OpenFlow-Based Server Load Balancing Gone Wild

Richard Wang, Dana Butnariu, and Jennifer Rexford
Princeton University; Princeton, NJ

Abstract

Today's data centers host online services on multiple servers, with a front-end load balancer directing each client request to a particular replica. Dedicated load balancers are expensive and quickly become a single point of failure and congestion. The OpenFlow standard enables an alternative approach where the commodity network switches divide traffic over the server replicas, based on packet-handling rules installed by a separate controller. However, the simple approach of installing a separate rule for each client connection (or "microflow") leads to a huge number of rules in the switches and a heavy load on the controller. We argue that the controller should exploit switch support for *wildcard* rules for a more scalable solution that directs large aggregates of client traffic to server replicas. We present algorithms that compute concise wildcard rules that achieve a target distribution of the traffic, and automatically adjust to changes in load-balancing policies without disrupting existing connections. We implement these algorithms on top of the NOX OpenFlow controller, evaluate their effectiveness, and propose several avenues for further research.

1 Introduction

Online services—such as search engines, Web sites, and social networks—are often replicated on multiple servers for greater capacity and better reliability. Within a single data center or enterprise, a front-end load balancer [2, 4] typically directs each client request to a particular replica. A dedicated load balancer using consistent hashing is a popular solution today, but it suffers from being an expensive additional piece of hardware and has limited customizability. Our load-balancing solution avoids the cost and complexity of separate load-balancer devices, and allows flexibility of network topology while working with unmodified server replicas. Our solution scales naturally as the number of switches and replicas grows, while directing client requests at line rate.

The emerging OpenFlow [8] platform enables switches to forward traffic in the high-speed *data plane* based on rules installed by a *control plane* program running on a separate *controller*. For example, the Plug-n-Serve [6] system (now called Aster*x [1]) uses OpenFlow to *reactively* assign client requests to replicas based on the current network and server load. Plug-n-Serve intercepts the first packet of each client request and installs an individual forwarding rule that handles the remaining packets of the connection. Despite offering great flexibility in adapting to current load conditions, this reactive solution has scalability limitations, due to the overhead and delay in involving the relatively slow controller in every client connection, in addition to many rules installed at each switch.

Our scalable in-network load balancer *proactively* installs *wildcard* rules in the switches to direct requests for large groups of clients without involving the controller. Redistributing the load is a simple as installing new rules. The use of wildcard rules raises two main problems: (i) generating an efficient set of rules for a target distribution of load and (ii) ensuring that packets in the same TCP connection reach the same server across changes in the rules. The load balancer is a centralized controller program [9] so we can determine the globally optimal wildcard rules. Our solutions achieve the speed of switch forwarding, flexibility in redistributing load, and customizable reactions to load changes of an in-network load balancing solution, with no modification to clients or servers.

In the next section, we present our load-balancing architecture, including the "partitioning" algorithm for generating wildcard rules and our "transitioning" algorithm for changing from one set of rules to another. We also present a preliminary evaluation of our prototype, built using OpenVswitch, NOX [5], and MiniNet [7]. Then, Section 3 discusses our ongoing work on extensions to support a non-uniform distribution of clients and a network of multiple switches. These extensions build on our core ideas to form a complete in-network load balancing solution with better flexibility in redistributing load, customizing reactions to load changes, and lower cost compared to existing solutions. The paper concludes in Section 4.

