Robust SDN Synchronization in Mobile Networks using Deep Reinforcement and Transfer Learning

Akrit Mudvari, Konstantinos Poularakis, Leandros Tassiulas
Yale University, New Haven, CT,
Email: akrit.mudvari@yale.edu, kpoularakis@gmail.com, leandros.tassiulas@yale.edu

***Abstract*—A logically centralized controller architecture for SDN deployments is a well understood and implemented method, however because of issues such as scalability, privacy and more, there is a need to develop and implement a robust physically distributed SDN controller architecture. In a distributed SDN environment, a centralized logical network view needs to be maintained, which means the distributed controllers need a robust method of remaining informed about other controller's network through synchronization. This is specially true in mobile, wireless networks with changing controller and network environment, so to this end we develop a deep reinforcement and transfer learning based method that provides the controllers with an efficient policy for synchronizing with other controllers and maintaining a logically centralized view in such networks. We show that our application-centric method performs well for different kinds of applications including shortest path routing and load balancing, outperforming a reinforcement learning based method as well as a round robin method.**

***Index Terms*—Software Defined Networking, SDN, Transfer Learning, robust learning, Deep Reinforcement Learning, DRL, DDRL, Q learning, synchronization.**

I. Introduction

Software Defined Networking (SDN) is a centralized programmable network architecture that has emerged in the recent decade in response to limitations suffered by the traditional communication architectures, such as complex design that requires expertise for setup and re-configuration, costs associated with hardware adjustments, lack of adaptability and more [1]. To overcome these limitations, SDN shifts the network control functionality (known as control plane) away from the network devices (known as data plane) to a designated software entity, commonly referred to as the controller.

As a larger number of network devices become part of the SDN architecture, scalablity challenges emerge since all the devices need to contact the centralized controller for instructions about their packet forwarding behavior [2]. Besides, security and authorization issues could also be a concern in an architecture where a centralized entity has full control over the network. Finally, a fully centralized SDN controller implies that a failure of the controller leads to all devices to fail [3]. These factors necessitate an SDN paradigm that is physically decentralized but to maintain the benefits of SDN, logically centralized. This means different SDN controllers will be responsible for different subsets of the network devices (we will refer to these subsets as domains) but each controller will coordinate (or synchronize) with the others in order to maintain a global picture of the entire network, allowing for globally optimal decision-making [4].

This work was supported by the Naval Research Laboratory under grant N0014-22-1-2147, National Science Foundation(NSF) under grant 2128530, and NSF-AoF(Academy of Finland) under grant 2132573.

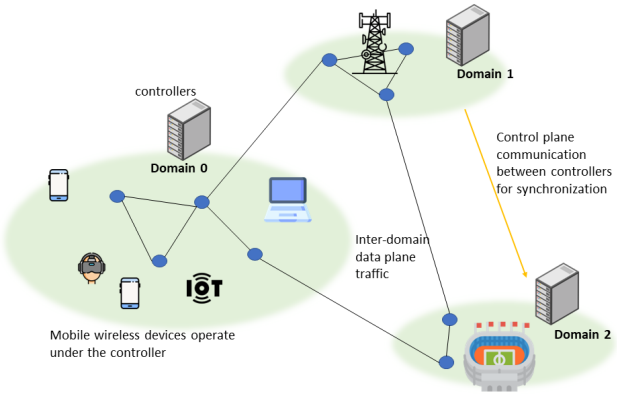

Fig. 1: Illustration of the distributed SDN controller paradigm in a dynamic wireless environment.

As distributed SDN controller is implemented to overcome the aforementioned issues faced by a physically centralized system, compatible protocols and further research is needed to ensure that the systems run correctly and efficiently [3]. For instance, OpenDaylight [5] and ONOS [6], which are some of the leading SDN implementations, rely on RAFT [7] for disseminating synchronization messages among the controllers. RAFT is a consensus algorithm for ensuring that all the participants agree on the same values, which in our case would be the state of the global network. RAFT uses a rigorous method to select a leader for deciding on network states and enforcing logical consistency, and as such is considered to implement a strong consistency model [8]. This approach ensures that an accurate and consistent model of the global network is available to each controller, but the communication costs of maintaining a high level of consistency are also high. So, an approach that could strive for a right balance between consistency requirements and communication overhead by partially sharing network information, called eventual consistent model, could help [9]. Under the eventual consistency model, we can allow for temporary inconsistencies in the model as long as the negative effects are not too adverse in the short run and the models do get eventually synchronized to become consistent. This would allow for the synchronization algorithm to run under a more relaxed setting where delays in updates become more tolerable and the communications cost is lowered. In [10], deep reinforcement learning is used to achieve controller synchronization to improve inter-domain

ICC 2023 - IEEE International Conference on Communications | 978-1-5386-7462-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICC45041.2023.10278580

routing.

