Technische Universität München
Fakultät für Informatik

Scalable Transactional Software Defined Networking

Maja Ćurić

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades einer

Doktorin der Naturwissenschaften

genehmigten Dissertation.

Vorsitzende: Prof. Dr. Jana G. Makreshanska

Prüfer der Dissertation:

1. Prof. Dr.-Ing. Georg Carle
2. Priv.-Doz. Dr.-Ing. Roland Bless

Die Dissertation wurde am 11.05.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 28.10.2021 angenommen.

Scalable Transactional Software Defined Networking

Maja Ćurić

Acknowledgments

I would like to express my deepest gratitude to all the people who have supported me on this journey in many different ways.

First of all I want to thank my supervisor Prof. Dr. Georg Carle for his support and guidance during this doctorate. I would also like to thank PD Dr. Roland Bless for being my second examiner and Prof. Dr. Jana Giceva for chairing the examination committee.

I also want to express my great gratitude to my supervisors at Huawei European Research Center in Munich, Dr. Zoran Despotović and Dr. Artur Hecker, who helped me at any time and who were always available for productive discussions. I would like to thank all the colleagues from Huawei ERC that I had the pleasure to meet or collaborate with during my time there.

My warmest thanks goes to my parents for their continuous support throughout the many years of my education and my husband Sandin for his patience and encouragement. My son Arian, who was born while I was doing my doctorate, has been a great source of strength for me. The support of my family was crucial in accomplishing this thesis.

Munich, March 2021

Maja Ćurić

Abstract

The software defined networking paradigm has power to transform the networking. By offering programmatic control over network devices, it paves the way for event-driven automated network management. To become a technology that network operators readily embrace to solve their problems, the SDN must demonstrate that it reduces network administration efforts and that its programming interface supports straightforward development of network control logic, without sacrificing the performance. We have found that coordination of concurrent network updates in SDN is one of the key challenges to overcome in achieving these goals.

In this thesis we investigate the possibility of introducing transactional-like properties for network updates and an integrated distributed concurrency control scheme in SDN. For each proposal made in this thesis, we provide open source extensions to the state of the art SDN controller (Floodlight) and the SDN switch (OVS) for the others to use and build upon.

We first demonstrated how SDN paradigm introduces novelties in networking. We implemented one specific network functionality as SDN application and demonstrated how easily it outperforms its legacy counterpart, by taking advantage of the SDN´s dynamic programmability. However, we found out that developing SDN applications is not a straightforward task. Network updates in SDN are asynchronous and, when multiple applications operate in parallel, a broad range of concurrency exceptions can occur, e.g. partial execution or mutually conflicting resulting configurations, which the application developer must address.

Next, to tackle coordination requirements, we propose and implement transactional SDN, a novel SDN architecture that parallels that of the database management systems. It adds a transaction manager (TM) in the SDN controller and a resource manager (RM) in the SDN switch. The TM enforces global serializability of the updates using Atomic Commit Protocol and provides a simple API, which the SDN applications use to profit from the transactional update support. Our RMs implement resource locking as a concurrency control mechanism. We show that transactional SDN performs well, is easy to implement and does not require any complex provisions on switches.

Finally, we built on the idea behind transactional SDN, namely conflict-free transactional access to the switches, and designed and implemented FitSDN, a novel distributed SDN controller, which does not replicate critical network state in the controller instances, but instead uses the state from the switches directly. Besides streamlining the application development with its transactional API, FitSDN minimizes the interdependence between the instances and thus streamlines the administration as well. The performance analysis show that FitSDN controller supports linear performance improvement with additional compute resources, while outperforming state of the art distributed controller ONOS in all settings. The price to pay for these gains is the increased complexity of the SDN switch, since it must host the RM. While RMs can implement a range of concurrency control mechanisms that trade performance for complexity, our implementation shows that our current choice in which RM combines validation protocol and locking is a simple extension that is almost negligible in runtime and achieves satisfactory performance at the same time.

Zusammenfassung

Das Paradigma der softwaredefinierten Vernetzung (engl. software defined networking, kurz: SDN) transformiert traditionelle Netzwerke. SDN führt die programmatische Steuerung von Netzwerkelementen ein und ebnet auf diese Weise den Weg für ein ereignisgesteuertes automatisiertes Netzwerkmanagement. Um eine Technologie zu werden, die Netzwerkbetreiber zur Lösung ihrer Probleme bereitwillig nutzen, muss SDN nachweisen, dass es den Netzwerkadministrationaufwand reduziert und dass seine Programmierschnittstelle die einfache Entwicklung der Steuerungslogik unterstützt, ohne die Leistung zu beeinträchtigen. Wir haben festgestellt, dass die Koordination gleichzeitiger Netzwerkaktualisierungen in SDN eine der wichtigsten Herausforderungen darstellt, die bei der Erreichung dieser Ziele zu bewältigen ist.

In dieser Dissertation untersuchen wir die Möglichkeit, das Transaktionskonzept für Netzwerkaktualisierungen und eine integrierte, verteilte Nebenläufigkeitssteuerung in SDN einzuführen. Für jeden in dieser Dissertation gemachten Vorschlag bieten wir Open-Source-Erweiterungen für den State of the Art SDN-Controller (Floodlight) und den SDN-Switch (OVS), damit andere Forscher diese verwenden und darauf aufbauen können.

Wir haben zunächst demonstriert, wie das SDN-Paradigma Neuheiten einführt. Wir haben eine bestimmte Netzwerkfunktionalität als SDN-Anwendung implementiert und damit gezeigt, wie leicht diese, mithilfe der dynamischen Programmierbarkeit von SDN, deren Gegenstück aus einem herkömmlichen Netzwerk übertrifft. Wir haben jedoch festgestellt, dass die Entwicklung von SDN-Anwendungen keine einfache Aufgabe ist. Netzwerkaktualisierungen in SDN sind asynchron und, wenn mehrere Anwendungen parallel arbeiten, können Wettlaufsituationen auftreten, z.B. eine teilweise Ausführung von Steuerungslogik oder gegenseitig widersprüchlich resultierende Konfigurationen. In SDN muss der Anwendungsentwickler diese erkennen und bearbeiten.

Um die Koordinierungsanforderungen zu erfüllen, schlagen wir als nächstes Transaktions-SDN vor und implementieren es. Das ist eine neuartige SDN-Architektur, die dem Datenbankmanagementsystem entspricht. Es fügt den Transaktionsmanager (TM) im SDN-Controller und den Ressourcenmanager (RM) im SDN-Switch hinzu. Das TM erzwingt die globale serielle Ausführung der Netzwerkaktualisierungen mithilfe des Commit-Protokolls und bietet eine einfache API, mit der die SDN-Anwendungen von der Transaktionsunterstützung profitieren. Unsere RMs implementieren die Lock-basierte Nebenläufigkeitssteuerung. Wir zeigen, dass Transaktions-SDN eine gute Leistung erbringt, einfach zu implementieren ist und keine komplexen Erweiterungen für Switches erfordert.

Schließlich bauten wir auf der Idee hinter dem Transaktions-SDN auf, nämlich konfliktfreier Transaktionszugriff auf die SDN-Switches, und entwurfen und implementierten FitSDN. Das ist ein neuartiger verteilter SDN-Controller, der den kritischen Netzwerkstatus in den Controller-Instanzen nicht repliziert, sondern den Status der Switches direkt verwendet. Neben dessen, dass es mithilfe der Transaktions-API die Anwendungsentwicklung erleichtert, minimiert FitSDN die gegenseitige Abhängigkeit zwischen den Instanzen und optimiert somit auch die Verwaltung. Die Leistungsanalyse zeigt, dass durch Zufügen von zusätzlichen Rechenressourcen in FitSDN eine lineare Leistungsverbesserung erzielt wird. Gleichzeitig übertrifft FitSDN den State of the

Art verteilten SDN-Controller ONOS in allen Einstellungen. Der Preis für diese Gewinne ist die erhöhte Komplexität des SDN-Switches, da dieser den RM hosten muss. Während RMs verschiedene Nebenläufigkeitssteuerungsmechanismen implementieren können, die Leistung gegen Komplexität tauschen, zeigt unsere Implementierung, dass unsere derzeitige Wahl, die Validierungsprotokoll und Sperren kombiniert, eine einfache Erweiterung von SDN-Switch ist, die zur Laufzeit nahezu vernachlässigbar ist und gleichzeitig eine zufriedenstellende Leistung erzielt.

Contents

Chapter 1

Introduction

1.1 Introduction

The legacy communication networks have become complex due to the co-existence of various vendor specific equipment and proprietary protocols of multiple generations. The maintenance and scaling of such networks and their services, as well as introduction of new services require manual intervention, which has become incredibly difficult [74, 91, 71, 94]. In addition, the heterogeneity of equipment and protocols hinders innovations in networking and experimenting with new control logic in realistic settings [90, 94]. This motivated the researchers to consider reorganization of network functionality and evolve rigid and cumbersome legacy networks into more flexible and programmable infrastructures. This resulted with proposals such as active networking [130, 131], D4 Project [115, 52], NETCONF [61] and Ethane [21], which all laid the foundation for what later became Software Defined Networking (SDN).

The legacy networks have two networking planes, which are coupled together in the network elements. The first is the data plane, which contains all functions and processes that are used to treat the user traffic (e.g. forward, block or shape). This includes all rules installed locally in network elements. The second is the control plane, which contains all functions and processes that are used to compute and install the rules that make up the data plane, such as distributed protocols and manual configuration.

The idea behind SDN is to decouple data and control planes and implement an open interface between them. In this process, all control plane functionalities are striped down from network elements, i.e. the *SDN switches*. By doing this, they become simple programmable forwarding devices. The SDN switches establish the control channel with

the *SDN controller*. The logically centralized omniscient controller provides a high-level network abstraction, called *network state*. It comprises the overall information about the network infrastructure, i.e. SDN switches, links and hosts in the network, as well as the data and control plane configuration of the network, i.e. the rules for treating the data and control traffic installed in all operational switches. The network state is created and managed by the controller's *core services*, which populate the internal data structures with the most recent information from the network. Besides, the core services implement basic network functions, such as path computation or resource allocation.

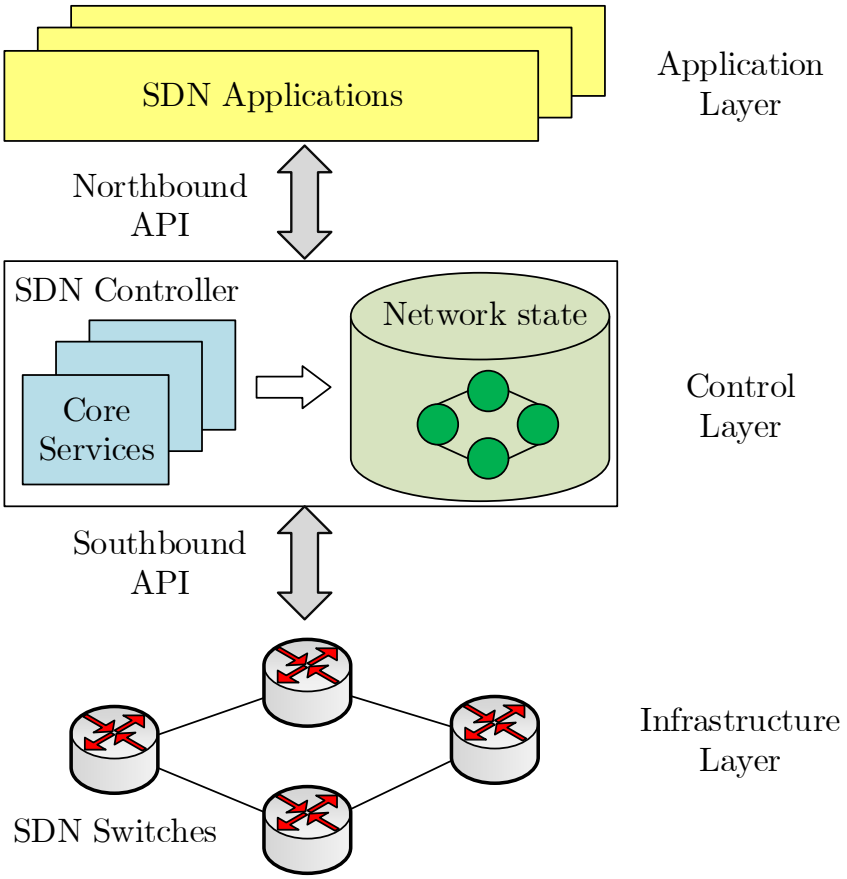

Figure 1.1: The SDN Architecture.

The control plane functionalities in SDN are implemented in a form of the *SDN applications*. They use the network state and the basic network functions provided by the core services to implement more complex control logic, such as load balancers or link failure recovery. The SDN applications program the SDN switches and instruct them how the traffic should be handled. Until now various SDN applications have been implemented, such as network virtualization [4], network orchestration [5], intelligent

service provisioning [129], elastic load balancing [136], intrusion detection [3], DoS and DDos protection [31] and IP address management [17].

The SDN controller's *southbound interface* is a collection of methods that handle communication between the controller and the SDN switches. The most commonly used southbound interface is OpenFlow [90]. It introduces *flow* as a basic unit of network traffic. The flow is a sequence of packets, identified with *flow match*, which is a combination of Layer-2 to Layer-4 header fields, the ingress port and the metadata value. OpenFlow is based on a match-action paradigm. The forwarding behavior of the switch is defined by the flow rules installed by the controller. A flow rule essentially couples the flow match and action (e.g. drop, forward via port, modify). A collection of flow rules installed in an OpenFlow-enabled switch is called *flow table*. In short, the SDN switches use OpenFlow to report the local network events to the SDN controller, while the SDN controller uses OpenFlow to program the switches according to the instructions of the control applications, i.e. by installing new flow rules to their flow tables. Besides OpenFlow there are other options for the southbound interface in SDN such as Forwarding and Control Element Separation (ForCES) [18], OpenState [16] or Cisco's proprietary OpFlex [24].

The controller's *northbound interface* is a collection of API calls, which the SDN applications use to read the network state, call the basic network functions (which are implemented by the core services) and update the network according to their control logic. The northbound API is essentially a software ecosystem, which is why it did not see as many standardization efforts as the southbound interface [74, 54, 81, 108]. Instead, the existing implementations of the SDN controller offer their own ad-hoc northbound APIs.

As a summary, the Figure 1.1 shows the tree layers in the SDN architecture: the infrastructure layer with the unintelligent programmable network infrastructure, the control layer with the SDN controller and the application layer with the SDN applications. The data and control plane separation enables independent evolution of forwarding and control solutions. It also simplifies network management and policy enforcement [14, 110]. By leveraging the centralized network view and dynamic network programmability, the SDN control plane functionalities can automatically reconfigure the network, optimize its utilization and improve traffic management. The SDN is, therefore, considered a key technological enabler for networks of the future, such as 5G, which must efficiently and in a flexible manner utilize limited network assets to offer ultra-high capacities and support expansion in the number of terminals and traffic volume.

The SDN operates as follows. The switches send notifications about network events, such as link failure, table-miss event or statistics thresholds exceeded, over the control channel to their managing SDN controller(s). Upon reception of the network event notification, the SDN controller inspects the event type, based on which it dispatches the notification to the SDN applications that registered to receive it. For example, the application for link failure recovery might register to receive link failure notifications only, while the application for load balancing might register to receive notifications of both, link failure and exceeded statistics thresholds. Upon reception of the event notification, the SDN application calculates the control logic with respect to the network state and deploys it in the network. In OpenFlow-based SDN deployments, for example, the latter implies installation of new flow rules in the affected switches.

However, the event-based network updates in the current form in SDN are not very useful to developers. Indeed, in the state of the art SDN implementations, there is no built-in mechanism for concurrency control that can guarantee that the network state is not altered by one SDN application, while some other application tries to apply its changes [20, 120, 141]. Network events arrive at the controller in an uncoordinated fashion, and, dispatched to SDN applications, could result in mutually conflicting or contradictory statements, if applied without order. This can have a negative overall effect, resulting in either partial execution of the update, conflicting or simply incorrect cumulative policies activated in the switches. Without concurrency control mechanisms, the standard SDN implementations cannot detect concurrency conflicts and offer proper exception handling. Instead, this burden is transferred to the application developer [89]. This complicates the SDN application development and prevents the developers from focusing on the actual application logic. We see this as a serious problem that is hindering wider SDN deployment. In fact, our standpoint is that *convincing casual developers that SDN efficiently supports the development of applications, without limiting the underlying SDN model*, is one of the two goals that SDN must fulfill to become production ready.

The second goal is *to demonstrate to network administrators that SDN exhibits very good performance and that it is resilient, yet easy to deploy and to maintain, including on larger scales.* It has been already demonstrated that physically centralized controllers cannot meet the requirements of large operational networks due to low failure resilience [71, 21, 121] and low scalability [69, 124, 142, 122]. It is, therefore, considered that a distributed controller design is a key to push the SDN to the next level of maturity. The SDN researchers are embracing the idea that a highly scalable and resilient distributed SDN controller must run as a set of instances, that these instances can be added or removed without disrupting the system and that they should work together to create

what appears to the applications as a single, logically centralized controller [15, 72, 133, 104, 56, 47, 87].

Until today, several distributed controllers have been proposed. In these proposals the network state is replicated across the controller instances, which is why their operation requires coordination to ensure the network state consistency. We divide the proposals into two camps based on the level and type of interaction among the controller instances. The proponents of the tight coupling, such as ONOS [15] and Onix [72], advocate using strong consistency protocols, e.g. Raft [101] and Paxos [79], to coordinate the operation of instances. These require coordination prior to performing any action in the network and are, as such, pessimistic, i.e. try to avoid problems. The other camp advocates loosely coupled systems, such as HyperFlow [133] and SCL [104], in which controller instances run mechanisms for eventual consistency, such as publish/subscribe messaging [126] or gossip protocol [32].

We believe that both camps cannot cope with the above two challenges at the same time. The strong consistency camp fails to provide satisfactory controller performance and presents complicated systems that are hard to administer, while the eventual consistency camp makes the administration lighter but delegates additional concurrency problems to application developers: to get the expected behavior and performance, it requires developers to have profound understanding of distributed systems and distributed programming.

By introducing the built-in concurrency control mechanism in the SDN, we strive in this dissertation to push the state of the art of SDN towards fulfilling the two previously defined goals and to make a step towards its wider adoption. Our goal in this dissertation is to design a novel SDN architecture that will (1) expose meaningful programming abstractions to the developers, without imposing any specific assumptions regarding the operation of applications or the network and (2) achieve good performance and streamline administration.

1.2 Research Questions

We now present the research questions (RQs) that this thesis raises and attempts to answer. These are as follows:

- How does SDN improve flexibility of networks?
- How can SDN coordinate concurrent network-wide updates in a scalable manner and streamline application development?

- How can we solve the SDN dilemma and achieve a distributed SDN design that appeals to both SDN application developers and administrators?

Next we discuss each research question in detail and identify the challenges that must be addressed in order to answer them.

RQ1: How does SDN improve flexibility of networks?

Mobile Core Networks (MCN) represent a typical example where legacy network architecture, i.e. LTE Evolved Packet Core (EPC), is reaching its maximum capacity and lacks flexibility [68], mainly due to its "one model fits all" approach [117, 112, 57]. Some state of the art proposals suggest evolving the LTE EPC towards a more flexible network, by using SDN-based network as MCN and implementing its functionalities as SDN applications [86, 12, 140]. In such an environment, the dynamic programmability of the SDN enables design of the flexible control logic that should achieve more efficient resource utilization. In this research question we align our view with these proposals and choose MCN as a sample network. Our goal is to investigate the possibility to achieve more flexible and dynamic network services in the SDN-based MCN.

We now identify the challenge C1, which needs to be tackled to answer the research question RQ1:

Challenge C1: Investigate the possibility to implement a set of the procedures from LTE EPC in SDN-based MCN.

One of the key functionalities in EPC is Mobility Management (MM), which provides procedures for service continuity as the mobile devices move across the network. The MM procedures are organized in two subsets, Handoff Management (HM) and Location Management (LM). Recent studies demonstrated practical feasibility of the SDN-based HM (path switching) [86]. In this challenge our goal is to investigate the same for LM (paging and area tracking procedures). To tackle this challenge, we first analyze the operation of LM processes and the amount of signaling that they generate in the network, and identify the areas for improvement. We then investigate the feasibility of the SDN-based LM.

By tackling the challenge C1 we can answer the RQ1, in which we demonstrate the design and implementation details of the novel, customizable and flexible SDN-based LM. However, our study has revealed that the application development in SDN is not a straightforward task. The SDN application developer must foresee and address a broad range of concurrency issues in order to correctly install the control logic and avoid data plane inconsistencies. These findings have guided the further direction of this thesis.

RQ2: How can SDN coordinate concurrent network-wide updates in a scalable manner and streamline application development?

The SDN applications hosted on the SDN controller operate independently in runtime. The uncoordinated concurrent updates from two applications, which affect two intersecting sets of the switches, can easily lead to partially or mutually conflicting resulting configurations. Because of that, the developer must design and implement the update validation mechanism in each SDN application to detect and resolve any inconsistent network state that might have occurred due to concurrency. The goal of this research question is to investigate the possibility to design a built-in mechanism for coordination of concurrent network-wide updates in SDN, while giving programmers appropriate programming abstractions and performance. The latter means that the concurrency control must not become the major bottleneck to the scalability of the system and must allow as much concurrency as possible.

To answer this research question we need to tackle the following three challenges:

Challenge C2: Investigate semantics of conflicts that can occur when multiple control applications simultaneously update the SDN network.

We tackle this challenge by analyzing a simple network controlled by the controller that hosts two concrete SDN applications and identify the conflicts that can occur when the applications operate simultaneously. By drawing a parallel to the transactional updates in database management systems (DBMS), our goal is to formalize the requirements on the history of the network updates in SDN that guarantee conflict-free operation. In addition, we aim to establish the set of criteria that a mechanism for coordination of concurrent network-wide updates in SDN must meet and asses the state of the art proposals for concurrency control in SDN against them.

Challenge C3: Model a novel SDN architecture that avoids concurrency issues by design.

The concurrency control in SDN must be able to schedule the execution of concurrent updates issued by the applications, such that it guarantees conflict-free access to the shared resources, i.e. the configuration of the SDN switches. In tackling this challenge we first analyze the existing concurrency control mechanisms from the DBMS and examine their applicability in SDN. We then investigate the applicability of these mechanisms in the SDN architecture and placement options for their modules. Finally, we present the novel SDN architecture that supports concurrency control by its design and offers programming abstractions for conflict-free installation of the updates.

Challenge C4: Evaluate the performance of the novel SDN architecture when it uses pessimistic concurrency control.

The concurrency control mechanisms in DBMS support concurrent update requests over the shared resource in a more or less rigorous manner. Depending on this rigor, concurrency control can be pessimistic or optimistic, leading to different isolation levels, progressively accepting possible conflicts in favor of more concurrency and, hence, better performance. To study the limits of such coordination proposals in the SDN context, we choose the most rigorous concurrency control method with the highest isolation guarantees. In this research question we focus on understanding if this is limiting to the overall performance and to what degree.

After tackling the three challenges, we can answer the research question RQ2, by presenting the concrete implementation of the novel transactional SDN architecture and assessing it against the set of criteria put before the concurrency control mechanism in SDN.

RQ3: How can we solve the SDN dilemma and achieve a distributed SDN design that appeals to both SDN application developers and administrators?

The state of the art distributed controller designs replicate the network state across the controller instances and use mechanisms that enforce strong or eventual consistency of the replicas. The replication imposes the fundamental trade-off between consistency and latency (i.e. performance) [1, 147]: while the strong consistency impacts the performance of the system and complicates its administration, the weak consistency exposes potentially inconsistent network state to the application developer and complicates the application development. In the latter, the developer must address typical complications of distributed programming, such as synchronization, coordination and timing. In this research question our goal is to investigate the alternative methods to distribute the SDN controller and possibly avoid the described trade-off.

To answer this research question we need to tackle the following two challenges:

Challenge C5: Investigate the possibility to eliminate the replication of critical network state in controller instances.

We first survey the state of the art distributed controllers. We focus on understanding how they replicate the network state, e.g. which replication mechanisms they leverage and which consistency levels they achieve. We then identify parts of the network state that can operate under eventual consistency, as well as critical network state parts, over which the updates must be serialized, i.e. which require strong consistency. Finally, we investigate the possibility to eliminate the replication of the critical network state

and instead use the unique state obtained directly from the network (and not from the controller instances). As a result, we present a novel distributed SDN controller design built around the transactional access to the switches.

Challenge C6: Evaluate the performance of the novel distributed SDN controller and compare it to the state of the art controllers.

