Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH PAPER

Load Migration in Distributed Softwarized Network Controllers†

Sepehr Abbasi Zadeh[1,3] | Farid Zandi[1,3] | Mohammad Amin Beiruti[1,2] | Yashar Ganjali[1]

[1]Department of Computer Science, University of Toronto, Canada
[2]Unfortunately, we lost this invaluable team member in the Ukrainian aircraft shot down.
[3]These authors contributed equally to this work.

Correspondence
Sepehr Abbasi Zadeh, Email: sepehr@cs.toronto.edu

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 among controller instances. A highly loaded controller instance can be slow in responding to datapath queries and can slow down the entire control platform, as state synchronization and consensus among controller instances are performed cooperatively.

In this paper, we present ERC, an efficient, novel protocol to migrate the load of a given switch from a controller instance to a different instance. Our protocol has three distinguishing properties compared with prior works in this area: 1) it is resilient to failures during migration, 2) it maintains consistency among all controller instances, and nevertheless, 3) it is more efficient than existing load migration protocols. Compared with state-of-the-art, ERC reduces the migration time by 23-50% depending on network load.

Maintaining the desirable properties of ERC, we introduce four variants of our protocol that can add to the versatility of the load migration in different scenarios. This is being achieved by considering variations of role-exchange between controller instances, which gives us an advantage over the fixed master-slave exchange that vanilla ERC or previous work support. We perform an extensive set of experiments to examine the impact of variable network parameters on the performance metrics of interest, and to show the effectiveness of the ERC family of protocols in load migration.

1 | 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 resilience to failures, and to decrease response times in large-scale networks (e.g. ONIX[3], ONOS[4], and DIFANE[5]).

In a distributed controller platform, different controller instances[1] can have unbalanced loads due to the natural differences in the arrival rates in the switches (e.g. core vs. edge switches)[6], as well as the fluctuations in the network traffic. Load imbalance

†A preliminary version of this work was appeared in the proceedings of IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary 2020.
[1]We use the term "controller instance" to refer to each physical/virtual machine serving as an SDN controller in the distributed control plane.

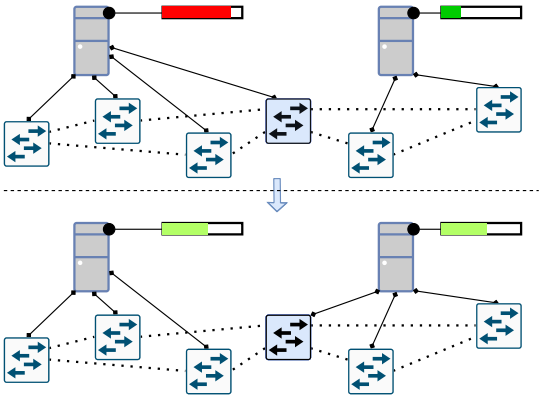


FIGURE 1 Load balancing; the most notable use case of a controller load migration protocol. The lightly-loaded instance on the right assumes control of the switch in blue, thereby balancing the load on the two controller instances. Solid lines depict control-path connections, while dotted lines show the datapath connectivity between the switches

not only leads to longer response times in the heavily-loaded controllers, but also slows down the entire controller platform. This is due to the necessity of state synchronization, which can only be performed in a cooperative manner by all controller instances.

The natural solution to the load imbalance problem is shifting some of the load associated with the overloaded controller instances towards the more lightly-loaded ones, as shown in Figure 1. However, given the distributed nature of the control plane, inherent latencies in the network, unpredictability of failures, and the variations in network traffic, designing a controller load migration protocol is a challenging task. An ideal protocol should have the following properties:

i) **Efficiency:** a controller load migration solution should be fast and efficient, which results in an agile control plane that can quickly react to changes in the underlying network and traffic conditions.

ii) **Resilience to failures:** the migration protocol 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, a load migration solution should ensure controller instances have a consistent state during the load migration process. Network view inconsistencies among controllers can lead to undesirable behavior in the control plane.

In this paper we introduce ERC, an Efficient, Resilient, and Consistent load migration protocol for distributed SDN controllers. Executing ERC protocol for a given switch converts a slave controller instance (which does not respond to switch queries) into the master (which should process the queries). At the same time, it demotes the initial master to a new slave instance. The result is a shift of load from the initial master to the initial slave. To the best of our knowledge, ERC is the first controller load migration protocol that explicitly deals with failures. Our protocol ensures we can roll back to an earlier, consistent state if a failure happens during the migration process by buffering all incoming messages in this period (in both the source and destination of the migration). ERC is also designed with consistency in mind, *i.e.*, it ensures all controller instances are kept in a consistent state (including the equal-mode controllers, which are neither the source nor destination of the migration).

A distinguishing factor of our work is improving the performance metrics of the migration, in spite of supporting resilience and consistency. ERC has 23-50% lower completion time compared with prior solutions. Notably, the improvement in completion time is even more apparent when the controllers are under higher stress levels (*i.e.*, when a migration is absolutely required).

ERC variants. In addition to load balancing, a fast load migration scheme can address many other problems in the network. For example, shifting the load away from a lightly-loaded controller instance and putting it in power-saving mode can be effective for energy reduction [7]. Being able to quickly migrate control traffic can also have a major impact on how fast our controller platform can react to failures. If the platform concludes that a controller instance is unable to continue its normal operation (e.g., due to a failed link or a damaged disk), then it should associate its connected switches with other operational controller instances. An agile load migration scheme can significantly reduce the response time in such cases, and ensures the network can resume its normal operation.

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 instances (or directed towards dedicated intrusion-detection systems) without impacting the normal operation of the controller platform.

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

Based on the described scenarios, we argue that exchanging master and slave controllers of a given switch (as in ERC) is not always the desired action. Therefore, we extend the ERC family by proposing four variants of the switch migration protocol. As a result, we achieve increased resilience and faster reaction times in the control plane. Through an extensive set of experiments, we show that these new protocols can improve migration and blackout times by up to 17% and 26%, respectively.

This paper is organized as follows. Section 2 provides the background and a brief overview of prior work in this area. In Section 3, we introduce our new ERC protocol, followed by its other variants in Section 4. We present the evaluation of these protocols in Section 5. Finally, Section 6 concludes the paper and describes directions for future work.

2 | BACKGROUND AND RELATED WORK

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

2.1 | Desired Properties

In an operational network, delegating the control of an active switch to a different controller instance cannot be done spontaneously. To ensure the control plane does not end up with a corrupt view of the network state, there are several properties that must uphold during the migration process.

Liveness: each switch must 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 order they are received by the control plane. For example, assume a switch sends a *link down* message to the controller, followed by a *link up* message right after. If the controller does not process these two messages in the order they were sent, it will perceive the link to be down, resulting in an incorrect view of the network state.

Safety: every asynchronous message should be responded to by at most one controller. If two controller instances respond differently to the same message (due to synchronization delays), the switch might end up with a corrupted state, which might be very difficult (if not impossible) to resolve or reverse on the switch.

