# DRL based SDN Controller Synchronization Policy Design for joint Communication and Computation Resource Optimisations

Ziyao Zhang*, Liang Ma†, Konstantinos Poularakis‡, Kin K. Leung*, Jeremy Tucker§ and Ananthram Swami ¶
*Imperial College London, London, United Kingdom. Email: {ziyao.zhang15, kin.leung}@imperial.ac.uk
†IBM T. J. Watson Research Center, Yorktown Heights, NY, United States. Email: maliang@us.ibm.com
‡Yale University, New Haven, CT, United States. Email: konstantinos.poularakis@yale.edu
§U.K. Defence Science and Technology Laboratory, Salisbury, United Kingdom. Email: jtucker@mail.dstl.gov.uk
¶U.S. Army Research Laboratory, Adelphi, MD, United States. Email: ananthram.swami.civ@mail.mil

***Abstract*—In distributed software-defined networks (SDN), multiple physical SDN controllers, each managing a network *domain*, are implemented to balance centralised control, scalability, and reliability requirements. In such networking paradigms, controllers synchronize with each other, in attempts to maintain a logically centralised network view. Despite the presence of various design proposals for distributed SDN controller architectures, most existing works only aim at eliminating anomalies arising from the inconsistencies in different controllers' network views. However, the performance aspect of controller synchronization designs with respect to given SDN applications are generally missing. To fill this gap, we formulate the controller synchronization problem as a *Markov decision process (MDP)* and apply reinforcement learning techniques combined with deep neural networks (DNNs) to train a *smart, scalable, and fine-grained* controller synchronization policy, called the *Multi-Armed Cooperative Synchronization (MACS)*, whose goal is to maximise the performance enhancements brought by controller synchronizations. Evaluation results confirm the DNN's exceptional ability in abstracting latent patterns in the distributed SDN environment, rendering significant superiority to MACS-based synchronization policy, which are $56\%$ and $30\%$ performance improvements over ONOS and greedy SDN controller synchronization heuristics.**

## I. Introduction

Software-Defined Networking (SDN) [1], a newly-developed networking architecture, improves network performance due to its programmable network management, easy reconfiguration, and on-demand resource allocation, which has therefore attracted considerable research interests. One key attribute that differentiates SDN from classic networks is the centralisation of network control, for which all control functionalities are abstracted and implemented in the *SDN controller* sitting in the control plane, for operational decision-making. While the data plane, consisting of *SDN switches*, only passively executes the instructions received from the control plane. Since the logically centralised SDN controller has full knowledge of the network status, it is able to make the global optimal decision. Yet, such centralised control suffers from major scalability and reliability issues. In this regard, *distributed SDN* [2] is proposed to balance the centralised and distributed controls.

A distributed SDN network is composed of a set of subnetworks, referred to as *domains*, each managed by a physically independent SDN controller. In the context of tactical coalition network, each coalition partner is responsible for deploying their own domains where designated controllers manage the respective domains in wireless manners. The physically distributed controllers synchronize with each other to maintain a logically centralised network view, which is referred to as *controller synchronization*. Since complete synchronization among controllers, i.e., all controllers always maintain the same global view, will incur high costs especially in large networks [3], [4], most practical distributed SDN networks can only afford partial inter-controller synchronizations and allow temporary inconsistency in controllers' network view, which is known as the *eventual consistency model* [5].

The eventual consistency model permits temporarily inconsistent network views among physical distributed controllers, to avoid otherwise excessive and prohibitive synchronization overheads. In the meantime, higher network availability, i.e., the ability to provide network services, is realised at the cost of temporary inconsistency according to the CAP (Consistency, Availability, Partition Tolerance) theorem [5]. Therefore, there exists a fundamental trade-off between the consistency of network state distributions and the control objective optimality, as reported in [6]. Existing works have identified and addressed some serious anomalies arising from controllers' inconsistent network views, such as loopholes [7], blackholes [8], and other problems caused by policy inconsistencies [9]. Yet, despite these efforts aimed at eliminating inconsistency-caused anomalies, *we have not seen any notable proposals on fine-grained controller synchronization designs which are tailored for SDN application scenarios with specific performance metrics.* The urgency to fill this gap is especially pronounced when SDN technologies are discussed in a wider range of contexts, where advanced applications are developed on top of SDN-enabled 5G, smart grid, and ISP networks; all these cases require the support of new and finer-grained controller synchronization models [10].

In this regard, we approach the controller synchronization problem with the aim of developing fine-grained controller synchronization policies for enhancing given performance metrics. Complementary to the existing works that propose controller architectures/mechanisms to enable controller synchronization and make sure the process is error-free; ours is performance-focused, for which we look at which controller(s) should synchronize at each time instance so that the given performance metric is maximised.

To this end, we first define (1) the SDN application of interest whose performance depends on joint communication and computation resources optimisations; and (2) the corresponding performance metric. Then, we formulate the problem of developing the controller synchronization policy that maximises the defined performance metric as a *Markov decision process (MDP)*, which can be solved by employing reinforcement learning (RL) techniques. RL-based approaches are especially appealing for developing the controller synchronization policy under distributed SDN for the following reasons. (1) The abundance of network data made available by SDN switches through the OpenFlow protocol [11] builds up a pool of past experiences which are the ideal "trial-and-error" inputs for RL algorithms. (2) Different SDN domains can be highly heterogeneous; as such, SDN networks are complex systems. Therefore, accurately modelling such systems becomes extremely difficult and mathematically intractable. In light

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

of this, the model-free RL-based approaches are especially attractive, as they come without any constraints on network's structure or its dynamicity, thus adaptable for handling real-world SDN networks. Inspired by its recent successes, we propose a deep reinforcement learning (DRL)-based controller synchronization framework, which employs deep neural networks (DNN) to generalise synchronization policy estimations, called Multi-Armed Cooperative Synchronization (MACS) to assist controllers in learning synchronization policies based on past experiences. Given the high complexity of our problem and multiple issues arise during training (see Section IV-A), we carefully design MACS to address these challenges (see Section IV-B). Since the synchronization decision at one time instance could have lasting effects, the goal of MACS is to maximise the long-term performance enhancement brought by controller synchronizations with respect to (w.r.t.) the given performance metric.

Extensive evaluations show that MACS, operating on limited controller synchronization budgets, is nearly as good as the full synchronization scenario where all distributed controllers are always synchronized. Overall, MACS outperforms the default anti-entropy synchronization algorithm implemented in ONOS controllers and a greedy heuristic by up to $56\%$ and $30\%$, respectively. To the best of our knowledge, MACS is the first DRL-based SDN controller synchronization scheduler which yields fine-grained synchronization polices aiming at optimising both communication and computation resources.

