Towards an Elastic Distributed SDN Controller

†Advait Dixit, ‡Fang Hao, ‡Sarit Mukherjee, ‡T.V. Lakshman, †Ramana Kompella
†Purdue University, ‡Bell Labs Alcatel-Lucent

ABSTRACT

Distributed controllers have been proposed for Software Defined Networking to address the issues of scalability and reliability that a centralized controller suffers from. One key limitation of the distributed controllers is that the mapping between a switch and a controller is *statically configured*, which may result in uneven load distribution among the controllers. To address this problem, we propose ElastiCon, an *elastic distributed controller architecture* in which the controller pool is dynamically grown or shrunk according to traffic conditions and the load is dynamically shifted across controllers. We propose a novel switch migration protocol for enabling such load shifting, which conforms with the Openflow standard. We also build a prototype to demonstrate the efficacy of our design.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

Keywords

Data center networks, software-defined networks

1. INTRODUCTION

Software Defined Networking (SDN) is revolutionizing the networking industry by enabling programmability, easier management and faster innovation [11, 8, 4, 14]. Many of these benefits are made possible by its centralized control plane architecture, which allows the network to be programmed by the application and controlled from one central entity. However, like any other centralized system, the centralized controller brings up issues of scalability and reliability. Hence the next logical step is to build a logically centralized, but physically distributed control plane, which can benefit from the scalability and reliability of the distributed architecture while preserving the simplicity of the centralized system.

A few recent papers have explored architectures for building distributed SDN controllers [10, 16, 13]. While these have focused on building the components necessary to implement a distributed SDN controller, one key limitation of these systems is that the mapping between a switch and a controller is *statically configured*, making it difficult for the control plane to adapt to traffic load variations. Real networks (e.g., data center networks, enterprise networks) may exhibit significant variations in both temporal and spatial traffic characteristics. First, along the temporal dimension, it is generally well-known that traffic conditions can depend on the time of day (e.g., less traffic during night), but there could be variations even in shorter time scales (e.g., minutes to hours) depending on the applications running in the network. For instance, based on measurements over real data centers in [2], we can estimate that the peak-to-median ratio of flow arrival rates can be almost 1-2 orders of magnitude (more details in Section 2). Second, there could be spatial traffic variations; depending on where applications are generating flows, some switches can observe a larger number of flows compared to other portions of the network.

Now, if the switch to controller mapping is static, a controller may become overloaded if the switches mapped to this controller suddenly observe a large number of flows, while other controllers remain underutilized. Furthermore, the load may shift across controllers over time, depending on the temporal and spatial variations in traffic conditions. Hence static mapping can result in suboptimal performance. One way to improve performance may be to over-provision controllers for an expected peak load, but this approach is clearly inefficient due to its high cost and energy consumption, especially considering load variations can be up to two orders of magnitude.

To address this problem, in this paper, we propose ElastiCon, an *elastic distributed controller architecture* in which the workload is dynamically shifted to allow the controllers to operate at a prespecified load window. When the aggregate load changes over time, the system dynamically expands or shrinks the controller pool as needed. Our goal in this paper is to explore what ingredients are necessary to enable such an elastic controller architecture. Clearly, as load imbalance occurs, it is desirable to *migrate* a switch from a heavily-loaded controller to a lightly-loaded one. However, such a migration operation is not supported natively in the current de facto SDN standard, OpenFlow. Designing such a conceptually-simple primitive is not straightforward since it requires minimal disruption to normal operations, while guaranteeing consistency and reliability. We design a new migration algorithm that achieves these properties based on the existing OpenFlow standard.

The migration primitive, however, is not sufficient by itself. We need additional mechanisms to support the following three main load adaptation operations: First, we need to periodically *load balance* the controllers by optimizing the switch to controller mapping on the fly. Second, if the load exceeds the maximum capacity of existing controllers, we need to *grow* the resource pool by adding

HotSDN'13, August 16, 2013, Hong Kong, China.

new controllers, and triggering switch migrations to utilize the new controller resource. Similarly, when the load falls below a particular level, we need to *shrink* the resource pool accordingly. All these actions involve both mechanisms for measuring and monitoring controller load and algorithms for deciding when and what switches to migrate.

