OPENFLOW-BASED LOAD BALANCING GONE WILD

RICHARD WANG

MASTER'S THESIS

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE MASTER OF SCIENCE IN ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE

PRINCETON UNIVERSITY

ADVISER: JENNIFER REXFORD

May 2011

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 a_j that determines the share of requests the replica should handle; for example, R_2 should receive 50% (i.e., 4/8) of 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.

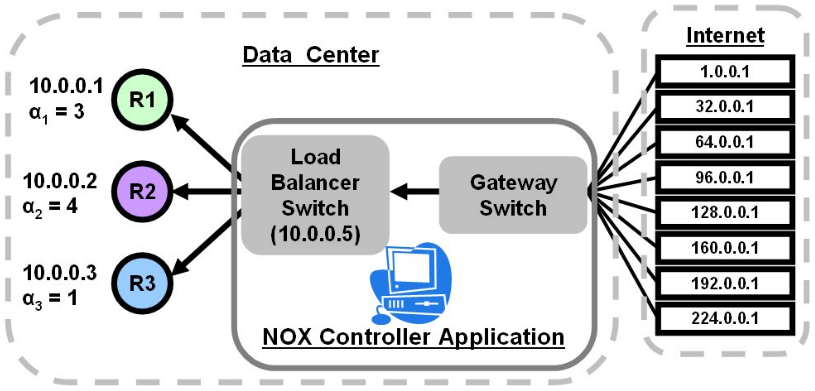

Figure 1: Basic model from load balancer switch's view

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 space1. 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 $\{a_j\}$ 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

a_j leaf nodes; for example, replica R_2 is associated with four leaves. However, the $\{a_j\}$ 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.

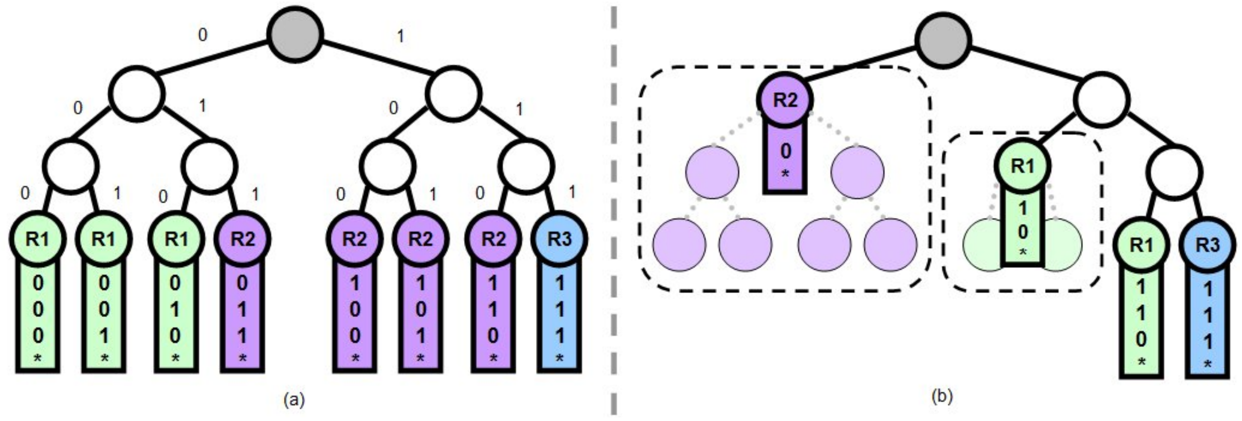

Figure 2: $a_1 = 3$, $a_2 = 4$, and $a_3 = 1$. Assuming uniform distribution of traffic: (a) wildcard rules assigning leaf nodes to a perfect binary tree achieving target distribution. (b) fewer wildcard rules.

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* associated with R_2. Similarly, the rule 00* could represent the two leaf nodes 000* and 001* associated with R1, 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 a_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 a1 = 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 a 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.

