# An Elastic Distributed SDN Controller

Xinyu Zhang
A53095838
xiz368@eng.ucsd.edu

Liqiong Yang
A53076313
liy007@eng.ucsd.edu

Guo Li
A53071472
gul027@eng.ucsd.edu

## ABSTRACT

Software Defined Networking(SDN) has become a popular paradigm for centralized control and management in many modern networking scenarios. However, for large data centers that have hundreds of thousands of servers and few thousands of switches, single controller mode will make system suffer from scalability and reliability issues. Previous researchers have proposed distributed controller architectures to address these problems, but the key limitation of these works is the mapping between a switch and a controller is static, which may result in imbalanced load among controllers.

In this paper we implemented a switch migration logic that can migrate a switch from a heavily-loaded controller to a lightly-loaded one. The migration decision is made by a monitor based on the network traffic. Our result shows improvement on controller's throughput and response time.

## 1. INTRODUCTION

*Software Defined Networking* (SDN) provides a centralized control panel, which allows the network to be programmed by the application and controlled from one central entity. This enables easy management and fast innovation. However, for large data centers that have hundreds of thousands of servers and few thousands of switches, single controller mode will make system suffer from scalability and reliability issues.

Recent papers have proposed the design of distributed SDN controller, which focus on how to implement a distributed fashion of controller and maintain consistency of the system. For these systems, the mapping between a controller and switch is statically configured. This means each controller will be responsible for a certain set of switches, and this set will not change no matter how the network traffic changes. Real data center network or enterprise network, however, exhibits a significant traffic variation. Some switches may receive a lot of packages at one time while very few packages at another time, when they may become relatively idle.

Since everytime when a flow arrives at a switch, the switch will send message to controller to ask for actions, e.g. drop or forward etc. The network vairation among switches will result in controllers' imbalanced workload. Some controllers remain overutilized while others remain underutilized. This will undermine the overall SDN performance. Besides, it is impossible to predict which switches and controllers would be over-loaded and pre-define the static flow table in controller.

As a result, for distributed SDN controller, there should be a migration logic that, when load imbalance occurres, system can automatically migrate a switch from a heavily-loaded controller to a lightly-loaded one. There should also be a monitoring logic that keeps monitoring the network traffic information, decides when to trigger migration, and what the migration source controller and destination controller are. In this way, the network could dynamically adapt to variated network traffic and keep adjusting itself to ensure the best performance.

Migrating a switch from one controller to another is not a trivial task, a naive solution will disrupt the ongoing flows and severely impact the applications running on the network. An ideal migration logic must ensure that it can happen at any moment: it doesn't need to wait for certain network condition; and it should be able to be completed seamlessly: it will not influence ongoing flows, and network application won't sense this operation.

In this project we focused on implementing this switch migration logic. We used Mininet to simulate a network environment, and used Ryu SDN framework to implement all the controller and monitor logic. We implemented a 4-phase migration protocol and then evaluated

the controller's response time and throughput. We also evaluated response time variation before and after migration. The idea and algorithm in the project is based on the 2014 SIGCOMM workshop paper "Towards an Elastic Distributed SDN Controller" [3] .

## 2. RELATED WORK

In recent years, the controller architecture has evolved from the original single threaded design [5] to multi-threaded design [8] [1] [2] [4] . However, the single-controller systems still have limits on scalability and vulberability. Distributed-controller system are come up in several researches [6] [8] [7] . Onix [6] uses a transactional database for persistent but less dynamic data and memory-only DHT for data that changes quickly but does not require consistency. It is a control plane platform designed to enable scalable control applications. Its main contribution is to abstract away the task of network state distribution from applications and provide them with a logical view of the network state. Onix provides a general API for control applications, while allowing them to make their own trade-offs among consistency, durability, and scalability. Hyperflow is the first distributed event-based control plane for OpenFlow. It enables network operators deploy any number of controllers and it keeps the network control logic centralized and localizes all decisions to each controller to minimize control plane response time. Hyperflow replicates the events at all distributed nodes .