Finally, to be able to answer the research question RQ3, we must extensively evaluate our novel distributed controller. In this challenge we investigate how our controller performs under different arrival rates and network sizes in simulation and emulation environments.

After tackling the two challenges, we are able to answer the research question RQ3, by presenting the concrete implementation of the novel distributed SDN controller and analyzing whether it has the properties that make it appealing to both of its end users, the application developers and the administrators.

1.3 Key Contributions of This Thesis

In this section we emphasize the key contributions that emerged from answering the research question raised in the thesis. In Table 1.1 we provide mapping between the research questions, key contributions and the thesis chapters.

Research question	Key Contribution	Thesis chapter
RQ1	SDN-based Location Management	Chapter 2
RQ2	Transactional SDN	Chapter 4
RQ3	Distributed controller FitSDN	Chapter 6

Table 1.1: Linking key contributions to posed research questions and thesis chapters.

SDN-based Location Management. We design and implement SDN-based Location Management (LM) procedures (UE state management and paging) for MCN. Our solution is fully capable of bringing a UE in a connected state when necessary, for example, prior to delivering downlink data to it. We defined a new set of states for UE, which relate to its activity, and introduced dynamically configurable inactivity timers that regulate transitions between the states. This enables the operator to flexibly regulate the amount of paging (and thus the utilization of network resources), which is not the case in the LTE EPC, where the timers are statically configured. We integrate our LM with the existing SDN-based Handoff Management (HM) procedures from [86] to obtain the SDN-based Mobility Management (MM) application, which the operators can use to fully support mobility of users in the network. Additionally, the operator

can install several instances of the MM application, each configured to serve a specific user group, e.g. with specific traffic pattern and level of mobility. Therefore, our design goes beyond "one model fits all" approach, which characterizes the legacy MCN.

Transactional SDN. We propose a novel SDN architecture that supports coordination of network-wide updates by its design. It is embodied in a simple and clean architecture that parallels that of database management systems and involves a transaction manager (TM) running on the controller and a resource manager (RM) that runs on the switch. The TM ensures atomicity of the updates using Atomic Commit Protocol (ACP) and the RM implements the concurrency control mechanism which isolates the updates. The transactional SDN provides an API for installation of the updates with the transactional semantics, which would allow any running SDN application to apply changes to the network in parallel, without fearing otherwise typical race conditions. We implement the TM and the RM as extensions to a state of the art controller (Floodlight) and SDN switch (OVS) and make them publicly available. To study the limits of our proposal, we use resource locking as the most rigorous concurrency control method with the highest isolation guarantees in our RM. Our evaluation show that although the resource locking can cause rejection of the concurrent updates and hence deteriorate the overall rate of successfully installed updates, enforcing multiple update installation attempts or increasing network redundancy result in almost 100% successfully installed updates.

FitSDN. We design and implement a novel Flexible Integrated Transactional SDN (FitSDN) that achieves at the same time the ease of application development and the ease of administration. FitSDN eliminates the replication of critical network state in the controller instances and taps switch flow tables under transactional semantics instead. FitSDN enables simple and intuitive APIs that permit SDN application developers to focus on the actual application logic. To demonstrate both the feasibility and the ease of realization of our proposal, we provide open source extensions to the state of the art controller (Floodlight) and switch (OVS), which we successfully test in a Mininet environment. Our evaluation of FitSDN shows that it performs almost without any deterioration due to the controller distribution per se; it rather behaves as a set of single image controllers that share their load. Comparing FitSDN to a popular distributed controller ONOS, FitSDN achieves around 4 times faster responses to network events.

1.4 Organization of This Thesis

The structure of this thesis corresponds to the posed research questions.

In the first part of the thesis we focus on understanding how SDN improves flexibility of networks. In Chapter 2 we present our SDN application for *Location Management*

(LM) in SDN-based Mobile Core Network. We first analyzed the relevant research work that considers optimization of signaling load generated by the LM procedures. We then introduced the design and implementation details of our LM proposal and presented emulation results of the solution. Finally, we discussed how our LM can tailor the amount of generated signaling according to the operator´s needs by taking advantage of the centralized network view and dynamic configurability of SDN. This part provides answers to RQ1.

The next two chapters of this thesis present our work that is related to answering the second research question, namely streamlining SDN application development by introducing scalable concurrency control mechanism in the SDN model. In Chapter 3 we first provide the examples that motivate the need for coordination of network-wide updates in SDN. We then survey the state of the art proposals for coordination of updates in SDN. Finally, we formulate a set of criteria that the coordination mechanism must meet and evaluate the existing proposals against these criteria. In Chapter 4 we present *transactional SDN*, a novel SDN architecture that supports concurrency control by its design in a scalable manner. Our proposal is inspired by the transactional architecture of the database management systems (DBMS). After extensive analysis of transactional SDN, we give answers to RQ2 and its challenges.

In the last part of the thesis, we focus on answering the third research question. In Chapter 5 we survey the state of the art distributed SDN controllers. We then define a set of criteria, which a distributed controller must fulfill in order to make SDN a technology that the operators readily embrace to solve their problems, and assess the existing distributed controllers against these criteria. In Chapter 6 we investigate the possibility to distribute the SDN controller using the concept of transactional network-wide updates, introduced in the second part of the thesis. We built upon transactional SDN and present *Flexible Integrated Transactional SDN (FitSDN)*, a novel distributed SDN controller that eliminates the replication of critical network state in the instances and taps switch flow tables under a transactional semantics instead. FitSDN streamlines administration and enables simple and intuitive APIs that permit SDN application developers to focus on the actual application logic. We finally evaluate the performance of FitSDN, after which we can answer RQ3 and its challenges.

Finally, in the fourth part we summarize the results of our research questions and challenges and provide recommendations for the future work.

1.5 Publications in the Context of Thesis

Following papers have been published in the context of this thesis.

Part I - Increasing Network Flexibility with SDN

- Maja Sulovic, Clarissa Cassales Marquezan and Artur Hecker "*Towards Location Management in SDN-based MCN*". In 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM). IEEE, Year 2017, pp. 1097–1102, Lisbon, Portugal.
  Accepted version: https://doi.org/10.23919/INM.2017.7987443

Part II - Transactional Concurrency Control in SDN

- Maja Curic, Georg Carle, Zoran Despotovic, Ramin Khalili and Artur Hecker "*SDN on ACIDs*". In 2nd Workshop on Cloud-Assisted Networking (CAN 2017). Association for Computing Machinery, Year 2017, pp. 19–24, Seoul, South Korea.
  Accepted version: https://doi.org/10.1145/3155921.3155924
- Maja Curic, Zoran Despotovic, Artur Hecker and Georg Carle "*Transactional Network Updates in SDN*". In 2018 European Conference on Networks and Communications (EuCNC), Year 2018, pp. 203-208, Ljubljana, Slovenia.
  Accepted version: https://doi.org/10.1109/EuCNC.2018.8442793

Part III - Admin or Developer: Solving the SDN Dilemma

- Maja Curic, Zoran Despotovic, Artur Hecker and Georg Carle "*FitSDN: Flexible Integrated Transactional SDN*". In 44th Annual IEEE Local Computer Networks Symposium on Emerging Topics in Networking, LCN Symposium, Year 2019, pp. 1–9, Osnabrück, Germany.
  Accepted version: https://doi.org/10.1109/LCNSymposium47956.2019.9000677

Due to copyright restrictions, we emphasize that the above listed publications are all accepted versions and not published versions. The listed publications are reprinted with the permissions of the publisher.

The above listed publications are included in original form in the Appendix at the end of this thesis.

Part I

Increasing Network Flexibility using SDN

Chapter 2

Location Management in SDN-based MCN

This chapter is based on the publication "Towards Location Management in SDN-based MCN" by Maja Curic, Clarissa Cassales Marquezan and Artur Hecker, published at IM 2017 [127]. The discussion on the contribution of the publication is provided in the Section 2.6.

In this chapter we investigate the possibility to implement Location Management (LM), a native MCN functionality, as SDN application. Next, we investigate whether the SDN-based LM can control its resource utilization at runtime by taking advantage of the dynamic configurability of SDN and in this way outperform the traditional LM form LTE EPC.

This chapter is organized as follows. In Section 2.1 we briefly analyze Mobility Management in the EPC and its two components, Handoff Management (HM) and Location Management (LM). Section 2.2 introduces the design of SDN-based HM solution from [86], which we use as a baseline for our work. Section 2.3 presents the design of our SDN-based LM solution. In Section 2.4 we evaluated our SDN-based LM. We provide the overview of the related work in Section 2.5. We conclude the chapter in Section 2.6, list key contribution in Section 2.7 and provide the statement on the author's contributions in Section 2.8.

2.1 Mobility Management in MCN

In the traditional LTE EPC, the Mobility Management Entity (MME) and the base stations (eNodeB) are responsible for handling UE mobility. The MME performs two following functions:

- *Location management (LM)*, which provides procedures for tracking and discovering the attachment points of the UEs.
- *Handoff management (HM)*, which provides procedures for maintaining the connectivity of the UEs as they move across the network and change their attachment points.

UE State Management is a concept from LTE that is closely related to MM. It defines two states for UEs that are registered in the network. A UE with active radio connection is in *connected state*. When such a UE attaches to a new serving base station, the HM switches the path of its active connection, i.e. it performs *handover procedure*. Therefore, the MCN is informed about each new attachment point of the UEs in connected state. After certain inactivity period, the UE transits to *idle state*. This transition depends on a static inactivity timer, so called RRC Inactivity Timer, usually in the range of a few seconds to a few tens of seconds [23]. It is pre-configured in each eNodeB in the network [135]. Whenever an eNodeB detects a UE's inactivity, it signals this event to the MCN. On this event, the MME orders a partial release of the UE's data plane resources and changes its state from connected to idle. From this moment on, the MCN is not aware of the UE's precise attachment point. It is only aware of the tracking area (TA), a group of eNodeBs, within which the UE in state idle can roam without reporting on each attachment point update. Only if the UE attaches to an eNodeB that belongs to a different TA, it informs the MCN about this even via *TA update procedure*, specified by LM. TA update allows the MCN to track the approximate attachment point of UEs. To deliver downlink data to an idle UE, the LM performs *paging procedure* by flooding its latest TA with paging messages in order to retrieve its precise attachment point and bring the UE back to connected state. The LM procedures essentially ensure reachability of the UEs in the network, which is a prerequisite to deliver downlink data to them.

The transition of UEs to idle state offers several advantages: it enables more efficient use of the data plane resources, since these are partially released with each transition to idle state, and it conserves the UE's limited battery resources, as it is no longer actively connected to the network. However, switching the UE state from active to idle too quickly pays the price of too often triggered paging procedure, which can generate

significant amount of signaling traffic due to flooding. Therefore, the state switching essentially results in a tradeoff, in which reducing UE battery consumption and releasing the unused network resources on one side and reducing network signaling overhead on the other is mutually conflicting.

2.2 SDN-based Handoff Management

Marquezan et al. proposed SDN-based HM application in [86]. Their solution is solely based on SDN to handle changes at the MCN level and on the assumption that eNodeBs support the OpenFlow protocol. When a UE attaches to a new eNodeB, the eNodeB detects the new flow and sends a message to the controller, requesting the forwarding information for the flow, i.e. PACKET_IN. The controller and the HM application interpret this message as a handover request, retrieve the new path for the UE's data traffic and install flow rules in the switches along it. The authors call this mechanism *reactive handover*. The reactive handover cannot switch the path in the time that is shorter than the round trip between the switches and the controller. Besides, if the new path requires updates of multiple switches, the overall handover time can significantly increase [69]. To address this issue, the authors proposed another more sophisticated mechanism, called *proactive handover*, whose goal is to minimize interaction with the controller during handover. The proactive HM prepares the network for path switching in advance. Prior to handover event, it pre-installs flow rules across all potential new paths that the UE can follow. Once the handover occurs, the proactive HM pushes minimal update to the network to complete path switching. The proactive handover switches the path quicker, but generates more signaling.

The architecture of the HM application is depicted in Figure 2.1. It contains three modules:

- Flow Information Base (FIB), which stores information about all UE flows, deployed flow matches and the current UE attachment point.
- Mobile Service State Information Manager (MSSIM), which interprets a message from the OpenFlow switch (i.e. a PACKET_IN) against the data in the FIB to decide which event is associated with the message.
- Handoff Management (HM), which is in charge to set up a new path when a handover event occurs.

We adopted the SDN-based HM application from [86] as the baseline architecture for our LM application, in which we implement the UE state management and the paging procedure.

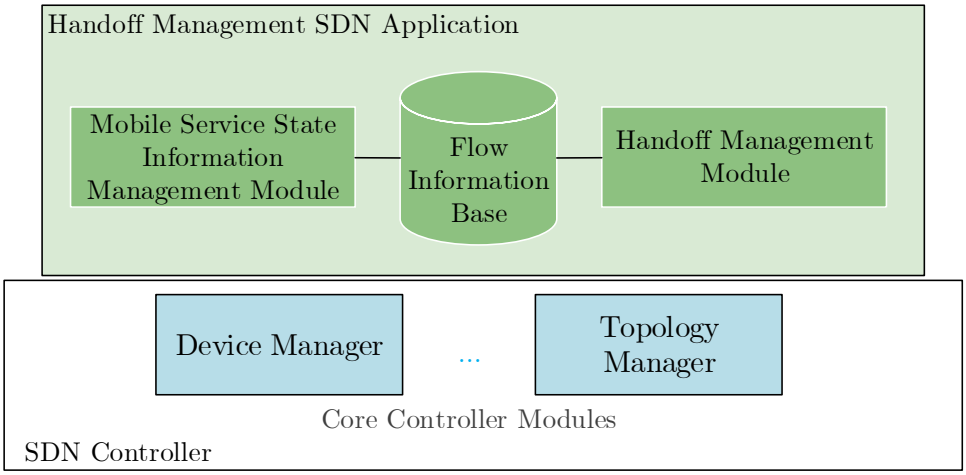

Figure 2.1: SDN-based Handoff Management Application Architecture.

2.3 SDN-based Location Management

LTE defines one state transition mechanism to fit all UE and application types in the network. This may not be optimal in future MCN, which should support the variety of use cases and massive amounts of mobile devices, as discussed in [119]. For example, the devices that produce infrequent small data bursts, such as smart meters and sensors, would stay too long in connected state and unnecessarily occupy the network resources. This would also unnecessarily waste the battery of such devices. On the other hand, too fast transition to idle state would cause signaling storms as each downlink data delivery for all other devices will trigger paging procedure.

In general, paging represents an important load in MCN: LM accounts for up to one third of overall LTE signaling traffic. More precisely TA updates account for 4.9% and paging represents more than 28% of the MME load [99].

In this section we would like to examine the possibility to implement SDN-based state management and paging procedure. Further, we would like to investigate whether they can be customized for specific device and application types, such that we can tune the trade-off between UE battery consumption and LM signaling for each type. For the former, our idea is to use the data available on the SDN controller to design a new set of UE states and inactivity timers. As for the latter, we want to take advantage of the programmability of SDN to allow dynamic updating of values of the UE inactivity timers.

2.3.1 Architecture

Our design leverages three mechanisms:

- Flow Information Base (FIB) from the previous work [86]. We extended it to store flow expiration timers that report on the flow activity.
- The native capabilities of spec-conform OpenFlow switches. OpenFlow defines the IDLE_TIMEOUT field in a flow rule as the period of time in seconds without packets matching the flow rule; after that time the flow rule expires and must be removed from the flow table. We set IDLE_TIMEOUT field of each flow rule to the value of previously mentioned flow expiration timer from FIB. A flow rule also has a flag field (i.e. OFPFF_SEND_FLOW_REM), which indicates to the OpenFlow switches to send flow removal notifications towards the SDN controller, using the message OFP_FLOW_EXPIRED.
- The SDN controller data about network devices. We extended this data to store information about the UE tracking area and UE state switching timers, which are essential for the paging procedure, as discussed below in Section 2.3.2. We combine the timers that control the UE states with flow (session) expiry timers and flow rule removal notifications to indicate when a UE needs to change its state.

We combine the modules proposed in [86] with two new modules: UE State Management and Paging module, shown in Figure 2.2. The former manages the state transitions for each UE, while the latter performs the paging procedure when necessary. We extended in our work the three modules from [86]: FIB, MSSIM and HM. We extended the MSSIM to identify the need to trigger paging when the destination UE is in idle state. The HM was extended to set up active flows with the flow expiry information from the FIB. One of the key features of our solution is the possibility to program both the timers that regulate flow expiration and the timers that regulate the UE state switching. Hence, our solution goes beyond the current design in LTE.

In our design, we assume a correctly dimensioned SDN layer. For approaches how to build this, see the comprehensive SDN survey [74]. Hence, in our design, as per [142], we make sure that the proposed SDN applications do not introduce global states. Indeed, in our design, any added state is strictly per UE, therefore allowing adding new application instances for (groups of) UEs as necessary, which solves scalability.

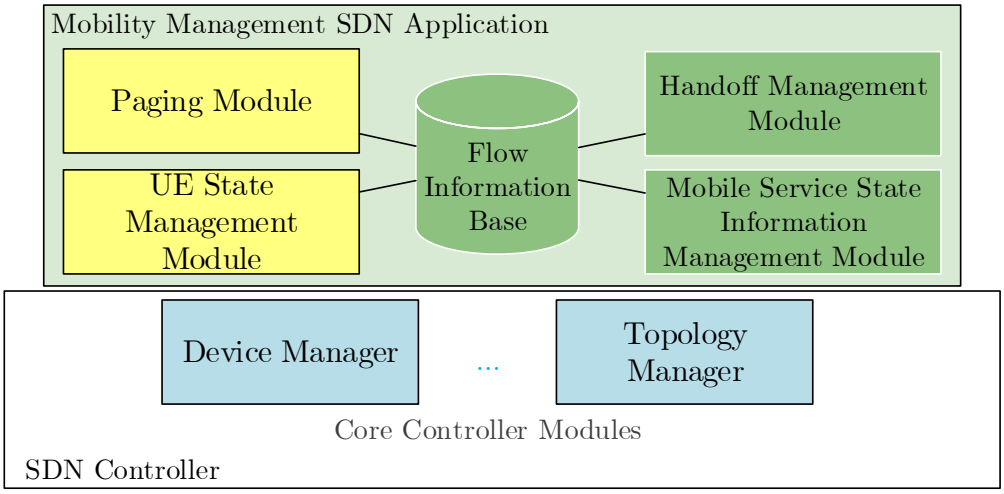

Figure 2.2: Proposed SDN-based Mobility Management (HM and LM) Application Architecture.

2.3.2 UE State Management

The proposed UE state management enables three states as illustrated in Figure 2.3: ACTIVE, IDLE or DEREGISTERED. UE with at least one active flow installed in the network is in state ACTIVE. Its access point is known to the accuracy of access switch and the port number. The UE State Management module registers to receive flow rule expiration notifications sent by the OpenFlow switches. Every time a flow expiry notification arrives, the UE State Management module marks the expired flow in FIB as inactive and check the remaining active flows of the source and destination UE. If there are no more active flows originating or terminating in these UEs (or in one of them), the UE State Management module starts the timer T_IDLE for that UE. If a new PACKET_IN message arrives, indicating new originating or terminating session on the UE, whose timer T_IDLE is counting, the UE State Management module stops the timer, resets its value and forwards the PACKET_IN for further processing, e.g. to the application that establishes flow path. Otherwise, if the timer T_IDLE expires, the UE transitions from ACTIVE to IDLE. The attachment point of a UE in IDLE state is considered to be known to the accuracy of the 5G TA (TA in LTE).

If a new PACKET_IN requires establishment of a new flow path towards a UE in IDLE state, i.e. whose precise location in the TA is not known, the MSSIM will interact with the Paging module to trigger paging and bring UE to ACTIVE state. Finally, a UE transits from IDLE state to DEREGISTERED, if its inactivity period exceeds the predefined period T_DEREGISTER ($\gg$T_IDLE), also controlled by the UE State Management module. Such UE is considered to be detached from the network and will

not be paged. A UE has to register to the network to change its state from DEREIGSTERED to ACTIVE.

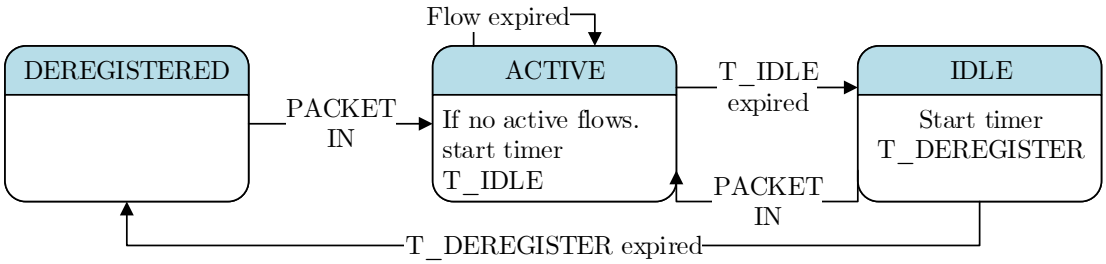

Figure 2.3: UE state transition diagram in SDN-based LM.

2.3.3 Paging Procedure

We now describe the SDN-based paging procedure in detail. Figure 2.4 depicts the mobile network with SDN-based MCN and RAN. In this specific example, the SDN controller controls the core switches CS1 to CS4, and access switches AS1 to AS4. There are two tracking areas: TA1 that includes access switches AS1 and AS2, and TA2 that includes access switches AS3 and AS4. We now analyze the example when Src UE initiates a mobile terminated (MT) voice session towards Dst UE.

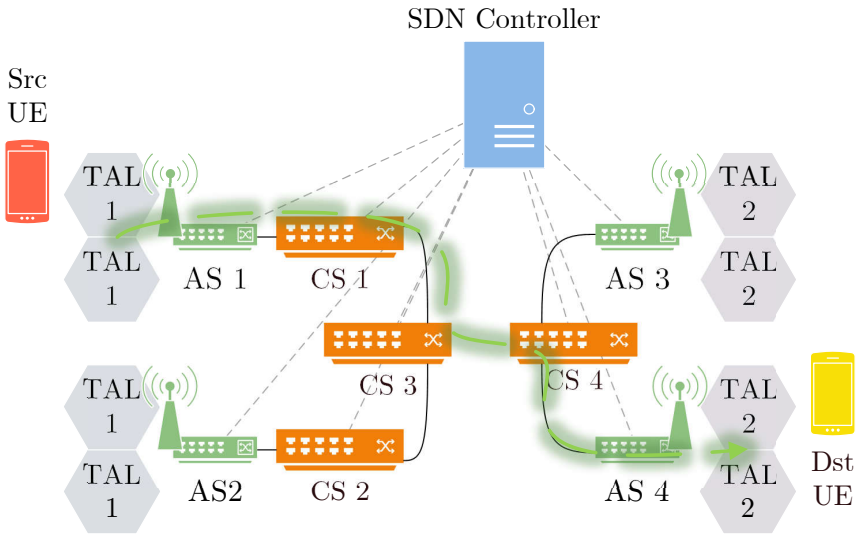

Figure 2.4: Example SDN-based MCN.

The steps for performing paging are illustrated in Figure 2.5. When Src UE starts data transmission towards Dst UE, the access switch AS1 matches incoming packets against

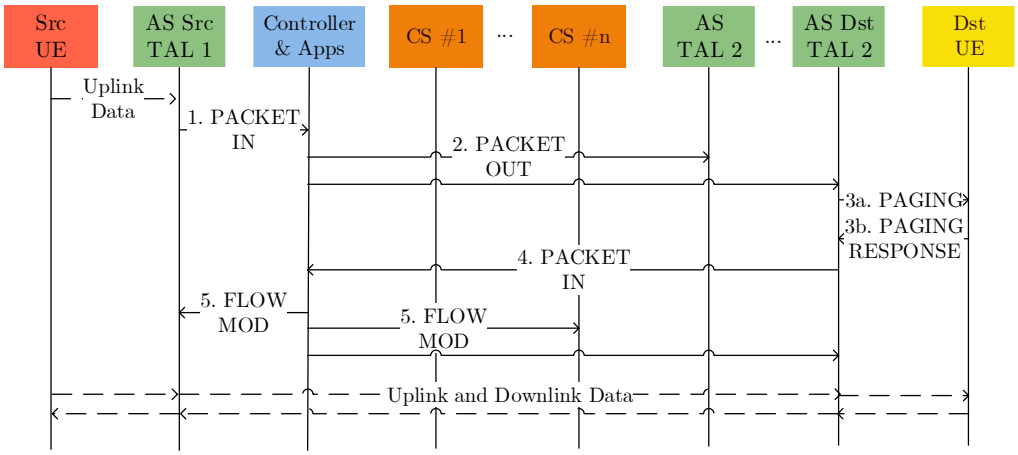

Figure 2.5: Call flow: Downlink data session establishment in SDN-based MCN.

the flow rules installed in its tables. Since no match is found for the packets, the switch sends PACKET_IN to the controller (step 1). The MSSIM module in the controller analyzes the PACKET_IN, reads the Dst UE from it and retrieves its state from the UE State Management module. At this point, we assume that the Dst UE is in state IDLE, which is why the MSSIM module triggers the paging procedure. The Paging module starts the paging procedure by sending PACKET_OUT messages (step 2) to all access switches that belong to the 5G TA of Dst UE, i.e. TA2. Each PACKET_OUT message encapsulates the paging message that targets the Dst UE. To ensure that the paging message reaches all devices registered in the 5G TA, we set the destination port in all PACKET_OUT messages to ALL. Upon reception of such PACKET_OUT, each access switch in 5G TA decapsulates the paging message and floods all ports with it, except the ingress port. This way, the paging message reaches all UEs registered in 5G TA (step 3.a), and only the one that is actually being paged sends paging response (step 3.b). When access switch receives the paging response message, it encapsulates it in PACKET_IN and sends towards the controller (step 4). The controller internally passes the message to MSSIM that recognizes this PACKET_IN as paging response and forwards the message to UE State Management module, which updates UE's current attachment point. At this point, the UE State Management module also changes the UE state back to ACTIVE and stops its T_DEREGISTERED timer. Once active, the SDN applications can establish flow paths towards the Dst UE (step 5).