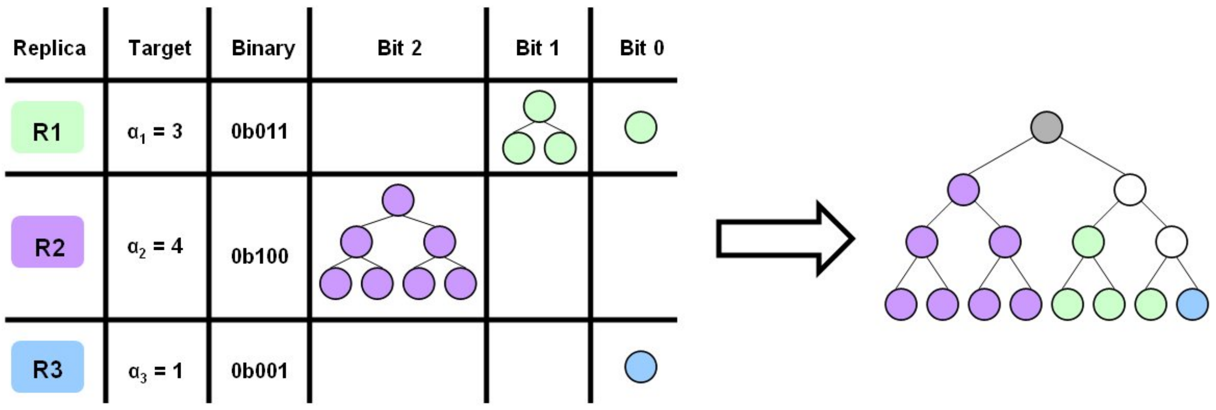

Figure 3: The binary representation of desired target reveals exactly the number of rules necessary for each replica server (R_1 requires two wildcard rules while R_2 requires a single wildcard rule). Starting with an empty IP tree, assigning groups of adjacent leaf nodes ordered based on the highest bit-i set to 1 first allows us to achieve the appropriate IP tree and replica assignments that we see here (right) and in Figure 2(b).

2.2.2 Minimizing Churn During Re-Partitioning

The weights $\{a_j\}$ 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., a_3 decreases to 0, and a_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 $\{a_j\}$ 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 a_j. Even if the total number of bits to represent the old a_j and new a_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*.

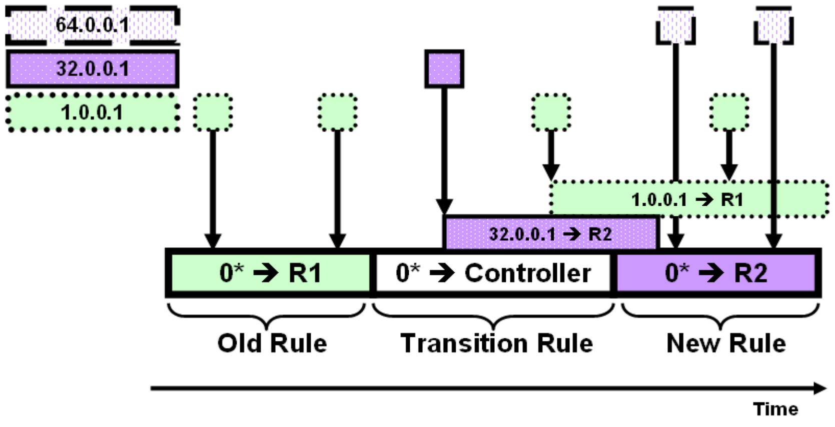

Figure 4: Transitions for wildcard rule changes: Square boxes represent packets sent by client on left. Traffic during transitions are assigned microflow rules.

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 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. 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 R_1. 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*.

2.4 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 $a_1 = 3$, $a_2 = 4$, and $a_3 = 1$, as in Figure 2.

At time 75 seconds, we change a_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 (indicatedby 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.