In summary, this paper makes the following contributions:

- We have designed an initial architecture for ElastiCon that provides the ability to dynamically adapt the controller resource pool.
- We have proposed a new migration protocol for seamless migration of a switch from one controller to another.
- To test ElastiCon at scale, we have developed a new OpenFlow-based network emulator that extends Mininet to span across multiple hosts.
- Using our emulator, we show that the migration protocol does not cause any message loss or observable delay, thereby showing the promise of our approach.

2. BACKGROUND AND MOTIVATION

The OpenFlow network consists of both switches and a central controller. A switch forwards packets according to *rules* stored in its flow table. The central controller controls each switch by setting up the rules. Multiple application modules can run on top of the core controller module to implement different control logics and network functions. Packet processing rules can be installed in switches either reactively (when a new flow is arrived) or proactively (controller installs rules beforehand). We focus on the performance of the reactive mode in this paper.

The controller architecture has evolved from the original single-threaded design [9] to the more advanced multi-threaded design [17, 1, 3, 5] in recent years. Despite the significant performance improvement over time, the single-controller systems still have limits on scalability and vulnerability. Some research papers have also explored the implementation of distributed controllers across multiple hosts [10, 16, 13]. The main focus of these papers is to address the state consistency issues across distributed controller instances, while preserving good performance. Onix, for instance, uses a transactional database for persistent but less dynamic data, and memory-only DHT for data that changes quickly but does not require consistency [10]. Hyperflow replicates the events at all distributed nodes, so that each node can process such events and update their local state [16]. [13] has further elaborated the state distribution trade-offs in SDNs.

All existing distributed controller designs implicitly assume static mapping between switches and controllers, and hence lack the capability of dynamic load adaptation and elasticity. However, the following back-of-the-envelope calculation using real measurement data shows that there is 1-2 orders of magnitude difference between peak and median flow arrival rates at a switch. In [2], Benson *et al.* show that the minimum inter flow arrival gap is $10\mu s$, while the median ranges roughly from $300\mu s$ to $2ms$ across different data centers that they have measured. Assuming a data center with 100K hosts and 32 hosts/rack, peak flow arrival rate can be up to $300M$ with the median rate between 1.5M and 10M. Assuming 2M packets/sec throughput[1] for one controller [17], it requires only 1-5 controllers to process the median load, but 150 for peak load. If we use static mapping between switches and controllers, it requires significant over-provisioning of resources which is inefficient in hardware and power; an elastic controller that can dynamically adapt to traffic load is clearly more desirable.

[1]This is based on the learning switch application. Throughput is lower for more complex applications, as shown in our experiments.

3. ELASTIC CONTROLLER DESIGN

In this section, we present the design and architecture of ElastiCon, an elastic distributed SDN controller. We describe ElastiCon in three phases: First, we start with a basic distributed controller design that spreads functionality across several nodes by extending a generic open source controller. We then provide a protocol for switch migration, which is one of the core procedures needed for implementing an elastic controller. Finally, we discuss briefly how elasticity and load adaptation are achieved in our design.

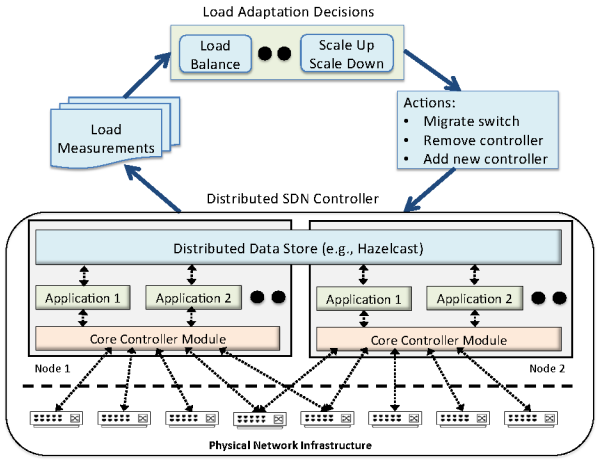


Figure 1: ElastiCon architecture.

3.1 Basic Distributed Controller

The key components in our distributed controller design are shown in Figure 1 (bottom part). It consists of a cluster of autonomous *controller nodes* that coordinate amongst themselves to provide a consistent control logic for the entire network. The *physical network infrastructure* refers to the switches and links that carry data and control plane traffic. A switch connects to multiple controller nodes, one of which acts as the master and the rest as slaves. Each node has a *core controller module* that executes all the functions of a centralized controller (i.e., connecting to a switch, event management between a switch and an application). In addition, it coordinates with other controllers to elect a master node for a newly connected switch and orchestrates the migration of a switch to a different controller.