2.3.4 Implementation

We implemented our solution using the open source Floodlight SDN controller (version 1.0). We emulated our solution on a Mininet platform extended to support mobility.

This platform has limitations related to the emulation of the radio signaling. For this reason, we could not implement the paging at the radio side. Instead, we created paging request and response messages that could be transmitted from the APs (which in our platform are OpenFlow switches). The paging request message is a UDP packet targeting a certain closed port at the UE in IDLE state. When the UE in IDLE state receives this UDP packet, it sends back a response message "ICMP UDP Port Unreachable". This message is the paging response in our solution. When AP (OpenFlow switch) receives paging response, it sends a PACKET_IN to the SDN controller. When this PACKET_IN arrives at the controller, the controller's core service Device Manager, which keeps track of the devices, will update its internal database with UE's attachment point and update last seen timestamp. The PACKET_IN message is further passed to the UE State Management module to update the UE state to ACTIVE and stop the T_DEREGISTERED timer.

2.4 Evaluation

In this section, our goal is to demonstrate the practical feasibility of the SDN-based LM approach. Note that we are not aiming to evaluate the LM procedures per se (i.e. paging). Instead, we pursue two goals: first, we want to know whether our SDN-based LM works as expected. Second, and more importantly, we want to show that the parameters of our proposal effectively tune the trade-off between UE battery usage and paging signaling and hence can regulate the amount of paging in the system. To do this, we study the impact of two parameters on the system paging: the frequency of PACKET_IN messages and the T_IDLE timer value. The evaluated metric is the percentage of downlink deliveries that triggered paging. The testbed for the evaluation is a two-layered (access and core) Mininet topology. Parameters used in the emulation are shown in Table 2.1.

Table 2.1: Paging emulation parameters.

Number of access switches	10
Number of core switches	50
Number of hosts per access switch	5
Number of sessions	250
Gamma distribution shape	2
Session duration	10s
Session expiry timer IDLE_TIMEOUT	5s
T_IDLE	5s

In our emulation, each host in the network starts communication with other randomly chosen host, whereby the time between two subsequent communication requests from one host is random and follows exponential distribution. Each new session request results with PACKET_IN on the SDN controller, so their arrival follows the same distribution. Since all hosts in the network communicate simultaneously and independently, the interarrival time of all PACKET_IN messages on the SDN controller (triggered by all subscribers) will be a sum of independent exponentially distributed random variables and, as such, it will follow a two-parameter gamma distribution (PDF and mean value as in Eq. 1 and Eq. 2, respectively). By varying the parameters of the gamma distribution, shape k and scale θ, it is possible to simulate more or less frequent arrival of PACKET_IN messages on the SDN controller. These interarrival times correspond to the different overall UE activity in the network, where less and more frequent arrival implies lower and higher overall activity of UEs, respectively.

$$f(x;k,\theta) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-\frac{x}{\theta}}\, x \geq 0\, and\, k,\theta > 0 \tag{2.1}$$

$$E[x] = k\theta \tag{2.2}$$

The goal of the first test is to determine, how the overall UE activity in the network impacts the amount of paging requests. In scenarios with increased UE activity (higher frequency of PACKET_IN arrivals), the probability that a randomly chosen host at a randomly chosen time has at least one active flow (i.e. it is in the ACTIVE state) will also be increased. Hence, the probability that such chosen UE will be paged before setting up the flow path for a received PACKET_IN is accordingly lower.

Figure 2.6 shows the results of the experiments for this scenario. We varied the load of the network and observed the percentage of paging triggered. The x-axis represents the average PACKET_IN interarrival time in the network and y-axis represents the percentage of PACKET_INs that triggered paging before the establishment of the flow path between the two communicating UEs. The first observation is that the higher the UE activity in the network, i.e. the lower the interarrival time, the lower is the amount of triggered paging. This is aligned with our expectations. For instance, when average interarrival time of PACKET_IN messages is set to 1s (shape 0.5 and scale 2), the percentage of paging is around 32%.

If the mean interarrival time is increased to 2s, the percentage of PACKET_INs that triggered paging almost doubles. The increase in the amount of paging remains, as we increase the interarrival time (which in its turn reduces the UE activity in the network);

nevertheless, this increase in the percentage has a lower trend of growth. The results show non-linear dependency on the overall UE activity, where the small difference, when there is high UE activity can significantly increase the amount of triggered paging requests, but the same variations with low UE activity does not generate the same increase in triggered paging requests.

We can conclude that the proposed solution can implement UE state aware paging procedure as in LTE today. However, the observed non-linear behavior confirms that an approach, where the timers can be programmed is able to tune the amount of signaling produced by the paging procedure.

The second test investigates the impact of timer T_IDLE value on the amount of paging requests. The tests have been carried out using three scenarios with different UE activity – the interarrival time between two subsequent PACKET_IN messages in the network was modeled using gamma(0.5, 2), gamma(0.75, 2) and gamma(1, 2), which corresponds to average interarrival time equal to 1s, 1.5s and 2s, respectively. Values used for the timer T_IDLE are 2, 5, 10, 20 and 30 seconds. The results are depicted in Figure 2.7.

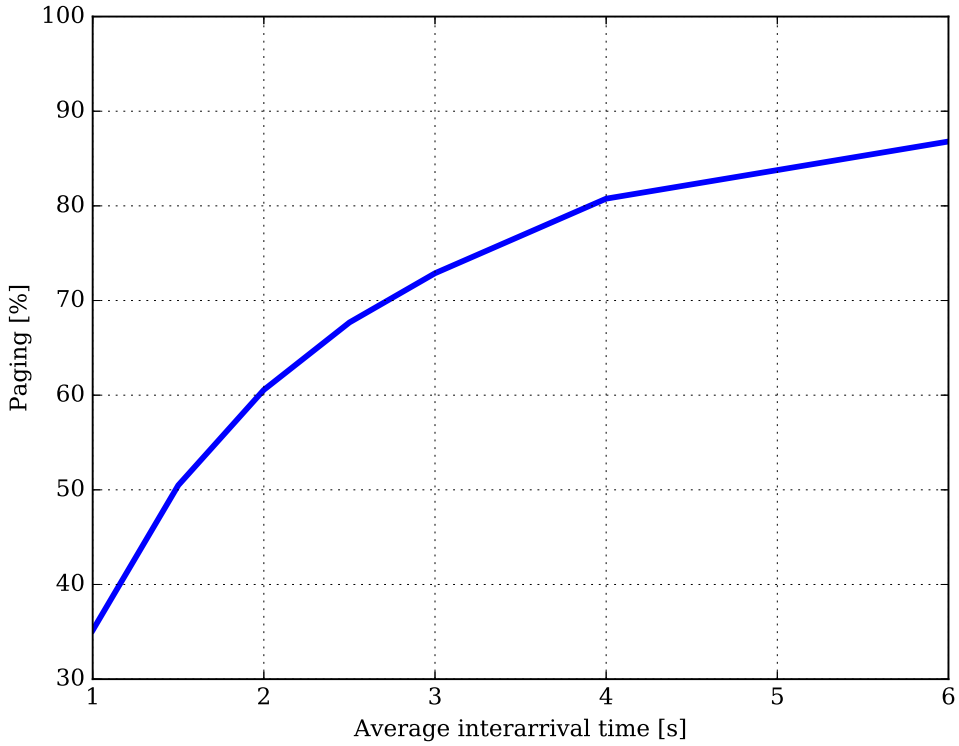

Figure 2.6: Percentage of paging depending on the network load.

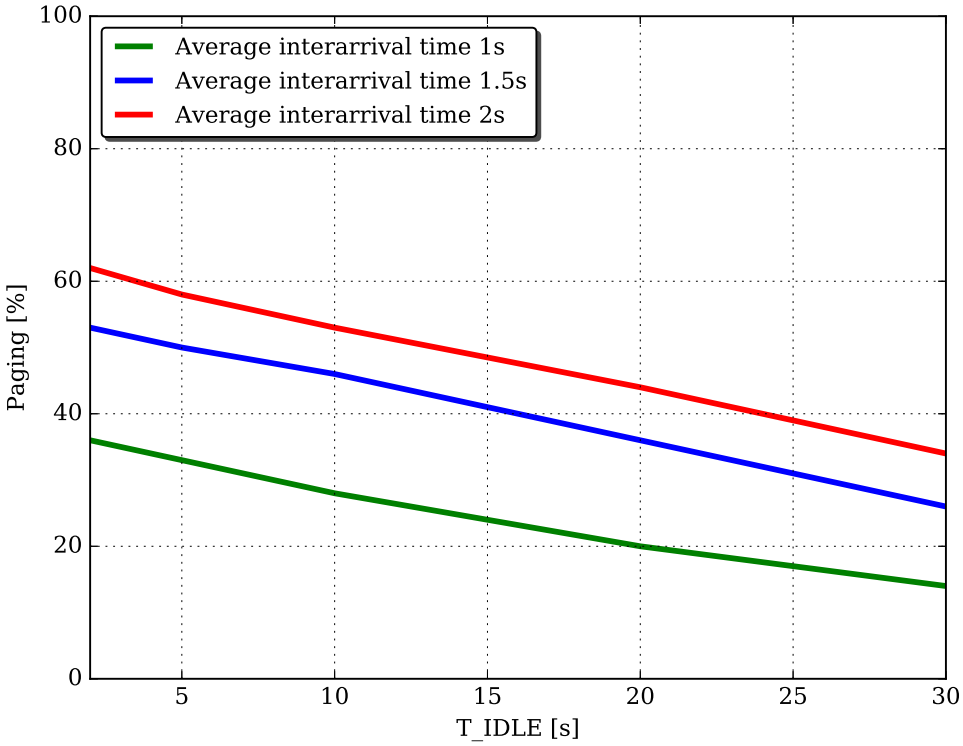

Figure 2.7: Paging percentage depending on the value of T_IDLE.

As expected, the increase of the inactivity timer reduces the amount of triggered paging requests. Moreover, there exists approximately linear relationship between T_IDLE timeout value and the percentage of PACKET_INs that triggered paging. When the T_IDLE value is increased of 5 seconds, we observe an average reduction of 4% in the amount of triggered paging requests in all tested scenarios.

This experiment shows that dynamic programmability of T_IDLE in the proposed approach unfolds the possibility to tune the trade-off between the UE battery consumption and the signaling load in the network. Our proposal enables using different T_IDLE and T_DEREGISTERED values for each UE in the network. Besides this, the flow expiration timers can also be customized and set differently according to application types. For example, for the applications that produce infrequent small data bursts, for which LTE is not optimized [98, 111], the flow expiration timers can be short, which will result with faster transition to idle state. Therefore, the trade-off can be tuned according to the needs of each type of device, application, or any other MCN operator requirement.

2.5 Related Work

Several authors advocated the need for optimization of MM signaling in future wireless networks. Bagaa et al. [11] proposed a framework for efficient tracking area list management (ETAM), that tunes the trade-off between tracking area update (TAU) and paging signaling messages. Arouk et al. [10] proposed an optimization of current 3GPP Group Paging (GP), which reduces the amount of signaling overhead. Similar ideas can be combined with our approach. Ali-Ahmad et al. [7] proposed distributed HM, but did not consider LM in their work. Sama et al. [119] is one of the few proposals to consider LM in SDN. They also consider the flow expiry mechanism of OpenFlow switches to determine whether a UE is in idle of connected states. However, this work does not specify how to page a UE in idle state.

Foddis et al. [42] present the analysis of the energy consumed by the UE and signaling traffic overhead in LTE for different values of the RRC Inactivity Timer. They propose a method for calculation of the RRC Inactivity Timer value based on the measurements of the data plane traffic. They show that it is possible to find the RRC Inactivity Timer value that saves UE energy and pays price of only low increase of signaling traffic overhead. Similar to this, Zhao et al. [146] present the tool for optimization of the RRC Inactivity Timer in LTE for individual users based on the user model. They suggest collecting information from cellular network in runtime and using machine learning method to predict the session length, which is used to dynamically adjust the timer value in eNodeB. Finally, their results show that optimization of the RRC Inactivity Timer can reduce the energy consumption of UEs by 33.5%. As these works are dedicated to determining the optimal value of the inactivity timer, they can be combined with our work.

2.6 Conclusion

In this chapter, we tackled the RQ1 and its challenge C1. By analyzing the LM procedures from LTE EPC, we identified that the LM signaling depends on a single static inactivity timer that is pre-configured in each eNodeB, generating in that way over 30% of the overall MM signaling in the MCN. We consider such capacity utilization to be unfavorable for future mobile networks. Finally, we demonstrated practical feasibility of the SDN-based application that implements LM procedures (UE state management and paging). We demonstrated that the SDN-based LM can tune the trade-off between UE battery usage and LM signaling amount in runtime and thus outperform its counterpart from the legacy LTE EPC.

2.7 Key Contributions of This Chapter

In the following we list the key contribution of this chapter.

SDN-based Location Management. The proposed LM is purely SDN-based. It uses only the data from the SDN controller, such as UE connectivity and flow information. It defines three states for the UE, connected, idle and disconnected, and UE inactivity timers to switch between the states. It uses the flow expiration timers to obtain the information about the flow activity in the data plane. By combining the two types of timers, we can customize the LM procedures to fit the specific UE types and applications.

After tackling the challenge C1 we can answer the research question RQ1. We conclude that SDN enables creation of new customizable functionalities to meet the needs of particular user groups and applications, rather than "one size fits all" models, which is a typical strategy in legacy networks. By leveraging the configurability of SDN, the network functionalities can be optimized to the user and application needs in runtime. In this way the SDN provides more flexible alternative to the legacy networks.

2.8 Statement on Author's Contributions

This chapter is based on the publications "Towards Location Management in SDN-based MCN" by Maja Curic, Clarissa Cassales Marquezan and Artur Hecker, published at IM 2017 [127].

The author contributed to the study presented in the IM paper and the adapted sections in this chapter. The author designed and implemented the LM application, namely UE state management and paging procedure, as well as designed and carried out the overall evaluation process. Finally, the author contributed to the discussion and analysis of the obtained results.

In the following we describe changes between the Section 3.1 and the study on SDN-based location management in [127]. Most of the analysis in the IM 2017 paper is present in this dissertation. We provide more details on our proposal, which had to be shortened in the constrained space of the IM 2017 paper. We omitted the parts that are not relevant for this dissertation, such as detailed analysis of LM in LTE EPC.

Part II

Transactional Concurrency Control in SDN

Chapter 3

Concurrent Network-wide Updates in SDN

In this chapter we analyze concurrency conflicts that occur when multiple SDN applications, which jointly control the network, operate in parallel. SDN does not have a built-in mechanism for coordination of concurrent updates and transfers the burden of addressing the concurrency issues on the SDN application developer. This increases the application development efforts and prevents the developers from focusing on the actual application logic.

This chapter is organized as follows. In Section 3.1 we analyze the semantics of the concurrency conflicts and their consequences. In Section 3.2 we survey the state of the art proposals related to coordination of concurrent network updates in the SDN. In Section 3.3 we define assumptions about the operational SDN environment and the requirements that an adequate mechanism for coordination must fulfill. We then assess the state of the art proposals against the defined requirements. We conclude this chapter in Section 3.4 and list its key contributions in Section 3.5.

3.1 Motivation Example

This section is based on the publication "Transactional Network Updates in SDN" by Maja Curic, Zoran Despotovic, Artur Hecker and Georg Carle, published at EUCNC 2018 [30]. The discussion on the contribution of the publication as well as the difference between it and the thesis text is provided in the Section 3.6.

We now provide two concrete examples of concurrency issues in the SDN. The example network from Figure 3.1 has eight switches (SW1 to SW8) and four user nodes (User 1 to User 4). It is controlled by a physically centralized controller with two SDN applications, the Green Application and the Blue Application. In the examples that follow we will give concrete meaning to the user nodes and the applications.

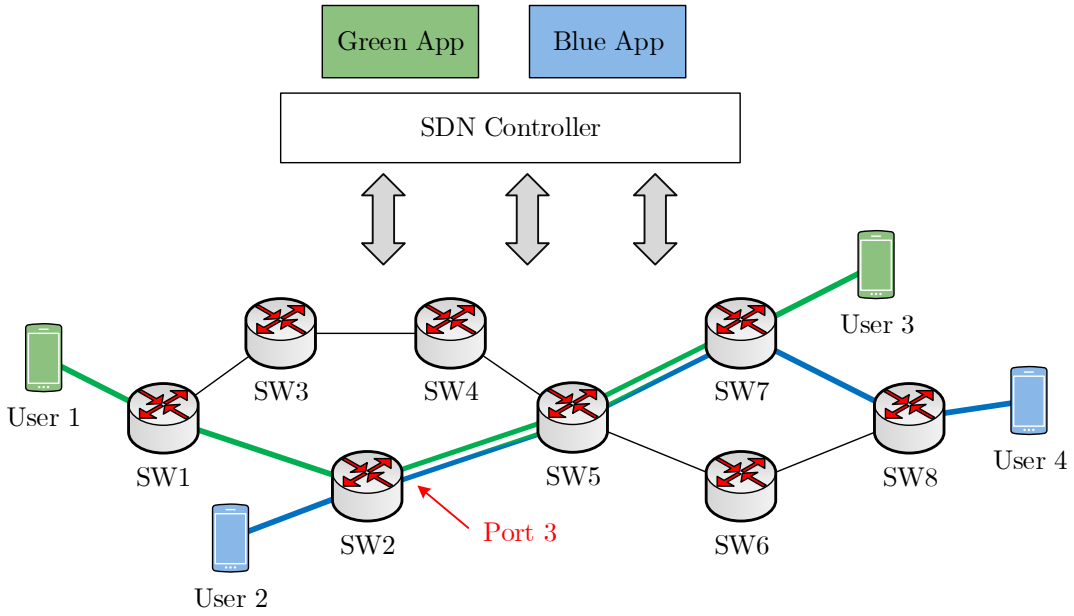

Figure 3.1: Example of non-atomic and uncoordinated network updates: The security application shuts down port 3 on switch SW5 at approximately the same time, when the MM application sets up a path that goes over that same port.

Example 1. Assume that the Green Application is the Mobility Management application, which includes reactive Handoff Management (HM) from [86] and our Location Management (LM) from Section 2.3. The reactive HM is in charge of path switching when a device moves in the network, while the LM provides procedures for device location tracking. Assume now that User 1 and User 3 are mobile devices with active connection between them. The Blue Application is a security application that monitors the traffic and takes appropriate protection measures whenever it detects sufficient evidence of a DDOS attack; such measures include blocking unauthorized flows and ports and possibly migrating valid flows to a new path. By definition of DDOS, we assume that it is not straightforward to determine and block the exact entry points of the DDOS packets. We also assume that the DDOS flows' characteristic pattern is not easy to determine either, which is why the Blue Application cannot simply block the precise flows by installing simple rules that drop the packets of the flows, but instead

blocks the entire ports. Now, imagine that User 1 performed the handover from SW3 and attached to SW1. Prior to this event, the path for the data packets between User 1 and User 3 was SW3 - SW4 - SW5 - SW7 (to keep the figure simple we do not show this path). The Green Application in response to handover event selects the path SW1 - SW2 - SW5 - SW7 as a new path from User 1 to User 3, shown in green. To install this path, the Green Application pushes new flow rules to SW1 and SW2. Besides, assume a DDOS attack has been reported from the upstream networks and by some customers, and the Blue Application, trying to mitigate the overall situation, after a traffic analysis, decides to close the entire Port 3 on SW2.

How severe is this conflict? Well, there is nothing that the Green Application itself cannot take care of. It can read the network state after it has pushed the update and verify if the flow rules for the traffic between User 1 and User 3 are installed in all switches along the path. It is rather annoying in the following sense. After the Green Application eventually realizes that there is a problem with SW2, it should select and set up another path, while performing rollback for the flow rules that were successfully installed during the previous attempt, i.e. flow rule in SW1. This, however, increases the application development effort: without any confirmation or indication of success or failure, a developer would have to use dedicated try-test-retry logic on every network-wide update, without actual guarantees of success; this is an additional burden, producing longer, possibly wildly varying, network update installation delays. A simple service that takes over this duty by providing atomic network-wide updates with "all or nothing" semantics would relieve the developers and fix the problem.

Example 2. Let us now consider a more severe example. First, think of the Green Application and the Blue Application as two SDN applications that compete for a given resource. That can be bandwidth of a link or even flow table of a switch. Note as well that in the latter case, the two can be literally any applications. What is more, the two can also be two instances of the same application: for example, if operator uses two instances of the SDN-based MM application, each for a different group of UEs. So let the bandwidth budget of all links in the network be 1.5 Mbps, while the two applications are about to reserve 1 Mbps paths each. If the reservations occur concurrently then the following situation is not improbable: one application has reserved the link SW2 - SW5, while the other one holds the link SW5 - SW7. So none of them can proceed, both will have to give up their reservations.

We ask the same question again, how severe is this? It looks indeed way more serious than the previous problem. It is an artefact of concurrency, which is inherent to any SDN deployment, centralized or distributed, with one or multiple controllers. Similar

to the previous example, the applications would have to validate the network state after each network update.

So the root problem here is the concurrency. In more concrete terms, we are dealing with an instance of the so called "inconsistent read", a well-known problem in the field of database transaction management. Let x and y denote the available bandwidth of the links SW2 - SW5 and SW5 - SW7, respectively. Then the description above is equivalent to the following pattern of interleaved read and write operations:

$$r_1(x)r_2(y)r_2(x)w_1(x)r_1(y)w_2(y)w_1(y)w_2(x),$$

where the subscripts denote the two applications. Reads do not happen "physically". They rather exist logically, as the applications tacitly assume that there is enough bandwidth. We argue that this schedule is not conflict serializable [48, 120], i.e. it cannot be equivalent to any serial schedule. Thus it must be eliminated by appropriate concurrency control mechanisms.

The SDN researchers have identified other examples of concurrency issues, such as partial update installations in [90], broken bandwidth guarantees [120, 93] or security violations where packets are bypassing firewalls and middleboxes [89, 88, 20].

3.2 Mechanisms for Coordination of Concurrent Updates in SDN

In this section we list the related work that tackles concurrency issues in SDN.

3.2.1 OpenFlow Bundle

OpenFlow v1.4 [90, 139] has introduced the concept of bundles. A bundle groups related state changes on a switch and executes them in an atomic manner and isolated from other concurrent updates. When a bundle is opened, the modifications from the bundle are saved in the staging area. The bundle is then pre-validated on switch and committed by the controller. Only when the bundle is committed, the changes from the staging area are applied to the switch state.

Bundles can also be used to better synchronize execution of network-wide updates in SDN. For example, the controller can open bundles on all switches affected by the update and add state update commands to each. Each switch validates its respective update and informs the controller about the validation outcome. If the controller receives successful validation response from each switch, it commits the bundles at approximately the same

time. This approach, however, cannot guarantee atomic execution of a network-wide update: if one of the switches fails to commit some command from the bundle, e.g. due to the resource unavailability, it discards the entire bundle, leading to partial update installation.

If multiple controllers control a switch, the switch maintains a separate staging area for each controller connection. This is called bundle parallelism. If two applications residing on two different controllers concurrently push network-wide updates to the overlapping sets of the switches, the same order of bundle commits on each affected switch cannot be guaranteed and previously discussed interleaved writes are likely to occur. Therefore, using bundles in this scenario cannot result with conflict serializable schedule.