In modern wireless communication paradigms including 5G communication [11] and tactical network [12], there is a need for traditional SDN networks to undergo further considerations before being implemented due to the added mobility and dynamism of the participating devices [13]. For instance, the dynamic nature of population density (such as around a football stadium) results in a massive concentration of cellular devices during sporadic occasions, requiring an increased demand for services. In such a case new controller/s could synchronize with the larger network and serve these devices. Another example would be tactical networks where certain users under a hierarchically designated controller would need synchronization with other domains under a separate controller, in a mobile, wireless environment. Figure 1 illustrates such dynamic wireless environments with distributed SDN controller paradigm, where the mobile, dynamic, wireless network environment is controlled by different controllers that need to remain synchronized for continued inter domain communication.

Motivated by the recent promising achievements in the field of deep reinforcement learning and the idea of transfer learning, we attempt to come up with a novel algorithm for robust SDN synchronization in wireless environment. Specifically:

- We propose a double deep reinforcement and transfer learning-based algorithm for optimally updating the network information among controllers in a dynamic environment, where the state of the controllers can abruptly change and new policy is suddenly needed for managing the global consistency under a strict communication budget.
- We show that our method produces state-of-the-art results for different types of applications under a dynamic network environment; this includes applications where the communication network features are shared among the domains (shortest path routing application) and where the end devices' properties are shared for decision making (load balancing computational task application).

II. Problem Formulation

As illustrated in Figure 1, the communication system consists of a certain number of controllers, and each controller is tasked with managing the devices in a particular domain. The domain itself consists of nodes participating in the network, and each node has its networking decisions made by the controller, i.e., flow tables being managed by the corresponding controller. There are different ways in which routing decisions can be made for the nodes in the domain, and these decisions can be made according to various features of the packets being forwarded such as application layer information, assigned priority, and more. In this work, we do not aim to develop or recommend any novel approaches for engineering the traffic between two nodes across the domains or within a domain; we rather work to develop a novel approach for communication between controllers such that the network applications will be automatically and efficiently implemented under the eventual consistency model. This ensures that our approach is easily transferred to any new application or environment. Also, our implementation will be controller-centric: the controller will learn about the best policy to interact with it's current neighbor in a dynamic and robust way, ensuring that the policy easily adapts to the changes/re-configurations across the communication network.

A. Network model

We begin by defining a controller $\hat{c}$ in a set of C controllers distributed across the network. $\hat{c}$ will have as its neighbor all the nodes $c \in C \backslash \hat{c}$, and a controller needs to remain in contact with all the neighbors as much as possible so that the most recent global network model can be maintained.

For any arbitrary application a, we will allocate a certain budget $B^a_{\hat{c}}$, which signifies how often the controller $\hat{c}$ communicates with its neighbors. We take discrete and constant time intervals δt after which the controller will update its version of global network model with relevant information about the neighboring domains. Only application-specific information is shared so as to avoid network overhead at any given update time $t \in 1, 2, 3.....$ So after every δt, $B^a_{\hat{c}}$ out of $|C \backslash \hat{c}|$ neighbors are communicated with, and the collected information is used to update $\hat{c}$'s global network model. $\hat{c}$'s policy for updating with a neighbor c at time t is given by $x_{\hat{c}c}(t) \in \{0, 1\}$, where a value of 1 signifies updating will take place, and a value of 0 signifies that it will not, resulting in a temporary inconsistency. The following inequality defines the constraint to ensure that communication budget is not violated by any controller and any application at a given time:

$$\sum_{c \in C \backslash \hat{c}} x_{\hat{c}c}(t) \leq B^a_{\hat{c}} \tag{1}$$

Depending on the environment and to a certain extent the requirements of the application as well, we expect the allocation policy, inferred from the values of $x_{\hat{c}c}(t)$, to be optimal in different ways. For instance, if the task is shortest path routing and if the domain has a network of nodes that is very large and dense, it probably makes sense to allocate more of the budget to communicating with that particular domain, since the failure to maintain consistency will result in failure to communicate with more nodes. Since the policy is dependent on multiple aspects of the application and the environment, it becomes very difficult to formulate and predict the exact policy that will lead to the best allocation of the budget (although it may sometimes be possible to have a simple algorithm do the optimal allocations in certain environment and applications.) To overcome this complexity, we develop a DDRL (Double Deep Reinforcement Learning) model that learns an efficient allocation policy over a period of time.