The *distributed data store* glues the cluster of controllers to provide a logically centralized controller. It stores all switch specific information that is shared among the controllers. Each controller node also maintains a TCP channel with every other controller node. A controller node may use this channel for sending messages to a switch controlled by another node or coordinating actions during switch migration. The *application* module implements the control logic of network applications, responsible for controlling the switches for which its controller is the master. Application state is kept in the distributed data store to facilitate switch migration and controller failure recovery.

Next we show how to support dynamic switch migration on top of this generic basic distributed controller architecture.

3.2 Switch Migration

In order to enable elasticity, controllers must be added to or removed from the existing controller pool when the aggregate traffic load goes below or beyond the pre-determined load thresholds. Even if the aggregate load remains the same, load balancing operation must be done periodically to adapt to traffic load imbalances. These operations require the ability to perform a switch migration operation, for which there is no native mechanism provided in the Openflow standard.

Before we describe a new migration protocol, it is important to review some terminology. OpenFlow 1.3 defines three operational modes for a controller: *master*, *slave*, and *equal* [7]. By default, the slave controller has read-only access to the switch, and does not receive asynchronous messages (e.g., `Packet-In`). Both master and equal controllers, however, can modify switch state and receive asynchronous messages from the switch. There can be at most one master for each switch, but many equal-mode controllers. The controller changes its role by sending `Role-Change` messages to the switch. The switch migration process needs to make one of the slaves the new master, and the old master should be role-reversed to a slave (or failed).

For any migration protocol, it is important to guarantee the standard *liveness* and *safety* properties.

- **Liveness.** At least one controller is active (either in master or equal role) for a switch at all times. For each asynchronous message, the controller that issues the command remains active until the switch finishes the command processing. This ensures that commands like `Packet-Out` and `Flow-Mod` always succeed without disruption.
- **Safety.** Exactly one controller processes every asynchronous message from the switch; duplicate processing of asynchronous messages such as `Packet-In` could result in duplicate entries in the flow table or inconsistency in the distributed data store.

Now, consider the following naive protocol: The target slave controller node simply sends a `Role-Request` message to the switch indicating that it wants to become the master. The switch would set that controller node as the master and the previous master as slave. Such a naive protocol, however, can violate the liveness property. Assume that the switch had sent a `Packet-In` message to the initial master. If the switch receives the `Role-Request` message from the slave before the `Packet-Out` message from the initial master, then the switch will ignore the `Packet-Out` message since it is designed to ignore messages from any controller which is not the master/equal. Ideally, the switch can buffer all these `Packet-In` requests and try retransmitting the `Packet-In` message to the new master, but that makes the switch design complicated, which is not desirable. One simple way to fix this would be to first transition the target controller into the equal role, so that both the initial and final masters can be responsible for the switch before detaching the initial master. Unfortunately, while this solves the liveness problem, it may violate the safety property since both controllers may process messages from the switch.

In our protocol design, we assume we cannot modify the switch; while it is technically feasible to update the OpenFlow standard, it is not desirable to put more functionality in the switches than necessary anyway. We can of course create a proprietary communication protocol between two controller nodes, which is what we exploit in our protocol design. There are two additional issues: First, the OpenFlow standard clearly states that a switch may process messages not necessarily in the order they are received, mainly to allow multi-threaded implementations. We need to factor this in our protocol design. Second, the standard does not specify explicitly whether the ordering of messages transmitted by the switch remains consistent across two controllers that are in master/equal mode. We need this assumption for our protocol to work; allowing arbitrary reordering of messages across two controllers will make an already hard problem significantly harder.

