ElastiCon: An Elastic Distributed SDN Controller

[†]Advait Dixit, [‡]Fang Hao, [‡]Sarit Mukherjee, [‡]T. V. Lakshman, [†]Ramana Rao Kompella
[†]Dept. of Computer Science, Purdue University, West Lafayette, IN, USA
{dixit0, kompella}@cs.purdue.edu
[‡]Bell Labs, Alcatel-Lucent, Holmdel, NJ, USA
{fang.hao, sarit.mukherjee, t.v.lakshman}@alcatel-lucent.com

ABSTRACT

Software Defined Networking (SDN) has become a popular paradigm for centralized control in many modern networking scenarios such as data centers and cloud. For large data centers hosting many hundreds of thousands of servers, there are few thousands of switches that need to be managed in a centralized fashion, which cannot be done using a single controller node. Previous works have proposed distributed controller architectures to address scalability issues. A key limitation of these works, however, is that the mapping between a switch and a controller is *statically configured*, which may result in uneven load distribution among the controllers as traffic conditions change dynamically.

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. To address the load imbalance caused due to spatial and temporal variations in the traffic conditions, ElastiCon automatically balances the load across controllers thus ensuring good performance at all times irrespective of the traffic dynamics. We propose a novel switch migration protocol for enabling such load shifting, which conforms with the Openflow standard. We further design the algorithms for controller load balancing and elasticity. We also build a prototype of ElastiCon and evaluate it extensively 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) has emerged as a popular paradigm for managing large-scale networks including data centers and cloud. The key tenet of SDN is the centralized control plane architecture, which allows the network to be programmed by applications running on one central entity, enabling easier management and faster innovation [13, 9, 5, 16]. However, many of these large-scale data center networks consist of several hundreds of thousands of servers interconnected with few thousands of switches in tree-like topologies (e.g., fat tree), that cannot easily be controlled by a single centralized controller. 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 a logically centralized system.

A few recent papers have explored architectures for building distributed SDN controllers [12, 20, 15]. 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) 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 are 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 [3], we estimate that the peak-to-median ratio of flow arrival rates is almost 1-2 orders of magnitude[1] (more details in Section 2). Second, there are often spatial traffic variations; depending on where applications are generating flows, some switches 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 sub-optimal performance. One way to improve performance is 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 controller pool expands or shrinks dynamically as the aggregate load changes over time. While such an elastic architecture can ensure there are always enough controller resources to manage the traffic load, per-

[1]This analysis is based on the reactive flow installation although our design supports proactive mode as well.

ANCS'14, October 20–21, 2014, Los Angeles, CA, USA.
Copyright 2014 ACM 978-1-4503-2839-5/14/10 ...$15.00.
http://dx.doi.org/10.1145/2658260.2658261.

formance can still be bad if the load is not distributed among these different controllers evenly. For example, if the set of switches that are connected to one controller are generating most of the traffic while the others are not, this can cause the performance to dip significantly even though there are enough controller resources in the overall system. To address this problem, ElastiCon periodically monitors the load on each controller, detects imbalances, and *automatically* balances the load across controllers by migrating some switches from the overloaded controller to a lightly-loaded one. This way, ElastiCon ensures predictable performance even under highly dynamic workloads.

Migrating a switch from one controller to another in a naive fashion can cause disruption to ongoing flows, which can severely impact the various applications running in the data center. Unfortunately, the current de facto SDN standard, OpenFlow does not provide a disruption-free migration operation natively. To address this shortcoming, we propose a new *4-phase migration protocol* that ensures minimal disruption to ongoing flows. Our protocol makes minimal assumptions about the switch architecture and is OpenFlow standard compliant. The basic idea in our protocol involves creating a single trigger event that can help determine the exact moment of handoff between the first controller and second controller. We exploit OpenFlow's "equal mode" semantics to ensure such a single trigger event to be sent to both the controllers that can allow the controllers to perform the handoff in a disruption-free manner without safety or liveness concerns.

Armed with this disruption-free migration primitive, ElastiCon supports the following three main load adaptation operations: First, it monitors the load on all controllers and periodically *load balances* the controllers by optimizing the switch-to-controller mapping. Second, if the aggregate load exceeds the maximum capacity of existing controllers, it *grows* the resource pool by adding new controllers, triggering switch migrations to utilize the new controller resource. Similarly, when the load falls below a particular level, it *shrinks* the resource pool accordingly to consolidate switches onto fewer controllers. For all these actions, ElastiCon uses simple algorithms to decide when and what switches to migrate.

We have described a preliminary version of ElastiCon in [7] where we focused mainly on the migration protocol. Additional contributions of this paper are as follows:

- We enhance the migration protocol to guarantee serializability. We show how these guarantees simplify application-specific modifications for moving state between controllers during switch migration. The serializability guarantee requires buffering messages from the switch during migration. This impacts worst-case message processing delay. Hence, we also explore the trade-off between performance and consistency.
- We propose new algorithms for deciding when to grow or shrink the controller resource pool, and trigger load balancing actions.
- We demonstrate the feasibility of ElastiCon by implementing the enhanced migration protocol and proposed algorithms. We address a practical concern of redirecting switch connections to new controllers when the controller pool is grown or away from controllers when the controller pool needs to be shrunk.
- We show that ElastiCon can ensure that performance remains stable and predictable even under highly dynamic traffic conditions.

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. Although proactive rule setup (e.g., DevoFlow [6]) can reduce controller load and flow setup time, it is not often sufficient by itself as only a small number of rules can be cached at switches, because TCAM table sizes in commodity switches tend to be small for cost and power reasons. Reactive mode allows the controller to be aware of the lifetime of each flow from setup to teardown, and hence can potentially offer better visibility than proactive mode. For low-end switches, TCAM space is a major constraint. It may be difficult to install all fine-grained microflow policies proactively. Reactive rule insertion allows such rules to be installed selectively and hence may reduce the TCAM size requirement. Thus, it is important to design the controller for predictable performance irrespective of the traffic dynamics.