B. RL formulation

The synchronization problem can be formulated as a Markov Decision Process (MDP); the first step towards this realization is deciding what state variables could be conveniently chosen to make the selections for synchronization, such that different applications, in different and dynamic environments, can use the same approach. Furthermore, minimizing the state

parameters allows us to avoid using large state space that makes the algorithm very slow to train and harder to transfer the learning to changing environments. So in our approach we choose as our state space, S, for each neighbor c, the number of time intervals since its domain information was communicated to the controller $\hat{c}$, resulting in a vector of size $|C\backslash\hat{c}|$. This information is universal to every application, allowing the reinforcement learning to be applicable to any arbitrary application and topology.

Action space in this case is also finite and discrete, with each action corresponding to a unique allocation decision. So action space is a a vector denoting which neighbors' information is to be updated at the given time slot, under the budget constraint B (which defines the number of neighbors the controller may interact with) , leading to an action space, A, of size:

$$|A| = \frac{|C\backslash\hat{c}|!}{B!(|C\backslash\hat{c}| - B)!} \tag{2}$$

Reward $R(S, A)$ represents the immediate reward allocated when for a given state in S, an action in A is selected. This is an arbitrary reward and can be selected according to the application requirements. Thus our MDP is denoted by $M = (S, A, R)$. The aim of our algorithm will be to find the best policy $\pi*$ that maximizes the sum of reward over a period of time T. Throughout this paper, we will refer to this time period as the number of steps, and the number of simulations we run for the RL agent to learn will be referred to as the number of episodes.

In this RL scheme, at a given time t and a certain network environment and state $s(t)$, the RL agent will decide on an action $a(t)$. The action changes the state of the environment, resulting in a new state $s(t + 1)$ and generates a reward $r(t)$, which represents how well the action was able to help minimize the inconsistency in the controller's global model in a way that benefits the application. To illustrate further let's refer to domain 0's controller in Figure 1, $\hat{c}$, as the primary controller and the rest (1 and 2) as neighboring controllers. Here, the primary controller is the controller that is trying to learn the optimal synchronization policy. Let's assume that for application a, the relevant information of a domain is the state of all the links within this domain. Then the state $s(t)$ is a vector where $\hat{c}$ stores the passage of time slots since the last time $\hat{c}$ communicated with each of the neighboring controllers. The action $a(t)$ then is the decision that $\hat{c}$ makes of utilizing the budget to communicate with certain neighbor/s at time t. Once this decision has been made, we calculate how well this decision resulted in the proper utilization of the partially consistent network information by the application a, resulting in a reward $r(t)$ accordingly. The optimal policy $\pi^a_{\hat{c}}$ for controller $\hat{c}$ and application a is defined as a set of actions across the time steps that yields the maximum long term reward as described in [14].

C. Robust DDRL formulation

The robust algorithm we implement to learn the policy uses a Q function as described in [15], with the values of Q function calculated for explored state-action pairs using DNNs. We implement double DQRL approach, where a lagging DNN model called target network is used to learn slower than the main network, and in the process avoid over-fitting that DNN models could run into [14].

The DNN used to implement the Q-function in our algorithm is multiplayer perceptron [16], with the state values as the inputs, the action values as the outputs, and two hidden layers in between. The input size of this network is thus $|C\backslash\hat{c}|$ representing the state values for each neighbors of controller $\hat{x}$, and the output size is the number of actions available, as calculated in Equation (2). For a given state (input parameters), the selected action is the action value where the output node has the highest value, which we will take as the Q function output for the given state-action pair. The MLP is implemented using Pytorch library [17], which allows us to construct a neural network as a graph-like structure and performs backpropagation without manually needing to calculate the gradients. The optimizer used by the DNN for learning purpose is Adam [18]. One major advantage of our DNN-based formulation is that not all of the possible state-action pairs need to be explored.