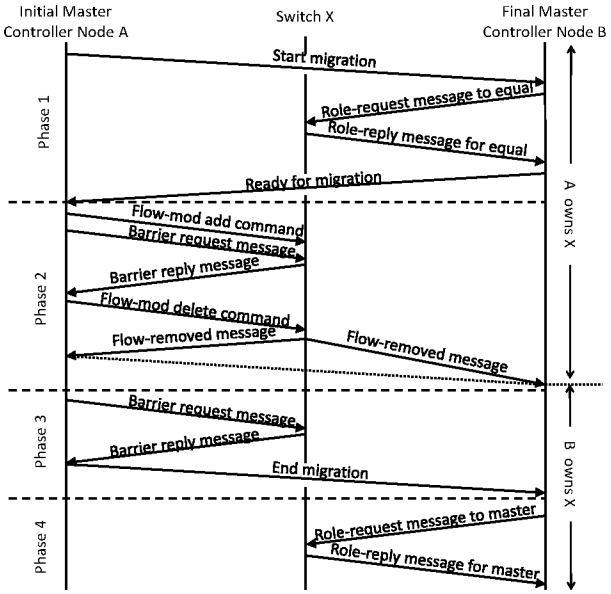


Figure 2: Message exchanges for switch migration.

Our protocol is built on the key idea that we need to first create a *single trigger event* to stop message processing in the first controller and start the same in the second one. Fortunately, we can exploit the fact that `Flow-Removed` messages are transmitted to all controllers operating in the equal mode. We therefore simply insert a dummy flow into the switch from the first controller and then remove the flow, which will provide a single trigger event to both the controllers in equal mode to signal handoff. Our proposed protocol for migrating switch X from initial controller A to target controller B operates in four phases as shown below (also shown in Figure 2).

Phase 1: Change role of the target to equal. In the first phase, target B is first transitioned to the equal mode for the switch X. Initial master A initiates this phase by sending a start migration message to B, on the controller-to-controller channel. B sends `Role-Request` message to the switch informing that it is an equal. After B receives a `Role-Reply` message from the switch, it informs the initial master A that its role change is completed. After B changes its role to equal, it receives asynchronous messages from the switch, but ignores them and does not respond. During this phase, A remains the only master and processes all messages from the switch guaranteeing liveness and safety.

Phase 2: Insert and remove a dummy flow. To determine an exact instant for the migration, A sends a dummy `Flow-Mod` command to X to add a new flow table entry that does not match any incoming packet. We assume all controllers know this dummy flow entry *a priori* as part of the protocol. Then, it sends another `Flow-Mod` command to delete this entry; in response, the switch sends a `Flow-Removed` message to both controllers since B is in the equal mode. This `Flow-Removed` event provides a transfer of ownership for switch X from A to B, and henceforth, only B will process all messages transmitted by switch. An additional barrier message is required after the insertion of the dummy flow and before the dummy flow is deleted to prevent any chance of processing

the delete message before the insert. Alternately, all switches could be started with a dummy flow entry already inserted in the flow table, so that we only do the deletion in this phase.

Note that we do not assume that the `Flow-Removed` message is received by A and B at exactly the same time (as shown in Figure 2). Since we make the assumption the message ordering is consistent across A and B after these controllers enter equal mode, this means that all messages before this `Flow-Removed` will be processed by A and after this will be processed by B, thus guaranteeing the safety property. Liveness is also clearly guaranteed since both controllers are active for switch X.

Phase 3: Flush pending requests with a barrier. While B has assumed ownership of X in the previous phase, the protocol is not complete unless A detaches from managing switch X. However, it cannot just detach immediately from the switch since there may be pending requests at A that arrived before the `Flow-Removed` message, for which A is still the owner. This appears easy since we assume same ordering at A and B, so all A needs to do is to just wait until all the messages that arrived before `Flow-Removed` are processed by A and committed to the switch. However, there is no explicit acknowledgment from the switch that these messages are committed; TCP-level acknowledgments do not mean anything since the switch still needs to commit these messages, and it does so in any order. If these messages are not committed and A detaches signaling to B to become the new master, that will automatically reduce the node A to a slave, which will cause the switch to ignore those previous commits. Thus, in order to make sure all these messages are committed, A transmits a `Barrier-Request` and waits for the `Barrier-Reply`, only after which it signals "end migration" to the final master B.

Phase 4: Make target controller final master. The final master B sets its role to master for the switch by sending a `Role-Request` message to the switch. It also updates the distributed data store to indicate this. The switch sets A to slave when it receives the `Role-Request` message from the final master B. B remains active and processes all messages from the switch for this phase, so both safety and liveness are guaranteed.

The above algorithm requires 6 round-trip times (including inter-controller node communication) to complete the migration. But note that we need to trigger migration only once in a while when the load conditions change, as we discuss in the next section.

3.3 Load Adaptation

Figure 1 (top part) shows the load adaptation procedure in three steps: load measurement, adaptation decision computation, and migration action.

