

Learning the Optimal Synchronization Rates in Distributed SDN Control Architectures

Konstantinos Poularakis¹, Qiaofeng Qin¹, Liang Ma², Sastry Kompella³, Kin K. Leung⁴, and Leandros Tassiulas¹

¹Department of Electrical Engineering and Institute for Network Science, Yale University, USA

²IBM T. J. Watson Research Center, Yorktown Heights, NY, USA

³U.S. Naval Research Laboratory, Washington, DC, USA

⁴Department of Electrical and Electronic Engineering, Imperial College London, UK

Abstract—To support advanced coalition operations like secure information exchange between teams of soldiers that belong to different command centers in a tactical network, the control plane of the Software Defined Network (SDN) should be distributed. However, having multiple controllers managing the network brings additional challenges. One such challenge is how to coordinate the management decisions made by the controllers which is usually achieved by disseminating synchronization messages in a peer-to-peer manner. While there exist many architectures and protocols to ensure synchronized network views and drive coordination among controllers, these protocols typically induce significant traffic overheads, as we show through emulation measurements on OpenDaylight and ONOS controllers. These overheads hinder the application of SDN in wireless resource constrained environments like the army’s coalition tactical ad hoc networks. To address this issue, in this paper, we introduce the SDN synchronization problem: how often to synchronize the network views for each controller pair. Our objective is the maximization of the performance of applications of interest which may be affected by the synchronization rate. Using techniques from learning theory, we derive an algorithm with provable performance guarantees. Evaluation results demonstrate significant benefits over baseline schemes that synchronize all controller pairs at an equal rate.

I. INTRODUCTION

A. Motivation

Software Defined Networking (SDN) is a rapidly emerging technology that brings new flexibility to network management and therefore facilitates the implementation of advanced traffic engineering mechanisms [1]. The main principle of SDN is to shift all the network control functions from the data forwarding devices to a programmable network entity, the *controller*. To ensure availability in case of controller failure and network fragmentation, which are common in coalition tactical edge network scenarios, typical SDN systems deploy multiple controllers. The controllers may be physically distributed across the network, but they should be “logically-centralized”. This means that the controllers should coordinate their decisions to ensure their collective behavior matches the behavior of a single controller.

The coordination among controllers is an active area of research with several protocols proposed thus far [2]. For

This publication was supported by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001.

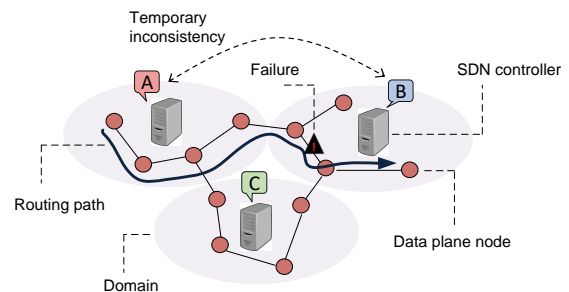


Fig. 1: Impact of inconsistency among controllers on routing application performance.

example, OpenDaylight [3] and ONOS [4], two state-of-the-art controller implementations, rely on RAFT and Anti-entropy protocols for disseminating coordination messages among controllers. Typically, each controller is responsible for a part of the network only, commonly referred to as the controller’s *domain*. The messages disseminated by a controller to the other controllers convey its view on the state of its domain (e.g., available links and installed flows). The composition of these messages allow the controllers to synchronize and agree on the state of the entire network.

While different coordination protocols may generate messages of different types and at different timescales, there exist two broad protocol categories [5]. The first category contains the *strongly consistent* protocols which strive to maintain all the controllers synchronized in all times. This is ensured by disseminating messages each time a network change (e.g., a node or link failure) happens followed by a consensus procedure. The second category contains the *eventually consistent* protocols which omit the consensus procedure, yet converge to a common state in a timely manner usually through periodic message dissemination.

Despite its benefits, strong consistency is difficult to ensure in practice as it is challenged by the unreliable nature of network communications. In addition, this approach generates significant *overheads* for message dissemination among controllers which may be prohibitively large especially when applied to wireless networks with in-band control channels of limited capacity [6], [7], [8]. On the other hand, eventual

consistency, where controllers are permitted to temporarily have inconsistent views of each other’s state, better suits the needs of the above networks, and, thus, can be used to extend the applicability of distributed controller solutions. Yet, the inconsistent views of controller states can harm the *performance of network applications*.

To illustrate the impact of inconsistency, we consider the toy example with three controllers (A, B and C) and their respective domains in Figure 1. Each pair of controllers synchronize periodically, e.g., every few seconds. At some time, controller A receives a request for routing a flow to a destination node inside the domain of B. Controller A will respond by computing and setting up a routing path based on its current view on the state (topology, traffic loads) of its domain and the other domains. However, controller A is not aware if the links on the routing path outside of its domain are still available or have failed (e.g., a failed link in domain B in Figure 1) since the last synchronization period. If failures happened, the packets of the flow will have to wait until the next synchronization period, although there is an alternative directly available routing path through the domain of C. Similar problems, if not more serious, can be identified for more advanced traffic engineering applications where inconsistency hinders the effective load balancing and distribution across multiple paths.

The eventually consistent model raises new technical challenges. In particular, it is important to decide *how often* (at what period or rate) to synchronize each pair of controllers in a given network. One might expect that the straightforward policy where all controller pairs synchronize at the same rate would work well. However, some may argue that the synchronization rate should be higher for domains that are more dynamic (with many changes in topology and flow configurations) in order to preserve consistency of the rest domains.

The issue is further complicated by the requirements of the network applications. Previous works [9], [10] showed that certain network applications, like load-balancers, can work around eventual consistency and still deliver acceptable (although degraded) performance. In such cases, some additional effort needs to be made to ensure that conflicts such as forwarding loops, black holes and reachability violation are avoided [11]. Therefore, synchronization policies that completely neglect the specific applications of interest in the network as well as the impact of synchronization rate on their performance may end-up being highly inefficient.

