Simple ElastiCon : A Distributed SDN Controller

Aditya V Kamath, Vedanuj Goswami, Abhishek Sen, Brenden Raulerson, Sai Teja Duthuluri

***Abstract*— Software Defined Networking (SDN) is becoming an increasingly popular technique for managing large scale networks including the cloud and datacenters. The main advantage SDN offers is the separation of the data plane and the control plane, allowing the network to be programmed by applications running on top of a central controller. With increasing adoption of SDN, it is gradually becoming necessary to build highly scalable and distributed SDN controllers. In this report, we describe the implementation of a distributed SDN controller which dynamically changes switch allocation among controllers depending on the load being experienced by the controllers.**

I. INTRODUCTION

The Software Defined Networking approach is generally thought to contain a single controller to manage network events from all switches in the network. This has been depicted in Figure reffig:centrallized. In large enterprise or datacenter networks, there are often 100s-1000s of switches and routers, which would all be replaced by SDN switches. As per the SDN design, each switch must contact the controller when a packet not matching any of the rules in its table arrives. The switch-controller interface is a regular network interface such as Ethernet running at 1Gbps or 10Gbps. Considering each switch contains more than 10 ports, the controller-switch interface can easily be overwhelmed. The controller also needs to perform some processing on each event that the switches sends. For large networks, the processing might be bottlenecked by the CPU in the controller causing delays in packet processing and buffering of these unprocessed packets in the switches. The controller also becomes a single point of failure in the entire network. If the controller goes down, only existing flows in the network might continue to operate, but no new flow would be able to run in the network.