All existing distributed controller designs implicitly assume static mapping between switches and controllers, and lack the capability of dynamic load adaptation and elasticity.

## 3. SYSTEM DESIGN

### 3.1 Background

The OpenFlow network consists of switches and a central controller. Packet fowarding rules are stored in each switch's flow table. When a flow arrives at a switch and does not match with any rule in the flow table, the switch buffers this 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. As soon as controller receives that `Packet-In` message, it may respond with a `Packet-Out` message which contains the buffer ID of the corresponding `Packet-In` message and a set of actions, e.g. 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 the switch's flow table. Besides updating the flow table, controller can also delete rules from a switch's flow table by using `Flow-Mod` with delete command. When a rule is deleted, the switch sends a `Flow-Removed` message to the controller.

### 3.2 General Design

On top of the original SDN architecture, we implement a monitor that can evaluate each controller's traffic burden. Besides processing messages from switch, our controller will keep collecting traffic information from switch and periodically send traffic data to monitor.

The monitor will be responsible for monitoring the network status, when it discovers that controllers' burdens are not balanced, it will migrate one or more switches from heavily-loaded one to lightly-loaded one. During the migration, all the ongoing flows shouldn't be disrupted. These requirements are satisfied by a carefully designed Migration Protocol which is explained in Section 4.

### 3.3 Controller

In distributed controller SDN, the key components are a cluster of autonomous controller nodes that coordinate with each other to provide a consistent control logic for the entire network. Besides the controllers, there should also be a physical network infrastructure that includes switches and links. Typically, each switch connects to one controller. However, for fault-tolerance purposes, one switch should connect to more than one controllers with one master and the rest as slaves. In this way, migration is a process of changing one controller's mode from master to slave and vice versa.

As described in former section, the burden of a controller comes from `Packet-In` messages. The controller constantly calculates the number of `Packet-In` messages as a way to evaluate its own burden. Every controller periodically sends the packet number information to monitor.

### 3.4 Monitor

The monitor periodically receives burden information from its underlying controllers and builds a whole picture of current network status. The average burden is calculated by dividing total burden by controller numbers. Once the monitor finds that a specific controller's burden is much larger than the average value, it will select an idle controller as the migration destination. Monitor will maintain a data structure to record the mapping between switches and controllers, and update it when migration happened.

Monitor will not participant in the migration process: it is the two controllers that coordiates with each other to finish the migration logic. Monitor only decides the migration time and signals the source/destination controllers to finish migrating targed switch. Once migration is completed, controller will signal the monitor that

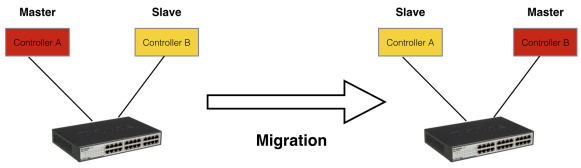


**Figure 1: Migration Process**

this migration process has completed.

# 4. MIGRATION PROTOCOL

The key component of this elastic SDN controller is the migration protocol; it must ensure that ongoing network flow can not be disrupted. In this section, we will describe the specific requirements that the protocol needs to meet, and the protocol details.

Since OpenFlow 1.3, OpenFlow starts to support different modes of controllers. Now one switch can connect to more than one controllers, and each of them could be in three different modes: *Master*, *Equal*, and *Slave*. The controller in slave mode has read-only access to the switch, and can not receive any asynchronous messages (e.g., `Packet-In`). Controller in Master and Equal modes can modify the state of switch and receive asynchronous messages from switches. For one switch, there could be only one controller in master mode. The number of controllers in Equal or Slave mode doesn't have this constrain. The single controller in master node will be primarily responsible for processing all the messages coming from that switch. A controller can apply for changing its role by sending `Role-Request` message to the switch.

