

Load Migration in Distributed SDN Controllers

Mohammad Amin Beiruti, Yashar Ganjali
Department of Computer Science, University of Toronto

Abstract—Distributed control solutions were introduced to address controller reliability and scalability issues in Software-Defined Networking (SDN). The dynamic nature of network traffic can lead to load imbalance amongst controller instances. A highly loaded controller instance can be slower in responding to datapath queries, and can slow down the entire control platform as state synchronization, and consensus amongst controller instances are performed cooperatively.

In this paper, we present a new and efficient load migration protocol, called ERC, for shifting input load associated with overloaded controller instances towards lightly loaded instances. Our protocol has three distinguishing properties compared to prior works on this area: it is extremely efficient, resilient to failures during migration, and ensures consistency among all controller instances. ERC can be used for a wide range of network applications including load balancing, power saving, and resource optimization. It is also significantly more efficient than existing load migration protocols with 25-55% reduction in migration time, and 10-20% reduction in required migration buffer size.

I. INTRODUCTION

Software-Defined Networking (SDN) can simplify network management, improve network resource utilization, and enable a wide range of network applications [1] by decoupling control plane and datapath elements [2]. Modern SDN solutions rely on distributed controllers to enhance resiliency to failures, and to decrease response times in large-scale networks (e.g. ONIX [3], ONOS [4], and DIFANE [5]).

In any distributed controller, different controller instances¹ can have unbalanced load due to natural differences in arrival rates of switches in various topological layers [6], as well as fluctuations in network traffic. A highly loaded controller instance can be slower in responding to datapath queries. It can also slow down the entire control platform as state synchronization and consensus amongst controller instances are performed in a cooperative manner.

The natural solution to the load imbalance problem is to shift some of the input load associated with overloaded controller instances towards lightly loaded controller instances. Fast load migration can address other problems in addition to load balancing. For example, shifting the load away from a lightly-loaded controller instance, and putting that instance in power-saving mode can be an effective way for *energy reduction* [7]. Also, being able to migrate control traffic rapidly can have a major impact on how fast our controller system can react to failures. As a given controller instance fails, we need to quickly change how various switches are associated

with that controller instance and ensure their load is quickly shifted to other controller instances. An agile load migration scheme can significantly reduce the response time in this case, and ensure the network resumes its normal operation after a failed controller instance.

Network security is another application for load migration: when parts of the network, or the control plane are under attack, we can shift suspicious traffic towards dedicated controller instances to protect the rest of the control plane. The suspicious traffic can be further analyzed on these dedicated controller instances (or directed towards dedicated intrusion-detection systems) without impacting the normal operation of the control system.

Finally, we can use controller load migration to take advantage of locality in computer networks. Certain network applications, for example Virtual and Augmented Reality (VR/AR) traffic, have extremely high volumes of traffic and are very sensitive to delays. To ensure short response times for these flows, we can push controller instances as close as possible to the datapath, introducing the notion of local controllers [8]. Such local controller instances can be essential for making highly adaptable software-based service chains that combine controller resources and datapath programmability.

The Problem. Given the distributed nature of SDN controllers, the dynamic nature of traffic, and in the presence of failures, we need to ensure any load migration solution has the following properties.

- i) **Efficiency.** Any controller load migration solution needs to be fast and efficient to have an agile control plane that can quickly react to changes in the underlying network and traffic conditions. Also, when a new controller instance is spawned to deal with increased demand, an efficient load migration solution can ensure optimal resource allocation, and thus shorter response times.
- ii) **Resilience to failures.** Given the distributed nature of the control plane, any load migration system should be resilient to failures to ensure the control plane can operate smoothly in the presence of link and controller instance failures.
- iii) **Consistency.** Finally, any load migration solution needs to ensure controller instances have a consistent state during the load migration process. Network view inconsistencies among controllers can lead to undesirable behavior of the control plane.

The distributed nature of the control plane, inherent latencies in the network, and unpredictability of failures and network traffic make it very difficult to simultaneously ensure all these properties in a load migration system.

¹We use the term “controller instance” to refer to each physical/virtual machine serving as an SDN controller in the distributed control plane.

Our New Protocol. In this paper we introduce *ERC*, an *Efficient, Resilient, and Consistent* load migration protocol for distributed SDN controllers. *ERC* is the first load migration protocol that explicitly deals with failures. By buffering all incoming messages (in both the source and destination of migration) during the migration process our protocol ensures we can roll back to a reasonable state if a failure happens. *ERC* is also designed with consistency in mind, i.e. it ensures all other controller instances (including controllers in equal mode, which are not either the source or destination of the migration) are kept in a consistent state.

What distinguishes *ERC* from prior solutions the most (in addition to failure resilience and consistency) is its efficiency. *ERC* has a completion time which is 25-55% less than prior solutions, which is a significant improvement given the importance of timing in load migration. Interestingly, the improvement in completion time is higher when the controllers have higher load. This is an extremely helpful feature when the system is under stress due to high volumes of traffic. Also, the buffering requirements of *ERC* is 10-20% (per controller instance) less than similar protocols. In Section IV, we will show this reduction in buffer size is a side effect of faster completion times.

This paper is organized as follows. Section II provides the background and a brief overview of prior work in this area. We introduce our new protocol in Section III, followed by an evaluation of the protocol in section IV. Section V concludes the paper and describes directions for future work.