Consistency: the master controller's view of a switch should be compatible with the views of all equal-mode controller instances of that switch. Otherwise, the controller instances in equal mode cannot take the responsibility of the switch during emergencies (overloaded master or node failures), which is the main reason for having equal-mode controllers.

Failure resilience: failure of destination controller instance during migration (while the initial master is still in charge) should not lead to state loss in the controller platform. During handover, at some point, the initial master should stop processing the messages received from the migrating switch. After exchanging the necessary messages, the final master will take over the responsibility of that switch. If the migration fails to finish with success, however, the initial master should be able to resume its normal operation with no state loss.

2.2 | Related Work

Switch migration in SDN controllers has been studied in various contexts. Obadia et al. proposed a solution to handle controller failures[9], in which the switches associated with a failed controller instance need to be migrated to an operational controller instance. In this work, 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 proposed mechanism is a simple role change for the new controller, and cannot be used as a building block for applications like load balancing, resource optimization, and power saving.

TABLE 1 Comparison of Openflow Controller Roles

Role	Load	Consistency
Master	High	Maintains the controller state
Equal	Comparable with master	In sync with master's state
Slave	Low (only sync. messages)	Read-only access to states

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 the 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[12,13,14,15].

Unfortunately, the 4-phase protocol lacks failure resilience *i.e.*, if the destination controller instance fails during the migration process, the entire controller system may fail. Moreover, the 4-phase protocol does not deal with controller state consistency. This solution relies on external database to store state variables, which pushes the state exchange between the instances out of the scope of their 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 view of state variables to ensure correct behavior. The 4-phase protocol also falls short in terms of efficiency, as will be shown in Section 5.

Control State in SDN. Any distributed SDN controller is comprised of various network applications. The control plane is expected to provide a mechanism to store the internal state of these applications. How the state is preserved depends on the design details of the SDN controller platform, bug regardless of those detials, the control state can be categorized as follows:

- *Switch State*. Each switch keeps some control state locally (e.g. packet/byte counters). A controller can access the data by directly querying the switch.
- *Control Application State*. Depending on the controller platform's design, network application state might be stored in a key-value store or a shared storage system[16,4,17]. Each controller instance has certain permissions to perform read or write operations on the shared storage. We note that core control applications (which are part of the network operating system) also use this storage for storing their state. The control platform or the storage system will ensure the reliability and consistency of this storage.
- *Local Controller State*. Each controller instance can have local state that is not managed by the control plane. It is usually considered to be ephemeral, *i.e.*, in case the controller instance fails, this state will be lost.

During the migration process, we are not concerned with the switch and control application states, as they are separately managed by the switches and/or the shared memory system. To ensure the continuous operation of the control plane, however, we need to ensure that local controller state is transferred during the migration process. This is one of the main responsibilities of any load migration protocol, as we will see next. We expect this state to be relatively small, for example, compared to a VM snapshot. However, we have much tighter timing requirements, which makes VM migration techniques non-optimal for our use case.

2.3 | 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, each assuming a different role. 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 the messages sent to the master, but it is not obligated to reply to these messages. There can be multiple controllers in the equal mode, all of which will receive the updates and the messages from the switch to keep the same network view as the master controller. Finally, any controller in slave mode only

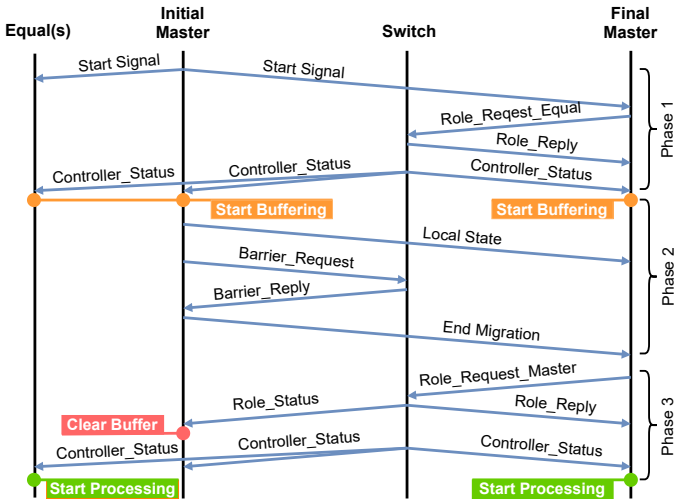


FIGURE 2 Newly proposed load migration protocol, ERC. The initial master well be demoted to a slave, while the slave will be promoted to become the master

receives a restricted subset of messages generated by the switch by default[18]. Therefore, they will not store the local controller state for the switch, as the master and equals do. Table 1 provides a qualitative comparison over the load and consistency of the described roles.

OpenFlow supports three types of messages: asynchronous, controller-to-switch, and symmetric messages. 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 messages of this type 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 status[2].
- *Packet_In*. In case the switch cannot make a decision on what to do with an incoming packet, it redirects the message to the controller. The controller, in turn, makes the decision and sends back a Packet_Out message to the switch.

The second type of messages, sent from a controller to a switch, are initiated by a controller instance to modify or inspect the state of a switch. The most important messages of this group are:

- *Packet_Out*. In response to a Packet_In message, the controller sends a Packet_Out toward the switch to update its flow table.
- *Barrier*. Switches do not necessarily process their received messages in the order they arrive. If a controller wants to dictate a specific order between the messages, it has to separate them by a barrier message. All packets that have arrived at the switch prior to the barrier will have to be processed before the switch proceeds to handle the packets arriving after the barrier.
- *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.

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

3 | ERC LOAD MIGRATION PROTOCOL

In this section, we present ERC, an efficient, failure-resilient, and consistent load migration protocol. Figure 2 shows how ERC migrates a given switch by handing its responsibility from the initial master controller to a slave instance. The initial master

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

buffers incoming messages during the migration to ensure it can recover the correct state if failures happen in this period. In addition to helping with failures, we exploit this buffer as a coordination mechanism between the initial and final master controllers, which leads to significant savings in migration time compared to prior solutions[11].

Also, as Figure 2 shows, we explicitly involve the equal-mode controllers 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 these controllers (e.g. in case of failure).

We assume both the initial and final master controllers are aware of the migration and their future roles. This can be a part of the control application (e.g. load balancer) that initiates the migration. We also assume switches send their asynchronous messages in the same order to all their corresponding controllers. We note that the final master is a slave controller at the beginning of the protocol. Our protocol consists of 3 phases, as described below.

Phase 1. The initial master controller initiates the migration by sending a "start signal" message to the final master and all the equal-mode controllers. Right after the final master receives this message, it requests a role change from slave to equal. The switch processes the role request message, and responds with a Role_Reply message back to the final master (to inform the controller of its new role). Changing to equal mode allows the final master to receive the same messages as the initial master. This is essential for the final master to build a complete and up-to-date network view by the end of the protocol.

