Meta-Migration: Reducing Switch Migration Tail Latency Through Competition

Sepehr Abbasi Zadeh, Farid Zandi, Matthew Buckley, Yashar Ganjali
Department of Computer Science, University of Toronto
{sepehr,faridzandi,mbuckley,yganjali}@cs.toronto.edu

***Abstract*—Resource management in distributed network control planes plays a vital role in the performance of the data plane and therefore the performance of network applications. Overwhelmed controller instances or underutilized instances could reshape their workloads by exchanging their load, i.e., switches that they control. To safely implement this exchange procedure, switch migration protocols are being used. As the migration procedure pauses processing new flows for a few milliseconds, these protocols are designed to be as fast as possible. Faster protocols add to the agility of the network to rapidly cope with the changing demand.**

In this paper, we introduce a general framework, called Meta-Migration, which focuses on expediting the existing time-sensitive controller load migration protocols. Based on the observation that these protocols impose low overheads on the involved parties, we modify them in a way that they can run in parallel toward multiple candidate destinations. Unlike the usual *Fixed* protocols that have to decide their destinations before running the protocol, here we rely on the real-time probes that we obtain from multiple systems and commit to only one of them in the middle of the procedure. Typically, migrations can complete on sub-second timescales, but sudden traffic bursts or system-level glitches can significantly slow down these protocols. We observe that by using Meta-Migration, we can dramatically diminish these negative effects. We show theoretical justifications for why this approach improves the overall performance of the migration, namely, its mean finishing time, and the tail latency of the migration. In addition, by developing a distributed controller simulator over real physical devices, we thoroughly measure the effectiveness of this approach as well as its incurred overheads. Our testbed results show up to a 53% tail reduction in the migration time.

I. Introduction

The traffic in cloud environments is controlled via a set of distributed controllers collectively known as the network control plane. Each of these controllers is responsible for managing the incoming requests from a number of (possibly software) switches. Under the OpenFlow protocol [15], this management provides access to the data plane. As the traffic patterns in the network change, the load on the controllers also becomes volatile. The load imbalance on the controllers results in a sequence of undesirable consequences. An overwhelmed controller instance inevitably experiences higher buffer thresholds which increase the response time. As a result, the traffic burstiness increases which degrades the performance of the data plane as well. Thus, the load on the controllers is a crucial performance factor that needs to be governed.

ISBN 978-3-903176-57-7©2023 IFIP

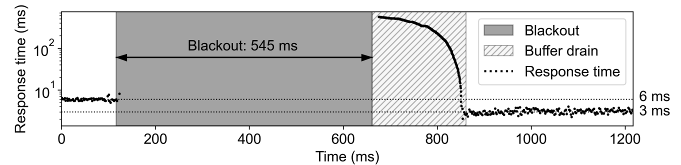


Fig. 1. Changing the corresponding controller of a switch can improve the performance of the control plane by reducing the response time of queries.

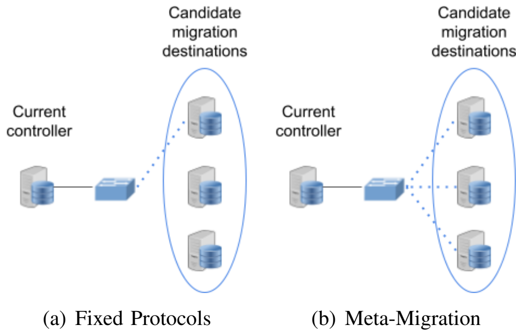
Switch migration protocols serve as a reliable solution to adjust the load distribution on controller instances [10], [6]. Using load migration, the total load on a controller instance could be changed at the granularity of the imposed load by each switch. More specifically, a switch migration protocol changes the corresponding controller of a given switch to another controller instance. This effectively reduces the load on the initial controller of the switch under the migration. Fig. 1 depicts the benefits of the migration procedure from the switch's point of view. In this example, a switch is migrated from a high-load controller instance to another with a lower load. We can see that after the migration and buffer draining, the response time to the switch's queries decreases eventually (from 6 ms to 3 ms) as the new controller has a higher capacity. However, this procedure comes at the cost of a short period of service *blackout* for the migrating switch.

During the blackout phase, all the incoming requests from the switch will be buffered to be responded to as soon as the migration ends. To minimize the mentioned buffering consequences and have a more agile network, the migration blackout period should be as short as possible. In response to that, traditional migration protocols (coined Fixed protocols in this paper) are optimized to accelerate the migration procedure.

A. Fixed Migration Protocols

Traditional migration methods rely on running protocols between a known source and a chosen destination. This means that as soon as the control plane decides on migrating the load of a switch from its current controller instance, it should also announce the destination controller of the migration to start the procedure [10]. This is illustrated in Fig. 2(a).

To optimize the performance of the control plane, this decision usually takes into account the recent statistics of all the possible candidates for the destination controller [14], [9]. While these optimizations can lead us to an eventually good



(a) Fixed Protocols (b) Meta-Migration

Fig. 2. Comparison of the two methods.

candidate, they fail to anticipate unforeseen network/system conditions. A sudden random drop of key migration protocol message, a surge in traffic, micro-bursts, or even micro-second level failures might cause unexpectedly long migration times. This can have a damaging effect on the blackout time of the migrating switch. Similar conditions could occur due to the softwarized nature of the controller instances. For example, the hosting system of the destination controller instance might be experiencing some system glitches or short service disruptions due to a garbage collection task. These unpredictable problems happen infrequently, but can significantly impact the migration time, leading to long tail latencies.

B. Meta-Migration Protocols

Fixed migration protocols involve exactly 3 parties in their procedure: 1) the switch to be migrated, 2) the source (i.e., current) controller instance, and 3) the destination controller instance. The idea behind the framework we propose is to involve multiple instances in parallel as the third party of the Fixed protocols, called *candidates*. Fig. 2(b) shows this setting.

