Towards multi-SDN services: Dangers of concurrent resource allocation from multiple providers

Guillaume Fraysse*†, Imen Grida Ben Yahia*
*Orange
France
Email: {guillaume.fraysse,imen.gridabenyahia}@orange.com

Jonathan Lejeune†, Pierre Sens†, Julien Sopena†
†Sorbonne Universités, UPMC Univ Paris 06
CNRS, Inria, LIP6
F-75005, Paris, France
Email: firstname.lastname@lip6.fr

***Abstract*—One of the Software-Defined Networking (SDN) promises is the programmability of networks through Application Programming Interfaces (APIs). Those APIs allow different users to access the network concurrently. Thus leading to the allocation of dedicated *critical resources* by a given Service Provider. In this paper we formalize the problem of *deadlocks*, and present a case where they occur.**

***Keywords*—SDN, Distributed Systems, multi-resource allocation problem, multi-SDN, drinking philosophers, deadlock**

I. Introduction

Cloud Computing offers abstractions of computing infrastructures through the exposition of APIs: Amazon EC2 API [**?**], Microsoft Azure [**?**], Google Cloud APIs [**?**] or OpenStack API [**?**] for Infrastructure as a Service (IaaS) Clouds. Likewise SDN offers abstraction of network resources and network services through a NorthBound API.

These APIs allow *users* to integrate the *services* exposed by *service providers* in their *applications*. Recently applications are composed of services from multi-Cloud providers while they used to integrate services from a single Cloud. SDN starts to follow a similar path. Indeed complex applications require to aggregate the services offered by multiple providers. Companies also consider adopting multiple providers for resilience to meet their Service Level Agreements (SLA).

SDN paradigm adds new levels of abstractions however for some requests resources are allocated by the provider (e.g. HW resources that are limited and not interchangeable). Thus some types of resources can then be considered critical resources. Complex network applications aggregate distributed resources across multiple service providers. In this case multiple users send concurrent requests to allocate critical resources from these providers and we find ourselves facing a well-known problem of distributed systems: the **drinking philosophers problem** [**?**]. This problem is characterized by the possible apparition of *deadlocks* which can lead to problems such as starvation.

In this paper we show that if nothing is anticipated to prevent this problem in the context of multiple SDN providers, deadlocks can occur when you have multiple users. We have set-up an experiment based on a SDN environment composed of the Open-Source Open Network Operating System (ONOS) controller and Open vSwitch switches. Simulations show the occurrence of deadlocks, and that the result of the allocation of resources is not satisfactory for any user in more that 25% of the cases when a resource is critical.

Next Section II will provide some background and related work about the allocation of resources in an SDN context. We will then propose in Section III a formulation of the problem statement for multi-user allocation of critical resources from multiple providers and the definition of critical resources. We then introduce a test case to prove the existence of the problem in Section IV as well as results confirming it. Last Section V introduces possible future work.

II. Related Work

In this section we present a twofold state of the art: first we present the background on SDN and secondly we present state of the art for resource allocation in the context of SDN.

A. Background on SDN

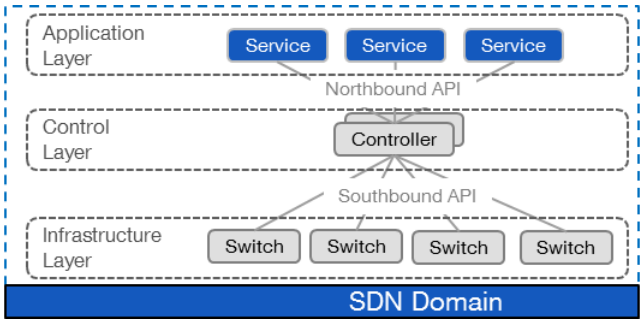

Fig. 1. The 3 layers of SDN

SDN emerged as a new architecture for networks. It is defined by the Open-Networking Foundation (ONF) in 2012 [**?**] as an architecture where *the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from the applications*. A SDN architecture as shown in Figure 1 is a three-layer architecture, from top to bottom:

- the Application layer running the Business Applications on top of the architecture,
- the Control layer or *Control Plane* which includes the SDN controller,

- the Infrastructure layer or *Data Plane* which includes the SDN capable devices, i.e. switches.