In OpenFlow 1.5 [18], when the role of a controller changes for any reason, the switch has to inform every connected controller by sending a Controller_Status message. Therefore, 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 message, it starts buffering all messages received from the switch (e.g. Packet_In messages). At this point, the final master is in equal mode, and similarly, receives and buffers these messages. Since the switch sends its asynchronous messages in the same order to all of its corresponding controllers, and since both the initial and final masters start buffering at the same packet (right after receiving the Controller_Status message), their buffered queues will be identical. All controllers in equal mode will also start buffering messages from the switch after receiving the Controller_Status message to ensure they have a consistent view with the master controller.

Since the final master wants to take the switch's responsibility from the initial master, it needs to have the exact same state variables. The only type of state that we need to deal with is the local state of the controller, as described in Section 2.2. To this end, 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 will use the same state variables that the initial master had stored right before receiving the Controller_Status message. Since the final master receives the same messages from the switch, all state updates after this point will also be reflected in the final master.

Moreover, to make sure that the switch has no pending, unprocessed messages from the initial master, this instance will send a Barrier_Request message to the switch. This will force the switch to process all buffered messages before replying with a Barrier_Reply message back to the master.

If the initial master does not wait for the Barrier_Reply, the switch might try to process outdated messages from the initial master at a later time when it is no longer the master (and has been demoted to a slave). Therefore, those messages will be undesirably dropped at the switch. The barrier message provides a safe mechanism for committing the transactions from the initial master before handing over the responsibility of the switch to the other controller.

Phase 3. Once the final master controller receives the end-of-migration signal and 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, while it grants the master role to the final master controller instance. These controllers will be notified about their new roles thorough Role_Reply messages. The switch also broadcasts a Controller_Status message to all its corresponding controllers, including the equal and slave-mode controller instances.

The initial master recognizes that its role has changed to a slave when it receives the Role_Status message. This instance would no longer need to keep the buffered messages, and can simply clear out its buffer. The final master and other equal controllers will start processing the buffered messages after receiving the Controller_Status message.

3.1 | Validation of Properties

Here, we will show the newly proposed ERC protocol has the desired properties described in Section 2.1.

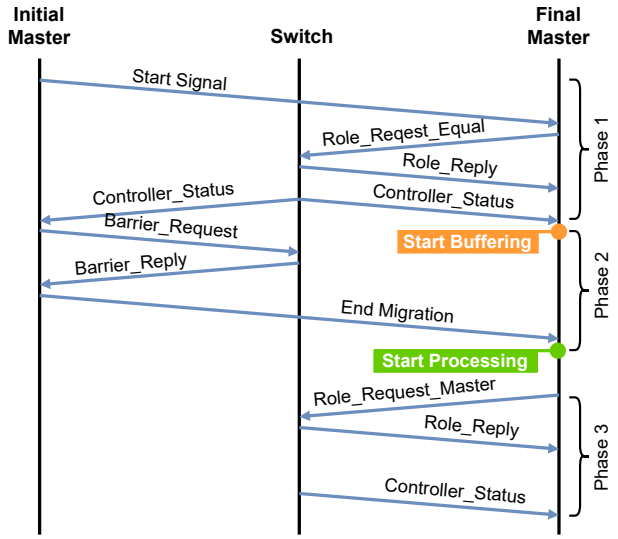


FIGURE 3 Simplified ERC protocol. Compared with the original protocol, this one does not support controller state transfer, consistency among controllers, and fault-tolerance

Liveness. During the migration process, the initial master continues to serve as the master for the switch. This role is relinquished to the final master in Phase 3 only after it has acquired the state needed to continue serving as the master for the switch. Thus, 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 so in the order they arrive. By the beginning of Phase 2, the processing is paused, and is 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 that of the initial master. By ensuring that the final master starts processing messages at the exact point (*i.e.*, message) that the initial master has paused, we guarantee that 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 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 the buffered messages. This ensures that at any moment one controller is responding to the switch, at most. Additionally, by ensuring the final master continues processing the buffered messages at the exact point the initial master has paused, we ensure that no request will receive more than one response.

Consistency. Our protocol ensures the master and equal-mode controllers start buffering messages at the same point[3]. At the end of the migration process, receiving the second Controller_Status triggers all the equals and the master to resume processing the messages. The delay in processing messages in equal-mode controllers ensures they have the same view as the initial master during the migration process.

Failure resilience. We note that a higher-level recovery mechanism is expected to detect failures and start the rollback process. Our migration protocol ensures such a mechanism has the ability to deal with failures without losing state. Throughout the migration, we assume that the switch always has a master that can be queried for as the ground truth. Otherwise, this implies a faulty behavior in the switch, which is out of the scope of the fault tolerance of the protocol.

By imposing a pessimistic timeout on every message reply that we expect to receive on the source of the migration, we can ensure that the protocol does not enter into a deadlock situation. With this assumption, not receiving the Controller_Status message in the first phase implies a broken state and we can roll back the migration plan by instructing the master controller to behave as before. Similarly, not receiving the Barrier_Reply from the switch in the second phase could mean an odd behavior from the switch. In this case both controllers start processing their buffers and we cancel the migration.

Finally, if we do not receive the last Role_Status in the third phase, we roll back to the ground truth stored in the switch. If the master controller for this switch has been updated, the final master will take over. Otherwise the initial master will continue its normal operation.

[3]Here, "same point" refers to the logical time defined by the order of messages sent by the switch.

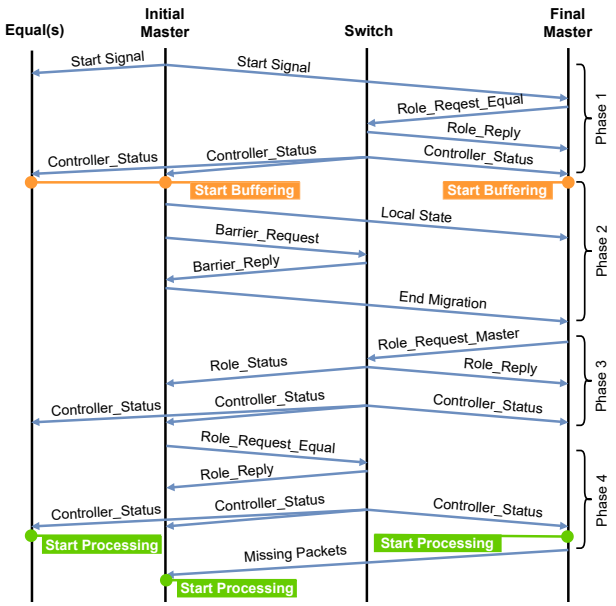


FIGURE 4 Variant 1: a master controller demotes to an equal and another slave controller becomes the master

3.2 | Simplified ERC Protocol

Figure 3 shows a simplified version of the ERC protocol. While this version can complete a master-slave controller migration, it does not support some of the desirable properties of the main protocol. This helps reduce some of the overhead imposed by the protocol. The simplified protocol will be used to perform an apple-to-apple comparison with the 4-phase protocol, which does not have resiliency and consistency properties.