So the migration process is actually changing the role of the controllers that switches connected to: upgrade a controller's mode from slave to master, and change the original controller's mode from master to slave. As showed in Figure 1, switch **S** connects to two controllers **A**, **B**. Initially controller **A** is master and controller **B** is slave, after migration, controller **B** becomes master and controller **A** becomes slave. In this way, the switch **S** has been migrated from **A** to **B**.

## 4.1 Requirement

Generally, there are three requirements that a safe migration protocol needs to provide.

- **Liveness** *At least one controller is active for a switch at all the time.*

  Here active means at least one controller is responsible for processing all the messages coming from this switch; this controller is not necessary to be a master controller. This requirement guarantees that at any time, for any flow arriving at this switch, there will always be a controller processing it. Therefore the network flow will not be disrupted.

- **Safety** *Exactly one controller processes every asynchronous messages from the switch at any moment.*

  This requirement guarantees that one message will be processed by only one controller. Otherwise, duplicate process of messages like `Packet-In` will result in duplicate entry in flow table. This will disrupt network flow.

Here we discuss a naive protocol and explain why it will not work. Let target controller, whose role will be changed from slave to master, directly send a `Role-Request` message to switch requesting to become master. Switch will then set the target controller to be master and the original master to be slave. This protocol, however, will violate liveness requirement. Assuming that switch has sent a `Packet-In` message to the original master controller, and before it receives the `Packet-Out` reply, it receives the `Role-Request` message from the slave controller. The switch then will set the original master to slave, and after that, the reply message arrives. But now since the original master controller has been set to slave, switch will just ignore that message. On the other hand, the new master won't receive that `Packet-In` message since it has already been sent. As a result, that `Packet-In` message are lost during the migration.

## 4.2 Protocol

OpenFlow standard clearly states that a switch may process messages not necessarily in the order that they are received. This is an important thing that should be carefully considered. Also, both liveness and safety requirements should be satisfied.

Here we describe the details of the four phase migration protocol, as showed in Figure 2. Suppose we are going to migration switch X from controller A to controller B.

***Phase 1: Change role of target to equal.***

Controller A signals B to start migration. After having received the message, controller B first sends a `Role-Request` message to switch X, asking to become equal, and then sends a ready message to A after it receives `Role-Reply` message from X.

Instead of directly changing the target controller into master mode, here we first change the controller B to equal mode. Now controller B can receive all the asynchronous messages from switch X, but will just ignore all of them, making sure there is no violation of the safety requirement.

***Phase 2: Insert and remove a dummy flow.***

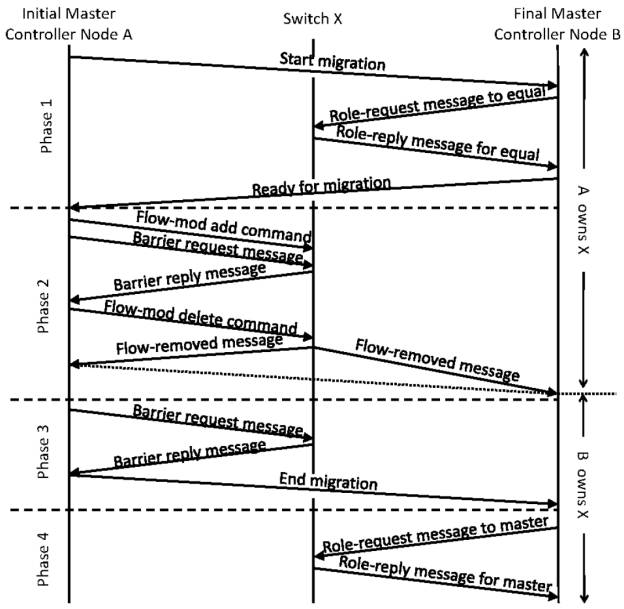


**Figure 2: Migration Protocol**

After A has received the ready message from B, it will insert a dummy flow into the flow table of switch X, through `Flow-Mod` add message. This dummy flow can be pre-configured to make sure that it will not be used for any packet routing. Then controller A send the `Flow-Mod` delete message to remove the flow which it just added. Both controller A and controller B will receive a `Flow-Remove` reply message.