## II. System Description

### A. SDN Controller

Under distributed SDN architectures, all control functionalities are abstracted and implemented in distributed SDN controllers, which are responsible for their designated domains (referred to as *domain controllers*), for operational decision-makings. Domain controllers have full and up-to-date view of their designated domains; they synchronize with each other to a maintain logically centralised network view for better network performance for the role of controller synchronization. Routine and frequent control tasks are handled by domain controllers in individual domains, based on their network views. In addition, we assume that there exists a *central controller*, whose only responsibility is to develop synchronization policies and coordinate controller synchronizations, based on information supplied by domain controllers. Such a central controller could be one of the existing distributed controllers, or an independent control unit. All controllers therefore forms the *SDN control plane*.

### B. SDN Domain

In this paper, an SDN domain refers to the collection of network elements managed by a domain controller. These elements may include SDN switches, servers, and users that are connected to the SDN switches. All SDN switches, which constitute the *data plane*, maintain flow tables with forwarding rules installed by domain controllers. Another important advantage brought by the SDN paradigm is the ability to virtualise network services and install them on general purpose *servers* inside SDN domains. This is in clear contrast to the traditional networking paradigm where network services are provided by dedicated equipments with designated functionalities running on specific protocols and proprietary configuration tools. Each domain contains one or multiple gateway routers (or switches, in this paper, they are all referred to as *gateway routers*) connecting to other SDN domains. In addition, users inside domains are connected to SDN switches that carry out data forwarding functionalities; they form the *SDN data plane*.

### C. Controller Synchronization Application

To materialise potential performance gains controller synchronization can bring under distributed SDN, we focus on an application scenario where find-grained information about communication and computation resources are essential for its operation. Therefore, we choose *service path construction* as the application of interest. In the Network-as-a-Service (NaaS) SDN environment, QoS-aware service path construction is a crucial problem in the context where network services are virtualised in servers. Specifically, we investigate the problem where several network services, e.g., wireless access admission, firewall, etc., are installed on servers across all SDN domains. Requests for services are submitted by users to domain controllers, who construct service paths for requests submitted, based on their network views. Note that the process of finding a service path is an anycast problem, as a service can have multiple installations in different domains. In order to calculate the best service path w.r.t. the given performance metric, domain controllers rely on up-to-date information about other domains (e.g., traffic levels, network delay, available computation and services, etc.) gained through controller synchronization.

### D. Network Model

We formulate the distributed SDN network as a directed graph, where $m$ vertices are connected via directed links. Let the graph representing the SDN network be denoted by $\mathcal{G} = (V, E)$ ($V$/$E$: set of vertices/edges in $\mathcal{G}$, $|V| = m$), which is referred to as the *domain-wise topology*. The existence of two edges in $e_{1,2}, e_{2,1} \in E$ connecting two vertices $v_1, v_2 \in V$ in the domain-wise topology implies that the two network domains corresponding to $v_1$ and $v_2$ are connected. Then, we further associate weights for the directed inter-domain links, which represent *gateway delays* (see Section III-A). We use $\mathcal{L}_i$ to denote the set of outgoing edge weights from the vertex of domain $\mathcal{A}_i$ in the network. For instance, let $\mathcal{A}_1$ and $\mathcal{A}_2$ denote the network domains corresponding to vertices $v_1$ and $v_2$ in the graph; then the weight of edge $e_{1,2}$, denoted by $l_{1,2} \in \mathcal{L}_1$, represents the estimation of latency a packet going thorough domain $\mathcal{A}_1$ and entering domain $\mathcal{A}_2$ should expect to experience in $\mathcal{A}_1$ (i.e., the *gateway delay*); similarly, $l_{2,1} \in \mathcal{L}_2$ is the latency a packet going thorough domain $\mathcal{A}_2$ and entering domain $\mathcal{A}_1$ would experience in $\mathcal{A}_2$.

Let $\mathcal{C}$ be the set of all installed services on servers located across different domains in the SDN network. Let $c_j^{(i)}$ denote service $i$ installed in domain $\mathcal{A}_j$. Note that we do not differentiate two identical services installed in the same domain. Then, $\mathcal{D}_j$ is the set of server delays of all service installations in domain $\mathcal{A}_j$, and $d_j^{(i)} \in \mathcal{D}_j$ denotes the waiting time before a request for service $i$ starts being processed in domain $j$ (i.e., the *server delay*, see Section III-A).

***Discussion:*** Note that due to network dynamicity, e.g., traffic levels, user demand for service patterns, etc., the values of $l_{i,j}$ and $d_j^{(i)}$ are time-varying. With the SDN settings described in this section, domain controllers always have the up-to-date status views of network elements residing in their domains; in other words, the controller of domain $\mathcal{A}_i$ always knows the newest $\mathcal{L}_i$ and $\mathcal{D}_i$. The controller of domain $\mathcal{A}_i$ relies on synchronizations with other domain controllers (see Section III-B) to learn up-to-date $\mathcal{L}_j (j \neq i)$ and $\mathcal{D}_j (j \neq i)$.

## III. Problem Formulation

### A. Performance Metric

For the user who submitted a service request, the sooner the request gets served, the better. Therefore, the gap in time between the submission of a service request and the server starting processing the request is a natural performance measurement of qualities of constructed service paths, which we call the *request latency*.

Request latencies consist of two parts: transit latency and the waiting time at the server. Many factors, such as the link congestion levels, the number of hops, may contribute to the overall transit latency for the inter-domain routing of a service request from the

user domain to the domain of the chosen server. For easier modelling, we use the delays incurred at egress gateway routers of SDN domains (referred to as *gateway delays*), which are usually the bottlenecks [12], to abstract the transit latency incurred traversing through SDN domains. If a service request is submitted to a server located in the same domain as the user, transit latency incurred is assumed to be negligible. On the other hand, delays incurred at the server are the waiting times in the server before available computation resources can be assigned for processing the submitted requests, which we refer to as the *server delay*.

Therefore, we define the performance metric w.r.t. a constructed service path as the accumulated gateway delays en route the service path and server delay at the chosen server, which is referred to as *request latency* in the sequence.

### B. Synchronization Among SDN Controllers

For the application of interest introduced in Section II-C, controller synchronization levels directly affect the quality of constructed service paths. Here, we formally define controller synchronization w.r.t. our service path construction problem in this subsection. First, we define the unit that quantises the synchronizable domain information.

**Definition 1.** *The gateway delay between a pair of domains, or the server delay of a service is referred to as a Basic Information of Synchronization (BIS).*

A BIS corresponds to the most fundamental piece of information that can be synchronized to domain controllers. Note that in existing distributed controller implementations such as ONOS, when two controllers synchronize, they exchange their entire state information. In this paper, we propose a more fine-grained synchronization policy where only selected state information is exchanged. According to the above definition, an SDN network $\mathcal{G}$, with the set of installed service $\mathcal{C}$, has $N = |E| + |\mathcal{C}|$ number of BISes in total (see Section II-D). With the concept of BIS, we then formally define controller synchronization in the following definition.