3.2.2 Scheduled Bundles

Mizrahi et al. [93] introduced the concept of scheduled bundles (meanwhile incorporated in OpenFlow v1.5). A scheduled bundle can be committed at pre-determined time. The applications can push network-wide updates using scheduled bundles and specify distinct execution time for each. This approach enables updates to be fully separated in time and can guarantee conflict serializable schedule for both single switch and network-wide updates. However, it comes at the cost of perfect time synchronization between network elements, where any synchronization incorrectness can violate the serialization order and cause incorrect network state.

Enforcing conflict serializability using scheduled bundles in a network controlled by a distributed SDN controller requires the existence of a central entity to collect the updates from all the controller instances and schedule them appropriately. Such an entity becomes a bottleneck and a single point of failure in the system. Alternatively, to avoid using the central entity, the controller instances could agree upon the execution times of their updates using consensus protocols [101, 79]. However, such protocols produce a lot of overhead and do not scale well as the number of updates increases [34].

3.2.3 Transactional Middleware

Canini et al. [20] proposed transactional middleware for semantic composition of concurrent distributed network-wide updates. It receives the updates from distributed controller instances, serializes them and modifies if necessary, to prevent overriding the existing network configuration. The middleware installs the updates atomically and notifies the respective controller instance about the installation outcome.

As the transactional middleware processes the updates sequentially, it must be single-threaded. This impedes the benefits of the controller's distribution - although the

distributed controller instances process events in parallel, they all forward their updates to the central entity, i.e. transactional middleware, which processes them sequentially. Such solution does not scale well when the number of updates increases. Besides, the proposal does not provide technical implementation details or an analysis of the middleware design complexity.

3.2.4 Synchronization Framework for Coordination of Distributed Updates

Schiff et al. [120] proposed a synchronization framework for coordination of distributed updates, in which multiple controllers concurrently update a switch. They stamp the switch with the installed policy identifier and add primitives for reading and updating the stamp value. The latter is the Compare-and-Set (CAS) primitive, a fundamental atomic instruction typically used for synchronization in multithreading environments. An SDN application can update the switch, only if it knows the current value of the stamp - this guarantees that the switch state was not changed meanwhile by another controller.

The update works as follows. The application first reads the state of the switch and receives the read response that contains the current switch stamp value. It then composes a bundle with the stamp update primitive, which contains the received stamp value and the new stamp value, followed by the update commands. When a switch receives the bundle, it validates whether the stamp value specified therein corresponds to the current stamp value stored in the switch memory. If yes, the bundle is executed, i.e. the switching state and the stamp value are both updated. Otherwise, the whole bundle is rejected. This approach essentially prevents installing single switch updates that are based on stale reads.

In short, the framework proposes synchronization of distributed single switch updates directly on the switch, using its data plane configuration space as a shared memory. By avoiding using central entity for that purpose, it does not experience low scalability and the failure scope stays local. Also, the controller instances do not have to use additional out-of-band coordination mechanisms, e.g. consensus protocols. It remains however unclear, how to extend this approach to a network-wide update.

3.3 Assumptions and Requirements

This section is based on the publication “SDN on ACIDs” by Maja Curic, Georg Carle, Zoran Despotovic, Ramin Khalili, and Artur Hecker, published at CAN,

CoNEXT 2017 [28]. The discussion on the contribution of the publication as well as the difference between it and the thesis text is provided in the Section 3.6.

In this section we present and explain our assumptions about the operational environment and formulate requirements that a mechanism for coordination of concurrent network-wide updates in SDN must fulfill.

REQ 1.1: *Serialized transactional network updates.* The updates in SDN can vary from simple single switch updates to network-wide updates that affect several switches. We consider that SDN would benefit from the support for serialized and transactional updates, as this would relieve the developers from having to detect and solve unexpected network behavior that might occur due to concurrency, e.g. when commands from one update interleave with commands from the other. Serialization implies that two overlapping updates, i.e. occurring at the same time and targeting the same switches, have the same execution order on all affected switches. As such, they cannot result in unexpected effects, e.g. a flow forwarded over a part of the intended path and blocked on the rest. Transactional semantic implies that network-wide updates are executed according to the ACID properties.

REQ 1.2: *Good scalability in all deployment strategies.* The control plane can be centralized, such that a single physically centralized controller hosts a set of independent control applications, or distributed, such that it consists of several controller instances, that share the network load and provide redundancy in case of controller failures. For the latter, we do not prescribe any particular decentralization strategy. We generally assume that different SDN control applications can run on different available SDN controller instances. The switches can be assigned to the instances in any way, including also non-disjoint groups where one switch maintains connection to several controller instances. With this assumption we aim to achieve scalability and applicability, as it requires the update coordination mechanism to work in all typical deployments: with central unique controllers, decentralized hierarchical or replicated controllers as well as many other fully distributed setups.

REQ 1.3: *No time synchronization.* For reasons explained in Subsection 3.2.2, we relax the requirement for strict time synchronization of all resources in the network, including controllers and switches. Networks are often used for time synchronization and should be able to survive without it, especially during outages and for simpler bootstrapping. We believe that such requirements are too burdensome and unrealistic for large virtual/physical infrastructures. Besides, simple network devices often do not store complex notions of local time [25]. Note that similar arguments led to a non-reliance on external time sources during the design of SNMPv3 [135]. This assumption

means that we cannot rely on any "do this after that" type of protocols, as in practice, there are no guarantees on the message delivery order.

3.4 Conclusion

We now compare the state of the art proposals against the requirements defined in Section 3.3.

Only the proposals from Mizrahi [93] and Canini [20] can serialize updates that go beyond one single switch. They both introduce a central entity that intercepts the updates and serializes their execution (REQ 1.1). In contrast, OF Bundle and the synchronization framework from Schiff [120] are limited to single switch updates.

The proposals from Mizrahi [93] and Canini [20] funnel the updates from distributed controller instances to the central entity, which impedes the benefits of the controller's distribution. It becomes the single point of failure, faces scalability issues and requires additional administration efforts and configuration. We, therefore, consider only OF Bundle and the proposal from Schiff [120] to be scalable for all controller deployment models, since they synchronize the update over the switches (REQ 1.2).

The last requirement (REQ 1.3) is fulfilled by all proposals except the scheduled bundles from Mizrahi [93].

Table 3.1 summarizes the previous discussion.

REQ		OF Bundle [2013]	Canini [2013]	Mizrahi [2016]	Schiff [2016]
1.1	Serialized transactional network updates	×	✓	✓	×
1.2	Scalability in all deployments	✓	×	×	✓
1.3	No time synchronization	✓	✓	×	✓

Table 3.1: Assessment of the state of the art proposals for the coordination of concurrent network updates against the requirements REQ 1.1 to REQ 1.3.

3.5 Key Contributions of This Chapter

In this chapter we analyzed the research question RQ2 and its challenge C1. The main contribution is:

Analysis of conflicts due to concurrency in SDN. We analyze network updates in SDN and find that they are neither atomic nor isolated. As such, when performed

concurrently over a target set of switches, they can lead to partial or mutually conflicting resulting configurations. We observe the semantics of the conflicts that arise due to concurrency in SDN and come to the conclusion that they can be eliminated using concurrency control mechanisms, which would enforce conflict serializable history of updates. By formalizing the requirements on the history of network updates in SDN, as well as defining the set of requirements that a concurrency control mechanism in SDN must fulfill, we tackled the challenge C2.

3.6 Statement on Author's Contributions

This chapter is partially based on the publications "SDN on ACIDs" by Maja Curic, Georg Carle, Zoran Despotovic, Ramin Khalili, and Artur Hecker, published at CAN, CoNEXT 2017 [28] and "Transactional Network Updates in SDN" by Maja Curic, Zoran Despotovic, Artur Hecker and Georg Carle, published at EUCNC 2018 [30].

The author contributed to the studies presented in CAN CONEXT paper [28] and EUCNC paper [30] and the adapted sections in this chapter. The author contributed to the study design, implemented its proposal and carried out the overall evaluation process. Finally, the author contributed to the discussion and analysis of the obtained results.

In the following we describe changes between the Section 3.1 and the study on transactional SDN presented in [30]. We use the section Motivation from the paper [29] in this section. We adapted the examples to use the application developed in the previous chapter of this dissertation.

In the following we describe changes between the Section 3.3 and the study on ACID properties for SDN updates presented in [28]. We partially use the section Assumptions and Requirements from the paper [28]. We only used the parts that are relevant in the context of this dissertation and omit other parts, such as the discussion on resource granularity.

Chapter 4

Transactional Semantics for Updates in SDN

This chapter is based on the publications "SDN on ACIDs" by Maja Curic, Georg Carle, Zoran Despotovic, Ramin Khalili, and Artur Hecker, published at CAN, CoNEXT 2017 [28] and "Transactional Network Updates in SDN" by Maja Curic, Zoran Despotovic, Artur Hecker and Georg Carle, published at EUCNC 2018 [30]. The discussion on the contribution of the publications as well as the difference between the publications and the thesis text is provided in the Section 4.8.

In this chapter we propose transactional SDN, a novel SDN architecture capable of coordinating concurrent network-wide updates from any SDN application anywhere in the network in any typical SDN deployment scenario. Our proposal is inspired by the previous work from database management system (DBMS) [48, 138, 128]. Our standpoint is that SDN would provide a richer programming model with the support for transactional network update logic with ACID properties, where network-wide updates are:

- Atomic, meaning that the whole operation in the network either succeeds completely, or does not apply at all. Specifically, this requires that the network-wide updates, going beyond one single switch, either succeed completely or not at all.
- Consistent, meaning that the network-wide invariants and policies should continue to hold as the network state evolves. i.e. constraints imposed by those policies are never violated. For example, if a policy says that no forwarding loop is permitted, then every update should be checked against that. Consistency thus checks se-

mantics of individual network updates against the invariants, policies, constraints that should hold in the network.

- Isolated, meaning that two overlapping atomic operations, e.g. occurring at the same time and targeting the same switches, cannot result in unexpected effects. In essence, isolation eliminates inconsistencies that result from the concurrency as such and not from the semantics of individual operations. Isolation makes sure that no invalid resulting state becomes effective, e.g. a flow -forwarded over a part of the intended path and blocked on the rest.
- Durable, meaning that the effect of an atomic transaction survives indefinitely, unless it is changed by another transaction. As some SDN implementations (e.g. OpenFlow) explicitly support expiring flow rules, we extend this to "or unless the effect is explicitly meant to be temporary". Note that the expiration requirement here has to refer to the atomic transaction as a whole.

Of the above four, only consistency has received considerable attention from the research community so far [70, 67, 109]. We argue that atomicity, isolation and durability are equally important. Having general support for these properties would allow any independently running SDN application to apply changes to the network in parallel, without fearing typical race conditions, such as ordering conflicts and partial execution.

This chapter is structured as follows. In Section 4.1 we survey the related work from DBMS. In Section 4.2 we explain the design choices in our proposed system architecture and mechanisms. We detail on the architecture and implementation of our proposal in Section 4.3. In Section 4.4 we extensively evaluate transactional SDN. In Section 4.5 we provide the related work on transactional aspects in SDN. In Section 4.6 we conclude this chapter, by assessing our proposal against the requirements previously defined in Section 3.3. Finally, we list the key contributions of this chapter in Section 4.7 and provide the statement on the author's contributions in Section 4.8.

4.1 DBMS Related Work

4.1.1 Concurrency Control

In transactional DBMS, a Transaction Manager (TM) coordinates transactions that span multiple Resource Managers (RM). RMs implement a concurrency control mechanism to support concurrent change requests to the local resource in a more or less rigorous manner. Depending on this rigor, concurrency control can be pessimistic or optimistic [113] and yields different isolation levels, progressively accepting possible conflicts in favor of more concurrency and, hence, better performance.

4.1.1.1 Pessimistic Concurrency Control

The pessimistic concurrency control mechanisms use locks to grant a transaction exclusive access to the concerned data. As long as one transaction holds lock over the data, it cannot be accessed by another concurrent transaction. When a new transaction requests a lock on a data unit, the mechanism first checks if another transaction already holds the lock. If not, it grants the lock to it. Otherwise, it detects lock conflict. When a lock conflict occurs, the transaction experiences lock wait and its execution gets suspended. This essentially means queuing the transaction until the moment when the lock can be granted. A transaction can also request to release the lock on the data; upon each lock release the mechanism checks which of the suspended transactions can be resumed.

The pessimistic concurrency mechanisms experience deadlocks. For example, consider two concurrent transactions, say T1 and T2 affecting the same set of data units, say x and y. If the concurrency control mechanism grants to each of the two applications locks only on the subset of the data, e.g. T1 obtains the lock on the data unit x and T2 obtains the lock on the data unit y, then both transactions will be suspended. The transaction T1 cannot proceed until the transaction T2 finishes and releases its lock over y. Similarly, the transaction T2 cannot proceed until the transaction T1 finishes and releases its lock over x. In this case, none of the transactions can proceed. When a deadlock occurs, the pessimistic concurrency control mechanisms must employ one of the deadlock handling techniques. These are deadlock resolution, deadlock prevention and timeouts. In deadlock resolution the mechanism maintains the graph of all waiting (suspended) transactions and selects those whose aborting would end the deadlock, and aborts them. In the previous example, the mechanism can abort any of the two transactions. For example, if it aborts the transaction T1, the lock over x can now be granted to the transaction T2, whose execution can then proceed. For deadlock prevention the mechanism aborts the transaction before it results with the deadlock. In this case the mechanism maintains the graph of all transactions as they arrive and notes the data units that they affect. Prior to granting any locks, it checks if locking will result with the deadlock and discards the problematic transaction. In previous example, the mechanism would analyze the transactions T1 and T2, immediately abort one of them, say T2, and grant locks over x and y to the transaction T1. Finally, when using timeouts for deadlock resolution, the mechanism starts the timer for each transaction at the moment when it starts and aborts the suspended transaction when the timeout occurs. In our example, the mechanism will start the timeout for both T1 and T2 and, as soon as one of them expires, the other transaction can proceed.

Two-Phase Locking (2PL) is a locking-based pessimistic concurrency control mechanism. 2PL locks all data units that a transaction appoints prior to executing its read or write

commands. In its basic form, 2PL guarantees serializability [113]. Serializability means that the outcome of all committed transactions corresponds to the outcome if they were executed sequentially, without overlapping in time. Two variants of 2PL, S2PL (Strict 2PL) and SS2PL (Strong 2PL), further provide recoverability (S2PL) and even cascadelessness (SS2PL) [48, 138]. Recoverability means that no committed transaction has ever read the data unit that is written by an aborted (and rolled back) transaction. Recoverability can lead to cascade aborts, where aborting a transaction TX implies aborting all other transactions that have read the data written by it. The property of cascadelessness avoids cascade aborts.

4.1.1.2 Optimistic Concurrency Control

The optimistic concurrency control mechanisms do not coordinate the transactions using locks. There are several approaches how to dispense with locking.

First, it is possible to use timestamps, in which the TM assigns a timestamp to each transaction. The timestamp can, for example, correspond to the TM clock value at the beginning of the transaction. The optimistic control mechanism implements the timestamp ordering, in which it serializes the concurrent transactions that affect the intersecting sets of data units. If, for example, two transactions T1 and T2 both affect the data units x and y, and the timestamp of T1 indicates that it started before T2, the timestamp ordering must guarantee that the commands from T1 are executed before the commands from T2 on each of the affected data units.

Second, the mechanism can maintain the precedence graph, which is a directed graph whose nodes are transactions and edges represent the existence of conflicting commands from the transactions. For example, consider two concurrent transactions T1 and T2: if there is a command in T1 that precedes and conflicts a command in T2, the mechanism adds the edge from the node T1 to the node T2. For each new command the concurrency control mechanism checks if a new node or edge should be added to the graph. If the graph is updated, it triggers the procedure called serialization graph testing (SGT) [138]. The goal of the SGT is to detect if the graph contains cycle and, if so, to resolve such occurrences. A cycle means that two transactions (that form it) are mutually conflicting and, therefore, are blocking each other. Every time when a cycle is detected in the graph, the mechanism selects transaction(s) whose aborting removes their respective node(s) from the graph and also removes the cycle. Alternatively, the concurrency control mechanism can implement commitment ordering (CO), which delays execution of commands within the update, but adds the nodes and edges to the precedence graph as they arrive, allowing creation of cycles. It then periodically parses the graph to find transactions that are not involved in any cycles and executes them. It then discovers

the transactions that are involved in the cycles, i.e. mutually blocking each other, and decides which to abort and which to commit, thereby removing the cycle. There are two variants of CO schedulers: COCO (CO Coordinator), which provides serializability, and CORCO (CO Recoverability Coordinator), which provides recoverability. The main drawback of this approach is that the space required to maintain the precedence graph grows with the square of the number of relevant transactions and easily can exhaust memory of the RM [138].

Third option is to use validation protocols. Validation protocols divide a transaction's execution into read, validation and write phase. In this case, transactions use data resources without acquiring locks on them. However, before committing a transaction, the mechanism must verify that no other transaction has modified the data that it previously read.

4.1.2 Distributed Transaction Management

If a transaction spans multiple RMs, global serializability must be achieved across all RMs.

If the RMs are synchronized and share common info, e.g. timestamps, the TM can enforce global serializability using a Distributed Timestamp Ordering protocol. Alternatively, the TM can use Atomic Commit Protocol (ACP), such as 2PC or 3PC [113]. 2PC starts with the commit-request phase (or voting phase), in which the TM prepares RMs to take the necessary steps (to vote) for either agreeing to or rejecting the suggested transaction. In the subsequent commit phase, it decides, whether to commit or to abort the transaction based on the received votes. 2PC has one disadvantage. If the TM fails during the commit phase, some RMs might end up blocked, infinitely waiting for the instruction from the (failed) TM to commit or abort the transaction. The 3PC protocol eliminates the 2PC blocking problem with the third phase, called pre-commit. If the TM fails before sending a pre-commit message, the RMs conclude that the transaction was aborted.

4.2 Design Choices

Our idea for coordination of network-wide updates in SDN is inspired by the related work in the area of DBMS, previously discussed in Section 4.1. Our proposal introduces transactional network-wide update logic into the SDN and relies on a transaction manager (TM) to coordinate transactions over multiple resource managers (RM). In the following sections, we elaborate our design choices.

4.2.1 Choice of Entity Locations

We discuss in this subsection how the new logical entities, TM and RM, relate to the SDN model, i.e. SDN controllers and SDN switches.

We design the TM as a core service in the SDN controller. This choice is straightforward because the SDN controller, as the entity that implements the north-bound interface (NBI), gets all the requests from the SDN applications. In our case, the new core service features an additional API call, which the SDN applications use to profit from the transactional update support. For a distributed SDN control plane, a separate instance of TM runs in each controller instance (e.g. in the ONOS architecture [15] an instance of TM would run in each OF manager). By design, the TMs do not need to synchronize or to coordinate in any manner. This will be explained in more detail in Section 4.3.1.1.

Now, there are several options for RM design and placement. The obvious option is to place it within the SDN controller as well. In this case, the RM would maintain an image of network resources and enforce some concurrency control mechanism over the image. The "communication" between the TM and RM would occur within the same address space and would not load the network with any additional signaling. However, this design would require a lot of state synchronization between RMs on different SDN controllers in case of a distributed control plane deployment. Indeed, the RMs on different SDN controllers could be holding an image of the same switch. Hence, such a solution would impose additional constraints on scalability.

There are two design options for an RM outside of the SDN controller. The first is to implement it as a middleware for a fraction of n switches. In this setting, m distributed RMs control n switches, where $m < n$. This implementation faces a few performance issues in which the middleware RM becomes a single point of failure and a bottleneck for the group of switches and, since it must interpret and serialize the received transaction primitives, it cannot be implemented as a multithreaded process. Finally, if an update crosses the resource domain of one RM, middleware RMs must implement an eastbound/westbound interface to, again, coordinate update installation across the domains. This reasoning makes us believe that the operational logic of the middleware RM in not very straightforward.

Another option for the RM outside of the SDN controller is to incorporate it directly into the SDN switch, where it implements the concurrency control mechanism only for the resources of the switch. In this case, each local RM is independent of other RMs and does not have to coordinate with them. Instead, it gets coordinated from any TM using the suitable mechanisms described in Section 4.1.2. The downside of the local

RM is that it requires new logic in the SDN switch, which should stay simple. On the positive side, the switch-local RM requires simpler logic than the middleware RM and supports distributed SDN control planes by design, making it suitable for large-scale deployments. Given the strong scalability argument. i.e. requirement REQ 1.2 from Section 3.3, we opt for the latest choice and place the RM directly on the SDN switch.

4.2.2 Choice of Protocols and Mechanisms

We now discuss the suitability of the concurrency control mechanisms from Section 4.1.1 and protocols for distributed transaction management from Section 4.1.2 to our RM and TM in the SDN.

To implement atomicity of a network-wide update, which involves several SDN switches, our TM provides an interface for 2PC. We consider 2PC to be sufficient despite its vulnerability to TM failure. The potential TM failures, which correspond to controller failure in our case, can be solved with the existing proposals for fault-tolerant SDN controller platform, which enforces replicated state machines [66]. We do not consider the Distributed Timestamp Ordering protocol as it requires synchronization of the RMs in the network, which does not comply with the requirement REQ 1.3 from Section 3.3.

For the general concurrency control in the RM, we have several choices: resource locking, maintaining precedence graph with SGT or CO, timestamp ordering or validation protocol. The CO schedulers (COCO or CORCO) require a complex logic in the switch that maintains the precedence graph, buffers the transactions and orders the commit events. The same applies for the SGT schedulers. The timestamp ordering also must provide a logic for buffering the transactions and ordering their execution according to the timestamp values. Since our RM is incorporated in the switch, we do not consider using SGT, CO and timestamp ordering, as they would introduce too much complexity in the switch. Furthermore, since we do not aim to prevent the situations where SDN applications might read network information that becomes obsolete only a moment later due to the data plane changes and rather concentrate on resolving potential conflicts from other concurrent updates, we disregard “read before write”. Hence, we do not consider validation protocol as the right mechanism for our RM. Trying to keep the switches as simple as possible, we decide to use the resource locking mechanism.

For these reasons, we develop a locking-inspired exchange and logic, adapted to SDN specifics. This is a conscious design choice with potential consequences: as strict serializability removes all concurrency (more precisely, it only supports concurrency on disjoint resource sets), it might induce a relatively high cost: locking places the strictest

constraints on the resource availability by locking the whole switch for reading and writing until the end of the transaction.

Hence, this approach raises doubts with respect to the transaction throughput at the system level (i.e. the system could starve or reach the so-called "thrashing point" in DBMS terms). To mitigate this, one could increase the granularity, locking e.g. ports instead of entire switches, etc. This comes at the expense of a slightly more complex RM implementation and could be used up to some granularity level. However, fundamentally, the problem remains the same, and while we would recommend it for commercial solutions, we do not pursue this vector in our implementation. At this point, we are interested in the limits of the most general solution to concurrent network-wide updates. Therefore, instead of making trade-offs in the design, we study the performance of the most rigorous design option in several problematic network topologies featuring bottleneck switches, more likely to be overbooked. Note that in a commercial deployment, similarly to DBMS, we would expect the support for different isolation levels, to best trade off the requirements of different application classes against the incurred costs. In this sense, our results can be seen as a "worst case" result or bottom line for network-wide concurrent updates performance.

4.2.3 SDN Model Richness

We make no assumptions on traffic arrival, distribution or on the type of applications supported. Any suitable method should support both proactive and reactive network updates. Moreover, the mechanism should be SDN-application agnostic. This means that we are not interested in resolving conflicts in application's own semantics (e.g. if an application creates a loop). This can be extended to several applications: for instance, one application installing a policy, and the next application removing it immediately after, is fine from our point of view. For mechanisms resolving semantic conflicts in SDN network policies, we divert an interested user to the policy-analysis tools such as Frenetic [45] and similar [70, 67]. We limit ourselves to the correct execution of the individual network-wide updates, which could occur in any order, quantities and places affecting any resource sets.

4.2.4 Support for Existing Consistency Mechanisms in SDN

We now briefly consider integration possibilities of our proposed architecture with some of the existing mechanisms that enforce consistent network updates.

In the proxy-based architectures, e.g. VeriFlow [70] or NetPlumber [67], TM would run as the core service in the proxy. Once VeriFlow or NetPlumber have verified the update, they dispatch it to the network using the TM's API call. The same applies

for integration with consistency enforcing mechanism(s) that are integrated in the SDN controller, e.g. SE-Floodlight [109] – they use the API of the local TM to dispatch the verified update in the network.