Here the protocol use the fact that, when a controller issues a remove-flow command, all the controllers connected to this switch will receive a `Flow-Remove` reply message. This reply message is the **migration movement** where switch X now is transferred to B. Two controller doesn't need to receive these reply message at the same time, but the order of messages received by these two controllers are consistent, which means no message will come before A receives the `Flow-Remove` message and after B receives the `Flow-Remove` message, therefore guarantees safety.

There is one more thing we need to worry about: the switch may process `Flow-Mod` delete operation before the `Flow-Mod` add operation. Therefore we need to send a barrier message after we insert the dummy flow to guarantee the order.

***Phase 3: Flesh pending request with a barrier.*** At the end of Phase 2, controller B will take over switch X, meaning that B will start to response all the asynchronous messages coming from X, and A would ignore all the messages coming after `Flow-Remove`. But we cannot change A from master to slave immediately, since A may be still processing some messages coming before `Flow-Remove` but haven't finished processing yet. Controller A must finish all the pending request before it becomes a slave controller.

Therefore controller A send a barrier message to the switch, and after the reply message is received, we are sure that all the messages before that barrier have been finished. Then we send a message to B, indicating that the migration is done.

***Phase 4: Make target controller final master.*** Now everything is ready: controller A has stopped responsing to any asynchronous message and controller B has taken over. Controller B then send a `Role-Request` message to switch X, requesting to become master. Switch will set controller B to master and at the same time set A to slave.

The above protocol contains 6 round-trip communications. The "Start migration", "Ready for migration" and "End migration" messages are the messages between two controllers, which need to be configured as a part of Elastic SDN controller. All the other messages are the standard OpenFlow messages.

# 5. IMPLEMENTATION

## 5.1 Technology

### 5.1.1 Mininet

We use Mininet to emulate our network environment. Mininet is a network emulator, using lightweight virtualization to simulate a large network of virtual hosts, switches, controllers, and links in a single linux system. Its switches support OpenFlow for highly flexible custom routing and Software Defined Networking.

Mininet run real code including standard Unix/Linux network applications as well as the real Linux kernel and network stack. This means that a design that works in Mininet can usually move directly to real systems with minimal changes.

### 5.1.2 Ryu Controller

We use Ryu framework to implement controller and monitor logic. Ryu is a python SDN controller frameworks, provides software components with well defined API that make it easy for developers to write network control applications with python script. Ryu supports various protocols for managing network devices, such as OpenFlow from 1.0 to 1.5, Netconf, OF-config, etc.

## 5.2 Generate Topology

We use Mininet to emulate a simple network environment. The network topology, as showed in Figure 3, has 8 switches which interconnected to each other into a tree style. Each switch is linked with N hosts, where the N

could be configured. We have two controllers which are fully connected to all the switches. We generate this topology using Mininet Python API.

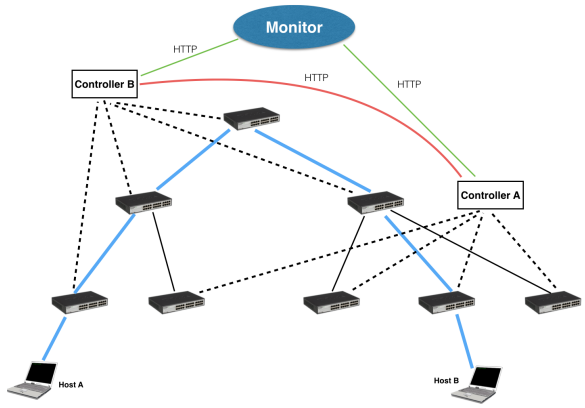


**Figure 3: Our Implementation Topology**

## 5.3 Monitor Implementation

Monitor initializes and maintains the role of each controller for all switches, it uses HTTP request to talk with controllers. Basically monitor will receive two kinds of messages coming from controllers:

- Each controller's `Package-In` number
- Notification that migration is done

Monitor will send one type of message to controllers:

- Tell controller to start migration

The reason we use HTTP request is that Ryu Framework provides HTTP Server API for controller by default. This won't cause another performance bottleneck because controller only submits traffic data periodically (10 seconds in our implementation).