**Definition 2.** *Controller synchronization is the process of domain controllers broadcasting/receiving up-to-date BISes originated in their domains/received from other domain controllers.*

Definition 2 implies that controllers synchronize with each other in the way that selected up-to-date BIS(es) are broadcasted to all domain controllers via control plane messages. Then, all domain controllers update their network views by incorporating the received up-to-date BIS(es) from other controllers and BIS changes reported in their own domains.

As discussed briefly in Section I, frequent dissemination of inter-controller synchronization messages introduce prohibitively large overheads [3], [4]. Therefore, the number of synchronization messages that can be exchanged at a time is limited, for which we introduce the *synchronization budget*.

**Definition 3.** *The synchronization budget at a time is the maximum number of BISes that can be broadcasted simultaneously.*

***Discussion:*** Note that in addition to the synchronizable BIS, all domain controllers always have knowledge of the correct domain-wise topology (without edge weights) and the available services in every other domains.

### C. Service Path Construction using BIS Information

Due to the flexibility and programmability of the SDN, there are potentially many ways in which domain controllers calculate service paths. In this section, we describe a simple service path construction mechanism that uses BISes and aims at minimising the overall request latency. Note that in this paper, it is *not* our intention to design any new such mechanisms; we use this simple and representative mechanism for the sake of problem formulation. Specifically, the service path construction mechanism consists of the following steps.

***Step 1***: The domain controller that receives a service request (*source controller* in the sequence) calculates the minimum accumulated gateway delay(s) of service path(s) leading to all possible server(s) that host the requested service (recall that the process of finding a server is an anycast problem).

***Step 2***: The domain controller calculates the request latency (latencies) for the requested service for all potential service path(s) in ***Step 1*** by combining their accumulated gateway delays and the corresponding server delays. The source controller chooses the service path that incurs the lowest request latency based on its calculation results.

***Step 3***: The forwarding rule of the service request is installed on involved SDN switches in forms of flow table entries.

Applying the above service path construction mechanism, only the source controller decides the domain-wise service path based on its view of the network status. Other domain controllers are deliberately left out in calculating service paths to avoid forwarding anomalies, e.g., routing loops and black holes, which could arise if different domain controllers with heterogeneous network views attempt to independently calculate service path for a packet transiting through their domains.

### D. The Objective of Controller Synchronization Policy

Two questions motivate our definition of the objective of controller synchronization. First, how does the central controller develop the synchronization policy that maximises the performance metric, given the limited synchronization budget? Second, since a synchronization decision has lasting effects, how does the synchronization policy maximise the performance enhancement of controller synchronization over time? Here, a *synchronization policy* refers to a series of synchronization decisions (i.e., which up-to-date BIS(es) to broadcast, subject to the available synchronization budget) over a period of time. With this in mind, we state the objective below.

***Objective***: In dynamic networks where gateway and server delays are time-varying, given the controller synchronization budget, how do domain controllers synchronize with each other by broadcasting up-to-date BISes, to maximise the performance improvements brought by controller synchronizations (i.e., reductions in average request latency due to the availability of accurate BISes via synchronizations) over a period of time?

### E. Markov Decision Process (MDP) Formulation

MDP [13] offers a mathematical framework for modelling serial decision-making problems. Specifically, our controller synchronization policy can be modelled as an MDP with the 3-tuple $(\mathcal{S}, \mathcal{A}, R)$ as follows.

- $\mathcal{S}$ is the finite state space. In particular, a state corresponds to the collection of the respective counts of time slots elapsed since the last broadcasts of up-to-date values of each BIS. The size of a state is $N$, which is the total number of BISes in the network.
- $\mathcal{A}$ is the finite action space. An action w.r.t. a state is defined as whether each up-to-date BIS should be broadcasted (indicated by 1) or not (indicated by 0), i.e., $\mathcal{A} \in \{1, 0\}^N$. As such, the size of an action is also $N$.
- $R$ represents the immediate reward associated with state-action pairs, denoted by $R(\mathbf{s}, \mathbf{a})$, where $\mathbf{s} \in \mathcal{S}$ and $\mathbf{a} \in \mathcal{A}$ are the state and action vectors. $R(\mathbf{s}, \mathbf{a})$ is calculated as the reductions in average request latency of all service requests in the network after taking action $\mathbf{a}$ in state $\mathbf{s}$.

With this MDP formulation, $\mathcal{S}$ and $R$ are collected by domain controllers from data planes through SDN's northbound interface [1] and are supplied to the central controller.

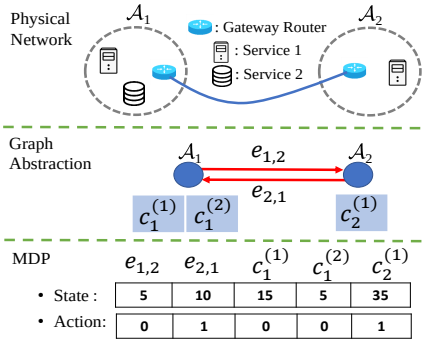



Fig. 1: An SDN network with 2 domains $\mathcal{A}_1$ and $\mathcal{A}_2$, where service 1 is available in both domains and service 2 is only available in $\mathcal{A}_1$. Two domains are connected by a pair of gateway routers. There are in total 5 BISes in the network.

Fig. 1 demonstrates the $(\mathcal{S}, \mathcal{A})$ pair in the formulated MDP in a simple example. The first entry in the state vector indicates that the last synchronization broadcast of the up-to-date value of gateway delay from $\mathcal{A}_1$ to $\mathcal{A}_2$ took place 5 time slots ago. An action consisting of binary entries indicates that the up-to-date value of gateway delay from $\mathcal{A}_2$ to $\mathcal{A}_1$ and the server delay of service 1 installed in domain $\mathcal{A}_2$ are to be broadcasted to all domain controllers at the current time slot.

The *optimal action* at each state is defined as the action that yields the maximum long-term reward, which reflects the fact that the synchronization decision at a time slot has lasting effects. In particular, the long-term reward is defined as the discounted sum of the expected immediate reward of all future state-action pairs from the current state. The reward for the state-action pair $\Delta t$ steps ahead of the current state is discounted by $\gamma^{\Delta t}$, where $\gamma$ is called the *discount factor* and $0 < \gamma < 1$. Here, $\gamma$ trades off the importance between the current and the future reward. Therefore, starting from an initial state $s_0$, the problem is formulated as finding a policy $\pi$ (i.e., the selection of a sequence of actions $\{\mathbf{a}_t\}_{t=0}^{T}$) such that the long-term accumulated reward expressed in the Bellman equation below is maximised:

$$V^{\pi}(\mathbf{s}_0) = \mathbb{E}_{\pi}\left[\sum_{t=1}^{T} \gamma^t R(\mathbf{s}_t, \mathbf{a}_t) | \mathbf{s}_0\right], \tag{1}$$

where $\mathbf{s}_t$ and $\mathbf{a}_t$ are the state-action pair at time $t$, and $T$ is the total time horizon of the problem.

