Learning the Optimal Synchronization Rates in Distributed SDN Control Architectures

Konstantinos Poularakis[1], Qiaofeng Qin[1], Liang Ma[2], Sastry Kompella[3], Kin Leung[4], and Leandros Tassiulas[1]

[1]Department of Electrical Engineering and Institute for Network Science, Yale University, USA
[2]IBM T. J. Watson Research Center, USA
[3]Naval Research Laboratory, USA
[4]Department of Electrical and Electronic Engineering, Imperial College, UK

***Abstract*—Since the early development of Software-Defined Network (SDN) technology, researchers have been concerned with the idea of physical distribution of the control plane to address scalability and reliability challenges of centralized designs. However, having multiple controllers managing the network while maintaining a "logically-centralized" network view 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, there is no systematic methodology for deciding the optimal frequency (or rate) of message dissemination. In this paper, we fill this gap by introducing the SDN synchronization problem: how often to synchronize the network views for each controller pair. Our objective is to maximize the performance of applications of interest which may be affected by the synchronization rate. Using techniques from learning theory, we derive algorithms with provable performance guarantees. Evaluation results demonstrate significant benefits over baseline schemes that synchronize all controller pairs at 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, 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 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.

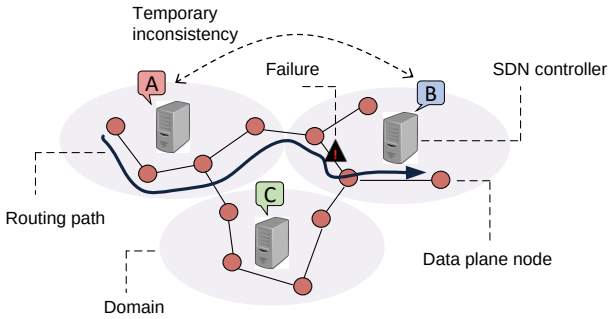


Fig. 1: Impact of inconsistency among controllers on routing application performance.

While different coordination protocols may generate messages of different types and at different timescales, there exist two broad protocol categories [5], [6]. 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 [7], [8], [9]. 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 [10], [11] 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 [12]. 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) [13] and SDN controller (RYU) [14]. While the results are quite unsteady, the average performance increases with the synchronization rate and saturates eventually showing that a diminishing return rule applies. Next, 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 with the objective of maximizing the performance of applications. To the best of our knowledge, this is the first work that studies this problem.
2) We emulate the performance of two popular applications and obtain insights about the impact of synchronization rates. 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 algorithm. We find that benefits are realized compared with the baseline policy that synchronizes all controller pairs at an 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 algorithm, while Section IV reviews our contribution compared to related works. We conclude our work in Section V.

II. Maximizing Application Performance

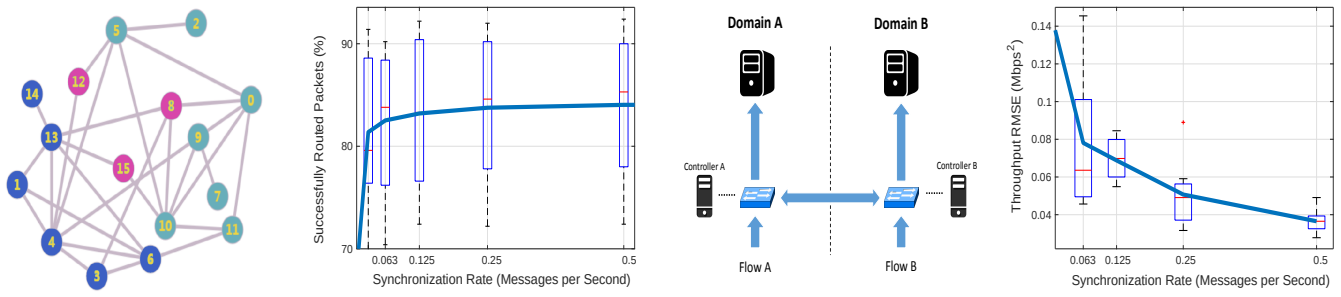
In this section, we present synchronization methods that optimize the performance of specific applications of interest by leveraging elements from the learning theory. Before that, we provide a brief emulation study that will illustrate the impact of synchronization rate on the performance of some popular 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* [13] 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* [14] which is open-source and allows us to develop our own protocols 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



(a) (b) (c) (d)

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.

