). If a flow entry does not have any actions, then the switch drops all packets for the particular flow.

Each Flow entry also has an expiration time after which the flow entry is deleted from the flow table. This expiration time is based on the number of seconds a flow was idle and the total amount the time (in seconds) the flow entry has been in the flow table. The NOX controller can chose a flow entry to exist permanently in the flow table or can set timers which delete the flow entry when the timer expires.

3 Load-Balancer Design

Our load balancing architecture consists of an OpenFlow switch with a NOX controller and multiple server machines connected to the output ports of the switch. The OpenFlow switch uses one interface to connect to the internet. Each server has a static IP address and the NOX controller maintains a list of servers currently connected to the OpenFlow switch. Each server is running a web server emulator on a well known port.

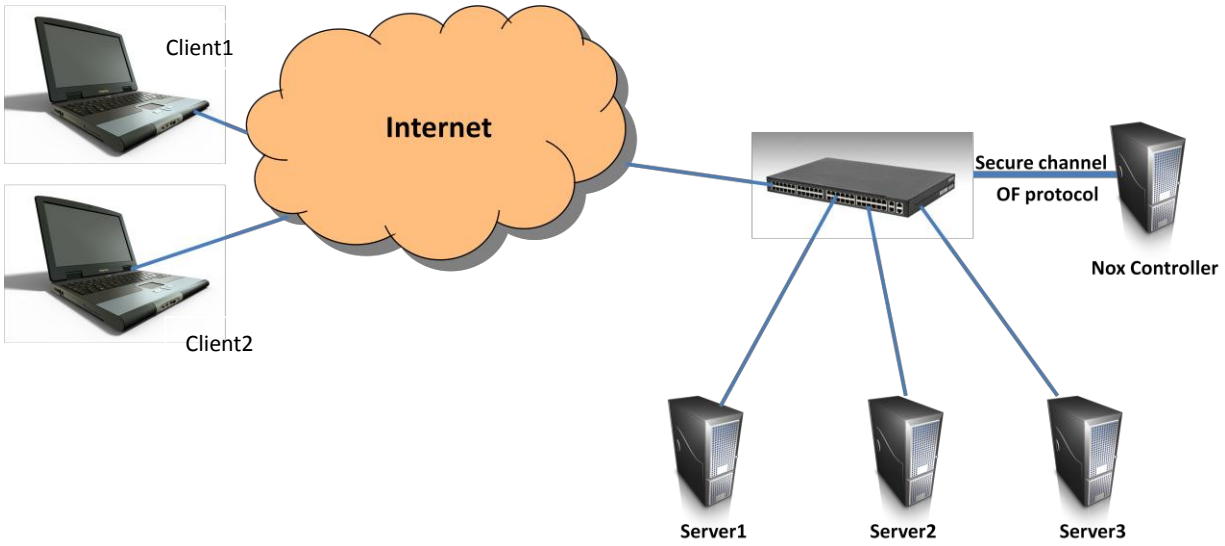

Figure 1. Load-balancer architecture using OpenFlow switch and NOX controller.

Each client resolves the hostname of a server to an IP address and sends its request to that IP address on the known port number. Figure 1 illustrates the design for our load-balancer. When a packet from a client arrives at the switch, the header information of the packet is compared with the flow table entries. If the packet's header information matches a flow entry, the counter for the number of packets and number of bytes is incremented, and the actions associated with the flow entry are performed on the packet. If no match is found, the switch forwards the packet to the NOX. The NOX then decides on how the packet for this flow should be handled by the switch. The NOX then inserts a new rule into the switch's flow table using the OpenFlow protocol. Figure 2 illustrates the flow diagram for packets arriving at the switch.

To achieve load-balancing features, we wrote C++ modules that are executed by the NOX controller. The NOX executes the *handle()* function declared in the module when a new flow arrives at the switch. This function defines the load balancing policies and adds new rules into the switch's flow table. Section 3.1 describes three load balancing policies that are implemented by the module.

Since all client requests are destined for the same IP address, the module executed by NOX adds rules for each flow that modify the destination MAC and IP address of the packet with a server's MAC and IP address. After the packet's header is modified, the switch forwards the packet to the output port of the server. When the servers send back a packet to the client, the module adds a flow entry that modifies the source IP address with the IP address of the hostname that the client sends its request to. Hence the client always receives packets from the same IP address. If the client/server connection is closed or connection stays idle for 10 seconds, then the idle timer expires causing the flow entry to be deleted from the flow table of the switch. This helps recycle flow entries.