2 Into the Wild: Core Ideas

The data center consists of multiple replica servers offering the same service, and a network of switches connecting to clients, as shown in Figure 1. Each server replica R_j has a unique IP address and an integer weight α_j that determines the share of requests the replica should handle; for example, R_2 should receive 50% (i.e., $4/8$) of

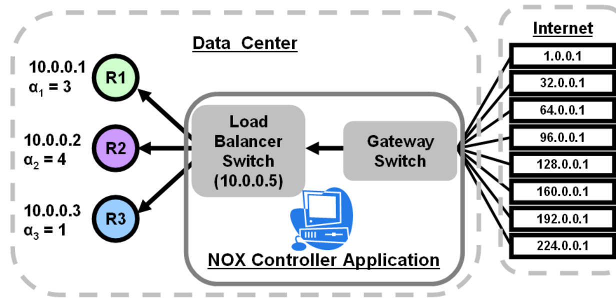

Figure 1: Basic model from load balancer switch's view

the requests. Clients access the service through a single public IP address, reachable via a gateway switch. The load-balancer switch rewrites the destination IP address of each incoming client packet to the address of the assigned replica. In this section, we first describe the OpenFlow features used in our solution. Next, we describe how our *partitioning* algorithm generates wildcard rules that balance load over the replicas. Then, we explain how our *transitioning* algorithm moves from one set of wildcard rules to another, without disrupting ongoing connections. Finally, we present an evaluation of our prototype system.

2.1 Relevant OpenFlow Features

OpenFlow defines an API for a controller program to interact with the underlying switches. The controller can install rules that *match* on certain packet-header fields (e.g., MAC addresses, IP addresses, and TCP/UDP ports) and perform *actions* (e.g., forward, drop, rewrite, or "send to the controller") on the matching packets. A *microflow* rule matches on all fields, whereas a *wildcard* rule can have "don't care" bits in some fields. A switch can typically support many more microflow than wildcard rules, because wildcard rules often rely on expensive TCAM memory, while microflow rules can leverage more abundant SRAM. Rules can be installed with a timeout that triggers the switch to delete the rule after a fixed time interval (a *hard* timeout) or a specified period of inactivity (a *soft* timeout). In addition, the switch counts the number of bytes and packets matching each rule, and the controller can poll these counter values.

In our load-balancing solution, the switch performs an "action" of (i) rewriting the server IP address and (ii) forwarding the packet to the output port associated with the chosen replica. We use wildcard rules to direct incoming client requests based on the client IP addresses, relying on microflow rules only during transitions from one set of wildcard rules to another; soft timeouts allow these microflow rules to "self destruct" after a client connection completes. We use the counters to measure load for each wildcard rule to identify imbalances in the traffic load, and drive changes to the rules to rebalance the traffic.

OpenFlow has a few limitations that constrain our solution. OpenFlow does not currently support *hash-based routing* [10] as a way to spread traffic over multiple paths. Instead, we rely on wildcard rules that match on the client IP addresses. Ideally, we would like to divide client traffic based on the *low-order* bits of the client IP addresses, since these bits have greater entropy than the high-order bits. However, today's OpenFlow switches only support "don't care" bits on the lower-order bits, limiting us to IP prefix rules. In addition, OpenFlow does not support matching on TCP flags (e.g., SYN, FIN, and RST) that would help us differentiate between new and ongoing connections—important when our system transitions from one set of wildcard rules to another. Instead, we propose alternative ways to ensure that successive packets of the same connection reach the same server replica.

2.2 Partitioning the Client Traffic

The partitioning algorithm must divide client traffic in proportion to the load-balancing weights, while relying only on features available in the OpenFlow switches. To ensure successive packets from the same TCP connection are forwarded to the same replica, we install rules matching on client IP addresses. We initially assume that traffic volume is uniform across client IP addresses (an assumption we relax later in Section 3.1), so our goal is to generate a small set of wildcard rules that divide the entire client IP address space[1]. In addition, changes in the target distribution of load require new wildcard rules, while still attempting to minimize the number of changes.

2.2.1 Minimizing the Number of Wildcard Rules

A *binary tree* is a natural way to represent IP prefixes, as shown in Figure 2(a). Each node corresponds to an IP prefix, where nodes closer to the leaves represent longer prefixes. If the sum of the $\{\alpha_j\}$ is a power of two, the algorithm can generate a tree where the number of leaf nodes is the same as the sum (e.g., the eight leaf nodes in Figure 2(a)). Each R_j is associated with α_j leaf nodes; for example, replica R_2 is associated with four leaves. However, the $\{\alpha_j\}$ may not sum to a power of 2 in practice. Instead, we determine the closest power of 2, and renormalize the weights accordingly. The resulting weights closely approximate the target distribution, and enable a simple and efficient partitioning of the IP address space.