SDN introduced two types of APIs : NorthBound APIs and SouthBound APIs. On one hand, the Control Plane communicates with the Data Plane using a **SouthBound API**. Several research work considered SouthBound APIs [**?**]. SDN seminal paper focused on OpenFlow which is a protocol for SouthBound API. Other protocols like ForCES [**?**] are also available.

On the other hand, the Control plane **NorthBound API** enable clients to program Network infrastructures in new ways or in new kind of applications. SDN controllers each have their own APIs (ONOS [**?**] for example propose a Java API and a REST API). Some work has also been done to propose programming language for the development of Network applications on top of SDN such as Frenetic [**?**] or Pyretic [**?**].

Wickboldt et al. [**?**] also propose to add to this SDN architecture a cross-layer *management plane* for the implementation of Operations, Administration and Management (OAM) functions.

Traditional network configurations are static and are handled by specialized network administrators. The move towards APIs is particularly to avoid static configurations and enable programmability. The pitfall is how to handle the number of users accessing the networks those APIs concurrently.

B. Resource allocation in SDN

In this section we describe the variety of resource allocation research in SDN. Due to concerns in scalability of the architecture if a large number of resources need to be allocated, several works in the recent years (Onix [**?**], ONOS [**?**], Kandoo [**?**], DISCO [**?**]) promoted architectures based on *distributed controllers* instead of monolithic ones. Onix and ONOS, focused on a distributed controller for a single SDN domain, *i.e.*, a centralized logic that is deployed on a distributed platform. The open-source OpenDaylight [**?**] SDN controller also proposes a distributed architecture. Distributed architectures introduced new potential issues linked to concurrency, if users request updates of the network configuration to different controllers. To avoid inconsistency of the distributed system, Onix and ONOS in their introducing papers [**?**], [**?**] both relied on Apache Zookeeper to elect the master controller. Since this initial work, the architecture of the Open-source project ONOS has evolved and now uses the RAFT consensus algorithm proposed by Ongaro et al. [**?**]. Dang et al. [**?**] worked on two implementations of the Paxos consensus algorithm specifically targeted for SDN switches [**?**], even one which had no impact on the OpenFlow API but required to make assumptions about how the network orders the messages. All are solutions to the consensus problem faced by a distributed controller that need to be in a consistent state but do not address multiple providers architecture nor potential deadlocks.

Another kind of architecture that has led to several studies is an architecture where large SDN networks are divided in