The above questions remain open since, until now, the inter-controller traffic has been often neglected in SDN literature with most of the existing works focusing on the routing and balancing of the data traffic (e.g., see the survey in [1] and the discussion of related work in Section IV).

B. Methodology and Contributions

Our goal in this paper is to investigate policies for the synchronization among SDN controllers, and focus particularly on the impact of the rate of synchronization on the performance

of network applications. This is a complex problem since, in practice, we do not know the function that maps the synchronization rate to application performance. To obtain some quantitative insights on this function, we emulate the performance of two applications of interest, namely shortest path routing and load balancing, using a commercial platform (Mininet) [12] and SDN controller (RYU) [13]. While the results are quite unsteady, the average performance increases with the synchronization rate and saturates eventually showing that a diminishing return rule applies. To overcome the unknown objective challenge, we use elements from the *learning theory*, and propose an algorithm that gradually trains the system and constructs a solution that is with high confidence close to the optimal. The contributions of this work can be summarized as follows:

- 1) We introduce the problem of finding the optimal synchronization rates among SDN controllers in a network. To the best of our knowledge, this is the first work that studies this problem.
- 2) We emulate the performance of two popular network applications and obtain insights about the impact of synchronization rates on application performance. We use these results to derive an algorithm that gradually trains the system in order to learn the optimal policy.
- 3) We perform evaluations to show the efficiency of our proposed algorithms. We find that significant benefits are realized compared with the baseline policy that synchronizes all controller pairs at equal rate.

The rest of the paper is organized as follows. In Section II, we present our emulation results and our learning algorithm for maximizing the network application performance. Section III presents the evaluation of our proposed algorithms, while Section IV reviews our contribution compared to related works. We conclude our work in Section V.

II. MAXIMIZING APPLICATION PERFORMANCE

It is known that certain applications can tolerate some inconsistency among controllers provided that conflicts are avoided, while other applications have stricter requirements [9], [10]. The above motivate us to look for synchronization methods that optimize directly the performance of specific applications of interest. In this section, we describe such a method by leveraging elements from the learning theory. Before that, we provide a brief emulation study that will highlight the impact of synchronization rate on the performance of some popular network applications.

A. Emulation Study

Below, we describe the emulation setup that will be later used to test the performance of two network applications, namely shortest path routing and load balancing.

Emulation setup. We use *Mininet* [12] to emulate virtual networks with several nodes and SDN controllers running on the same CPU machine. Among the set of commercial controllers that are available online we pick *RYU* [13] which is open-source and allows us to develop our own protocols

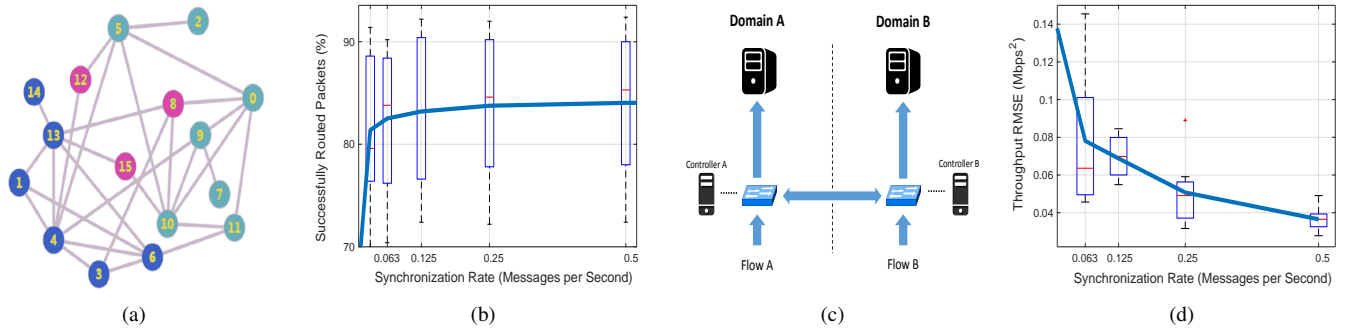


Fig. 2: Emulation results. Topology and impact of synchronization rate on the performance (box plots and average values) of (a)(b) shortest path routing and (c)(d) load balancing applications.

for the synchronization among controllers. Specifically, we implement a simple eventually-consistent protocol which periodically disseminates synchronization messages between each controller pair. Our code is parameterized to allow for any synchronization period. The disseminated messages convey the local views of controllers about the topology and installed flow tables. This information is made available to the controllers by the OpenFlow protocol.

Emulation results. We first test the performance of a shortest path routing application. With this application, packets are routed to their destination following the path of minimum hop count, calculated by Dijkstra’s algorithm. We generate the random network of 16 nodes and 3 controllers, depicted in Figure 2a, where links fail or recover randomly and independently every one second with probability 0.05, and nodes with the same color are managed by the same controller. We further generate data packets with random source-destination nodes. Unless the controllers synchronize at the time of packet generation, the packet is at risk of following a failed routing path.

The performance of routing application is determined by the number of packets that are successfully routed (without traversing any failed link) to their destinations. We emulate the performance for five different scenarios where all the controller pairs synchronize at the same rate equal to (i) 0.5, (ii) 0.25, (iii) 0.125, (iv) 0.063 and (v) 0.031 (messages per second). This translates to a single message disseminated every 2, 4, 8, 16 or 32 seconds. For each scenario, emulations are run for multiple times and the results are depicted in Figure 2b. Despite a large extent of randomness, we observe that the average performance (calculated over 20 minutes) increases with the synchronization rate and saturates eventually showing that a *diminishing return rule* applies.

