Dynamic SDN Control Plane Request Assignment in NFV Datacenters

Junxiao Wang, Heng Qi, Member, IEEE, Wenxin Li, Keqiu Li, Senior Member, IEEE, Steve Uhlig, and Yuxin Wang

***Abstract*—Owing to powerful programmability, software defined networking (SDN) is well aligned with the requirements of NFV datacenters. Although the mainstream SDN frameworks can effectively respond to the requests of control plane, the load distribution among controllers is not balanced due to the request dynamics and request diversity. The existing solutions balance the load of control plane by migrating switches between controllers. However, these solutions are based on the binding between switches and controllers, so they are difficult to adapt to the dynamic and diverse requests. In this paper, we propose a new framework to decouple the binding. The new framework performs modular management for request queues. A complete request queue is provided for each type of request between each switch controller pair, so that the assignment between requests is independent of each other. Based on the proposed framework, we transform the request assignment problem into a variant of the scheduling problem in a Stochastic Processing Network (SPN), and propose a Maximum Pressure Policy (MPP) which can provide runtime guarantees on request throughput and response latency. To fit with the constraints inherent to large-scale deployment, we propose a distributed version of MPP, named DMPP. DMPP runs in local state on each switch and performs scheduling logic for batch requests. We implement a protosystem of our solution and evaluate it on the settings representing real-world scenarios. The results show that our solution can provide guarantees on request throughput and response latency, and significantly outperforms state-of-the-art solutions through a more efficient resource usage.**

***Index Terms*—Software defined Networking, Datacenter, Control Plane, Stochastic Processing Networks.**

I. Introduction

Software-defined networking (SDN) [1] has rapidly grown in the datacenters of network function virtualization (NFV), in particular with the deployment of service function chains [2], [3], [4], [5]. According to the IETF specifications [6], a typical SDN based service function chain architecture consists of the components grouped into two planes, the control plane and data plane. The SDN control plane is built in software as applications running on top of multiple controllers, whereas general purpose switches constitute the SDN data plane. The control plane is responsible for adjusting the service function paths as a result of their status requests (i.e., overloaded, active, inactive, failed, etc.). Both the planes communicate with each other via a standardized protocol such as OpenFlow [7]. Architecturally seen, SDN decouples the control plane from data plane and enables the traffic steering across service functions in a dynamic and flexible manner [8], [9], [10].

Although the requests to the control plane can be responded via some up-to-date SDN frameworks such as ONOS [11] and OpenDaylight [12], the load distribution among controllers unfortunately appears imbalance, due to the following factors.

Request dynamics: Spatially, switches in the different layers of topology experience significantly different flow arrival rates [13]. Temporally, the aggregate traffic usually peaks in daytime and falls at night [14]. Traffic variability also exists in shorter time scales even the aggregate traffic remains the same [15]. For these reasons, the arrival rate of requests is dynamic. However, the existing SDN frameworks assign requests by statically assigning controllers to switches. These frameworks make each controller handle requests from the same amount of switches. Consequently, the controllers differ in the amount of received requests per unit time and some of them inevitably become hot spots.

Request diversity: There are many types of requests in the control plane [16]. For example, some requests involve flow status, while others involve port status. Different request types result in different processing time on the controller. Even if the same type of requests belong to different service chains, the processing time on the controller is different [5]. For these reasons, the processing time of requests is diverse. However, the existing SDN frameworks are based on the binding between switches and controllers, so that the assignment between requests is dependent of each other, which further incurs load imbalance among controllers.

The load imbalance will eventually reduce the performance of control plane, such as the throughput of request and the latency of response [17], [18].

The existing solutions balance the load of control plane by migrating switches between controllers [16], [18], [19]. Firstly, these solutions cannot adapt to the dynamic arrival rate of requests because they rely on the heavy message exchanges, such as 6 round-trip message exchanges per migration in [17]. They must make the migration frequency low enough to limit overhead, so they cannot cope with real-time changes in request arrival rate. Secondly, these solutions cannot adapt to the diverse processing time of requests because they are based on the binding between switches and controllers. They must assign all requests of the switch to one controller, and part of requests are therefore forced to inappropriate controllers.

Junxiao Wang, Heng Qi, Keqiu Li, and Yuxin Wang are with the School of Computer Science and Technology, Dalian University of Technology, P.R. China. Wenxin Li is with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. Steve Uhlig is with the School of Electronic Engineering and Computer Science, Queen Mary University of London, UK. Heng Qi is the corresponding author (hengqi@dlut.edu.cn).

In this paper, we propose a new framework to decouple the binding. The new framework performs modular management for request queues. A complete request queue is provided for each type of request between each switch controller pair, so that the assignment between requests is independent of each other. Based on the proposed framework, we can achieve dynamic request assignment by selecting an appropriate request queue for the requests arriving in real-time.

Then, we study an online decision-making problem, that is, how to select an appropriate request queue for requests arriving in real time. We show that this problem is a variant of the scheduling problem in a stochastic processing network (SPN) [20] which has similar properties. By transforming this problem into an SPN one, we apply the results of the SPN problem and propose the maximum pressure policy (MPP), which shows the asymptotically optimal request throughput. MPP is also asymptotically optimal for minimizing a cost function of buffer occupancy levels, hence MPP provides approximate guarantees on response latency. The time complexity of MPP is bounded by a linear term on the number of switches and makes MPP applicable in NFV datacenters.

Despite those benefits, MPP still faces practical constraints inherent to large-scale deployment. MPP is a strategy that executes a scheduling for each request, and it needs the global state information to make decisions. On the one hand, with the expansion of the network scale, the cost of obtaining network-wide state information is increasing. On the other hand, even if the logic of each scheduling is very lightweight, scheduling for each request still brings excessive runtime overhead to the switch, which will cause the switch to overload. To deal with these constraints, we propose the distributed maximum pressure policy (DMPP). DMPP runs in local state on each switch and performs scheduling logic for batch requests.

Based on the settings representing real-world scenarios, we evaluate the performance of our solution, using a protosystem we implement as testbed, whose source code is available at *https://github.com/wangjunxiao/AgileScheduler*. Experimental results show that the performance of DMPP is close to that of MPP, and DMPP requires much less resources when assigning requests, so the resource utilization of DMPP is higher. In terms of request throughput, response latency, etc., compared with state-of-the-art solutions based on switch migration like [16], [18], the performance of our solution is significantly better than them.

In summary, we make the following contributions:

1) We design a new framework of request assignment that is performed between SDN controllers and switches in NFV datacenters. Compared with the existing frameworks based on switch migration, the new framework can better adapt to the request dynamics and request diversity.
2) Based on the new framework, we transform the request assignment problem into a variant of the scheduling problem in a Stochastic Processing Network (SPN), and propose a Maximum Pressure Policy (MPP) which can provide runtime guarantees on request throughput and response latency. To fit with the constraints inherent to large-scale deployment, we propose a distributed version of MPP, named DMPP. DMPP runs in local state on each switch and performs scheduling logic for batch requests.
3) We implement a protosystem of our solution and evaluate it on the settings representing real-world scenarios. The results show that our solution can provide guarantees on request throughput and response latency, and significantly outperforms state-of-the-art solutions through a more efficient resource usage.

The remainder of this paper is organized as follows. Section II introduces the history of request assignment in the control plane and the new framework we propose. In Section III, we model the request assignment problem, and show that the problem is a variant of the scheduling problem in a stochastic processing network. In Section IV, we introduce the maximum pressure policy, and show its runtime guarantees on request throughput and response latency. In Section V, we explore practical constraints inherent to large-scale deployment, and introduce the distributed maximum pressure policy. In Section VI, we evaluate and analyze our solution. Section VII summarizes related work, and we conclude the paper in Section VIII.

II. Framework of Request Assignment

A. Background

The emergence of SDN architecture decouples the data plane and control plane of the underlying network. However, with the expansion of datacenter scale and the increase of traffic, researchers have to rethink the scalability of SDN architecture, especially the scalability of SDN control plane [22], [23], [24]. If we look back at the history of SDN control plane scalability, we can find that this history is divided into three stages.

In the first stage, all control plane requests are processed by a single controller. At this time, the concept of request assignment has not yet appeared. Since the computing resources of this single controller such as CPU and memory are limited, and as the scale of the data plane expands, this single controller will have to handle more requests, which will lead to performance bottlenecks. NOX [21], as the representative framework of this stage, can serve only 30K flow requests per second with a response time less than 10 ms. Although network operators continue to improve the performance of a single controller, it is still not enough to meet the increasing demand.

In the second stage, control plane requests can be handled by multiple controllers. The advantages of distributed controllers include load distribution and avoiding single controller failures. The concept of request assignment has appeared, however, the request assignment at this time is static. Take the mainstream framework ONOS [11] at this stage as an example, it statically lets each controller manage the same number of switches, ignoring that the request arrival rate is dynamically changing in space and time dimensions. Due to unbalanced load among controllers, this static request assignment leads to very poor request throughput and response latency.

In the third stage, control plane requests can not only be handled by multiple controllers, but can also be dynamically

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2020.3048188, IEEE Transactions on Network Science and Engineering

historical progression	framework	multiple controllers	dynamic assignment	assignment granularity	request diversity
stage one	NOX [21]	×	×	switch level	×
stage two	ONOS [11]	✓	×	switch level	×
stage three	ElastiCon [17]	✓	✓	switch level	×
stage four	our framework	✓	✓	request level	✓

TABLE I
THE DIFFERENCES BETWEEN OUR FRAMEWORK AND EXISTING FRAMEWORKS.

assigned among controllers. Dixit et al. [17] first design a switch migration protocol to make dynamic request assignment technically feasible. Then, based on the protocol, many solutions for dynamic request assignment such as [16], [18], [19] are come up with. However, these solutions are coarse-grained. They rely on heavy information exchanges, so they must make the frequency of migration slow enough to limit overhead. They are based on the binding between switches and controllers, causing the assignment between requests to depend on each other. Due to these reasons, they therefore fail to use the resources of control plane effectively.