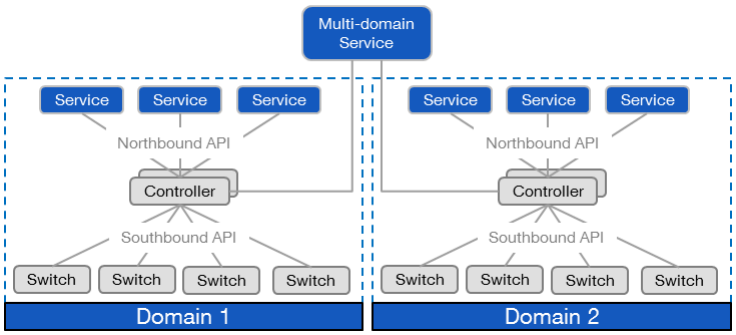

Fig. 2. A multi-domain SDN architecture

multiple *domains*. In their 2015 survey of SDN Xia et al. [**?**] use the following definition based on previous earlier works: *A SDN domain is defined as the portion of the network being managed by a particular SDN controller*. Figure 2 showcases differences between single and multi-domain SDN architectures. Inside of these domains, the controllers themselves can be based on different architectures : centralized, distributed or hybrid. In 2014 Phemius et al. introduced DISCO (for *(DIstributed SDN Controller)* [**?**] a distributed SDN controller targeted at WANs and overlay networks. It is based on the Floodlight OpenFlow controller. Their controller is composed of two parts: an intra-domain part which have the traditional capabilities of a SDN controller and an inter-domain part which manages the east-west communications with other DISCO controllers based on the AMQP protocol. They considered a per-domain organization where each controller was in charge of one SDN domain. DISCO does not rely on all controllers having a consistent network-wide state. Their solution adapts to network congestions between controllers by selecting a more viable route to avoid having control traffic add even more traffic. Kandoo introduced by Hassas et al. in 2012 [**?**] has a different approach, it relies on a hierarchical distribution of two layers of controllers. A root layer that manages the network-wide view and a local layer that has no knowledge of the overall network and no interconnection. None of these solutions address the case of one user composing services offered by multiple providers as a provider that has a multi-domain SDN still expose a single network to its users. Katsalis et al. [**?**] propose a multi-domain orchestration architecture which can be used as an enabler for Network Slices, a concept target for 5G mobile networks and listed challenges for multi-domain for Network Functions Virtualization (NFV) and composition of services as an open problems.

Ferguson et al. [**?**] addressed the problem of having multiple users (they used the term 'authors') trying to update the network configuration of a SDN domain, they focused on problems linked to the overlapping of those requests. They studied situations where one user can send a request that is in total contradiction with what another user might want, for example security rules where one user want to open a specific port and another want to restrict access to this port. On this

problem Canini et al. [**?**] introduced the concept of *Software Transactional Networking* (STN) and proposed two algorithms to manage the consistent installation of overlapping or concurrent policies when using a distributed SDN Controller. This is another kind of concurrency problem but different from potential deadlocks.

With LazyCtrl, Zheng et al. [**?**] also consider that the controller can be exposed to a large number of requests and propose a architecture that relates to the inter-domain composition. This architecture is composed of a central controller that communicates on groups of switches where distributed control modules are deployed to limit the load to which the controller is exposed. But they do not address concurrent access to the critical resources.

Finally, in [**?**] Yeganeh et al. analyzed SDN architecture and showed no specific bottleneck has been introduced by SDN compared to legacy networks. But bottlenecks already present in traditional network equipment remain present. Memory in particular is often the bottleneck in network equipment, rules are typically stored in Ternary Content Addressable Memory (TCAM) [**?**], a purpose-specific kind of memory which can search its entire content in a single operation and is by essence limited.

III. Proposed formulation of the problem statement

A. Definitions and assumptions about the system

A *user* is an application or a person (e.g. system administrator) that uses NorthBound APIs to interact with the SDN service providers.

We consider a context where an application requires to allocate resources from multiple SDN Service Providers. A SDN *Service Provider(SP)* offers one or more SDN networks as services through the exposition of an API. This can be the NorthBound API of a SDN controller or any higher-level API that acts as a proxy to the controller NorthBound API.

However, in SDN a deadlock can not happen on any kind of resources, it can only happen for *critical resources* which are resources that have the following properties:

- ***unshareable****: a critical resource can be used by only one user at a given time*
- ***not preemptable****: only the user who has allocated a critical resource can release it*
- ***not interchangeable****: no other resource from another component of the system can be allocated to obtain the same result,*

B. The dining and drinking philosophers problems

Handling deadlocks is a textbook problem from Computer Science. The base problem is now known as the **dining philosophers** problem, a metaphor first introduced by Dijkstra as a student exercise in 1971 [**?**]. In this metaphor, the users are philosophers who are sitting at a round table and the resources are forks. Each philosopher shares with his two neighbors a fork. To start eating a philosopher needs two forks. A *deadlock* can happens when two philosophers each waits for the other to put back the fork he has used on the table so he can start eating. When no synchronization mechanism is in place deadlocks can appear, leading to starvation.

In 1984, this was generalized to the allocation of multiple resources by Chandy and Misra who coined it the **drinking philosophers** problem [**?**]. In this problem multiple users (the *philosophers*) can request concurrently multiple resources (the *bottles*). The set of required resources/bottles may change at each request.

C. Problem definition

We reuse this metaphor to illustrate the problem in our multiple providers context. Considering multiple *users* who want to allocate multiple *critical resources* that could come from different SPs, the users play the role of the philosophers and the resources of the SPs the role of the bottles from the original metaphor.