We perform additional emulations to test the performance of a load balancing application. We consider a similar setup with the work in [9], depicted in Figure 2c. That is, we generate a network with two controllers. Each controller manages two nodes, a switch and a server. The switches generate flows uniformly at random. The flows can be routed and queued to any of the two servers. Each controller is aware of the load of

the server it manages. It also receives periodic synchronization messages about the load of the other server by the other controller. Each time a new flow is generated, the responsible controller routes it to the server with the currently observed lowest load. However, this may not be the least loaded server in reality, since the controllers are not synchronized at all times.

The emulation results are depicted in Figure 2d. The metric we consider is the root-mean-square deviation (RMSE) of two servers’ throughputs. The better the two server loads balance, the lower the value of this metric becomes. Therefore, this metric captures the performance of a load balancing application. For convenience, we claim it the cost function, and denote the performance metric the opposite value of cost function. Then, coinciding with the routing application, we observe that the performance improves with the synchronization rate but gradually saturates showing that a *diminishing return rule* applies.

B. Learning Framework

Subsequently, we study the objective of maximizing the performance of a network application such as the applications emulated in the previous subsection. While the objective function is expected to have a curve shape similar to those reported in Figure 2, we cannot express in closed-form how exactly the synchronization rates will affect application performance. Therefore, the objective function is *unknown*.

To overcome the unknown objective challenge, we propose to leverage methods from the *learning theory*. Such methods typically *train* the system by trying-out a sequence of solutions (synchronization rates) over some training period $\mathcal{T} = \{1, 2, \dots, T\}$ of T time slots, until they can infer a “sufficiently good” solution.

We adopt a general model representing an eventually-consistent SDN system with a set \mathcal{C} of C controllers distributed in a network, as shown in Figure 1. Each controller is responsible for managing a subset of the data plane nodes in the network, referred to as a domain. The controllers are aware of the current state information inside their domains (e.g., available links, flow table entries). This can be achieved

by using a SouthBound protocol (e.g., OpenFlow) for signaling and statistic collection from the data plane nodes.

We introduce the vector of synchronization rate variables:

$$\mathbf{x} = (x_{ij}^t \in \{0, 1, \dots, R\} : \forall i, j \in \mathcal{C}, j \neq i, t \in \mathcal{T}) \quad (1)$$

where x_{ij}^t indicates the synchronization rate between controllers i and j tried-out in time slot t . R denotes the maximum rate. We further denote by the vector \mathbf{x}^t all the variables in time slot t . We emphasize that the variable values will be typically different from slot to slot as different synchronization rates need to be explored in order to train the system.

Given the synchronization rate vector \mathbf{x}^t tried-out in a slot t , the application performance will be $\Psi_t(\mathbf{x}^t)$. Here, $\Psi_t(\cdot)$ is an unknown function that governs the application performance in slot t . While the overall function is unknown, the single value $\Psi_t(\mathbf{x}^t)$ can be *observed* by the system operator *after* the synchronization rate decision \mathbf{x}^t is made, in the end of the slot. For a shortest path routing application, for example, this is possible by measuring the number of data packets that reached their destination in time. Such information can be estimated by the controllers using the TCP acknowledgement packets. The information can be then passed to the system operator (e.g., one of the controllers) which can simply aggregate and sum the respective values.

We emphasize that the function $\Psi_t(\cdot)$ is time slot-dependent, meaning that the performance value might change with time even for the same synchronization rate decision. That is, we may try-out the same synchronization rate vector $\mathbf{x}^t = \mathbf{x}^{t'}$ in two slots t and t' but observe different performance values $\Psi_t(\mathbf{x}^t) \neq \Psi_{t'}(\mathbf{x}^{t'})$. Such *uncertainty of observations* is due to the stochastic nature of the network. Intuitively, the performance value will be large if the network happens to be stable in a slot but will be much worse in other slots during which many changes happen.

Despite the uncertainty of observations, the learning method should be able to infer by the end of the training period \mathcal{T} a “sufficiently good” synchronization rate decision $\hat{\mathbf{x}} = (\hat{x}_{ij} : i, j \in \mathcal{C}, j \neq i)$. This should, ideally, maximize the *average* performance denoted by an (also unknown) function $\hat{\Psi}(\cdot) = \mathbb{E}[\Psi_t(\cdot)]$. While the system operator does not know the average performance values, we assume that they do not change over a period of time (e.g., a few hours). Therefore, our second objective can be written as:

$$\text{Obj. 2 : } \max_{\hat{\mathbf{x}}} \hat{\Psi}(\hat{\mathbf{x}}) \quad (2)$$

$$\text{s.t. } \sum_{i \in \mathcal{C}} \sum_{j \in \mathcal{C}, j \neq i} \hat{x}_{ij} b_{ij} \leq B \quad (3)$$

where inequality (3) ensures that the inferred synchronization rate decision will satisfy the resource constraint.

We need to emphasize that the average performance $\hat{\Psi}(\hat{\mathbf{x}})$ can be in fact the aggregate of many (rather than only one) applications. Either way, the performance is not the only criterion that determines the efficiency of a learning method. Another important criterion in this context is the *running (or training) time* T , i.e., how many time slots are required for

training in order to infer the synchronization rate decision $\hat{\mathbf{x}}$. In the next subsection, we will propose a learning method that has adjustable average performance and running time.

C. Learning Algorithm

To handle the uncertainty of an observed performance value $\Psi_t(\mathbf{x}^t)$, a learning method would typically try-out the *same* synchronization decision \mathbf{x}^t *multiple times*, in different time slots. Then, the empirical mean of the observations will be used to estimate the average performance value $\hat{\Psi}(\mathbf{x}^t)$. By repeating the above training process for every possible synchronization decision, an estimate of the entire objective function $\hat{\Psi}(\cdot)$ can be obtained. However, there exists an *exponential number of possible decisions*; $(R + 1)^{C(C-1)}$ decisions in total. Therefore, this approach would require an exponential number of time slots for training, which is clearly not practical.