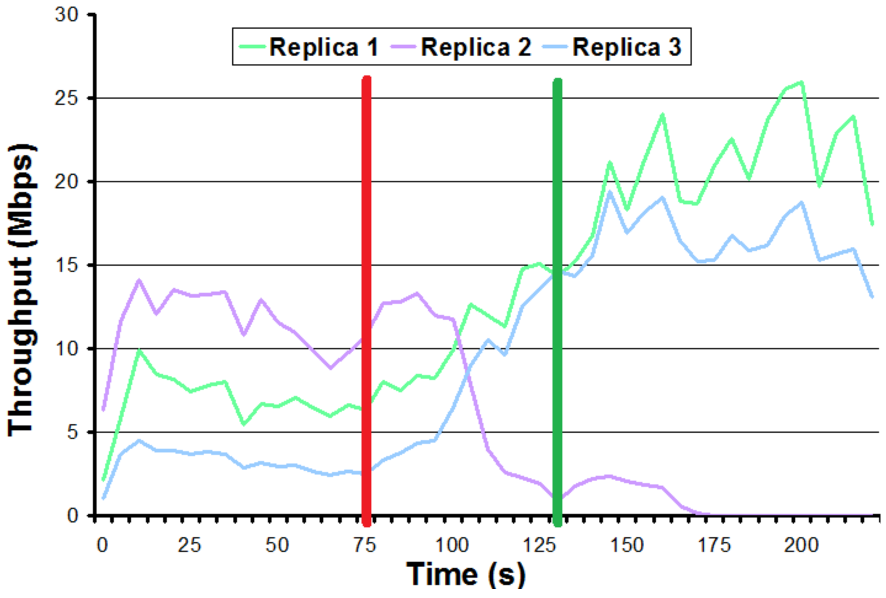

Figure 5: Throughput of experiment demonstrating ability to adapt to changes in division of load. Vertical lines indicate start and end of transitions.

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 initial assumption in determining the wildcard rules is that the amount of traffic from client requests is uniformly distributed across all IP addresses. The implication of this meant that all wildcard rules with the same number of wildcard rules each experience the exact same amount of traffic. As a result, our goal was to minimize the number of wildcard rules required to represent a particular matching of wildcard rules to replica servers.

Eliminating the uniformity assumption of traffic means that our algorithm not only needs to discover the amount of traffic for each wildcard rule but also assign them to replica servers such that the desired distribution is achieved. We notice that such a scenario, the traffic for each currently installed set of wildcard rules achieves a particular distribution among replica servers. If this does not achieve the desired distribution, the load balancer must determine the appropriate set of wildcard rules for the next iteration. As we can see, this is a feedback loop where the set of wildcard rules adjust to the traffic measurements from the previous iterations.

In essence, adjusting to the non-uniformity of traffic is accomplished by finding the set of wildcard rules that each account for a significant amount of traffic, regardless of the number of wildcard bits. While we would ideally like to have an algorithm that both searches for significant rules while also minimizing the number of rules, this seems like it would be a difficult optimization. As a result, our algorithm shifts its focus away from minimizing the number of wildcard rules from earlier sections and to focus on finding a simple algorithm to discover significant rules.

Our goal is to find a set of wildcard rules that handle traffic within a particular threshold. As a result, each wildcard rule handles approximately the same amount of traffic so dividing wildcard rules as specified by the proportion of the desired load to each replica will achieve a reasonable approximate distribution of traffic. A rule whose current proportion of traffic falls below this threshold is considered to be handling less than its fair share of traffic. A rule that handles a proportion of traffic above this threshold is considered to be handling more than its fair share of traffic. The amount of traffic handled by each wildcard rule is easy to gather because Openflow keeps counters for the number of packets as well as bytes that match any particular rule. Our algorithm simply needs to periodically send requests to the Openflow switches for these counter values.