B. Motivation

Throughout history, the existing frameworks mainly assign requests based on the granularity of switches, which makes request assignment inefficient. Bearing above point in mind, we propose a new framework, *the framework is no longer based on the granularity of switches, but based on the granularity of requests*. In this new framework, we propose to decouple the binding between switches and controllers, so that the assignment between requests is independent of each other. In other words, every request on a switch will be handled by a suitable but not necessarily the same controller. We compare the differences between our framework and the existing frameworks in Table I.

We are motivated to design such a new framework to assign the requests of control plane. We also expect the new framework can bring the historical progression of request assignment to a next stage, where the assignment is no longer based on the granularity of switches but rather based on the more efficient granularity of requests. We overall design the new framework keeping in mind three goals below:

1) Flexible: The diverse requests to the control plane should be assigned in a flexible manner to decouple the binding between switches and controllers. Even for the same switch, assignment between requests should be independent of each other. The request assignment with flexibility will achieve a better resource usage.
2) Agile: The process of request assignment should be agile to quickly react to the dynamic request arrival. Facing with request arrival rates that may be sudden and skewed on space and time scales, the request assignment with agility will be more likely to make the ideal decision.
3) Scalable: Request assignment should adapt to large-scale deployment. The logic and overhead of each assignment should be light enough to accommodate the growth of the deployment scale.

C. Modular Request Queue Management

The overview of our request assignment framework is illustrated in Fig. 1. The framework manages the request assignment process modularly through the middleware composed of `SBSwitch` module and `NBSwitch` module. When the switch needs to upload a request to the control plane, the framework will first send the request to the `SBSwitch` module, then the `SBSwitch` module will send the request to the corresponding `NBSwitch` module, and finally the `NBSwitch` module will send the request to the corresponding controller for processing. Being both the ingress and egress of requests, `SBSwitch` module and `NBSwitch` module are responsible not only for uploading requests to controllers, but also for returning responses back to switches. Each switch uses one `SBSwitch` module and multiple `NBSwitch` modules to assign requests to controllers.

1) South Bound Switch module: i.e., `SBSwitch` module, attached to physical switches by one-to-one mapping. Thus, the total amount of `SBSwitch` modules depends on how many switches the data plane contains. For every `SBSwitch` module, the amount of request queues it has depends on how many services the control plane can provide with. Upon a receipt of a request, `SBSwitch` module categorizes the request according to its service type, pushes it into correspondent request queue (of this `SBSwitch` module), and then decides which controller to send it to.

2) North Bound Switch module: i.e., `NBSwitch` module, attached to controllers by n-to-one mapping. n is equal to the number of switches in the data plane. Each `NBSwitch` module is attached to a controller. For every `NBSwitch` module, the amount of request queues it has depends on how many services the attached controller can provide with. According to the predefined mapping between the two layers of request queues, the request from the request queue of the `SBSwitch` module will be sent to the corresponding request queue of the `NBSwitch` module.

To clarify the mapping between the request queues, we take the case in Fig. 1 as an example to illustrate the process of sending requests from the switch to the controller. The example has three `SBSwitch` modules and nine `NBSwitch` modules totally. For every `SBSwitch` module, it has three request queues since there are three services in the control plane. Each switch uses one `SBSwitch` module and three `NBSwitch` modules to assign requests to controllers. For instance, `SBSwitch1`, attached to physical `Switch1`, can send requests to anyone within `NBSwitch1`, `NBSwitch4` and `NBSwitch7` through `Queue1`, `Queue2` and `Queue3`. For every `NBSwitch` module, it has two request queues since the attached controller provides with two services. Hence, as depicted in Fig. 1 with dotted arrow, requests of `Service1` from `SBSwitch1` can be sent to `NBSwitch1 Queue1` or `NBSwitch4 Queue2` via `SBSwitch1 Queue1`.

In general, the framework allows multiple controllers to process requests from the same switch, so as to decouple the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2020.3048188, IEEE Transactions on Network Science and Engineering

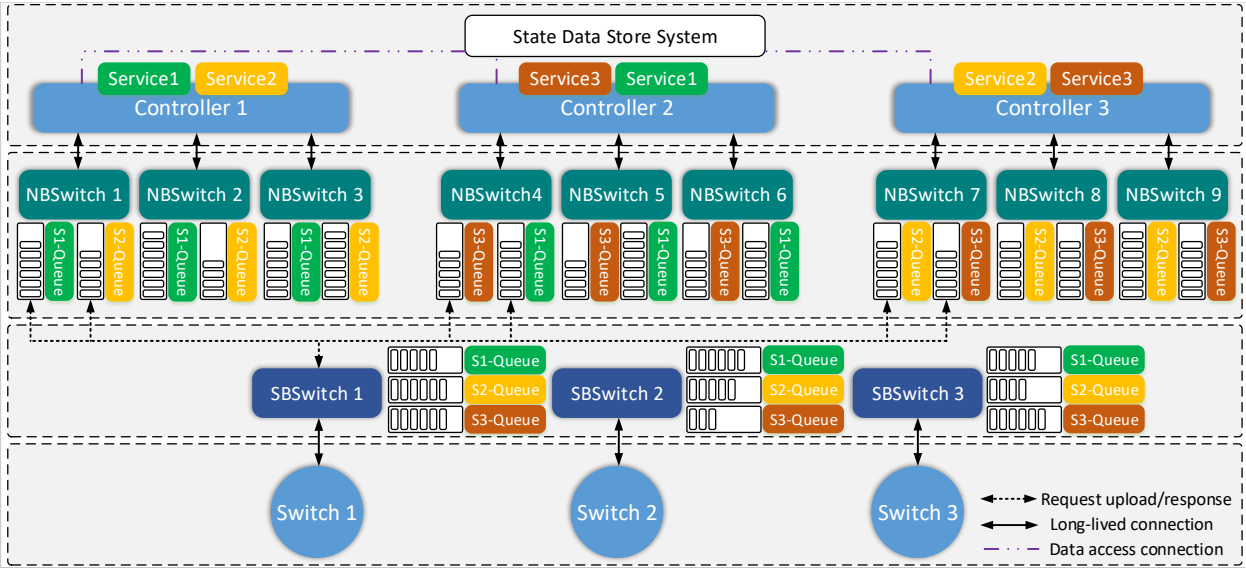


Fig. 1. Framework overview. The example has three `SBSwitch` modules and nine `NBSwitch` modules totally. `SBSwitch` modules are attached to physical switches by one-to-one mapping while `NBSwitch` modules are attached to controllers by n-to-one mapping. n is equal to the number of switches in the control plane. Each switch uses one `SBSwitch` module and three `NBSwitch` modules to assign requests to controllers. For every `SBSwitch` module, the amount of request queues it has depends on how many services the control plane can provide with. For every `NBSwitch` module, the amount of request queues it has depends on how many services the attached controller can provide with. Upon a receipt of a request, `SBSwitch` module categorizes the request according to its service type, pushes it into correspondent request queue (of this `SBSwitch` module), and then decides which controller to send it to. According to predefined mapping between two-layered request queues (as depicted with dotted arrow), the request is first sent from the `SBSwitch` module to correspondent `NBSwitch` module, and then to a controller for processing.

binding between switches and controllers. At the same time, for each type of request between each switch controller pair, the framework can provide a complete request queue, so that the assignment of requests is independent of each other. Based on the proposed framework, we can achieve dynamic request assignment by customizing the strategy to schedule between the request queues.

D. Statelessness of Request Assignment

Since there are distributed controllers in the control plane, it is a challenge to maintain state consistency during request assignment. For the existing mainstream framework ONOS [11], each controller will regularly back up its own state information and periodically exchange this information with each other to maintain state consistency during request processing. However, backup and information exchange require too much time, so it is difficult to adapt to dynamic, agile and flexible request assignment.

In the proposed framework, by separating the state from the controller into an independent data storage, we adopt the statelessness of request assignment. Statelessness has the following advantages: when the controller fails, a new controller can be instantiated immediately, and the new controller can directly access all the required states. When the control plane expands horizontally, the request can be sent to the new controller for processing immediately, without worrying about the inconsistency of the state between the new and old controllers. Most importantly, because each controller shares state, the controller's processing of requests does not depend on how the requests are assigned. With this state-separated architecture, the state can be pushed to a dedicated cache or back-end storage server, thus supporting the efficient and modular deployment of the control plane.

Although there are multiple services on the control plane, their states can generally be divided into the following two categories: static state, such as service configuration and SLA rules, and dynamic state, that is, the state that is continuously updated by the controller process. Among them, the dynamic state can be further divided into: internal state, such as file descriptors and temporary variables, and network state, such as routing strategies and service relevance in the network.

In fact, not all states need to be separated into data storage. When assigning requests, only the network state must be consistent for each controller, while the static state and internal state can be stored and accessed locally in the controller. Literature [25] shows that the use of an independent data storage layer can completely separate the state into the back-end storage or cache to maintain state consistency without significantly degrading processing performance. Therefore, we have reason to believe that the statelessness of request assignment is technically feasible, for example, using technologies such as RAMCloud [26] to build the state data storage system shown in Fig. 1. Bear in mind though, our motivation in this paper is not to design such a data storage system. Also note that we cannot claim that statelessness is necessary or that it is the optimal choice to solve state consistency problem in the control plane. We argue that statelessness can naturally fit with dynamic request assignment, so it should be a good complement to our framework.