II. BACKGROUND AND RELATED WORK

In this section, we review the desired properties for any load migration protocol, present a brief review of previous work in this area, and provide some background required in the paper.

A. Desired Properties

In an operational network, delegating the control of an active switch from its master controller instance to another one can not be done spontaneously. In order to ensure the control plane does not end up with a corrupt view of the network state, there are several properties that we need to uphold during the migration process.

Liveness: each switch has to have one active master controller at all times. Otherwise, the switch will not be able to route new flows, or handle changes to network conditions.

Serializability: controller instances should process messages of each switch in the same order they are received by the control plane. For example, assume a switch sends a link down message to the controller and a link up message right after that. If the controllers do not process these two messages in the right order, they might decide that the link is down, resulting in an incorrect view of network state.

Safety: every asynchronous message should be responded by at most one controller. In a distributed system, the state of all devices is not synchronized at all times. If two different controller instances respond to the same message, the switch might end up with a conflicted state, which might be very difficult (if not impossible) to resolve or reverse on the switch.

Consistency: at all times, the master controller's view of a switch should be compatible with the view of all equal controller instances of that specific switch. Otherwise, the controller instances in equal mode cannot take responsibility of the switch during emergencies (overloaded master, or failures), which is the main reason to have controllers in equal mode.

Failure resiliency: failure of destination controller instance during migration (while the initial master is still in charge) should not lead to state loss in the initial controller instance. During handover, at some point, the initial master should stop processing received messages from the switch that is being migrated. After exchanging a few messages, the final master will take the responsibility of that switch. During this time, if the migration doesn't succeed due to failure, the initial master should be able to resume its normal operation without any state loss.

B. Related Work

Switch migration in SDN controllers has been studied in various contexts. Obadia et al. proposed a solution to handle controller failures [9]: switches associated with the failed controller instance need to be migrated to an operational controller instance in this scenario. Here, there is no elaborate handover mechanism, and we cannot provide performance or behavior guarantees as the initial controller is already failed and cannot participate in the migration process. The mechanism proposed is a simple role change of the new controller, and cannot be used as a building block for applications like load balancing, resource optimization, and power saving.

Liang et al. designed a dynamic load adapting mechanism to shift the extra load away from highly loaded controllers [10]. Their proposed solution is also a simple role exchange between controllers without considering failure, state consistency, and efficiency of the handover process.

The most comprehensive solution for load migration in SDN controllers is a 4-phase protocol proposed by Dixit et al. [11]. This protocol is carefully designed to ensure safety, serializability, and liveness, and it has been used as a building block for more complex applications like power saving and load balancing [12]–[15].

Unfortunately, the 4-phase protocol lacks failure resiliency, i.e. if a controller instance fails during the migration process the entire controller system might fail. Moreover, the 4-phase protocol does not deal with controller state consistency. The solution relies on an external database to share state, which pushes the state exchange between controllers out of the scope of the work. In practice, we need to ensure all controllers in a distributed control system (i.e. all master, equal, and slave mode controllers) have a consistent state to ensure correct behavior. Finally, the 4-phase protocol is not very efficient as we will show in this paper.

Load Migration Granularity. There are different options in terms of granularity when it comes to migrating load from one controller instance to another. *Packet-based load balancing* deals with individual packets. It has an extremely high overhead and might not be feasible in certain control

models. When the distributed controller plane relies on state sharding, or locality of state (to accelerate response time) for example, any packet-based load balancing scheme will result in a significant overhead for state synchronization amongst controllers. Also, ensuring messages sent to different controller instances are processed in the order they are received by the control plane is extremely challenging at packet-level.

Flow-based load balancing is more manageable in terms of controller state since one controller instance will be responsible for keeping state associated with that specific flow. However, flow-based schemes create new challenges in dealing with short-lived and long-lived flows. The majority of network traffic is short-lived with extremely small duration/length [6]. For such flows, the gain for shifting a flow from one controller instance to another might not be enough to offset the cost of migration. Long-lived flows, on the other hand, have the extra complexity and cost of heavy-hitter detection [16], [17] which might be substantial in the presence of a large number of flows.

In *switch-based load migration* (e.g. [11]) the entire load of a given switch is shifted from one controller instance to another. This approach has significantly lower overhead compared to packet-based schemes, and eliminates the complexity of flow-based schemes. Also, due to statistical multiplexing, the aggregated traffic of each switch is more predictable (compared to individual flows for example) which makes planning load migration (for various applications including load balancing, power saving, resource allocation, etc.) more tractable. Finally, switch-based migration is a perfect match for OpenFlow, the *de facto* standard for SDN data/control plane interfaces, which associates each switch with one master controller instance.

Based on all of these arguments, we focus on switch-based load migration in this paper.

Control State in SDN. Any distributed SDN controller is comprised of various *network applications*. The control plane is expected to provide a mechanism to keep the *state* of these applications. The details of how this state is preserved depends on the design of each SDN controller platform, but regardless of these details, the control state can be categorized as follows:

- *Switch State.* Each switch keeps some control state locally (e.g. packet/byte counters). Any controller instance can access this state by directly querying the switch.
- *Control Application State.* Depending on the control platform's design, network application state might be stored in a simple key-value store or a shared storage system [4], [18], [19]. Each controller instance has certain permissions to read or write on the shared storage. We note that core control applications (which are part of the network operating system) also use this storage for keeping their state. The control platform ensures reliability and consistency of this storage.
- *Local Control State.* Each controller instance can have a local state that is not managed by the control plane. This is usually considered to be *ephemeral*, i.e., in case the controller instance fails, this state is going to be lost.