Switch–controller communication. The OpenFlow protocol defines the interface and message format between a controller and a switch. When a flow arrives at a switch and does not match any rule in the flow table, the switch buffers the packet and sends a `Packet-In` message to the controller. The `Packet-In` message contains the incoming port number, packet headers and the buffer ID where the packet is stored. The controller may respond with a `Packet-Out` message which contains the buffer ID of the corresponding `Packet-In` message and a set of actions (drop, forward, etc.). For handling subsequent packets of the same flow, the controller may send a `Flow-Mod` message with an add command to instruct the switch to insert rules into its flow table. The rules match the subsequent packets of the same flow and hence allow the packets to be processed at line speed. Controller can also delete rules at a switch by using `Flow-Mod` with delete command. When a rule is deleted either explicitly or due to timeout, the switch sends a `Flow-Removed` message to the controller if the "notification" flag for the flow is set. In general, there is no guarantee on the order of processing of controller messages at a switch. Barrier messages are used to solve the synchronization problem. When the switch receives a `Barrier-Request` message from the controller, it sends a `Barrier-Reply` message back to the controller only after it has finished processing all the messages that it received before the `Barrier-Request`.

Controller architecture. The controller architecture has evolved from the original single-threaded design [10] to the more advanced multi-threaded design [21, 2, 4, 8] 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 [12, 20, 15]. 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 [12]. Hyperflow replicates the events at all distributed nodes, so that each node can process such events and update their local state [20]. [15] has further elaborated the state distribution trade-offs in SDNs. OpenDaylight [17] is a recent open source distributed SDN controller. Like ElastiCon, it uses a distributed data store for storing state information.

All existing distributed controller designs implicitly assume static mapping between switches and controllers, and hence lack the ca-

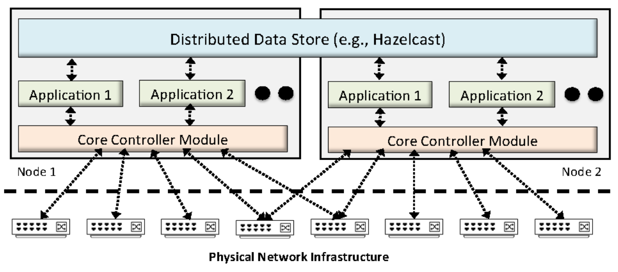

Distributed Data Store (e.g., Hazelcast)
Application 1
Application 2
Core Controller Module
Node 1
Node 2
Physical Network Infrastructure

Figure 1: Basic distributed controller architecture.

pability 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 [3], 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[2] for one controller [21], 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 and install all flow table entries reactively, 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.

3. ELASTIC CONTROLLER DESIGN

We present the design and architecture of ElastiCon, an elastic distributed SDN controller in this section. We describe the architecture of ElastiCon in three phases: First, we start with a basic distributed controller design that spreads functionality across several nodes by extending Floodlight, a Java-based open source controller [8]. We then describe the 4-phase protocol for disruption-free switch migration, which is one of the core primitives needed for implementing an elastic controller. Finally, we discuss the algorithms we use for elasticity and load adaptation in our design.

3.1 Basic Distributed Controller

The key components in our distributed controller design are shown in Figure 1. 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. Note that, for simplicity, we have omitted showing the physical topology of the network that includes the hosts and their interconnections with the switches in the network.

Typically, each switch connects to one controller. However, for fault-tolerance purposes, it may be connected to more than one controller with one master and the rest as slaves. We assume the control plane is logically isolated from the data plane, and the control plane traffic is not affected by data plane traffic. Each controller 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* provides the glue across the cluster of controllers to enable a logically centralized controller. It stores all switch-specific information that is shared among the controllers. Each controller node also maintains a TCP connection with every other controller node in the form of a full mesh. This full-mesh topology is mainly for simplicity, but as the number of controllers become exceedingly large, one may consider adding a point of indirection, similar to the route-reflector idea in scaling BGP connections in ISP networks. For today's data centers, maintaining a full mesh across a few 100 controllers does not pose any scaling concerns. A controller node uses this TCP connection for various controller-to-controller messages, such as when 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. The fact that state is maintained distributed data store makes switch migration easier and also helps fast recovery from controller failures.

3.2 4-Phase Switch Migration Protocol

If we use a single SDN controller, since all switches are always connected to this controller, there is no break in the control plane processing. Moving to a distributed controller architecture does not necessarily pose a problem so long as the switch-to-controller mapping stays static. However, such an architecture, which is employed by previously proposed distributed controllers, cannot adapt to the load imbalances caused by spatial and temporal variations in traffic conditions. Once a controller becomes overloaded, the response time for control plane messages becomes too high, thus impacting flows and applications running in the data center. We can mitigate such imbalances by dynamically shifting load between existing controllers or by adding new nodes to the controller pool. The basic granularity at which one can shift load is at a switch-level; simply migrate a switch from an overloaded controller to a lightly loaded one.

Unfortunately, there is no native support for safely migrating switches in existing de facto SDN standard, OpenFlow, without which one cannot guarantee that there is no impact to traffic during migration. In particular, there are three standard properties any migration protocol needs to provide—*liveness*, *safety* and *serializability*.

- **Liveness.** At least one controller is active for a switch at all times. Otherwise, a new flow that arrives at a switch cannot be properly routed causing disruption to that application. In addition, if a controller has issued a command to a switch, it needs to remain active until the switch finishes processing that command.
- **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 even worse, inconsistency in the distributed data store.
- **Serializability.** The controller processes events in the order in which they are transmitted by the switch; if events are processed in a different order, the controller's view of the network may be inconsistent with the state of the network. For instance, if a link goes down and comes back up, the switch will generate a port status down message followed by a port status up. However, if these events are processed in the wrong order, the controller may assume that the link is permanently down.

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

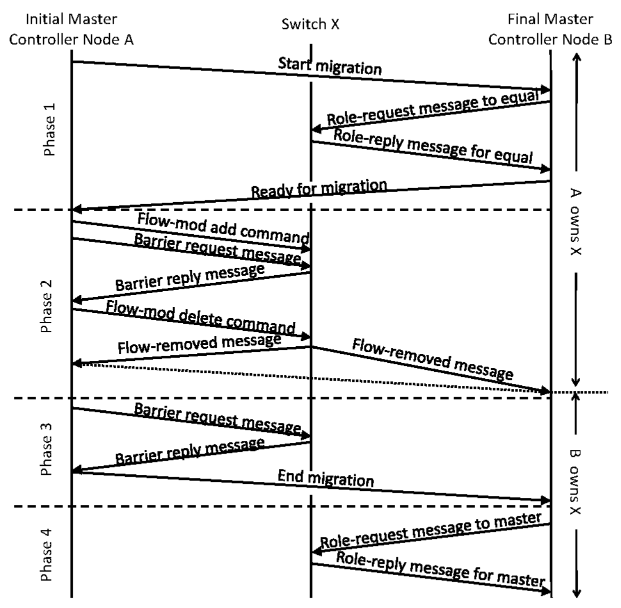


Figure 2: Message exchanges for switch migration.