III. Model of Request Assignment

The problem studied in this section is an online decision problem, that is, how to select an appropriate request queue

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2020.3048188, IEEE Transactions on Network Science and Engineering

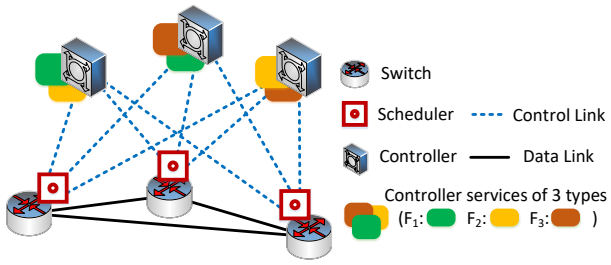


Fig. 2. Abstraction for the example depicted in Fig. 1. The abstracted control plane has three controllers and three switches. Connected to each switch is an abstracted scheduler, whose functionality is same with `SBSwitch` module and `NBSwitch` module, responsible for assigning requests to controllers, as well as returning the responses back to the switches. The control plane provides with three services in total. Every controller provides with two services, known as having two service replicas.

for the requests arriving in real time. We will show that this problem is a variant of the scheduling problem in a Stochastic Processing Network (SPN) [20] which has similar properties. By transforming this problem to the SPN one, we can apply the results from the SPN problem.

In order to better model the request assignment problem, we abstract the example in Fig. 1 and replace the `SBSwitch` module and the `NBSwitch` module with an abstracted scheduler. In the original example, each switch uses one `SBSwitch` module and three `NBSwitch` modules to assign requests to the controller. After abstraction, each switch uses a scheduler to assign requests. As shown in Fig. 2, the abstracted control plane contains three controllers and three switches. Connected to each switch is a scheduler, which is responsible for assigning requests to controllers and returning responses to the switches. The control plane provides three different services in total. Each controller provides two services, that is, has two service replicas.

Without loss of generality, we consider a request assignment case similar to Fig. 2. The control plane consists of multiple distributed controllers, each of which has computing resources for running multiple replicas of services. Connected to each controller is a group of schedulers, responsible for scheduling requests in the switch, and processing the response returned from the controller. The main symbols used in the model are shown in Table II.

A. Control Plane

We model the control plane as a graph $\mathcal{G} = (\mathcal{V}, \mathcal{S}, \mathcal{E})$, where $\mathcal{V}$ is the set of switches, $\mathcal{S}$ is the set of controllers, and $\mathcal{E}$ is the set of links interconnecting controllers and switches. Each controller $S \in \mathcal{S}$ has a total available resources of c_S ($c_S > 0$), and offers at least one service. For each link $E \in \mathcal{E}$, we use $d_E(l)$ to denote the l-th request transmission delay on link E, where $d_E(l)$ ($l \geqslant 1$) is assumed to be a sequence of i.i.d. random variables, with an average of $\overline{d}_E$ and a finite variance. In commercial datacenters, the average request transmission delays [27], [28] are usually stable (in μs) and significantly less than the average request processing delays (in ms) [17]. We therefore assume the request transmission delays are negligible compared to the request processing delays in our model.

Symbol	Meaning
$V \in \mathcal{V}$	Set of switches, each switch V can upload requests
$S \in \mathcal{S}$	Set of controllers, each controller S runs one or more replicas
c_S	Computational resources available on controller S
$E \in \mathcal{E}$	Set of links, each link E is connected to a switch-controller
$\overline{d}_E$	Average transmission delay of link E
$F \in \mathcal{F}$	Set of services, each service F has one or more replicas
μ_F	Request processing rate of service F under unit resource
$I \in \mathcal{I}$	Set of replicas, each replica I runs on a controller
$I_S \in \mathcal{I}_S$	Set of replicas running on controller S
w_{I_S}	Resource occupancy rate of replica I on controller S
$B \in \mathcal{B}$	Set of request queues, each request queue B is on a scheduler
$\vec{z}$	Vector of request queue occupancy rate
$A \in \mathcal{A}$	Set of activities in a SPN
R	Input-output matrix of the control plane

TABLE II
THE SYMBOLS IN THE MODEL

B. Service Replicas

Each service provided by the control plane is a processing logic applied to a specific type of request. We denote by $\mathcal{F}$ the set of services. As shown in Fig. 2, each service F has one or more service replicas in the control plane. We denote by $\mathcal{I}$ the set of service replicas, and by $\mathcal{I}_S$ the set of service replicas running on controller S. As we do not study the controller deployment problem, we assume that the controllers and their service replicas have been deployed in the control plane, by using any of the solutions proposed in the literature (see Section VII).

Without loss of generality, we assume that given equal resources, all service replicas of the same type have the same request processing rate. We denote by μ_F the processing rate of service F when provided one unit of computational resource. Thus, $k \times \mu_F$ is the processing rate of a replica of type F when k units of resources are allocated to this replica. This is a common model assumption in the literature [29] and this linear relationship between processing rate and resources has been verified in the literature [30]. Moreover, we assume that the available resources of a controller is shared among all the co-located replicas according to some given policy. Under such a policy, a replica $I_S \in \mathcal{I}_S$ has a resource occupancy rate w_{I_S} and the resource occupancy rates are constrained by enforcing $\sum_{I_S \in \mathcal{I}_S} w_{I_S} \leqslant 1$. Furthermore, we assume that each service replica has a local cache to store requests and the replica processes the requests in a non-preemptive manner.

C. Switch and Scheduler

Each switch $V \in \mathcal{V}$ runs a scheduler which behaves as both the ingress and egress for the requests. In our model, each scheduler has a set of buffers, which is an abstraction of the request queues of the `SBSwitch` module and the `NBSwitch` module.

We assume that the scheduler can perform the classification of the requests according to: the message type of the request, and the service chain attribution of the request. Generally, the message type of the request under the OpenFlow protocol can be determined by the type of OpenFlow message [7], and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2020.3048188, IEEE Transactions on Network Science and Engineering

the service chain attribution of the request can be determined by the service path identifier carried in the Network Service Header (NSH) [2]. After classification, the scheduler will label each request a service type, that is, the type of service replica corresponding to the request.

D. Variant of Stochastic Processing Network

The problem of dynamic request assignment is how to assign the requests in the request queues to the appropriate controller, and how much computational resources are allocated to each service replica on each controller, so as to maximize the request throughput of the control plane while optimizing the average response delay experienced by the requests.

The properties of this problem are similar to the ones of a Stochastic Processing Network (SPN). SPNs [20], [31] are a general class of network models that have been used in a wide range of fields [32], including manufacturing systems and cross-training of workers at a call center. The key elements of an SPN include a set of buffers, a set of processors, and a set of activities. Each buffer holds jobs that await service. Each activity takes jobs from at least one of the buffers and requires at least one available processor to process the jobs.

We can transform this problem into a variant of a SPN:

1) Buffer: In our model, each scheduler has a set of request queues for storing requests, namely buffers. All requests of the same type on the switch are stored in the same buffer. We denote by $\mathcal{B}$ the set of buffers in the model. When the switch receives a request, the scheduler will determine its type and push it to the corresponding buffer of that type. The requests stored in the same buffer will be scheduled in the order of First-in-First-out (FIFO).

2) Processor: Each controller in our model corresponds to a processor in the SPN. Each service has multiple service replicas in the control plane, so multiple controllers can handle requests from the same buffer, and the scheduler decides which controller to handle the requests.

3) Activity: In our model, each activity is to process a request from a buffer by an eligible controller. The set of all potential activities can be expressed as $\mathcal{A} = \{B \mapsto S | B \in \mathcal{B} \wedge S \in \mathcal{S}_B\}$, where $\mathcal{S}_B \in \mathcal{S}$ is the set of controllers eligible for the requests in buffer B. $B \mapsto S$ means an activity that processes a request from buffer B by controller S. We denote by $\mathcal{A}_B$ the set of activities that are related to buffer B and by $\mathcal{A}_S$ the set of activities that are related to controller S. We denote by μ_A the processing rate of activity A. We assume that μ_A is determined by the function $g(\cdot)$, and $g(\cdot)$ is related to the processing rates of requests and the transmission delays of requests, i.e., $\mu_A = g(\mu_F, \overline{d}_E)$. As mentioned in Section III-A, since the transmission delays of requests are negligible compared to the processing delays of requests, we hence assume that the processing rate of the activity is dominant by the request processing rate of the service in our model.

In order to further clarify the transformation between our model and the SPN, we take the case depicted in Fig. 3 as an example. Since there are three switches and three types of requests, there are nine buffers in Fig. 3, which is the same as the number of `SBSwitch` module request queues in Fig. 1. Activity A_3 and A_4 connect buffer B_2 to S_1 and S_3, respectively. $\mu_{A_3} = g(\mu_{F_2}, \overline{d}_{E_1})$, where E_1 is the link between `switch1` and `controller1`. $\mu_{A_4} = g(\mu_{F_2}, \overline{d}_{E_2})$, with E_2 being the link between `switch1` and `controller3`. And so on, knowing the control plane graph $\mathcal{G}$, given the set of service replicas $\mathcal{I}$, the set of buffers $\mathcal{B}$ and the set of activities $\mathcal{A}$ can be determined accordingly.

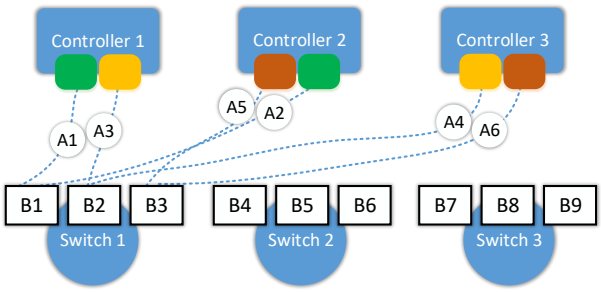


Fig. 3. A SPN representation of the example depicted in Fig. 2. The SPN has three classes of requests and nine buffers.