Specifically, we consider that multiple SPs have no prior knowledge of each others and cannot share information. This is different from a multi-domain context where each domain is a part of a larger network of a single SP. This prohibits direct interaction between the two SPs. The assumption is that only the user has knowledge of both SPs. We don't make any assumption on the architecture of the SDN controllers themselves: they can be either centralized (like NOX [**?**]) or distributed (like Onix [**?**] or ONOS [**?**]).

In this case, concurrent allocation of critical resources from multiple SPs is similar to the drinking philosophers problem. Deadlocks may occur when multiple philosophers/users want to allocate non-disjoint sets of required critical resources/bottles. SPs have no knowledge of each other, users don't know each other. SPs and users can not synchronize to avoid the deadlocks.

Figure 3 illustrates a use-case where two users, Alice and Bob. Each user wants to allocate one critical resource from each of the SPs of a same set of two SPs α and β. They may face deadlocks. Depending on how their requests are processed, four results are possible.

1) Alice can allocate resource from both SPs α and β, Bob cannot allocate any
2) Alice can allocate one resource from SP β and Bob one from α
3) symmetrically to the previous case, Alice can allocate one resource from SP α and Bob one from SP β
4) symmetrically to the first case, Bob can allocate resource from both SPs α and β, Alice cannot allocate any

IV. Experimental proof of the problem

We are proposing in this paper a simple use case in order to better illustrate the problem through a basic experimentation. We consider two users, Alice and Bob, who each want to request the creation of two flows. One flow in each of the two distinct SDN networks of SPs α and β. Their expectations is that their applications hosted by these two SPs can communicate once the network is configured.

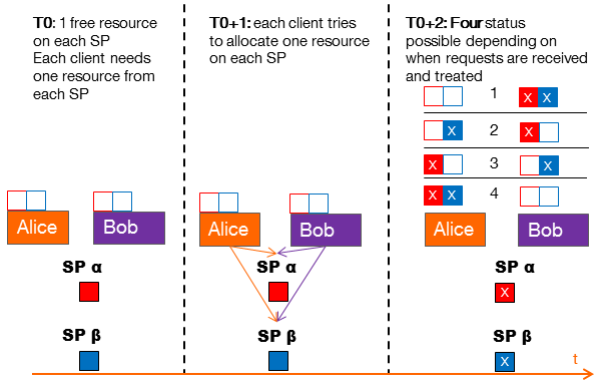

Fig. 3. Illustration of the concurrent allocation of two critical resources from two SDN SPs: 4 results are possible

These two SPs are each simulated by one instance of the open-source ONOS controller (version 1.10.3, running in one Virtual Machine (VM) based on Linux Debian 9 with 1 vCPU, 2 Go RAM) and one virtual switch (based on Open vSwitch 2.26, running in a dedicated VM running Debian 9 with 1 vCPU and 256 Mo RAM). Each ONOS controller is connected to its virtual switch using the OpenFlow 1.3 protocol as the SouthBound API.

The expected result is that once the flows in each SP are set-up, App1a (resp. App1b) the application of Alice (resp. Bob) which runs in Domain α is able to send messages to App2a (resp. App2b) the application of Alice (resp. Bob) which runs in Domain β. They are able to communicate because flows will be created between the appropriate ports of the switches. Those applications are Python applications that send ICMP ECHO messages (App1a,b) echoed on the standard output by App2a,b. For this Alice (resp. Bob) requests the creation of a first flow between Pa1 and Pa2 (resp. Pb1 and Pb2) to SP α. It then requests a second flow between Pa5 and Pa6 (resp. Pb5 and Pb6) to SP β. Tunnels are pre-configured for each user to allow direct communication between the two SPs: between Pa3 and Pa4 for Alice, between Pb3 and Pb4 for Bob.

On a fifth VM (running Debian 9 with 1 vCPU and 256 Mo RAM), Alice and Bob are simulated by Bash and Python scripts that use the NorthBound REST API of ONOS to request the creation of flows on the controllers. Each script requests the creation of a single flow by each SP. An umbrella Bash script runs the Alice and Bob scripts in a uniform random order. Random timers are also introduced in the scripts to ensure that requests are not send in the same order every time. This allows to simulate the concurrent allocation of resources by multiple users in varying conditions.