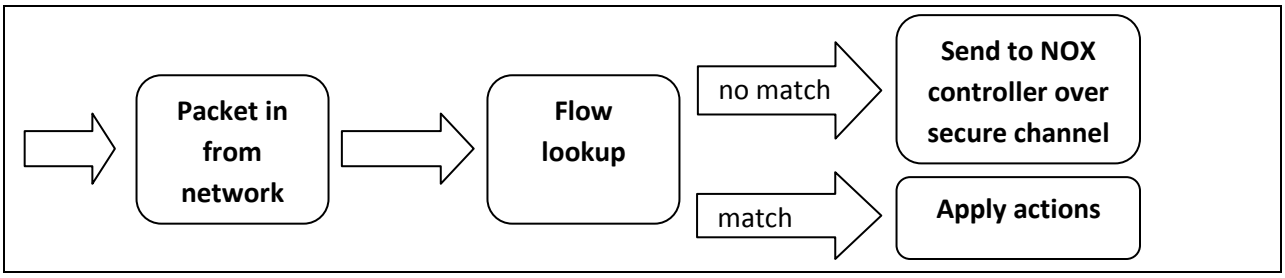

Figure 2. Flow diagram of functions performed on packets. [1]

3.1 Load-balancing policies

One can think of many possible scenarios as to what sort of policy would be implemented. We implement and benchmark three simple policies as proof of concept, but the modular nature of the software would allow any arbitrarily complex policy to be plugged in to the system.

3.1.1 Random

For each new flow forwarded to the NOX, the NOX randomly selects from a list of registered servers which server will handle the request.

3.1.2 Round Robin

For each new flow, the NOX rotates which server is the next server in line to service a request.

3.1.3 Load-Based

Servers wait for a NOX to register and then report their current load on some schedule similar to the Listener Pattern. What defines load at the servers is arbitrary but we used the number of pending requests in the server's queue as the load. The NOX listens in a separate thread on a UDP socket for heartbeats with reported loads from the servers and maintains an array with the current loads of all servers. When a request for a new flow is received, it chooses the server with the current lowest load and increments that server's current load. This prevents a flood of flows all being routed to the same server until that server reports a new load. This also breaks ties by turning them into a round robin until the servers report their real load via heartbeat.

4 Evaluation

To demonstrate our design we evaluate different pieces of our load-balancer including the hardware switch, NOX controller, and the overall performance of the system.

4.1 Micro-benchmarks

4.1.1 OpenFlow Switch

To evaluate the performance of the hardware switch we used a single UDP echo client server pair to measure per-packet latency with a rule for the flow already inserted into the flow-table.

Figure 3 shows latencies for 1000 packets sent with a stop-and-wait applied to prevent buffering. Packets were sent once where the switch modifies the packet header and once with simple forwarding. The same experiment was executed on a standard switch for reference. The OpenFlow switch has similar latency to a normal switch when headers are not modified. The average latency for a rule with packet modification is 933µs vs. 408µs without. We suspect that the reason for slow rewrites is that the switch is pushing the rewrite up into a software layer in the switch as opposed to it happening in optimized hardware. We also found that the flow table can hold a maximum of 2053 flow entries and causes rules to be pushed out of the table.

Figure 4 shows that the OpenFlow switch utilizes the switch at full capacity like the regular switch when packet headers are not modified. However, when packet headers are modified throughput is reduced by two orders of magnitude. This is caused by packets buffering while waiting for headers to be rewritten and packets being dropped when the buffer is full.