In this way, the dynamic request assignment problem is transformed into the SPN scheduling problem defined in the literature [20], [33]. Its goal is to provide a scheduling strategy for activities to maximize the throughput of the SPN while ensuring the stability of all buffers.

IV. Maximum Pressure Policy

In this section, we present the maximum pressure policy (MPP), and show its runtime guarantees on request throughput and response latency. Assume that in our request assignment framework, every scheduler is aware of the state of all the buffers, i.e., the buffer utilization given by $\vec{z}$ (a vector of size $|\mathcal{B}|$), and also of the state of every controller $S \in \mathcal{S}$, $q_S = \{0, 1\}$, where $q_S = 0$ if S is idle and 1 otherwise.

Prior works such as [20], [33] have shown that the asymptotically optimal scheduling can be obtained for SPNs by following the MPP. We now show that the MPP can also be applied to the dynamic request assignment problem thanks to its similar properties.

A. Scheduling of MPP

We denote by a column vector $\vec{h}$ of size $|\mathcal{A}|$ the resources consumed by the activities. If an activity is performed at h_A, it consumes a fraction of h_A resources of correspondent controller, where $0 \leqslant h_A \leqslant 1$ and $A \in \mathcal{A}$. Let $\mathcal{H}$ be the set of all feasible allocations for activities. For each buffer $B \in \mathcal{B}$ and each activity $A \in \mathcal{A}$, we define

$$r_{BA} = \begin{cases} \mu_A, & A \in \mathcal{A}_B, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

The matrix $\mathrm{R} = (r_{BA})$ is named the input-output matrix of the SPN. It captures the average processing rate of requests from buffer B consumed by activity A [31]. Given a weight vector $\vec{\alpha}$ of size $|\mathcal{B}|$, we define by

$$\Phi_{\vec{\alpha}}(\vec{h}, \vec{z}) = (\vec{\alpha} \times \vec{z}) \cdot \mathrm{R}\vec{h} \tag{2}$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2020.3048188, IEEE Transactions on Network Science and Engineering

Algorithm 1: MPP at the `SBSwitch` module

Input: the set of controllers $\mathcal{S}$;
the set of request queues $\mathcal{B}$;
the buffer utilization vector $\vec{z}$ (a vector of size $|\mathcal{B}|$);
the input-output matrix R;
the weight vector $\vec{\alpha}$ (a vector of size $|\mathcal{B}|$);
Output: the assignment A^* for the request;

let $\mathcal{S}^* = \emptyset$; let $\mathcal{B}^* = \emptyset$; let $\Phi^*_{AS} = 0$;
Thread **while** *True* **do**
  foreach *controller* $S \in \mathcal{S}$ **do**
    if *controller* S *is idle* **then**
      $\mathcal{S}^* = \mathcal{S}^* \cup \{S\}$; continue;
    $\mathcal{S}^* = \mathcal{S}^* \cup \{S\} - \{S\}$;
Thread **while** *True* **do**
  foreach *request queue* $B \in \mathcal{B}$ **do**
    if *request queue* B *has requests* **then**
      $\mathcal{B}^* = \mathcal{B}^* \cup \{B\}$; continue;
    $\mathcal{B}^* = \mathcal{B}^* \cup \{B\} - \{B\}$;
Thread **while** *True* **do**
  foreach *controller* $S \in \mathcal{S}^*$ **do**
    foreach *request queue* $B \in \mathcal{B}^*_S$ **do**
      Φ_{AS}, calculated refer to *Equation (4)*;
      $\Phi^*_{AS} = \max(\Phi_{AS}, \Phi^*_{AS})$;
    $A^* = \arg \Phi^*_{AS}$, calculated refer to *Equation (5)*;
    select assignment A^*;
    send a request from the selected request queue of the `SBSwitch` module to correspondent request queue of the `NBSwitch` module;

the network pressure with parameter $\vec{\alpha}$ under allocation $\vec{h} \in \mathcal{H}$ and buffer utilization $\vec{z}$. The MPP aims to maximize the network pressure by choosing suitable allocations

$$\vec{h}^* \in \arg\max_{\vec{h} \in \mathcal{H}} \Phi_{\vec{\alpha}}(\vec{h}, \vec{z}). \tag{3}$$

Note that $\mathcal{H}$ is bounded and convex. Since $\Phi_{\vec{\alpha}}(\vec{h}, \vec{z})$ is linear in $\vec{h}$, according to the *Corollary 1*, the maximum of $\Phi_{\vec{\alpha}}(\vec{h}, \vec{z})$ will be achieved at one of the extreme points.

Corollary 1. For any buffer utilization $\vec{z}$ ($z_B \geqslant 0, \forall B \in \mathcal{B}$), there exists an extreme allocation $\vec{h}^* \in \mathcal{H}$ that maximizes the network pressure $\Phi(\vec{h}, \vec{z})$ such that for each buffer B of $\vec{h}^*$, the buffer utilization z_B is positive.

Proof. Since each activity is only associated with one buffer, the SPN here is a strict Leontief network [34]. If only the preemptive scheduling in the network is considered, the corollary directly established [33]. For non-preemptive scheduling, if the SPN is a reversed Leontief network, the corollary still holds. Since each activity is only associated with one processor, the SPN satisfies the definition of a reversed Leontief network, so the corollary is also true for non-preemptive scheduling.

Corollary 2. The extreme allocation for maximum network pressure is an integer allocation.

Proof. For each processor $S \in \mathcal{S}$, let $\mathcal{A}_S \in \mathcal{A}$ be the set of activities that the processor can take. We assume that when the processor is idle, it takes on a dummy activity A_0. Thus, processor S will be able to take any of the activities in $\mathcal{A}^0_S = A_0 \cup \mathcal{A}_S$. We then prove the contradiction while considering an extreme allocation $\vec{h}$ such that $h_{\tilde{A}} \in (0, 1)$ for some activity $\tilde{A} \in \mathcal{A}$ (non-integer allocation).

Let $\tilde{S}$ be the processor that holds activity $\tilde{A}$. For each $A \in \mathcal{A}^0_{\tilde{S}}$, we define a new allocation $\vec{h}'$ by modifying $\vec{h}$: We process A with $h_A = 1$ at processor $\tilde{S}$ and keep the others unchanged. It is easy to check that $\vec{h}'$ is a feasible allocation. It follows that $\vec{h} = \{h_A h'_A : A \in \mathcal{A}^0_{\tilde{S}}\}$, where we set $h_{A_0} = 1 - \sum_{A \in \mathcal{A}^0_{\tilde{S}}, A \neq A_0} h_A$. Due to the fact that $\sum_{A \in \mathcal{A}^0_{\tilde{S}}} = 1, \tilde{A} \in \mathcal{A}^0_{\tilde{S}}$, and $h_{\tilde{A}} < 1$ by assumption, $\vec{h}$ is a linear combination of feasible allocations. As a result, $\vec{h}$ cannot be an extreme allocation, contradicting the assumption. Therefore, any extreme allocation must be an integer allocation.

According to the *Corollary 2*, the extreme allocation produced by the MPP will never split the processing capacity of a processor. Hence, the scheduling of MPP can be simplified as follows. For any controller $S \in \mathcal{S}$ and any activity $A \in \mathcal{A}_S$, we define

$$\Phi_{AS} = \sum_{B \in \mathcal{B}} \alpha_B r_{BA} z_B. \tag{4}$$

If controller S is in idleness, the scheduler selects activity

$$A^* \in \arg\max_{A \in \mathcal{A}_S} \Phi_{AS} \tag{5}$$

to be served by the controller. A tie-breaking rule will be applied in case more than one allocation attains the maximum.

By applying the above scheduling logic to our framework, we can execute the MPP on the `SBSwitch` module to schedule between the request queues. By monitoring the changes in the `NBSwitch` module request queues, we can track the workload on the controllers. The pseudo code of MPP is shown in Algorithm 1.

B. Performance Analysis

We analyze the performance of MPP from three perspectives: request throughput, response latency, and time complexity, and show that the MPP can provide runtime guarantees on the performance of control plane.

1) Request Throughput: Corollary 2 shows that the allocation produced by the MPP will never split the processing capacity of a processor. Under such a property, we can have the SPN stability result as shown in the *Corollary 3*. Based on the *Corollary 3*, we can further have *Corollary 4* to show that the MPP is asymptotically optimal with respect to request throughput.

Corollary 3. Non-preemptive MPP can stabilize the network.

Proof. To prove this corollary, we first introduce an auxiliary linear program that was called the static planning problem [31] as follows:

$$\begin{aligned} \min \quad & \rho \\ s.t. \quad & \mathrm{R}\vec{x} = 0, \\ & \sum_{A \in \mathcal{A}_S} x_A \leqslant \rho, \forall S \in \mathcal{S}, \\ & x_A > 0, \forall A \in \mathcal{A}. \end{aligned} \tag{6}$$

Here $\vec{x}$ is a column vector of size $|\mathcal{A}|$ representing the long-term fraction of time during which each activity is used. The problem indicates that the long-term input rate to the buffer is equal to the long-term output rate from the buffer.

According to the *Theorem 1* of [20], the SPN is stable if the static planning problem has a feasible solution with $\rho \leqslant 1$. According to the *Corollary 1* and the *Corollary 2*, the SPN is a reversed Leontief network and the MPP will never split the processing capacity of a processor. When applying *Theorem 9* of [20], we can prove that the non-preemptive non-processor-splitting MPP can stabilize the network if the static planning problem has a feasible solution with $\rho \leqslant 1$.

Corollary 4. For any $\vec{\alpha} > 0$, the MPP with parameter $\vec{\alpha}$ is asymptotically optimal with respect to request throughput.

Proof. Since the *Corollary 1* has implied that the SPN as well as its assumptions satisfy the extreme-allocation available (EAA) condition. Hence, in combination with the *Corollary 3* and the *Theorem 1* of [33], the MPP with parameter $\vec{\alpha}$ is asymptotically optimal for the throughput of the SPN.