To overcome the high dimensionality of the synchronization decision space, we could leverage learning methods proposed recently that do not require the estimation of the objective function at every possible decision. For instance, the *Exp-Greedy algorithm* proposed in [14] can infer a close-to-optimal decision in *polynomial-time* provided that the objective function follows a diminishing return rule, as the one observed in the emulation results in Figure 2. Still, however, the running time of this algorithm may be too large for our problem, as we will show numerically in the next section, hindering its application in practical scenarios.

Based on the above, we propose an alternative more-practical learning algorithm for which we can flexibly adjust the running time by setting appropriate values to its input parameters. We refer to this algorithm as *Stochastic Greedy* and summarize it in Algorithm 1. To ease presentation, we have assumed that the resource costs are equal and normalized to one for all the controller pairs, i.e., $b_{ij} = 1 \forall i, j$. However, the algorithm and analysis can be easily extended for heterogeneous resource costs.

In a nutshell, the Stochastic Greedy algorithm starts with the all-zero synchronization decision and then gradually constructs the decision to be returned by iteratively increasing by 1 the synchronization rate of a single controller pair. This procedure will end when the B resource constraint is reached, i.e., after B iterations. Each iteration requires multiple time slots for training so as to be confident that the controller pair selected to increase its rate by 1 will improve the average performance more than other controller pairs. The length of the training period can be adjusted by two input parameters σ and τ . The value of σ is between 1 and $C(C - 1)$, while τ can take any positive integer value.

Formally, the algorithm maintains a synchronization rate decision $\hat{\mathbf{x}}$, initially set to the zero vector $\mathbf{0}$ (line 1). It spends the first τ time slots trying out the zero synchronization decision and uses the τ observations to estimate $\hat{\Psi}(\mathbf{0})$ (lines 2-3). In the next B iterations (lines 4-11), the algorithm will iteratively select a controller pair and increase the respective synchronization rate by 1, updating $\hat{\mathbf{x}}$. At each iteration

Algorithm 1: Stochastic Greedy with (σ, τ) input

```
1 Initialize  $\hat{\mathbf{x}} = \mathbf{0}$ ;  
2 Try out  $\mathbf{x}^t = \hat{\mathbf{x}}$  and observe  $\Psi_t(\mathbf{x}^t) \forall t \in \{1, \dots, \tau\}$ ;  
3 Estimate  $\hat{\Psi}(\hat{\mathbf{x}}) = \frac{1}{\tau} \sum_{t=1}^{\tau} \Psi_t(\mathbf{x}^t)$ ;  
4 for each iteration  $k$  from 1 to  $B$  do  
5   Pick  $\sigma$  random controller pairs  $p$  for which  
    $\hat{x}_p < R$ ;  
6   for each picked pair  $p$  from 1 to  $\sigma$  do  
7     Set  $\hat{\mathbf{x}}' = \hat{\mathbf{x}}$  where  $\hat{x}'_p = \hat{x}_p + 1$ ;  
8     Try out  $\mathbf{x}^t = \hat{\mathbf{x}}'$  and observe  $\Psi_t(\mathbf{x}^t) \forall t \in$   
      $\{(k-1)\sigma\tau + p\tau + 1, \dots, (k-1)\sigma\tau + p\tau + \tau\}$ ;  
9     Estimate  $\hat{\Psi}(\hat{\mathbf{x}}') = \frac{1}{\tau} \sum_{t=(k-1)\sigma\tau + p\tau + 1}^{(k-1)\sigma\tau + p\tau + \tau} \Psi_t(\mathbf{x}^t)$ ;  
10    Set  $D(\hat{\mathbf{x}}, \hat{\mathbf{x}}') = \hat{\Psi}(\hat{\mathbf{x}}') - \hat{\Psi}(\hat{\mathbf{x}})$ ;  
    end  
11   Update  $\hat{\mathbf{x}} = \operatorname{argmax}_{\hat{\mathbf{x}}'} D(\hat{\mathbf{x}}, \hat{\mathbf{x}}')$ ;  
    end  
12 Output:  $\hat{\mathbf{x}}$ ;
```

$k = 1, 2, \dots, B$, the algorithm will initially pick σ random pairs of controllers as candidates (line 5). For each such pair $p = 1, 2, \dots, \sigma$, the synchronization decision $\hat{\mathbf{x}}'$ will be set accordingly (line 7) and τ time slots will be spent to estimate $\hat{\Psi}(\hat{\mathbf{x}}')$ (lines 8-9). The marginal performance gain of switching from decision $\hat{\mathbf{x}}$ to $\hat{\mathbf{x}}'$, denoted by $D(\hat{\mathbf{x}}, \hat{\mathbf{x}}')$, will be estimated (line 10). Among the σ candidate controller pairs, the algorithm will include in the current decision $\hat{\mathbf{x}}$ the pair with the maximum estimated marginal performance gain (line 11).

The algorithm spends τ time slots to estimate $\hat{\Psi}(\hat{\mathbf{x}})$ for $\hat{\mathbf{x}} = \mathbf{0}$, and $\sigma\tau$ more slots for each iteration. Therefore, the total running (or training) time is $T = \tau + \sigma\tau B$ time slots. The following theorem describes the average performance of the algorithm. Since the algorithm makes random decisions, the average performance bound holds in expectation.

Theorem 1. *Algorithm 1 achieves average performance $\hat{\Psi}(\hat{\mathbf{x}})$ that is in expectation a factor $1 - e^{-(1-\epsilon)\mu}$ from the optimal where $\epsilon = e^{-\sigma \frac{B}{\sigma(\sigma-1)R}}$ and μ is the expected fraction of the observed marginal gain in a slot over the actual marginal gain.*

To facilitate reading, we defer the proof of the theorem to the appendix. We emphasize that the average performance bound depends on the value of μ . This value captures the uncertainty of the observed performance values since the changes in network state may be unevenly distributed across the time slots. If $\mu = 1$, it means that the performance value does not depend on the time slot of observation and hence the estimated maximum performance will be the actual one. However, as the μ value goes to 0 the observations become more uncertain.