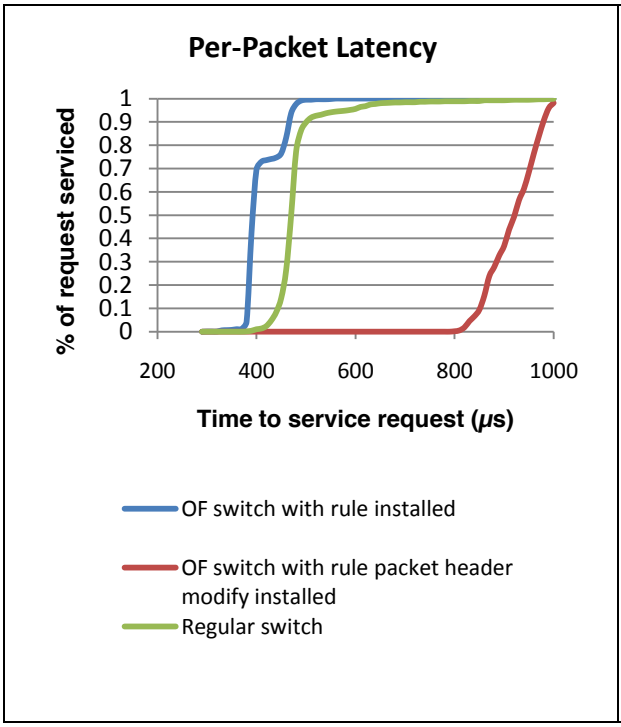

Figure 3. Per-packet latency with OpenFlow switch and standard switch.

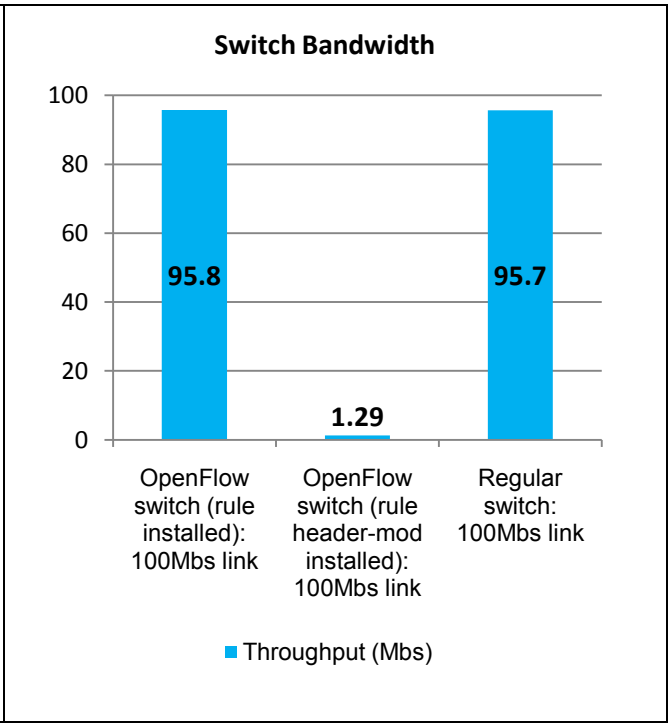

Figure 4. Throughput with OpenFlow switch and standard switch.

This is cripplingly slow for a real-world application and shows that with current hardware, rewriting headers is not practical. It would be possible to use a modified ARP protocol to avoid rewrites at the switch and instead modify the Linux kernel to accept packets addressed to a made up MAC address.

4.1.2 NOX Controller

To find the latency of sending packets to the NOX and rules being added, we generate single packet flows by creating new socket connections for each packet sent. Figure 5 shows that the NOX controller average latency is ~7ms. Figure 6 illustrates that ~100 flows/s can be added to the switch. A real server would be servicing on the order of thousands of requests per second and again we see a limitation of the hardware for this application.

4.2 Web-Traffic Simulation

To emulate web-traffic we ran three servers behind the switch. For the load-based policy, the servers report their load to the NOX every 100ms. Web traffic follows a Zipf-like distribution per Breslau et al. [2] so we use a Zipf distribution with alpha value of .625 to determine how many requests are sent per client with minimum 1 and mean 15 requests sent. This represents multiple embedded objects per webpage requested. Response size in bytes is modeled by a Zipf distribution with alpha value of .95 as a 100B min. and 1500B mean. All client requests are 100B. To emulate burstiness of request arrivals, we used a Poisson distribution where 100 clients sent requests with a mean of 20 flows per second. More flows/s incurred significant packet loss.