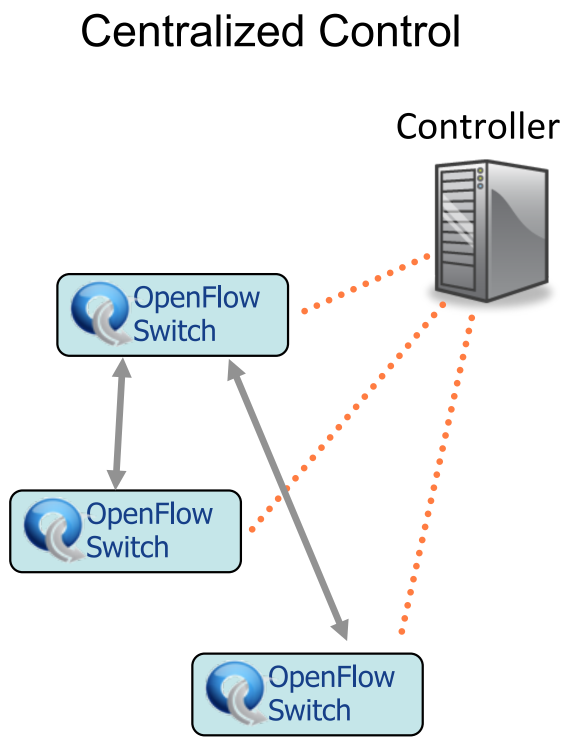

Fig. 1. Centrallized Controller Architecture

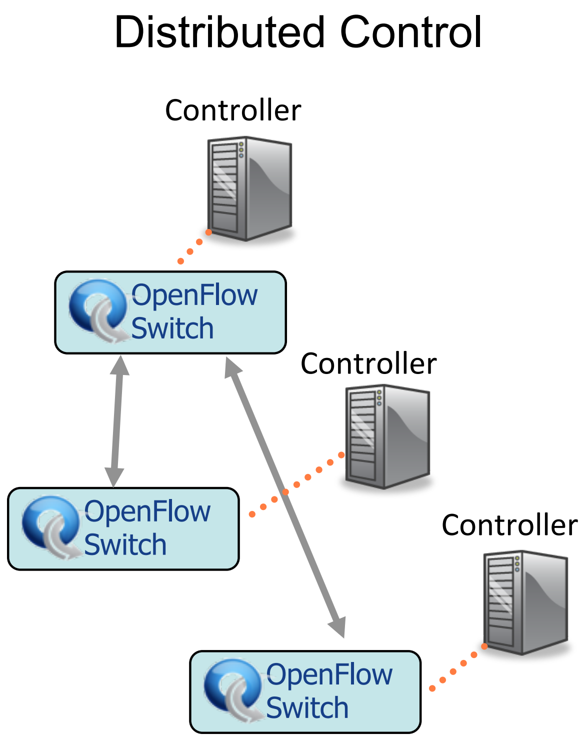

Fig. 2. Distributed Controller Architecture

A natural move forward is to use distributed controllers which act like a logically centralized controller to the rest of the network, although physically separated. This is visually represented in Figure 2. Many papers have been written proposing different schemes of distributed controllers to control SDN networks. Most approaches however assume a static mapping between the controllers and the set of switches in the network that they manage. This introduces a new problem in the system, which is an uneven load distribution among the controllers. It is not uncommon for different parts of a large network to experience different load conditions, and also at different times of the day. The overall result is that, the network administrator might have to over-provision the controllers handling the busier parts of the network, while wasting compute resources in controllers which handle the less loaded regions.

A solution to this problem was proposed in [3] wherein the assumption of static mapping was removed and replaced by a dynamic assignment of switches to controllers based on the load experienced at each controller. The authors also propose, what they call an *elastic* approach, wherein the number of controllers managing the entire network could be (1) shrinked, if the overall load in the network was low and (2) increased if the current number of controllers was insufficient to handle the load. The elastic approach is made feasible by a cloud computing environment in which controllers are virtual machines (VMs) which can be shut off, or brought up as and when required.

Through this project we aim to implement the approach described in [3] but without considering the elastic component of the approach. This would mean that the number of controllers in the network would be decided and fixed at the start of the system and will not change over the course of time. [3], [4] also do not give very low level details on the implementation of the system. In the following section we describe the system design that we came up with to implement the idea presented. We then move to describe the implementation in detail and present our experimental results.

II. LITERATURE SURVEY

Controller architecture has evolved from the original single threaded design [5] to more advanced multi threaded design in recent years [1], [7], [9] . Despite the significant performance improvement over time, single controller systems still have limits on scalability and vulnerability. Some research papers have explored implementation of distributed controllers across multiple hosts [6], [8]. However they assume static mapping between the switches and hence lack the capability of dynamic load balancing and elasticity. However it is found that there is 12 orders of magnitude difference between peak and median flow arrival rates at a switch [2], [9]. Using static mapping between switches requires significant overprovisioning of of resources which is inefficient in hardware and power.

III. SYSTEM ARCHITECTURE

The high level architecture of the system remains the same as described in our proposal which is shown in Figure 3. The system architecture consists of a logically centralized distributed store which is used to maintain and coordinate state information across the different controllers in the network. There are n controllers in the network, each of which are responsible to manage some set of switches in the network. A switch always has exactly one Master controller which will process all its requests. Only during a switch migration operation, which we will discuss shortly, the Master controller of a switch might change. However the migration protocol guarantees that there will always be one controller to handle the switch events at all points in time. The distributed data store is used by the controllers to update their state and make it visible to the others, and using the state information from other controllers in making routing and switch migration decisions.

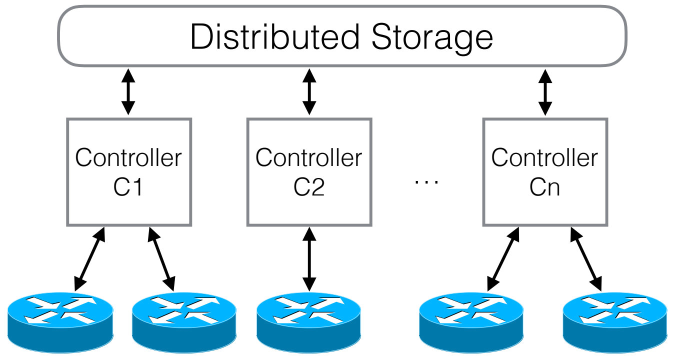

Fig. 3. System Architecture

Although [3], [4] provide high level overview of the system and the functions of each part of the system, many low level implementation details are omitted. In order to complete this project, our team discussed and thought through the missing pieces. One of the outcomes of this process was the component design of each controller in the system and their functions. The various modules within the controller are shown in Figure 4. For brevity, only 2 controllers and a single application is shown, however both can be generalized to arbitrary numbers. Following is a description of each of the components and their functions.