In the simplified protocol, the migration will end for the initial master after sending the "End Migration" message. This instance will not buffer the messages throughout the protocol since the failure-resilience requirement is relaxed. For the same reason, the final master will not buffer the messages through the third phase, and will start processing the messages right after receiving the end-of-migration message. However, the buffering in the second phase will remain in place, to satisfy the serializability requirement. Furthermore, similar to previous works, no local state transfer will happen in the second phase, and the equal-mode controllers are not explicitly taken into account. We will compare the performance of the original and simplified ERC with the 4-phase protocol in Section 5.

4 | APPLICATION-AWARE ERC EXTENSIONS

One of the potentials of the ERC protocol is that in addition to safely handing over the master responsibilities to a slave controller, it provides a mechanism to change the role of the source master controller as well. With an implicit assumption of reducing the load on the initial controller, the ERC protocol assigns the slave role to this initial node. In general, if load reduction was not a concern, we could assign the equal mode rather than the slave. This gives us the luxury to spin up an equal instance without paying the extra time of synchronization. This means that for any migration scenario that does not intend to reduce the load on the initial controller (an example would be latency reduction), we gain a bonus equal controller, whereas if we only had the ERC protocol, then we had to run an additional synchronization protocol to promote the remaining slave from the ERC protocol to an equal instance.

In what follows, we use this idea and introduce four other ERC extensions as well as a comparison over their use cases. Note that all these protocols satisfy the five required properties of *liveness*, *serializability*, *safety*, *consistency*, and *failure resilience* as described in Section 2.1.

- *Variant 1*: initial master controller demotes itself to equal mode and another controller in slave mode changes its role to master (Figure 4). This scenario does not change the load of the master but leads to a new equal mode controller instance.

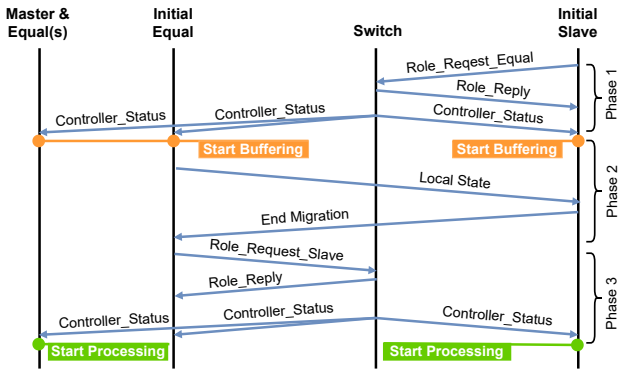


FIGURE 5 Variant 2: a slave controller takes the responsibilities from an equal controller, and this equal controller turns to the slave mode

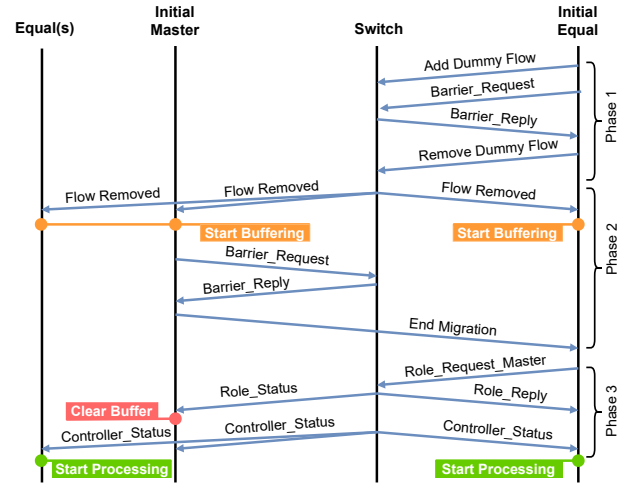


FIGURE 6 Variant 3: an equal controller takes the responsibilities from the master, and this master controller turns to the slave mode

It can be used for resource optimization while enhancing the resilience of the control plane to failures (by adding an extra equal controller).

- *Variant 2*: this protocol involves a symmetric role exchange between an equal mode controller and another controller in the slave mode (Figure 5). In other words, initial equal controller changes to slave mode, and another slave controller turns to equal mode. Unlike all other considered protocols, this one does not involve the master node. This is generally useful when we want to move one of our backup equal nodes to another faster/closer controller, or when we simply want to reduce the load from the current equal instance.

- *Variant 3*: initial master controller changes its role to slave and another equal controller takes the responsibility of the master role (Figure 6). This protocol assumes the existence of an equal controller, and as a result it improves upon the ERC protocol as it can quickly shift the load away from the master instance without waiting to transfer the local state.

- *Variant 4*: this variant is very similar to the previous one, with the slight difference that it will keep the initial master controller as an equal rather than a slave7). To do so, we have to wait for an additional extra phase to claim the equal mode, but given the already achieved synchronized state, this waiting time is much smaller than bringing up another equal-mode controller instance and wait for it to be synced with the master. As mentioned, this does not have a major impact on load balancing, but can be very helpful for adding to the resilience while moving the master controller.

Table 2 summarizes the features and differences between the introduced protocols. Here, fast reaction refers to those protocols that assume the existence of an equal in the setting and are supposed to be faster due to the fact that they do not need to communicate their local states. As this table suggests, assuming the existence of an equal, variants 3 and 4 are the faster versions of ERC and the first variant, respectively. Variant 2 is also the only protocol that contributes to load balancing without directly impacting the master controller.

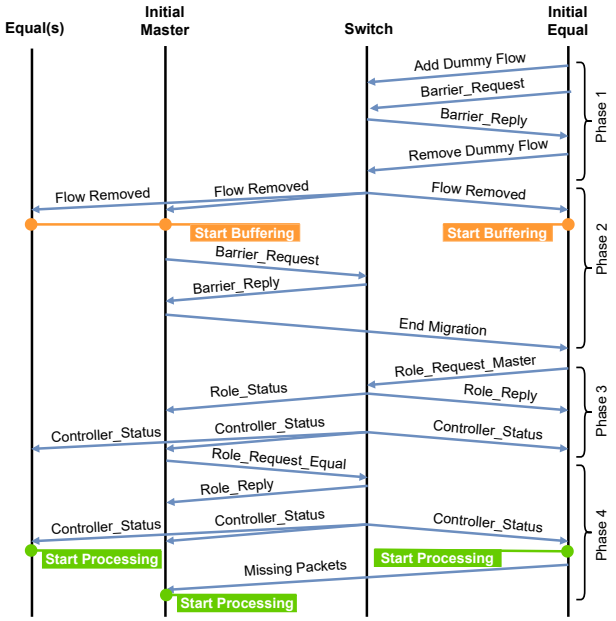


FIGURE 7 Variant 4: an equal controller takes the responsibilities from the master, and this master controller turns to the equal mode