In every 20 seconds, monitor checks the recent collected traffic data to determine whether a migration command should be issued or not and which switch should be migrated.

## 5.4 Controller Implementation

There are three main functions in the controller.

1. ***Collect traffic data and send it to monitor periodically***

   For the traffic data collection, we record the number of latest 10 seconds `Packet -In` message that this controller has received, send to monitor and clear the cache.

2. ***Learn forwarding table for each switch***

   Controller learn the routing strategy using a simple algorithm, as showed in Algorithm 1. However, controller only maintains the routing roles locally, without writing them into the switch's flow table. In this way, each switch will send a `Packet-In` message to controller every time it receives a network package, this network will then generate enough `Packet-In` messages for us to do the test.

3. ***Conduct migration***

   For the migration logic, two controller coordinate with each other through the default HTTP Server provided by Ryu. Each controller maintains the migration state for itself. The detail migration logic has been presented in Section 4.

**Algorithm 1** Switch Learning algorithm

```
1: procedure ReceivePacketIn(p)          ▷ When a packetin arrives
2:     srcIp ← p.src
3:     dstIp ← p.dst
4:     dict[dstIp][srcIp] ← p.in_port
5:     if dict[srcIp][dstIp] then
6:         Forward p to dict[srcIp][dstIp]
7:     else
8:         Flood p
9:     end if
10: end procedure
```

# 6. EVALUATION

We compared the two metrics before and after the migration: controller throughput and response time. We wrote python script to send UDP packets in different rates from Host A to Host B in Figure 3.

As showed in the figure, the thick blue line indicates the path of the packet sending from Host A to Host B. Although both controller A and B are responsible for four switches, in this scenario, the traffic will go through all 4 switches that belong to controller B, and only one switch that belongs to A. Therefore, controller B has approximate four times of traffic than A, which is a huge load imbalance. Monitor will then detect this imbalance and migrate one switch from B to A, so A will end up with 5 switches while B only has 3, which means an equalized traffic distribution and higher overall controller network capacity.

## 6.1 Throughput

### *6.1.1 Before Migration*

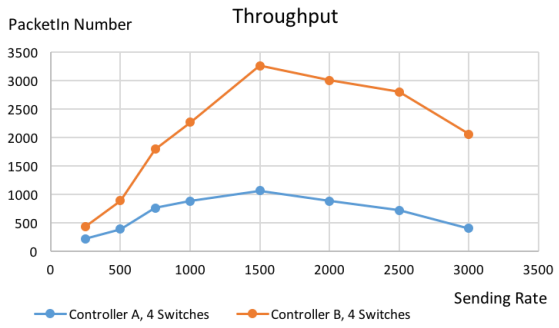


**Figure 4: Throughput of Two Controllers Before Migration**

In the Figure4, it's clear to see that the Controller B is heavily loaded, while the A still has a lot of unused bandwidth. However, the overall performance of the controller network already reaches the ceiling after the sending rate of 1500 packets/s because of congestion.

### *6.1.2 After Migration*

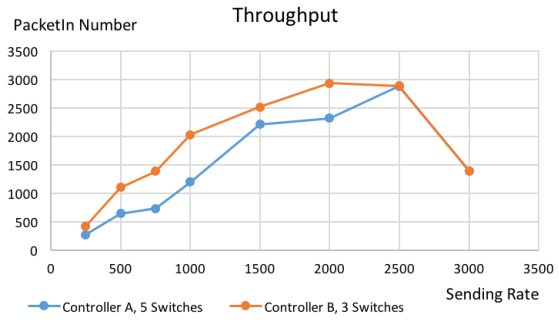


**Figure 5: Throughput of Two Controllers After Migration**

After the migration, we see significant improvement in Figure5.

- The sum of these two lines is significantly higher after migration, which means the overall throughput is improved.