Finally, our proposed architecture can be integrated with the most recent proposals where the responsibility of maintaining the consistent state during the update is delegated from the controller to the switches, e.g. ez-Segway [97], proof labeling schemes [44, 43] and DDP [83]. The controller pre-computes the information that is relevant for the update and use TM's API call to forward it to the switches, which then use direct message passing to execute the update in the consistent way.

We do not consider integration with mechanisms such as Dionysus [63] and Cupid [137] since these assume specific application types and network states, which opposes to our own assumptions and requirements to achieve SDN-application agnostic mechanism.

4.3 Transactional SDN

In this section we present our solution for transactional network-wide updates in SDN.

4.3.1 Architecture

Figure 4.1 shows our high level architecture. The additions to the standard SDN models are shown in red. As discussed, these include the transaction manager (TM) that runs in the SDN controller, the resource manager (RM) running in the switch and the transactional SBI (T-SBI), an extension of the southbound interface for the TM-RM communications.

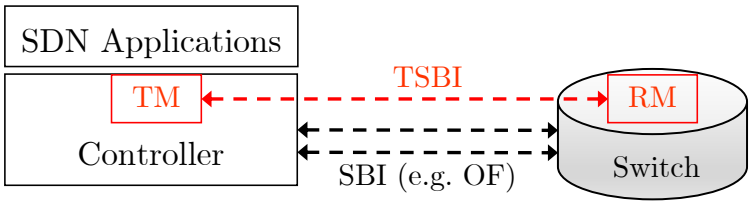

Figure 4.1: Transactional SDN architecture.

4.3.1.1 Transaction Manager

The TM is a module in the SDN controller that gets update requests from SDN applications and communicates with the required RMs on behalf of them, i.e. it handles the transactions. TM acts as a bookkeeper, tracking the progress of transactions and

communicating with the required RMs. It provides the API to the applications, to be used for installation of network updates that require transactional semantics. When installing an update across multiple switches, the TM acts as the ACP coordinator.

For deployments with multiple controllers, each controller runs an independent TM. Note that every TM is assumed to be able to reach all the switches in the network concerned by the requested update; this is consistent to the usual assumptions in the general distributed SDN control planes. We do not consider any "pre-established hierarchies", as updates from applications cannot be generally limited to a subgroup of switches. Different TMs do not require any coordination, except that the IDs they assign to transactions should be globally unique. This uniqueness is however easy to achieve by assigning unique IDs to the controller instances and having transaction ID consist of the instance ID and a local transaction ID, which is unique within the instance.

4.3.1.2 Resource Manager

The RM controls the resource, i.e. the SDN switch. Our RM essentially transforms the SDN switch into a transaction-aware medium with well-defined states that can be accessed and changed only according to transactional semantics (e.g. failed transactions can be undone).

The essential part of RM is a concurrency control mechanism (algorithm) that schedules concurrent write operations over the resource so as to achieve a desired level of their isolation. There are plenty of available mechanisms to select from, which we discussed in the Section 4.1.1. However, for the reasons explained in the Section 4.2.2 we select the simplest one that just locks the resource so as to enforce sequential access to it. The RM can retrieve the transaction ID from the received commands, grant exclusive access to a specific transaction, track the IDs of the committed transactions and interpret 2PC messages. The switch is capable of performing all the functions supported by OpenFlow v1.5, e.g. installation of related state changes into staging area.

4.3.1.3 T-SBI Interface

The T-SBI is an extension of the SDN southbound interface that enables the TM-RM message exchange. The T-SBI essentially transports messages of a protocol or protocols used to provide transactional operation. This thesis assumes that the TMs use ACP, such as 2PC, to coordinate the RMs, which in turn use locking concurrency control mechanisms to control the local resource.

4.3.2 Transactional Network Update Cycle in SDN

We here show a typical sequence of steps in a transactional network update cycle (Figure 4.2):

1. Upon the reception of a network update request from an SDN application, the TM sends a so-called `Vote` message to the RMs involved in the update. The `Vote` message carries the unique ID of the transaction. The resource (switch) interprets this message as both transaction initiation and an ACP vote.

2. On the reception of `Vote`, as per locking mechanism, each RM locally locks the resource, i.e. the switch, so the resource becomes unavailable for other requests (e.g. from another TM or application) until further notice from the initiating TM. In other words, the RM will reject all other transactions, i.e. the other transactions will not wait for the lock to be released, risking deadlocks. Upon successful resource locking, every RM tries to apply all requested changes to its respective staging area. Note that the updates applied to the staging do not affect the traffic traversing the switch. If this is successful, the RM sends a `Confirm` message back to the TM. Otherwise, the RM sends `Reject` and immediately unlocks the resource.

3. If the TM got confirmations from all the RMs in the current update, it starts the commit phase by sending a `Commit` message to them. If at least one RM rejected the request (e.g. it is already locked) or did not reply within the expected time, the TM aborts the update by sending `Rollback` to the RMs that replied with `Confirm`. Thus, even if the TM receives `Confirm` from all but a single switch, it will release the obtained locks and start anew.

4. On `Commit`, the RMs activate their staging areas, whereas on `Rollback`, they discard these. In both cases, they send `Finished` back to the TM and release the locks.

Figure 4.2 shows a typical message exchange of a committed transactional update, while Figure 4.3 depicts a situation, where one of the involved RMs rejects the update and the transaction has to be aborted. The TM therefore sends `Rollback` to all RMs that previously confirmed the update, which then discard the changes in the staging area and release the lock.

Note that we merge the transaction initiation and the beginning of the ACP, which might limit the range of supported transactions to write updates only, i.e. we rule out the read requests. However, this complies with our goal, as we disregard “read before write” transactions, which we previously elaborated in Section 4.2.2. Merging the locks

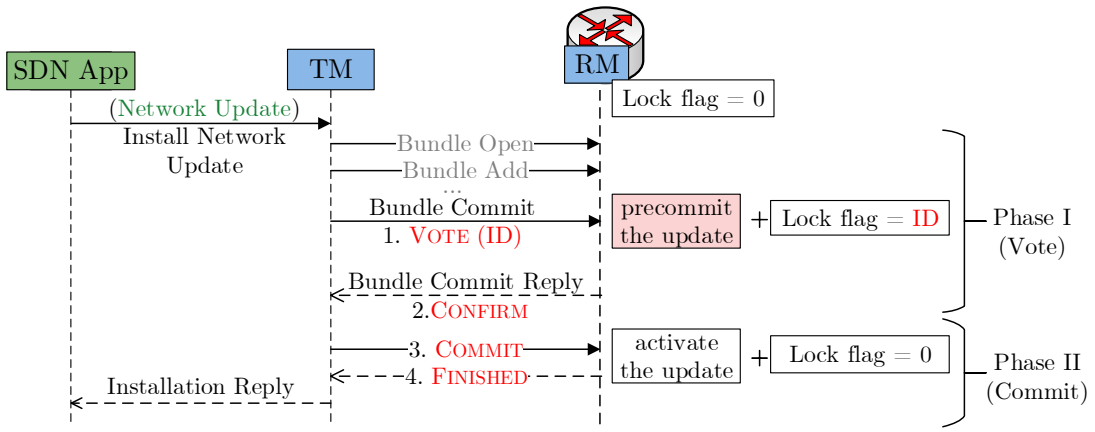

Figure 4.2: TM-RM message exchange for a successful (commit) transactional network update.

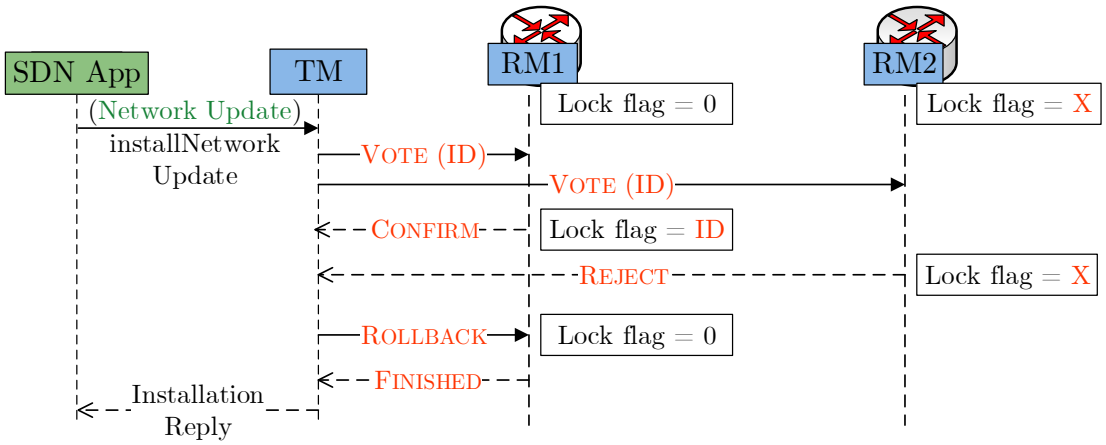

Figure 4.3: TM-RM message exchange for a failed (rollbacked) transactional network update.

and the ACP `Vote` messages permits to reduce the time of the switch lock, which is a performance-related parameter, the implications of which we analyze and show later in Section 4.4.

4.3.3 Implementation

Our proof-of-concept implementation is based on OpenFlow and popular SDN software packages, notably FloodLight and OpenVSwitch. We implemented TM in Java, as a new core service of the Floodlight controller [41]. The RM is implemented in C, as an extension of OpenVSwitch (OVS) [107]. A brief description of changes introduced at all SDN layers is shown in Table 4.1. In our current implementation, the T-SBI, i.e.

the TM-RM transactional protocol and the ACP, uses standard, unchanged OpenFlow (v1.4 and higher).

SDN Controller	Includes the TM as a new core service. The TM provides an API call `transactionalUpdate()` to the application layer, to execute their updates in transactional manner.
SDN Application	The applications should register to the TM. They push their updates through the TM, by using its API.
OpenFlow Protocol	No changes required
SDN Switch	Includes the RM that can lock the switch and handle ACP exchanges with the TM.

Table 4.1: Transactional updates implementation summary.

4.3.3.1 T-SBI Implementation

We implement the T-SBI using existing OpenFlow messages, as per Table 4.2. For example, we implement `Vote` as an OpenFlow bundle [90] with a transaction init message at the beginning (i.e. `OFPT_FLOW_MOD` with a specific command and tableId combination), followed by the commands from the update. Therefore, when an RM receives a bundle in which the first command indicates the beginning of a new transactional update installation, it proceeds with the necessary steps as ordered by ACP. At the end of such bundle, the RM does not activate the changes in the staging area, but sends `Confirm` to the TM, which is in our implementation a `OFPT_BUNDLE_CONTROL` message with type set to `OFPBCT_COMMIT_REPLY`, and waits for the `Commit` or `Rollback` message from the TM.

`Commit` in our implementation is an `OFPT_FLOW_MOD` message with the tableId set to 255 and the command to `DELETE`. `Rollback` in our implementation is the same as the `Commit`, only with the difference in the command, which is set to `DELETE_STRICT`. The RM reads the command value and interprets the ACP message accordingly.

The implementation details of `Reject`, `Rollback` and `Finished` are also given in the Table 4.2.

Primitive	OpenFlow message	Command `OFPFC_*`	Type	Code	TableId
`Vote`	`OFPBCT_OPEN_REQUEST`	-	-	-	-
`Init Trans`	`OFPT_FLOW_MOD`	`MODIFY_STRICT`	-	-	255
	(Network Update)	-	-	-	-
	`OFPBCT_COMMIT_REQUEST`	-	-	-	-
`Reject`	`OFP_ERROR_MESSAGE`	-	`OFPET_BUNDLE_FAILED`	`OFPBFC_MSG_FAILED`	-
`Confirm`	`OFPT_BUNDLE_CONTROL`	-	`OFPBCT_COMMIT_REPLY`	-	-
`Commit`	`OFPT_FLOW_MOD`	`DELETE`	-	-	255
`Rollback`	`OFPT_FLOW_MOD`	`DELETE_STRICT`	-	-	255
`Finished`	`OFP_ERROR_MESSAGE`	-	`OFPET_FLOW_MOD_FAILED`	`OFPBRC_UNKNOWN`	-

Table 4.2: T-SBI protocol implementation summary.

4.3.3.2 TM Implementation

Our TM is a new core service of Floodlight. It implements the interface IOFMessageListener, to be notified when the controller receives an OF message. It registers to receive only `OFPT_BUNDLE_CONTROL` and `OFP_ERROR_MESSAGE`, which are used to implement `Confirm`, `Reject` and `Finished`. The other messages are irrelevant to the TM.

Similarly, to use the TM, SDN applications must register. Strictly speaking, this renders the transactional updates unusable to old applications. However, this is just our current implementation choice. They then use the exposed method transactionalUpdate() to push their updates in a transactional manner. The method parses the update, detects the affected switches and starts the transactional update. Each update gets a unique, 4-byte ID. To ensure ID uniqueness across different TMs (i.e. different controllers), we use the first byte to identify the TM, while the last 3 bytes identify the updates from a single TM. We transfer update IDs in the xid field of the corresponding OpenFlow messages.

Each TM maintains a local data structure that stores the complete info for each update, such as update itself, its ID, state, reattempt counter, etc. For each received `Confirm`, the TM retrieves the update ID from it and checks in this data structure if all `Vote` messages of that update resulted in `Confirm` messages. If yes, the TM commits the update. The TM again checks if each `Commit` message resulted with `Finished` and only then removes the update from the data structure. On the other hand, the first time the TM receives a `Reject`, it marks the update as failed and aborts it. The same occurs if the TM does not receive all the expected `Finished` messages within a given waiting period. The TM does not however remove the update until it sends `Rollback` to all RMs that confirmed the update after the arrival of the first `Reject` or, in case of missing `Finished` messages, to all RMs that did send out `Finished`.

The transactionalUpdate() call gives the applications also the possibility to set the maximum number of update installation attempts, MAX_N_INST. When an update installation fails, the TM retries after a random period of time, but at most MAX_N_INST times. To implement this, the TM maintains an attempt counter as part of its data structure.

4.3.3.3 RM Implementation

We extend OVS to include the following additions. The switch must a) set and hold a lock flag as necessary, b) perform updates in the staging area and c) activate them on `Commit`. We implement the lock flag as a 4-byte unsigned integer and add it to the ofproto, an OVS library that implements an OpenFlow switch. When transiting the state to locked, the switch sets the lock flag to the update ID, which it reads from the

xid field of the `Vote`. We use a mutex to provide exclusive access to the lock flag. So if concurrent `Vote` messages arrive, only one of them can successfully set the lock flag.

To update the staging area, we split the existing bundle commit phase (implemented in method do_bundle_commit() in ofproto) in two. In the first, the switch creates the staging area with a new version number, equal to the update ID, and installs the update in it. Note that we do not require any change in case of failure; the standard OVS code suffices. In case of success, our modied switch does not activate the new version immediately; instead, it holds back the activation, but it sends a `OFPBCT_COMMIT_REPLY` to the TM.

We added two methods to ofproto, transaction_commit() and transaction_rollback(), to process the second phase messages. They get called through the OpenFlow message handlers of OVS, for example handle_flow_mod(), that in turn is called from the main message handler of ofproto, handle_openflow(). We extend handle_flow_mod() to include a check if the lock flag is set. If yes, the switch will discard all messages whose xid is not equal to the lock flag - in case that these messages belong to other transaction, the switch will also send the `Reject` to the initiating TM. Otherwise, if the switch receives the message whose xid is equal to the lock flag, it checks if the received message should be interpreted as `Commit` or `Rollback`. In the case of a `Commit`, the switch activates the new version equal to the update ID. Since this procedure is a single write operation on a pointer, we consider it atomic and assume that it cannot fail. In case of `Rollback`, the switch discards the staging area. Finally, the switch clears the lock flag.

Finally, to achieve durability, we disallow explicit self-expiring rules in the resources; instead, we use a coordinated combination of the methods above to achieve a correct network-wide effect.

4.3.4 Code Availability

We publish the source code for the TM and RM together with setup instructions on GitHub [2].

4.4 Evaluation Results

4.4.1 Evaluation Methodology and Setup

Our evaluations have two goals. On the one hand, we want to demonstrate the feasibility and correctness of our proof-of-concept implementation, while, on the other, we want to provide a comprehensive set of evaluations that permit us to judge on the

performance of our transactional SDN. In that respect, we use a rather complex evaluation environment that mixes simulations and Mininet/Floodlight based emulation. As for the PoC implementation, we show in Section 4.4.2 performance results collected from our *emulation*. However, the scale of the tests performed here is rather small, as Mininet did not permit scaling them up as much as we wanted (see below). To reach the desired scale, we use *simulations*, i.e. we test the system in a custom-made, event driven simulator, that, in turn, learns and sets up a number of relevant parameters with help of our emulation. Section 4.4.3 explains this in detail.

The setup described next underlies both our emulation and our simulations. (The emulation is just limited to smaller ranges of relevant parameters, as will be indicated). We use network topologies generated from [95, 59], representing backhauls of carrier networks. They are hierarchical and consist of three layers: access, aggregation, and core. They contain k switches at the core and k pods of size k switches, i.e. a total of k^2 switches, at the aggregation. $k/2$ switches of each aggregation pod are connected to the access, with five access switches per each aggregation switch. This results with the total of $5k^2/2$ access switches in the network. The rest of $k/2$ aggregation switches from the pod are connected to the core, each of which is connected to n core switches. Therefore, n represents the core aggregation parameter. We consider no redundancy at the access part, i.e. each access switch is connected to only one aggregation switch. This is not however a problem as we do not lock access switches. Finally, there are five users attached to each access switch. Hence, the total number of users is $25k^2/2$.

There is a single controller in the network, and thus a single TM instance. Path setup requests arrive from the users in the network according to a Poisson process with parameter λ requests/sec. The source-destination pairs are selected randomly, under the constraint that their access switches are not connected to the same aggregation pod. This guarantees that the paths will always traverse core switches. For each request, we determine the shortest path from the source to the destination (if multiple shortest paths are available, one is selected at random) and lock all switches on the path except those in the access.

When a path setup request gets rejected, the TM backs off and reschedules the request at a later time, but at most 3 times. Each backoff time T_{BACK_OFF} is drawn from the exponential distribution with the parameter 100 (i.e. the average waiting time is thus 0.01s). We are not interested here in more sophisticated backoff strategies. They might be investigated in the future.

We measure two performance metrics:

- Success rate: The fraction of requests that have been successfully processed.

- Path setup time: The time elapsed from the moment the TM receives a request until the moment the last switch in the path receives the `Commit`. We measure the latencies of successful requests only.

In summary, the parameters in our simulations are λ and the core aggregation parameter n. We vary λ from 10 to 1000 requests/sec, n from 2 to 4, and consider three different topologies:

- a small size topology ($k = 4$) with 40 access, 16 aggregation, and 4 core switches,
- a medium size topology ($k = 8$) with 160 access, 64 aggregation, and 8 core switches,
- a large size topology ($k = 12$) with 360 access, 144 aggregation, and 12 core switches.

4.4.2 Emulation Results

We test our implementation in a Mininet [80] /Floodlight [41] testbed running OpenFlow version 1.4. Users generate requests, these reach the controller as PACKET_INs from access switches, where they are handed to forwarding application that calculates the update and pushes it to the TM for transactional installation.

Figure 4.4 depicts the success rate in the small network, with $k = 4$ and $n = 2$, when the request arrival rate varies from 10 to 100 requests/sec in increments of 10. Backoffs are turned off here. Each point in the graph represents an average over 10 independent experiments and its corresponding 95% confidence interval. We observe from the results that the success rate decreases with the increase of the request arrival rate. This is expected since the increase in the arrival rate of flows increases the possibility that two requests collide with each other.

While it indeed demonstrates the feasibility of our design and the ease of implementation, this testbed is bound to a number of problems, the most important of which is that it does not permit large scale tests, i.e. with larger networks and larger request arrival rates. To see this, bear in mind that Mininet runs a daemon (ovs-vswitchd) that sets up the switches and operates switching across the network. Only a single instance of ovs-vswitchd runs at a time. It listens for OpenFlow messages and delivers them internally to the destination switch. The increase in the number of managed switch instances increases the time required to deliver an OpenFlow message to the appropriate switch. This internal delivery time reaches few hundreds of milliseconds for emulated networks with around 500 switches and varies so much that it cannot be easily factored out and deducted from the final results. In plaintext, for any large scale Mininet test,

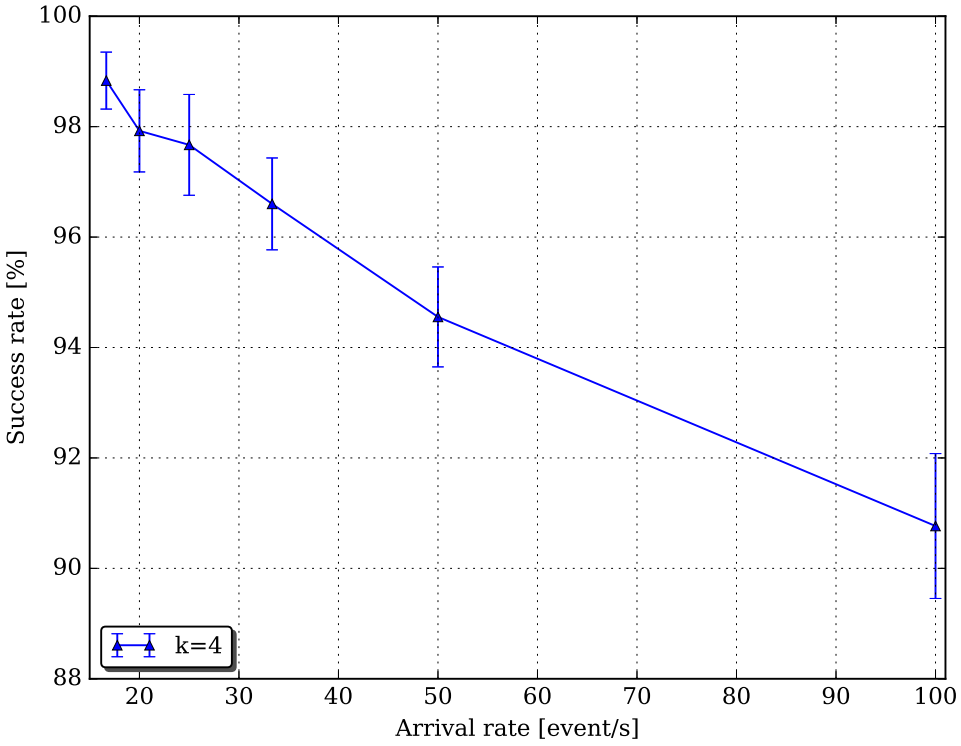

Figure 4.4: Emulation results: success rate without backoffs, for the small network, with $k = 4$ and $n = 2$, and for λs between 10 and 100 requests/sec.

the transactional updates experience delays that are artifacts of Mininet itself, not of the system we want to test, and make the performance figures hard to understand. That it the reason why we proceed with simulations.

4.4.3 Simulations

In our simulator, we first generate a list of individual requests that arrive at a desired aggregate rate. As earlier stated, each request corresponds to a transactional setup of a path between the requester and a randomly selected destination node. The simulator maintains a data structure that tracks the locks of each individual switch. So when an application sends vote requests to the switches $s_1, \ldots, s_k$, we change the state of $s_1, \ldots, s_k$ to "locked" and also indicate the times when the locks start and stop (explained shortly), as well as the ID of the installation request that holds the locks. Thus we can see if a switch is free or not when a new vote arrives and record that as a vote failure.

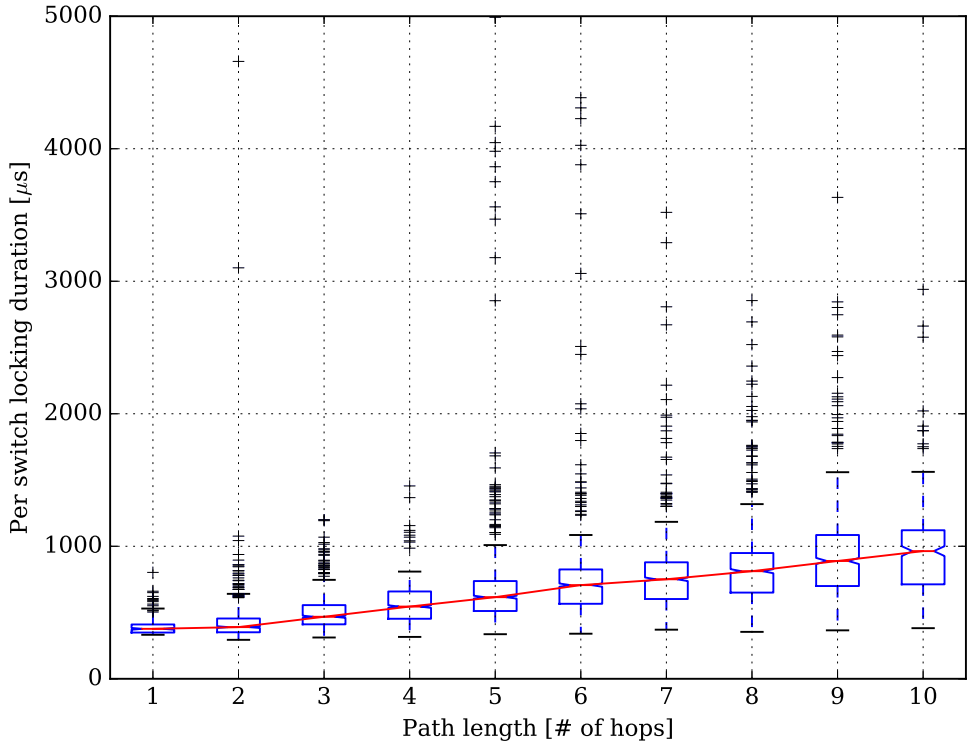

Figure 4.5: Emulation results: Locking duration per switch measured over our Mininet/Floodlight testbed. The results are shown for different path lengths.