During the migration process, we are not concerned with

switch state and control application state: these are managed independently by the switches, and the shared memory system. To ensure the continuous operation of the control plane however, we need to ensure that local control state is transferred during the migration process. That is one of the main responsibilities of any load migration protocol as we will see next. We expect this to be a relatively small state, compared to a VM snapshot for example. However, we have much tighter timing requirements which makes VM migration techniques unusable in this setting.

C. OpenFlow

Our load migration protocol relies on OpenFlow 1.5 as the API between controllers and switches. Here, we briefly explain OpenFlow in the context of switch migration. OpenFlow allows a given switch to establish connections with more than one controller. These controllers assume different roles. A unique *master* controller is responsible for replying to any queries and messages generated by the switch. Any controller in *equal* mode receives an exact copy of each message that the switch sends to the master, but it is not obligated to reply to these messages. Multiple controllers can be in the equal mode. These controllers receive all the messages from the switch in order to keep the same view of the network as the master controller. Finally, any controllers in *slave* mode only receives a restricted subset of messages generated by the switch by default [20].

OpenFlow supports three types of messages: *controller-to-switch messages*, *asynchronous messages*, and *symmetric messages*. Controller-to-switch messages are initiated by a controller instance to modify or inspect the state of a given switch. The most important messages of this group are:

- *Barrier.* Switches do not necessarily process their received messages in the same order they arrived. If a controller wants to dictate a specific order between two messages, it has to separate them by a barrier message.
- *Role_Request.* These messages are used to set or request the role (master/equal/slave) of a controller instance with respect to a specific switch.
- *Packet_Out.* In response to a *Packet_In* message, the controller sends a *Packet_Out* toward the switch to update its flow table.

Asynchronous messages are initiated by the switch without solicitation from a controller. The switch informs the control plane about network events and updates its state through these messages. The most important ones in this category are:

- *Role_Status.* When the role of a controller changes, its corresponding switch informs the controller by sending this message.
- *Controller_Status.* The switch notifies its related controllers about changes in OpenFlow channel status².
- *Packet_In.* In case the switch can not decide about an incoming packet, it redirects the message to the con-

²The *Controller_Status* message has a key role in our protocol and is supported in OpenFlow 1.5.

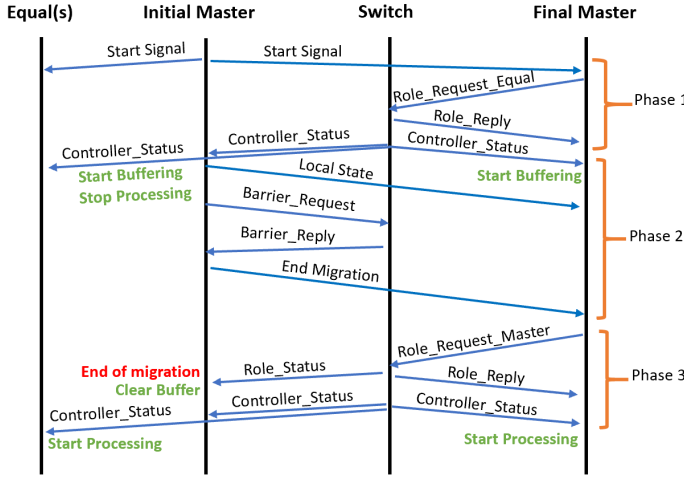


Fig. 1. Newly proposed load migration protocol, ERC.

troller. The controller, in turn, decides and sends back a **Packet_Out** message to the switch.

Finally, symmetric messages are mostly used to check the liveness of connection between a switch and a controller instance, or to commence the initiation of a connection when a new device is plugged in.

III. ERC LOAD MIGRATION PROTOCOL

In this section, we present ERC, an *Efficient, failure Resilient*, and *Consistent* load migration protocol. Figure 1 shows how ERC migrates a given switch by shifting its load from the initial master controller instance to a slave controller. The initial master controller buffers incoming messages during the migration to ensure state can be recovered in case of failures during migration. In addition to helping with failures, we exploit this buffer as a coordination mechanism between initial and final master controllers, leading to significant savings in migration time compared to prior solutions [11].

Also, as Figure 1 shows, we explicitly deal with controllers in equal mode to ensure their state is kept up-to-date during the migration process. This ensures equal mode controllers have a consistent view with the master controller, which is essential if we need to use equal mode controllers (in case of failure for example).

We assume that both the initial and final master controllers are aware of the migration and their future roles. This can be part of the control application (e.g. load balancing) that initiates the migration. And, we assume that each switch sends its asynchronous messages in the same order to all corresponding controllers. We note that the final master is a slave controller at the beginning of the protocol.³ Our protocol has 3 phases described below.

³We also note that a higher-level recovery mechanism, which is out of the scope of this work, is expected to initiate failure recovery, and to decide, for example, to roll back the state of a master controller, or use an equal mode controller instance as master after a failure. Our migration protocol ensures such a mechanism has the ability to deal with failures without losing state, but does not enforce a specific mechanism.