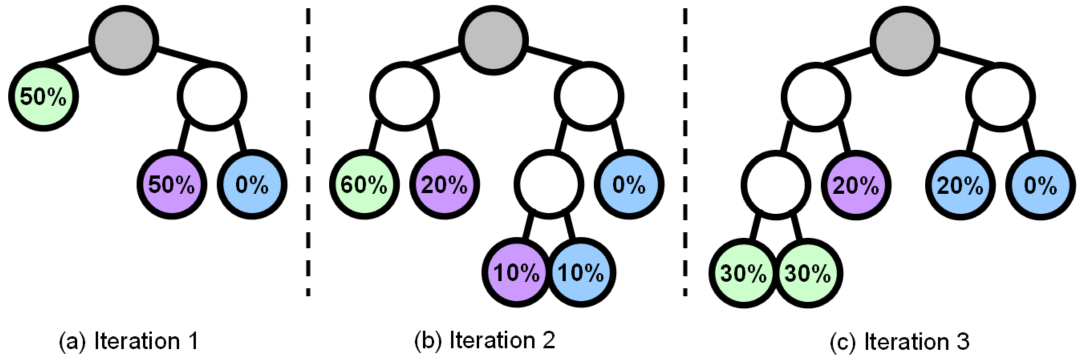

Figure 6: Wildcard rules reacting to non-uniform traffic (15% < Threshold < 40%): (a) Two rules are above the threshold so generate more precise rules. (b) Traffic has shifted across IP addresses and one rule is above threshold

while two adjacent rules are under threshold. (c) The resulting traffic measurements suggest a stable set of rules for this iteration. Key Concepts: Adjust rules based on traffic for currently installed rules with no assumptions about where traffic will arrive for the next iteration to discover set of significant rules. The threshold and iteration period can be adjusted so that the traffic measurements are not too short such that rules constantly oscillate and not too coarse grain such that rules react to shifts in traffic too slowly.

Comparing the counter values with the threshold will reveal if a particular wildcard rule matches too much or too little traffic. A wildcard rule that handles too much traffic is split into two more precise wildcard rules. A wildcard rule that handles too little traffic will be combined with its adjacent wildcard rule into a single rule if and only if the adjacent wildcard rule also handles too little traffic. While this allows for the existence of wildcard rules under the threshold, this is only the case if either the adjacent rules represent significant traffic. Unfortunately, there is not much that can be done in this scenario because of the way in which wildcard rules are specified. Combining all under threshold rules with their adjacent rules would most likely end up with a rule that would end up above the threshold and would be split during subsequent iterations anyway. As a result, the set of wildcard rules is able to react to any changes in traffic across IP addresses as well as discovering the set of rules that each represent approximately the same amount of traffic.

3.2 Network of Multiple Switches

Effectively taking advantage of the links connecting the network of Openflow switches is important to avoiding the network becoming a bottleneck so that replica servers are fully utilized. Given the set of wildcard rules that we wish to install and their replica assignments from previous sections, each Openflow switch must determine the appropriate set of forwarding paths to reach each of the replica servers so determing the forwarding path is independent of determing the wildcard rule and replica assignments. Taking Dijkstra’s algorithm to determine the shortest path from each replica server to each of the Openflow switches, it becomes trivial to determine the appropriate next hop for each wildcard rule's replica assignment.

This simple approach will ensure that each Openflow switch will have a path to each replica server if possible. Unfortunately, in much the same way that we do not assume client traffic is uniformly distributed across IP addresses, we also cannot assume that client requests will be uniformly distributed across the Openflow switches. Let us consider a worst-case scenario where all client requests initially arrive at the same Openflow switch. In such a scenario, we only utilize a single shortest path to reach each replica server and we can see that these few paths will become heavily congested. If there were any alternate paths that could help alleviate this congestion, they would be unused.