We aim to run the load migration procedure for all these possible candidates in parallel without any race conditions and commit only to the one that finishes faster.

Having the ability to run multiple instances of a migration protocol for multiple candidates can reduce the chances of hitting the mentioned corner cases in the traditional Fixed protocols. This is mainly because those mentioned cases usually impact the environment locally (temporally and spatially). As an example, in an operational network, software glitches don't happen to all the controller instances simultaneously. Similarly, intermittent micro-bursts usually happen in certain traffic paths that employ incast or outcast patterns[5] which again only influences the bottlenecks that the casting endpoint shares with other candidates.

Therefore, involving more systems in the protocol means smaller chances of simultaneous unwanted events and it becomes less probable that multiple candidates are affected. It also eliminates the effect of *stale information* that negatively influences the choice of Fixed migration protocols [19]. [1]

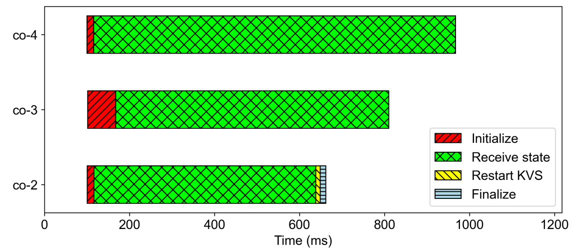


Fig. 3. Concurrent Controller Migrations: Meta-Migration execution stages for migrating a switch from `co-1` to any of the three migration candidates. `co-2` finishes first and becomes the new controller for the switch.

These benefits over Fixed protocols do not come for free and it should be noted that we are temporarily sacrificing more system/network resources to obtain the final controller among a pool of candidates. However, note that this temporary sacrifice incurs negligible overhead on the involved parties as we will show in Section IV. In the realm of time-sensitive applications such as AR/VR, self-driving cars, or tactile internet with their stringent latency requirements, this cost can be justified with the saved time from the tail of the migration time (e.g., as much as 53% in our experiments).

Fig. 3 shows an example of how such a protocol works. The migration is initiated from `co-1` toward three candidates, `co-2`, `co-3`, and `co-4`. While `co-2` and `co-4` can finish the initialization phase faster, `co-3` is slowed down because of an internal or network-related issue. All candidates then enter the state transfer phase, in which `co-2` happens to finish first. This controller finalizes the migration and becomes the new controller instance for the switch, processes the buffer, and the switch operation goes back to normal again. If a fixed choice had to be made for the destination, the slower controllers could have been chosen, leading to a longer service blackout for the switch. On a similar note, if the choice had to be made between `co-3` and `co-4` after the initialization phase, the suboptimal `co-4` would have been chosen.

In order to safely run this new family of protocols, which we call Meta-Migration, the mechanism should assure us that:

1) Eventually, exactly one controller instance obtains the control for the migrating switch, or otherwise, the migration is aborted.
2) The final chosen destination controller has completely received its required state to continue its operation.
3) The protocol can terminate the migration for the other candidates that are not involved in the migration anymore so that they can also release the reserved resources and roll back any undesired state changes.

[1]From a different perspective, this framework could be viewed as yet another application of the "power of k choices"[16] but in an online fashion. Within that context, it becomes more intuitive to realize the effectiveness of the approach. However, the math used there for load balancing notions is not directly applicable to the statistics of interest in our framework, specifically, the tail latency. Thus our analysis is problem-specific and different from the original proofs.

Having such a mechanism, the control plane can benefit from the fact that, unlike the Fixed protocols, we are less likely to hit one of the mentioned corner cases.

In this work, we make the following main contributions:

- We propose a general framework for converting existing switch migration protocols to a Meta-Migration one with the objective of reducing worst-case migration latency.
- We develop theoretical justifications for this framework.
- Through extensive testbed experiments, we show that with minimal network/CPU overheads, the framework significantly reduces migration time in the worst case.

We define the necessary conditions under which such a framework can be applied and focus our evaluations on the switch migration setting. In support of the justification that our theoretical model provides, the evaluations demonstrate that Meta-Migration protocols can be effective in real settings. The reduction in 99th percentile latency ranges from 15% to 53% in our experiments. This demonstrates the robustness of the method which is due to the fact that it is able to pick a different candidate controller when unexpected delays are introduced. Moreover, even in some cases where there are no unexpected delays, a Meta-Migration protocol is able to reduce the tail latency. Even though the additional overhead is small, we envision this framework being used for particularly time-sensitive migrations, in order to guarantee a better worst-case latency.

II. Background and Related Work

For the purpose of protocol transformation, we have chosen the ERC family of migration protocols that support the roll-back mechanism by design [6], [1]. This mechanism would allow us to easily ignore the unsuccessful candidates by issuing a rollback command for them. We would like to emphasize that any protocol with rollback support can be used as the basis for our solution. This is a reasonable requirement as any load migration protocol that is able to deal with failures needs to have the ability to roll back the controller state.

A. ERC Fixed Protocols

The ERC family of protocols is designed for switch migration tasks with the additional property that they maintain the consistency of the locally stored states between the source and the destination of the migration. The state consistency maintenance property provides transparency for the control applications operating in the controller at the cost of a short *blackout period* in which the controller is not responding to the switch queries.

Based on the OpenFlow specifications, each controller of a switch can be either in master, slave, or equal role. The master controller instance is the only controller that can modify the switch's state and the equal controllers maintain a synchronized view of the switch with the master controller. Meanwhile, slave controllers only receive a portion of control messages and they only have read access to the switch.