***Time Scale of the MDP:*** The basic unit of time in the defined MDP include a series of events, which are jointly referred to as a *synchronization time slot* (*time slot* for short) in the sequence. Specifically, synchronization broadcastings take place at the start of a time slot. After all domain controllers finish broadcasting and receiving up-to-date BIS, they recalculate service paths for requests originated from their respective domains, based on the updated network views. Then, actual service latencies en route for new service paths are recorded for the calculation of $R$. For modelling tractability, we assume that newly broadcasted BIS values remain accurate until actual service latencies are recorded. This is true in practical SDN networks where periodical synchronization means that the collection of latest network information takes place at certain time. The collected information is considered to be "up-to-date" for a short while, as "real time" is a relative concept.

## IV. The MACS

To solve the formulated MDP, we use RL techniques to find the sequence of actions that maximises the Bellman equation in (1). For RL, imagine an agent who jumps from state to state in the formulated MDP by taking some actions associated with certain rewards. The agent's goal is to discover a sequence of state-action pairs, i.e., a *policy*, that maximises the accumulated time-discounted rewards. By interacting with the environment modelled by the MDP, the agent's experiences build up through "trial-and-error" where good decisions are positively enforced by positive rewards, and bad ones the opposite.

During training, the most important aspect is how the agent generalises and memorises what it has learned. Traditionally, the agent's estimations of future reward following state-action pairs are kept in tabular fashion. However, this approach soon becomes impractical in most RL tasks because of large state-action space sizes. Indeed, the state-action space is enormous in our controller synchronization problem. Consider a scenario with $N$ number of BIS and a time horizon of $T$ time slots, then there are as many as $T^N$ states and $2^N$ actions associated with each state. In light of this, function approximators have been proposed to approximate the $Q$-function, which represents the agent past experiences as it estimates the potential value of a given state-action pair (see Section IV-C). Among these approximators, DNN stands out due to its exceptional ability in capturing latent and complicated relationships from input data. Therefore, we also employ DNN in MACS to help the agent make sense of and generalise past experiences.

In this section, we first discuss design challenges and the MACS architecture in Section IV-A and Section IV-B, respectively. Then, mathematical details of how policies are estimated and the weights update process for the DNN in MACS are discussed in Section IV-C. Finally, we present the training algorithm for MACS in Section IV-E.

### A. Challenges

From our experiences leveraging several RL techniques to solve the formulated MDP, we identify some non-trivial challenges arising mainly from the two following aspects. First, the problem is a discrete control problem. This prevents the application of a number of well-established and relatively mature actor-critic approaches [14] based on the policy gradient theorem [15]. For example, DeepMind's recent work on the deep deterministic policy gradient (DDPG) agent [16] is the state-of-the-art for continuous control problems. Second, our formulated MDP has a high-dimensional state-action space. In particular, there are up to $2^N$ number of possible actions for any state. Thus, the size of the action space increases exponentially with the number of BISes in the network. It has been shown that such large action space is very difficult to explore and generalise from [16].

### B. The MACS Architecture

Considering the challenges discussed in the previous section, we need a DRL solution that is designed for discrete problems and can perform well in the presence of enormous state-action spaces. To this end, we build our learning architecture based on proposals in [17], [18], which have design features suitable for the nature of our formulated MDP.

DeepMind's dueling network architecture [17] explicitly separates the training for estimations of state-values and the advantages for individual actions, i.e., these values can be obtained separately. This is in contrast to most existing DRL architectures where the output of the DNN is conventionally a single value, i.e., the estimated $Q$-value for the input state-action pair. The dueling network architecture is particularly helpful in situations where there are many similar-valued actions.

The action branching architecture (ABA) [18] takes the dueling network a step further by categorising all actions as belonging to an *action dimension*. Moreover, a separate action advantage estimator is assigned to each action dimension (referred to as an *action arm* in the neural network) to estimate the advantage of all actions belonging to the action dimension (referred to as *sub-actions*). Under such arrangements, the $Q$-value estimation for each action is obtained by combining (1) the state value estimation, which is shared by all possible actions given the state; and (2) the sub-action advantage

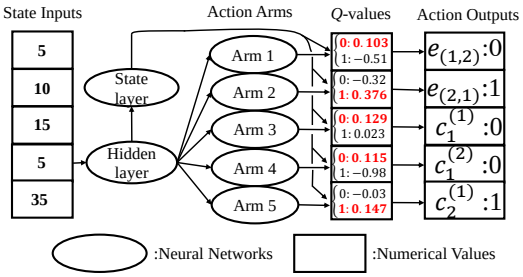



Fig. 2: The MACS network based on the example in Fig. 1. State inputs are first fed to the hidden layer, which is shared by the state layer and all action arms. The state layer is responsible for estimating the shared state value, whereas all action arms are responsible for estimating advantages of their 2 sub-actions.

estimated by the assigned action arm. A key characteristic of the action branching architecture is that a degree of freedom is given to each action dimension by dedicating a separate arm in the network for advantage estimations of sub-actions belonging to that action dimension. This design greatly improves learning efficiency and reduces complexity which arises from the combinatorial increase of the total number of action dimensions. Therefore, the design principles in the ABA are suitable for approximating the $Q$-function in our formulated MDP, as they address the challenges discussed in Section IV-A. In the following, we use the example in Fig. 1 to demonstrate how these design principles work in MACS.

The diagram of the DNN constructed for MACS shown in Fig. 2 is based on the example scenario in Fig. 1, where there are 5 BISes. In Fig. 2, there are 5 action arms, each corresponds to a BIS. Moreover, there are two sub-actions in every action dimension, i.e., 0: not to broadcast the corresponding up-to-date BIS value; and 1: to broadcast the corresponding up-to-date BIS value. An action arm outputs the estimation of action advantages of the two sub-actions under that action dimension. Furthermore, there is a separate state layer which outputs the estimation of the state value given the state inputs. Both the state layer and all semi-independent action arms are preceded by the input layer, which is designed for coordination. Based on the estimated $Q$-values, which are obtained by combining the estimated state value (output of the state layer) and action advantages (outputs of action arms), the sub-action with the highest $Q$-value is selected for each action arm. Finally, the outputs of all action arms concatenate and form the chosen action w.r.t. the state input.