Creating a wildcard rule for each leaf node would lead to a large number of rules. To reduce the number of rules, the algorithm can *aggregate* sibling nodes associated with the same server replica; in Figure 2(a), a single wildcard rule 10* could represent the two leaf nodes 100* and 101*

[1] The IP addresses above 224.0.0.0 are not used for unicast traffic. For ease of explanation, the rest of this section assumes that *all* IP addresses are used; in practice, our implementation renormalizes the $\{\alpha_j\}$ values and assigns only the lower 7/8 of IP address space to server replicas.

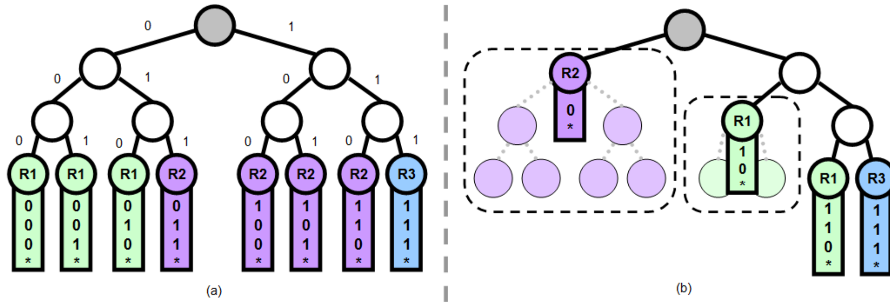

Figure 2: $\alpha_1 = 3$, $\alpha_2 = 4$, and $\alpha_3 = 1$. Assuming uniform distribution of traffic: (a) wildcard rules assigning leaf nodes to a perfect binary tree achieving target distribution. (b) fewer wildcard rules.

associated with R_2. Similarly, the rule 00* could represent the two leaf nodes 000* and 001* associated with R_1, reducing the number of wildcard rules from 8 to 6. However, the assignment of leaf nodes in Figure 2(a) does not lead to the minimum number of rules. Instead, the alternate assignment in Figure 2(b) achieves the minimum of four rules (i.e., 0*, 10*, 110*, and 111*).

The binary representation of the weights indicates how to best assign leaf nodes to replicas. The number of bits set to 1 in the binary representation of α_j is the minimum number of wildcard rules for replica R_j, where each 1-bit i represents a merging of 2^i leaves. R_1 has $\alpha_1 = 3$ (i.e., 011 in binary), requiring one rule with two leaves and another with one leaf. Our algorithm assigns leaf nodes to replicas ordered by the highest bit set to 1 among all α values, to prevent fragmentation of the address space. In Figure 2(b), R_2 is first assigned a set of four leaves, represented by 0*. Once all leaf nodes are assigned, we have a complete and minimal set of wildcard rules.

2.2.2 Minimizing Churn During Re-Partitioning

The weights $\{\alpha_j\}$ may change over time to take replicas down for maintenance, save energy, or to alleviate congestion. Simply regenerating wildcard rules from scratch could change the replica selection for a large number of client IP addresses, increasing the overhead of transitioning to the new rules. Instead, the controller tries to minimize the fraction of the IP address space that changes from one replica to another. If the number of leaf nodes for a particular replica remains unchanged, the rule(s) for that replica may not need to change. In Figure 2(b), if replica R_3 is taken down and its load shifted to R_1 (i.e., α_3 decreases to 0, and α_1 increases from 3 to 4), the rule for R_2 does not need to change. In this case, only the IP addresses in 111* would need to transition to a new replica, resulting in just two rules (0* for R_2 and 1* for R_1).