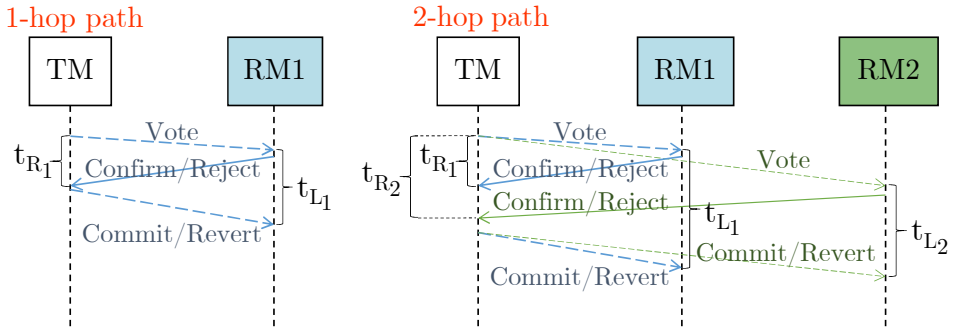

Figure 4.6: Illustration of the locking duration per switch for a 1-hop and a 2-hops path.

But what locking times do we indicate for the switches? This is where we make use of our emulation. We create a linear Mininet network with the diameter k between 1 and 10 (networks with larger diameters are hard to expect [85]) and low network load (to avoid any congestion of the SDN controller), and measure the time in which the switches stay in the state "locked", for different path lengths. We plot these times as

boxplots for each $k = 1, \ldots, 10$ in Figure 4.5. We also create the CDF for each k value. So, to determine the lock times, we simply see how many switches k a request should lock, pick the CDF for that k and generate k random times from that CDF, one for each switch involved in the update.

Notice that the median and 95-percentile of the locking duration per switch increase with the path length. This is expected as the TM must wait for vote responses from *all* affected switches before it decides on the transaction outcome and release the locks. To illustrate this, we show message flow diagrams for a 1-hop and a 2-hop path in the Figure 4.6. In case of the 1-hop path, the TM decides whether to commit or abort the transaction immediately on receiving the vote response from the only one affected switch. In the second case, although the TM sends votes simultaneously to the both switches, their responses arrive at different times, t_{R_1} and t_{R_2} ($t_{R_1} < t_{R_2}$), and the TM must wait for the arrival of the second vote response to decide on the transaction outcome. It is this time that determines how long the *both* switches, not only the second one, must hold their locks. Therefore, the locking times of the switches, (t_{L_1} and t_{L_2}), both depend on the maximum response time, in this case t_{R_2}.

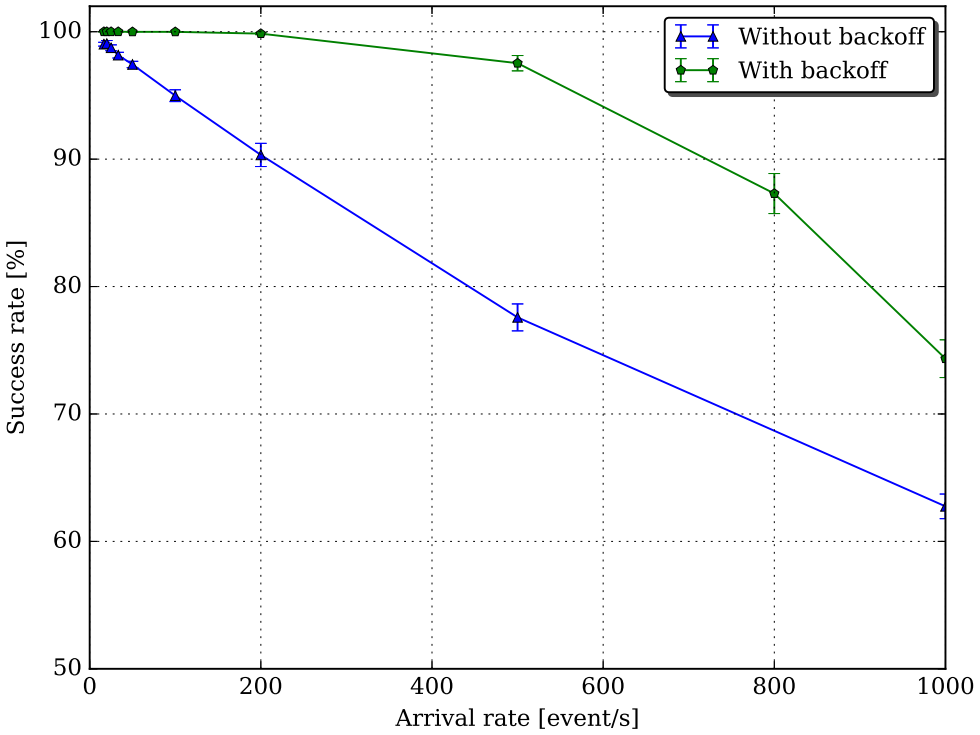

Figure 4.7: Success rate for different request arrival rates. $n = 2$.

4.4.4 Simulation Results

Figure 4.7 presents the success rate as a function of the request arrival rate λ in the small network where $k = 4$, with and without backoffs. Each point in the graph is an average over 10 independent experiments. We run each experiment with 10000 flows in total generated by the users.

The results show that the success rate decreases when λ grows, especially without backoffs, as indicated by the blue curve in Figure 4.7. This is just as expected, because the probability that two requests collide increases with λ. Further, Figure 4.7 clearly indicates that our backoff mechanism helps. For example, with backoffs and for the arrival rate of 1000 requests/sec, 25% of the requests were rejected. At the same time, the success rate is limited to less than 65% without backoffs.

Finally, comparing the emulation results from Figure 4.4 with the simulation results in Figure 4.7 for the same setting (e.g. $k = 4$, without backoffs and for λs below 100 requests/sec), we can observe that our simulator provides similar results as our emulator. This indicates the precision and accuracy of the simulator.

We identified a clear, combined effect of the two topological properties on the performance of transactional SDN: the distribution of betweenness centrality [96] of nodes in the network graph and the distribution of the shortest path lengths between the nodes that have users attached. Because we lock the switches during a path setup, we expect to see better performance for networks with even distribution of the betweenness centrality of the nodes and with smaller shortest paths.

We reduce the skewness of the distribution of the betweenness centrality and the shortest paths between the access switches in the hierarchical networks by increasing the n value (recall that n represents the number of core switches that each of the $k/2$ switches from the aggregation pods are connected to). Figure 4.8 shows the impact of this. So we run similar tests as above, but now for the medium network where $k = 8$, with and without backoffs and two additional n values, 3 and 4. The effects are rather striking. Without backoffs, the success rate goes up 12% and 15% when n is 3 and 4, respectively (compared to the results obtained when n is 2). With backoffs, there are almost no rejected requests when $n = 3$ and $n = 4$. So, with enabled backoffs, under the load of 1000 requests/sec, small redundancy cuts down the rate of rejected requests to virtually zero.

See Annex A for the detailed analysis of the effects of the two network topological properties in the transactional SDN.

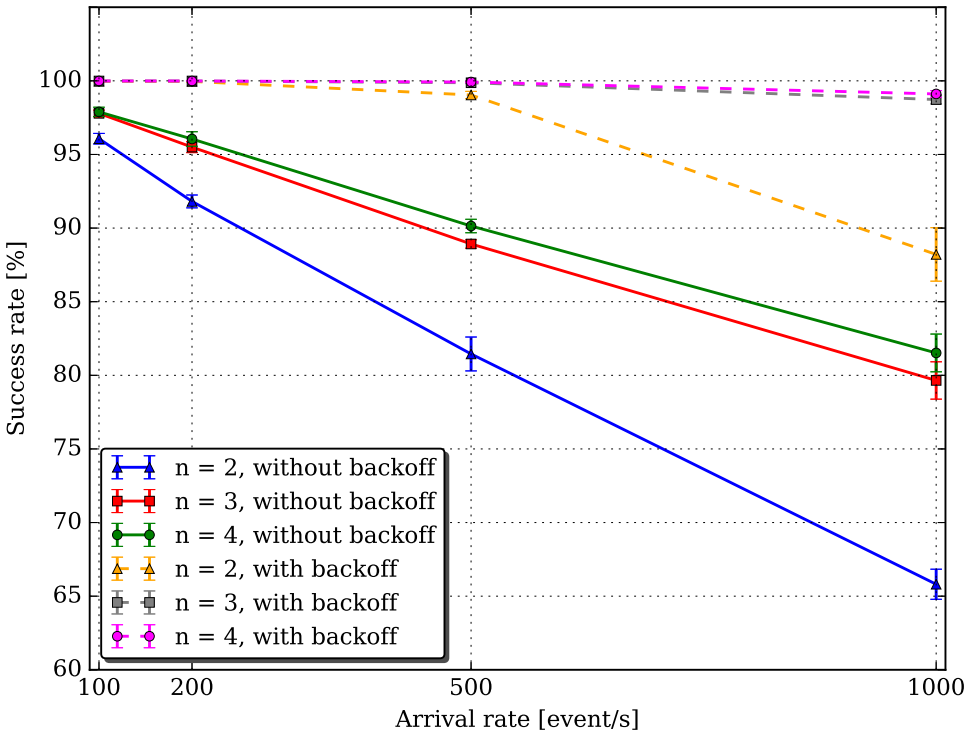

Figure 4.8: The effect of (path) diversity: Success rate for different values of the core-aggregation connectivity parameter n and different request arrival rates. Medium size network, $k = 8$, in all cases.

Figure 4.9 depicts the CDF of the path setup time. The results shown hold for the large network where $k = 12$, $n = 2$ and $\lambda = 1000$ requests/sec, backoffs were on. We observe that the setup time of around 75% of the updates is below 6ms and that the 95-percentile of the path setup time is below 30ms. The long tail of distribution is related to the updates which were installed after backoffs, through reattempts. To provide more insights into this, we break down this CDF into a histogram that shows normalized frequencies of requests installed in their first attempt and after 1, 2 or 3 backoffs. Rejected requests are included too. We observe that about 65% of the updates are installed in their first attempt, 20% of the updates on the first reattempt, around 9% on the second, and 6% on the third. 5% of the updates are rejected.

Finally, in order to precisely understand the costs of our transactional updates, we compare the path setup times we experience with those of a baseline that hold for the state of the art SDN, namely two-phase update protocol from Reitblatt et al. [114]. The two-phase update protocol uses versions to achieve per-packet consistency, which guarantees that each packet is processed by a single network configuration and never by

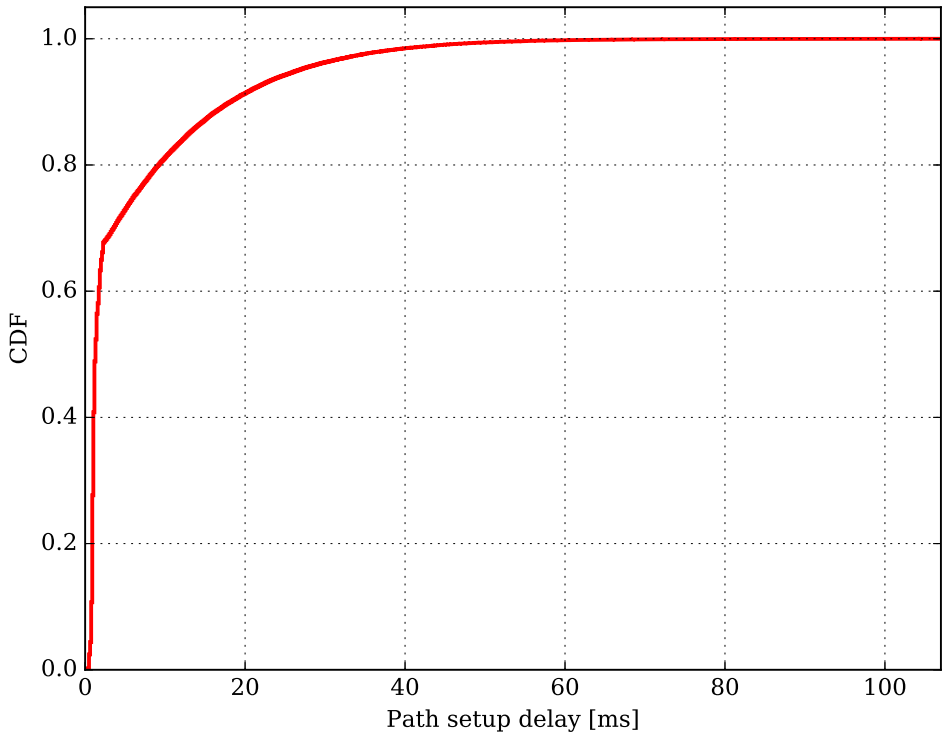

Figure 4.9: CDF of the path setup delay with backoffs. The results hold for the large network (k=12), $n = 2$, and $\lambda = 1000$ requests/sec.

the mix of two configurations. In the first phase of the update, the new configuration is installed on the switches to match the packets that contain the new version number. In the second phase, the protocol updates the ingress switch to stamp packets with the new version number. In our experiment we compare update installation using the two-phase update protocol without and with transactional support. The former essentially corresponds to the described update procedure from [114]. For the latter, we use the new API call `transactionalUpdate()` in both update installation phases. We measure the end-2-end delays, as observed by the end users. This is the time from the moment a packet is sent by a user until the moment the same packet is received on the destination. We perform these measurements with help of our emulator (not simulator), on an 8-hop linear topology (the problems discussed in Section 4.4.3 can be ignored, as this network is fairly small). The results are depicted in Figure 4.11. The message to pick from it is that the delays introduced by the transactional updates should not be considered as absolute values. Instead, they should be regarded relative to the delays that already exist in the

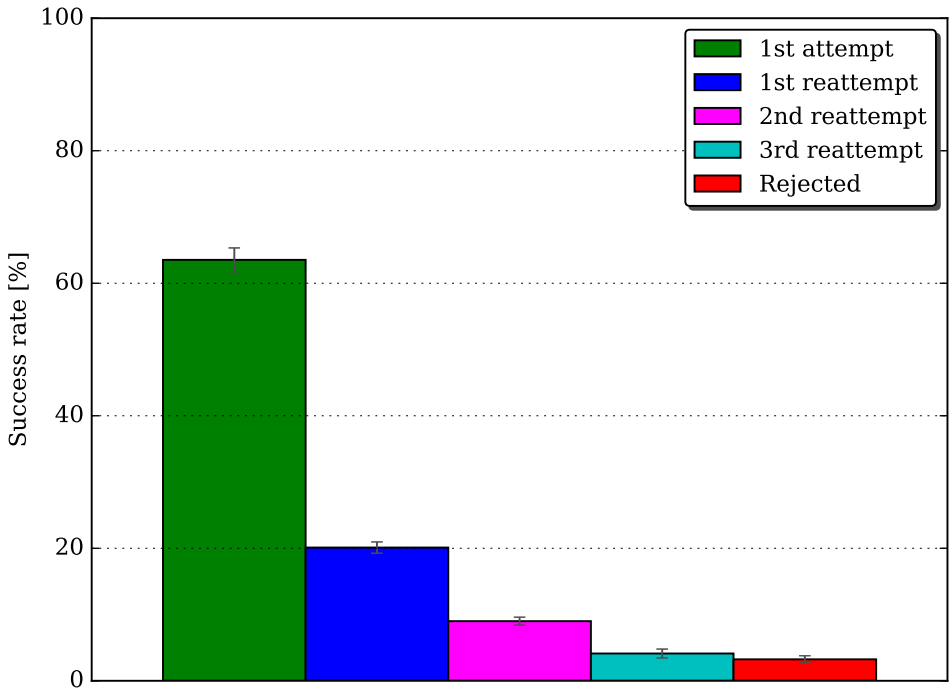

Figure 4.10: Histogram of updates that were: satisfied after the first attempt, satisfied after 1, 2 and 3 reattempts, rejected. The results hold for the same setting as in Figure 4.9.

baseline SDN. With respect to that, the small increase we observe in Figure 4.11 does appear as an acceptable price to pay for the service our transactional updates provide.

4.5 Related Work on Transactional Aspects in SDN

In this section we focus on the related work that examines the possibility to introduce transactional semantics in SDN.

Previous work has considered consistency of the network state during the update and proposed various mechanisms that provide different update properties. The general finding is that a consistent update cannot be done in one shot, such that the update commands are sent to all affected switches simultaneously. Reitblatt et al. [114] proposed two-phase update protocol to achieve per-packet consistency during the update. Jin et al. [137] proposed Dionysus, a mechanism that achieves congestion-free update by dynamically scheduling the execution of commands within the update on individual switches, based on their dependencies and runtime characteristics of the switches.

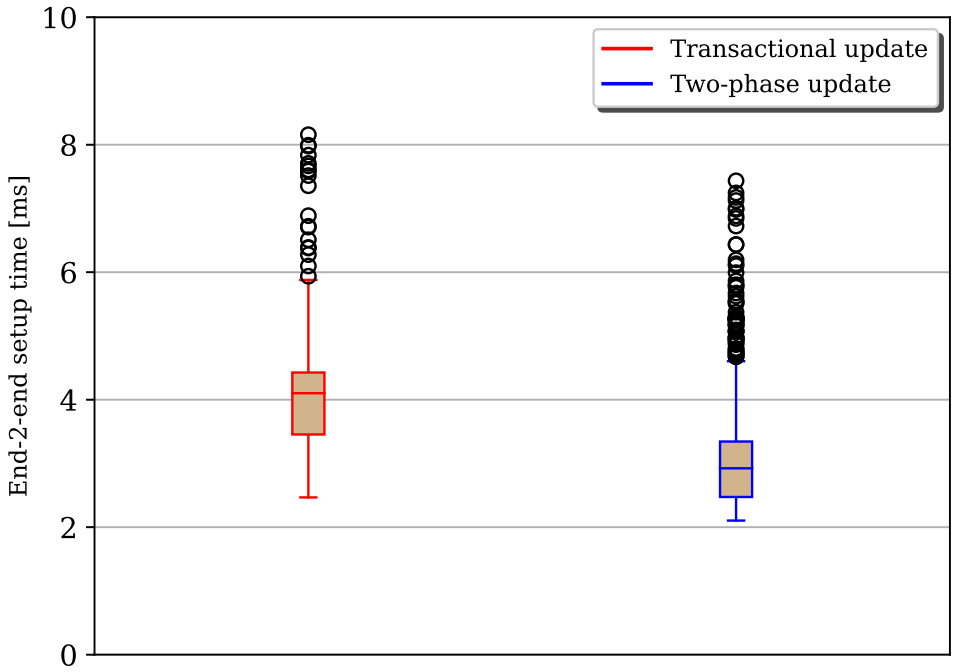

Figure 4.11: End-2-end delay, as observed by the end users, on an 8-hop linear topology, with and without transactional updates.

Similarly, Wang et al. [63] proposed Cupid, which improves the update time of [137] by enforcing parallelism for commands that do not interfere with each other. However, these proposals do not solve concurrency issues in SDN.

We discussed in Section 3.2.4 the synchronization framework from Schiff et al. [120] that enforces atomicity and isolation of the distributed single-switch updates. In an SDN environment, in which a switch state can be modified by multiple controllers, they propose using data plane configuration space of the switch as a shared memory and synchronization primitives to avoid conflicts. However, the authors only considered single-switch updates.

Fu et al. [46] designed transactional update service (called TUS), which is implemented as a layer between the SDN applications and the controller. Similar to our proposal, their service provides API for network update with ACID properties, whose main goal is to ease the application development. TUS uses optimistic concurrency control to isolate the updates and installs them using two phases, as proposed by Reitblatt et al.

in [114]. It also uses a log module to record the updates prior to sending them to the network. If the update installation fails, it uses the logged update to reattempt the installation or to rollback. While this work is limited to SDN-based environments with physically centralized controller, the realistic SDN deployments are expected to support distributed controller deployments. Besides, the authors did not provide any qualitative comparison with our proposal.

Katta et al. [66] introduced the notion of transaction in SDN as well. However, they observe the entire event processing cycle in SDN as transaction. In their work the transaction includes event delivery from the switch, event processing on the controller and update execution on the switches (note that we only observe the latter). They investigate fault-tolerance of SDN with replicated controller and demonstrate that failures can lead to non-atomic update processing, which is reflected in repetition of commands and unreliable event delivery that can finally lead to incorrect network behavior. The authors show that replication of the controller state *only* is insufficient to mitigate these issues. Instead, they advocate incorporating switch state in the SDN model and propose Ravana, the SDN model that guarantees that each transaction is processed atomically and exactly once. Ravana extends the SDN switch with lightweight mechanism that tracks the state of ongoing transactions. However, this work does not have concurrency control in its focus. Still, we consider it to be complementary to ours: transactional SDN and Ravana can easily be integrated with each other, especially since they both propose extensions to the SDN switch. Chandrasekaran et al. [22] also analyze fault-tolerance in SDN and propose LegoSDN, a layer between the SDN applications and the controller to support atomic network-wide updates and rollbacks in case of application crashes. However, the authors left dealing with concurrency issues for future work.

4.6 Conclusion

We now discuss how our architecture fulfills the requirements REQ 1.1 to REQ 1.3 from the Section 3.3.

Transactional SDN can serialize all updates, regardless if they affect a single switch or several switches. Using the locking mechanism in the RMs achieves the strictest isolation level between the updates. Hence, transactional SDN does not accept any level of concurrency of the two updates that affect the intersecting sets of resources. 2PC guarantees atomic execution of the updates. Therefore, our proposal fulfills the requirement REQ 1.1.

Our architecture incorporates the RM in the switch. This is the least constraining solution in different SDN controller deployments and in line with the established postulate

that the control should be distributed (or, at least, distributable). The other two RM placement possibilities (in the controller or as middleware) face scalability issues in deployments with a distributed SDN controller, and, therefore, do not comply with the requirement REQ 1.2.

Finally, our architecture does not use any time synchronization techniques and complies well with the requirement REQ 1.3.

We add transactional SDN to the Table 3.1 introduced in the Section 3.4 and fill the rows according to the previous discussion.

REQ	OF Bundle [2013]	Canini [2013]	Mizrahi [2016]	Schiff [2016]	**Transactional SDN** [2017]
1.1	×	✓	✓	×	✓
1.2	✓	×	×	✓	✓
1.3	✓	✓	×	✓	✓

Table 4.3: Assessment of the state of the art proposals for the coordination of concurrent network updates and transactional SDN against the requirements REQ 1.1 to REQ 1.3.

4.7 Key Contributions of This Chapter

In this chapter we analyzed the research question RQ2 and its challenges C3 and C4. The main contributions are as follows:

Transactional SDN. We propose a novel SDN architecture with built-in support for coordination of concurrent network-wide updates. The architecture extends the controller with the transaction manager (TM) and the switch with the simple resource manager (RM). The latter implements locking as a concurrency control mechanism, thus isolating updates in SDN. The TM and the RMs communicate using 2-Phase Commit protocol, enforcing atomic execution of the updates in that way. While our mechanism provides atomicity, rigorous isolation guarantees and controlled durability of the network-wide SDN updates, including the cases of time-limited or self-expiring policies, it supports both distributed and standalone SDN controllers and integrates well with the existing consistency methods. Finally, our proposal offers novel intuitive API that the developers can use and profit from the transactional guarantees without having to care about them. With the proposal of transactional SDN we tackled the challenge C3.