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 [10], 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 $\mathcal{T} = \{1, 2, \ldots, T\}$ of T time slots, until they can infer a "sufficiently good" solution. To describe such training process, we introduce the vector of synchronization rate variables:

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

where x_{ij}^t indicates the synchronization rate between controllers i and j tried-out in time slot t. R denotes the maximum possible synchronization rate. We further denote by the vector $\boldsymbol{x}^t$ 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 $\boldsymbol{x}^t$ tried-out in a slot t, the application performance will be $\Psi_t(\boldsymbol{x}^t)$. Here, $\Psi_t(.)$ is an unknown function that governs the application performance in slot t. While the overall function is unknown, the single value $\Psi_t(\boldsymbol{x}^t)$ can be *observed* by the system operator *after* the synchronization rate decision $\boldsymbol{x}^t$ 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 is available to the controllers through 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 $\Psi_t(.)$ 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 $\boldsymbol{x}^t = \boldsymbol{x}^{t'}$ in two slots t and t' but observe different performance values $\Psi_t(\boldsymbol{x}^t) \neq \Psi_{t'}(\boldsymbol{x}^{t'})$. 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 $\widehat{\boldsymbol{x}} = (\widehat{x}_{ij} : i, j \in \mathcal{C}, j \neq i)$. This should, ideally, maximize the *average* performance denoted by an (also unknown) function $\widehat{\Psi}(.) = \mathbb{E}[\Psi_t(.)]$. 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, the problem can be formulated as:

$$\max_{\widehat{\boldsymbol{x}}} \quad \widehat{\Psi}(\widehat{\boldsymbol{x}}) \tag{2}$$

$$s.t. \quad \sum_{i \in \mathcal{C}} \sum_{j \in \mathcal{C}, j \neq i} \widehat{x}_{ij} b_{ij} \leq B \tag{3}$$

where B is a constant and inequality (3) ensures that the inferred synchronization rate decision will satisfy a resource constraint.

We need to emphasize that the average performance $\widehat{\Psi}(\widehat{\boldsymbol{x}})$ 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 $\widehat{\boldsymbol{x}}$. 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 $\Psi_t(\boldsymbol{x}^t)$, a learning method would typically try-out the *same* synchronization decision $\boldsymbol{x}^t$ *multiple times*, in different time slots. Then, the empirical mean of the observations will be used to estimate the average performance value $\widehat{\Psi}(\boldsymbol{x}^t)$. By repeating the above training process for every possible synchronization decision, an estimate of the entire objective function $\widehat{\Psi}(.)$ 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 [15] 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 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 $\widehat{\boldsymbol{x}}$, 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 $\widehat{\Psi}(\mathbf{0})$ (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 $\widehat{\boldsymbol{x}}$. At each iteration $k = 1, 2, \ldots, B$, the algorithm will initially pick σ random pairs of controllers as candidates (line 5). For each such pair $p = 1, 2, \ldots, \sigma$, the synchronization decision $\widehat{\boldsymbol{x}}'$ will be set accordingly (line 7) and τ time slots will be spent to estimate $\widehat{\Psi}(\widehat{\boldsymbol{x}}')$ (lines 8-9). The marginal performance gain of switching from decision $\widehat{\boldsymbol{x}}$ to $\widehat{\boldsymbol{x}}'$, denoted by $D(\widehat{\boldsymbol{x}}, \widehat{\boldsymbol{x}}')$, will be estimated (line 10). Among the σ candidate controller pairs, the algorithm will include in the current decision $\widehat{\boldsymbol{x}}$ the pair with the maximum estimated marginal performance gain (line 11).