2) Response Latency: According to the *Corollary 5*, the MPP is asymptotically optimal for minimizing a cost function of buffer occupancy levels. And the waiting time of the request in the buffer accounts for most of the response time, so the MPP provides a approximate guarantee on the response latency.

Corollary 5. For any given $\varepsilon > 0$, there exists a $\vec{h}^*$ that is asymptotically optimal for a quadratic cost function of the buffer utilization $\vec{z}$, i.e., $\sum_{B\in\mathcal{B}} \alpha_B (z_B)^2$.

Proof. The proof of the corollary follows from the fact that the SPN and its assumptions satisfy *Assumptions 1-4* of [33]. Thus, the same result on asymptotically optimality of quadratic cost holding in *Theorem 3* of [33] applies here.

3) Time Complexity: According to the *Corollary 6*, the time complexity of MPP is bounded by a linear term on the total number of switches, which we expect to be much smaller than the number of servers or the number of service function chains. This low time complexity indicates that the MPP can provide runtime guarantees on the performance of control plane.

Corollary 6. The MPP scheduler has a time complexity of $O(|\mathcal{V}|)$, where $|\mathcal{V}|$ is the total number of switches.

Proof. To find the optimal allocation, and for a given $S \in \mathcal{S}$, the MPP scheduler need to perform the *Equation (4)* for all $A \in \mathcal{A}_S$ and then apply *Equation (5)*. Note that r_{BA} under the summation has non-zero values for only one or two $B \in \mathcal{B}$ (refer to *Equation (1)*). The *Equation (4)* can be regarded as the summation of these two terms, and hence has $O(1)$ time complexity. The MPP scheduler therefore has a time complexity of $O(|\mathcal{B}|)$ as $|\mathcal{A}_S| \leqslant |\mathcal{B}|$. Since $|\mathcal{B}| = k|\mathcal{V}|$ where k is the total number of request types which is a constant, the MPP scheduler has a time complexity of $O(|\mathcal{V}|)$.

V. Distributed MPP

Despite those benefits, MPP still faces practical constraints inherent to large-scale deployment. MPP is a strategy that executes a scheduling for each request, and it needs the global state information to make decisions. On the one hand, with the expansion of the network scale, the cost of obtaining network-wide state information is increasing. On the other hand, even if the logic of each scheduling is very lightweight, scheduling for each request still brings excessive runtime overhead to the switch, which will cause the switch to overload. To deal with these constraints, we propose the distributed maximum pressure policy (DMPP). DMPP runs in local state on each switch and performs scheduling logic for batch requests.

A. Local State

In the DMPP, we assume that the scheduler running at each switch $V \in \mathcal{V}$ only knows the local state related to itself. These states include: the state of the buffers (i.e., buffer utilization), and the state of the connected controllers (i.e., whether controllers are busy or idle). Other information, such as the position of the service replica in the control plane, the average processing rate of the service replica, and the correspondence between the switch and the service replica, are staticallly known to each switch. The scheduler does not use the information of request queues, controllers, and service replicas that are not related to itself.

The DMPP applies similar scheduling logic as the MPP, but only on local buffers, local processors, and local activities. In other terms, it assumes that the SPN is only composed of the buffers at the local switch, the controllers that are connected to the local switch, and the activities which connect these buffers to these controllers, then it performs the MPP over the local SPN to pick the activity whenever a controller is idle. The DMPP applies a local version of *Equation (4)* as follows:

$$\hat{\Phi}_{AS} = \sum_{B\in\hat{\mathcal{B}}} \hat{\alpha}_B \hat{r}_{BA} \hat{z}_B. \tag{7}$$

Here $\hat{\vec{z}}$ is the local buffer utilization and $\hat{\mathrm{R}} = (\hat{r}_{BA})$ is the local input-output matrix, with values as $\vec{z}$ and R for buffers and activities that are local (and 0 otherwise).

Performance analysis of DMPP. According to the *Corollary 7*, the DMPP is locally asymptotically optimal with respect to request throughput, and is also locally asymptotically optimal for minimizing a cost function of buffer occupancy levels. According to the *Corollary 8*, the DMPP scheduler has a time complexity of $O(1)$.

Corollary 7. For any $\hat{\vec{\alpha}} > 0$, the DMPP with parameter $\hat{\vec{\alpha}}$ is locally asymptotically optimal with respect to request throughput. For any given $\varepsilon > 0$, there exists an allocation that is locally asymptotically optimal for a quadratic cost function of the buffer utilization $\hat{\vec{z}}$, i.e., $\sum_{B\in\hat{\mathcal{B}}} \hat{\alpha}_B (\hat{z}_B)^2$.

Proof. Since the DMPP executes the scheduling logic of the MPP on the local SPN, the same result on asymptotically optimality holding in *Corollary 4* and *Corollary 5* applies here. Therefore, in terms of the request throughput and the quadratic cost function of the buffer utilization, DMPP can reach asymptotically optimal on the local SPN.

Corollary 8. The DMPP scheduler has a time complexity of $O(1)$.

Proof. On the local SPN, and for a given $S \in \hat{\mathcal{S}}$, the DMPP scheduler has $O(1)$ time complexity for all $A \in \hat{\mathcal{A}}_S$, according to the *Corollary 6*. Since $|\hat{\mathcal{A}}_S| = c$ where c is the number of buffers on the switch which is a constant, the DMPP scheduler has a time complexity of $O(1)$.

B. Batch Scheduling

To alleviate the constraints on runtime scheduling overhead, we introduce the batch scheduling, where the batch size n

Algorithm 2: DMPP at the `SBSwitch` module

Input: the set of controllers $\hat{\mathcal{S}}$;
the set of request queues $\hat{\mathcal{B}}$;
the buffer utilization vector $\hat{\vec{z}}$ (a vector of size $|\hat{\mathcal{B}}|$);
the local input-output matrix $\hat{R}$;
the weight vector $\hat{\vec{\alpha}}$ (a vector of size $|\hat{\mathcal{B}}|$);
the size of request batch n;
Output: the assignment $\hat{A}^*$ for the request batch;
let $\hat{\mathcal{S}}^* = \emptyset$; let $\hat{\mathcal{B}}^* = \emptyset$; let $\hat{\Phi}^*_{AS} = 0$; let $n^* = n$;
Thread **while** *True* **do**
  foreach *controller* $\hat{S} \in \hat{\mathcal{S}}$ **do**
    if *controller* $\hat{S}$ *is idle* **then**
      $\hat{\mathcal{S}}^* = \hat{\mathcal{S}}^* \cup \{\hat{S}\}$; continue;
    $\hat{\mathcal{S}}^* = \hat{\mathcal{S}}^* \cup \{\hat{S}\} - \{\hat{S}\}$;
Thread **while** *True* **do**
  foreach *request queue* $\hat{B} \in \hat{\mathcal{B}}$ **do**
    if *request queue* $\hat{B}$ *has requests* **then**
      $\hat{\mathcal{B}}^* = \hat{\mathcal{B}}^* \cup \{\hat{B}\}$; continue;
    $\hat{\mathcal{B}}^* = \hat{\mathcal{B}}^* \cup \{\hat{B}\} - \{\hat{B}\}$;
Thread **while** *True* **do**
  foreach *controller* $\hat{S} \in \hat{\mathcal{S}}^*$ **do**
    foreach *request queue* $\hat{B} \in \hat{\mathcal{B}}^*_{\hat{S}}$ **do**
      $\hat{\Phi}_{AS}$, calculated refer to *Equation (7)*;
      $\hat{\Phi}^*_{AS} = \max(\hat{\Phi}_{AS}, \hat{\Phi}^*_{AS})$;
    $\hat{A}^* = \arg \hat{\Phi}^*_{AS}$, calculated refer to *Equation (5)*;
    select assignment $\hat{A}^*$;
    let m = the amount of pending requests in the selected buffer;
    if $m < n$ **then**
      $n^* = m$;
    send n^* requests from the selected request queue of the `SBSwitch` module to correspondent request queue of the `NBSwitch` module;

specifies the (maximum) number of requests which DMPP would send over to a controller at each scheduling round. More precisely, each DMPP scheduler keeps track of how many requests are pending, queued, at service replicas running at each of controllers. If the pending requests in a controller is close to drained, the scheduler selects the next activity over this controller, sending up to n requests from the selected buffer to the controller (if the buffer has more than n requests, the scheduler sends n requests over the controller, otherwise, it sends as many requests as are in the buffer). Using the batch scheduling, we reduce the scheduling granularity to one decision per batch, and hence reduce the runtime overhead of scheduling. The pseudocode of DMPP is shown in Algorithm 2.

It is worth noting that the larger the batch size we set, the less likely the DMPP will perform the best scheduling decision. Therefore, how to balance the runtime overhead of batch scheduling and the loss of optimality is an important issue. If the granularity of batch scheduling is set too finely, the rounds of scheduling may be too intensive, which brings unnecessary runtime overhead to the scheduler or the `SBSwitch` module. If the granularity is set too coarsely, the scheduling will not be able to catch up with the dynamic change of the request arrival rate, which will reduce the optimization degree of the scheduling result.

Overall, MPP is a method of scheduling each request once, while DMPP is a method of scheduling each request batch once, so DMPP changes the granularity of request assignment from a single request to N requests. We choose DMPP mainly for the following two reasons: First, very fine-grained request scheduling may not be required in actual scenarios. Accordingly, the request assignment can be appropriately coarse-grained. Even so, the granularity of DMPP is still finer than that of the existing solutions based on switch migration [16], [18]. Second, appropriately coarse-grained request assignment can effectively adjust the scheduler's scheduling rounds in each time period, thereby reducing the scheduler's runtime overhead.