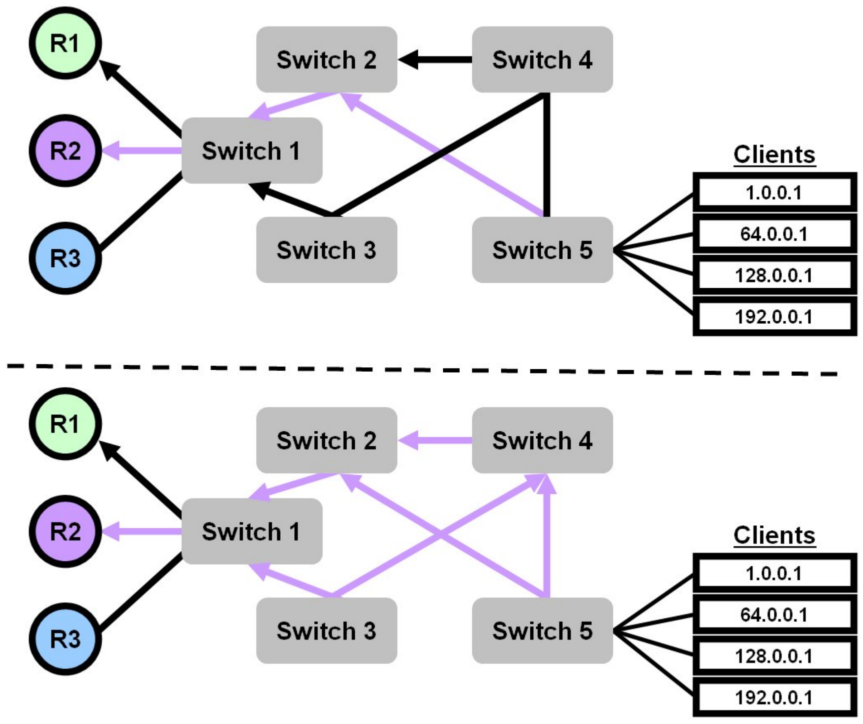

Figure 7: Forwarding with multiple switches on this 5 switch topology: (top) Installing wildcard rules that choose a single shortest forwarding path means that if all client traffic initially arrives at a single switch, they will all traverse

the same forwarding path to the destined replica R2. (bottom) Installing a more precise set of wildcard rules that utilize available forwarding paths that do not create cycles utilizes more available links for higher throughput.

Utilizing these alternate paths to alleviate congestion requires finding these alternate paths available and then distributing client traffic over these paths. Alternate paths are easy to find for each replica server from the shortest path tree. Each switch’s next hop to a replica can be the set of all neighboring switches that have a shorter path to the replica. Taking the shortest path tree we computed earlier, a switch can utilize each path that is closer to the destined path than itself. This simple approach prevents any forming cycles. Distributing requests over these paths can be accomplished by taking a similar approach to distributing load over replicas with wildcard rules. For each wildcard rule, we generate a set of more precise wildcard rules that together represents the same IP address range. We can then distribute the forwarding paths based on the available next hops for each switch.

Taking the worst-case scenario of all client requests arriving at a single Openflow switch in ref/multiple, we ran a simulation with 20 clients performing simultaneous requests for a 128 MB file from a single replica server. As we can see from ref/multiple, the topology was specifically designed to try to isolate the effects of using multiple paths. All clients are connected to a single switch and they are all assigned to replica R2. It is easy to see that the shortest single path solution neglects many available links while the multiple path solution can take advantage of these additional links. On average, we found that it took each client .4199 seconds to finish downloading when using just a single shortest path versus an average of .3837 seconds when each wildcard rule assignment is replaced with several more precise rules forwarding on available paths. This is about an 8% decrease in completion time. More impressive is in the standard deviation of completion time, which was .132 for shortest path and .059 for multiple paths. In this simulation, not only were clients completing their downloads much faster, they are also more likely to finish around the same time.

We can see from our simulation that even a simple algorithm in using available forwarding paths was quite effective. While there are many options in the assignment of wildcard rules and next hops, this basic mechanism of using wildcard rules as a hashing function to utilize multiple paths allows us to avoid typical multipath issues such as packet re-ordering and any additional hashing computation but still manage to distribute client traffic among available paths.