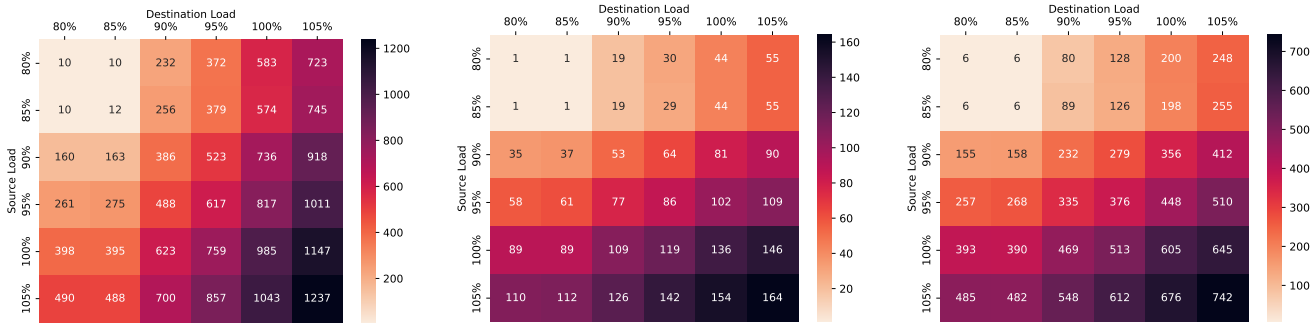
5 | EVALUATION

In this paper, we have upgraded our testing environment to emulate the switches and the controllers on a much larger scale compared with the preliminary version of this testbed[19]. We use Mininet network emulator[20] to set up the nodes and provide connectivity between them. Linux Traffic Control (TC) is used for limiting the bandwidth and latency of the links. DITG tool is used for generating the background traffic and simulating the requests. The experiments are run on two machines with 48-core Intel Xeon E7-4807 CPUs and 64 GB of memory, running Ubuntu 20.04.2.

Simulation Setup. To model the network, we set up 5 interconnected Mininet hosts to work as controllers, switches, and traffic generators. More specifically, we assume there are two controllers ($C1$ and $C2$) in each experiment acting as the source and destination of the migration. There is a single switch of interest for which we measure the desired metrics during the migration. To emulate all the other switches connected to the controllers (the ones that are not being migrated), we set up two more hosts, one connected to each of the controllers. For each background switch, these hosts will make one DITG connection to their respective controllers. For example, 200 connections are made to the controller to represent 200 switches. Through these connections, the Packet_In messages destined for the controllers are simulated. The number of the background switches connected to the two controllers varies in our experiments to change the load inflicted on the controllers. Each generator sends packets with inter-departure intervals drawn from an exponential distribution with a mean of 1000. The size of all packets (including both the simulated Packet_In queries and the protocol-related messages) are set to 250 bytes.

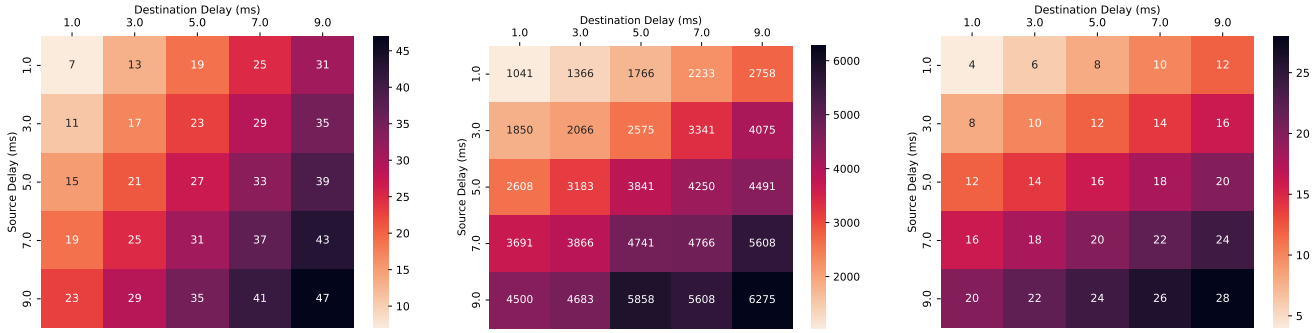
TABLE 2 Comparing the features of different ERC variants

Feature/Protocol	ERC	Variant 1	Variant 2	Variant 3	Variant 4
Load balancing	✓		✓	✓	
Added resilience		✓			✓
Fast reaction				✓	✓
Requires an equal				✓	✓
Moves master	✓	✓		✓	✓
Moves equal			✓		



(a) Total migration time (ms) (b) Buffer size (KBytes) (c) Controller blackout (ms)

FIGURE 8 Reported metrics for different sets of loads imposed on source controller (vertical) and destination controller (horizontal)



(a) Total migration time (ms) (b) Buffer size (Bytes) (c) Controller blackout (ms)

FIGURE 9 Reported metrics for different sets of delays between switch and source controller (vertical) and destination controller (horizontal)

In order to take message processing times into account in our simulations, we use a drop-tail PFIFO_FAST queue discipline on the link connecting the controller to the network (applied via the TC module in Linux). This queue limits the number of messages the controller can process per second. A 50 MBps bandwidth limit is imposed on the links, which translates to 200k packets per second in each of the links. Unless mentioned otherwise, we assume switch-to-controller and controller-to-controller delays are equal and set to 1 ms, similarly imposed by TC. The equal-mode controllers have no effect on the performance metrics we evaluate here, so they are not included in our simulations (note that the equal-mode instances should be notified by the switch to start and stop buffering their incoming messages so that they can maintain their consistency with other controllers. Their presence, however, does not affect the timing and buffer metrics in other controllers).

Performance Metrics. The experiments are repeated 30 times for each setting, and the average values are reported. If possible, the 95% confidence intervals are also shown on the graphs. The controller load levels, the delays between the nodes, and the state size are the parameters that we manipulate in this section. The metrics of interest reported through these experiments are defined as:

- *Total migration time*: shows the time required for the entire protocol to run. It measures the time interval from when the source controller sends out the start signal, to the moment the destination has assumed control of the migrating switch. This explanation applies to both of the ERC and 4-phase algorithms.
- *Buffer requirements*: depicts the amount of buffer build-up during the migration process at the destination controller (although we record this on the destination controller, it is the same across all controller instances). For the ERC protocol, the starting and stopping points for buffering are after receiving the first and second Controller_Status messages, respectively, as shown in Figure 2. For the simplified version of ERC, the finishing time for buffering is after receiving the