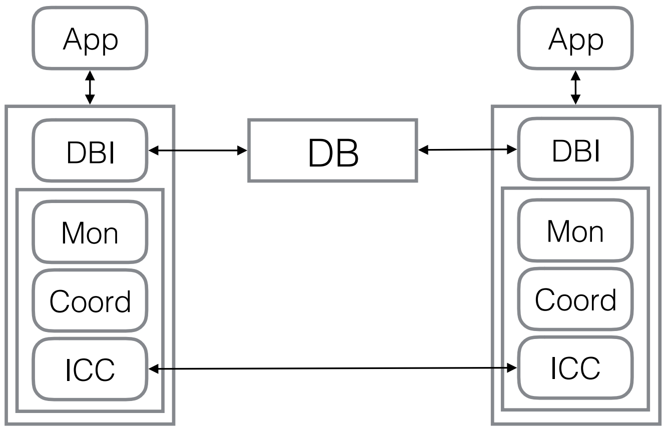

Fig. 4. Controller Architecture

1) **App**: This is the SDN application that is to be run in the network. For simplicity only one application is

shown in the diagram, however there could be multiple applications running simultaneously. Each application needs to implement in addition to its core functionality, 2 methods to dump their current state in the database and to update state by reading from the database. These functions are called whenever a switch migration takes place, which will be described shortly.

2) **DBI**: This is the database interface module which is responsible for exposing simple APIs to the rest of the controller whenever they need to access the shared database for information. These APIs can also be used by the application to store and retrieve its state.
3) **DB**: This is a database instance used to store the state of the network, controllers and their switch assignments and is also used to store the App state. It is a logically centralized system that is accessible from all controllers.
4) **Mon**: This module monitors the CPU load on each controller and also approximate rate of incoming messages from each switch the controller is responsible to handle. This information is also stored in the database for all other controllers to see, and is used to decide when to migrate a particular switch over to another controller. The migration is determined by the optimal load balance and hence the Mon has load balancing capabilities, the details of which we describe in the Details of Implementation section below.
5) **Coord**: This is the coordinator module which is to coordinate actions between the different components within a single controller. This module is also responsible for handling the OpenFlow messages that arrive from the switch and pass them onto the other modules which require them. Finally some OpenFlow messages are also passed onto the App, for it to implement its functionality.
6) **ICC**: The Inter Controller Communication module is used for controllers to communicate events with each other during the migration process. The migration protocol as described in [3], [4] requires the controllers involved in migration to exchange messages to synchronize their states. This is done through ICC. [3], [4] advocate a full mesh connection between the controllers, however we take an on-demand approach.

IV. OVERALL APPROACH

In the setting we are aiming for as part of this project, the network will start with an empty database instance, a set of controllers, and all switches assigned to a single controller from this set. As the control plane traffic starts to increase from the switches, the load balancer will kick in and start moving over the switches to other controllers which are experiencing relatively lesser load. If the overall control plane load can be handled by the set of controllers, the system should reach a stable state without too many migrations happening.

Software libraries being used for different components: We are using an open source SDN controller written in Python called Ryu which supports OpenFlow 1.3, which we require in order to implement switch migration. We will be using Mininet to create a virtual test bed for testing and evaluation purposes. Mininet internally uses OpenVSwitch as the software switch in the network. For the distributed store, we are using MongoDB, a popular distributed NoSQL databases available for free. For the Mon and ICC components, we will just be using the Python standard libraries.