3.3 Packet Rewriting and Transitioning for Multiple Switches

Two important issues that should not be overlooked are how packet rewriting and how microflow rule installation should operate in a multiple switch environment. The appropriate approach depends on the designated role of the load balancer. Load balancers have typically been assigned to perform the task of assigning clients to particular replica servers, more or less like a proxy. An alternative is to treat the load balancer more than just a simple proxy, but also combine with network issues like alleviating congestion and addressing locality.

Packet rewriting for the traditional load balancer whose main task is to assign clients to replica servers is accomplished by performing packet rewriting for client’s packets at the first load balancer switch that they arrive at. Since the header for incoming clients are immediately rewritten to the destine replica server for both wildcard and microflow rules, subsequent load balancing switches simply continue forwarding the packets to the destined replica server. Unfortunately, the immediate packet rewriting limits us from performing implementing something like the multipath solution because of the uncertainty of where these packets originated and what paths they have already traversed.

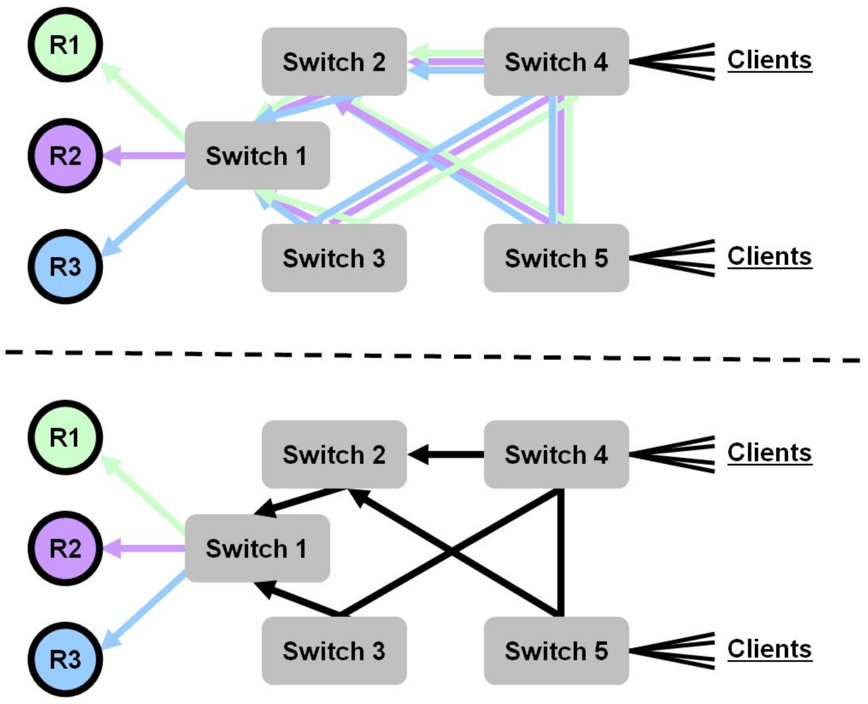

Figure 8: Packet rewriting of the destination IP from the virtual IP address to the IP address of the destined replica server: (top) Packet rewriting at the first switch a client's request arrives at means that the network will be filled with lots of traffic towards various destination replicas. (bottom) Delaying packet rewriting until the switch directly connected to the destined replica server allows the load balancer to customize wildcard rules to cater the network to address additional issues that can be addressed with clever path selection.

The alternative is to delay packet rewriting until the switch immediately connected to the destined replica server. As a result, because the load balancing switches are programmed by a central controller that is reflected in the wildcard rules that are installed, it becomes much easier for the load balancer to address more issues than just the basic task of replica assignment by customizing wildcard rules like we did in the multipath section. These wildcard rules could additionally be modified to account for typically network-relevant issues like congestion and locality that require a top-down understanding of the state of the load balancing switches that the controller is aware of.

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. Our algorithm also works on multiple switches in a data center where we have explored the initial steps for the load balancer to be integral in routing traffic within the network in addition to the basic client to replica server assignments. There remain many exciting opportunities to further extend the capabilities of a load balancer to be integrated with networking decisions to provide low-latency, high throughput sessions with clients.

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 Open-Flow, 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 Hot-Nets 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.