Load Estimation. A load estimation module runs on the controller to report load statistics, including CPU and memory usage, and network I/O rate. It also reports the average message arrival rate from each switch. Our experiments show that the CPU is typically the throughput bottleneck and CPU load is roughly in proportion to the message rate.

Adaptation Decision Computation. We set high and low thresholds both for the individual controller load and for the overall aggregated load of the controller pool. When an individual controller load is beyond the thresholds but aggregated load is within the threshold, we invoke load balancing by migrating selected switches to their new master controllers. When the aggregated load is beyond the threshold, new controllers will be added or existing controllers will be removed. Switch and the new master controller selection are based on both the load conditions and network topology. For example, it may be desirable to have neighboring switches to be controlled by the same master to reduce inter-controller communication.

Adaptation Action. Following the adaptation decision, a switch can be migrated to a former slave by following the steps in Section 3.2. In case of controller addition or removal, one or more switches may need to be assigned to new master controllers that they are not currently connected with. This can be done by replacing one slave controller IP address of the switch with the new controller using the `edit-config` operation of OpenFlow Management and Configuration Protocol [6]. Once the connection between the new controller and the switch is established, we then invoke the migration procedure to swap the old master with the new slave controller. If a switch does not support updating controller IP addresses at runtime, other workarounds based on controller IP address virtualization are also possible (not described due to lack of space).

3.4 Implementation Status

We implemented a prototype ElastiCon by modifying and adding components to the centralized controller, and using Hazelcast as the distributed data store. We use *routing application* as a canonical example for our prototype, although our design is generic. The routing application consists of four modules: link discovery, topology, device manager, and forwarding. We are currently implementing the load adaptation modules.

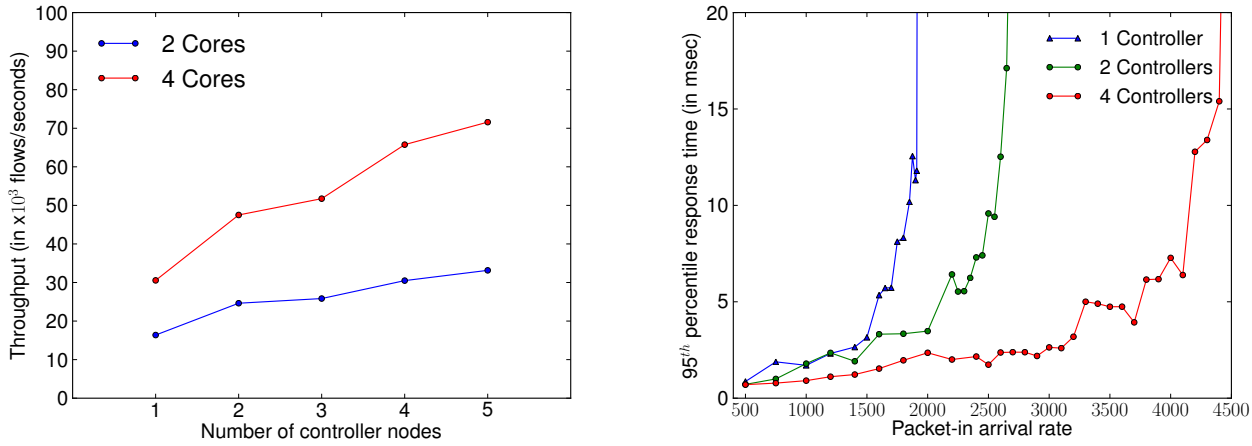
4. EVALUATION

In this section, we evaluate the performance of our ElastiCon prototype using an emulated SDN-based data center network testbed. We first describe the enhanced Mininet testbed that we used to carry out the evaluation, and then present our experimental results.

4.1 Enhanced Mininet Testbed

Our experimental testbed is built on top of Mininet [12], which emulates a network of Open vSwitches [15]. Open vSwitch is a software-based virtual Openflow switch. It implements the data plane in kernel and the control plane as a user space process. Mininet has been widely used to demonstrate the functionalities, but not the performance, of a controller because of the overhead of emulating data flows. First, actual packets need to be exchanged between the vSwitch instances to emulate packet flows. Second, a flow arrival resulting in sending a `Packet-In` to the controller incurs kernel to user space context switch overhead in the Open vSwitch. From our initial experiments we observe that these overheads significantly reduce the maximum flow arrival rate that Mininet can emulate, which in turn slows down the control plane traffic generation capability of the testbed. Note that for the evaluation of ElastiCon, we are primarily concerned with the control plane traffic load and need not emulate the high overhead data plane. We achieve this by modifying Open vSwitch to inject `Packet-In` messages to the controller without actually transmitting packets on the data plane. We also log and drop `Flow-Mod` messages to avoid the additional overhead of inserting them in the flow table. Although we do not use the data plane during our experiments, we do not disable it. So, the controller generated messages (like LLDPs, ARPs) are still transmitted on the emulated network.