To minimize the number of rules, while making a "best effort" to reuse the previously-installed rules, the algorithm creates a new binary tree for the updated $\{\alpha_j\}$ and pre-allocates leaf nodes to the potentially re-usable wildcard rules. Re-usable rules are rules where the ith highest bit is set to 1 for both the new and old α_j. Even if the total number of bits to represent the old α_j and new α_j are different, the ith highest bit corresponds to wildcard rules with the same number of wildcards. However, smaller pre-allocated groups of leaf nodes could prevent finding a set of aggregatable leaf nodes for a larger group; when this happens, our algorithm allocates leaf nodes for the larger group to minimize the total number of rules, rather than reusing the existing rules.

2.3 Transitioning With Connection Affinity

The controller cannot abruptly change the rules installed on the switch without disrupting ongoing TCP connections; instead, existing connections should complete at the original replica. Fortunately, we can distinguish between new and existing connections because the TCP SYN flag is set in the first packet of a new connection. While OpenFlow switches cannot match on TCP flags, the controller can check the SYN bit in a packet, and install new rules accordingly. Identifying the *end* of a connection is trickier. Even a FIN or RST flag does not clearly indicate the end of a connection, since retransmitted packets may arrive *after* the FIN; in addition, clients that fail spontaneously never send a FIN or RST. Instead, we infer a connection has ended after (say) 60 seconds of inactivity.

We have two algorithms for transitioning from one replica to another. The first solution directs some packets to the controller, in exchange for a faster transition; the second solution allows the switch to handle *all* packets, at the expense of a slower transition. To reduce the number of extra rules in the switches, we can limit the fraction of address space in transition at the same time. For example, transitioning 111* from R_3 to R_1 could proceed in stages, where first 1110* is transitioned, and then 1111*.

2.3.1 Transitioning Quickly With Microflow Rules

To move traffic from one replica to another, the controller temporarily intervenes to install a dedicated *microflow*

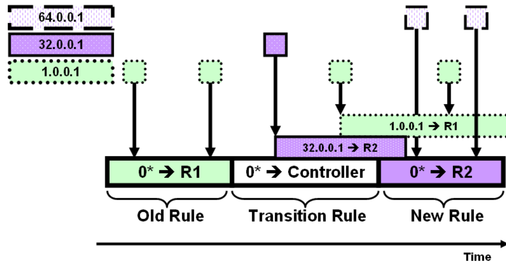

Figure 3: Transitions for wildcard rule changes: Square boxes represent packets sent by client on left. Traffic during transitions are assigned microflow rules.

rule for each connection in the affected region of client IP addresses. For example, suppose the client traffic matching 0* should shift from replica R_1 to R_2 as in Figure 3. The controller needs to see the *next* packet of each connection in 0*, to decide whether to direct the rest of that connection to the new replica R_2 (for a SYN) or the old replica R_1 (for a non-SYN). As such, the controller installs a rule directing all 0* traffic to the controller for further inspection; upon receiving a packet, the controller installs a high-priority microflow rule for the remaining packets of that connection[2]. In Figure 3, the controller receives a SYN packet from client 32.0.0.1 during the transition process, and directs that traffic to R_2; however, the controller receives a non-SYN packet for the ongoing connection from client 1.0.0.1 and directs that traffic to R_1.

Our algorithm installs a microflow rule with a 60-second soft timeout to direct specific connections to their appropriate replicas during these transitions. The controller does not need to intervene in the transition process for long. In fact, any ongoing connection should have at least one packet before sixty seconds have elapsed, at which time the controller can modify the 0* rule to direct all future traffic to the new replica R_2; in the example in Figure 3, the new flow from client 64.0.0.1 is directed to R_2 by the new wildcard rule.