A. System Components

1) **Switch migration including ICC**: The switch migration part of the project involves almost all of the modules described in the controller architecture diagram. The ICC module of the controller starts a TCP server on each controller that is running. Suppose the load balancer decides to move a switch from controller C1 to C2, then the ICC module in C1 initiates a connection to C2s waiting TCP server. The Coord module sets up all the required OpenFlow message handlers and redirects them to the ICC module during a migration process, and the Mon module is responsible for triggering migration. The Coord module also calls into the application to dump and retrieve its state from the database to make its operation seamless once migration has completed. The ICC component along with the Coord component implement the switch migration protocol that is described in detail in [3], [4] and and has been successfully implemented and tested. The protocol diagram is given in Figure 5.

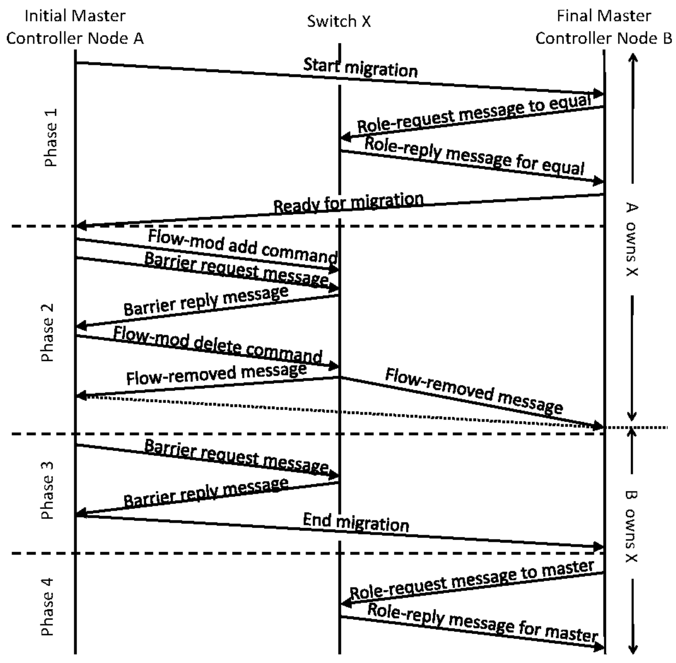

Fig. 5. Switch Migration Protocol

The main considerations for the switch migration protocol are 3 properties:

a) **Liveness**: The switch must have at least one controller active at all times - before the migration, during the migration and after migration. If this property is not met, the switch might get disconnected and its events lost.

b) **Safety**: During the migration phase, the switch events will be sent to the original master and the future master. However since the state of the network is shared across all controllers, the events from the switch must be processed by exactly one controller in order to keep the application's view of the network consistent.

c) **Serializability**: The events generated by the switch for the controller to process must be processed in the same order as they are generated. An out-of-order processing might cause incorrect decisions taken by the SDN application and causing the network to be in an inconsistent state.

[3], [4] describe how the protocol satisfies the above requirements. The switch migration protocol uses some of the features of OpenFlow Version 1.3 in order to guarantee these requirements. In order to get this working in our project, both the controller software and Mininet had to be configured correctly to use OpenFlow Version 1.3. Although the verion of OpenVSwitch (v2.2) does not have a complete implementation of the OpenFlow 1.3 protocol, everything required to implement the switch migration protocol existed, which made the development of the project much easier.

2) **DB setup and APIs for controller**: A MongoDB database is setup which is a NoSQL database. It uses JSON-like documents with schemas for storing key value pairs. This type of database is chosen for faster write/updation/deletion operations which is important during switch migration operations. APIs are implemented for performing CRUD operations on the database for two collections.

- *Controller* : This collection stores the controller specific information which includes the CPU load, switches owned by the controller.
- *Switches* : This collection stores the forwarding tables for the switches.