Now, consider the following naive protocol that OpenFlow readily provides: The target controller can be first put in the slave mode for the switch (see Section 4 for implementation details). The target controller then simply sends a `Role-Request` message to the switch indicating that it wants to become the master. The switch would set that controller as the master and the previous master as slave. Such a naive and intuitive protocol can cause serious disruption to traffic since it 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.

In our protocol design, we assume we cannot modify the switch. 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. This assumption, which is clearly logical, is required for our protocol to work; allowing arbitrary reordering of messages across two controllers will make an already hard problem significantly harder. For ease of exposition, we use X to denote the switch, which is being migrated from initial controller A to target controller B. We first outline the key ideas that provide the desired guarantees and then describe the protocol in detail.

Liveness. To guarantee liveness, we first transition the target controller B to equal mode. After that, we transition initial controller A from master to slave mode and then transition controller B to master mode. This ensures guarantees liveness since at least one controller is active (master or equal mode) at a time.

Safety. Using an intermediate equal mode for the controller B solves the liveness problem but it may violate the safety property since both controllers may process messages from the switch causing inconsistencies and duplicate messages. To guarantee safety, we 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 entry into the switch and then remove the flow entry, which will provide a single `Flow-Removed` event to both the controllers to signal handoff.

Serializability. To guarantee serializability, the controller A should complete processing its last message before the controller B can process its first message. However, the first message for the B may arrive before A completes processing its last message. So, we cache messages at B until the A has finished processing its last message and committed it to the switch.

Our protocol operates in four phases described below (shown in Figure 2). We now describe each phase in detail and highlight a trade-off between performance and serializability.

Phase 1: Change role of the target to equal. In the first phase, target B's role is first changed to equal mode for the switch X. Initial master A initiates this phase by sending a start migration message to B using a proprietary message 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 control messages (e.g., `Packet-In`) from the switch, but ignores them and does not respond.

Phase 2: Insert and remove a dummy flow. To determine an exact instant for the migration, A sends a dummy (but well-known) `Flow-Mod` command to X to add a new flow table entry that does not match any incoming packet. Then, it sends another `Flow-Mod` command to delete this flow table entry; in response, the switch sends a `Flow-Removed` message to both controllers since B is in the equal mode. This `Flow-Removed` event signals a handoff of switch X from A to B, and henceforth, only B will process all messages transmitted by switch. Here, our assumption that both controllers in equal mode receive messages from the switch in the same order is needed to guarantee the safety property. 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.

Although B processes all messages after the `Flow-Removed` message, it does not do so immediately. It caches all the messages after the `Flow-Removed` message and begins processing them in the next phase. This is needed to guarantee the serializability property. Processing of messages from the north-bound interface can continue uninterrupted.

Phase 3: Flush pending requests with a barrier. Now, B has taken over responsibility of switch X, but A has not detached from X yet. 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. Controller A processes all messages that arrived before `Flow-Removed` and transmits their responses. Then, it transmits a `Barrier-Request` and waits for the `Barrier-Reply`. Receiving a `Barrier-Reply` from switch X indicates that X has finished processing all messages that it received before the `Barrier-Request` messages. So, only after receiving the `Barrier-Reply` message, controller A signals "end migration" to the final master B. The "end migration" message is a signal to B that A has finished processing all its messages and committed them to the switch. Once B receives the "end migration" message, it processes all the cached messages in the order that they were received. Note that delay in end migration message can potentially cause message processing latency at B. This delay can be avoided if we do not need to guarantee serializability. In that case B can start processing `Packet-In` messages right after receiving `Flow-Removed`.

Phase 4: Make target controller final master. Here, A would have already detached from X and has signaled to B to become the new master, which it does by 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 after which it processes all messages from the switch.

Performance-Serializability Trade-off. Buffering messages from the switch at the end of phase 2 is needed to guarantee serializability. It ensures that B begins processing messages only after A has completed processing messages before the `Flow-Removed` message. The duration for buffering messages will depend on the network latencies, message loss ratio, controller processing times, etc. In our experiments, we observed that messages were never buffered for more than 50msec. However, the worst case will depend on many network characteristics and may be larger. While buffering is needed to guarantee serializability, it has two undesirable side-effects. First, the controller will be unable to respond to events from the switch while messages are being buffered. Second, buffered messages will be processed late and may be irrelevant by the time they are processed. So, the network operator should choose between two configurations of the migration protocol depending on network characteristics and application requirements. The "consistency configuration" buffers messages as described above and provides all three guarantees. The "performance configuration" does not buffer messages. It does not provide serializability but responds faster to switch events during migration.

3.3 Application State Migration

Safety, liveness and serializability guarantees of the migration protocol simplify controller application changes to support switch migration. The three guarantees together ensure that applications do not miss any asynchronous events and do not have to check for duplicate or reordered asynchronous messages from the switch before processing them. We describe the modifications to the applications and their interface with the core controller module below. We have implemented them for the routing applications in ElastiCon.

We added two methods to the interface between the core controller module and each application module. The first method, named "switch_emigrate", is invoked at the initial master controller (controller A in the above example). The core controller module invokes this method for each application after it has finished processing all messages before the `Flow-Removed` message from the switch. The method returns after the application has flushed all switch-specific state to the distributed data store. Applications also stop any switch-specific execution (like timers). The controller sends the "end migration" message only after all applications execute their "switch_emigrate" method. The second method, "switch_immigrate", is invoked at the target master controller (controller B in the above example) for each application. It is invoked after the controller receives the "end migration" message. Each application reads switch-specific state from the distributed data store to populate local data structures and starts switch-specific execution. The distributed data store should guarantee that the controller reads the state written in the "switch_emigrate" method earlier. The

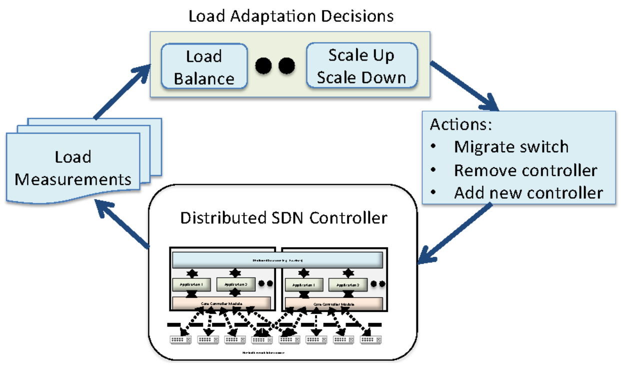


Figure 3: Load adaptation in ElastiCon.

controller starts processing cached asynchronous messages after all applications have executed their "switch_immigrate" methods.