The protocol, as shown in Fig. 4, is designed to exchange the roles between a master (i.e., initial master) and a slave instance

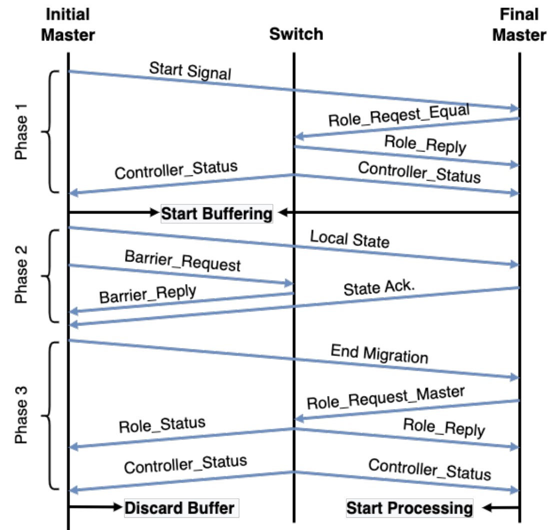


Fig. 4. ERC Fixed Protocol

(i.e., final master). Towards that end, it starts the procedure from the source controller. At the end of the first phase of the protocol, both controllers have reached an agreement that they are in the middle of a migration procedure. Once the destination controller accepts its new *equal* role at this phase, a copy of all the switch requests is guaranteed to be received at the destination (this is enforced by the OpenFlow specifications for any equal instance). In order to maintain state consistency, both controllers start buffering newly incoming requests so that whoever obtains the master role at the end of the migration can respond to them. Upon reaching the end of the second phase, the controllers have ensured that their required local states are synchronized, allowing them to proceed to the next phase. Finally, the third phase is transferring the master role responsibilities associated with the migrated switch to the destination of the migration.

Through the final two phases, the switch is experiencing its blackout time while all of its requests are being buffered at the controllers. If any undesired event (other than losing the initial master) stops the migration from proceeding. In this case, the initial controller issues a rollback command; starts to process its buffered requests; and eventually continues its normal operation.

B. Related Work

Switch migration is an important tool for managing the distribution of the control traffic in distributed Software-Defined Networks (SDNs). As this procedure effectively moves the control of a switch from one controller instance to another one, it has been used as a building block in various control-plane load-balancing methods [14], [18], [26]. These methods consider a migration protocol as given, and their main objective is on optimizing either the controller assignment or placement[17], [23], migration scheduling [8], or migration/resource cost[22], [12].

As for the employed migration techniques, these works usually consider a 4-phase protocol, based on OpenFlow 1.3, proposed by Dixit et al. [10]. Even though this protocol provides the required migration functionality, it lacks a number of important features such as failure resiliency and state synchronization in a distributed manner. To fill these gaps, [6] introduces a new 3-phase protocol called ERC that leverages OpenFlow 1.5 and outperforms the older protocol in terms of its migration time. While both of these protocols migrate a switch via a role exchange between a master and a slave controller instance, [1] extends the ERC family by identifying other migration use cases that allow faster and more efficient role exchange protocols involving the equal controllers as well.

To the best of our knowledge, there are currently no works on the use of a cooperative and competition-based framework like Meta-Migration in migration schemes. However, there have been prior studies with similar ideas in adaptable network algorithms. Remy [24] is a framework in which a computer generates congestion control algorithms based on users' assumptions about the network and high-level objectives. It is based on a game-theoretic paradigm in which all network elements converge to a common algorithm based on the network state. CCmatic [4] is a framework that uses formal logic to synthesize congestion control algorithms. It compares different congestion control heuristics and is able to select and combine algorithms to work under specified network conditions. It also identifies underlying assumptions in the algorithms that must hold in the network to ensure the desired performance criteria are met. DCM [21] allows clients to declare cluster management policies using SQL queries over cluster state stored in a relational database. The framework combines cluster state and user-specified policies to derive an optimization model for cluster management decisions. For the load-balancing problem in the data plane, Malcolm [3] proposes a decentralized design for distributed servers and micro-second scale workloads. The problem is modeled as a cooperative stochastic game. The scheduling decisions are made by distributed Malcolm nodes that maximize their own utility by migrating workloads to other nodes when experiencing overutilization and stealing workloads from other nodes when they are underutilized. In the context of blockchains, DispersedLedger [25] tries to address the issue of broadcasting a block of data to multiple destinations. By dividing the block into smaller chunks and using erasure coding, the block proposer ensures one straggler destination cannot slow down the entire system which effectively cuts the tail latency. As long as a majority of the destinations continue their normal operation, every destination can eventually build the data block from the available chunks.

III. Design Rationale

In this section, we first argue why the proposed technique results in better migration performance, and then we identify the changes necessary to convert a Fixed protocol to a Meta-Migration protocol.

A. Justification

The primary motivation for Meta-Migration is to reduce the time required to complete a migration. Numerous studies have shown that the distribution of datacenter traffic is heavy-tailed [11], [5], [7]. Such a heavy tail can contribute to events such as buffer-buildup, congestion, or system errors. Thus, although the network typically performs very well on average, it can experience much longer delays in rare cases.

In the context of switch migration, events caused by heavy-tailed traffic and longer delays can lead to transient degraded performance from the switch and/or controllers involved in the migration. In addition, events such as micro-bursts, which occur frequently in datacenters [7], can lead to increased response time. These factors are expected to contribute to a heavy-tailed distribution for the migration time. This is confirmed by results of previous experiments on switch migrations [10] as well as our own results. These distributional properties imply that many migrations can be finished quickly, but a small proportion requires a much longer time. Using Meta-Migration, multiple migration protocols to different destination controllers can be initiated in parallel. The migration that finishes first is taken and the others are rolled back. This has the effect of shrinking the heavy-tail of the time to complete a migration.