The algorithm spends τ time slots to estimate $\widehat{\Psi}(\widehat{\boldsymbol{x}})$ for $\widehat{\boldsymbol{x}} = \mathbf{0}$, 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 $\widehat{\Psi}(\widehat{\boldsymbol{x}})$ that is in expectation a factor $1 - e^{-(1-\epsilon)\mu}$ from the optimal where $\epsilon = e^{-\sigma \frac{B}{C(C-1)R}}$ 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 online technical report [16]. We emphasize that the average performance bound depends on the value of μ. This

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

Initialize $\widehat{\boldsymbol{x}} = \mathbf{0}$;
Try out $\boldsymbol{x}^t = \widehat{\boldsymbol{x}}$ and observe $\Psi_t(\boldsymbol{x}^t)\ \forall t \in \{1, \ldots, \tau\}$;
Estimate $\widehat{\Psi}(\widehat{\boldsymbol{x}}) = \frac{1}{\tau}\sum_{t=1}^{\tau} \Psi_t(\boldsymbol{x}^t)$;
for *each iteration* k *from* 1 *to* B **do**
  Pick σ random controller pairs p for which $\widehat{x}_p < R$;
  for *each picked pair* p *from* 1 *to* σ **do**
    Set $\widehat{\boldsymbol{x}}' = \widehat{\boldsymbol{x}}$ where $\widehat{x}'_p = \widehat{x}_p + 1$;
    Try out $\boldsymbol{x}^t = \widehat{\boldsymbol{x}}'$ and observe $\Psi_t(\boldsymbol{x}^t)\ \forall t \in \{(k-1)\sigma\tau + p\tau + 1, \ldots, (k-1)\sigma\tau + p\tau + \tau\}$;
    Estimate $\widehat{\Psi}(\widehat{\boldsymbol{x}}') = \frac{1}{\tau}\sum_{t=(k-1)\sigma\tau+p\tau+1}^{(k-1)\sigma\tau+p\tau+\tau} \Psi_t(\boldsymbol{x}^t)$;
    Set $D(\widehat{\boldsymbol{x}}, \widehat{\boldsymbol{x}}') = \widehat{\Psi}(\widehat{\boldsymbol{x}}') - \widehat{\Psi}(\widehat{\boldsymbol{x}})$;
  end
  Update $\widehat{\boldsymbol{x}} = \text{argmax}_{\widehat{\boldsymbol{x}}'} D(\widehat{\boldsymbol{x}}, \widehat{\boldsymbol{x}}')$;
end
Output: $\widehat{\boldsymbol{x}}$;

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* $\widehat{\Psi}(\widehat{\boldsymbol{x}})$ *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 Results

In this section, we carry out evaluations to show the benefits of the proposed algorithm. Overall, we find that benefits are realized compared with the baseline algorithm that synchronizes all the controller pairs at equal rate (referred to as Homogeneous). Moreover, our Stochastic Greedy algorithm achieves better performance-training time tradeoff than a state-of-the-art learning algorithm (ExpGreedy in [15]).

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). We compare our Stochastic-Greedy algorithm with both the Homogeneous and ExpGreedy algorithms. To eliminate randomness, we run each algorithm 10 times and take the average value.

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 routes optimally more packets than Homogeneous and ExpGreedy algorithms*. The training time required by our algorithm increases linearly with B. On the other hand, the time of ExpGreedy increases more dramatically, which shows that our algorithm is more scalable. 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.