State-transfer between applications can also be performed over TCP connections between applications instead of using the distributed data store. The above design simplified our implementation since we reused the interface between the application and the distributed data store. Using this disruption-free migration protocol as a basic primitive, we now look at load adaptation aspects of ElastiCon.

3.4 Load Adaptation

There are three key operations we envision for load adaptation in ElastiCon. If the aggregate traffic load is greater (smaller) than aggregate controller capacity, we need to *scale up (down)* the controller pool. In addition, we need to periodically *load balance* the controllers by migrating switches to newer controllers to adapt to traffic load imbalances. We show our basic approach to achieve this in Figure 3. It consists of three steps:

- Periodically collect load measurements at each controller node.
- Determine if the current number of controller nodes is sufficient to handle the current load. If not, add or remove controller nodes. In addition, if any controller is getting overloaded, but aggregate load is within the capacity, we need to trigger load balancing actions.
- Finally, adjust the switch to controller mapping by adding or removing the controllers and triggering switch migrations as needed.

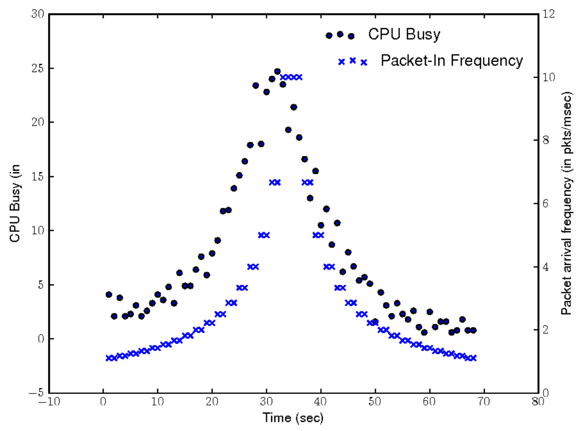


Figure 4: CPU vs. packet frequency.

3.4.1 Load Measurement

The most direct way to measure the load on a controller is by sampling response time of the controller at the switches. This response time will include both computation and network latency. However, switches may not support response time measurements, since that requires maintaining some amount of extra state at the switches that may or may not be feasible. Since the controller is more programmable, ElastiCon maintains a load measurement module on each controller to periodically report the CPU utilization and network I/O rates at the controller. Our experiments show that the CPU is typically the throughput bottleneck and CPU load is roughly in proportion to the message rate (see Figure 4). The module also reports the average message arrival rate from each switch connected to the controller. This aids the load balancer in first dissecting the contribution of each switch to the overall CPU utilization, and helps making optimal switch to controller mapping decisions. We assume that the fraction of controller resources used by a switch is proportional to its fraction of the total messages received at the controller, which is typically true due to the almost linear relationship between throughput and messages. The load measurement module averages load estimates over small time intervals (we use three seconds) to avoid triggering switch migrations due to short-term load spikes.

Algorithm 1 Load Adaptation Algorithm

```
while True do
    GET_INPUTS()
    $migration\_set \leftarrow$ DOREBALANCING()
    if $migration\_set == NULL$ then
        if DORESIZING() then
            if CHECKRESIZING() then
                $migration\_set \leftarrow$ DOREBALANCING()
            else
                REVERTRESIZING()
            end if
        end if
    end if
    EXECUTE_POWER_ON_CONTROLLER()
    EXECUTE_MIGRATIONS($migration\_set$)
    EXECUTE_POWER_OFF_CONTROLLER()
    SLEEP(3)
end while
```

3.4.2 Adaptation Decision Computation

The load adaptation algorithm determines if the current distributed controller pool is sufficient to handle the current network load. It sets a high and low thresholds to determine whether the distributed controller needs to be scaled up or down. Difference between these thresholds should be large enough to prevent frequent scale changes. Then, the algorithm finds an optimal switch to controller mapping constrained by the controller capacity while minimizing the number of switch migrations. Some CPU cycles and network bandwidth should also be reserved for switches connected to a controller in slave mode. Switches in slave mode impose very little load on the controller typically, but some headroom should be reserved to allow switch migrations.

While one can formulate and solve an optimization problem (e.g., linear program) that can generate an optimal assignment of switch-to-controller mappings, it is not clear such formulations are useful for our setting in practice. First, optimal balancing is not the primary objective as much as performance (e.g., in the form of response time). Usually, as long as a controller is not too overloaded, there is not much performance difference between different CPU utilization values. For example, 10% and 20% CPU utilization results in almost similar controller response time. Thus, fine-grained optimization is not critical in practice. Second, optimal balancing may result in too many migrations that is not desirable. Of course, one can factor this in the cost function, but then it requires another (artificial) weighting of these two functions, which then becomes somewhat arbitrary. Finally, optimization problems are also computationally intensive and since the traffic changes quickly, the benefits of the optimized switch-controller mapping are short-lived. So, a computationally light-weight algorithm that can be run frequently is likely to have at least similar if not better performance than optimization. Perhaps, this is the main reason why distributed resource management (DRM) algorithms used in real world for load balancing cluster workloads by migrating virtual machines (VMs) do not solve any such optimization problems and rely on a more simpler feedback loop [11]. We adopt a similar approach in our setting.

Our load-adaptation decision process proceeds in two phases, as shown in Algorithm 1. First, during the rebalancing step the load adaptation module evenly distributes the current load across all available controllers. After rebalancing, if the load on one or more controllers exceeds the upper (or lower) threshold, the load adaptation module grows (or shrinks) the controller pool.