Phase 1. The initial master controller initiates the migration by sending a “start signal” message to the final master and all the other controllers which are in equal mode. Right after the final master receives the “start signal” message, it requests a role change from slave to equal. The switch processes the role request message, followed by a **Role_Reply** message back to the final master (to inform the controller of its new role). Changing to the equal mode is essential for the final master controller since it needs to have the same view as the initial master by the end of this procedure, and before it can take the responsibility of being a master controller.

In OpenFlow 1.5 [20], when the status of a controller changes for any reason, the switch has to inform all the connected controllers by a **Controller_Status** message. At the end of Phase 1, the switch broadcasts a **Controller_Status** message to all associated controllers, triggering the next phase.

Phase 2. After the initial master receives the **Controller_Status** packet, it starts buffering any messages (e.g. **Packet_In** messages) from the switch. At this point, the final master is also in equal mode, and receives and buffers a copy of all the messages from the switch. Since the switch sends the asynchronous messages in the same order to all its corresponding controllers, and since both the initial and final master controller instances start buffering at the same time (right after receiving the **Controller_Status** message), their buffered queues are identical. All the controllers in equal mode also start buffering messages from the switch after receiving the **Controller_Status** message (to have a consistent view with the master controller instance).

Since the final master wants to take responsibility of the switch from the initial master, it needs to have the exact same state as the initial master. We only need to deal with the local state of the controller (see **Controller State** in Related Work Section II-B). Network application state is synchronized through a shared memory mechanism. At the beginning of Phase 2, the initial master sends a copy of its local state to the final master. This ensures the final master starts with the exact same state as the initial master, right before receiving the **Controller_Status** message. Any changes in state beyond this point is replicated on the final master since it receives the same set of messages from the switch.

The initial master sends a **Barrier_Request** to make sure that the switch does not have any pending requests. The switch processes all incoming requests before this **Barrier_Request**, and responds **Barrier_Reply**. When the initial master receives the **Barrier_Reply**, it can safely hand over the responsibility to the final master.

Phase 3. Once the final master controller receives the end of migration signal and after it is done processing the local state messages, it sends a **Role_Request** message to the switch. The switch changes the role of the current (initial) master to slave (and inform it with a **Role_Status** message). It also grants the master role to the final master controller instance. The switch broadcasts a **Controller_Status** message to all its corresponding controllers including equal and slave mode controller instances.

The end of migration for the initial master is when it recognizes its role has changed to slave. This controller instance no longer needs the messages buffered during the migration, and can simply clear the buffer. The final master and other equal controllers will start processing the buffered messages right after receiving the Controller_Status message.

A. Validation of Properties

Here, we will show the newly proposed ERC protocol has the desired properties described in Section II-A.

Liveness. During the migration process, the initial master continues serving as the master for the switch. This role is relinquished to the final master in Phase 3, only after the final master has acquired the state needed to continue serving as the master for the switch. With this, we have exactly one master associated with the switch at all times, which proves the liveness property.

Serializability. As long as the initial master processes the messages from the switch, it does it in the same order they arrive. By the beginning of Phase 2, this processing is paused and continued later by the final master controller. Since the switch sends out asynchronous messages in the same order to all its connected controllers, the order of buffered messages in the final master will be the same as initial master. By ensuring the final master starts processing at the exact point (i.e. message) that the initial master has paused, we guarantee messages are processed in order.

Safety. Up until the end of Phase 1, the initial master is the only controller instance which responds to queries from the switch. During Phase 2 and Phase 3, responding to the switch is paused and messages are buffered. After the final master receives an acknowledgement of its role change to master, it starts processing buffered messages. This ensure that at anytime only one controller instance is responding to the switch. Additionally, by ensuring the final master continues processing buffered messages at the exact point that the initial master pauses, we ensure that each message does not receive more than one response.

Consistency. Our protocol ensures the controllers in equal mode stop processing (and start buffering) messages at the same point⁴ as the master controller stops processing and starts buffering the messages from the switch. At the end of the migration process, receiving the second Controller_Status triggers all the equals and the master to resume processing.

This delaying of the processing of messages in equal mode controllers ensures that they have the same view as the initial master during the migration process.

Failure Resiliency. If the initial master fails during migration, any failure recovery mechanism can choose any of the equal mode controllers as the new master. The consistency property ensures all equal mode controllers have the same view as the initial master.

Let us consider the case where the final master fails during the migration process before the initial master has relinquished

⁴Here, “same point” refers to the logical time defined by the order of messages sent by the switch.

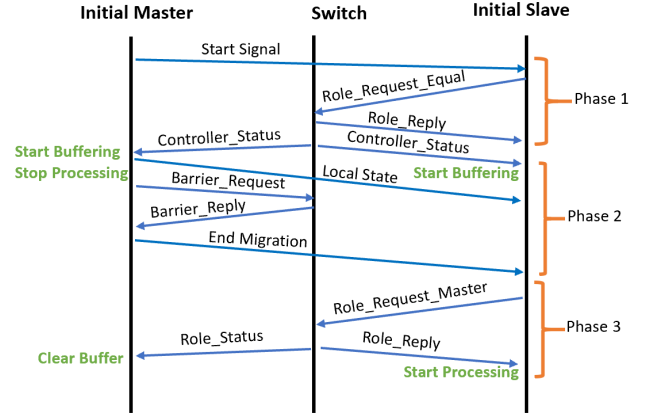


Fig. 2. Simplified ERC protocol.

its responsibility. In this case, the initial master can resume its operation by first processing all the buffered messages before resuming its normal operation. Prior protocols cannot handle failures in this case.