2.3.2 Transitioning With No Packets to Controller

The algorithm in the previous subsection transitions quickly to the new replica, at the expense of sending some packets to the controller. In our second approach, *all* packets are handled directly by the switches. In Figure 3, the controller could instead divide the address space for 0* into several smaller pieces, each represented by a high-priority wildcard rule (e.g., 000*, 001*, 010*, and 011*) directing traffic to the old replica R_1. If one of these rules has no traffic for some configurable timeout of sixty seconds, no ongoing flows remain and that entire group of client addresses can safely transition to replica R_2. A soft timeout ensures the high-priority wildcard rule is deleted from the switch after 60 seconds of inactivity. In addition, the controller installs a single lower-priority rule directing 0* to the new replica R_2, that handles client requests that have completed their transition.

While this solution avoids sending data packets to the controller, the transition proceeds more slowly because some *new* flows are directed to the old replica R1. For example, a new connection matching 000* that starts during the transition period will be directed to R_1, and would extend the time the 000* rule must remain in the switch. By installing a larger number of temporary rules, the controller can make the transition proceed more quickly. As the switch deletes some rules, the controller can install additional rules that further subdivide the remaining address space. For example, if the switch deletes the 000* after the soft timeout expires, the controller can replace the 001* rule with two finer-grain rules 0010* and 0011*.

[2] This transitioning technique is vulnerable to a rare corner case, where a retransmitted SYN packet arrives *after* a connection is established to R_1. If this retransmitted SYN arrives just after the transition begins, our algorithm would (wrongly) direct the rest of the flow to R_2. This scenario is extremely unlikely, and would only happen very early in the lifetime of a connection. The client can simply retry the request.

2.4 Implementation and Evaluation

We have built a prototype using OpenVswitch (a software OpenFlow switch) and NOX (an OpenFlow controller platform), running in Mininet. Our prototype runs the partitioning algorithm from Section 2.2 and our transitioning algorithm from Section 2.3.1. We use Mininet to build the topology in Figure 1 with a set of 3 replica servers, 2 switches, and a number of clients. The replica servers run Mongoose [3] web servers. Our NOX application installs rules in the two switches, using one as a gateway to (de)multiplex the client traffic and the other to split traffic over the replicas. Our performance evaluation illustrates how our system adapts to changes in the load-balancing policy, as well as the overhead for transitions.

Adapting to new load-balancing weights: Our three replica servers host the same 16MB file, chosen for more substantial throughput measurements. For this experiment, we have 36 clients with randomly-chosen IP addresses in the range of valid unicast addresses. Each client issues *wget* requests for the file; after downloading the file, a client randomly waits between 0 and 10 seconds before issuing a new request. We assign $\alpha_1 = 3$, $\alpha_2 = 4$, and $\alpha_3 = 1$, as in Figure 2. At time 75 seconds, we change α_2 from 4 to 0, as if R_2 were going down for maintenance. Figure 4 plots the throughput of the three servers over time. As the clients start sending traffic, the throughput ramps up, with R_2 serving the most traffic. The division of load is relatively close to the 3:4:1 target split, though the relatively small number of clients and natural variations in the workload lead to some understandable deviations. The workload variations also lead to fluctuations in replica throughput over time. After 75 seconds (indicated

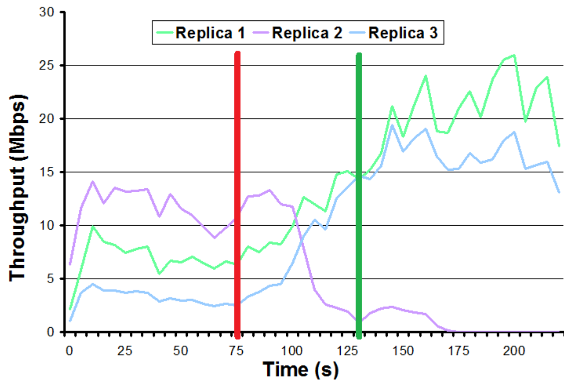

Figure 4: Throughput of experiment demonstrating ability to adapt to changes in division of load. Vertical lines indicate start and end of transitions.

by the first vertical bar), the load on server R_2 starts to decrease, since all new connections go to replicas R_1 and R_3. Sixty seconds later (indicated by the second vertical bar), the controller installs the new wildcard rules. R_2's load eventually drops to 0 as the last few ongoing connections complete. Initially, there were 6 wildcard rules installed. 4 of these were aggregated into a single wildcard rule after reassigning load with only 3 requiring a transition, 2 of which were rules to R_2 which is unavoidable. The resulting experiment concluded with only 3 wildcard rules.