Another issue is that the performance bound in Theorem 1 holds in expectation, which means that it may be violated in practice. Therefore, it is important to bound the extent to

which this happen, as we show in the following theorem.

Theorem 2. *Algorithm 1 achieves average performance $\hat{\Psi}(\hat{\mathbf{x}})$ that is a factor $1 - e^{-(1-\epsilon)(1-\gamma)\mu}$ from the optimal with probability $1 - e^{-\frac{\gamma B \tau}{2}}$ for any $\gamma \in (0, 1)$.*

The average performance bounds of our algorithm can be better understood through an example. In particular, consider the system with $C = 5$ controllers, $B = 10$ available resources and $s = 30$ seconds per time slot. By picking $\sigma = 5$ out of the 20 possible controller pairs and $\tau = 3$ time slots per try-out, the total running (training) time of the algorithm will be about one hour. Moreover, if the observed marginal performance gains are 50% or more of the actual ones ($\mu = 0.5$) and $R = 1$, the average performance achieved by the algorithm will be in expectation at least 37% of the optimal. Picking a larger σ value will increase the average performance (cf. Theorem 1). Picking a larger τ value will increase the probability that the performance bound is not violated (cf. Theorem 2).

III. EVALUATION

In this section, we carry out evaluations to show the benefits of the proposed algorithm. We find that our Stochastic Greedy algorithm achieves a more favorable tradeoff between performance and training time than several state-of-the-art learning algorithms.

A. Evaluation setup

In order to show that our proposed algorithm is the most suitable for determining the SDN controller synchronization rates, we investigate and compare a number of state-of-the-art algorithms from different families of learning methods that are capable of optimizing an unknown function in an online manner. Specifically, we implement the following three state-of-the-art learning algorithms:

- 1) **Online Convex Optimization (OCO).** This type of algorithms [20] generalize the basic gradient descent algorithm used for standard offline optimization to explore an unknown function. In our synchronization problem, the form of the gradient function is also unknown. We therefore adopt the technique proposed in [21] to estimate the gradient.
- 2) **Reinforcement Learning with Deep Q-Network (DQN).** The system of multiple controllers can also be regarded as a state machine with unknown rewards when taking different actions, where reinforcement learning techniques can be applied. In particular, we define the state as a vector of synchronization rates between each pair of controllers. Initially, the synchronization rates are homogeneous, and a feasible action is to decrease the rate of any pair, and increase the rate of another pair by the same amount. We notice that the number of states increases exponentially with the problem scale. To solve it efficiently, we deploy a neural network based agent (Deep Q-Network [22]) for reinforcement learning.
- 3) **ExpGreedy.** We also consider another greedy algorithm [14], which proposes a different way to take

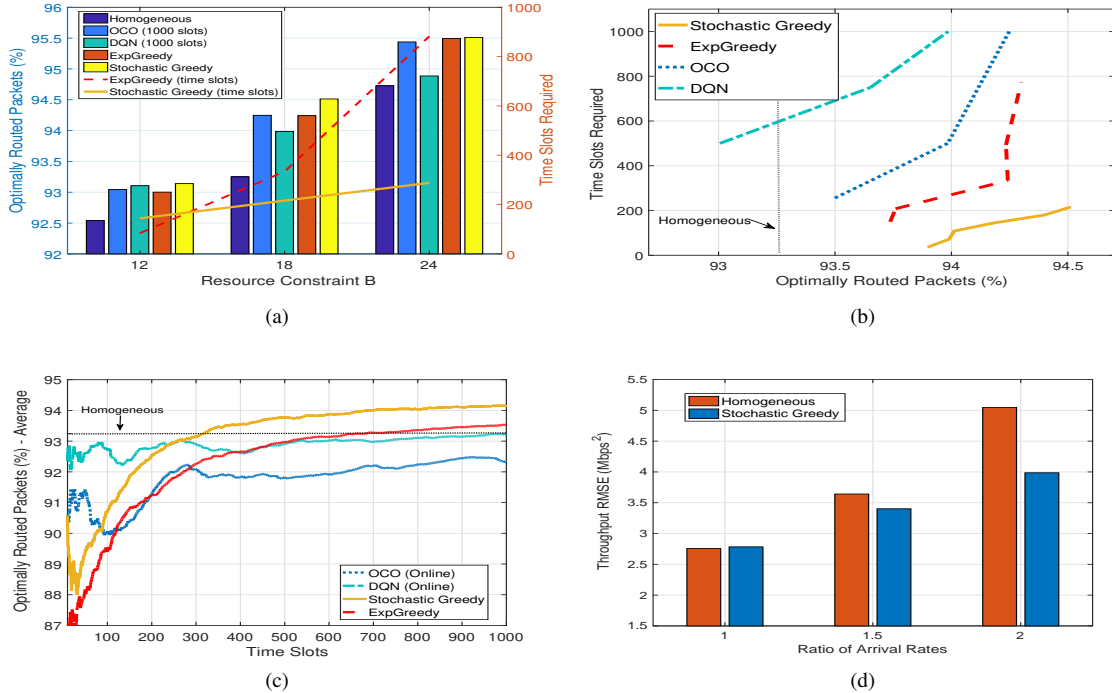


Fig. 3: (a) Performance and training time for different resource budgets, (b) tradeoff between performance and training time and (c) learning process under the shortest path routing application. (d) RMSE cost for different ratios of flow arrival rates under the load-balancing application.

samples and estimate the objective function to infer close-to-optimal synchronization decisions.

Besides of the above algorithms, we also consider the Homogeneous synchronization rates as a baseline of algorithm performance, i.e., allocating the resource budget B evenly to each pair of controllers. To eliminate the randomness, we run each algorithm 10 times and report the average value in the results. Throughout the evaluations, we choose the same network topologies and applications as in our emulations in Section II-A (16-node shortest path routing and 2-server load balancing).