B. Simplified ERC Protocol

Figure 2 shows a simplified version of the ERC protocol that does not support consistency. We can also relax failure resiliency if the initial master controller does not buffer messages during migration. In this case, the end of migration for initial master controller will be after sending “End Migration” message, and does not need to wait until receiving the Role_Status message. This simplified protocol serves as a tool for apple-to-apple comparison of our protocol with previous protocols [11] that do not support failure resiliency and consistency.

IV. EVALUATION

We have conducted an extensive set of simulations to evaluate the performance of our protocols. We designed and implemented an object-oriented distributed SDN simulation environment in C#. Our simulator supports processing rate limiting for switches and controllers, and is completely configurable in terms of buffer sizes. We implemented OpenFlow 1.5 as the API between switches and controllers, and added customizable input/output message buffers, and flow tables in our switches⁵. Each network controller instance runs a basic discovery application (to identify switches and network topology) and a routing application (to ensure connectivity between controllers). Each controller instance uses a priority queue for keeping incoming messages, as well as a migration buffer to keep messages during the migration process.

When a new flow arrives at a switch (i.e. when no flow table rule matches the incoming packet), the switch sends a Packet_In message to the associated controller(s). These Packet_In messages constitute the majority of the load for each

⁵The commonly used SDN emulation environment mininet does not support OpenFlow 1.5 which we need here. It also lacks essential features (including limiting the processing rate of switches and controllers) as well as built-in support for a distributed control platform.

controller. Each controller instance processes the incoming request and responds with a `Packet_Out` message to the switch instructing it to install a new entry in its flow table.

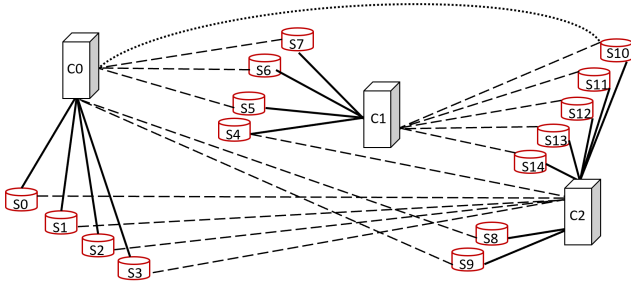


Fig. 3. Network topology.

Simulation Setup. Figure 3 shows the topology of our network. There are three controllers ($C0$, $C1$, and $C2$) connected to 15 switches. A solid line between a controller and a switch indicates the controller is the master for the switch. The dashed lines represent equal mode, and the dotted lines represent the slave mode relationship. Unless mentioned otherwise, we assume switch-to-controller and controller-to-controller delays are equal. Assuming all switches generate an identical amount of load, the controller $C2$ will have the highest load (compared to $C0$ and $C1$). In our evaluation, we focus on the handover of switch $S10$ from $C2$ to $C0$. Each controller has a limit on the number of messages it can process (set to 10 messages per time unit). This is to take message processing times into account in our simulations.

The inter-arrival time distribution of `Packet_In`s for each switch is defined by two parameters. A load factor p ($0 \leq p \leq 1$) identifies the load of the switch and a multiplexing factor f (an integer ≥ 1) represents the number of hosts connected to each switch. Each host generates packets according to a Bernoulli IID process with probability p/f . With $f = 1$ we have a simple Bernoulli IID process. Higher values of f increases the burstiness of traffic as several hosts can generate packets simultaneously. The multiplexing factor is set to 10 by default. Later, we study the impact of traffic burstiness on load migration by changing f from 1 to 20.

Experiments. We perform several rounds of experiments. We fix the processing capacity and buffer sizes for each switch and controller instance in each experiment, and vary the load factor p , multiplexing factor f , the relative controller-to-controller delay, and the load of the final master controller.

We run each simulation 100 times and compare the running time of the 4-phase protocol (as the closest prior work) with ERC and the simplified version of ERC. We also compare the buffer size requirements of ERC and 4-phase protocols. ERC buffers messages in both initial and final master controllers to cope with failures. The 4-phase protocol buffers messages in the final master only. We keep track of the number of buffered messages in both protocols to compare buffering requirements of the two protocols.

Next, we describe our experiments under four different scenarios.

A. The Impact of Controller Load

We start by a direct comparison of ERC and 4-phase protocols in a setting similar to a data center, where the controller-to-controller and switch-to-controller delays are equal (1ms in this case). Each controller receives the load of 90 to 120 different hosts through connected switches (with multiplexing factor $f = 10$ for each switch). We consider three scenarios with controllers having low, medium, and high level of load, corresponding to 30%, 80%, and 105% loads, respectively⁶.

The blue and orange bars in Figure 4(a) show the running time of the 4-phase and ERC protocols for various controller load levels. ERC has a run-time which is about 30% less than the 4-phase protocol, despite the fact that the 4-phase protocol does not have failure resiliency and consistency properties. The gray bar in Figure 4(a) represents the simplified version of the ERC protocol, that does not support resiliency and consistency. The running time of simplified ERC provides an apple-to-apple comparison with the 4-phase protocol. We can see that the simplified ERC protocol runs in less than 50% of the time for the 4-phase protocol. Interestingly, the improvement in running time is higher under higher load scenarios, with 55% reduction in run-time compared to the 4-phase protocol.