Overhead of transitions: To evaluate the overhead and delay on the controller during transitions, we have ten clients simultaneously download a 512MB file from two server replicas. We start with all traffic directed to R_1, and then (in the middle of the ten downloads) start a transition to replica R_2. The controller must install a microflow rule for each connection, to ensure they complete at the old replica R_1. In our experiments, we did not see any noticeable degradation in throughput during the transition period; any throughput variations were indistinguishable from background jitter. Across multiple experimental trials, the controller handled a total of 18 to 24 packets and installed 10 microflow rules. Because of the large file size and the small round-trip time, connections often had multiple packets in flight, sometimes allowing multiple packets to reach the controller before the microflow rule was installed. We expect fewer extra packets would reach the controller in realistic settings with a smaller per-connection throughput.

3 Wild Ideas: Ongoing Work

Our current prototype assumes a network with just two switches and uniform traffic across client IP addresses. In our ongoing work, we are extending our algorithms to handle non-uniform traffic and an arbitrary network topology. Our existing partitioning and transitioning algorithms are essential building blocks in our ongoing work.

3.1 Non-Uniform Client Traffic

Our partitioning algorithm for generating the wildcard rules assumed uniform client traffic across source IP addresses. Under non-uniform traffic, the wildcard rules may deviate from target division of traffic. Consider Figure 5, where the target distribution of load is 50%, 25%, and 25% for R_1, R_2, and R_3, respectively. Our partitioning algorithm would correctly generate the set of wildcard rules on the left of Figure 5. Unfortunately, if traffic is non-uniform as expressed below the leaf nodes, then the actual division of load would be an overwhelming 75% for R_1, and an underwhelming 12.5% for R_2 and R_3 each.

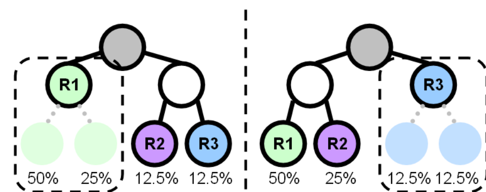

Figure 5: $\alpha_1 = 2$, $\alpha_2 = 1$, and $\alpha_3 = 1$. At the bottom are the non-uniform traffic measurements. Left: wildcard rules assuming uniform distribution of traffic. Right: wildcard rules adjusted for non-uniform traffic.

The set of wildcard rules that account for non-uniform traffic is on the right in Figure 5. To go from the rules on the left to the ones on the right, the algorithm must measure the traffic matching each rule using OpenFlow counters. Next, the algorithm should be able to identify severely overloaded and underloaded replicas and then identify the set of rules to shift. This may involve splitting a wildcard rule into several smaller ones to collect finer-grain measurements (e.g., replacing 0* with 00* and 01* to have separate counters for the two sets of clients). These finer-grain measurements make it easier to incrementally shift smaller groups of clients to avoid a large shift that results in an even less accurate division of load. The algorithm can repeat this process recursively to more closely approximate the desired distribution of traffic. After identifying which wildcard rules to shift, the transitioning algorithm can handle the necessary rule changes.

The result of these operations may not achieve the minimal set of wildcard rules. Ideally, the algorithm needs to strike a balance between minimizing the number of wildcard rules and dividing load accurately. Our initial partitioning algorithm emphasizes generating a minimal number of wildcard rules, at the expense of some inaccuracy in the load distribution. In our ongoing work, we are exploring algorithms that make incremental adjustments to the wildcard rules based on measurements, with the goal

of achieving a more accurate distribution of load with a relatively small number of rules.