Memory on a SDN switch is the critical resource in this case. It limits the maximum number of flows that can be created on each switch. In the test environment it has been set very low to allow the creation of a single flow on each switch (using the *flow_limit* parameter in Open vSwitch). This resource satisfies the three properties we listed in the definition for a critical resource in section III. It is unshareable as these flows are between two ports of a SDN switch, and each port is dedicated to one user. It is not preemptable as only the user who requested one flow can remove it, no other user can remove it for security reasons. Most importantly it is not interchangeable, the user requires each of the flows to be set-up between two specific ports, the flow can not be created between others ports as the result would not be the one expected.

One simulation consist on running the umbrella script, which in turn runs the two Alice and Bob scripts. After each simulation the configurations of the switches are reset.

All the VMs are hosted by the same computer (Core i5-5300U 2.3GHz CPUs with 2 cores/4 threads, and 16GB of RAM) running the VirtualBox hypervisor version 5.1.16r11.38.41 under Windows 7.

The architecture of the test-case is represented in Figure 4.

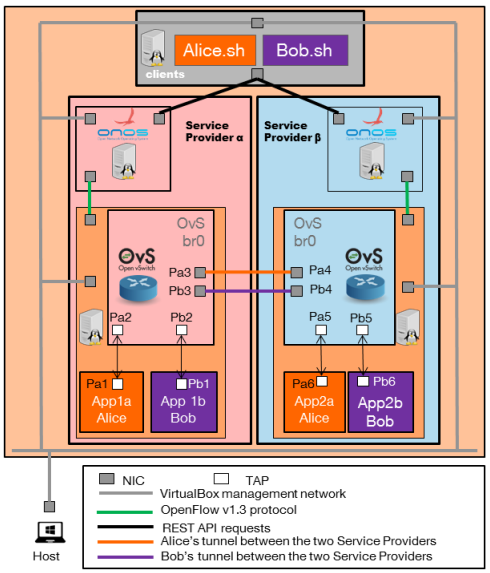

Fig. 4. Sample test case: 2 authors Alice and Bob and 2 SDN SPs

When we run the Alice and Bob scripts concurrently, as introduced in section III, four outcomes are expected. In all cases this is a deadlock: at best only one user gets the two resources it requires and the other is left waiting. Worst cases are the cases labeled 2 and 3, when none of the user gets all the resources required.

We ran the simulation 1000 times, the results we observed are shown in Table I.

Result	Occurrence
Alice gets none, Bob gets two resources	33.8%
Alice and Bob gets one resource each (either from α or β	26.8%
Alice gets two resources, Bob gets none	39.4%

TABLE I
SAMPLE RESULT OBSERVED WHEN SIMULATION IS RUN 1000 TIMES

We observe in this run that 26.8% of the time each script/user/philosopher only gets one of the two resources it

is expecting. Alice got both resources 39.4% of the time and Bob 33.8% of the time. This differs from the theoretical results in section III, where it is expected that each user gets both resources 25% of the time, and in the remaining 50% they each get only one resource. It is possible to obtain a result closer to the theoretical 25% apparition of the four results but this would require to coordinate the users. This shows that the results are very difficult to anticipate. They depend on when each user sends his requests, when these requests are received and how they are managed by the SPs.

As expected due to the random factors, running this simulation multiple times leads to slightly different numbers. However this result is sufficient to show that without synchronization mechanisms deadlocks may occur when multiple users allocate concurrently critical resources from multiple SPs. It also shows that a significant number of times, in this run 26.8%, none of the user can move forward as it only managed to allocate one of the two critical resources he wanted.

V. Future Work

In this paper, we formalized and showcased the problem of deadlocks. The testbed illustrates that they occur in the concurrent allocation of critical resources from multiple SDN SPs.

Towards a solution to this problem, we will study in more depth in which situations these deadlocks can occur in SDN and NFV environment. In a short term perspective, we plan to apply the algorithm proposed by Lejeune et al. in 2015 [**?**] while taking into account the challenges that may arise such as the latency caused by the algorithm, the QoS to be fulfilled to the users as well as the adaptation of the algorithm to SDN and NFV environment.