3) **Load Balancer**: Optimization problems are computationally intensive. As noted in [3], since traffic changes quickly, the benefits of combinatorially searching for optimal load balance is short-lived. Therefore, we we use a simple greedy heuristic for load balancing to determine the switch to controller mapping. Each controller's Mon component determines when the controller is being overloaded. Once the event is triggered, the controller uses the load information of other controllers in the system from the distributed store to determine the controller to migrate to, and initiate a migration. An important piece of migration is also to determine which switch from the current assignment is to be migrated. For this purpose, Mon keeps track of the rate of incoming events from each switch which the controller is currently managing. The switch having the highest rate of incoming events is the one which would cause a higher CPU utilization on the controller, and hence the controller decides to migrate that switch. The process is described stepwise below:

- Determine switch S having the highest rate of events.
- Get the loads of all other controllers in the network L.
- From L find the controller which has minimum load: C$_{min}$
- Initiate migration of S to C$_{min}$

This greedy heuristic effectively maps the most loaded switch to the least loaded controller and outputs a reasonable load balanced switch to controller mapping and runs much faster than the combinatorial solution.

4) **App**: The SDN application used in [3], [4] is a simple Layer 2 routing application. The controller maintains a forwarding table for each switch in the network and forwards packets based on this. If an incoming packet does not have a matching entry in the forwarding table, the packet is flooded through all other ports of that switch. This application has been implemented by us.

V. IMPLEMENTATION DETAILS

This section describes the implementation specifics of our system.

A. Coord

This class instantiates a controller object which can perform several functions, explained below:

- Initialize all the other sub-modules needed for a controller, like ICC, Mon and DBI.
- For a given switch, one controller declares itself as the master controller, while all others become slave controllers.
- The controller can handle packet-in, barrier-request, barrier-reply, flow-add and flow-remove events.
- The controller can handle Role-Request and Role-Reply messages.

B. Inter-Controller Communication

This class defines the protocol regarding communication between two controllers. This class implements a purely control plane functionality. An instance of this class can:

- Initialize a TCP server at a given specific port number, passed as argument
- Initialize migration from the current controller to another controller.
- Accept a switch migration request, and become the master for a given switch, for which it is already a slave.

C. Monitor

This class instantiates a monitor which performs three tasks:

- It calculates the load being experienced by the controller at a given time instance.

- It can return the switch ID which is causing the maximum load on the controller.
- Based on the load being experienced by the controller, the monitor can decide whether a switch migration is necessary or not. This is performed on a periodic basis.

D. *Database Interface*

This class exposes endpoints which the controller instances can utilize to perform CRUD operations on the distributed store. There are two main interfaces exposed:

- The Controller interface which stores experienced load and switch information for every controller in the network.
- The application data, i.e., the forwarding table of each switch.

VI. DEVIATIONS

Our system is an implementation of the idea presented in [4]. However, we have made some modifications in order to address certain practical issues observed during development:

1) **Sticky Migration**: Suppose that switch S1 is migrated from controller C1 to C2. We observed that in some scenarios, a migrated switch might be migrated back to C1 because the load on C2 increases beyond the threshold limit. This can result in a state of constant flux for the switch which is highly undesirable. To solve this problem, we introduce sticky migration. Once S1 is migrated to C2, S1 cannot be migrated again for a preset amount of time. This ensures that *a)* The packets queued during migration are serviced, and *b)* The switch belongs to a particular controller for at least some minimum time, giving an opportunity for traffic to settle down, and possibly reduce the load.
2) **Dynamic TCP connections**: In the original implementation, the TCP connections between controllers are persistent and exist even after migration is complete. In our implementation, when a controller C1 decides to migrate a switch S1 to another controller C2, it dynamically creates a TCP connection with the TCP server running on a pre-defined port on C2, performs the migration, and then closes the connection. This was done to simplify the process of maintaining controller state. Also, since our experimental setup was small, there was no major overhead with dynamic TCP connections.

VII. EXPERIMENTS AND RESULTS

In order to evaluate the system, we conducted 3 experiments. Before we describe the experiments and the results, we first describe the setup used to conduct these experiments. For our experiments we used two laptops, eacho running an Intel Core i7 processor running at 2.7GHz with 8GB of RAM. The two laptops were connected via Gigabit ethernet. One of the laptops ran multiple controllers as multiple processes, while the second laptop ran Mininet inside a virtual machine emulating different topologies that we created for the experiments.