3.2 Network of Multiple Switches

In addition to non-uniform traffic, our algorithms must handle larger networks topologies. The simplest approach is to treat server load balancing and network routing separately. After the controller partitions client IP addresses based on the load-balancing weights and computes the shortest path to each replica, the controller installs rules that direct traffic along the shortest path to the chosen replica. The ingress switches that receive client traffic apply wildcard rules that modify the destination IP address and forward the traffic to the next hop along the shortest path; the subsequent switches merely forward packets based on the modified destination IP address.

In the example in Figure 6, packets entering the network at switch 1 with source IP addresses in 1* would be forwarded out the link to switch 3 with a modified destination address of R_1. Similarly, switch 1 would forward packets with sources in 00* and 01* out the link to switch 2, with the destination IP addresses changed to R_2 and R_3, respectively. Switch 2 then forwards the packets to the appropriate server, using a rule that matches on the destination IP address. If switch 2 also served as an ingress switch, then the switch would also need wildcard rules that match on the source IP address and modify the destination address. In practice, a network has a relatively small number of ingress switches that receive client requests, limiting the number of switches that match on client IP addresses and rewrite the destination address.

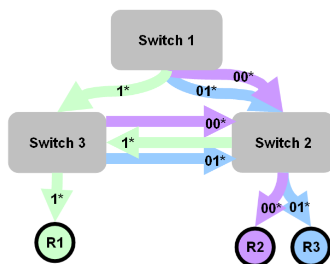

Figure 6: Network of multiple switches: R_1, R_2, and R_3 receive 50%, 25%, and 25% of load, respectively using wildcard rules 1*, 00*, and 01*. Arrows show shortest-path tree for each replica.

The controller can use the existing transitioning algorithms for the ingress switches to change from one set of wildcard rules to another. For example, consider the transitioning algorithm in Section 2.3.1 that directs the next packet of each connection to the controller where it is determined if the connection is old or new. The corresponding microflow rule will rewrite the packet's destination address to direct the packet to the correct replica. Only the ingress switches need to install microflow rules, since all other switches merely forward packets based on the destination IP address. Installing each microflow rule at every ingress switch ensures that the connection's traffic would be assigned the correct replica, even if the traffic enters the network at a new location. We are currently exploring a complete solution for ensuring that all connections are directed to the correct server replica, across changes in the ingress point during the transitions of wildcard rules.

4 Conclusion

Online services depend on load balancing to fully utilize the replicated servers. Our load-balancing architecture proactively maps blocks of source IP addresses to replica servers so client requests are directly forwarded through the load balancer with minimal intervention by the controller. Our "partitioning" algorithm determines a minimal set of wildcard rules to install, while our "transitioning" algorithm changes these rules to adapt the new load-balancing weights. Our evaluation shows that our system can indeed adapt to changes in target traffic distribution and that the few packets directed to the controller have minimal impact on throughput.

References

[1] Aster*x GEC9 demo. http://www.openflowswitch.org/foswiki/bin/view/OpenFlow/AsterixGEC9.

[2] Foundry ServerIron load balancer. http://www.foundrynet.com/products/webswitches/serveriron/.

[3] Mongoose - easy to use web server. http://code.google.com/p/mongoose/.

[4] Microsoft network load balancing. ttp://technet.microsoft.com/en-us/library/bb742455.aspx.

[5] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an operating system for networks. *ACM SIGCOMM Computer Communications Review*, 38(3), 2008.

[6] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. Plug-n-Serve: Load-balancing web traffic using OpenFlow, Aug. 2009. Demo at *ACM SIGCOMM*.

[7] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *ACM SIGCOMM HotNets Workshop*, 2010.

[8] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communications Review*, 38(2):69–74, 2008.

[9] 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 *ACM SIGCOMM HotNets Workshop*, Monterey, CA.

[10] M. Schlansker, Y. Turner, J. Tourrilhes, and A. Karp. Ensemble routing for datacenter networks. In *ACM ANCS*, La Jolla, CA, 2010.