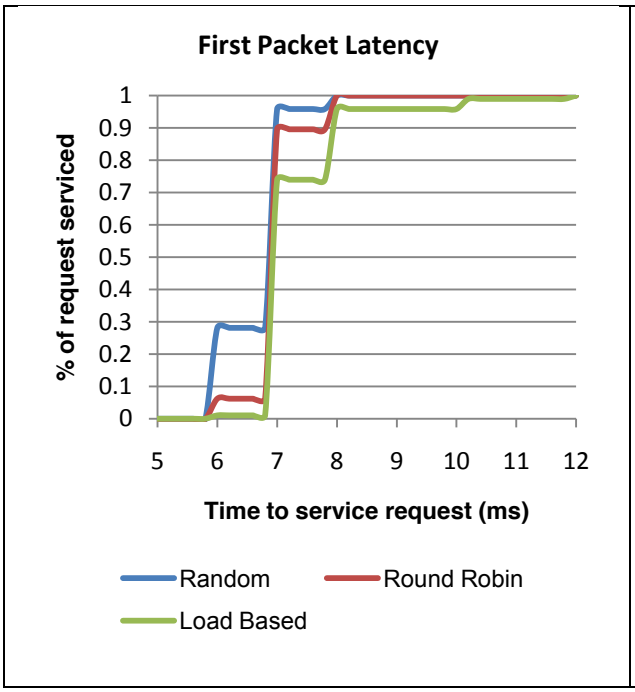

Figure 5. Latency of first packet of flow. The first packet of a flow is redirected to the NOX and must incur extra latency.

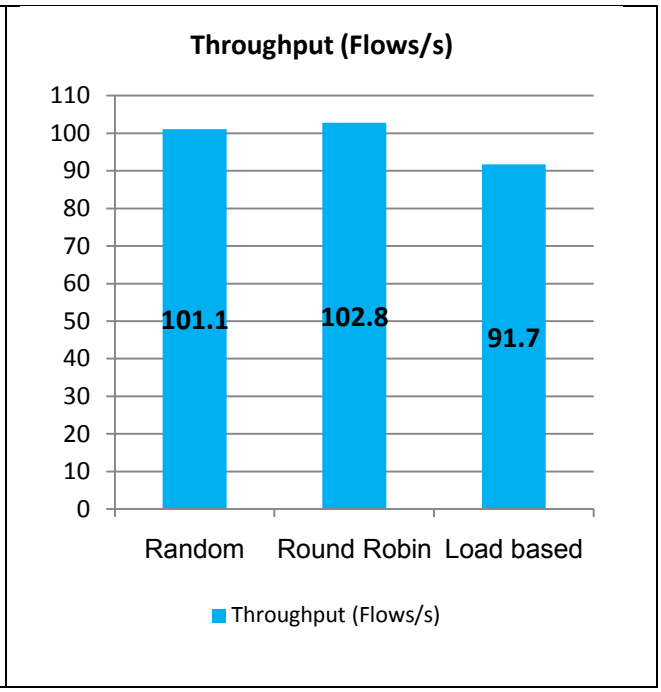

Figure 6. Flows per second per load-balancer policy that can be pushed to the NOX.

Figures 7 and 8 show packet latencies for 10ms and 20ms server processing times per packet. At 10ms process time, the policies all perform about as well with mean latencies of RR=149, random=134ms, and load-based=153.ms. When we slow the servers down by increasing process time to 20ms, the bursty nature of the traffic makes the load based policy perform better with mean latencies of random=696ms, RR=493ms, load=376ms.

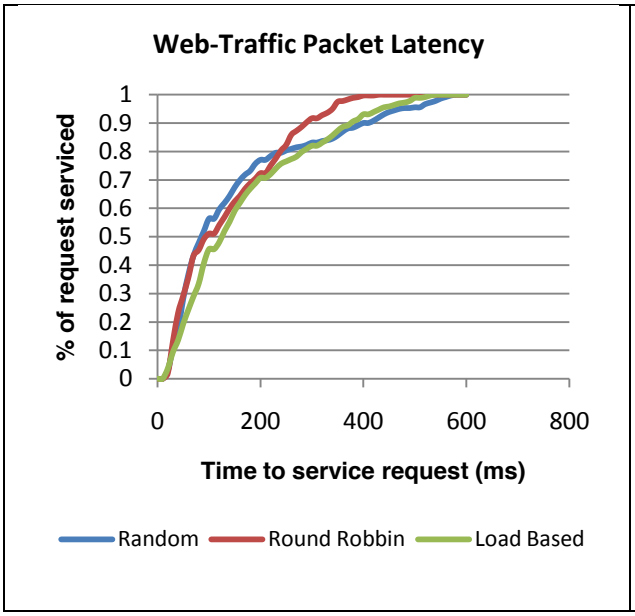

Figure 7. Emulated web-traffic with 10ms server processing time.