We extend the DDRL algorithm to account for changing environments, i.e., where the domains and the corresponding controllers in the global network evolves overtime. Each time a new controller is introduced as a neighbor to controller $\hat{c}$, the state space changes to add the new neighboring controller, resulting in a state space of size $|S(t + 1)| = S(t) + 1$. Similarly, the action space changes as well, with the new action-space size given by Equation (2).

Certain aspects of the above DDRL are left unmodified each time a new controller enters the global network and transfer learning has to take place for the new environment. Specifically, we implement a form of transfer learning , as described in Algorithm 1, that adapts the parameters from the previously learned DNN such that the hidden layers remain unchanged, and so do the previously trained neurons of the input and the output layers corresponding to the controllers that have not changed since the previous time interval. What changes is the input and output layer neurons we add to capture the new controllers entering the network compared to the previous time interval. The values of these new neurons are obtained by finding the "nearest neighbor" from the existing neurons. For a given application and input neurons, a certain feature/s, $F(a)$ is decided as being relevant. This feature space can be shared among applications, i.e., for both shortest path routing and load balancing applications, we consider F to include the network's size. For action space, we simply consider a neighbor with similar vector features, i.e, k-nearest neighbor has as input space the action vectors. The parameters for this new DNN, θ_{temp}, is now used as θ for the RL. The new eplison decay rate, ϵ, is calculated as $P_\epsilon = \frac{1}{(1+E/n\epsilon)}$. The memory of the DDRL used to remember batches for learning is cleared since the state and action space has changed. With these changes, made, the learning process can continue.

We verify the efficiency and robustness of our method on multiple test cases to ensure that it works for different types of applications. The two applications we select for this purpose

Algorithm 1 Robust DDRL scheduler for SDN synchronisation

Input: parameters for initializing the NNs (NNs will start with randomized parameters θ), learning rate α, batch size b, decay rate ϵ, reset rate n, soft update rate for target network τ, environment features for transfer learning F
Output: learned parameters θ for DNN
Initialize: Main and target NN with parameters θ (randomized)
Initialize: matrix D with (s, a, r) tuples to store

for episodes $E = 1, ..., M$ **do**
  Set: exploration probability $P_\epsilon = \frac{1}{(1+E/\epsilon)}$
  Set: initial state s_0 to zero vector, $t = 0$
  while $t \leq T$ **do**
    if NewControllerEvent **then**
      reset D
      reset exploration probability to $P_\epsilon = \frac{1}{(1+E/n\epsilon)}$
      create new DNN with parameters θ_{temp}
      for layers l in DNN **do**
        if l is first or last layer **then**
          if l is first layer **then**
            add 1 node
            $k = |S(t-1)|$
            obtain k-nearest neighbors according to $F(a)$
          else
            add $|A(t-1)|\frac{B}{|C\setminus\hat{c}|+1-B}$ nodes
            $k = |A(t-1)|$
            obtain k-nearest neighbors according to action-space vectors
          end if
          assign parameters of the nearest neighbors to each new node/s created
        end if
      end for
      $\theta \leftarrow \theta_{temp}$
    end if
    with probability P_ϵ select random action a_t
    otherwise select $a(t) = argmax_a Q(s(t), a; \theta)$
    From network environment obtain the reward $r(t)$ according to $(s(t), a(t))$, and the observation $\phi(t) = s(t+1)$
    Store $(s(t), a(t), r(t), \phi(t))$ in D
    Sample random mini-batch of size b from D, each entry denoted by $(s_j, a_j, r_j, \phi_j), j = 1, ..., b$
    for $j = 1, ..., b$ **do**
      $y_j = r_j + \gamma Q_\theta(S_j, \underset{a\in A}{\text{argmax}} Q_{\hat{\theta}}(S_j, A_j)$
      perform gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to θ, to obtain $\nabla_\theta L(\theta)$
    end for
    $\theta \leftarrow \theta - \alpha\nabla_\theta L(\theta)$
    $\hat{\theta} \leftarrow \tau\theta + (1-\tau)\hat{\theta}$
  end while
end for

are shortest path routing (SPR) and load balancing (LB). As we will explain in detail in the next sections, SPR is an application that relies on the communication links, and LB is an application that primarily relies on the node features. This way, we show that the method is likely to work for a broad array of applications, even in a dynamic environment where network topologies and controller assignments are changing.