B. Evaluation Results

We evaluate the shortest path routing and load balancing applications separately.

Shortest path routing application. We first consider the shortest path routing application in the 16-node network. A performance metric of interest for this application is the percentage of packets that are optimally routed to their destinations, i.e., following paths of the same number of hops as the optimal path. Figure 3a depicts the performance for different resource budgets B . We notice that *the proposed Stochastic-Greedy algorithm has the best performance among algorithms in all cases*. We also consider the required training time. With the same set of parameters, the training time required by our algorithm increases linearly with B . On the other hand, the training time of ExpGreedy increases more dramatically. Specifically, *our algorithm requires around 200 time slots*

(*about an hour and a half*) for training while ExpGreedy may consume more than 800 time slots (6-7 hours), which may be prohibitively large in practice. As for the two online learning algorithms OCO and DQN, they generally require even more training time to achieve a high performance. We run them for 1000 time slots and still observe some unsatisfying results (e.g., DQN at $B = 18$ behaves almost as bad as the Homogeneous policy).

To further explore the relationship between performance and training time, we adjust some input parameter values the algorithms take (σ and τ for our algorithm). Shorter training time will be at the cost of reduced performance (as we described in Theorems 1 and 2). Figure 3b depicts the detailed tradeoff between performance and training time for all the algorithms. It shows that we can flexibly trade the performance and training time of our algorithm (from 93.9% to 94.5% optimally routed packets and from 50 to 210 slots). To achieve the same performance level, ExpGreedy typically takes more than twice the time compared to our algorithm. As for OCO and DQN, they generally need more training time than greedy algorithms. In order to merely have better performance than the baseline, more than 400 time slots are required, limiting their applicability in practical scenarios.

We then examine the performance over time of the proposed algorithm and compare it with the two online learning algorithms, OCO and DQN. We track the objective function for 1000 time slots in the $B = 18$ case ($\sigma = 2$ and $\tau = 4$),

during which the Stochastic-Greedy algorithm is finished first and then the output policy is applied for the rest slots. In the cases of OCO and DQN, we keep them in the training stage for all slots. Figure 3c depicts the average value of the objective function during the learning process for the different algorithms. For OCO, additional randomness is introduced by the gradient estimation procedure, causing it to deviate from the optimal point frequently which leads to inferior performance. For DQN, the learning process is relatively slow. It can be observed that the average objective function keeps fluctuating in first 500 slots, during which the greedy algorithm has already finished.

Load balancing application. Next, we examine the load balancing application. Similar to the emulations in Section II-A, we randomly generate flows at two switches. We define one time slot as 60 seconds. Under the same B value, we compare the Stochastic Greedy and Homogeneous algorithms for various flow arrival rates. When the arrival rates at the two switches are equal, the Homogeneous algorithm should be optimal because of the symmetry. In this case, as Figure 3d shows, our algorithm gets almost the same RMSE cost. Next, we set different arrival rates at the two switches. As a result, when the ratio of arrival rates gets larger, our algorithm leads to lower cost than the Homogeneous algorithm. For example, our algorithm can decrease the RMSE by around 20% when the ratio of flow arrival rates at the two switches is equal to 2.

IV. RELATED WORK

Distributed SDN controller deployments require a coordination protocol among controllers, which could easily generate significant amount of control traffic, e.g., see the measurement studies in [6] and [7]. However, the control traffic is often neglected in literature with most of the existing works focusing on the routing and balancing of the data traffic, e.g., see [15] and the survey in [1].

Some recent works proposed to reduce the overheads of control traffic by strategically placing the controllers in the network [16] or by finding the appropriate forwarding paths for load balancing on control traffic [8]. Nevertheless, the above approaches should be considered as complementary to our work, rather than competitive. On the one hand, the controller placement decisions are taken in a different (much slower) timescale than the synchronization. On the other hand, the control traffic forwarding can be combined with the synchronization rate decisions we make, since the former directly impacts the resource cost values b_{ij} used as input to our problem.

Eventual consistency, where the controllers coordinate periodically rather than on demand basis, is another way to reduce control overheads. Levin et al. [9] showed that certain network applications, like load-balancers, can work around eventual consistency and still deliver acceptable performance. This would require some additional effort to be made to ensure that conflicts such as forwarding loops, black holes and reachability violation are avoided [11].

Few recent works suggested the dynamic adaptation of synchronization period (or rate) among controllers in an eventually-consistent system so as to improve the performance of network applications while maintaining a scalable system [17], [18]. While interesting and relevant, the above works did not provide any mathematical formulation or optimization framework. To the best of our knowledge our work is the first to systematically study the synchronization problem and propose optimization and learning methods.

V. CONCLUSION

In this paper, we studied the problem of finding the optimal synchronization rates among controllers in a distributed eventually-consistent SDN system. Our objective was the maximization of the performance of applications which may be affected by the synchronization decisions, as highlighted by emulations on a commercial SDN controller. We proposed a novel learning algorithm to achieve the optimal synchronization rates. Evaluation results demonstrated significant performance benefits over the baseline policy that synchronizes all controller pairs at an equal rate as well as state-of-the-art algorithms from different families of machine learning such as online convex optimization and deep reinforcement learning.