- The threshold of sending rate is around 2500 packets/s, compared to the 1500 packets/s before.

## 6.2 Response Time

### *6.2.1 Before Migration*

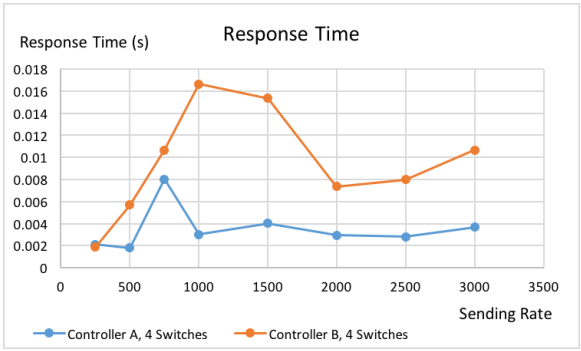


**Figure 6: Response Time of Two Controllers Before Migration**

In the Figure6, the respond time of Controller B has a higher respond time than the Controller A. Because B is under more traffic pressure.

### *6.2.2 After Migration*

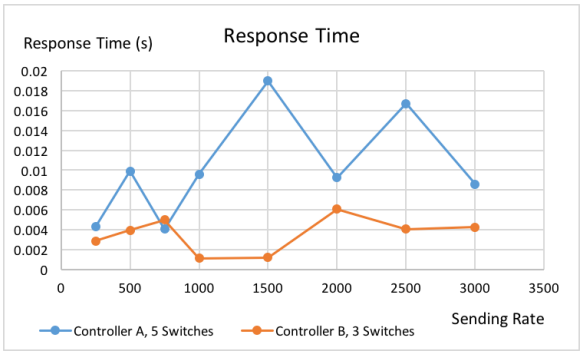


**Figure 7: Response Time of Two Controllers After Migration**

In the Figure7, we can see the the response time of Controller B reduces a lot while that of Controller A goes up because of the balanced traffic. While the weird thing is the response time of Controller A is increased too much, with two big hazard. We think the network congestion could be the possible cause.

## 6.3 Migration Speed

In the Figure8, we illustrate the change of response time during the process of migration. From the graph we can see that the migration is finished very fast, less than 300ms, and the two lines of response time quickly become closer.

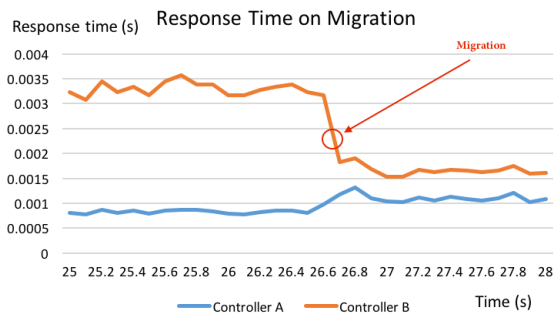


**Figure 8: How Migration Happens**

## 7. DISCUSSION

The result has showed that after migration, the load is more balanced between two controllers, especially for throughput. The overall throughput increased after migration. However, the result of the response time is little weird. We think the main reason for this is the network congestion. Since Mininet only provides a very limited bandwidth network, when we increase the packet sending rate, a lot of packets will be dropped and resent because of congestion, therefore disrupt our test result.

We conduct our experiment on a very simple network topology, with only one application: packet sending application. It is a proof of concept experiment. For future work, we need to deploy the system on large scale network, with real world network application running on top of it, to see how it performs and how much it can help to balance the load within the network.

## 8. REFERENCES


[1] Maestro: A system for scalable openflow control.

[2] "Beacon": `openfolw.stanford.edu/display/Beacon/Home`.

[3] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed sdn controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 7–12, New York, NY, USA, 2013. ACM.

[4] "Floodlight": `floodlight.openflowhub.org`.

[5] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[6] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[7] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically centralized?: State distribution trade-offs in software defined networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 1–6, New York, NY, USA, 2012. ACM.

[8] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On controller performance in software-defined networks. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.