Input to the Algorithm. A load adaptation module within ElastiCon periodically receives inputs from the load measurement module on each controller. The input contains the total CPU usage by the controller process in MHz. It also contains a count of the number of packets received from each switch of which that controller is the master. The packet count is used to estimate the fraction of the load on the controller due to a particular switch. The load adaptation module stores a moving window of the past inputs for each controller. We define utilization of a controller as the sum of the mean and standard deviation of CPU usage over the stored values for that controller. The rebalancing and resizing algorithms never use instantaneous CPU load. Instead they use CPU utilization to ensure that they always leave some headroom for temporal spikes in instantaneous CPU load. Also, the amount of headroom at a controller will be correlated to the variation in CPU load for that controller.

Output of the Algorithm. After processing the inputs, the load adaptation module may perform one or more of the following actions: powering off a controller, powering on a controller, or migrating a switch from one controller to another.

Main Loop of the Algorithm. First, the load adaptation module receives the inputs from all controllers and augments them to its stored state. All functions except the EXECUTE_* functions only modify this stored state and they do not affect the state of the controllers. After that, the EXECUTE_* functions determine the changes to the stored state and send migration and power on/off commands to the appropriate controllers.

There are two main subroutines in the rebalancing algorithm: DOREBALANCING and DORESIZING. DOREBALANCING distributes the current load evenly among the controllers. DORESIZING adds or removes controllers accordingly to the current load. DORESIZING is invoked after DOREBALANCING since resizing the controller pool is a more intrusive operation than rebalancing the controller load, and hence should be avoided when possible. Although one can estimate average load per controller without actually doing rebalancing and then determine whether resizing is needed or not, this often suffers from estimation errors.

If the first invocation of DOREBALANCING generates any migrations, we execute those migrations and iterate over the main loop again after 3 seconds. If there are no migrations (indicating that the controllers are evenly loaded), ElastiCon generates resizing (i.e., controller power on/off) decisions by invoking DORESIZING. The power off decision needs to be verified to ensure that the switches connected to the powered off controller can be redistributed among the remaining controllers without overloading any one of them. This is done in the CHECKRESIZING function. This function uses a simple first-fit algorithm to redistribute the switches. While other more sophisticated functions can be used, our experience indicates first-fit is quite effective most of the time. If this function fails, the (stored) network state is reverted. Otherwise, ElastiCon calls DOREBALANCING to evenly distribute the switch load. Finally, the EXECUTE_* functions implement the state changes made to the network by the previous function calls. Since a migration changes the load of two controllers, all stored inputs for the controllers involved in a migration are discarded. The main loop is executed every 3 seconds to allow for decisions from the previous iteration to take effect.

Algorithm 2 The rebalancing algorithm

```
procedure DOREBALANCING()
    $migration\_set \leftarrow NULL$
    while True do
        $best\_migration \leftarrow$ GET_BEST_MIGRATION()
        if $best\_migration.std\_dev\_improvement \geq THRESHOLD$ then
            $migration\_set$.INSERT($best\_migration$)
        else
            return $migration\_set$
        end if
    end while
end procedure
```

Rebalancing. The rebalancing algorithm, described in Algorithm 2, tries to balance the average utilization of all controllers. We use the standard deviation of utilization across all the controllers as a balancing metric. In each iteration, it calls the GET_BEST_MIGRATION function to identify the migration that leads to the most reduction in standard deviation of utilization across controllers. The GET_BEST_MIGRATION function tries every possible migration in the network and estimates the standard deviation of utilization for each scenario. It returns the migration which has the smallest estimated standard deviation. To estimate the standard deviation, this function needs to know the load imposed by every switch on its master controller. Within each scenario, after a hypothetical migration, the function calculates the utilization of each controller by adding the fractional utilizations due to the switches connected to it. It then finds the standard deviation across the utilization of all the controllers. If reduction in standard deviation by the best migration it finds exceeds the minimum reduction threshold, ElastiCon adds that migration to the set of migrations. If no such migration is found or the best migration does not lead to sufficient reduction in standard deviation, it exits.

Resizing. The resizing algorithm, shown in Algorithm 3, tries to keep the utilization of every controller between two preset high and low thresholds. Each invocation of the resizing algorithm generates either a power on, or power off, or no decision at all. The resizing algorithm is conservative in generating decisions to prevent oscillations. Also, it is more aggressive in power on decisions than power off. This is because when the utilization exceeds the high threshold, the network performance may suffer unless additional controllers are put in place quickly. However, when the utilization goes below the low threshold, network performance does not suffer. Removing controllers only consolidates the workload over fewer controllers sufficient to handle existing traffic conditions, mainly for power

Algorithm 3 The resizing algorithm

```
procedure DORESIZING()
    for all $c$ in $controller\_list$ do
        if $c.util \geq HIGH\_UTIL\_THRESH$ then
            SWITCH_ON_CONTROLLER()
            return True
        end if
    end for
    $counter \leftarrow 0$
    for all $c$ in $controller\_list$ do
        if $c.util \leq LOW\_UTIL\_THRESH$ then
            $counter \leftarrow counter + 1$
        end if
    end for
    if $counter \geq 2$ then
        SWITCH_OFF_CONTROLLER()
        return True
    else
        return False
    end if
end procedure
```

and other secondary concerns than network performance. Thus, we generate a power on decision when any controller exceeds the high threshold while requiring at least two controllers to fall below the low threshold for generating a power off decision. Triggering a decision when just one or two controllers cross the threshold might seem like we aggressively add or remove controller. But, our decisions are quite conservative because the resizing algorithm is executed only when the load is evenly distributed across all controllers. So, if a controller crosses the threshold, it indicates that all controllers are close to the threshold.

3.4.3 Extending Load Adaptation Algorithms

The load adaptation algorithms described above can be easily extended to satisfy additional requirements or constraints. Here we describe two such potential extensions to show the broad applicability and flexibility of the algorithm.

Controller Location. To reduce control plane latency, it may be better to assign a switch to a closeby controller. We can accommodate this requirement in ElastiCon by contraining migrations and controller additions and removals. To do so, in every iteration of the rebalancing algorithm (Algorithm 2), we consider only migrations to controllers close to the switch. This distance can be estimated based on topology information or controller to switch latency measurements. If the operator wants to set switch-controller mapping based on physical distance (in number of hops), he/she can use the network topology. The operator should use latency measurements when he/she wants to set switch-controller mapping based on logical distance (in milliseconds). Similarly, in the resizing algorithm (Algorithm 3), the new controllers added should be close to the overloaded controllers so that switches can migrate away from the overloaded controller. The first-fit algorithm used in CHECKRESIZING function should also be modified such that a switch can only "fit" in a closeby controller.