Figure 4(b) shows that ERC requires 10-15% less buffering compared to the migration buffer size in the 4-phase protocol⁷. The migration buffer fills up during the migration process as controllers temporarily pause processing messages. Since ERC has a shorter completion time, it receives fewer messages during migration and therefore has a smaller buffer build up.

B. Impact of Link Delays

Next, we focus on the impact of controller-to-switch vs. controller-to-controller delays. This shows how ERC performs in various networks where the ratio of link delays between switches and controllers are different (e.g. data center vs. wide area networks). In these experiments, we assume controller-to-controller delays are fixed (as the unit) and vary the switch-to-controller delays from 0.1 to 10.

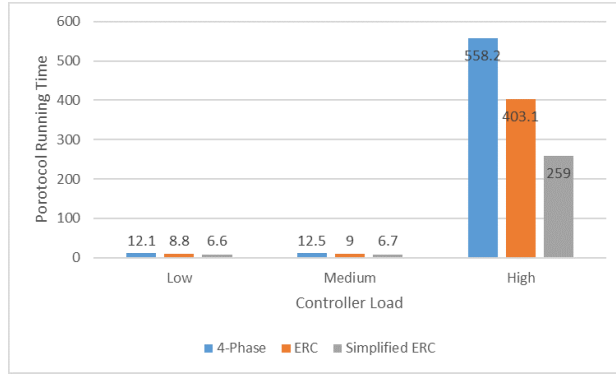
Figures 5(a) compares the running time of ERC, simplified ERC, and the 4-phase protocol as a function of delay between controllers and switches. Here, we are assuming an 80% controller load (called medium load in previous experiment). We can see that in all three protocols the running time increases as the switch-to-controller delay grows. This is expected as messages take longer to travel between switches and controllers and increased delays lead to increased running times for the protocol.

We can also see that ERC takes about 25% less time to complete compared to the 4-phase protocol, and the simplified ERC runs in less than half of the time regardless of the switch-to-controller delays.

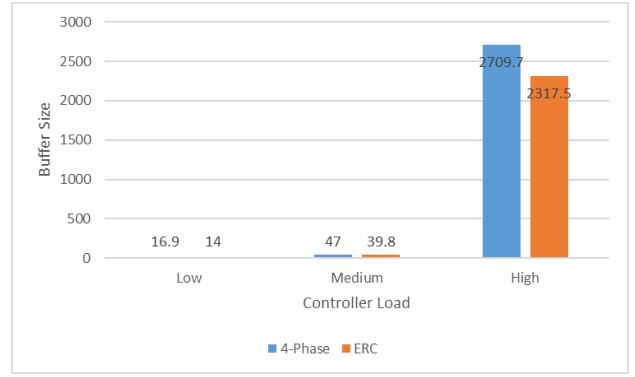
Figure 5(b) shows the migration buffer sizes for ERC and 4-phase protocols as a function of switch-to-controller delays.

⁶In our simulations, the buffer sizes are big enough to avoid packet drops during the simulation time. We note that the load level of 105% is temporary for the duration of the migration.

⁷Simplified ERC does not have extra buffer requirements since it does not support failure resiliency, and therefore it is not included in this graph.

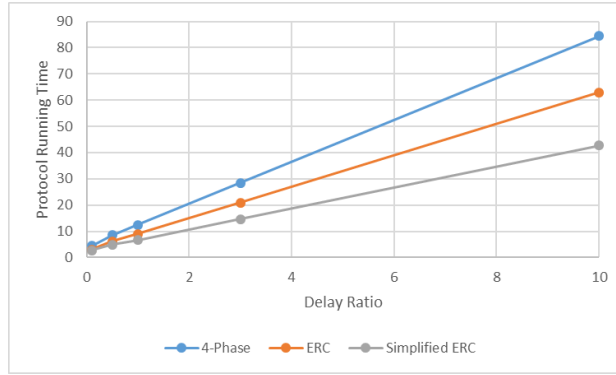


(a)

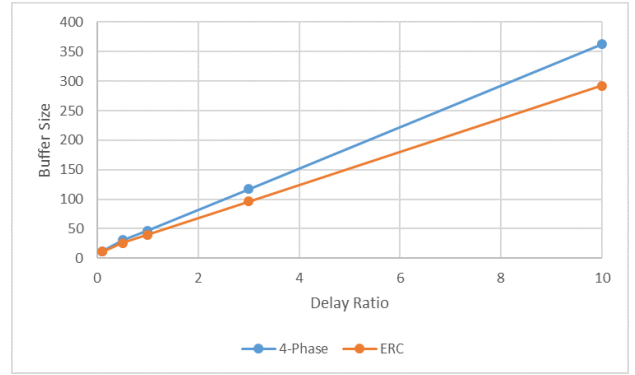


(b)

Fig. 4. Comparing ERC, simplified ERC, and 4-phase protocols under various load levels (low, medium, high): (a) migration running time, (b) migration buffer size.



(a)



(b)

Fig. 5. Impact of varying switch-to-controller delays on (a) migration completion time, and (b) migration buffer size.

Here, we also see that ERC requires 15-20% less buffering compared to the 4-phase protocol.

C. Impact of Final Controller Load

In the scenarios we have studied so far, the main bottleneck has been the initial master controller. The initial master is assumed to be highly loaded, and its performance determines the overall behavior of the protocol. This is reasonable in load balancing scenarios as we usually move switches from high-load controller instance to ones with lower load.