In order to experiment with larger networks we deployed multiple hosts to emulate the testbed. We modified Mininet to enable us to run the Open vSwitch instances on different hosts. We created GRE tunnels between the hosts running Open vSwitch instances to emulate links of the data center network. Since we do not actually transmit packets in the emulated network, the latency/bandwidth characteristics of these GRE tunnels do not im-



(a) Controller throughput

(b) Response time

Figure 3: Performance with varying number of controller nodes.

pact our results. They are used only to transmit link-discovery messages to enable the controllers to construct a network topology. To isolate the switch to controller traffic from the emulated data plane of the network, we run Open vSwitch on hosts with two Ethernet ports. One port of each host is connected to a gigabit Ethernet switch and is used to carry the emulated data plane traffic. The other port of each host is connected to the hosts that run the controller. We isolated the inter-controller traffic from the controller-switch traffic too by running the controller on dual-port hosts.

4.2 Experimental Results

We report on the performance of ElastiCon using the routing application. All experiments are conducted on k=4 fat tree emulated on the testbed. We use 4 hosts to emulate the entire network. Each host emulates a pod and a core switch. Before starting the experiment, the emulated end hosts ping each other so that the routing application can learn the location of all end hosts in the emulated network.

Throughput. We send 10000 back-to-back `Packet-In` messages and plot the throughput of ElastiCon with varying number of controller nodes (see Figure 3(a)). We repeat the experiment while pinning the controllers to two cores of the quad-core server. We observe two trends in the results. First, adding controller nodes increases the throughput almost linearly. This is because there is no data sharing between controllers while responding to `Packet-In` messages. Second, the throughput reduces when we restrict the controllers to two cores indicating that CPU is indeed the bottleneck.

Response time. We plot the response time behavior for `Packet-In` messages with changing flow arrival rate (see Figure 3(b)). We repeat the experiment while changing the number of controller nodes. As expected, we observe that response time increases marginally up to a certain point. Once the packet generation rate exceeds the capacity of the processor, queuing causes response time to shoot up. This point is reached at a higher packet-generation rate when ElastiCon has more number of nodes.

Impact of load balancing. We show how switch migration can improve response time. In this experiment, ElastiCon has two controllers, A and B. In the beginning of the experiment, the load is

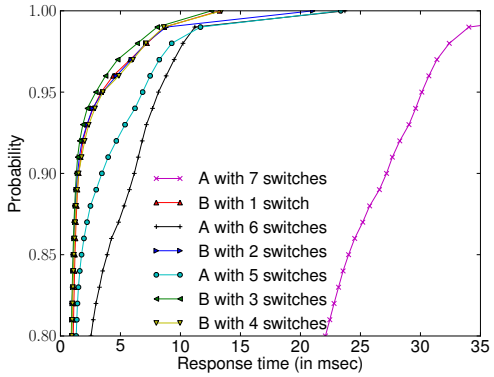


Figure 4: Impact of switch migration

unequally divided between the two nodes. Of the eight top-of-rack switches in a fat tree that generate `Packet-In` messages, seven are connected to A and one is connected to B. Note that this scenario is not that different from having four switches connected to each of the controllers, but with different load on each of the switches (and consequently controllers). Each switch generates traffic at 2000 `Packet-In` messages per second. We plot the tail of two CDFs of response time, one each for controller A and controller B. These two CDFs correspond to the first two curves in Figure 4. As the figure shows, the switch connected to controller A experiences higher response time due to the load imposed by other switches connected to the same controller. Then, we migrate one switch from controller A to controller B. Now, controller A has six switches connected to it and controller B has two. Again, we plot the CDFs of the response times of both controllers. These CDFs correspond to the third and fourth curves. As the figure shows, the response time of controler A improves due to reduced load on the controller. Response time of controller B remains almost unchanged since the load imposed by two switches is still well below its processing capability. We continue to migrate switches from controller A to B until both controllers are equally loaded. The response time of controller A reduces with its load. When a controller is connected to