Since main objective of the experiments are to study the control-plane scalability characteristics that the distributed controllers enable and not the data-plane or routing scalability, we modify the SDN application to not install any flow rule in the switches, and forcing the switch to request the controller for forwarding information for every packet. This implies that for every packet that arrives at the switch, 2 packets are exchanged between the switch and controller, one from the switch to the controller with the packet-in event, and the second from controller to switch directing the switch to forward the packet at a specific port. This overhead is incurred for every switch that a controller manages in the network. The same changes to the routing algorithm are also made in [4].

However there is one main difference between the experiments of [4] and our experiments. [4] modified OpenVSwitch to inject Packet-In events directly to the controller without actually generating a packet in order to increase efficiency and test with higher packet-in rates at the controller. They also modify Mininet to run across different hosts, and essentially run a switch on a different physical machine. However for our experiments, these changes were not incorporated. In that sense, our experiments and results are more realistic, except that the emulated switches all shared the same CPU resources.

A. *Throughput Experiment*

The first experiment we conducted was to test the throughput gain by using a distributed controller as compared to a single centralized controller. For this experiment we used the emulated network topology shown in Figure 6. It consists of 2 hosts and three switches connected in a linear fashion. In the single controller setting, only one controller manages the switch events from all the switches in the network. In the multiple controller setting, we configure our system with the required number of controllers. All switches are initially mapped to the first controller C1 and as load on the controller increases, migrations will be triggered to move switches to the other controllers. When the number of controllers is 2, the ideal mapping is 2 switches assigned to one controller and the third mapped another, and in the 3 controller case, each controller is assigned one switch.

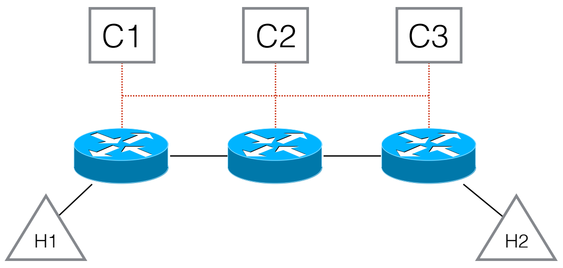

Fig. 6. Topology for throughput experiments

In order to test the throughput, we ran iperf bandwidth testing tool between the two hosts in TCP mode and used the throughput logged at the server side for the results. All numbers were averaged across 5 runs of the experiment. The results of the experiment are show in Figure 7. We see that as the number of controllers is increased, there is a near linear growth in the throughput achieved between the two hosts. The absolute value of throughput is in the 10s of Mbps due to the overhead incurred in having the switch contact the controller for each packet and also due to the small overheads in crossing the VM boundaries. We also observe while running the experiment that our load balancing approach does stabilize to the ideal mapping of switches to controllers in this experiment. The trends in the graph and the stabilization at the ideal mapping gives evidence that our implementation is correct and the results obtained are valid.

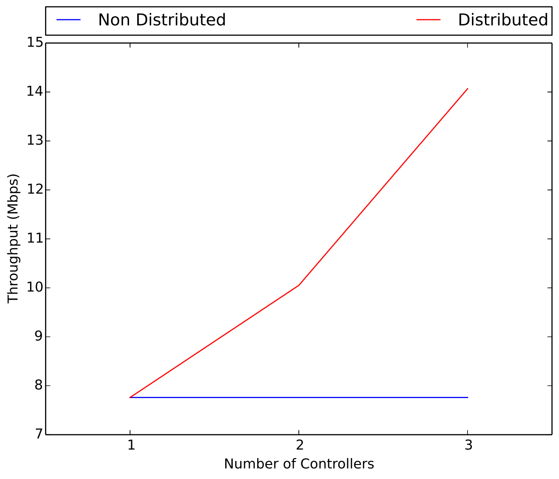

Fig. 7. Throughput variation with number of controllers

B. Migration Jitter Study

The team decided to conduct another interesting experiment. In order to maintain the 3 properties that the switch migration protocol must satisfy, the switch has to buffer some of the events during the time of migration. This buffering introduces jitter in the flows flowing through the network. The team wanted to study this aspect of the system, which wasn't conducted in [4]. For this experiment we used the same hardware setting, but a different topology shown in Figure 8.