### C. Design Details of MACS

*Q-function*, which originates from the classic *Q-learning* algorithm [19], is commonly used to estimate the quality (i.e. potential value) of a state-action pair:

$$\mathcal{S} \times \mathcal{A} \to \mathbb{R}.$$

In particular, the $Q$-function for a state-action pair following a policy $\pi$ is defined as:

$$Q^{\pi}(\mathbf{s}, \mathbf{a}) = \mathbb{E}_{\mathbf{s}'}\left[R(\mathbf{s}, \mathbf{a}) + \gamma \mathbb{E}_{\mathbf{a}' \sim \pi(\mathbf{s}')}[Q^{\pi}(\mathbf{s}', \mathbf{a}')]\right], \tag{2}$$

where $(\mathbf{s}', \mathbf{a}')$ is the state-action pair at the next step. $Q^{\pi}(\mathbf{s}, \mathbf{a})$ estimates the long-term value of a particular state-action pair following policy $\pi$; whereas the *state value*, denoted by $V^{\pi}(\mathbf{s})$, estimates the expected long-term value of the state $\mathbf{s}$:

$$V^{\pi}(\mathbf{s}) = \mathbb{E}_{\mathbf{a} \sim \pi(\mathbf{s})}[Q^{\pi}(\mathbf{s}, \mathbf{a})]. \tag{3}$$

Furthermore, to better distinguish the relative qualities of all possible actions under a given state $\mathbf{s}$, we define the *advantage* of an action $\mathbf{a}$ following policy $\pi$, denoted by $A^{\pi}(\mathbf{s}, \mathbf{a})$, as:

$$A^{\pi}(\mathbf{s}, \mathbf{a}) = Q^{\pi}(\mathbf{s}, \mathbf{a}) - V^{\pi}(\mathbf{s}). \tag{4}$$

W.r.t. the action branching architecture we employ, let $a_i \in \mathbf{\Omega}_i, i \in \{1, \ldots, N\}$ denote a sub-action belonging to action arm $i$, where $\mathbf{\Omega}_i$ is the set of all sub-actions of action arm $i$. Then, the $Q$-value and the action advantage of $a_i$ are denoted by $Q_i^{\pi}(\mathbf{s}, a_i)$ and $A_i^{\pi}(\mathbf{s}, a_i)$, respectively. A straightforward way to combine the state value and action advantages for $Q$-values is to follow (4), i.e., $Q_i^{\pi}(\mathbf{s}, a_i) = V^{\pi}(\mathbf{s}) + A_i^{\pi}(\mathbf{s}, a_i)$. However, as suggested by authors in [18], normalising the action advantage by the mean of action advantages before combining it with the shared state value yields better performance. Thus, the following aggregation method is used instead:

$$Q_i^{\pi}(\mathbf{s}, a_i) = V^{\pi}(\mathbf{s}) + \Big(A_i^{\pi}(\mathbf{s}, a_i) - \frac{1}{|\mathbf{\Omega}_i|} \sum_{a_i \in \mathbf{\Omega}_i} A_i^{\pi}(\mathbf{s}, a_i)\Big). \tag{5}$$

For action arm $i$, since we use DNN as the function approximator of its sub-action's $Q$-function $Q_i^{\pi}(\mathbf{s}, a_i)$, it is parameterised by the set of adjustable parameters $\boldsymbol{\theta}_i$ which are weights of the DNN. Then, the parameterised $Q$-function is denoted by $Q_i^{\pi}(\mathbf{s}, a_i; \boldsymbol{\theta}_i)$. The *value iteration update* of the $Q$-function uses the estimation of future rewards at the next state to update current $Q$-function, with the reasoning that estimations at the next state are more accurate, hence increasingly-more-accurate $Q_i(\mathbf{s}, a_i; \boldsymbol{\theta}_i)$ is eventually able to converge to the optimal policy $\pi^*$ after enough rounds of iterations. During weight update, $\boldsymbol{\theta}_i$ is adjusted to reduce the gap between current prediction (i.e., current $Q_i(\mathbf{s}, a_i; \boldsymbol{\theta}_i)$) and the next state estimate. Specifically, we define the target for arm $i$ as:

$$y_i = R(\mathbf{s}, \mathbf{a}) + \gamma \max_{a_i' \in \mathbf{\Omega}_i} Q_i(\mathbf{s}', a_i'; \boldsymbol{\theta}_i). \tag{6}$$

Then, the following loss function using the mean-squared error measurement is defined for adjusting $\boldsymbol{\theta}_i$:

$$L(\boldsymbol{\theta}_i) = \mathbb{E}[\big(y_i - Q_i(\mathbf{s}, a_i; \boldsymbol{\theta}_i)\big)^2]. \tag{7}$$

Before the weight update process takes place, the total loss is calculated as the the mean across all arms:

$$L(\boldsymbol{\theta}) = \mathbb{E}\left[\frac{1}{N} \sum_{i=1}^{N} L(\boldsymbol{\theta}_i)\right]. \tag{8}$$

Then, by differentiating $L(\boldsymbol{\theta})$ w.r.t. $\boldsymbol{\theta}$, weights of the DNN are updated for the next iteration:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}), \tag{9}$$

where $\alpha$ is the learning rate.

Since the gradient descent updates of $\boldsymbol{\theta}$ is different from canonical supervised learnings because the training target $y_i = R(\mathbf{s}, \mathbf{a}) + \gamma \max_{a_i' \in \mathbf{\Omega}_i} Q_i(\mathbf{s}', a_i'; \boldsymbol{\theta}_i)$ is generated by the same $Q$-function $Q_i(s, a_i; \boldsymbol{\theta}_i)$ that is being trained. Therefore, to improve stability and performance of the training process, we improve the training algorithm in the following three ways.

***1)***. We maintain a delayed version of the $Q$-function, $Q_i(\mathbf{s}', a_i'; \boldsymbol{\theta}_i')$, for the estimation of the maximum next state reward, which was proposed [20] to improve the stability of their DQN. As such, the target function in (6) is updated to

$$y_i = R(\mathbf{s}, \mathbf{a}) + \gamma \max_{a_i' \in \mathbf{\Omega}_i} Q_i(\mathbf{s}', a_i'; \boldsymbol{\theta}_i'). \tag{10}$$

The delayed $Q$-function is updated with the newest weights every $C$ ("target sync gaps") steps by setting $\boldsymbol{\theta}_i' \leftarrow \boldsymbol{\theta}_i$.

***2)***. To overcome the overestimation of action values, we implement *Double Q-learning* [21] to address the positive bias in estimation introduced when the maximum expected action values are instead approximated by the maximum action values in $Q$-learning. Specifically, we use the up-to-date $Q$-function $Q_i(\mathbf{s}, a_i; \boldsymbol{\theta}_i)$ to determine the optimal sub-actions, i.e.,

$$a_i'^{*} = \arg \max_{a_i' \in \mathbf{\Omega}_i} Q_i(\mathbf{s}', a_i'; \boldsymbol{\theta}_i). \tag{11}$$

The accumulated reward of the returned action $a_i'^*$ is estimated by the delayed $Q$-function using (10). Therefore, the target in (6) is further improved to be:

$$y_i = R(\mathbf{s}, \mathbf{a}) + \gamma Q_i(\mathbf{s}', \arg\max_{a_i' \in \mathbf{\Omega}_i} Q_i(\mathbf{s}', a_i'; \boldsymbol{\theta}_i); \boldsymbol{\theta}_i'). \quad (12)$$

***3)***. We implement the "replay memory" [22] in which some of the agent's past experiences in $(\mathbf{s}, \mathbf{a}, r, \mathbf{s}')$ tuples are stored and might be used more than once for training. In particular, a matrix $\mathcal{D}$ is created that can store up to $K$ $(\mathbf{s}, \mathbf{a}, r, \mathbf{s}')$ tuples. At each training iteration where $Q$-learning update takes place, samples of experiences are pulled randomly from $\mathcal{D}$ for training. Matrix $\mathcal{D}$ is updated on a first-in-first-out basis.