More formally, suppose switch s can be migrated to any of n controllers $c_0, c_1, ..., c_{n-1}$. As a candidate heavy-tailed distribution, we suppose that the migration time of switch s to controller c_i follows a Pareto distribution[2] with scale parameter k_i and shape parameter α_i. Let X_i be a random variable denoting this migration time. Then the probability density function of X_i is given by

$$f_{X_i}(x) = \begin{cases} 0 & \text{if } x < k_i, \\ \alpha_i k_i^{\alpha_i} x^{-(\alpha_i+1)} & \text{otherwise.} \end{cases}$$

And the cumulative distribution function is given by

$$F_{X_i}(x) = \Pr(X_i \leq x) = 1 - \left(\frac{k_i}{x}\right)^{\alpha_i}. \tag{1}$$

Thus, under the assumption of a Fixed migration protocol, some controller c_i, $0 \leq i < n$, is chosen and the probability that the migration takes no longer than x is given by $F_{X_i}(x)$. In contrast, for a Meta-Migration protocol, we choose the destination controller that can complete the migration the fastest. This corresponds to the minimum of the random variables and assuming independence gives the following result.

Lemma 1. *Consider n independent random variables X_i, $0 \leq i < n$, such that X_i follows a Pareto distribution with scale parameter k_i and shape parameter α_i. Let $X = \min\{X_0, X_1, ..., X_{n-1}\}$. Then the cumulative distribu-*

[2] Although the specifics of this distribution may not apply in practice, this is merely used to illustrate the effect of Meta-Migration protocols. Pareto distributions have previously been used in the context of data center workloads [2], but a similar analysis applies to other heavy-tailed distributions.

tion function of X is given by

$$F_X(x) = Pr(X \leq x) = 1 - \prod_{i=0}^{n-1} \left(\frac{k_i}{x}\right)^{\alpha_i}.$$

This result follows from standard probability arguments, but the proof is provided in the appendix for completeness.

Notice that for X_i, the cumulative distribution function is non-zero only when $x \geq k_i$ and in particular at the tail $x >> k_i$. Thus, comparing the result of the lemma with (1) shows that a Meta-Migration protocol has the effect of shrinking the tail of the migration time distribution. Put another way, *for a given large threshold x, the probability that a Meta-Migration protocol completes within time x is higher than the probability that any Fixed migration protocol finishes within time x for the same set of controllers*. This effect is shown in a sampling process from different Pareto distributions in Fig. 5.

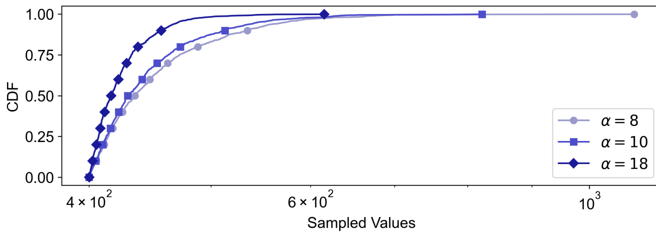


Fig. 5. CDFs of samples from Pareto distributions with scales $k = 400$ and different shape parameters α. From (2), the Pareto distribution with $\alpha = 18$ results from taking the minimum of the $\alpha = 10$ and $\alpha = 8$ Pareto distributions, and results in a significantly reduced tail compared to the individual distributions.

This becomes even more clear if we assume that $k = k_0 = ... = k_{n-1}$ in which case, the cumulative distribution function given in the lemma simplifies to

$$F_X(x) = \Pr(X \leq x) = 1 - \left(\frac{k}{x}\right)^{\sum_{i=0}^{n-1} \alpha_i}. \quad (2)$$

For the remainder of the derivations, we make this simplifying assumption on the scale parameters k_i. We now proceed to examine the effect of a Meta-Migration protocol on the mean. Note that using the definition of $f_{X_i}(x)$ we have that

$$E[X_i] = k\Big(\frac{\alpha_i}{\alpha_i - 1}\Big) \qquad \forall 0 \leq i < n, \alpha_i > 1; \quad (3)$$

Next, define $\alpha = \sum_{i=0}^{n-1} \alpha_i$. Substituting this into (2) gives $F_X(x) = 1 - \left(\frac{k}{x}\right)^{\alpha}$ and differentiating gives that $f_X(x) = \alpha k^{\alpha} x^{-(\alpha+1)}$ if $x \geq k$ and 0 otherwise. We then see that

$$E[X] = k\Big(\frac{\alpha}{\alpha - 1}\Big) = k\Big(\frac{\sum_{i=0}^{n-1} \alpha_i}{\sum_{i=0}^{n-1} \alpha_i - 1}\Big). \quad (4)$$

Comparing (3) and (4) we see that there is also a small decrease in the mean completion time under a Meta-Migration protocol as compared to a Fixed migration protocol.

This model provides the basis for the Meta-Migration framework under the simplifying assumption of independence[3] As shown in Section IV, the protocol adds additional overhead to the switch and the controllers during the migration process. However, the large reduction in tail latency more than offsets this overhead, especially in the context of delay-sensitive network applications.

B. Required Protocol Changes

In what follows, we describe the required changes for the ERC protocol as it has been chosen for our migration baseline. However, the same changes could be applied to any other migration protocol.

Due to the nature of the control plane, and the rigidity of the switches, the migration protocols are always initiated from the controllers rather than the switches. This gives us leverage in terms of choosing the synchronization point of the Meta-Migration protocols. In other words, we can assume that the controller that initiates the migration has full power to manage all the candidates. This power allows the initiator to bear the full responsibility of synchronization and therefore the initiator can run multiple migrations towards all the candidates in parallel. Then, the initiator must decide to commit to only one of these parallel migrations at some point and generate a rollback command towards all the other *unsuccessful* candidates. In this way, the underlying protocol remains intact with the initiator maintaining the relative state for each candidate separately to choose the successful one. This state corresponds to the status of each migration in terms of the migration phase that the candidate has achieved so far at any given point in time. We define the notion of *committing point* to be the stage in which the initiator makes a decision about the successful candidate.

The committing point in the ERC protocol could be chosen to be either at the end of Phase 1, i.e., after receiving the first `Controller_Status` from the switch, or before sending the `End_Migration` message at the end of Phase 2. As soon as the first candidate controller reaches its committing point, the initiator sets the state of that controller to become a successful controller and continues the migration with that one. At the same time, it sends a predefined rollback message to all the other unsuccessful candidates so that they can release the resources of the ongoing migration.