In scenarios where load balancing is not the main objective, we can envision applications where the load of the final master controller is also high. For example, when our goal is to save power we might even push load away from lightly loaded controller instances to higher load instances, in order to turn off lightly loaded instances and to ultimately save power.

In this section, we study the impact of final controller's load on migration protocols. We fix the initial controller's load (to high, i.e. 105%) and gradually increase the final controller's load. The goal is to study the impact of the final controller's load on run-time and buffer size requirements of migration protocols.

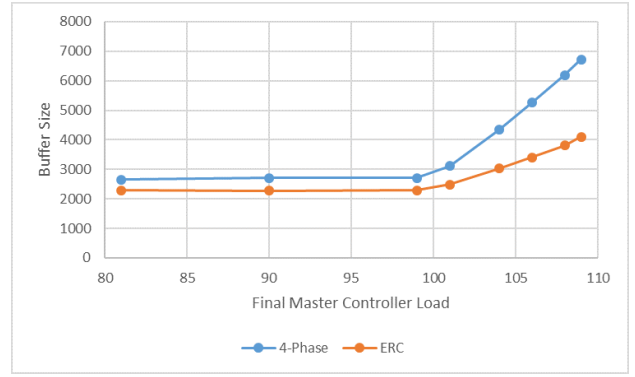
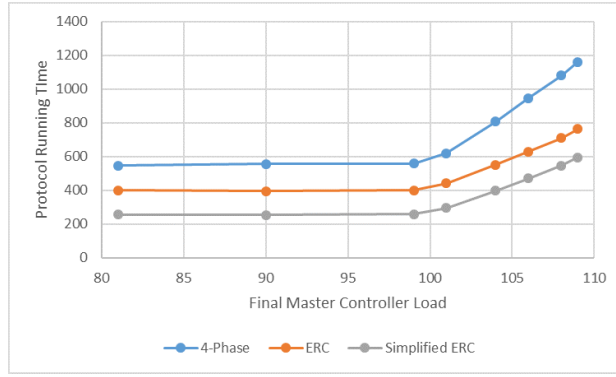
Figure 6(a) shows the impact of final master controller's load on migration run-time. As before, ERC and simplified ERC outperform the 4-phase protocol in terms of running time.

For load levels ranging from 80% to 98% ERC and simplified ERC run in 25-50% less time than 4-phase. As the load level on the final controller goes above 100% (temporarily during migration) the running time of all three protocols increases. Interestingly, the running time of ERC and simplified ERC grows at a lower rate (thus increasing the improvement in run-time).

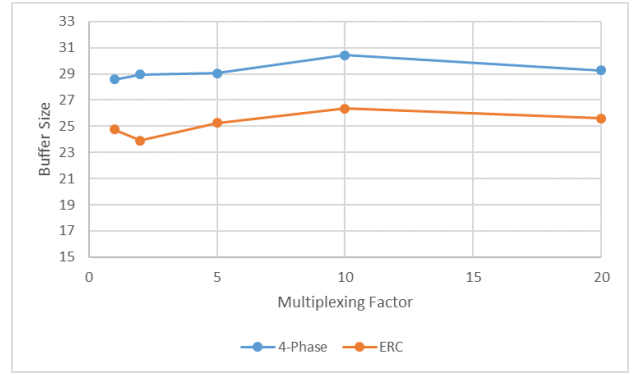
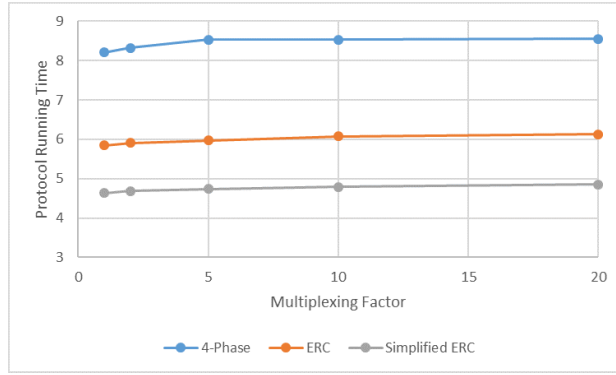
Figure 6(b) shows that ERC has a similar improvement in terms of buffer sizes. Normally, we expect 10-15% improvement in migration buffer sizes in ERC (compared to the 4-phase protocol). Here, we can see that the improvement can grow up-to 45% when the final controller is overloaded (up-to 110%).

D. Impact of Traffic Burstiness

In our first set of experiments, we investigated the impact of controller load on migration running time and buffer size. For a given controller load level, we can have different burstiness levels. In our final set of experiments we study the impact of load burstiness. For each switch, we keep the load factor p fixed (the same as the medium load in previous scenarios), and change the multiplexing factor f , which represents the number of hosts connected to the switch. Each host generates traffic at a rate of p/f . As we increase f we are adding f Bernoulli IID processes with probability p/f each, which makes the switch traffic more bursty.



(a) (b)
Fig. 6. Impact of final Controller load on (a) migration run-time, and (b) migration buffer size.



(a) (b)
Fig. 7. Impact of controller traffic burstiness on (a) migration run-time, and (b) migration buffer size.

Figure 7(a) compares the running time of ERC, simplified ERC and the 4-phase protocol. Here, the switch-to-controller delays are set to half of the controller-to-controller delays. The x-axis represents the multiplexing factor f changing from 1 to 20. We can see that increasing the burstiness initially leads to a slight increase in run-time, but all three protocols show a fixed running time for values of f beyond 5. In all cases, ERC and simplified ERC show the same level of improvement in run-time as in previous experiments.