### *D. Details of the DNN*

The structure of MACS is demonstrated in the example in Fig. 2, which can be categorised as a *Multilayer Perceptron (MLP)*. The input to the MLP is of dimension $N$, which corresponds to a state of the formulated MDP (see Section III-E). The input layer, which precedes and is fully connected to all action arms and the state layer, consists of 2 hidden fully connected layers with 512 and 256 neurons, respectively. Every action arm contains a hidden layer of 128 neurons which is followed by an output layer with 2 neurons that output the advantage estimations for 2 sub-actions, respectively. The state layer has a hidden layer of 128 neurons followed by a single neuron in its output layer, which gives the estimation of state value. Overall, the output of the MLP is a vector of $Q$-values whose dimension is $2N$. Note that the state value (output of the state layer) and the advantages of all sub-actions (outputs of the all action arms) are combined according to (5). When the trained MLP is used for making action predictions, the chosen sub-action for each action arm is decided by comparing $Q$-values of its two sub-actions, whichever is greater gets picked, as demonstrated in Fig. 2. In cases where the number of arms giving "1" output is larger than the given synchronization budget, those sub-actions with the greater $Q$-values get picked first until the budget is reached.

The MLP is realised using Keras model with TensorFlow, in which Adam is chosen as the optimiser with initial learning rate of $0.0001$ and Rectified Linear Unit (ReLU) is employed as activation functions for all neurons except for the outputs. The discount factor is set to $\gamma = 0.99$; while the target network is updated every 20 steps (i.e., $C = 20$).

### *E. The Training Algorithm*

The training of MACS takes place in a semi-online fashion in the sense that new $(\mathbf{s}, \mathbf{a}, r, \mathbf{s}')$ tuples are gradually stored in matrix $\mathcal{D}$ (i.e., the agent's "reply memory"), while past $(\mathbf{s}, \mathbf{a}, r, \mathbf{s}')$ tuples stored are also used for training. As time proceeds, more $(\mathbf{s}, \mathbf{a}, r, \mathbf{s}')$ tuples are generated and recorded in matrix $\mathcal{D}$ for training. Since we limit the number of entries of $\mathcal{D}$ to $K$, older tuples are gradually replaced by new entries.

In order to make sure that the agent can keep discovering new (potentially better) actions while exploits what it had already learned, i.e., to balance *exploration* and *exploitation*, we employ the $\epsilon$-greedy algorithm for which the agent selects random actions or selects actions based on prediction with probabilities $0.20$ and $0.80$, respectively (i.e., $\epsilon = 0.2$).

So far, the design of immediate reward only takes into account the average reduction in service latency after synchronization broadcastings. Since most broadcastings of up-to-date BIS bring positive rewards (there are cases where broadcasting some up-to-date BISes leads to worse performance, because the controllers do not have the *fully up-to-date* network view; nevertheless, synchronizations improve performance overall), this makes the agent think that it is always better to broadcast as many as possible. However, as we have the synchronization budget constraint, which is intended to curb the overheads brought by excessive number of synchronization messages, we need to make this realisable to the agent during training. To this end, we offset the immediate reward defined in the MDP by a small value for the sub-actions where up-to-date BISes are broadcasted (i.e., sub-actions indicated by 1, referred to as *positive sub-actions*). In particular, let $\iota$ denote the unit offset for each positive sub-action, and $\rho$ be the number of positive sub-actions for the state-action pair $(\mathbf{s}, \mathbf{a})$; then, the 4-tuple stored is $(\mathbf{s}, \mathbf{a}, R(\mathbf{s}, \mathbf{a}) - \rho\iota, \mathbf{s}')$.The training process is summarised in Algorithm 1.

**Algorithm 1:** Training algorithm for MACS

**input :** MLP model settings; distributed SDN settings; simulation program for generating rewards.
**output:** Trained parameterized $Q$-functions for all arms.

Initialize $Q$-functions $Q_i(\mathbf{s}, a_i; \boldsymbol{\theta}_i), i \in \{1, \ldots, N\}$ by instantiating the MLP; Set initial state $s_0$; $t = 0$;
Initialize matrix $\mathcal{D}$ with past $(\mathbf{s}, \mathbf{a}, R(\mathbf{s}, \mathbf{a}) - \rho\iota, \mathbf{s}')$ tuples;
Initialize the delayed $Q$-functions $\boldsymbol{\theta}_i' \leftarrow \boldsymbol{\theta}_i, i \in \{1, \ldots, N\}$.
**while** $t \leq T$ **do**
  **foreach** *time instant* $t$ **do**
    `indicator` $= 0, 1$ with probabilities $1 - \epsilon, \epsilon$, respectively;
    **if** `indicator` $= 0$ **then**
      Select an action $a_t$ randomly;
    **else**
      Select an action $a_t$ according to Section IV-D;
    **end**
    Pass on the $(\mathbf{s}_t, \mathbf{a}_t)$ to the simulation program to get return $r_t$ and $\mathbf{s}_{t+1}$ ;
    Store $(\mathbf{s}_t, \mathbf{a}_t, r_t - \rho\iota, \mathbf{s}_{t+1})$ in $\mathcal{D}$;
    Pull minibatch $\mathcal{D}_t$ of $(\mathbf{s}_i, \mathbf{a}_i, r_i - \rho\iota, \mathbf{s}_{i+1})$ from $\mathcal{D}$;
    **foreach** $(\mathbf{s}_i, \mathbf{a}_i, r_i - \rho\iota, \mathbf{s}_{i+1})$ *in* $\mathcal{D}_t$ **do**
      **foreach** $g \in \{1, \ldots, N\}$ **do**
        Calculate $L(\boldsymbol{\theta}_g)$ from $a_g$ and $y_g$ using (7), (11), and (12), respectively;
      **end**
      Calculate aggregrated loss $L(\boldsymbol{\theta})$ according to (8);
      Update weights: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$;
    **end**
    **if** $t \mod C = 0$ *(`C: target sync gap`)* **then**
      **foreach** $i \in \{1, \ldots, N\}$ **do**
        Update weights: $\boldsymbol{\theta}_i' \leftarrow \boldsymbol{\theta}_i$.
      **end**
    **end**
  **end**
**end**

## V. Evaluation

### *A. Evaluation Scenarios and Performance Benchmarks*

Two scenarios are considered for evaluations, where the first scenario (*Scenario 1* in he sequence) is an ideal, purely random case in the sense that the patterns of network dynamicity, service distributions, and service request arrivals are uniform in all domains; whereas the second scenario (*Scenario 2* in the sequence) is a more realistic case in that (1) the values of BIS change more frequently with some servers/gateway routers than others; (2) services are more likely to be installed in some domains; (3) the requests for services follow Zipf-Mandelbrot distributions. See Section V-B for details of the settings used in two scenarios. In the following, we briefly discuss the benchmarks for evaluating the performance of MACS.