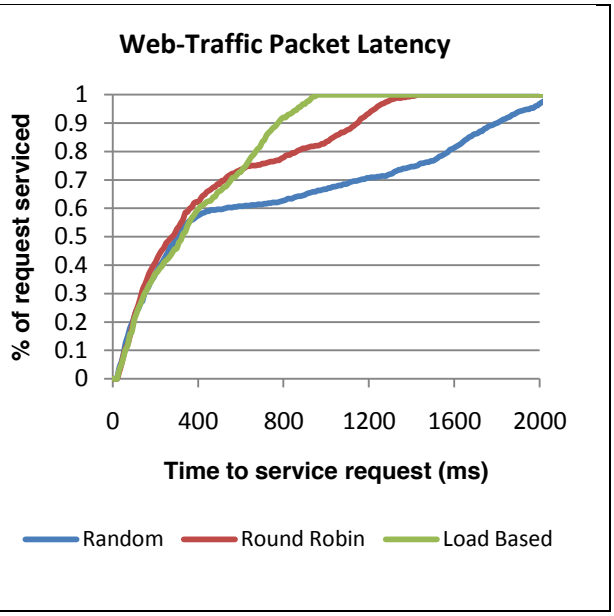

Figure 8. Emulated web-traffic with 20ms server processing time

There are many ways to run the experiments, but ultimately the system is limited by how fast the switch can rewrite the headers of the packets in transit. Being limited to pushing 20 new flow rules into the switch and running at ~1Mbs is a limitation that would prevent this from being deployed in a datacenter as datacenter traffic would imply a gigabit switch and each server servicing on the order of thousands of clients per second.

Despite the throughput of the switch with rule-rewriting bottlenecking performance, the experiment shows that it is indeed possible to implement a load-balancer with an OpenFlow switch and modular policy in software. This provides the openness and flexibility we set out to achieve.

We expect that the next generation of switches or perhaps even firmware updates to be much higher performance and possibly able to allow rule-rewriting at speeds that can keep up with the specified throughput of the switch. Our current switch (an HP 5412zl) is a research prototype and not all features of the OpenFlow specification have yet been implemented. We suspect that much of the OpenFlow functionality is not yet optimized in hardware.

5 Future Work

As described thus far, our implementation suffers as a single point of failure just like a commercial load-balancer. In both cases, if the computer running the policy fails, then all traffic connections will fail. We propose a novel solution in software which would make our implementation robust to failure of the server hosting the NOX.

The OpenFlow specification includes an optional feature that would allow multiple NOXs to make active connections to the switch. In the case then of the NOX failing, another machine could assume the role of the NOX and continue routing traffic. Naturally the system would need to detect the failure, have a mechanism to remember any state associated with the current policy, and all servers would have to agree on who the new NOX was. These requirements naturally lend themselves to the Paxos consensus algorithm in which policy and leader elections can be held and preserved with provable progress [3]. We have implemented Paxos in another research project and could add it to our server implementation at the controller/signaler layer. As long as at least half of the nodes in the cluster stay up, state will be preserved and traffic should continue to flow.

By adding the distributed control layer and fully decoupling the policy implementation from the switch, we would reduce the single point of failure to the switch itself.

6 Conclusion

We have shown that it is possible to get similar functionality to a commercial load-balancer using only commodity hardware switches. The OpenFlow switch provides the flexibility to implement arbitrary policy in software and decouple policy from the switch itself. Since the policy is decoupled from the switch, we could avoid making the machine implementing the policy a single point of failure and creating a more robust system. Unfortunately the current first generation OpenFlow switches cannot rewrite packet headers fast enough to make this implementation deployable in a datacenter environment. Hopefully as hardware improves, the switches will be able to gain the benefits we see possible.

References

[1] OpenFlow Switch Specification. Version 0.8.9 (Wire Protocol 0x97). Current maintainer: Brandon Heller (brandonh@stanford.edu). December 2, 2008.
[2] Web caching and Zipf-like distributions: evidence and implications. Breslau, L. Pei Cao Li Fan Phillips, G. Shenker, S. Xerox Palo Alto Res. Center, CA. INFOCOM 1999.
[3] Paxos Made Simple. Leslie Lamport.