Analysis of pessimistic concurrency control in transactional SDN. To study the limits of transactional updates in the SDN context, we choose the most rigorous concurrency control method with the highest isolation guarantees, which locks the en-

tire switch for reading and writing until the end of the transaction. Our simulation results show that, although the resource locking can cause rejection of the concurrent updates and hence deteriorate the overall rate of successfully installed updates, enforcing multiple update installation attempts or increasing network redundancy result in almost 100% successfully installed updates. Additionally, we find out that even with small path redundancy, the flow setup latency stays reasonable low: notably, we can set up almost all flows that arrive at a rate as high as $1000s^{-1}$, while providing to each update a delay of only few milliseconds. With this analysis we tackled the challenge C4.

After tackling the challenges C2, C3 and C4 we can answer the research question RQ2, in which we conclude that SDN can coordinate concurrent network-wide updates in a scalable manner by integrating mechanisms from DBMS, such as 2PC and locking concurrency control. This novel design streamlines the application development by featuring an API call, which the SDN applications use to profit from the transactional update support. The proposed service is easy to implement, does not require any complex provisions on switches and scales well to higher traffic arrival rates and larger networks.

4.8 Statement on Author's Contributions

This chapter is based on the publications "SDN on ACIDs" by Maja Curic, Georg Carle, Zoran Despotovic, Ramin Khalili, and Artur Hecker, published at CAN, CoNEXT 2017 [28] and "Transactional Network Updates in SDN" by Maja Curic, Zoran Despotovic, Artur Hecker and Georg Carle, published at EUCNC 2018 [30].

The author contributed to the studies presented in the CAN CONEXT paper [28], EUCNC paper [30] and the adapted sections in this chapter. The author contributed to the study design. The author implemented the transactional SDN and carried out the overall evaluation process. Finally, the author contributed to the discussion and analysis of the obtained results.

In the following we describe changes between this chapter and the study on transactional SDN presented in [28] and [30]. Most of the analysis in the EUCNC 2018 paper is present in this dissertation. We extend the discussion on the DBMS related work in Section 4.1, provide more detailed reasoning on the design choices in Section 4.2 and add details regarding the implementation of the transactional SDN in Section 4.3. Additionally, we include in this dissertation the larger version of the extensive evaluation of transactional SDN, which was not feasible in the constrained space of the above mentioned papers.

Part III

Admin or Developer: Solving the SDN Dilemma

Chapter 5

Distributed SDN Controller

Physical distribution of the SDN controller has two main benefits. First, it increases scalability as each distributed controller instance processes a portion of the events from the network. Second, it improves the reliability: if a distributed controller instance fails, its controlled switches can fail over to another instance. Our goal in this chapter is to understand whether the controller distribution has implications that affect application development, administration effort and performance.

This chapter is structured as follows. We first survey the state of the art distributed controllers in Section 5.1. In Section 5.2 we define a set of requirements that a distributed controller must fulfill to become popular with its end users, the administrators and the developers. We conclude this chapter in Section 5.3 by assessing the state of the art controllers against the requirements. Finally, Section 5.4 lists the key contributions of this chapter.

5.1 Overview of Distributed SDN Controller Designs

The distributed SDN controller runs as a set of instances that can be added or removed without disrupting the system and that operate jointly to create what appears to the applications as a single, logically centralized controller. The state of the art distributed controllers replicate the network state across the instances. The replicas are stored either within the controller instances [104, 56, 87, 47] or in a distributed storage system [133, 72, 15, 94], usually collocated with the instances. The operation of the instances is coordinated to achieve one of the following guarantees:

- *Strong consistency* of the replicated network state, which guarantees that a read from any controller instance at certain time always returns the most recent network state. A network state update performed by one instance must be propagated across the distributed controller, so that all network state replicas can converge to the most recent state. During the convergence process, all reads or updates of the state or a part of it, whose change is being propagated, must be delayed until all replicas have successfully converged. Hence, strong consistency pays the price of the increased reaction time to network events. Besides this, the CAP theorem [103] states that the replicated data can achieve only two out of three following guarantees: strong consistency, availability and partition tolerance. This means that opting for strong consistency of the network state replicas in distributed controller instances implies sacrificing one of the remaining two factors - availability or partition tolerance. Since the failures of the instances or the links interconnecting them cannot be avoided, the distributed controller must provide partition tolerance. Therefore, besides compromising performance with delayed reaction to network events, selecting strong consistency for the network state also compromises the controller's availability.
- *Eventual consistency* of the replicated network state, which achieves high availability and informally guarantees that if no new updates are made to a given network state all instances will eventually converge and return the latest network state. The network state reads and updates are performed instantaneously. Therefore, two instances can read or update the network concurrently and in an uncoordinated manner, which can easily lead to read or write conflicts and put the network in a false state.

In a nutshell, opting for one of the consistency guarantees brings to the fore the fundamental trade-off that affects either the performance or the application logic complexity [82]. If the network state is strongly consistent, all network state reads and writes experience a non-negligible delay [118, 1], which leads to a sluggish network and dissatisfied network administrators. On the other hand, if the network state is eventually consistent the applications are at risk to observe the stale data. If so, the application developer must have a profound distributed systems understanding and address common complexities of distributed programming, such as consistency, synchronization, concurrency and coordination [141]. We analyze the trade-off later in more detail in Section 5.3.

Next, we provide the overview of state of the art distributed SDN controllers, which are divided in two categories based on the physical organization of their instances. We show the two categories in Figure 5.1. The first category represents *flat design proposals*, which horizontally partition the network into domains, each controlled by one instance.

The second category represents *hierarchical design proposals*, which vertically partition the control plane into multiple layers, each supporting a particular set of services. For each proposal, we specifically focus on understanding the mechanisms used to coordinate the instances, as well as the consistency level between the network state replicas.

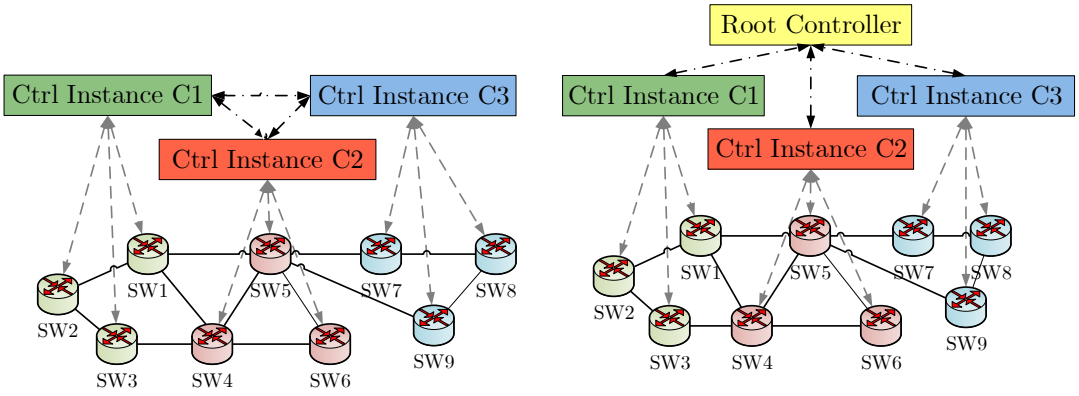

Figure 5.1: Flat vs. hierarchical distributed controller design.

5.1.1 Flat Controller Design

In this subsection we provide the overview of the flat SDN controller proposals.

5.1.1.1 HyperFlow

Tootoonchian et al. proposed the SDN application HyperFlow [133], which binds several instances of the single-image controller NOX [132] to form a distributed controller. The HyperFlow application runs in each NOX instance. It implements the westbound interface, which enables interaction between the instances, e.g. to synchronize the network state replicas or to reach the switches in remote domains. This inter-instance communication in HyperFlow is achieved using publish/subscribe messaging paradigm [104], which is implemented on top of the eventually consistent distributed storage system WheelFS [126]. More specifically, HyperFlow defines several channels and the instances can publish and subscribe to each of them.

Each HypeFlow instance directly controls the subset of the switches that make up its own domain and reach the switches in the remote domains over other instances. If an instance aims to reach a switch in a remote domain, it publishes the control messages to the channel of the instance that controls the remote domain, which then forwards them to the switch. Similarly, the remote instance publishes the response message from the switch to the channel of the instance that the message is targeting.

HyperFlow synchronizes the network state replicas as follows. When an instance receives an event notification from its domain, it updates its own network state replica and forwards the notification to the applications, which have the handler for such an event, for further processing. It then publishes the event notification to the data channel to which all instances are subscribed. The instances then receive the published event notification and replay its processing. To guarantee that the network state replicas eventually converge, the processing of event notifications in each instance must be identical and result with idempotent update. HyperFlow sets two prerequisites to achieve idempotent updates. First, the instances must run the exact same controller software and the same sets of deterministic applications. Second, the instances must process updates in the same order.

However, HyperFlow does not provide any internal mechanism for event ordering to achieve the second requirement. Hence, it cannot guarantee correct operation in the network with high rate of network events. The authors of HyperFlow suggest that the events in a 1000-node network should not arrive at a rate higher than tens per second.

5.1.1.2 Onix

Onix [72] is a platform that runs on one or more physical servers and implements the distributed SDN controller. It collects the network state by probing the network and stores it in a data structure called Network Information Base (NIB). The NIB is replicated and distributed across all instances.

Onix provides two types of storage management for the NIB data. The first is the transactional persistent database, supported by a consensus-based replicated state machine (e.g. Paxos [79]), which serializes all operations over the data and enforces strong consistency. The second is the one-hop memory-only distributed hash table (DHT), which enforces eventual consistency. To prevent system performance degradation, Onix suggest using strong consistency only for slowly changing NIB parts, e.g. network topology information, and the eventual consistency for the rapidly changing NIB parts, e.g. link load.

Onix exposes the NIB via its northbound interface, upon which the network management applications can be developed. The applications read and update the NIB, while Onix's internal mechanism maps the NIB updates to the underlying network infrastructure. Onix provides a synchronization primitive, which the applications use to register a listener on a data entity in the NIB. Once the update of the data entity has been pushed to the network, the application receives a callback.

Onix implements internal mechanism that guarantees serialized operation of the applications within the same instance, such that two applications can never update the NIB at the same time. However, it cannot guarantee that the NIB remains untouched by the applications that reside on other instances. For that reason, each application must provide its own solution for race conditions due to concurrent NIB updates from other instances.

5.1.1.3 ONOS

ONOS [15] is an open-source distributed SDN controller. The network state in ONOS is organized into logical units, called *stores*, which are replicated across the ONOS instances. The most important stores are:

- Mastership store – maintains mapping between the ONOS instances and their controlled switches.
- Network topology store – maintains information about switches, hosts and links.
- Resource store – maintains information about the allocated network resources, such as bandwidth, VLAN IDs and MPLS labels, and their consumers.
- Flow rule store – maintains information about the flow rules installed in the switches.

ONOS implements the stores with eventual or strong consistency guarantees. For example, the mastership store must guarantee that each switch is controlled by exactly one ONOS instance at a time (so-called master), which is why it is implemented with strong consistency guarantees. The same applies for the resource store, since a resource can be allocated by only one consumer at a time. On the other hand, the network topology store and the flow rule store are both implemented with eventual consistency guarantees.

To achieve eventual consistency of network state, ONOS uses the distributed key-value store Cassandra [78], which implements timestamping and gossip protocol. An update of an eventually consistent store is disseminated across the cluster as follows. The ONOS instance, from which the update originates, first applies the update to the local store. It then timestamps the update and broadcasts it to other instances in the cluster, which must apply it to their local stores as well. The gossip protocol performs as follows: at fixed intervals one instance randomly selects another instance from the cluster and they synchronize their stores.

To achieve strong consistency of network state ONOS is using distributed storage system Zookeeper [60], which implements Raft protocol [101]. In Raft, each instance that hosts

replicated data has a shared database called "log". Each log contains the same read and write commands, to be executed over the data in the same order. To enforce the same execution order across instances, the execution of the commands is coordinated by the leader instance (other instances are called followers). All followers must forward their write commands to the leader. When the leader receives a write command from certain follower, it appends it to its log. The leader then forwards each appended command to all followers, instructing them to replicate it to their own logs as well. When the majority of the followers acknowledged replication of the write command, the leader flags the command as committed and applies it to the local data and instructs the followers to do the same. Finally, the leader informs the followers on the commit, which then apply commands to their store. Read commands are also forwarded to the leader. However, they are served without writing into the log.

Raft faces scalability issues as all reads and writes ultimately go through the Raft leader. This can cause serious imbalance of load in the system: it can easily overload the leader [9, 116]. To alleviate this issue, ONOS splits the key space of the strongly consistent network stores in several subsets, i.e. Raft shards (partitions), and runs as many Raft consensus clusters, each enforcing consistency of the assigned shard. By default, ONOS creates as many Raft shards, as there are controller instances. The shard k is maintained by controller instances k, $k + 1 mod N$ (N is the number of instances) and $k + 2 mod N$. ONOS then computes a hash of the keys from the strongly consistent stores. The hash range is $[1, N]$, and the hash value determines the shard, to which the key is assigned. (Note that the addition of a new controller instance initiates a new Raft shard and a new Raft consensus cluster, which triggers redistribution of the entire key space to the new range $[1, N + 1]$, during which the data from the stores is unavailable for reads and writes from applications.)

If an update spans several Raft shards and requires transactional semantics, ONOS uses 2PC protocol prior to consensus. The instance that issues such update request is called coordinator. The coordinator starts 2PC with a randomly selected instance in every involved Raft partition. Each Raft partition then performs consensus independently and informs the coordinator about the outcome. If the coordinator receives consensus agreement from each involved Raft partition, it commits the update.

5.1.1.4 SCL

Panda et al. in [104] argue that maintaining strongly consistent network state replicas across distributed controller cluster imposes severe drawbacks: it delays the controller's response time to events from the network, hinders its availability and, finally, the involvement of consensus protocols complicates its design. For these reasons, the authors

consider the constrained SDN environment in which the eventually consistent network state replicas are sufficient for correct network operation. Following conditions apply in such an environment:

- There is a quiescent period of time after each event, during which new network events do not occur.
- Reactive control applications, which respond to individual PACKET_IN events, are not supported. The network is using data plane mechanism that responds to changes in real time without consulting the control plane.

The authors propose SCL [104] (Simple Coordination Layer), an additional layer in SDN high-level architecture that can turn a set of identical instances of single-image SDN controllers into a distributed controller cluster. SCL introduces two new components into the SDN architecture. The first is the *switch agent*, which intercepts the messages sent by the switch and forwards them to all live instances in the controller cluster. Note that this requires the existence of the path between each switch and each instance in the SCL cluster. The switch agent also delivers the control messages from the SCL instances to the switch (without any ordering or consistency considerations). The second component is the *controller proxy*, which implements gossip protocol [32] on the controller's westbound interface to facilitate communication between the instances.

SCL does not maintain the network state in the same way as previously discussed distributed controllers HyperFlow, Onix and ONOS, which store the state and incrementally update it on each network event. Instead, each SCL instance maintains the log of the observed events, based on which they construct the network state when necessary. The switches stamp event notifications with the local sequence number, which is used to position the events in the logs in the same order how they occurred on the switch. The SCL instances periodically exchange the logs of observed events using gossip protocol. Therefore, if one instance did not receive a network event notification, e.g. due to short-term failures, it can eventually find out about it from the other instances in the cluster.

Additionally, SCL instances periodically probe the network to obtain the most recent network state. This way, they can reconcile the network state that they calculate based on the event log with the actual network state. The authors base the design choice to construct the network state by querying the network on the fact that the network state is an "open-world", in which the truth resides in the network elements and not in the controllers: if a link in the network fails or experiences congestion, its true state lies in its functioning and operation and not in the state maintained in the controller.

An event in SCL is processed as follows. When an SCL instance receives an event notification, it adds it to the log of the observed events and constructs the network state, without coordinating with other controllers. The control application, which has the handler for such event, processes it, i.e. it calculates the update based on the constructed network state and pushes the update to the network. This same processing occurs in each instance in the cluster, as they all receive the event notification. Therefore, each event in the network controlled by the SCL cluster with N instances, triggers event processing N times, as well as N network updates. In SCL, each of these N updates must be idempotent. To achieve this, similar to the requirement of HyperFlow from Section 5.1.1.1, the instances in the cluster must run the same controller software and the same set of deterministic SDN applications.

In SCL the event response time corresponds to the time when first of the N idempotent updates reaches the network. However, the repeated installation of idempotent commands is not completely harmless for the network. Katta et al. demonstrated in [66] how that can lead to incorrect network behavior.

5.1.2 Hierarchical Controller Design

In this subsection we provide the overview of the proposals of distributed hierarchical SDN controller.

5.1.2.1 Kandoo

Yeganeh et al. proposed Kandoo [56], a hierarchical 2-layered distributed SDN controller. The lower layer contains *local controllers*, each controlling one or several switches in the network, while the upper layer contains the *root controller*, which controls all local controllers. To increase availability and reliability of the system, the authors suggest physical distribution of the root controller using proposals such as Onix from Section 5.1.1.2 or HyperFlow from Section 5.1.1.1.

Each local controller collects and maintains the state of its respective domain and distributes it to the root controller, which constructs the global network state. The local controllers in Kandoo are proxies for the root controller. They host the applications that can operate based on the local domain state, such as local policy enforcer, elephant flow detection [6] and Link Layer Discovery Protocol (LLDP) [84]. Besides, they detect events whose processing requires insight into the full network state, such as routing and spanning tree calculations, and pass such events to the root controller using Kandoo events.

Kandoo achieves good control plane scalability if the events that require processing on the root controller occur rarely. If that is not the case, its authors recommend using some of the flat controller designs.

5.1.2.2 xBar

McCauley et al. [87] discuss the possibility to incorporate hierarchy into the SDN network by introducing two recursive building blocks in it. The first is the *logical xBar*, which is a programmable entity that can switch packets, e.g. an OpenFlow switch. The second is the *logical server*, a unit that performs all control plane communications for the xBar, e.g. forwarding table management.

The xBar holds two types of information, the state of the underlying network and its configuration, which specifies its behavior. The recursion is performed as follows: several xBars can be composed to form a larger xBar. In this process, the forwarding tables of the smaller xBars are combined to create a single larger forwarding table for the larger xBar. The logical server computes the configuration of its directly controlled xBars and programs them. A further aggregation of xBars and logical servers ultimately leads to the abstraction of the hierarchy in the network, although the network does not necessarily have to be physically hierarchical.

The authors suggest that the logical servers architecturally appear as logically centralized single machine, but in practice they can be distributed to achieve fault tolerance, e.g. using the replication techniques, as suggested in flat distributed controllers in Section 5.1.

5.1.2.3 Orion

Fu et al. proposed Orion [47], a two layered hierarchical controller to solve the path stretch problem [27], which can occur in previously analyzed hierarchical controller designs Kandoo and xBar. The problem arises due to the high computational complexity of the process for identification of the shortest routing paths. The path stretch problem manifests itself in the way that the communication between end hosts in large SDN network deployments is established over suboptimal paths rather than the shortest paths, which leads to suboptimal resource utilization.

Orion divides the network into domains, which are further divided into sub-domains, also called areas. The *area controllers* are controlling the physical switches from the assigned area and collecting link information from it. Unlike in Kandoo, which suggest one controller in the upper layer, Orion proposes several *domain controllers* in the upper layer, each controlling its assigned area controllers.

The area controllers store the domain state in the form of the shortest paths between all the end users in the area and between all end users and all edge switches in the area. They distribute this information to the domain controllers, which construct the global network state, a collection of the global shortest paths between all end users in the network. When a new communication request reaches an area controller, it first checks if the destination address is in the same area. If yes, it retrieves the intra-area shortest path from the domain state. Otherwise, it forwards the request to the domain controller, which retrieves the inter-area shortest path from the global network state and sends the routing information to the controllers of the involved areas.

The authors suggest that the domain controllers should synchronize their network state replicas using a distributed protocol. However, they do not detail about specific consistency guarantees.

5.2 Requirements

This section is based on the publication "FitSDN: Flexible Integrated Transactional SDN" by Maja Curic, Zoran Despotovic, Artur Hecker and Georg Carle, published at LCN 2019 [29]. The discussion on the contribution of the publication as well as the difference between it and the thesis text is provided in the Section 5.5.

We now establish the criteria that a distributed controller must fulfill to become popular with the end users, i.e. the network administrators and the SDN application developers. Besides the considerations on the application development complexity, which we analyzed in previous chapter of this thesis, we add three additional requirements. We believe that SDN must achieve highly positive scores along each of four following dimensions in order to establish itself as a technology that network operators readily embrace to solve their problems:

REQ 2.1: System performance, meaning not only short delays in responding to network events, but also high availability and resilience, as well as the ability to scale well with network sizes and geographical footprints.

REQ 2.2: System administration, meaning acceptably low involvement of human administrators in the course of both normal and exceptional system operation, e.g. failures.

REQ 2.3: Application development, meaning rich APIs available to the developers to easily and quickly develop their applications, as well as supporting the trial and error model that would attract the critical mass of developers.

REQ 2.4: SDN model richness, meaning that range of possible tasks to perform in the network, i.e. types of events to handle, should be as wide as possible, at least matching what OpenFlow [90] sets as a benchmark.

5.3 Conclusion

This section is based on the publication "FitSDN: Flexible Integrated Transactional SDN" by Maja Curic, Zoran Despotovic, Artur Hecker and Georg Carle, published at LCN 2019 [29]. The discussion on the contribution of the publication as well as the difference between it and the thesis text is provided in the Section 5.5.

In this Section we assess the state of the art distributed controllers against the requirements REQ 2.1 to REQ 2.4. In the discussion that follows, we only consider the flat controller proposals. We do not consider hierarchical controller proposals for two reasons. First, the hierarchical distributed controllers score poorly in the SDN model richness (REQ 2.4), since they are designed to fit the networks with specific traffic patterns, e.g. rare occurrence of events processed by the root controller in Kandoo, or to run only a limited set of applications, e.g. flow setup application in Orion. Second, they suggest distributing the root controller using flat distributed controller proposals and as such inherit the scores that the flat distributed controllers achieve along the four dimensions defined in REQ 2.1 to REQ 2.4.

In the discussion that follows we use following categorization of flat controllers:

- *Tightly coupled systems*, such as ONOS and Onix, which coordinate operation of instances using strong consistency protocols, e.g. Raft and Paxos.
- *Loosely coupled systems*, such as HyperFlow and SCL, which coordinate operation of instances using exclusively mechanisms for eventual consistency, such as publish/subscribe messaging and gossip protocol.

5.3.1 System Performance

In tightly coupled systems, the controller instances run strong consensus protocols to agree on the order of network state updates. Each write request requires several exchanges between the consensus protocol participants, i.e. instances in the cluster. These exchanges introduce extra delay in responding to network events [104, 118, 49], i.e. require time that can be prohibitively high, especially when the cluster is geographically distributed [104, 101]. Besides, in consensus protocols only one participant, usually called the leader, can serve read requests. Other instances, usually called followers, sim-

ply forward the read requests to the leader, which can overload the leader and lead to under-utilization of the followers [9, 116]. ONOS aims to mitigate this by partitioning its strongly consistent stores into a set of shards and runs a small (with three nodes only), separate Raft cluster for each shard. Now, when an update spans multiple shards, ONOS uses the two-phase commit (2PC) protocol [138] to coordinate operation of the involved Raft clusters. That is, however, another source of delay, a fact also known to ONOS developers [62, 55]. This coordination delay is the reason, why the current ONOS implementation maintains certain stores as eventually consistent, even though they semantically require strong consistency [38].

In contrast, loosely coupled systems dispense with strong consensus protocols. At a first glance, this permits a controller instance to quickly respond to a network event, as it does not need to coordinate with others. However, as is the case with SCL and HyperFlow, each instance should locally replay processing of every event from the network in order to arrive at the correct network state. But therein lies a serious performance penalty, as such a strategy scales badly.

5.3.2 System Administration

Strong consensus protocols are complex and exhibit problems, as soon as they get out of their comfort zone and face failures. One should keep in mind that they are susceptible to failures of both running nodes (processes) and the transport network interconnecting them, as they require a full mesh connectivity between the nodes. There is a well-documented list of failure scenarios, in which Raft (or, at least, the available implementations thereof) fails at providing liveness and safety and leaves the system in a deadlock state, such as oscillating leadership [58], permanent split brain [100] and serious performance degradation [39], which can only be solved by a network administrator's intervention. It is important to stress out that these failure scenarios occur already in Raft clusters with five or even less instances, while the probability of a failure normally grows with the cluster size. For example, the studies of network-partitioning failures in Google's B4 network [51], CENIC network [134] and data center networks [50] show that they can occur once a week and may require minutes to hours to be repaired. Particularly interesting insights offer Alquraan et al. [8], showing that 21% of such failures in the transport network of the consensus clusters leave the cluster in a lasting erroneous state that persists even after the partition heals, and finally require intervention of the network administrator.

ONOS, with its "sharding" strategy, makes things even more problematic. Namely, every instance in ONOS is a Raft leader for a certain network state partition; thus, each and every node failure in ONOS will trigger an error-sensitive leader election procedure.