Switch Grouping. Assigning neighboring switches to the same controller may reduce inter-controller communication during flow setup and hence improve control plane efficiency. Graph partitioning algorithms can be used to partition the network into switch groups; and the result can be fed into ElastiCon. ElastiCon can be modified to treat each group as a single entity during migration and resizing, so that the switches of the same group are always controlled by the same controller except for short intervals during migration. The load measurements module should be modified to combine load readings of switches of a group and present it as a single entity to the load adaptation algorithm. When the rebalancing algorithm determines that the entity needs to be migrated, the EXECUTE_* functions should migrate all the switches of the group.

3.4.4 Adaptation Action

Following the adaptation decision, adaptation actions are executed to transform the network configuration (i.e., switch to controller mapping). A switch is migrated to a former slave by following the steps in our 4-phase migration protocol described before. In case of controller addition or removal, one or more switches may need to be reassigned to new master controllers that they are not currently connected to. This can be done by replacing one of the existing slave controllers' IP address of the switch with that of the new controller using the `edit-config` operation of OpenFlow Management and Configuration Protocol [1]. 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 (discussed in Section 4).

4. IMPLEMENTATION

In this section, we present further details on how we implement ElastiCon by modifying and adding components to the centralized Floodlight controller.

Distributed Data Store. We use Hazelcast to implement the distributed data store. Although other NoSql databases may have also worked here, we find Hazelcast a good choice due to its performance and flexibility. Hazelcast provides strong consistency, transaction support, and event notifications. Its in-memory data storage and distributed architecture ensures both low latency data access and high availability. Persistent data can be configured to write to disk. It is written in Java, which makes it easy for integration with Floodlight. We include the Hazelcast libraries in the Floodlight executable. The first Hazelcast node forms a new distributed data store. Subsequently, each Hazelcast node is configured with the IP addresses and ports of several peers. At least one of the peers needs to be active for the new node to join the distributed data store.

Controller. When a controller boots up, it publishes its own local data and retrieves data of other controllers by accessing Hazelcast. One such data is the IP address and TCP port of each controller needed for inter-controller communication. This allows the controllers to set up direct TCP connections with each other, so that they can invoke each other to set up paths for flows.

The switch to master controller mapping is also stored in Hazelcast using the unique switch datapath-id as the key. We have modified the core controller in Floodlight to allow a controller to act in different roles for different switches. The initial switch to master mapping can be determined in one of two ways. In the first method, the load adapter module running in the controller (described later) reads in the mapping from a configuration file and stores the information in Hazelcast. We also implement an ad hoc strategy by letting the controllers try to acquire a lock in Hazelcast when a switch connects to them. Only one controller can succeed in acquiring the lock; it then declares itself as the master for the switch.

Load Adaptation Module. The load measurement module is integrated into the controller. We use SIGAR API [19] to retrieve the CPU usage of the controller process. We enhanced the REST API of the controller to include CPU usage queries. The adaptation decision algorithm run on a separate host. It communicates with all controllers over the REST API. It requires the REST port and IP address of one of the controllers. Using that, it queries the controller for the IP address and REST port of all other controllers and switch-to-controller mappings of all switches in the network. In each iteration, the program queries the CPU usage information from each controller and sends migration requests to the master controller of a switch when the switch needs to be migrated.

Adding and Removing Controllers. Migration of a switch to a newly connected controller is done in two steps. First, we replace the IP address and TCP port number of one of the slave controllers of the switch with those of the new controller. This can be done by using the `edit-config` operation of OpenFlow Management and Configuration Protocol [1]. 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.

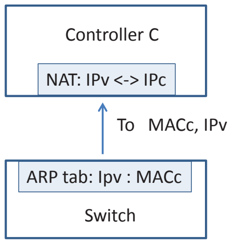


Figure 5: Controller virtual IP address binding

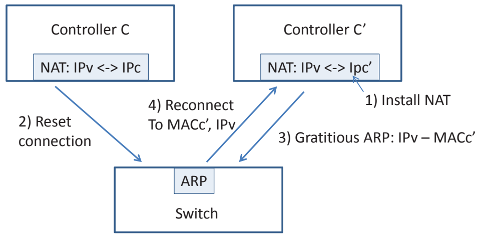


Figure 6: Controller binding change

If a switch does not support updating controller IP addresses at runtime, we can use the following procedure as a workaround, which is suitable when the control plane is configured to use the same layer 2 network (e.g., on the same VLAN). All switches are configured to use a set of virtual controller IP addresses, which will be mapped to the real controller IP addresses at runtime according to load condition. Such mapping can be realized by using ARP and Network Address Translation (NAT), as shown in Figure 5. When the virtual controller IP address ip_v for a switch is mapped to controller C's IP address ip_c, we use gratuitous ARP to bind the MAC address of the controller C with ip_v, so that the packets to ip_v can reach controller C. At controller C, we do NAT from ip_v to ip_c, so that the packets can be handled by the controller transparently.

Figure 6 shows how such binding can be changed when we need to replace controller C with controller C'. We first send a TCP reset message from C to disconnect the switch from the controller, and then use gratuitous ARP to bind MAC address of C' with ip_v. Note that connection reset to C is only done when C is not a master controller to avoid disruption in normal switch operation. When the switch tries to reconnect to ip_v, the message will reach C' instead of C. We then do a NAT from ip_v to $ip_{c'}$ at controller C' as before. Note that if the gratuitous ARP does not reach the switch before the reconnection request is sent, controller C simply rejects the reconnection request and the switch ultimately gets connected to controller C'.

5. 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.

5.1 Enhanced Mininet Testbed

Our experimental testbed is built on top of Mininet [14], which emulates a network of Open vSwitches [18]. 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 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 impact 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.

5.2 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 10,000 back-to-back `Packet-In` messages and plot the throughput of ElastiCon with varying number of controller nodes (Figure 7(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 7(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 nodes.