VI. Evaluation

In this section, we use the settings that can represent real scenarios to evaluate the performance of the proposed framework in request assignment. To this end, we implement a prototype system and use it as a test platform for simulation experiments. The prototype system is compatible with the control plane specification of the OpenFlow protocol [7].

In general, the results of the experiment show that the solution we propose can more effectively use the resources of the control plane, and can provide effective guarantees for the requested throughput and response latency. The results of the experiment also show that DMPP has performance close to that of MPP, and that DMPP requires much less resources when assigning requests on the switch, so resource utilization is higher. In addition, we compare the proposed framework with existing solutions based on switch migration [16], [18]. The results show that our proposed framework makes full use of agile and flexible request assignment capabilities, so its performance is significantly better than existing solutions.

The entire experimental part is trying to answer the following three questions: First, what is the performance of MPP and DMPP in request assignment. Second, what are the benefits of using fine-grained mechanisms to assign requests. Finally, why does DMPP need a batch scheduling mechanism, and what impact does the batch size have?

A. Simulation Setup

In order to ensure the fairness of comparison, we used experimental settings similar to those in the literature [18].

1) Topology: We use fat-tree [27] and VL2 [28] topologies that are common in datacenters for simulation. For the fat-tree topology, we set the number of pods to 24, so there are a total of 3456 hosts and 720 switches in the topology. For the VL2 topology, we set the degree of the intermediate switch and the aggregate switch to 48, so there are a total of 576 racks in the topology, each rack has 10 hosts, and a total of 648 switches. Such a topology scale can be compared with the general commercial datacenter [13].

2) Trace: Since the request is usually triggered by the flow arrival event on the switch, we simulate the arrival rate of the request and make it follow the flow arrival rate distribution measured in the actual datacenter [13], where the CDF of the flow inter-arrival time is plotted in Fig. 9. We have also

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2020.3048188, IEEE Transactions on Network Science and Engineering

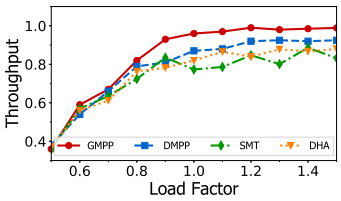


Fig. 4. Normalized throughput for fat-tree topology.

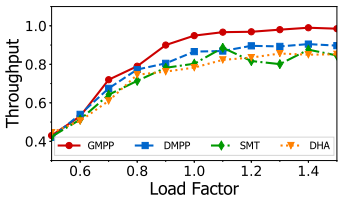


Fig. 5. Normalized throughput for VL2 topology.

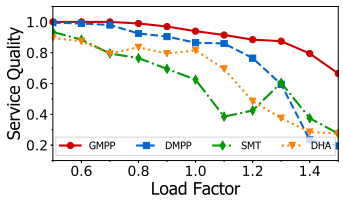


Fig. 6. Ratio of the responses observed satisfying the QoS deadlines to the total for fat-tree topology.

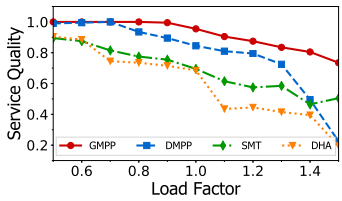
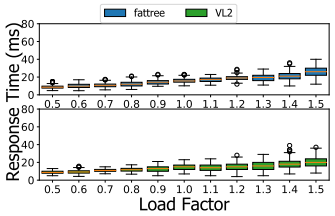
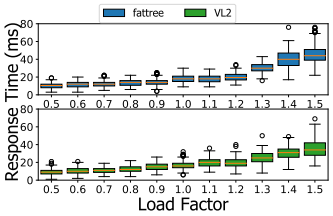


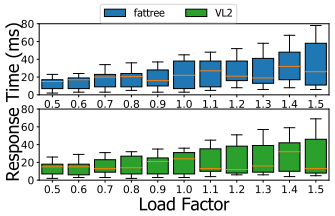
Fig. 7. Ratio of the responses observed satisfying the QoS deadlines to the total for VL2 topology.



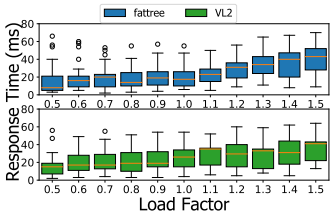
(a) GMPP.



(b) DMPP.



(c) SMT.



(d) DHA.

Fig. 8. Distribution of response time for fat-tree and VL2 topology.

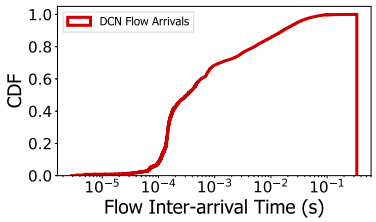


Fig. 9. CDF of flow inter-arrival times in the UNI1 DCN traffic dataset [13].

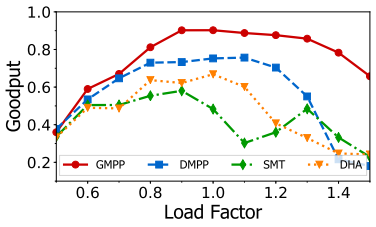


Fig. 10. Normalized throughput satisfying the QoS deadlines for fat-tree topology.

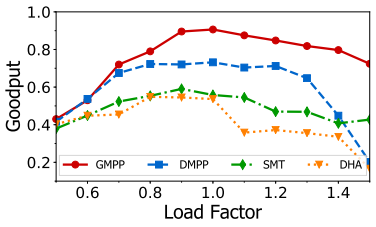


Fig. 11. Normalized throughput satisfying the QoS deadlines for VL2 topology.

introduced a load factor to change the arrival rate of requests to evaluate the performance under different load conditions. We place 16 controllers on the control plane in advance to meet the peak load of all switches, and all controllers have the same capacity. In the datacenter, although the transmission delays of requests (in μs) are much smaller than the processing delays of requests (in ms) and can be ignored [17], [27], [28], we still simulate the transmission delays with a positive Gaussian distribution $\mathcal{N}(1, 0.1)$. If the transmission delay is not considered here, the experimental results will be more ideal.

3) Request Class: We simulate 10 different types of requests. According to literature [17], the processing rate of each controller is 3.7×10^4 requests/sec, and the worst-case response time is 24.5 ms. Therefore, we simulate the processing rate of each controller for these 10 type of requests as $\{1.0, 1.6, 2.2, 2.8, 3.4, 4.0, 4.6, 5.2, 5.8, 6.4\} \times 10^4$ requests/sec. In order to evaluate the QoS guarantee ability of the proposed solution, we simulate the response deadline of these 10 type of requests as $\{29, 28, 27, 26, 25, 24, 23, 22, 21, 20\}$ms. The total number of requests for each type is the same.

B. Solutions Compared

1) GMPP: In GMPP, the schedulers use global state information to make decisions. Each scheduler knows the occupancy rate of all request queues, and the workload of all service replicas. The scheduling granularity of GMPP is to schedule once for each request.

2) DMPP: In DMPP, the schedulers use local state information to make decisions. Each scheduler only knows the occupancy rate of local request queues, and the workload of local service replicas. The scheduling granularity of DMPP is to schedule once for each batch of requests, so DMPP sends a batch of requests to the controller in each scheduling round. Unless explicitly stated, the batch size is set to 50.

3) SMT: SMT is a state-of-the-art solution based on switch migration [18]. It uses the RFHC framework to decompose the online switch migration problem into a series of matching problems under a single time slot, and the matching problem under each single time slot is divided into two stages to solve. In the first stage, it will produce an initial match between the switches and the controllers. In the second stage, it will input the generated initial match into an alliance game model to further optimize the match. SMT needs to predict the request arrival rate in the future time slots, and use the above-mentioned optimization method to make decisions based on the predicted request arrival rate. According to the setting in the literature [18], we use the prediction method based on the harmonic mean to achieve the required prediction. The number of predicted time slots is 2, and the length of each time slot is set to 10s.

4) DHA: DHA is also a state-of-the-art solution based on switch migration [16]. It uses the Markov approximation

framework to dynamically determine the starting point and the destination of switch migration. If the load of a certain controller exceeds the threshold in consecutive time slots, it will activate the corresponding switch migration mechanism. In order to match the setting in [16], we set the above threshold to 80%, the duration to 2 time slots, and the length of each time slot to 10s.

TABLE III
EXPERIMENTAL METHODOLOGY.

Methods	Scheduling Granularity	Service Differentiation	Distributed	Proactive
GMPP	request granularity	✓	×	✓
DMPP	request granularity	✓	✓	✓
SMT [18]	switch granularity	×	×	✓
DHA [16]	switch granularity	✓	✓	×

As shown in Table III, we compare these solutions in multiple dimensions. Both GMPP and DMPP are based on the granularity of requests, while SMT and DHA are based on the granularity of switches. Except for SMT, all other solutions consider the diversity of requests. DMPP and DHA decompose the request assignment problem on the switches and the controllers respectively, while other solutions do not. Only DHA re-assign requests in a reactive way, while all other solutions are proactive.

Additionally, these existing solutions require the priori knowledge such as the arrival rate of requests in order to perform scheduling. Compared with the existing solutions, our solutions (including GMPP and DMPP) do not require more state information to make decisions. We consider the runtime scheduling of requests and assume no priori knowledge of request arrival rate distribution.

C. Metrics Cared

1) Request Throughput: The number of requests processed per unit time in the control plane. This indicator reflects the ability of the scheduling strategy to utilize control plane resources. Generally, the request throughput is affected by the load balancing between controllers.

2) Response Latency: The time taken from the request to the switch to leave the output queue of the controller. This indicator reflects the quality of service provided by the scheduling strategy. Generally, the response latency for each request is determined by its waiting time in the request queue.