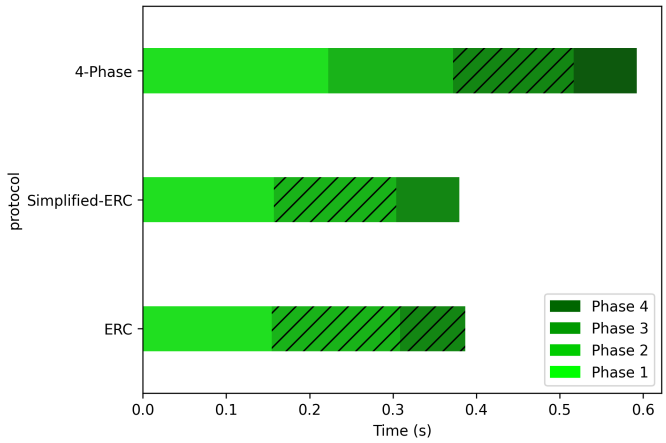


FIGURE 10 Comparing the time taken in each phase, for the three protocols: ERC, Simplified ERC, and 4-phase. The hatched boxes show when each protocol buffers the incoming requests

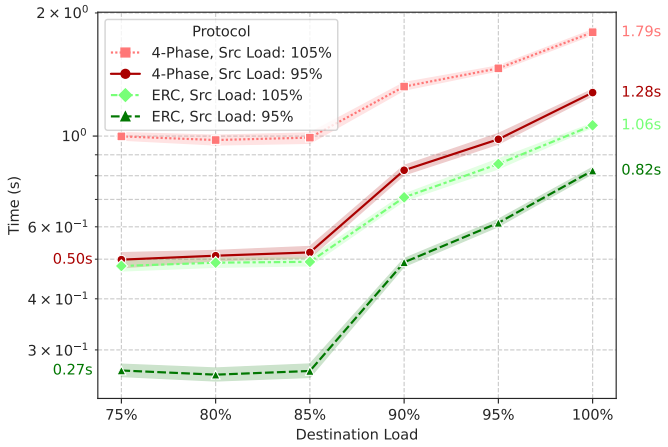


FIGURE 11 Total Migration time (s) for different destination controller loads, when migrating the switch from a highly-loaded controller to another controller

End_Migration signal. Finally, for the 4-phase algorithm, the buffering should start after receiving the Flow_Removed message at the end of phase 2, and should stop upon the receipt of the End_Migration message (which corresponds to their third phase).

- *Controller blackout period*: refers to the period of time the controller platform is not responding to the messages sent by the migrating switch. The boundaries of this period are the same as those of the buffering, delineated above.

Experiments. In Sections 5.1 and 5.2, we will evaluate the performance of the ERC protocol and compare it with the 4-phase protocol. In Section 5.3, we will vary the transferred state size to study how it affects the performance metrics. In Section 5.4 the performance of the variants will be studied.

5.1 | ERC Performance

First, we measure the performance metrics for the ERC protocol, under various load conditions. For this part, we set all delays (controller-to-controller and controller-to-switch) to 1 ms, and state size to zero. Six different load levels for the source and destination controllers are considered here, ranging from 80% to 105%, which covers highly-loaded to over-loaded scenarios. Figure 8 show the performance metrics for an execution instance of the ERC protocol. It can be seen that as the load level on the source and destination increases, the performance metrics exponentially increase. The total migration time depends more on the destination load than it does on the source load, since more interactions are made with the destination in our protocol (compare the diagonally symmetric cells in the heatmap). However, the blackout period is affected more by the source load, as it plays a more prominent role in the second phase.

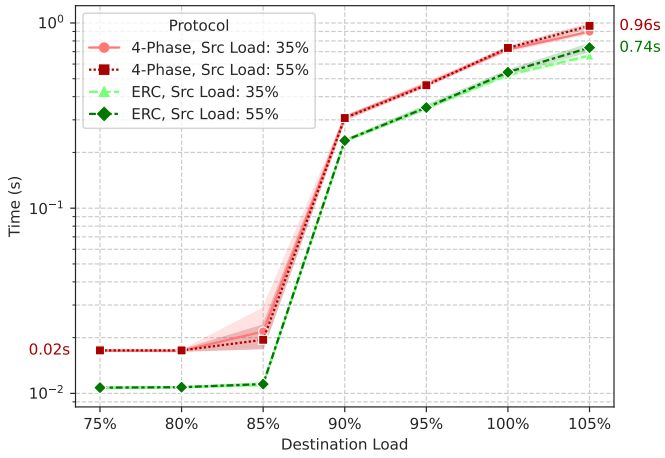


FIGURE 12 Total migration time (s) for different destination controller loads, when migrating the switch from a lightly-loaded controller to another controller

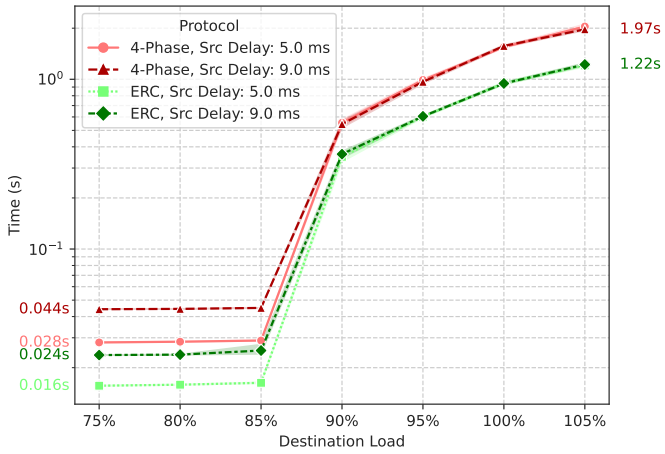


FIGURE 13 Total migration time (s) for different controller loads, when migrating the switch from a far controller instance to a closer one, under various controller loads

Next, we study the effect of the link delays on the performance of the ERC protocol. We change the delay between the switch and each of the two controllers from 1 ms to 9 ms. Figure 9 shows the reported metrics for different sets of switch-to-source and switch-to-destination delays. In each experiment, the delay between the source and destination controllers is set to be the sum of the two aforementioned values. Loads of both source and destination controllers are set to 50% here. Higher loads typically mask the effect of these delays, as their imposed performance hit is orders of magnitude larger than the numbers seen in this experiment. The results show that increasing delays can be associated with a linear increase in the studied performance metrics.

5.2 | Comparison with 4-phase

In this part, we compare the performance of the ERC protocol (along with its simplified version) and the 4-phase protocol. To this end, we apply a 90% load on the source and destination controllers, 1 ms latency on the links, and set the state size to zero. Figure 10 shows how much time is taken to complete each phase of the mentioned protocols. Regarding the total migration time, both simplified and normal ERC protocols offer significant improvements over the 4-phase protocol (35%, in this scenario). The simplified ERC protocol, which delivers the same level of desired properties as the 4-phase protocol, has a similar buffering period (shown with hatched boxes in the figure). Meanwhile, the ERC protocol has a longer blackout period, and consequently higher buffering requirements, compared with these protocols, as it will continue to buffer the requests in its final phase (51% buffer size increase in this scenario). We acknowledge that the higher buffering requirement is the price we have to pay to achieve resiliency and consistency.