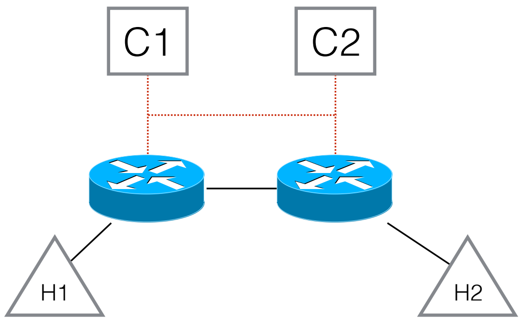

Fig. 8. Topology for migration jitter experiments

To study the jitter introduced, we ran ping from one host to another and noted the round-trip latency experienced. The regular ping command sends an ICMP Echo Request every second, however this was too slow for the purpose of this experiment. The ping utility has a useful option *-i* with which the interval between subsequent ping requests can be specified. We set it to generate a request one every millisecond. We then ran the experiment in two scenarios, which we present below.

1) No additional load: In this setting the network is handling no other load other than the ping packets that are going between the hosts in the network. We run the ping utility with the lower inter-request interval and trigger a migration in our system. We then plot the latency experienced by each ping packet to visualize the effects. Figure 9 shows the results of this experiment. The migration is triggered right around the 1500 mark where we see the jitter introduced. There is one peak towards the end of the migration phase when the switch finally assigned to the new controller. However this is experienced only by a single packet. The entire migration process takes around 10ms and the network stabilizes quickly after that. The average latency is about 8ms during the time of migration, which is 4ms up from the initial average latency. For most applications this jitter would be acceptable and might even go unnoticed.

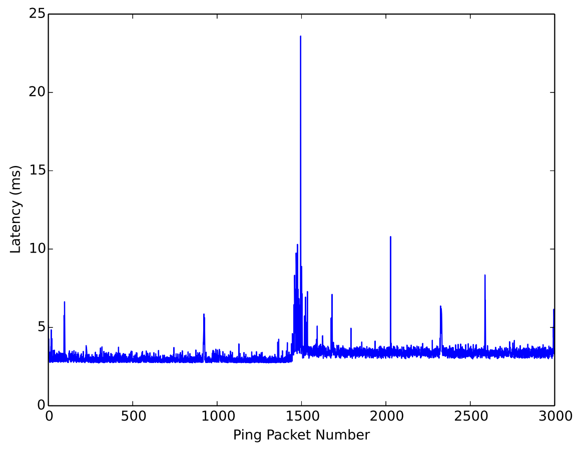

Fig. 9. Migration Jitter No Load

2) With additional load: As mentioned in [4], the migration time increases as the rate of packet-in messages increase at the switch. This is due to the way the migration protocol is designed. To study this, we repeat the previous experiment but also run a parallel UDP flow with a constant bit rate between the hosts. The constant bit rate was set to 80% of the achieved throughput through iperf in the network. The results of this experiment are shown in Figure 10. In this experiment the migration is triggered right around the beginning of the graph. The plateau in the graph is the migration period, and the final peak, similar to the previous experiment occurs in the final phase of migration, after which the network stabilizes. The total time for migration now increases to almost a 1 second with an average latency experienced during migration to be 19ms, up from the initial average of 6ms. However the latency during migration is stable with little jitter. Although we see that the migration time has increased significantly, the network is still operational albeit with a higher latency value.