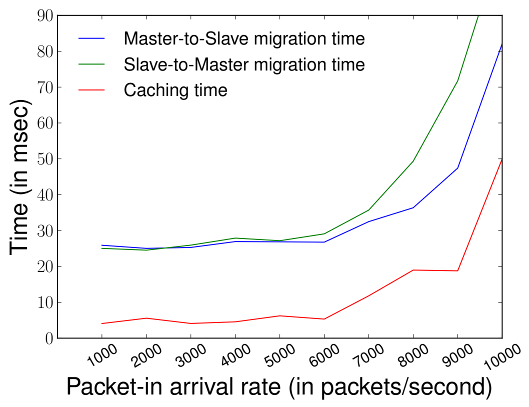
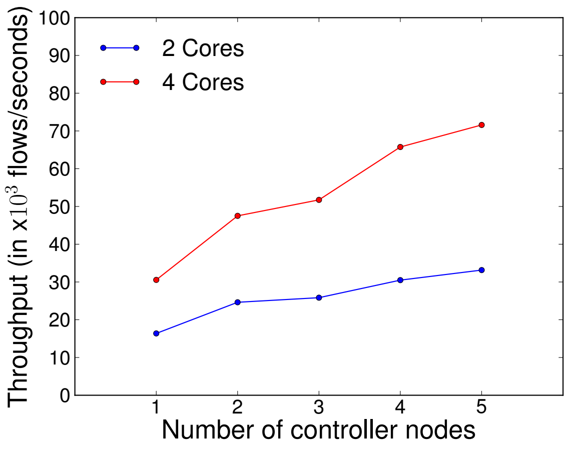


Figure 8: Migration time

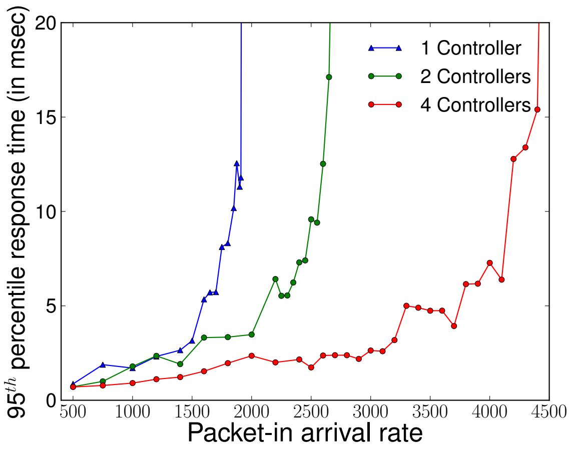
Migration time. The time taken to migrate a switch is critical for the load balancing protocol to work efficiently. We define migration time for controller A as the time between sending the "start migration" message and "end migration" message. We define migration time for controller B as the time between receiving the "start migration" and sending the `Role-Request` to change to master. In a network with 3 controllers, we perform 200 migrations and observe the migration time for each migration at both controllers. We also observe the time for which controller B caches messages from the switch. We plot the 95th percentile of the migration and caching times in Figure 8. The plot shows that the migration time is minimal (few tens of milliseconds) and increases marginally as the load on the controller increases. The caching time is even smaller (around 5ms). This keeps memory usage of the message cache small (few KBs).

Automatic rebalancing under hot-spot traffic. We use a N=4 fat tree to evaluate the effect of the automatic load balancing algorithm. Three of the four pods of the fat tree are evenly loaded, while the flow arrival rate in the fourth pod is higher than that in the other three. We configure ElastiCon with four controllers, one assigned to all the switches of each pod. The master controller of the fourth pod is obviously more heavily loaded than the other three. Figure 9(a) shows the 95th percentile of the response time of all `Packet-In` messages before and after rebalancing. The `Packet-In` message rate in the fourth pod is varied on the X-axis. We truncate the y-axis at 20ms, so a bar at 20ms is actually much higher.

We observe that as traffic gets more skewed (i.e., the `Packet-In` rate in the fourth pod increases), we see a larger benefit by doing rebalancing corresponding to the 65-75% bars. At 70-80% hot-spot, the system is unstable. The 95th percentile can be arbitrarily high depending on the amount of time the experiment is run before rebalancing, since the one of the controllers is overloaded (i.e., the `Packet-In` rate exceeds the saturation throughput). At 80% hot-

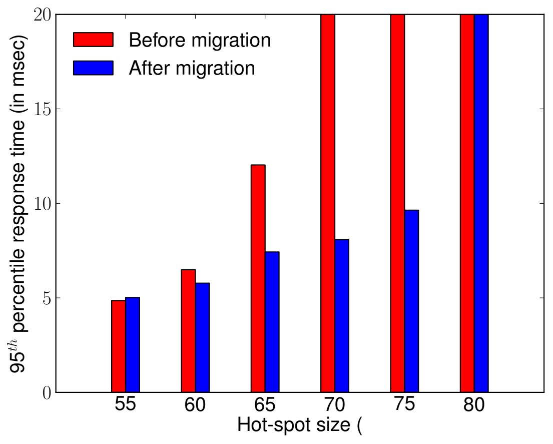


(a) Controller throughput

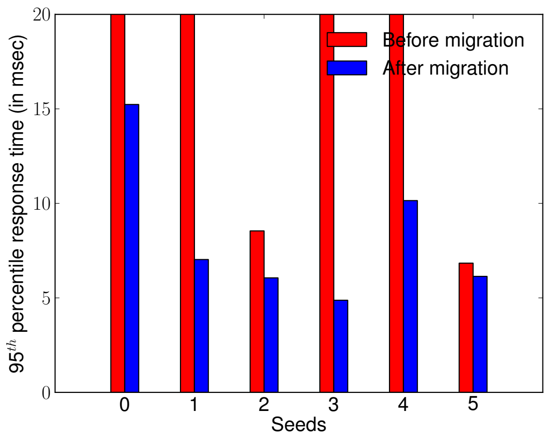


(b) Response time

Figure 7: Performance with varying number of controller nodes.



(a) Single hot-spot traffic pattern



(b) Pareto distributed traffic pattern

Figure 9: Benefit of automatic rebalancing. We truncate the y-axis at 20ms. So a bar at 20ms is actually much higher.

spot, rebalancing by itself does not help as seen by the blue bar exceeding 20ms since there is no way to fit the workload among existing controllers.

Automatic rebalancing under Pareto distribution. We also evaluate the benefit of the rebalancing algorithm in the case where multiple hot spots may appear randomly following a Pareto distribution. As before, we use a N=4 fat tree with 4 controllers. The network generates 24,000 `Packet-In` messages per second. The message arrival rate is distributed across all the switches in the network using a Pareto distribution. We repeat the traffic pattern with 6 different seeds. We start with a random assignment of switches to controllers and apply the rebalancing algorithm. Figure 9(b) shows the 95th percentile response time with random assignment and with rebalancing.