D. Shortest path routing

In an SDN environment, unlike traditional shortest path based routing protocols such as OSPF [19] and OSLR [20], the roles of the nodes are limited to simply receiving the flow table information form the controller for packet forwarding. The controller will obtain the latest network topology information and make the globally optimal decision for each of the nodes in its domain [21]. Hence, it falls on the controllers to constantly monitor and maintain the up-to-date and efficient routing policy, as well as forward these policies to the nodes they control. In the distributed SDN environment that we deploy, there is a need to not just monitor the links within a controller's domain, but also the topologies of the neighboring domains. By synchronizing with the neighboring controllers, each controller is able to keep an up-to-date information for the neighboring domains, allowing the controllers to correctly populate the routing tables of the switches in its domain without leading to wrong or inefficient routing.

E. Load balancing

Load balancing refers to effectively distributing incoming traffic across a group of potential servers, and SDN has been shown to effectively improve performance over traditional methods [22]. In our distributed SDN environment, we will refer to LB as controller deciding on which neighboring domain has the best resources to complete the tasks required by its domain. The packets for the task could be forwarded more efficiently when the domain is synchronized and made aware about the state of all its neighboring domains.

III. Evaluation and Discussion

In this section, we demonstrate the performance of our proposed robust scheduler compared to other synchronization schemes. First, we discuss our simulation environment in section III-A, followed by the results showing improvements due to our method in III-B. We will present some further findings in III-C, i.e., the performance over a range of synchronization budgets.

A. Network model setup

For both applications, we use the same network topologies that are randomly generated and evolve over time. It is ensured that the topology generated at the beginning is fully connected. We deploy c controllers, with each of them randomly getting a domain in the network. The number of controllers in the topology vary from 2 to 15, with 7 controllers initially selected. Since the global network environment is dynamic, new controllers with their own domains enter the communication network, leading to up to 18 neighbors, each controlling their own domain. The budget of communication at each time

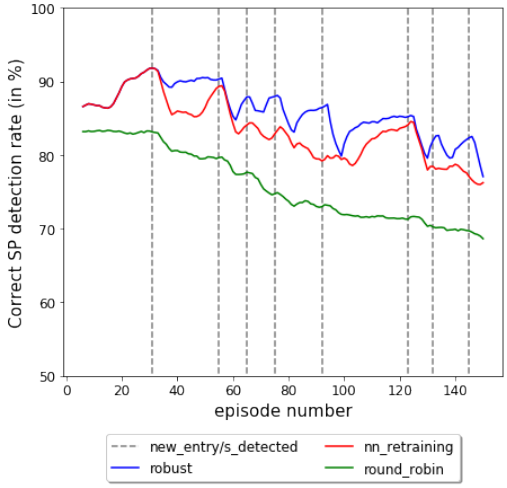

Fig. 2: Performance Results for SP application

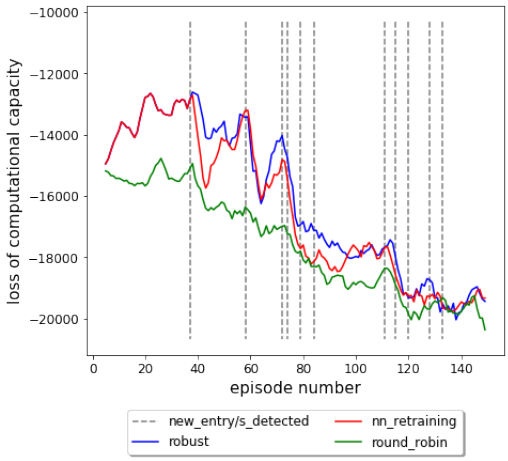

Fig. 3: Evaluation Result for LB application

slot is limited to 2/7 ($\frac{B}{|C\backslash\hat{c}|}$) at first, but we could see the efficiency at different budget levels as well. As the network evolves, the capacity of each server in different domains is randomly changing, and the network links randomly break or re-attach to simulate a dynamic network environment. We also demonstrate the effect of changing network budget for each of the applications.