3) Service Quality: The ratio of the responses that satisfy the QoS deadlines to the total. This indicator can measure the strength of the scheduling strategy to guarantee service quality.

4) Goodput: The throughput satisfying the QoS deadlines. This metric reflects not only the effect of load balancing in the control plane, but also reflects the guarantee effect of the scheduling strategy on the quality of service.

D. Request Throughput

The request throughput for fat-tree and VL2 is plotted in Fig. 4 and 5, where we compare GMPP, DMPP, SMT and DHA under different request loads (x-axis on the figures). From the results, we make the following observations: (a) As the request load increases, the throughput also increases since the resource bottleneck does not appear at the beginning (i.e., load factor below 1.0); (b) When the load grows to a certain point, the throughput stops increasing. This is expected, because long running requests accumulate in the controllers over time, and it becomes difficult for any strategy to exploit more processing rates from the control plane; (c) Compared to other solutions, SMT has the most unstable and unpredictable throughput; (d) GMPP outperforms DMPP and increases the throughput by 7.3% in fat-tree and 7.9% in VL2 on average, thanks to its global view. DMPP outperforms DHA by 4.6% in fat-tree and 5.8% in VL2, and outperforms SMT by 7.1% in fat-tree and 5.4% in VL2, thanks to its fine-grained scheduling.

E. Response Latency

We also investigate how our scheduling policy performs in terms of response latency. Fig. 8 provides a comparison of the response time across different solutions under different request loads. We observe that as the total number of requests increases, the response time also increases since the available computational resources on controllers are limited. We also find that our scheduler offers a better response time than state-of-the-art solutions. Specially, GMPP outperforms DMPP and decreases the response time by 18.4% in fat-tree and 20.9% in VL2 on average. DMPP outperforms DHA by 9.2% in fat-tree and 12.4% in VL2, and outperforms SMT by 6.1% in fat-tree and 7.6% in VL2. Additionally, GMPP and DMPP provide a more stable response time than DHA and SMT, thanks to the fine-grained scheduling. Even with a load factor above 1.0, GMPP still answers 91.3% of requests in less than 30ms in fat-tree and 93.5% in VL2 on average. Because of the reactive nature of DHA, some controllers are always left as hot-spots. Similar to the result on request throughput, the scheduling in SMT also provides the most unstable and unpredictable response time.

F. Quality of Service

Fig. 6 and 7 plot the results of QoS in fat-tree and VL2 across different solutions under different request loads. As the total number of requests increases, the ratio of the responses satisfying the QoS deadlines to the total decreases gradually. GMPP outperforms DMPP and increases the QoS by 20.8% in fat-tree and by 16.0% in VL2 on average. DMPP outperforms DHA by 16.6% in fat-tree and 27.5% in VL2, and outperforms SMT by 22.8% in fat-tree and 15.3% in VL2. When the load factor is greater than 1.3, the QoS of DMPP is worse than that of SMT. This shows that in terms of QoS, the global view based SMT has more advantages than the local view based DMPP when the load is heavy. However, compared with GMPP, which is also based on a global view, SMT still has a significant gap.

Fig. 10 and 11 plot the comparisons on goodput across different algorithms under different request loads. We observe that the goodput first increases then falls down. We also find that our scheduler performs better than other solutions. GMPP outperforms DMPP and increases the goodput by 34.2% in fat-tree and 27.9% in VL2 on average. DMPP outperforms DHA by 22.1% in fat-tree and 39.4% in VL2, and outperforms SMT by 32.3% in fat-tree and 21.3% in VL2. Similar to the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2020.3048188, IEEE Transactions on Network Science and Engineering

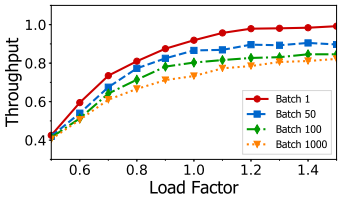


Fig. 12. Normalized throughput for fat-tree topology under different batch sizes.

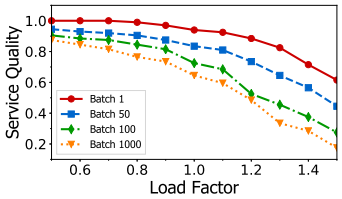


Fig. 13. Ratio of the responses observed satisfying the QoS deadlines to the total for fat-tree topology under different batch sizes.

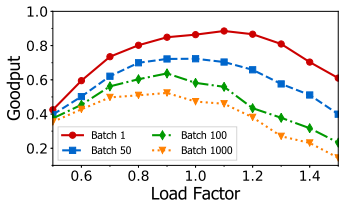


Fig. 14. Normalized throughput satisfying the QoS deadlines for fat-tree topology under different batch sizes.

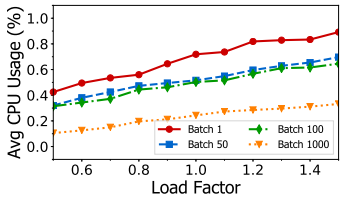


Fig. 15. Average cpu utilization of schedulers for fat-tree topology under different batch sizes.

result on request throughput and response latency, SMT's QoS and goodput are also the most unstable and unpredictable. As shown in Fig. 10, SMT has two obvious local maximums and minimums under different load factors, and a similar situation also appears in Fig. 6. Firstly, this is because SMT does not consider the difference in request types, and secondly, this is because SMT is too sensitive to errors in the request arrival rate prediction model. The above two reasons cause the performance of SMT to be unstable and unpredictable.

G. Batch Scheduling

We also study the effect of different batch sizes on the performance of DMPP. Fig. 12, 13 and 14 plot the performance degradation of using large batch size in fat-tree. Overall, as the size of request batch increases, the performance degradation, in terms of throughput, response time, and goodput increases gradually. This is expected, since the larger the batch size set, the fewer possibilities a DMPP scheduler has for choosing the best scheduling decision. Fig. 15 plots the average cpu usage of the schedulers under different batch sizes in fat-tree. We find that the batch scheduling significantly reduces the runtime overhead, and this effect is more significant as the batch size increases. This is also expected, because the larger the batch size is set, the fewer scheduling rounds the scheduler executes at each time period. Therefore, the batch scheduling trades slightly reduced optimization for significantly improved scalability in terms of runtime overhead.

VII. Related Work

1) NFV and Service Function Chaining: Software-defined networking (SDN) and Network Function Virtualization (NFV) is the key enablers of network softwarization. Prior study show that integrating the SDN and NFV may trigger innovative designs that fully exploit the advantages of both paradigms [35]. 5G networks will also rely on these technologies to create end-to-end logical networks on demand [36], [37]. Normally, such self-contained networks depends on flexible service function chaining mechanism [2] to simultaneously accommodate diverse service slices on a common network infrastructure [38]. Each slice is defined to have particular workloads and meet particular service requirements [39], [40], bringing with significant differentiated SDN control plane processing [5].

2) Distributed Control Plane: Originally, the control plane of SDN was tied to a single point [21], [41], even in the presence of high-availability clusters [12]. When more larger-scale deployment is considered for datacenters, a distributed control plane with multiple controllers becomes necessary. To address this, several distributed controller platforms have been proposed [11], [42], [43], [44], [45], [46]. Distributed controller platforms require policy to efficiently use resource and to balance load in the control plane, bringing with them the problem of dynamic request assignment.

3) Dynamic Request Assignment: For better utilization of controller resources, Dixit et al. [17] propose a protocol to enable switch migration across multiple controllers. Dynamic switch migration across controllers to assign requests becomes technically feasible. On basis of switch migration framework, some solutions on dynamic request assignment such as [16], [18], [19], [47] are come up with, in which the request assignment of the control plane is periodically adapted to the dynamic arrival of requests. These coarse-grained solutions however cannot do load balancing in run-time, because they rely on heavy message exchanges which are time-consuming, and the time complexity of their decision algorithms is much too high. The frequency of their decision-making is thus limited to allow for the migration to happen slow enough to fit with the overhead. The effect of their decision algorithms is not ideal because they require aggregated information about request arrival which provides only an estimation of the actual request arrival. Due to these reasons, they fail to explore and exploit the resources that become available on the fly as a result of real-time changes in the control plane, yielding only poor resource usage. Besides, although one [16] of them considers different request classes, the binding between switches and controllers makes their request assignment inflexible, and inherently difficult to adapt to the practice of request diversity.

4) Placement of Controllers: We assume that the controllers, with possibly multiple replicas for each service, are already deployed in the control plane, e.g., by using existing algorithms [48], [49], [50]. The majority of existing deployment solutions focus on the placement problem, by deciding where controllers should be deployed (e.g., at which geographical locations) and how much resources (e.g., computing capacity, memory, storage) should be allocated to each of them. The request assignment proposed by these solutions is performed in a rather static manner, where the requests are assigned among the deployed controllers without dynamic load-balancing performed among them. A few solutions that pay close attention

to the incremental deployment of control plane have also recently been proposed [51], [52].

5) Control Plane Request Bypass: These approaches are to offload part of the control logic to the data plane [53], [54]. Unfortunately, the limited function and space available at the data plane prevents such solutions from handling the complex control logic of many services. Despite their limitations, and the fact that dynamic request assignment is still necessary, these approaches actually improve the utilization of the switch resources and reduce the overhead to the control plane to a certain extent, hence are complementary to our work.

6) Stochastic Processing Networks: SPNs [20], [31] are a general class of network models that have been used in a wide range of fields [32], including manufacturing systems and cross-training of workers at a call center. The key elements of an SPN include a set of buffers, a set of processors, and a set of activities. Each buffer holds jobs that await service. Each activity takes job(s) from at least one of the buffers and requires at least one processor available to process the job(s). In this paper, we consider the dynamic request scheduling problem to be a variant of the scheduling problem in an SPN. The results from the SPN problem [33] can then be applied to our problem to find the optimal policy.