Since a Pareto distribution is highly skewed, the improvement varies widely depending on the seed. If the distribution generated by a seed is more skewed, rebalancing is likely to deliver better response times over a random switch to controller assignment. But, if the Pareto distribution evenly distributes traffic across switches (see seeds #2 and #5), random assignment does almost as well as rebalancing. In the Figure 9(b), we can observe that for all cases, rebalancing at least ensures that there is no controller that is severely overloaded while at least in four cases, random load balancing led to significant overload as evidenced by the high red bar.

Effect of resizing. We demonstrate how the resizing algorithm adapts the controllers as the number of `Packet-In` messages increases and decreases. We begin with a network with 2 controllers and an aggregate `Packet-In` rate of 8,000 packets per second. We increase the `Packet-In` rate in steps of 1,000 packets per second every 3 minutes until it reaches 12,000 packets per second. We then reduce it in steps of 1,000 packets per second every 3 minutes until it comes down to 6,000 packets per second. At all times, the `Packet-In` messages are equally distributed across switches, just for simplicity. We observe 95th percentile of the response time at each minute for the duration of the experiment. We also note the times at which ElastiCon adds and removes controllers to adapt to changes in load. The results are shown in Figure 10.

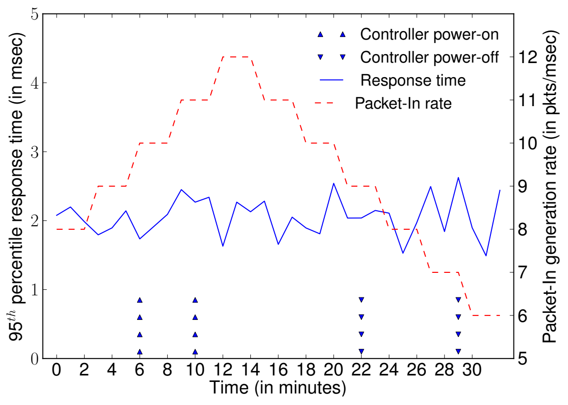


Figure 10: Growing and shrinking ElastiCon

We observe that ElastiCon adds a controller at the 6th and 10th minute of the experiment as the `Packet-In` rate rises. It removes controllers at the 22nd and 29th minute as the traffic falls. Also, we observe that the response time remains around 2ms for the entire duration of the experiment although the `Packet-In` rate rises and falls. Also, ElastiCon adds the controllers at 10,000 and 11,000 `Packet-In` messages per second and removes them at 9,000 and 7,000 `Packet-In` messages per second. As described earlier, this is because ElastiCon aggressively adds controllers and conservatively removes them.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented our design of ElastiCon, a distributed elastic SDN controller. We designed and implemented algorithms for switch migration, controller load balancing and elasticity which form the core of the controller. We enhanced Mininet and used it to demonstrate the efficacy of those algorithms. Our current design does not address issues caused by failures, although we believe fault tolerance mechanisms can easily fit into this architecture. This may require running three or more controllers in equal role for each switch and using a consensus protocol between them to ensure there is always at least one master even if the new master crashes. We also plan to study the impact of application data sharing patterns on switch migration and elasticity. In addition, we plan to consider other factors like controller placement and controller performance isolation in a multi-tenant data center.

7. ACKNOWLEDGEMENTS

We thank the anonymous reviewers and Alex Bachmutsky, our shepherd, for their comments that helped improve the paper. This work was supported in part by NSF grants CNS-1054788, IIS-1017898.

8. REFERENCES

[1] OpenFlow Management and Configuration Protocol (OF-Config 1.1).
[2] Beacon. `openflow.stanford.edu/display/Beacon/Home`.
[3] Benson, T., Akella, A., and Maltz, D. Network traffic characteristics of data centers in the wild. In *IMC* (2010).
[4] Cai, Z., Cox, A. L., and Ng, T. S. E. Maestro: A system for scalable OpenFlow control. Tech. rep., CS Department, Rice University, 2010.
[5] Casado, M., Freedman, M. J., and Shenker, S. Ethane: Taking Control of the Enterprise. In *ACM SIGCOMM* (2007).
[6] Curtis, A., Mogul, J., Tourrilhes, J., Yalagandula, P., Sharma, P., and Banerjee, S. DevoFlow: Scaling Flow Management for High-Performance Networks. In *ACM SIGCOMM* (2011).
[7] Dixit, A., Hao, F., Mukherjee, S., Lakshman, T., and Kompella, R. Towards an Elastic Distributed SDN Controller. In *HotSDN* (2013).
[8] Floodlight. `floodlight.openflowhub.org`.
[9] Greenberg, A., Hjalmtysson, G., Maltz, D., Myers, A., Rexford, J., Xie, G., Yan, H., Zhan, J., and Zhang, H. A clean slate 4D approach to network control and management. In *SIGCOMM CCR* (2005).
[10] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. NOX: Towards an Operating System for Networks. In *SIGCOMM CCR* (2008).
[11] Gulati, A., Anne Holler, A. M. J., Shanmuganathan, G., Waldspurger, C., and Zhu, X. VMware Distributed Resource Management: Design, Implementation and Lessons Learned.
[12] Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., and Shenker, S. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI* (2010).
[13] Lakshman, T., Nandagopal, T., Ramjee, R., Sabnani, K., and Woo, T. The SoftRouter Architecture. In *ACM HOTNETS* (2004).
[14] Lantz, B., Heller, B., and McKeown, N. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *HotNets* (2010).
[15] Levin, D., Wundsam, A., Heller, B., Handigol, N., and Feldmann, A. Logically Centralized? State Distribution Trade-offs in Software Defined Networks. In *HotSDN* (2012).
[16] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR* (2008).
[17] OpenDaylight. `http://www.opendaylight.org/`.
[18] Open vswitch. `openvswitch.org`.
[19] Hyperic SIGAR API. `http://www.hyperic.com/products/sigar`.
[20] Tootoonchian, A., and Ganjali, Y. HyperFlow: A Distributed Control Plane for OpenFlow. In *INM/WREN* (2010).
[21] Tootoonchian, A., Gorbunov, S., Ganjali, Y., Casado, M., and Sherwood, R. On Controller Performance in Software-Defined Networks. In *HotICE* (2012).