***The Full/Worst Controller Synchronization Levels*** correspond to the case where all up-to-date BISes are broadcast to domain controllers at every time slot; and the case where there is no broadcasting of any up-to-date BIS at any time slot, respectively. We can see that the full synchronization level is identical to having one logical central

TABLE I: Evaluation Parameters

| Parameter | Scenario 1 | Scenario 2 |
|---|---|---|
| Probabilities that BISes change value | Uniform | The probability of BIS $i$ changing value is proportional to $\frac{1}{\sigma\sqrt{2\pi}}e^{-(x-\mu)^2/2\sigma^2}, (\mu = 30, \sigma = 10)$ |
| Probabilities that service installations present in domains | Uniform | 30% probability in any 4 domains and 70% probability in the other 4 domains |
| Service request pattern | Uniform | The probability that service $i$ is requested is proportional to $(q+i)^{-\beta} (q = 5, \beta = 0.8)$ |
| The number of domains | 8 domains | |
| The number of services | 10 unique services, each installed twice in 2 different domains | |
| BIS value distribution | $\{1, 2, 4, 6, 8, 13, 17, 20, 25, 30\}$ uniformly distributed | |

controller and that it incurs the maximum synchronization overheads. Note that these two cases serve as the lower and upper bounds of our performance metric and they are not subject to the given synchronization budgets.

***The Greedy (MinMax) Algorithm*** is a simple controller synchronization scheme that aims at reducing the staleness of controller-perceived BIS values by minimising the maximum state value of the defined MDP. Specifically, with the given synchronization budget at a time slot, the up-to-date values of those BISes that have not been synchronized to all domain controllers the longest get broadcasted first.

***The Anti-entropy [23] Algorithm,*** which is implemented in the ONOS controller [24], is based on a simple gossip algorithm that controllers randomly synchronize with each other [23]. W.r.t. the definition of controller synchronization in our problem (Definition 2) and the given synchronization budget, controller synchronization with the anti-entropy algorithm is carried out such that up-to-date values of BIS are randomly selected for synchronization broadcastings.

### B. Network Settings

Other than the differences between the two scenarios discussed in Section V-A, the domain-wise topologies, i.e., how domains are connected to each other, are constructed according to the same dataset. Specifically, there are 8 domains that are connected according to the *degree distribution* extracted from the "CAIDA AS-27524" dataset [25], which refers to the distribution of the number of neighbouring domains of an arbitrary domain. 10 unique services are considered in both scenarios, with two installations in two different domains for each service. Initial and new (if any changes) BISes are randomly drawn from the set $\{1, 2, 4, 6, 8, 13, 17, 20, 25, 30\}$.

The number of BIS changes (network dynamicity) and the synchronization budget at any time slot are modelled by Poisson processes with mean $\lambda = 3$ for both evaluation scenarios. However, network dynamicity in Scenario 1 is uniform in the sense that all BISes change value with the same probability; whereas the probabilities of all BIS changes jointly follow Gaussian distribution in Scenario 2, which means some BIS change more frequently than others. As for service installations patterns, Scenario 1 assumes that a service installation has the same probability being in any domain. In comparison, for Scenario 2, domains are divided randomly and equally into two groups, service installations are more likely to be inside the first group (with 70% probability) than the second one (with 30% probability). Moreover, service request patterns in domains are uniform in Scenario 1, i.e., users request different services at the same rate. In Scenario 2, however, service request patterns follow Zipf-Mandelbrot distribution [26], which is widely used to model the content popularity in content delivery networks (CDN). Specifically, the popularity of $i$-th most popular service is proportional to $(q+i)^{-\beta}$, where $q = 5$ and $\beta = 0.8$. The evaluation settings are summarised in Table I.

***Remark:*** It should be noted that the network settings described are only for the sake of evaluations, there are no assumptions on any of the parameters used above.

### C. Evaluation Results

In order to compare the performance of MACS under two evaluation scenarios, we compare their respective average request delays to the corresponding full synchronization levels in Fig. 5. In particular, we calculate and plot the percentage increase of average request latency in each time slot, from the full synchronization cases to MACS for the two scenarios.

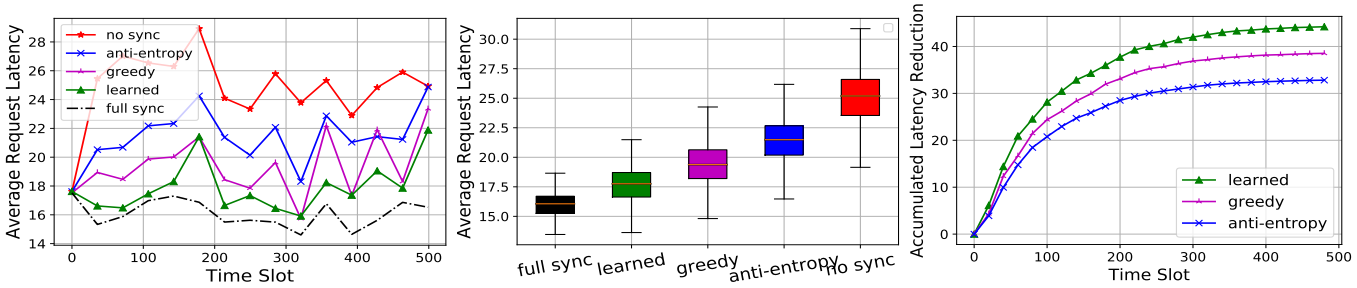
*1) Superiority of MACS over time:* From Fig. 3(a) and Fig. 4(a), we can see that the gaps between "full sync" and "no sync" curves clearly demonstrate the important role controller synchronization plays in improving performance. Among the three synchronization algorithms implemented, MACS consistently performs the best in both scenarios. Surprisingly, after 300 time slots in Scenario 2, MACS, which runs on a limited synchronization budget, can almost achieve full sync performance, as can be seen in Fig. 4(a) where two curves overlap. The superiority of MACS over the other two schemes is also evident in sub-figures (b), which compare the statistic properties of the request latency results of different synchronization regimes. In addition, these results reveal that while the goal is to minimise long-term average request latency, this objective is achieved by consistently minimising average request latency at each time slot.

*2) Superiority of MACS for maximising the Bellman equation:* Recall that our objective is to maximise the accumulated reductions in request latency over a period of time, i.e., to maximise $V^{\pi}(\mathbf{s}_0) = \mathbb{E}_{\pi}\left[\sum_{t=1}^{T}\gamma^t R(\mathbf{s}_t, \mathbf{a}_t)|\mathbf{s}_0\right]$. The evaluation results in Fig.3(c) and Fig.4(c) confirm the superiority of MACS in achieving this goal. In particular, during the testing period of 500 time slots, MACS outperforms the greedy algorithm by approximately 15%, 30%; the anti-entropy by 36%, 56%, for two scenarios, respectively.