Note that as soon as the initiator declares a candidate as a successful one, it can ignore all the migration-related messages from the unsuccessful candidates and as it is the point of synchronization of the Meta-Migration protocol, there would be no race condition between the other controllers to continue their migrations. In addition, the described transformation preserves all the design properties of the initial ERC Fixed protocol such as its state consistency. Also, note that this design allows us to easily push some functionalities to the hardware which will be discussed in Section V.

[3]While true independence is likely not the case in real networks, the issues we are designing for are mostly local to a given switch; software issues are specific to a switch and micro-bursts are localized issues. Furthermore, we can select migration candidates to maximize independence.

IV. Evaluations

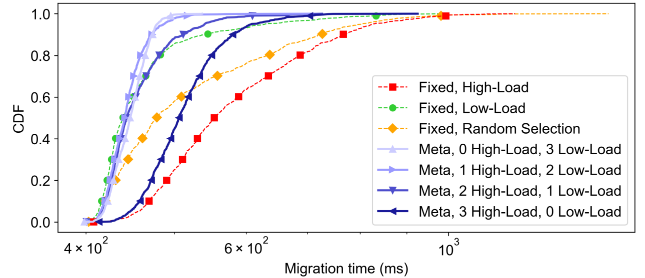
A. Methodology and the Testbed Description

We constructed a testbed for emulating the migration of a switch between controllers, using a cluster of 9 machines with Intel Xeon E7-4807 processors and Ubuntu 20.04. One machine served as the experiment coordinator, 4 as OpenFlow switches, and the remainder as network controllers. The switches set up message publishers, to which all connected controllers (in both master and equal modes) subscribe. In normal operation mode (*i.e.*, no migrations taking place), each switch connects to a single controller in master mode and generates `Packet_In` messages. These queries are sent with intervals drawn from a Poisson distribution. Note that this is only a simulation of the traffic between the switch and the controller, not the data-plane traffic through the switches.

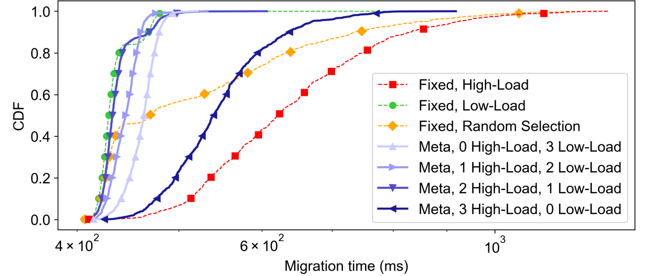
The master controller for each switch responds to the queries with `Packet_Out` messages after interacting with a local key-value store (KVS) to imitate a stateful control-plane application. The controllers also establish a publisher-subscriber system of their own, mainly to coordinate the migration process. We implemented the switch and controller functionalities using Python3 scripts, utilized `ZeroMQ` for inter-device communications and used `Redis` as the local KVS. Additionally, to simulate real-world scenarios, we added extra load on the controllers using the `stress-ng` module.

In our experiments, we study the migration of a single switch from its initial master controller to a different instance. The initial master controller initiates the ERC migration protocol towards all participating destinations, which compete to become the new master for the switch. The winning controller finalizes the migration by promoting itself to the master mode. The new master initiates its local Redis instance with the received state and begins processing the switch queries that were buffered during the migration. The other controllers pause their migration efforts by unsubscribing from the switch, discarding their buffered messages, and relinquishing their equal role for the switch (acquired after Phase 1). Since the Equal role is used only for high-availability purposes, adding and removing these equal-mode controllers does not cause any issues for the switch.

In the following sections, we will study different aspects of Meta-Migration by changing the number of participating destinations, the load imposed on the controllers, and the committing point for the migration. We record the total migration time, which serves as the main indicator of the protocol performance, as well as finer-grained measurements for the different stages of the migration. We also measure the processing and network overhead caused by the migration to show the additional cost of adopting Meta-Migration. As the baseline, we run the Fixed ERC protocol toward lightly-loaded and heavily-loaded controller instances, as well as a random choice between the two. For each configuration, we have repeated our experiments 1000 times, and the cumulative probability distribution of the desired metrics is reported.



(a) 50% as low load, 75% as high load



(b) 10% as low load, 90% as high load

Fig. 6. Comparison of migration time distribution between Meta-Migration and Fixed protocol baselines, with various load combinations on the three Meta-Migration candidates. The markers show every 10th percentile as well as the 99th.

B. Effect of the destination load on tail latency

Fig. 6 compares the CDF of the total migration time between the Fixed protocol and the Meta-Migration protocol. In this experiment, Meta-Migration is run with three candidate destinations. In each configuration, the candidates are divided between having low or high loads imposed on them. Setting 50% and 75% loads as low and high (Fig. 6(a)), Meta-Migration reduces the 99th percentile of migration time by 28% compared to the random selection baseline if all candidates have a high load. If there are low-load destinations among the candidates, the 99th percentile is reduced by 38% to 49% depending on their count. Meta-Migration can even beat the Fixed low-load protocol in 40% of instances and reduces the 99th percentile by 15% to 40%.

Considering the more extreme case of having 10% and 90% as our low and high loads (Fig. 6(b)), the gains in 99th percentile range from 27% to 53% compared to the random selection baseline depending on the number of low-load candidates. If there are any low-load destinations among the candidates, Meta-Migration delivers a shorter tail latency compared to the Fixed low-load protocol, but consistently performs slightly worse than the Fixed protocol in the normal case, due to the overhead of running multiple migrations at the same time from the source. Surprisingly, however, having more low-load destinations makes the migration time worse.