To further compare the protocols, we consider three scenarios that might cause a migration event.

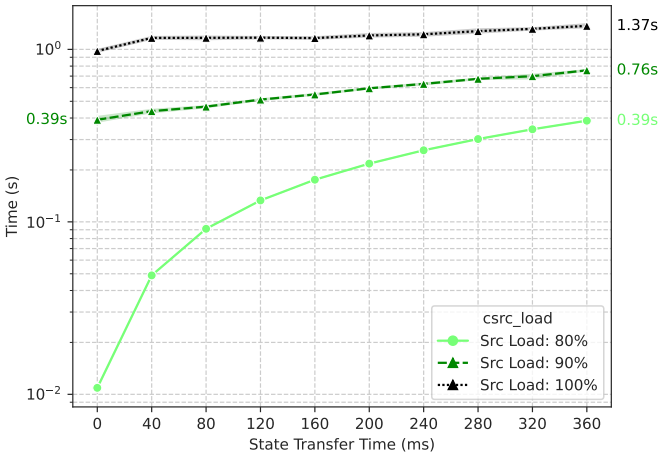
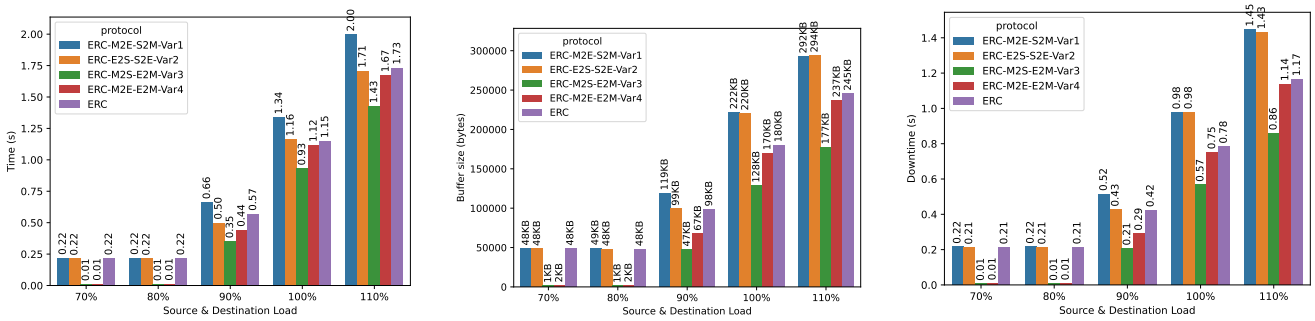
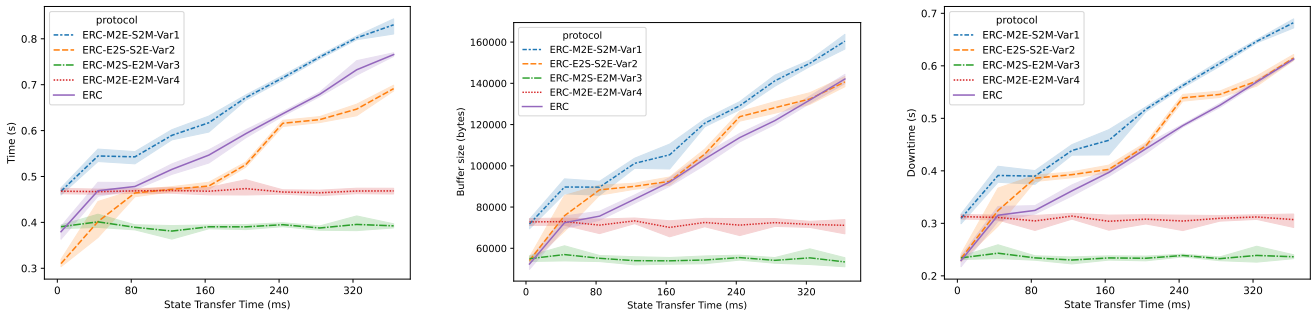


FIGURE 14 Total migration time (s) for different controller loads, as the size of the transferred state increases



(a) Total migration time (ms) (b) Buffer size (Bytes) (c) Controller blackout (ms)

FIGURE 15 The variants performance in various load levels



(a) Total migration time (ms) (b) Buffer size (Bytes) (c) Controller blackout (ms)

FIGURE 16 The variants performance with different state sizes

1. *Load balancing*: First, we consider the case where the migration is taking place to shift the load away from the highly-loaded instances toward lightly-loaded ones, to balance the load among the instances. Therefore, we set the load of the source controller at 95% and 105% (corresponding to highly-loaded and overloaded scenarios) at the time of migration. Then, we vary the load of the destination controller from 75% to 100%, and record the results, as shown in Figure 11. The results show a 40-50% decrease in the migration time in these settings.

2. *Power saving*: In this scenario, we assume the migration is occurring to reduce power consumption in the network. Therefore, the load will be shifted away from a lightly-loaded controller instance. We set the load of the source controller at

35% and 55% for this experiment, and change the destination's load. The results can be seen in Figure 11. In this case, ERC can reduce the total migration time by 23-32%, based on the set parameters.

3. *Control latency reduction*: Another motivation to migrate a switch is handing its responsibility to a controller instance that is located physically closer to it, and thus, is faster to respond. This scenario might be of interest when the latency of the responses from the control platform is critical to our application's performance. For this experiment, we set the delay between the switch and the source controller at 5 ms and 9 ms, while its delay to the destination controller is set to be 1 ms. The controller-to-controller delay is also set at 6 ms and 10 ms respectively. We assume the load of the source and destination controllers are the same in these experiments (*i.e.*, we are not trying to balance loads here). We change the loads from 75% to 105% and record the results, as shown in Figure 13. The results show that ERC can reduce the total migration time by 36-46% in the studied scenarios.

5.3 | Impact of state size

As mentioned in Section 2.2, the control platform can store local state for the switches it is responsible for. This state must be transferred during migration to avoid disruption in the controller's functionality. As can be seen in Figure 2, at the beginning of ERC's second phase, the state is sent from the source controller to the destination. However, it is not reasonable to believe that all the state variables stored for the migrating switch can be transferred in a single network packet. Therefore, we consider multiple state sizes and study how it affects the performance of the ERC protocol.

For this experiment, we set the delays to 1 ms and the load on the source and destination controllers at 80, 90, and 100 percent. We describe the different state sizes in terms of the time it takes to be sent . We artificially add delays to the protocol when the state has to be sent. The added delay ranges from zero (*i.e.*, no state transfer) to 360 ms in our experiments, through which, a state packet is sent every 4 ms to the destination. The results can be seen in Figure 14. As expected, the lower the imposed loads on the controllers are, the more impact the state transfer will have, as it will make up a greater part of the total migration time.