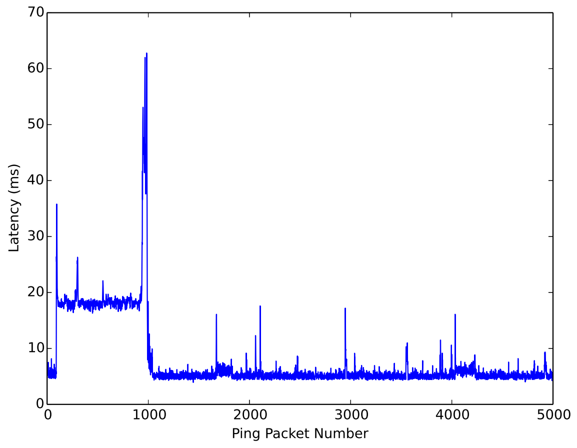

Fig. 10. Migration Jitter With Load

VIII. DISCUSSIONS

Although most of the system architecture and functional details were mentioned in [3], [4], some of the lower level details were not. As described in Section VI, the frequent migrations was something we immediately observed when we first started running our experiments. The authors of [4] modified Mininet and were able to run the switches in different machines and in their experiments generated Packet-In messages at a constant rate. However in our experiments we chose to perform experiments using the iperf bandwidth testing tool which would try to transmit as many packets as possible, limited only by TCP congestion control. This caused a higher CPU utilization on the controllers and even after migration the new controller experienced high loads causing frequent switching.

With a careful choice of transmission rate and CPU usage threshold for migration, the authors of [4] would have been able to avoid such a scenario. This was another area which we had to spend some time on during the development of the system, which was to determine the various parameters such as CPU threshold for detecting overload and how often the monitoring and load balancing should happen.

The team also brainstormed on implementing a global load balancer in comparison to the current greedy heuristic of each controller deciding for itself. The team members however were not experienced enough with common distributed system approaches to guarantee coordination and consistency, which would be required to implement such an approach.

IX. CONCLUSIONS AND FUTURE WORK

As part of the course project, we successfully built a dynamic load-based distributed controller system for an SDN network as described in [3] with a few modifications. Our throughput experiments validated the gains obtained from a distributed controller architecture and our migration jitter experiments provides more insight into the working of the system, which the network administrator and application writer could use, especially for latency critical applications. One approach could be to move the flow through a different route during migration and bring it back once migration is complete. Care must be taken not to trigger migrations on the other routes.

As part of future work, we would like to setup the system on a real testbed SDN network, as opposed to the current virtual testbed, and run the same experiments to study the characteristics. Other extensions of this system include support for fault tolerance. The ICC module could be used to send handshake messages every few seconds to detect the liveness of controllers. In case a controller goes down, the switches could be dynamically re-assigned to the remaining controllers.

REFERENCES

[1] `https://openflow.stanford.edu/display/Beacon/Home`.

[2] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement* (2010), ACM, pp. 267–280.

[3] DIXIT, A., HAO, F., MUKHERJEE, S., LAKSHMAN, T., AND KOMPELLA, R. Towards an elastic distributed sdn controller. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 7–12.

[4] DIXIT, A. A., HAO, F., MUKHERJEE, S., LAKSHMAN, T., AND KOMPELLA, R. Elasticon: an elastic distributed sdn controller. In *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems* (2014), ACM, pp. 17–28.

[5] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review 38*, 3 (2008), 105–110.

[6] KOPONEN, T., CASADO, M., GUDE, N., STRIBLING, J., POUTIEVSKI, L., ZHU, M., RAMANATHAN, R., IWATA, Y., INOUE, H., HAMA, T., ET AL. Onix: A distributed control platform for large-scale production networks. In *OSDI* (2010), vol. 10, pp. 1–6.

[7] NG, E. Maestro: A system for scalable openflow control. *Rice University* (2010).

[8] TOOTOONCHIAN, A., AND GANJALI, Y. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking* (2010), pp. 3–3.

[9] TOOTOONCHIAN, A., GORBUNOV, S., GANJALI, Y., CASADO, M., AND SHERWOOD, R. On controller performance in software-defined networks. In *Presented as part of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services* (2012).