The reason for this outcome can be seen in Fig. 7, which compares the different combinations of low and high load destinations with more detail. When there is only one low-load destination among the three candidates, it is more likely that

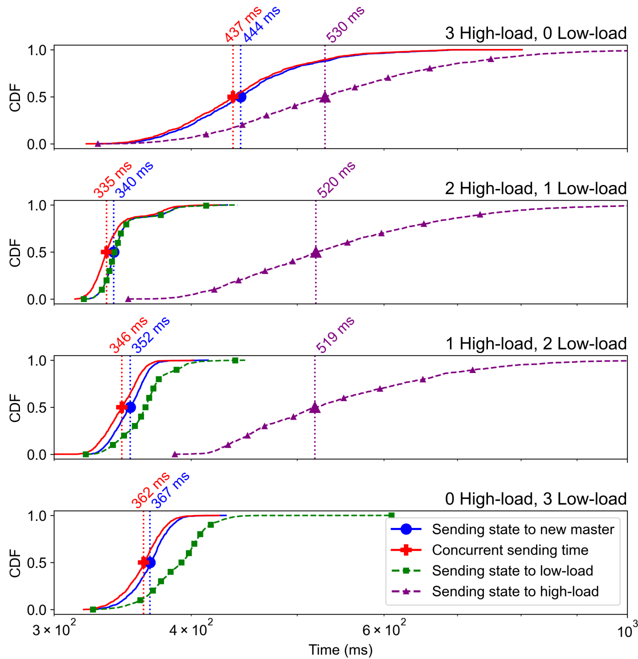
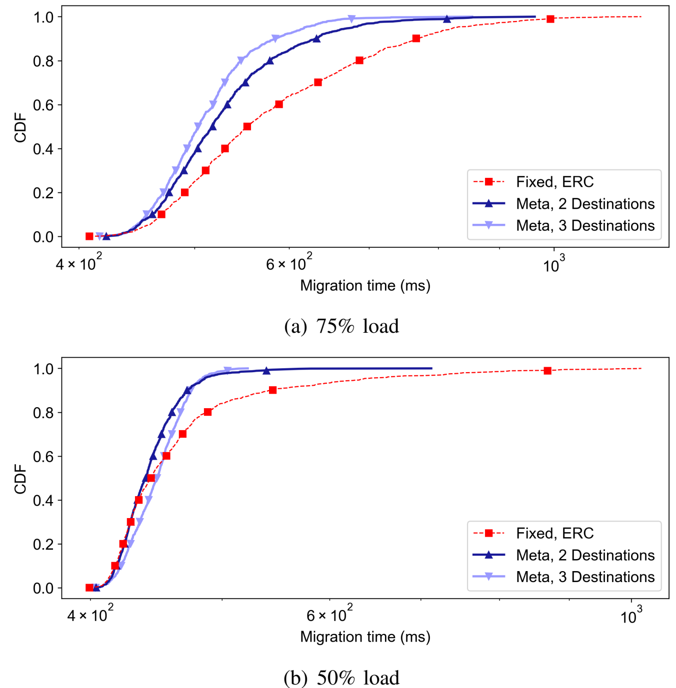


Fig. 7. Comparing the state transfer time from the initial master toward the destination candidates with different combinations of loads. High and low loads are 90% and 10% for this experiment. Having many candidates with the same load increases resource contention and increases transfer time.



(a) 75% load

(b) 50% load

Fig. 8. CDF of the total migration time when changing the number of migration candidates under different loads. While the heavy tails of the distribution are decreased by adding more candidates, the normal-case runtime might suffer due to the added overhead.

the state transfer to this destination has a head start compared to the other candidates. As the number of low-load destinations increases, it will be more likely that multiple transfers take place at the same time, leading to contention over the available resources, and finally increasing the transfer time even for low-load destinations. The red line in Fig. 7 shows the amount of time that the initial master is transferring the state to all three destinations at the same time, which consistently increases as the number of low-load candidates increases from one to three. Therefore, while having low-load destinations among the candidates is generally good, some variation among the loads helps with avoiding resource contention.

C. Number of candidates and resource overhead

Fig. 8 shows the effect of varying numbers of candidates on the migration time distribution. With a 75% load imposed on all migration candidates, as in Fig. 8(a), increasing the number of candidates from one (Fixed ERC) to two, reduces the median and the 99th percentile by 7% and 18%, respectively. Increasing the number of candidates to three increases the gains to 9% and 32%. With lower loads imposed on the controllers, however, the effect of increasing the number of candidates is more nuanced. As seen in Fig. 8(b), increasing the candidates to two reduces the 99th percentile by 38%, but in 40% of the runs, there is almost no difference between the two. While increasing the candidates to 3 can reduce the 99th percentile by 42%, it makes the migration time slightly worse in 60% of the runs, due to the overhead of Meta-Migration.

The overhead comes in the form of processing and network load. Fig. 9 compares the distribution of the imposed CPU

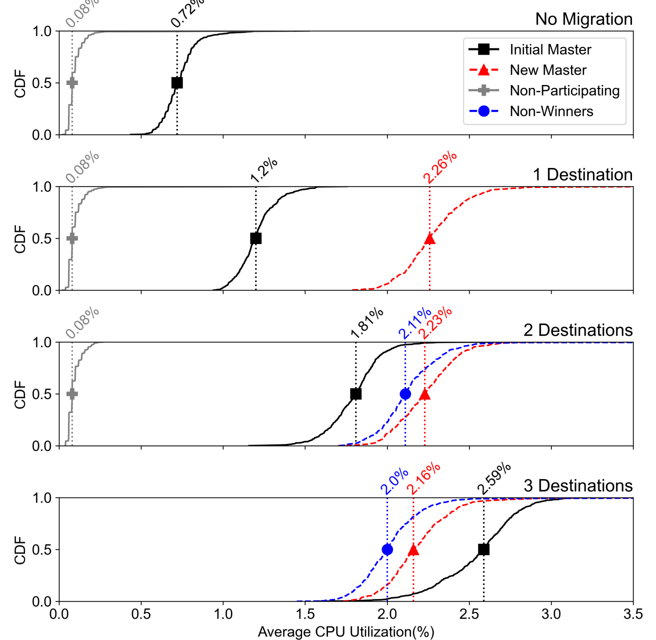


Fig. 9. CDF of CPU usage on the controllers during a migration. Adding more candidates increases the processing overhead on the initial master, who has to handle multiple concurrent migrations.