5.4 | Variant Performance

Figure 15 shows the performance metrics of all the introduced variants of the ERC protocol. For these experiments, we assumed a 200 ms state transfer to represent a medium state size. Then we varied the source and destination loads from 70% to 110% (equal loads were imposed on both). The results show that the added phase in the first variant incurs around 16% penalty in the total migration time and up to 25% in blackout time while adding 1 to the number of equal-mode controllers. The second variant has almost the same total time as ERC but has up to 25% higher blackout time. The third variant, which converts an equal to the master, shows the effect of not sending the local state to the destination, and helps with a 17% decrease in the total time and a 26% decrease in blackout time, in high loads. The fourth variant shows the effects of adding the extra phase to the third variant and almost undoes the benefits of the third variant to get similar performance to ERC.

In Figure 16, the effect of the state size on the performance of the variants is studied. As expected, the variants that involved state transfer (Variants 1 and 2, and ERC), showed a mostly linear increase in their performance metrics as the state transfer time increases, while the others are not affected by this parameter. This emphasizes the importance of this step in the protocol; if there are large states to be transferred, the third and fourth variants are the preferred choices to reduce the total and blackout times.

6 | 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 set of switch migration protocols that can be utilized for a wide range of network applications. Our protocols have liveness, serializability and safety along with consistency and failure resilience, which are essential properties for any distributed SDN control platform.

We evaluated our main protocol by comparing it with the best previously known solution, *i.e.*, the 4-phase protocol. Our simulations show that the ERC protocol still outperforms the original 4-phase protocol while it supports the failure resilience and the state consistency. We can still maintain a 23% to 50% reduction in the migration time.

Considering the fact that different network applications require different responses from the control plane during the migration, we pursued the idea of application-aware protocols. With the same initial design goals as the ERC, we expanded our migration toolset by supporting other role exchanges than only slave and master swap. We showed that these optimizations can lead to up to 26% and 17% reduction in blackout and total migration time, respectively,.

The ERC protocol introduced in this paper deals with the migration of 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.

7 | 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. Akyildiz IF, Lee A, Wang P, Luo M, Chou W. A roadmap for traffic engineering in SDN-OpenFlow networks. *Computer Networks* 2014; 71: 1–30.

2. McKeown N, Anderson T, Balakrishnan H, et al. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* 2008; 38(2): 69–74.

3. Koponen T, Casado M, Gude N, et al. Onix: A distributed control platform for large-scale production networks.. In: . 10. USENIX Association; 2010; Berkeley, CA, USA: 1–6.

4. Berde P, Gerola M, Hart J, et al. ONOS: towards an open, distributed SDN OS. In: ACM; 2014; New York, NY, USA: 1–6.

5. Yu M, Rexford J, Freedman MJ, Wang J. Scalable flow-based networking with DIFANE. *ACM SIGCOMM Computer Communication Review* 2011; 41(4): 351–362.

6. Benson T, Akella A, Maltz DA. Network traffic characteristics of data centers in the wild. In: ACM; 2010; New York, NY, USA: 267–280.

7. Yeganeh SH, Tootoonchian A, Ganjali Y. On scalability of software-defined networking. *IEEE Communications Magazine* 2013; 51(2): 136–141.

8. Hassas Yeganeh S, Ganjali Y. Kandoo: a framework for efficient and scalable offloading of control applications. In: ACM; 2012; New York, NY, USA: 19–24.

9. Obadia M, Bouet M, Leguay J, Phemius K, Iannone L. Failover mechanisms for distributed SDN controllers. In: IEEE; 2014: 1–6.

10. Liang C, Kawashima R, Matsuo H. Scalable and crash-tolerant load balancing based on switch migration for multiple open flow controllers. In: IEEE; 2014: 171–177.

11. Dixit A, Hao F, Mukherjee S, Lakshman TV, Kompella R. Towards an elastic distributed SDN controller. *ACM SIGCOMM computer communication review* 2013; 43(4): 7–12.

12. Dixit A, Hao F, Mukherjee S, Lakshman TV, Kompella RR. ElastiCon; an elastic distributed SDN controller. In: IEEE; 2014; New York, NY, USA: 17–27.

13. Krishnamurthy A, Chandrabose SP, Gember-Jacobson A. Pratyaastha: an efficient elastic distributed sdn control plane. In: ACM; 2014: 133–138.

14. Wang T, Liu F, Guo J, Xu H. Dynamic SDN controller assignment in data center networks: Stable matching with transfers. In: IEEE; 2016: 1–9.

15. Ye X, Cheng G, Luo X. Maximizing SDN control resource utilization via switch migration. *Computer Networks* 2017; 126: 69–80.

16. Yeganeh SH, Ganjali Y. Beehive: Simple distributed programming in software-defined networks. In: ACM; 2016: 4.

17. Macedo R, Castro Rd, Santos A, Ghamri-Doudane Y, Nogueira M. Self-Organized SDN Controller Cluster Conformations against DDoS Attacks Effects. In: IEEE; 2016: 1–6

18. Nygren A, Pfaff B, Lantz B, et al. Openflow switch specification version 1.5. 1. *Open Networking Foundation, Tech. Rep.* 2015.

19. Beiruti MA, Ganjali Y. Load Migration in Distributed SDN Controllers. In: IEEE; 2020: 1-9

20. Keti F, Askar S. Emulation of Software Defined Networks Using Mininet in Different Simulation Environments. In: IEEE; 2015: 205-210

AUTHOR BIOGRAPHIES

Sepehr Abbasi Zadeh is a Ph.D. Candidate in the Computer Systems and Networks Group at the University of Toronto, Canada. He obtained his M.Sc. at the same department, and prior to this, he completed his B.Sc. major in Computer Engineering and minor in Mathematics at Sharif University of Technology. His research interests include designing network migration protocols and optimizations towards achieving large-scale migrations.

Farid Zandi is a Ph.D. student at the University of Toronto. He has been a researcher in the Computer Systems and Networks Group since 2020. He had previously received his B.Sc. degree from Sharif University of Technology. His research interests include software-defined networks, state managements in network functions, and migration protocols.

Yashar Ganjali is a chief scientist, and the director of Data Center Networking Laboratory at Huawei, and a professor of computer science at the University of Toronto. His research interests include data center networking, packet switching architectures/algorithms, software deïňĄned networking and controller design, network application integration, and congestion control. His work on router buffer sizing, as well as distributed controllers for software-defined networks has had a major impact on research trends in these areas, and major practical implications for industry. Dr. Ganjali has received several awards for his research including best paper award in Internet Measurement Conference 2008, best paper runner up in IEEE INFOCOM 2003, best demo runner up in SIGCOMM 2008, first and second prizes in NetFPGA Design Competition 2010, best paper award IEEE/IFIP Network Operations Management Symposium 2020, Cisco Research Award, and a Distinguished Faculty Award from Facebook. Dr. Ganjali has served as a member of the executive committee for ACM SIGCOMM (2013-17), as well as an associate editor for IEEE/ACM Transactions on Networking (2014-18).