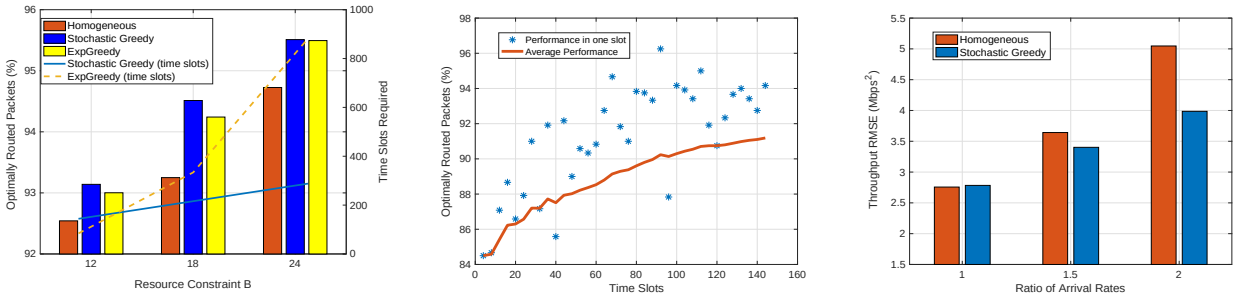
Figure 3b illustrates the learning process when Stochastic Greedy is run for $B = 18$, $\sigma = 2$ and $\tau = 4$. Although in each time slot the algorithm observes a performance value with large randomness, it is able to allocate resources to proper pairs and increase the average performance over time.

Finally, 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 3c 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 [7] and [8]. 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 [17] and the survey in [1].

Some recent works proposed to reduce the overheads of control traffic by strategically placing the controllers in the network [18] or by finding the appropriate forwarding paths for load balancing on control traffic [9]. Nevertheless, the above approaches should be considered as complementary to our work, rather than competitive. On the one hand, the



(a) (b) (c)

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

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. [10] 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 [12].

Few recent works suggested the dynamic adaptation of synchronization period (or rate) among controllers in an eventual-consisted system so as to improve the performance of network applications while maintaining a scalable system [19], [20]. 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 to maximize the performance of applications which may be affected by the synchronization decisions, as highlighted by emulations on a commercial SDN controller. For this objective, we characterized the complexity of the problem and proposed algorithms to achieve the optimal synchronization rates. Evaluation results demonstrated significant performance benefits over the baseline policy that synchronizes all controller pairs at equal rate. Overall, the synchronization problem deserves more research attention, analogous to other problems in SDN framework. Our work in this paper can be seen as a kick-off for systematically studying optimization and learning methods to tackle this important problem.

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] A. Panda, C. Scott, A. Ghodsi et al., "CAP for Networks", *ACM HotSDN*, 2013.

[6] 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.

[7] 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.

[8] Q. Qin, K. Poularakis, G. Iosifidis, L. Tassiulas, "SDN Controller Placement at the Edge: Optimizing Delay and Overheads", *IEEE Infocom*, 2018.

[9] S.-C.- Lin, P. Wang, I.F. Akyildiz, M. Luo, "Towards Optimal Network Planning for Software-Defined Networks", *IEEE Transactions on Mobile Computing*, 2018.

[10] D. Levin, A. Wundsam, B. Heller, N. Handigol, A. Feldmann, "Logically Centralized? State Distribution Trade-offs in Software Defined Networks", *ACM HotSDN*, 2012.

[11] A. Panda, W. Zheng, X. Hu, A. Krishnamurthy, S. Shenker, "SCL: Simplifying Distributed SDN Control Planes", *USENIX NSDI*, 2017.

[12] 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.

[13] B. Lantz, B. Heller, N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-defined Networks", *ACM HotSDN*, 2010.

[14] http://osrg.github.io/ryu

[15] A. Singla, S. Tschiatschek, A. Krause, "Noisy Submodular Maximization via Adaptive Sampling with Applications to Crowdsourced Image Collection Summarization", *AAAI*, 2016.

[16] www.dropbox.com/s/mbmpm46nomi1jfz/AFM18TR.pdf?dl=0

[17] 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.

[18] Z. Su, M. Hamdi, "MDCP: Measurement-Aware Distributed Controller Placement for Software Defined Networks", *IEEE ICPADS*, 2015.

[19] M. Aslan, A. Matrawy, "Adaptive Consistency for Distributed SDN Controllers", *IEEE Networks* 2016.

[20] E. Sakic, F. Sardis, J.W. Guck, W. Kellerer, "Towards Adaptive State Consistency in Distributed SDN Control Plane", *IEEE ICC*, 2017.