load as a result of the migration, as the number of Meta-Migration candidates increases. For this graph, we turned off the extra load source (`stress-ng`) to obtain the net operation overhead. We probe the CPU usage every 10 ms while the controllers are active in our experiments and report the average value over the migration interval. With no migration, the initial master uses 0.72% of CPU (in median) for responding to the switch queries. As the number of migration candidates increases from one to three, CPU overhead on the initial controller increases to 1.2%, 1.81%, and 2.59%. Furthermore, in each run, the new master is likely to be the destination candidate that has the lowest transient load among the candidates at the time of migration. This leads to a slight decrease in the new master's median load from 2.26% to 2.16% as the number of candidates increases from 1 to 3.

Table I also shows the network overhead caused by the migration. In this experiment, only one switch is active and connected to `co-1`, and all other switches are deactivated. The normal operation of the initial master generates 645 KB of sent and 554 KB of received network traffic when no migration is taking place. Every added migration candidate adds around 5MB of sent network traffic due to the state transfer. The winner of the competition (new master) and the other migration candidates (non-winner) each receive the state, which adds roughly 5MB of received data to their network traffic. The new master receives and processes all the requests generated throughout and after the migration, which adds a few hundred KB to its network traffic compared to non-winners.

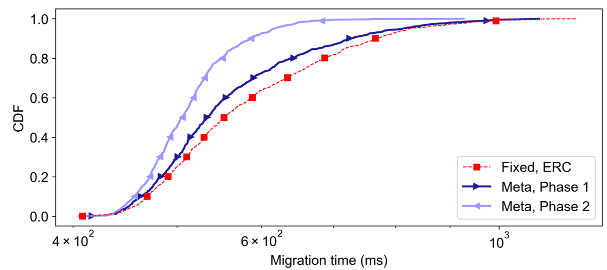
Overall, we conclude that determining the best number of candidates depends on a multitude of factors, such as resource availability, the current status of migration candidates, and the trade-off that the network administrators are willing to make between the average-case and worst-case outcome.

D. Effect of the committing point

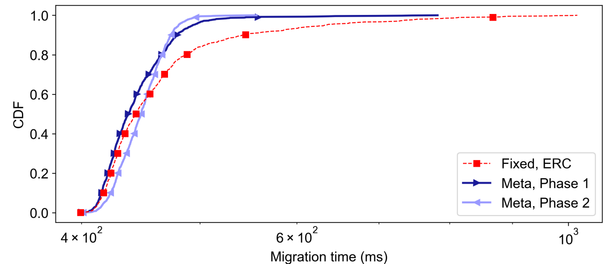
Finally, we study how the choice of the committing point affects the performance of Meta-Migration. As described in Section III-B, the end of Phase 1 and Phase 2 in the ERC protocol can be chosen as committing points. Choosing Phase 1 completion reduces the network and processing overhead since the state transfer will not happen for all candidates.

TABLE I
THE NETWORK TRAFFIC GENERATED ON DIFFERENT CONTROLLERS THROUGH THE COURSE OF MIGRATION. IN EACH CELL, THE UPPER AND LOWER NUMBERS SHOW THE SENT AND RECEIVED TRAFFIC, RESPECTIVELY. REPORTED NUMBERS ARE MEDIANS ACROSS ALL RUNS.

Destination Count	Not Active	Initial Master	New Master	Non Winner
No Mig.	6 KB 17 KB	645KB 554KB	-	-
1	5 KB 16 KB	5.5 MB 260 KB	440 KB 5.6 MB	-
2	5 KB 16 KB	10.8 MB 287 KB	446 KB 5.6 MB	49 KB 5.3 MB
3	-	16.0 MB 314 KB	445 KB 5.6 MB	49 KB 5.3 MB



(a) 75% load



(b) 50% load

Fig. 10. Effect of the committing point on the migration time distribution. While earlier committing reduces resource overhead, it does a poorer job of avoiding the heavy tails in the distribution.

However, state transfer is the longest stage in the protocol and is more prone to transient network problems and system delays. Fig. 10 shows the effect of the committing point with three candidates with a similar load. With 75% load (Figure 10(a)), committing after Phase 1 and 2 reduces the 99th percentile by 2% and 32%, respectively, while at the median, the reductions are 4% and 8%. The difference shows the causes of extreme tail latency usually lie in the state transfer phase and committing after Phase 1 does not help with that. With 50% load on the controllers, the trade-off between the two choices is more visible. While a Phase 2 committing can reduce the 99th percentile more than Phase 1, it leads to longer execution times in 80% of the runs.

V. Discussion and Conclusion

In what follows we discuss different paths that could be taken to further improve and expand the Meta-Migration idea which we leave for future works. We then conclude by summarizing the main contributions of this work.

Overhead Minimization. Two main overheads that Meta-Migration protocols are faced with are the network overhead and the CPU utilization at the source of the migration. To improve the latter, similar to the ideas mentioned in [13], part of the synchronization functionalities of the protocol could be pushed to the switches. For the network side, as we observed in our experiments, most of the extra traffic due to the addition of new candidates comes from state synchronization. One way to move the burden from the source node, which itself could be an already overwhelmed end-host, is to use unreliable multicast protocols to send the states toward all the candidates by using programmable switches [20]. As a result, the source

needs to do less work in terms of the state transfer which in turn contributes positively to the CPU utilization as well.

Other Use Cases. As shown in our experiments, the committing point in Meta-Migration protocols has a huge impact on its performance. Having the commit point later in the protocol results in a smoother estimation of the best candidate, but this means that possibly more data needs to be duplicated to run all the migrations in parallel. In the context of switch migration in which the transferred state is still relatively small, we can afford to commit at a later point to save more time in general. The question to be answered is which other contexts have similar requirements and limitations as the switch migration problem. Identifying these use cases can result in effortless immediate improvements in those areas as well.