REFERENCES

- [1] D. Kreutz, F. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, S. Uhlig, "Software-Defined Networking: A Comprehensive Survey", *Proc. of the IEEE*, vol. 103, no. 1, pp. 14-76, 2015.
- [2] Y.E. Oktian, S. Lee, H. Lee, J. Lam, "Distributed SDN Controller System: A Survey on Design Choice", *Computer Networks*, vol. 121, no. 5, pp. 100-111, 2017.
- [3] <https://www.opendaylight.org>
- [4] <http://onosproject.org>
- [5] F. Botelho, T. A. Ribeiro, P. Ferreira, F. M. V. Ramos, A. Bessani, "Design and Implementation of a Consistent Data Store for a Distributed SDN Control Plane", *IEEE EDCC*, 2016.
- [6] A.S. Muqaddas, P. Giaccone, A. Bianco, G. Maier, "Inter-controller Traffic to Support Consistency in ONOS Clusters", *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 1018-1031, 2017.
- [7] Q. Qin, K. Poularakis, G. Iosifidis, L. Tassiulas, "SDN Controller Placement at the Edge: Optimizing Delay and Overheads", *IEEE Infocom*, 2018.
- [8] S.-C. Lin, P. Wang, I.F. Akyildiz, M. Luo, "Towards Optimal Network Planning for Software-Defined Networks", *IEEE Transactions on Mobile Computing*, 2018.
- [9] D. Levin, A. Wundsam, B. Heller, N. Handigol, A. Feldmann, "Logically Centralized? State Distribution Trade-offs in Software Defined Networks", *ACM HotSDN*, 2012.
- [10] A. Panda, W. Zheng, X. Hu, A. Krishnamurthy, S. Shenker, "SCL: Simplifying Distributed SDN Control Planes", *USENIX NSDI*, 2017.
- [11] Z. Guo, M. Su, Y. Xu, Z. Duan, L. Wang, S. Hui, H. J. Chao, "Improving the Performance of Load Balancing in Software-Defined Networks through Load Variance-based Synchronization", *Computer Networks*, vol. 68, pp. 95-109, 2014.
- [12] B. Lantz, B. Heller, N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-defined Networks", *ACM HotSDN*, 2010.
- [13] <http://osrg.github.io/ryu>
- [14] A. Singla, S. Tschitschek, A. Krause, "Noisy Submodular Maximization via Adaptive Sampling with Applications to Crowdsourced Image Collection Summarization", *AAAI*, 2016.
- [15] D. Tuncer, M. Charalambides, S. Clayman, and G. Pavlou, "Adaptive Resource Management and Control in Software Defined Networks", *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 18-33, 2015.

- [16] Z. Su, M. Hamdi, "MDCP: Measurement-Aware Distributed Controller Placement for Software Defined Networks", *IEEE ICPADS*, 2015.
- [17] M. Aslan, A. Matrawy, "Adaptive Consistency for Distributed SDN Controllers", *IEEE Networks* 2016.
- [18] E. Sakic, F. Sardis, J.W. Guck, W. Kellerer, "Towards Adaptive State Consistency in Distributed SDN Control Plane", *IEEE ICC*, 2017.
- [19] B. Mirzasoleiman, A. Badanidiyuru, A. Karbasi, J. Vondrak., A. Krause, "Lazier Than Lazy Greedy", *AAAI*, 2015.
- [20] E. Hazan, "Introduction to online convex optimization", *Foundations and Trends in Optimization*, pp. 157-325, 2016.
- [21] A. D. Flaxman, A. T. Kalai, A. T. Kalai, and H. B. McMahan, "Online Convex Optimization in the Bandit Setting: gradient descent without a gradient", *ACM-SIAM symposium on Discrete algorithms*, 2005.
- [22] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, and G. Ostrovski, "Human-level Control through Deep Reinforcement Learning", *Nature*, vol. 518, no. 7540, pp. 529, 2015.

APPENDIX PROOF OF THEOREM 1

To facilitate the presentation of the proof, we describe an alternative representation of the synchronization rate decisions using the following set of elements (ground set):

$$\mathcal{G} = \{g_{ij}^r : \forall i, j \in C, j \neq i, r \in \{1, \dots, R\}\} \quad (4)$$

Here, each of the elements $\{g_{ij}^1, g_{ij}^2, \dots, g_{ij}^R\}$ indicates a separate message disseminated between controllers i and j . Each subset of elements $\hat{\mathcal{X}} \subseteq \mathcal{G}$ indicates a synchronization policy \hat{x} where the synchronization rate \hat{x}_{ij} is equal to the number of the aforementioned elements included in $\hat{\mathcal{X}}$.

We denote by the subsets $\mathcal{A} \subseteq \mathcal{G}$ and $\mathcal{O} \subseteq \mathcal{G}$ the solution returned by the Stochastic Greedy approximation algorithm and the optimal, respectively. We also denote by the subset $\mathcal{A}_k = \{\alpha_1, \dots, \alpha_k\} \subseteq \mathcal{A}$ the solution returned by the Stochastic Greedy algorithm after the first $0 \leq k \leq B$ iterations. Then, similar to the proof in [19], we compute the probability that the set \mathcal{S} of σ elements that is randomly picked by Stochastic Greedy at iteration $k+1$ does not overlap with the optimal set \mathcal{O} besides of the elements already in \mathcal{A}_k :

$$\begin{aligned} \Pr[\mathcal{S} \cap (\mathcal{O} \setminus \mathcal{A}_k) = \emptyset] &= \left(1 - \frac{|\mathcal{O} \setminus \mathcal{A}_k|}{|\mathcal{G} \setminus \mathcal{A}_k|}\right)^\sigma \\ &\leq e^{-\sigma \frac{|\mathcal{O} \setminus \mathcal{A}_k|}{|\mathcal{G} \setminus \mathcal{A}_k|}} \leq e^{-\sigma \frac{|\mathcal{O} \setminus \mathcal{A}_k|}{C(C-1)R}} \end{aligned} \quad (5)$$

where the first inequality is because $(1-x)^a \leq e^{-ax}$ for any $x \in (0, 1)$. The second inequality is because $|\mathcal{G}| = C(C-1)R$. Then, we have:

$$\begin{aligned} \Pr[\mathcal{S} \cap (\mathcal{O} \setminus \mathcal{A}_k) \neq \emptyset] &\geq 1 - e^{-\sigma \frac{|\mathcal{O} \setminus \mathcal{A}_k|}{C(C-1)R}} \\ &\geq (1 - e^{-\sigma \frac{B}{C(C-1)R}}) \frac{|\mathcal{O} \setminus \mathcal{A}_k|}{B} = (1 - \epsilon) \frac{|\mathcal{O} \setminus \mathcal{A}_k|}{B} \end{aligned} \quad (6)$$

where the second inequality is because the function $1 - e^{-\sigma \frac{x}{C(C-1)R}}$ is concave with respect to $x \in [0, B]$. The last equality is because of the definition of ϵ .