*3) MACS performs better in Scenario 2 over Scenario 1:* The evaluation results in Fig. 3 – Fig. 4 clearly demonstrate that while MACS are better than the other two algorithms in all evaluation results, MACS in Scenario 2 outperform itself in Scenario 1, which is also confirmed by the explicit comparison shown in Fig. 5. In particular, Fig. 5 shows that after about 250 time slots in Scenario 2, although MACS operates on a limited synchronization budget, its synchronization policy consistently results in performance close to the full sync case, where all controllers are always mutually synchronized at the cost of incurring the maximum synchronization overheads. This phenomenon can be explained by the fact that more patterns exist in network dynamicity, service distribution, and user demands under Scenario 2; whereas these settings are purely random in Scenario 1. Due to the DNN's exceptional ability to capture complicated and latent relationships in input data, MACS could learn these pattern after some time and eventually manages to make best use of the limited synchronization budget to reach near optimal performance.

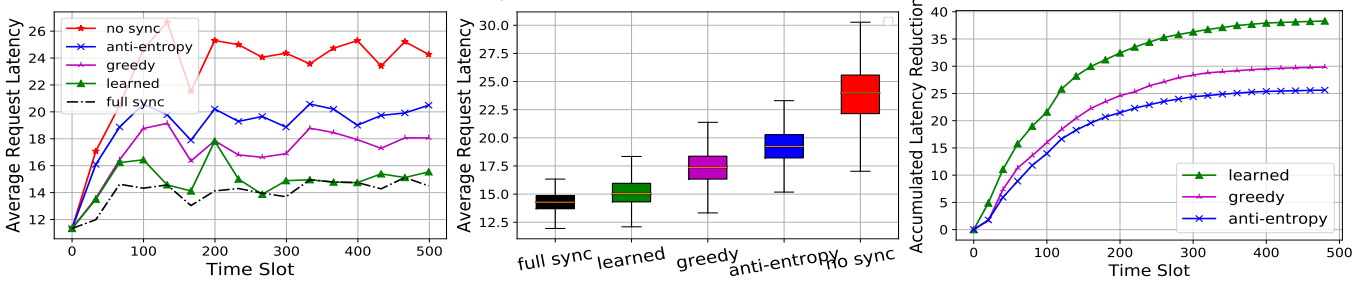
## VI. Conclusion

In this paper, we investigated the controller synchronization problem with limited synchronization budget in distributed SDN, for which our aim was to find the policy that maximises the performance enhancements brought by controller synchronizations over a period of time. We formulated the controller synchronization problem as an MDP which has a large state-action space. We identified challenges in solving the formulated MDP and designed a DRL-based algorithm, called MACS, which absorbs various DNN design principles to tackle these challenges. Due to the DNN's ability in learning the network




(a) Average request latency at every time slot. (b) Box plot of average request latencies. (c) Accu' average request latency reductions.

Fig. 3: Evaluation results for Scenario 1.




(a) Average request latency at every time slot. (b) Box plot of average request latencies. (c) Accu' average request latency reductions.

Fig. 4: Evaluation results for Scenario 2.

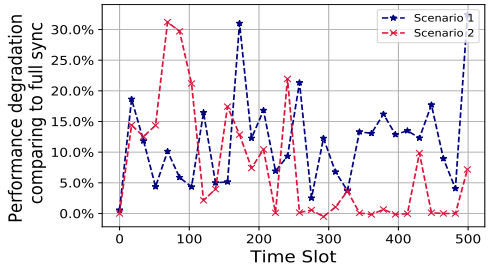



Fig. 5: Performance comparison of MACS in two scenarios.

dynamicity patterns which results in near optimal use of the given limited synchronization budget, evaluation results showed that MACS consistently outperforms state-of-the-art SDN controller synchronization algorithms/heuristics operating under the same synchonization budget by large margins.

## References


[1] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[2] F. Bannour, S. Souihi, and A. Mellouk, "Distributed SDN control: Survey, taxonomy, and challenges," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 333–354, 2018.

[3] A. S. Muqaddas, P. Giaccone, A. Bianco, and 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.

[4] Q. Qin, K. Poularakis, G. Iosifidis, and L. Tassiulas, "Sdn controller placement at the edge: Optimizing delay and overheads," in *IEEE INFOCOM 2018*.

[5] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker, "CAP for networks," in *ACM HotSDN*, 2013.

[6] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized?: state distribution trade-offs in software defined networks," in *ACM HotSDN*, 2012.

[7] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *ACM HotSDN*, 2013.

[8] K.-T. Förster, R. Mahajan, and R. Wattenhofer, "Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes," in *IFIP Networking Conference (IFIP Networking) and Workshops*, 2016.

[9] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," *ACM SIGCOMM Computer Communication Review*, vol. 42, pp. 323–334, 2012.

[10] K.-T. Foerster, S. Schmid, and S. Vissicchio, "Survey of consistent software-defined network updates," *IEEE Communications Surveys & Tutorials*, 2018.

[11] P. Amaral, J. Dinis, P. Pinto, L. Bernardo, J. Tavares, and H. S. Mamede, "Machine learning in software defined networks: Data collection and traffic classification," in *IEEE 24th International Conference on Network Protocols (ICNP)*, 2016.

[12] S. M. Das, H. Pucha, and Y. C. Hu, "Mitigating the gateway bottleneck via transparent cooperative caching in wireless mesh networks," *Ad Hoc Networks*, vol. 5, no. 6, pp. 680–703, 2007.

[13] C. White, *Markov decision processes*. Springer, 2001.

[14] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *NIPS 2000*.

[15] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *NIPS 2000*.

[16] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[17] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.

[18] A. Tavakoli, F. Pardo, and P. Kormushev, "Action branching architectures for deep reinforcement learning," in *AAAI 2018*.

[19] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[20] 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, p. 529, 2015.

[21] H. V. Hasselt, "Double Q-learning," in *NIPS*, 2010.

[22] L.-J. Lin, "Reinforcement learning for robots using neural networks," Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, Tech. Rep., 1993.

[23] A. S. Muqaddas, A. Bianco, P. Giaccone, and G. Maier, "Inter-controller traffic in ONOS clusters for SDN networks," in *IEEE ICC*, 2016.

[24] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "ONOS: Towards an open, distributed SDN OS," in *ACM HotSDN*, 2014.

[25] "Stanford network analysis platform (SNAP)," Standford University, 2017. [Online]. Available: https://snap.stanford.edu/snap/

[26] S. Borst, V. Gupta, and A. Walid, "Distributed caching algorithms for content distribution networks," in *IEEE INFOCOM*, 2010.