Conclusion. In this work, we introduced the notion of Meta-Migration protocols by which any low-overhead migration protocol can be transformed to a faster version of itself at the cost of employing more destinations, and therefore sacrificing more resources. We examined this idea in the context of switch migration protocols. Experimental results support our theoretical analysis regarding cutting the tail latency of the migration. In this way, we can bypass network or system-related issues that can slow down the migration process.

Acknowledgements. We would like to thank anonymous reviewers for their helpful comments. We are especially grateful to Soheil Abbasloo, Mahmoud Bahnasy, and Ali Munir for their insightful comments and feedback on this work.

References

[1] S. Abbasi Zadeh, F. Zandi, M. A. Beiruti, and Y. Ganjali, "Load migration in distributed softwarized network controllers," *International Journal of Network Management*, vol. 32, no. 6, p. e2214, 2022.

[2] S. Abbasloo, Y. Xu, and H. J. Chao, "To schedule or not to schedule: When no-scheduling can beat the best-known flow scheduling algorithm in datacenter networks," *Computer Networks*, vol. 172, p. 107177, 2020.

[3] A. H. A. Abyaneh, M. Liao, and S. M. Zahedi, "Malcolm: Multi-agent learning for cooperative load management at rack scale," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 3, pp. 1–25, 2022.

[4] A. Agarwal, V. Arun, D. Ray, R. Martins, and S. Seshan, "Automating network heuristic design and analysis," in *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, 2022, pp. 8–16.

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, p. 63–74, 2010.

[6] M. A. Beiruti and Y. Ganjali, "Load migration in distributed sdn controllers," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–9.

[7] 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*, 2010, pp. 267–280.

[8] M. Buckley, S. Abbasi-Zadeh, M. A. Beiruti, S. Abbasloo, and Y. Ganjali, "Switch migration scheduling in distributed sdn controllers," in *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*. IEEE, 2022, pp. 348–356.

[9] J. Cui, Q. Lu, H. Zhong, M. Tian, and L. Liu, "A load-balancing mechanism for distributed sdn control plane using response time," *IEEE transactions on network and service management*, vol. 15, no. 4, 2018.

[10] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. R. Kompella, "Elasticon; an elastic distributed sdn controller," in *2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2014, pp. 17–27.

[11] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: A scalable and flexible data center network," *ACM SIGCOMM*, vol. 39, no. 4, p. 51–62, 2009.

[12] T. Hu, J. Lan, J. Zhang, and W. Zhao, "Easm: Efficiency-aware switch migration for balancing controller loads in software-defined networking," *Peer-to-Peer networking and applications*, vol. 12, no. 2, 2019.

[13] I. Kettaneh, A. Alquraan, H. Takruri, A. J. Mashtizadeh, and S. Al-Kiswany, "Accelerating reads with in-network consistency-aware load balancing," *IEEE/ACM Transactions on Networking*, 2021.

[14] 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.

[15] 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.

[16] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

[17] H. K. Rath, V. Revoori, S. M. Nadaf, and A. Simha, "Optimal controller placement in software defined networks (sdn) using a non-zero-sum game," in *Proc. IEEE Int. Symp. World Wireless, Mobile Multimedia Netw.*, 2014, pp. 1–6.

[18] K. S. Sahoo, D. Puthal, M. Tiwary, M. Usman, B. Sahoo, Z. Wen, B. P. Sahoo, and R. Ranjan, "Esmlb: Efficient switch migration-based load balancing for multicontroller sdn in iot," *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 5852–5860, 2019.

[19] E. Sakic, F. Sardis, J. W. Guck, and W. Kellerer, "Towards adaptive state consistency in distributed sdn control plane," in *2017 IEEE International Conference on Communications (ICC)*. IEEE, 2017, pp. 1–7.

[20] M. Shahbaz, L. Suresh, J. Rexford, N. Feamster, O. Rottenstreich, and M. Hira, "Elmo: Source routed multicast for public clouds," in *Proceedings of the ACM SIGCOMM*, 2019.

[21] L. Suresh, J. Loff, F. Kalim, S. A. Jyothi, N. Narodytska, L. Ryzhyk, S. Gamage, B. Oki, P. Jain, and M. Gasch, "Building scalable and flexible cluster managers using declarative programming," in *14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[22] C. Wang, B. Hu, S. Chen, D. Li, and B. Liu, "A switch migration-based decision-making scheme for balancing load in sdn," *IEEE Access*, vol. 5, pp. 4537–4544, 2017.

[23] 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.

[24] K. Winstein and H. Balakrishnan, "Tcp ex machina: Computer-generated congestion control," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 123–134, 2013.

[25] L. Yang, S. J. Park, M. Alizadeh, S. Kannan, and D. Tse, "Dispersedledger: High-throughput byzantine consensus on variable bandwidth networks," *arXiv preprint arXiv:2110.04371*, 2021.

[26] H. Zhong, J. Xu, J. Cui, X. Sun, C. Gu, and L. Liu, "Prediction-based dual-weight switch migration scheme for sdn load balancing," *Computer Networks*, vol. 205, p. 108749, 2022.

Appendix

A. Proof of Lemma 1

Observe that from (1) we have

$$\Pr(X_i > x) = \left(\frac{k_i}{x}\right)^{\alpha_i} \qquad \forall\ 0 \le i < n.$$

Then, given that $X = \min_{i=0}^{n-1}\{X_i\}$ and the X_is are independent gives

$$\begin{aligned} F_X(x) = \Pr(X \le x) &= 1 - \Pr(X > x) \\ &= 1 - \prod_{i=0}^{n-1} \Pr(X_i > x) \\ &= 1 - \prod_{i=0}^{n-1} \left(\frac{k_i}{x}\right)^{\alpha_i} \end{aligned}$$

□