For the robust DDRL learning, we selected multiple hidden layers with 2-3 layers and around 20-30 neurons per hidden layer (the state and action space are a function of neighbor space, as discussed in section II). In a future work, we could easily reduce this number as changing the values was still giving a better performance for our method, which suggests that pruning the model significantly should be possible. The learning rate was set between 0.001 (for LB) and 0.01 (for SPR), the batch size for learning was 256, the buffer size for storing $(s(t), a(t), r(t), \phi(t)$ was maintained at 40,000, the epsilon decay denominator ϵ was 10, the double deep learning's target network soft-learning rate τ was set to 0.01, and the discount rate γ was set to 0.1 (for LB) and 0.4 (or SPR). The epsilon reset rate for tranfer learning n was 3, while the application features for all applications F was a dictionary consisting of the number of nodes and the number of links for each domain. While we allowed for best hyper parameters, it could be observed that any of these parameters could be set to the same values for all the different applications without losing performance significantly. But we will focus on each application next. The reward function is a normalized sum of immediate rewards per step for each episode, which is unique to each application, and is defined in a way that best-suites the optimization goals for the application.

B. Evaluation results

Evaluation of SPR: In Figure 2, we demonstrate that our method performs better than other approaches in predicting the best scheduling policy for managing shortest path routing application under the budget constraint. We can observe that our method, called robust, does significantly better than the round robin approach. It also does better than a simple non-robust implementation, where no transfer learning takes place (called nn retraining). This is observed by the fact that in Figure 2 the correct shortest-path detection rate for our method, robust is higher when compared to other methods. This signifies that our method was able to more frequently detect the shortest paths to nodes in neighboring domains. Over all episodes, our method outperforms the non-transfer (nn-retraining) approach by 2.9 percent and the round robin approach by 13.1 percent when it comes to correctly identifying the shortest paths. The reward used for RL evaluation is given by $\sum_t k.detect(t)$ where $detect(t)$ is sum of all correct detection of link status in each step t throughout an episode. But for measuring the performance of robust, we opted to measure the correct SP detection rates, i.e. what portion of actual shortest paths were correctly identified with the limited information.

Regarding the improvement over nn training with our robust method, first please note that each time a dotted vertical line appears in the graph, it indicates that the number of domains increased, as new controllers join as a neighbor. While the method not utilizing the transfer learning approach suffered a significant drop in rewards each time a new controller was added, robust method tended to recover much faster.

Evaluation of LB: Similarly, as shown in Figure 3, we see our approach surpasses other approaches in predicting the best load balancing policy, i.e., in determining where the best server location is in each time step. We observe that robust does better than nn retraining and round robin approaches most of the episodes, as signified by higher reward values; here the reward for each step is the difference between the assigned server capability of the actual best server location and the perceived server location under the synchronization policy. Note that if all neighboring domains were perfectly synchronized, this difference would not exist but under the limited budget, our policy outperforms nn-retraining approach by 1.5 percent and round robin approach by 8.0 percent. We also tested with the rewards only accrued when the best server was correctly detected and in that case, robust outperformed non-transfer method by 6 percent and round robin by 40

percent, i.e., the best server was detected 40 percent more often.

C. Other findings

In the earlier sections, we showed how our method produced better results for different types of applications for a given budget. For the method to function in a range of network environments where the communication overhead budget may vary depending on multiple factors such as data plane traffic and controller's capabilities, we would have to ensure that the approach is efficient most of the times. In Figure 4, we show that our method outperforms other approaches for different budget levels in case of each of the applications. Figure 4(left) shows that the loss of computational capacity is always better for our method across different budgets ranging from 28 percent to 58 percent, while in Figure 4(right) we see that the correct shortest path detection rates are always the highest for our method across the same budget range.