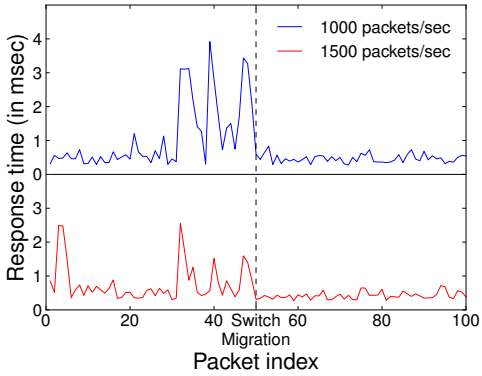


Figure 5: Effect of migration on response time

4 or fewer switches, the response time settles down. This shows load-balancing via switch migration can improve performance.

Impact due to migration. Finally, we demonstrate the feasibility of the migration algorithm described in Section 3.2. We plot the response times of 50 `Packet-In` messages before and after migration in Figure 5 for two flow arrival rates. We observe a minor increase in response time just before migration, possibly due to extra messages exchanged. But no messages are lost or duplicated. The migration process takes about 20ms. This shows that migration can be done quickly and with minimal impact on response time.

5. CONCLUSION AND FUTURE WORK

In this paper, we have presented our design of ElastiCon and showed that our initial evaluation results are very encouraging. We are continuing the implementation of the load adaptation modules, with focus on developing load adaptation algorithms that compute optimal switch migration strategy under dynamic traffic load. In addition, we plan to explicitly address the fault tolerance issues in the design by modifying the switch migration procedure. This may require running three controllers in equal role and using a consensus protocol between them. We also plan to study the impact of application data sharing patterns on scalability and elasticity.

6. ACKNOWLEDGEMENTS

The authors thank anonymous reviewers for comments that helped improve the paper. This work was supported in part by NSF Awards CNS-1054788 and CNS-1219004.

7. REFERENCES

[1] "Beacon," `openflow.stanford.edu/display/Beacon/Home`.
[2] T. Benson, A. Akella, and D. Maltz, "Network traffic characteristics of data centers in the wild," in *IMC*, 2010.
[3] Z. Cai, A. L. Cox, and T. S. E. Ng, "Maestro: A system for scalable OpenFlow control," Tech. Rep. TR10-11, CS Department, Rice University, Dec. 2010.
[4] M. Casado, M. J. Freedman, and S. Shenker, "Ethane: Taking Control of the Enterprise," in *ACM SIGCOMM*, 2007.
[5] "Floodlight," `floodlight.openflowhub.org`.
[6] Open Networking Foundation, "OpenFlow Management and Configuration Protocol (OF-Config 1.1)," June 2012.
[7] Open Networking Foundation, "OpenFlow Switch Specification (Version 1.3.0)," June 2012.
[8] A. Greenberg, G. Hjalmtysson, D. A. Maltz, et al., "A clean slate 4D approach to network control and management," in *SIGCOMM CCR*, 2005.
[9] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. Mckeown, and S. Shenker, "NOX: Towards an Operating System for Networks," in *SIGCOMM CCR*, 2008.
[10] T. Koponen et al., "Onix: A Distributed Control Platform for Large-scale Production Networks," in *OSDI*, 2010.
[11] T.V. Lakshman, T. Nandagopal, R. Ramjee, K. Sabnani, and T. Woo, "The SoftRouter Architecture," in *ACM HOTNETS*, 2004.
[12] B. Lantz, B. Heller, and N. McKeown, "A network in a Laptop: Rapid Prototyping for Software-Defined Networks," in *HotNets*, 2010.
[13] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically Centralized? State Distribution Trade-offs in Software Defined Networks," in *HotSDN*, 2012.
[14] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, et al., "Openflow: enabling innovation in campus networks," *SIGCOMM CCR*, 2008.
[15] "Open vswitch," `openvswitch.org`.
[16] A. Tootoonchian and Y. Ganjali, "HyperFlow: A Distributed Control Plane for OpenFlow," in *INM/WREN*, 2010.
[17] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On Controller Performance in Software-Defined Networks," in *HotICE*, 2012.