VIII. Conclusion

Experimental results show that the framework and request assignment strategies (including MPP and DMPP) proposed in this paper can effectively utilize the resources of controllers, balance the load, improve the throughput, and reduce the response time of requests. We expect the new framework can bring the historical progression of request assignment to a next stage, where the assignment is no longer based on the granularity of switches but rather based on the more efficient granularity of requests. For DMPP, how to balance the runtime overhead of batch scheduling and the loss of optimality remains an important issue. In the future, we will explore in more details the trade-offs related to scheduling granularity, to better understand how to achieve fine-grained scheduling within an acceptable runtime overhead.

Acknowledgment

This work was supported in part by the National Key R&D Program of China under Grant 2019YFB2102404, in part by the NSFC under Grants 61772112, 62072069, 61672379, in part by the Science Innovation Foundation of Dalian under Grant 2019J12GX037.

References

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

[2] P. Quinn and J. Guichard, "Service function chaining: Creating a service plane via network service headers," *IEEE Computer*, vol. 47, no. 11, pp. 38–44, 2014.

[3] A. M. Medhat, T. Taleb, A. Elmangoush *et al.*, "Service function chaining in next generation networks: State of the art and research challenges," *IEEE Communications Magazine*, vol. 55, no. 2, pp. 216–223, 2016.

[4] L. Guo, J. Pang, and A. Walid, "Dynamic service function chaining in sdn-enabled networks with middleboxes," in *Proc. of IEEE International Conference on Network Protocols*, 2016.

[5] I. Afolabi, T. Taleb, K. Samdanis *et al.*, "Network slicing and softwarization: A survey on principles, enabling technologies, and solutions," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 3, pp. 2429–2453, 2018.

[6] M. Boucadair *et al.*, "Service function chaining (sfc) control plane components & requirements," *Work in Progress, draft-ietf-sfc-control-plane-08*, 2016.

[7] N. McKeown, T. Anderson, H. Balakrishnan *et al.*, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[8] S. K. Fayazbakhsh, V. Sekar, M. Yu *et al.*, "Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions," in *Proc. of the ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013.

[9] Y. Zhang, N. Beheshti, L. Beliveau *et al.*, "Steering: A software-defined networking for inline service chaining," in *Proc. of the IEEE international conference on network protocols*, 2013.

[10] Z. A. Qazi, C.-C. Tu, L. Chiang *et al.*, "Simple-fying middlebox policy enforcement using sdn," in *Proc. of the ACM SIGCOMM conference*, 2013.

[11] P. Berde, M. Gerola, J. Hart *et al.*, "Onos: towards an open, distributed sdn os," in *Proc. of ACM HotSDN*, 2014.

[12] J. Medved, R. Varga, A. Tkacik *et al.*, "Opendaylight: Towards a model-driven sdn controller architecture," in *Proc. of IEEE WoWMoM*, 2014.

[13] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. of ACM IMC*, 2010.

[14] A. Roy, H. Zeng, J. Bagga *et al.*, "Inside the social network's (datacenter) network," vol. 45, no. 4, pp. 123–137, 2015.

[15] S. Kandula, S. Sengupta, A. Greenberg *et al.*, "The nature of data center traffic: measurements & analysis," in *Proc. of the ACM SIGCOMM conference on Internet measurement*, 2009.

[16] X. Ye, G. Cheng, and X. Luo, "Maximizing sdn control resource utilization via switch migration," *Computer Networks*, vol. 126, pp. 69–80, 2017.

[17] A. Dixit, F. Hao, S. Mukherjee *et al.*, "Elasticon: an elastic distributed sdn controller," in *Proc. of ACM/IEEE ANCS*, 2014.

[18] T. Wang, F. Liu, and H. Xu, "An efficient online algorithm for dynamic sdn controller assignment in data center networks," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 2788–2801, 2017.

[19] Y. Xu, M. Cello, I.-C. Wang *et al.*, "Dynamic switch migration in distributed software-defined networks to achieve controller load balance," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 515–529, 2019.

[20] J. G. Dai and W. Lin, "Maximum pressure policies in stochastic processing networks," *Operations Research*, vol. 53, no. 2, pp. 197–218, 2005.

[21] N. Gude, T. Koponen, J. Pettit *et al.*, "Nox: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[22] 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.

[23] Y. Zhang, L. Cui, W. Wang *et al.*, "A survey on software defined networking with multiple controllers," *Journal of Network and Computer Applications*, vol. 103, pp. 101–118, 2018.

[24] T. Hu, Z. Guo, P. Yi *et al.*, "Multi-controller based software-defined networking: A survey," *IEEE Access*, vol. 6, pp. 15 980–15 996, 2018.

[25] M. Kablan, A. Alsudais, E. Keller *et al.*, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proc. of USENIX NSDI*, 2017.

[26] D. Ongaro, S. M. Rumble, R. Stutsman *et al.*, "Fast crash recovery in ramcloud," in *Proc. of ACM SOSP*, 2011.

[27] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," vol. 38, no. 4, pp. 63–74, 2008.

[28] A. Greenberg, J. R. Hamilton, N. Jain *et al.*, "Vl2: a scalable and flexible data center network," vol. 39, no. 4, pp. 51–62, 2009.

[29] G. P. Katsikas, T. Barbette, D. Kostic, R. Steinert, and G. Q. Maguire Jr, "Metron: Nfv service chains at the true speed of the underlying hardware," in *Proc. of USENIX NSDI*, 2018.

[30] J.-J. Kuo, S.-H. Shen, H.-Y. Kang *et al.*, "Service chain embedding with maximum flow in software defined network and application to the next-generation cellular network architecture," in *Proc. of IEEE INFOCOM*, 2017.

[31] J. M. Harrison, "Brownian models of open processing networks: Canonical representation of workload," *The Annals of Applied Probability*, vol. 13, no. 1, pp. 390–393, 2003.

[32] M. J. Neely, "Stochastic network optimization with application to communication and queueing systems," *Synthesis Lectures on Communication Networks*, vol. 3, no. 1, pp. 1–211, 2010.

[33] J. G. Dai and W. Lin, "Asymptotic optimality of maximum pressure policies in stochastic processing networks," *Annals of Applied Probability*, vol. 18, no. 6, pp. 2239–2299, 2008.

[34] M. Bramson and R. Williams, "Two workload properties for brownian networks," *Queueing Systems*, vol. 45, no. 3, pp. 191–221, 2003.

[35] Q. Duan, N. Ansari, and M. Toy, "Software-defined network virtualization: An architectural framework for integrating sdn and nfv for service provisioning in future networks," *IEEE Network*, vol. 30, no. 5, pp. 10–16, 2016.

[36] P. Rost, C. Mannweiler, D. S. Michalopoulos *et al.*, "Network slicing to enable scalability and flexibility in 5g mobile networks," *IEEE Communications magazine*, vol. 55, no. 5, pp. 72–79, 2017.

[37] X. Foukas, G. Patounas, A. Elmokashfi *et al.*, "Network slicing in 5g: Survey and challenges," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 94–100, 2017.

[38] K. Samdanis, X. Costa-Perez, and V. Sciancalepore, "From network sharing to multi-tenancy: The 5g network slice broker," *IEEE Communications Magazine*, vol. 54, no. 7, pp. 32–39, 2016.

[39] J. Ordonez-Lucena, P. Ameigeiras, D. Lopez *et al.*, "Network slicing for 5g with sdn/nfv: Concepts, architectures, and challenges," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 80–87, 2017.

[40] F. Z. Yousaf, M. Bredel, S. Schaller *et al.*, "Nfv and sdnłkey technology enablers for 5g networks," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, pp. 2468–2478, 2017.

[41] D. Erickson, "The beacon openflow controller," in *Proc. of ACM HotSDN*, 2013.

[42] T. Koponen, M. Casado, N. Gude *et al.*, "Onix: A distributed control platform for large-scale production networks," in *Proc. of USENIX OSDI*, 2010.

[43] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in *Proc. of ACM HotSDN*, 2012.

[44] S. Jain, A. Kumar, S. Mandal *et al.*, "B4: Experience with a globally-deployed software defined wan," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[45] M. Yu, J. Rexford, M. J. Freedman *et al.*, "Scalable flow-based networking with difane," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 351–362, 2011.

[46] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," vol. 41, no. 4, pp. 254–265, 2011.

[47] T. Wang, F. Liu, J. Guo *et al.*, "Dynamic sdn controller assignment in data center networks: Stable matching with transfers," in *Proc. of IEEE INFOCOM*, 2016.

[48] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *Proc. of ACM HotSDN*, 2012.

[49] J. Liao, H. Sun, J. Wang *et al.*, "Density cluster based approach for controller placement problem in large-scale software defined networkings," *Computer Networks*, vol. 112, pp. 24–35, 2017.

[50] B. Görkemli, S. Tatlıcıoğlu, A. M. Tekalp, S. Civanlar, and E. Lokman, "Dynamic control plane for sdn at scale," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 12, pp. 2688–2701, 2018.

[51] X. Lyu, C. Ren, W. Ni *et al.*, "Multi-timescale decentralized online orchestration of software-defined networks," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 12, pp. 2716–2730, 2018.

[52] H. Xu, X.-Y. Li, L. Huang *et al.*, "Incremental deployment and throughput maximization routing for a hybrid sdn," *IEEE/ACM Transactions on Networking*, vol. 25, no. 3, pp. 1861–1875, 2017.

[53] G. Zhao, H. Xu, S. Chen *et al.*, "Deploying default paths by joint optimization of flow table and group table in sdns," in *Proc. of IEEE ICNP*, 2017.

[54] M. Huang, W. Liang, Z. Xu *et al.*, "Dynamic routing for network throughput maximization in software-defined networks," in *Proc. of IEEE INFOCOM*, 2016.