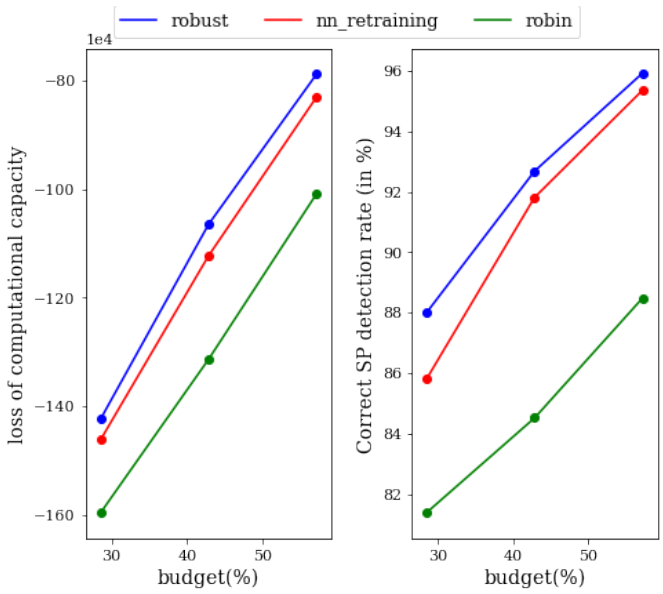

Fig. 4: Left: Loss of computational Capacity for different budget availabilities for LB application. Right: Correct SP detection rate for different budget availabilities for SPR applcation.

IV. Conclusion

In this paper, we explored a novel approach for scheduling synchronization messages among SDN controllers that takes into consideration the dynamic nature of the network topology, including the nodes and the links, but also the dynamic behavior of the controllers in a distributed SDN paradigm. This paradigm will become vital especially in wireless, mobile communication networks where the scheduler would need to be robust in order to tackle the volatility of such network environment. Towards this goal, we utilized a double deep reinforcement learning approach, alongside transfer learning, to develop a scheduler that outperforms other existing methods for quickly learning the best policies and adapting to network dynamics. We showed that our scheduler works for different types of network applications, i.e., shortest path routing and load balancing.

References

[1] M. Karakus and A. Durresi, "A survey: Control plane scalability issues and approaches in software-defined networking (sdn)," *Computer Networks*, vol. 112, pp. 279–293, 2017.

[2] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 136–141, 2013.

[3] Y. E. Oktian, S. Lee, H. Lee, and J. Lam, "Distributed sdn controller system: A survey on design choice," *computer networks*, vol. 121, pp. 100–111, 2017.

[4] K. Poularakis, Q. Qin, L. Ma, S. Kompella, K. K. Leung, and L. Tassiulas, "Learning the optimal synchronization rates in distributed sdn control architectures," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1099–1107.

[5] L. Foundation. opendaylight. [Online]. Available: https://www.opendaylight.org/

[6] O. N. Foundation. Onos. [Online]. Available: https://opennetworking.org/onos/

[7] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, 2014, pp. 305–319.

[8] E. Sakic and W. Kellerer, "Response time and availability study of raft consensus in distributed sdn control plane," *IEEE Transactions on Network and Service Management*, vol. 15, no. 1, pp. 304–318, 2017.

[9] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker, "Cap for networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013, pp. 91–96.

[10] Z. Zhang, L. Ma, K. Poularakis, K. K. Leung, and L. Wu, "Dq scheduler: Deep reinforcement learning based controller synchronization in distributed sdn," in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–7.

[11] A. Ghosh, A. Maeder, M. Baker, and D. Chandramouli, "5g evolution: A view on 5g cellular technology beyond 3gpp release 15," *IEEE access*, vol. 7, pp. 127 639–127 651, 2019.

[12] K. Phemius, J. Seddar, M. Bouet, H. Khalifé, and V. Conan, "Bringing sdn to the edge of tactical networks," in *MILCOM 2016-2016 IEEE Military Communications Conference*. IEEE, 2016, pp. 1047–1052.

[13] G. Li, A. Mudvari, K. Gokarslan, P. Baker, S. Kompella, F. Le, K. M. Marcus, J. Tucker, Y. R. Yang, and P. Yu, "Magnalium: Highly reliable sdc networks with multiple control plane composition," in *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 2019, pp. 93–98.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[15] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.

[16] M. Kumar and N. Yadav, "Multilayer perceptrons and radial basis function neural network methods for the solution of differential equations: a survey," *Computers & Mathematics with Applications*, vol. 62, no. 10, pp. 3796–3811, 2011.

[17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[18] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[19] J. Moy, "Ospf version 2," Tech. Rep., 1997.

[20] T. Chausen and P. Jacquet, "Optimized link state routing protocol (oslr)," IETF RFC, Tech. Rep., 2003.

[21] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about sdn flow tables," in *International conference on passive and active network measurement*. Springer, 2015, pp. 347–359.

[22] M. Qilin and S. Weikang, "A load balancing method based on sdn," in *2015 Seventh International Conference on Measuring Technology and Mechatronics Automation*. IEEE, 2015, pp. 18–21.