References

[1] Web Services Amazon, Inc. Amazon Elastic Compute Cloud. http://docs.aws.amazon.com/AWSEC2/latest/APIReference/, 2017.

[2] Bryan Lamos, Andy Pasic, Jim Wells, and Huang Xueyuan. Azure REST API Reference. https://docs.microsoft.com/en-us/rest/api/, 2017.

[3] Google. Google Cloud APIs — Cloud APIs. https://cloud.google.com/apis/docs/overview, 2017.

[4] OpenStack. OpenStack Docs: OpenStack API Documentation. https://developer.openstack.org/api-guide/quick-start/, 2017.

[5] K. M. Chandy and J. Misra. The Drinking Philosophers Problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, October 1984.

[6] ONF. Software-Defined Networking: The New Norm for Networks. Technical report, Open Networking Foundation, April 2012.

[7] B. A. A. Nunes, M. Mendonca, X. N. Nguyen, K. Obraczka, and T. Turletti. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *IEEE Communications Surveys Tutorials*, 16(3):1617–1634, Third 2014.

[8] Hormuzd Khosravi and Todd A. Anderson. Requirements for Separation of IP Control and Forwarding. https://tools.ietf.org/html/rfc3654, 2003.

[9] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. ACM.

[10] Nate Foster, Michael J. Freedman, Rob Harrison, Jennifer Rexford, Matthew L. Meola, and David Walker. Frenetic: A High-level Language for OpenFlow Networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, pages 6:1–6:6, New York, NY, USA, 2010. ACM.

[11] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software Defined Networks — USENIX. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/monsanto, 2013.

[12] J. A. Wickboldt, W. P. De Jesus, P. H. Isolani, C. B. Both, J. Rochol, and L. Z. Granville. Software-defined networking: Management requirements and challenges. *IEEE Communications Magazine*, 53(1):278–285, January 2015.

[13] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 351–364, Berkeley, CA, USA, 2010. USENIX Association.

[14] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 19–24, New York, NY, USA, 2012. ACM.

[15] K. Phemius, M. Bouet, and J. Leguay. DISCO: Distributed multi-domain SDN controllers. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–4, May 2014.

[16] The OpenDaylight Project, Inc. OpenDaylight - Technical Overview. http://archive15.opendaylight.org/project/technical-overview, 2013.

[17] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[18] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at Network Speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 5:1–5:7, New York, NY, USA, 2015. ACM.

[19] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie. A Survey on Software-Defined Networking. *IEEE Communications Surveys Tutorials*, 17(1):27–51, Firstquarter 2015.

[20] K. Katsalis, N. Nikaein, and A. Edmonds. Multi-Domain Orchestration for NFV: Challenges and Research Directions. In *2016 15th International Conference on Ubiquitous Computing and Communications and 2016 International Symposium on Cyberspace and Security (IUCC-CSS)*, pages 189–195, December 2016.

[21] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 327–338, New York, NY, USA, 2013. ACM.

[22] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. A distributed and robust SDN control plane for transactional network updates. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 190–198, April 2015.

[23] K. Zheng, L. Wang, B. Yang, Y. Sun, and S. Uhlig. LazyCtrl: A Scalable Hybrid Network Control Plane Design for Cloud Data Centers. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):115–127, January 2017.

[24] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali. On scalability of software-defined networking. *IEEE Communications Magazine*, 51(2):136–141, February 2013.

[25] T. Mizrahi, O. Rottenstreich, and Y. Moses. TimeFlip: Scheduling network updates with timestamp-based TCAM ranges. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 2551–2559, April 2015.

[26] Edsger W. Dijkstra. The Origin of Concurrent Programming. pages 198–227. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[27] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[28] J. Lejeune, L. Arantes, J. Sopena, and P. Sens. Reducing Synchronization Cost in Distributed Multi-resource Allocation Problem. In *2015 44th International Conference on Parallel Processing*, pages 540–549, September 2015.