Figure 7(b) compares the buffer size requirements. We can see slight variations in migration buffer size for ERC and 4-phase as f changes from 1 to 20, but the ratio of buffers is lower by 15-20% as in previous scenarios.

To summarize, our experiments show that ERC consistently outperforms the 4-phase protocol in various scenarios, and leads to significant reduction in migration time, as well as migration buffer size. The improvements are even more significant under high loads.

V. CONCLUSION AND FUTURE WORK

In this paper, we investigated the essential properties of a switch migration mechanism for SDN controllers, and its applications. We proposed a switch migration protocol that can be utilized for a wide range of network applications. Our protocol has liveness, serializability and safety along

with consistency and failure resiliency, which are essential properties for any distributed SDN control platform.

We evaluated our protocol by comparing it with the best previously known solution as the baseline. Our simulations show that ERC takes less time to complete, and consequently requires smaller buffers for migration (25-55% reduction in running time, and 10-20% reduction in buffer sizes). Interestingly, the improvement is even higher under high loads.

The ERC protocol introduced in this paper deals with one switch at a given point of time. An interesting problem here is to come up with a solution that can schedule a large number of switch migrations in parallel. This is an extremely challenging problem as running migration protocols on multiple controllers/switches might lead to conflicts. We can envision a system that a specific controller acts as the coordinator for such scheduling tasks trying to utilize a series of sequential and parallel switch migrations. This is left as an interesting future work.

VI. ACKNOWLEDGEMENTS

We would like to thank Huawei Canada for their support and partial funding of this project. We are especially grateful to Zhenhua Hu, for insightful comments and feedback on this work.

REFERENCES

- [1] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "A roadmap for traffic engineering in SDN-OpenFlow networks," *Computer Networks*, vol. 71, pp. 1–30, 2014.
- [2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.
- [3] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, and T. Hama, "Onix: A distributed control platform for large-scale production networks," in *OSDI*, vol. 10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6.
- [4] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, and W. Snow, "ONOS: towards an open, distributed SDN OS," in *Proceedings of the third workshop on Hot topics in software defined networking*. New York, NY, USA: ACM, 2014, pp. 1–6.
- [5] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 351–362, 2011.
- [6] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. New York, NY, USA: ACM, 2010, pp. 267–280.
- [7] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 136–141, 2013.
- [8] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in *Proceedings of the first workshop on Hot topics in software defined networks*. New York, NY, USA: ACM, 2012, pp. 19–24.
- [9] M. Obadia, M. Bouet, J. Leguay, K. Phemius, and L. Iannone, "Failover mechanisms for distributed SDN controllers," in *2014 International Conference and Workshop on the Network of the Future (NOF)*. IEEE, 2014, pp. 1–6.
- [10] C. Liang, R. Kawashima, and H. Matsuo, "Scalable and crash-tolerant load balancing based on switch migration for multiple open flow controllers," in *2014 Second International Symposium on Computing and Networking*. IEEE, 2014, pp. 171–177.
- [11] A. Dixit, F. Hao, S. Mukherjee, T. V. Lakshman, and R. Kompella, "Towards an elastic distributed SDN controller," *ACM SIGCOMM computer communication review*, vol. 43, no. 4, pp. 7–12, 2013.
- [12] A. Dixit, F. Hao, S. Mukherjee, T. V. Lakshman, and R. R. Kompella, "ElastiCon; an elastic distributed SDN controller," in *2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. New York, NY, USA: IEEE, 2014, pp. 17–27.
- [13] A. Krishnamurthy, S. P. Chandrabose, and A. Gember-Jacobson, "Pratyastha: an efficient elastic distributed sdn control plane," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 133–138.
- [14] T. Wang, F. Liu, J. Guo, and H. Xu, "Dynamic SDN controller assignment in data center networks: Stable matching with transfers," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.
- [15] X. Ye, G. Cheng, and X. Luo, "Maximizing SDN control resource utilization via switch migration," *Computer Networks*, vol. 126, pp. 69–80, 2017.
- [16] Z. Liu, D. Gao, Y. Liu, and H. Zhang, "An enhanced scheduling mechanism for elephant flows in SDN-based data center," in *2016 IEEE 84th Vehicular Technology Conference (VTC-Fall)*. IEEE, 2016, pp. 1–5.
- [17] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *International Conference on Database Theory*. Springer, 2005, pp. 398–412.
- [18] S. H. Yeganeh and Y. Ganjali, "Beehive: Simple distributed programming in software-defined networks," in *Proceedings of the Symposium on SDN Research*. ACM, 2016, p. 4.
- [19] R. Macedo, R. d. Castro, A. Santos, Y. Ghamri-Doudane, and M. Nogueira, "Self-Organized SDN Controller Cluster Conformations against DDoS Attacks Effects," in *2016 IEEE Global Communications Conference (GLOBECOM)*. IEEE, Dec. 2016, pp. 1–6.
- [20] A. Nygren, B. Pfaff, B. Lantz, B. Heller, C. Barker, C. Beckmann, D. Cohn, D. Malek, D. Talayco, and D. Erickson, "Openflow switch

specification version 1.5. 1," *Open Networking Foundation, Tech. Rep.*, 2015.