At iteration $k+1$, Stochastic Greedy adds the element α_{k+1} to the solution \mathcal{A}_k which is estimated to maximize the marginal gain $\widehat{\Psi}(\mathcal{A}_{k+1}) - \widehat{\Psi}(\mathcal{A}_k)$. However, the element with the real maximum marginal gain may be different, namely

$\alpha'_{k+1} \neq \alpha_{k+1}$. Given that α_{k+1} is picked after τ try-outs, the following equation holds:

$$\begin{aligned} \widehat{\Psi}(\mathcal{A}_k \cup \{\alpha_{k+1}\}) - \widehat{\Psi}(\mathcal{A}_k) &= \\ \left(\sum_{t=1}^{\tau} \frac{\mu_{k+1}^t}{\tau}\right) \left(\widehat{\Psi}(\mathcal{A}_k \cup \{\alpha'_{k+1}\}) - \widehat{\Psi}(\mathcal{A}_k)\right) \end{aligned} \quad (7)$$

where μ_{k+1}^t is the ratio of marginal gains according to try-out $t = 1, 2, \dots, \tau$. Each μ_{k+1}^t value is taken from a distribution with mean value μ .

By definition, $\widehat{\Psi}(\mathcal{A}_k \cup \{\alpha'_{k+1}\}) - \widehat{\Psi}(\mathcal{A}_k)$ is at least as much as the marginal value of an element randomly chosen from the set $\mathcal{S} \cap (\mathcal{O} \setminus \mathcal{A}_k)$ (if non-empty). This is actually an element randomly chosen from the entire set $\mathcal{O} \setminus \mathcal{A}_k$, since the set \mathcal{S} itself is randomly chosen. Thus, we have:

$$\begin{aligned} \widehat{\Psi}(\mathcal{A}_k \cup \{\alpha'_{k+1}\}) - \widehat{\Psi}(\mathcal{A}_k) &\geq \Pr[\mathcal{S} \cap (\mathcal{O} \setminus \mathcal{A}_k) \neq \emptyset] \frac{\sum_{o \in \mathcal{O} \setminus \mathcal{A}_k} (\widehat{\Psi}(\mathcal{A}_k \cup \{o\}) - \widehat{\Psi}(\mathcal{A}_k))}{|\mathcal{O} \setminus \mathcal{A}_k|} \\ &\geq \frac{1 - \epsilon}{B} \sum_{o \in \mathcal{O} \setminus \mathcal{A}_k} (\widehat{\Psi}(\mathcal{A}_k \cup \{o\}) - \widehat{\Psi}(\mathcal{A}_k)) \\ &\geq \frac{1 - \epsilon}{B} (\widehat{\Psi}(\mathcal{O}) - \widehat{\Psi}(\mathcal{A}_k)) \end{aligned} \quad (8)$$

where the second inequality is because of (6). The third inequality is due to the rule of diminishing returns. By combining (7) and (8) we obtain:

$$\begin{aligned} \widehat{\Psi}(\mathcal{A}_{k+1}) - \widehat{\Psi}(\mathcal{A}_k) &= \widehat{\Psi}(\mathcal{A}_k \cup \{\alpha_{k+1}\}) - \widehat{\Psi}(\mathcal{A}_k) \\ &\geq \frac{(1 - \epsilon) \sum_{t=1}^{\tau} \mu_{k+1}^t}{B} (\widehat{\Psi}(\mathcal{O}) - \widehat{\Psi}(\mathcal{A}_k)) \end{aligned} \quad (9)$$

By induction, we can show that:

$$\begin{aligned} \widehat{\Psi}(\mathcal{A}_B) &\geq \left(1 - \left(1 - \frac{(1 - \epsilon) \sum_{k=1}^B \sum_{t=1}^{\tau} \mu_k^t}{B\tau}\right)^B\right) \widehat{\Psi}(\mathcal{O}) \\ &\geq (1 - e^{-(1-\epsilon) \frac{\sum_{k=1}^B \sum_{t=1}^{\tau} \mu_k^t}{B\tau}}) \widehat{\Psi}(\mathcal{O}) \end{aligned} \quad (10)$$

Since the μ_k^t values are drawn from a distribution with mean value μ , it will be $\frac{\sum_{k=1}^B \sum_{t=1}^{\tau} \mu_k^t}{B\tau} = \mu$ in expectation, which concludes the proof.

APPENDIX PROOF OF THEOREM 2

Let $\mu_1^1, \dots, \mu_B^\tau$ be the marginal gain ratios associated with the $B\tau$ try-outs of the Stochastic greedy algorithm. Since $\mu_k^t \in (0, 1)$, $\forall t, k$ with mean value μ , we can apply the Chernoff bound and obtain for each $\gamma \in (0, 1)$:

$$\Pr\left[\frac{1}{B\tau} \sum_{k=1}^B \sum_{t=1}^{\tau} \mu_k^t < (1 - \gamma)\mu\right] < e^{-\frac{\gamma\mu B\tau}{2}} \quad (11)$$

Therefore, with probability $1 - e^{-\frac{\gamma\mu B\tau}{2}}$, the empirical mean value will be at least as much as $(1 - \gamma)\mu$. With the same probability, the performance will be at least as much as $1 - e^{-(1-\epsilon)(1-\gamma)\mu}$.