Even worse, a failure of two out of three nodes that constitute a default Raft cluster means that a majority of instances in a cluster have failed, bringing that cluster to a state that requires human intervention.

SCL and HyperFlow tolerate failures much better, as failing nodes or the network do not render the whole controller unusable. In particular, failures in the transport network are not a problem for SCL, because it uses a gossip protocol to disseminate network events among the instances. Gossip protocols are normally robust to network disruptions [32]. This is less the case for HyperFlow, as it relies on a publish/subscribe system for that purpose: these are less robust to network failures. Failing nodes are not a problem either: SCL connects all switches to all instances, so failure of an instance, or of a modest fraction of these, is not an issue. In turn, this strategy represents a severe management problem, as it requires a setup and maintenance of a fairly complex control network that is supposed to provide the said switch-controller connectivity.

5.3.3 Application Development

The existing SDN controllers (all, not only distributed) do not provide an API for exception handling capable of informing the application about the update installation outcome [77, 105]. Yet, there are a number of causes that can lead to incorrect configurations at runtime, such as switch hardware failure [143], priority bugs [65, 75] and switch software bugs [76, 77]. It is then the application developer who has to address this e.g. by verifying if the new network state matches the requested change and bring the network to the previous state in case it does not. (Note that this again requires verification of the new state.) This complicates the life of a developer, as it becomes necessary to know controller internals [73], a fact that hardly helps a wider SDN adoption. Alternatively, after each update there is a possibility to rely on mechanisms, such as Monocle [106], RuleScope [19], RuleChecker [145] or ATPG [144], which can combine active probing and test packets to verify rule existence on switches. However, as these proposals are mainly focused on periodical monitoring of static flow tables, they would face serious performance degradation and low scalability in SDN networks with high dynamics.

Loosely coupled distributed controllers complicate the matter even further. For example, when an SCL instance reads from its (locally maintained) network state, it may not be aware of the effects of possible more recent writes, just performed on another instance. Such stale reads can lead to conflicts that have been long known in the database management systems (DBMS) as lost update, inconsistent read and dirty read [48]. The complaints from the DBMS application developers which build the applications on top of the eventually consistent data have been reported in [82, 26, 125]. Corbett et al.

address this in [26] and present the Spanner, Google's globally distributed database that supports distributed transactions. They claim that application developers cope better with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions. Similar observations have been reported in the SDN in [82] and several studies have investigated the negative impact of stale and inaccurate network state on QoS-based path selection [53, 123].

Interestingly, SCL tends to eliminate the problems that emerge due to eventual consistency by constraining the ways the network should be used or, again, by transferring the problem to the developer. Both SCL and HyperFlow allow only deterministic applications which guarantee idempotent updates and removes any randomness from the application logic.

Tightly coupled distributed controllers, ONOS and Onix, both aim to facilitate the application development by maintaining strongly consistent replicas of the network state. However, they still trade-off application development complexity for performance, by enforcing strong consistency only for the parts of the network state. There is, however, a difference between ONOS and Onix when it comes to selecting which network state parts leverage strong consistency. ONOS opts for network state parts with low tolerance to inconsistencies, such as the resource and the mastership stores. The stale reads of such stores can lead to incorrect network behavior for which there does not exist any straightforward detection method. For example, because of stale reads, two applications may route two flows along the link whose bandwidth can support only one of them. This problem can only be detected once the congestion occurs. Therefore, to prevent such occurrences, ONOS selects strong consistency for the resource store. At the same time, it implements the topology store with eventual consistency. Onix, on the other hand, focuses mainly on having the least impact on the controller's performance and proposes the opposite: strong consistency for network state parts that are rarely changing, such as topology information, and eventual consistency for the rapidly changing network state parts, such as link load. In this case, stale reads of available network bandwidth are possible, which can easily lead to previously described link congestion. Therefore, between the two, ONOS provides more favorable environment for the application development.

5.3.4 SDN Model Richness

Networks controlled by a loosely coupled distributed controller, SCL and HyperFlow in particular, are subject to a number of severe constraints in order to guarantee error-free operation. The first is the maximum acceptable level of dynamicity. For example, a HyperFlow cluster with ten instances limits the controlled network to send up to 1000

events per second. SCL, similarly, cannot handle high event rates from the network. To guard against them, SCL proposes a range of mechanisms that proactively set up paths in the network: TeXCP [64] and MATE [36] and methods for the switches to choose between these paths. The generality of these solutions is fairly questionable. The second constraint is that SCL and HyperFlow support only deterministic SDN applications. Namely, all instances are presumed to process all network events equally and at the same time. This is to ensure that all instances converge to the same state over time. However, this calls for strictly idempotent network updates, or requires additional distributed machinery to check, if an update is being repeated or not. Moreover, even if the updates are idempotent, the authors of Ravana [66] show that they can lead to incorrect network behavior.

In contrast, tightly coupled distributed controllers do not impose such constraints. They are designed to be able to converge whatever is the request arrival order and semantics.

Table 5.1 summarizes the above discussion.

REQ		*Tightly coupled* Onix [2010] ONOS [2014]	*Loosely coupled* HyperFlow [2010] SCL [2017]
2.1	System Performance	×	✓✓
2.2	System Administration	×	✓✓
2.3	Application Development	✓	×
2.4	SDN Model Richness	✓✓	×

Table 5.1: Assessment of the state of the art distributed flat SDN controllers against the requirements REQ 2.1 to REQ 2.4.

5.4 Key Contributions of This Chapter

This chapter addressed the research question RQ3 and its challenge C5. We surveyed the state of the art distributed SDN controllers and analyzed how they maintain the network state replicas across the controller cluster. We specified a set of criteria that a distributed SDN controller should fulfill in order to make it popular with the end users.

In the following we list the key contributions of this chapter:

Overview and assessment of distributed controllers. We defined system performance, system administration, application development and SDN model richness as four dimensions along each of which a distributed SDN controller must collect highly positive score, in order to become appealing to its end users, i.e. the SDN application developers and the network administrators. Our findings show that the state of the art

distributed controllers fail along at least two of the four said dimensions, which is the direct consequence of the network state replication in the instances.

By surveying the state of the art distributed controllers and their design choices regarding the network state in the controller's instances, we obtained valuable insight that is necessary to further tackle the challenge C5 in the next chapter.

5.5 Statement on Author's Contributions

This chapter is partially based on the publications "FitSDN: Flexible Integrated Transactional SDN" by Maja Curic, Zoran Despotovic, Artur Hecker and Georg Carle, published at LCN 2019 [29].

The author contributed to the study presented in the LCN paper and the adapted sections in this chapter. The author designed the FitSDN. The author implemented the FitSDN and carried out the overall evaluation process. Finally, the author contributed to the discussion and analysis of the obtained results.

In the following we describe changes between the Section 5.2 and Section 5.3 and study on FitSDN presented in [29]. We use the section Background from the paper [29] in this section. We add more detailed analysis of how the state of the art controllers fulfill the requirements that we pose as a challenge for the distributed SDN controller, which we had to shorten in the constrained space of the LCN 2019 paper.

Chapter 6

FitSDN: Flexible Integrated Transactional SDN

This chapter is based on the publication "FitSDN: Flexible Integrated Transactional SDN" by Maja Curic, Zoran Despotovic, Artur Hecker and Georg Carle, published at LCN 2019 [29]. The discussion on the contribution of the publication as well as the difference between it and the thesis text is provided in the Section 6.6.

In this chapter we present Flexible Integrated Transactional SDN (FitSDN), a novel distributed SDN controller specifically designed to overcome the trade-off between consistency and performance that the state of the art distributed controllers face, which we discussed previously in Section 5.1. FitSDN builds upon transactional SDN from Chapter 4. It achieves the ease of development and administration, as well as good performances.

This chapter is organized as follows. We describe the concept behind FitSDN in Section 6.1. Afterward, in Section 6.2 we detail about its design and architecture, as well as its operation and implementation. Section 6.3 presents our simulation and emulation results, where we compare the performance of FitSDN to state of the art controllers ONOS and SCL. We conclude this chapter in Section 6.4, where we analyse how FitSDN fulfills our requirements previously defined in Section 5.3. Finally, Section 6.5 lists the key contributions of this chapter and Section 6.6 provides the statement on the author's contributions.

6.1 FitSDN Concept

We propose to eliminate state replication in the controller instances. Instead, our novel distributed SDN controller uses the state from the switches directly. To prevent conflicts, FitSDN uses transactional access to switches, which allow controller instances to perform their tasks atomically and in isolation from each other on the unique network state.

Figure 6.1 illustrates a high level FitSDN architecture. We list the main aspects below, together with their brief descriptions. The detailed descriptions of the key and novel aspects of the architecture are in the subsections to follow.

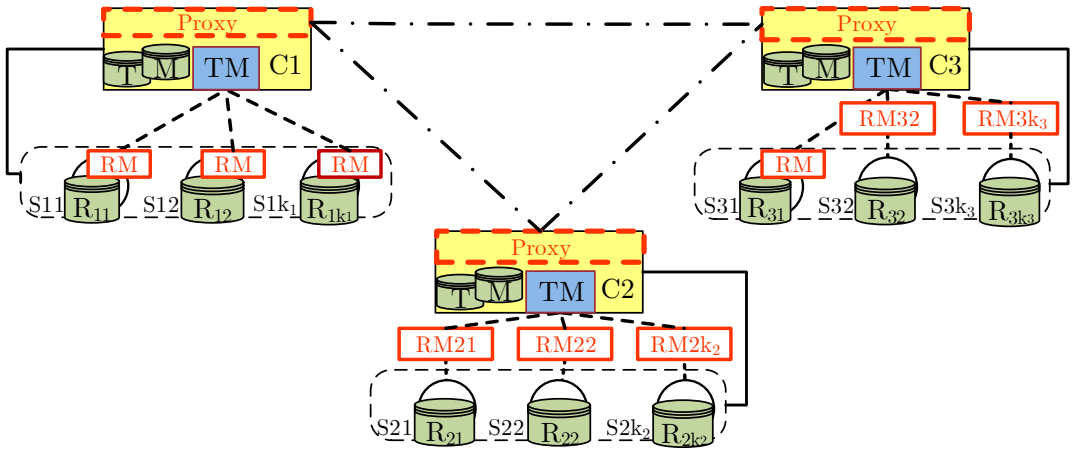

Figure 6.1: The distributed transactional control plane of FitSDN.

FitSDN is Flexible. Figure 6.1 shows multiple FitSDN controller instances, $C1$ to $C3$, in a network divided into non-strict subdomains. This contrasts the classical SDN deployments with distributed controller, e.g. with ONOS, where network division into subdomains is strict, i.e. each switch must be controlled by exactly one controller instance (so-called master). FitSDN supports this mode as well for the compatibility reasons. However, in general, in FitSDN all controller instances remain equal at any moment of operation, and several instances can be active for one switch. This is achieved using the OFPCR_ROLE_EQUAL option in OpenFlow 1.5, whereby each controller instance in this role has full access to the switch.

The goal of our design is to increase the degrees of freedom for the switches, such that they can connect to any number of controllers at will. On the one hand, this contrast to the classical OpenFlow SDN, in which the potential control conflicts are avoided by dedicating only one controller instance to control a switch. In FitSDN all connected controller instances can have full control to the switch (e.g. $S1k_1$ is controlled

by both $C1$ and $C2$) and the mechanism that we propose resolves the related control conflicts automatically. On the other hand, there is no strict requirement to connect all switches to all controllers, like e.g. in SCL, to converge to the same network state. If one instance does not have direct access to switches in some subdomains, e.g. due to connectivity or policy constraints, FitSDN enables such access by featuring simple proxies, which implement a stripped-down westbound interface (WBI). The information, which proxy to use to reach a remote switch is available in the FitSDN mastership store of the requesting instance (see below). For example, switches $S21, S22, \ldots, S2k_2$ form a subdomain exclusively controlled by instance $C2$, so $C1$ and $C3$ would need to send requests to it to read or update the state of e.g. $S21$.

Finally, with this flexibility, FitSDN improves the overall resilience of the controller and simplifies the administration compared to state of the art controllers: the addition of a new controller instance requires launching the instance and configuring some switches to use it, regardless of their previous control relationships.

FitSDN is Integrated. FitSDN adopts the notion of data stores from ONOS and organizes the network state into a set of independent stores, among which mastership, network topology and resource stores are the most important. They are denoted by symbols M, T and R in Figure 6.1. In contrast to prior art, the resource store in FitSDN is globally unique, i.e. it is *not* replicated in the controller instances. To realize this, FitSDN integrates the resource store within the network elements and maintains it as a distributed transactional database. The resource store includes the content of the flow tables, meters, bands, etc. In short, it comprises any data that an application can possibly modify in a switch. This is the part of the network state that pertains to network programming as such and contains data that change most often.

Whereas it is essential to integrate the resource store, i.e. not replicate it, FitSDN does replicate the mastership and the topology stores at the controller instances. However, both can operate under eventual consistency in FitSDN. This contrasts the design choice in ONOS, in which the mastership store is strongly consistent. We believe that this is not necessary: if a controller instance, due to the eventual consistency of its mastership store, were to see multiple other instances as possible masters of a switch, it could still relay a command over all of them. Only the instance that is the true master of the switch would relay the command to the switch, the others would discard it. In FitSDN the mastership store maintains the mapping between each switch in the network and all controller instances that have direct access to it. If an instance in FitSDN wants to send a command to a switch to which it has no direct access, it forwards the command via one of the instances (e.g. randomly selected) that do have direct access to the switch.

FitSDN is Transactional. FitSDN enforces transactional access to the resource store. This means that a controller instance can update a single switch or a set of switches atomically and in isolation from other instances. In the case of a network-wide updates, the affected switches do not necessarily have to fall into the same domain. To keep the state in the switches, i.e. the resource store, consistent in presence of concurrent updates from multiple controller instances, and to avoid problems such as lost update, dirty and inconsistent read of the resource store [48, 138], the total schedule of reads and writes over a switch must be serializable and recoverable. To achieve this, we adopt prior work in the area of DBMS as well as transactional SDN, which we proposed in Chapter4. Transactional SDN is natively suitable for distributed SDN control planes: it does not require any state synchronization between the transactional managers (TM) and resolves concurrency conflicts on the resource managers (RM). FitSDN, therefore, inherits transactional SDN architecture and integrates TM and RM in controller instances and switches, respectively. These are depicted as boxes labeled TM and RM in Figure 6.1.

The resource store in FitSDN is fetched from the switches when necessary. FitSDN must therefore support "read before write" data access patterns, which is why it is necessary to use a concurrency control algorithm that can keep track of the conflicting reads and writes over the resource store. In general, the RMs can run any concurrency control algorithm. As it is implemented on the switch, this in itself presents an interesting design question that trades off switch complexity for performance. However, we opt for a rather simple mechanism, which combines locking and validation protocol described in the Section 4.1.1. The former guarantees serializability and recoverability, and the latter supports "read before write" transactions. Our proposed concurrency control mechanism can be easily implemented even in hardware switches. We show that in spite of this simplicity, it achieves good performance.

Notice that some RMs are depicted in Figure 6.1 outside the respective switches (e.g. RM21, RM32). This is to demonstrate that FitSDN is backward compatible, i.e. it can support hardware switches that do not have RMs. (Note that we label such RMs with the ID of their respective switches in Figure 6.1. We do not do the same for the RMs that reside within the switches - these are label as RM only, as it is straightforward to identify for which switch they are responsible). Thus, if it is not possible to run the RM in its managed switch, the RM can run as a switch image in the controller or it can be implemented on a server attached to the control port of the switch. Figure 6.1 also illustrates various interfaces that FitSDN requires. The dashed lines present the transactional TM-RM interface. The dotted lines present the RM-switch interface, for switches without an RM. The solid lines between the instances and the domains present

a controller's southbound interface (SBI) such as OpenFlow (to keep the figure simple, we show only one line per domain). The dash-dotted lines between controller instances present the WBI, which the instances use to reach switches in remote domains. All these extensions are described in detail in Section 6.2.1.

6.2 FitSDN Design and Architecture

Next we present FitSDN, its transactional architecture, operation and implementation.

6.2.1 Transactional Architecture

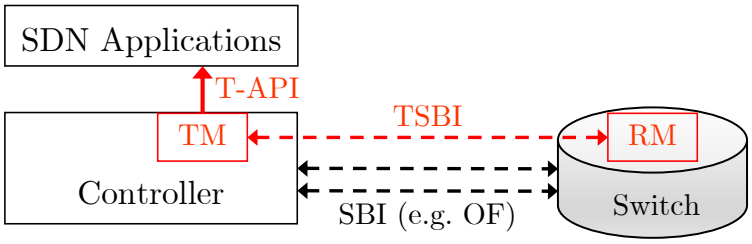

Figure 6.2: FitSDN transactional architecture.

Figure 6.2 shows FitSDN transactional architecture. We show in red the new entities that we added to the standard SDN model. These are the transaction manager (TM) that runs in each SDN controller instance, the resource manager (RM) that runs in each switch, the transactional SBI (T-SBI), an extension of the southbound interface for the TM-RM communications, and a transactional API (T-API), which is exposed to the applications. The details about these entities are given in Section 4.3. We next provide the details that apply for these entities in FitSDN:

1. **TM**. The controller instances run independent TMs, which do not require any coordination, except that the IDs they assign to transactions should be globally unique. We elaborate this in more detail below.

2. **RM**. The essential part of RM is a concurrency control mechanism (algorithm) that schedules concurrent read and write operations over the resource to achieve a desired level of their isolation. In FitSDN we use combination of two mechanisms from DBMS: validation protocol and locking. From the former we take over the three phases in which the transactional update is executed, i.e. read, validation and write phase. We apply the latter, i.e. locking, in the write phase, such that we lock the resource so as to enforce sequential write access to it. We later explain this in detail.

3. **T-API**. We assume that the TMs use ACP, such as 2PC, to coordinate the RMs. There are three T-API calls relevant for the application:
   - `startTransaction()`, to signal the beginning of a transaction.
   - `read(...)`, to retrieve parts of the network state.
   - `transactionalUpdate(...)`, to update the network, i.e. bring a set of switches to the desired state (both are the call parameters)

6.2.2 Operation

While previously described general and transactional FitSDN architecture apply to any FitSDN instantiation, the details we are about to provide now pertain to one specific instantiation, namely the one that uses the combination of validation protocol and locking for concurrency control in the RM.

In FitSDN all reads and writes over the resource store are executed on the respective RMs. Each RM validates whether the write phase of the transaction violates serializability. If the validation process succeeds, the locking mechanism takes over to ensure recoverability and cascadelessness. FitSDN locking mechanism adheres to a rather simple principle: when a write arrives at a switch, its RM tries to grant the lock over the switch to the requesting transaction. As simple as it may sound, it still has a number of implications, in particular for understanding the notion of transaction and choice of identifiers that are exchanged between the TM and RM, as well as stored at RM.

6.2.2.1 Transactions in SDN

FitSDN defines a transaction quite generally, but more narrowly than databases. A transaction is a two phase processing: in the first phase, an application reads the network state. These reads can be interleaved or followed by application specific processing (e.g. path calculation). In the second phase, the application updates the network state, i.e. writes to the appropriate stores. FitSDN enforces atomicity of the second phase only. So either all writes succeed or none. In contrast, reads are not atomic, neither on their own, nor together with writes.

The reason for this decision lies in the use of locking as concurrency control in the RM, whose goal is to prevent dirty reads from uncommitted transactions and a series of transaction rollbacks. Precisely, locking the switches at the beginning of a transaction, i.e. in the read phase, could lead to excessive lock durations, which would finally affect the performance. For that reason, FitSDN tries to reduce lock times as much as possible. Potential conflicts that this design may cause, e.g. an application reading a switch state that another application changes a moment later, are solved by holding in the switch

the ID of the transaction that updated it last. This ID is used to link the network state update from the second phase to a specific network state that was read in the first phase and used for its calculation. The RM then performs the validation check for each network state update and rejects those that are based on the stale reads. More details about this follow shortly. To support locking, the RM maintains a lock flag and sets it to the ID of the transaction that holds the lock.

There are three API calls relevant for the application. The first one, `startTransaction()`, is used by the application to signal the beginning of the said two-phase transaction cycle. With the second one, `read(...)`, the application retrieves needed parts of the network state. Finally, with the third one, `transactionalUpdate(...)`, the application supplies a list of switches to update, along with the new state to write (e.g. flow rules to install).

6.2.2.2 IDs in FitSDN

There are two types of IDs that are critical in FitSDN:

- The global transaction ID (GT ID). The GT ID is globally unique at the network level. The TM stamps all commands, reads and writes, within one transaction with the global transaction ID (GT ID) before sending them to the RMs. The RM reads the GT ID from the received commands and always includes it in its response to the TM. This way, the TM can identify the transaction that the response belongs to. FitSDN follows a simple way to achieve GT ID uniqueness: it assigns a unique ID to each TM. Each TM, in turn, assigns an ID to each transaction it handles, which is unique within its own scope and appends it to its own ID. The only purpose of the startTransaction() call is in fact to generate the transaction GT ID.
- The last write global transaction ID (LW GT ID). When the RM receives a write request, it stores the GT ID from it locally (but persistently). The RM thus identifies the transaction that updated the switch, i.e. the resource store, last. We call this ID last write GT ID (LW GT ID). It essentially identifies the current resource store state. Whenever replying to a read, the RM sends back the LW GT ID, along with the GT ID retrieved from the read. The TM uses the LW GT ID to associate the writes from the second phase of the transaction with the correct network state reads. In addition to the GT ID, the TM adds the LW GT ID to each write. When the RM receives the writes, it first starts the validation phase, in which it validates if the writes are based on the network view that still holds. That is why all write commands must be stamped with the LW GT ID. The RM rejects writes that carry obsolete LW GT IDs. Notice that the LW GT

ID essentially has the same purpose as the policy identifier from synchronization framework proposed by Schiff et al. in [120], which we previously described in Section 3.2.4.

6.2.2.3 Transaction Cycle in FitSDN

To put all this in use, we now go step by step through a typical transaction processing cycle, Figure 6.3 (steps 1-5) and 6.4 (steps 6-10). For the sake of simplicity we assume that the transaction updates switches that are directly connected to the controller instance.

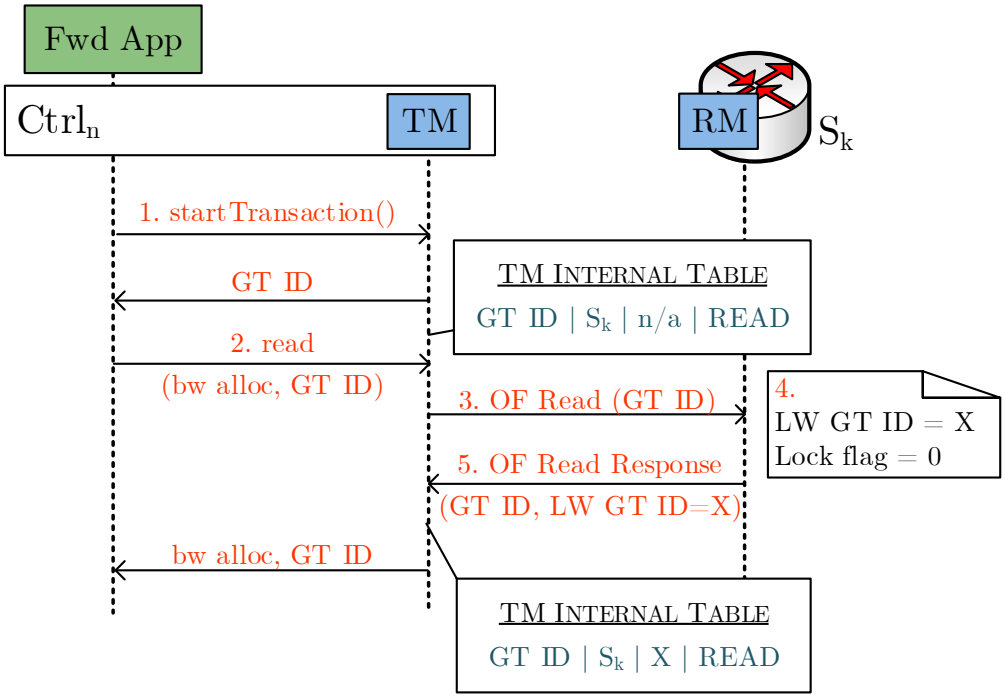

Figure 6.3: TM-RM message exchange for reading the network state, steps 1-5.

1. The application receives a network event notification that requires a reliable response and calls `startTransaction()` to receive a GT ID.
2. The application calls a `read(...)` on the TM to retrieve the needed data. The example from Figure 6.3 assumes that this data is bandwidth allocated over a link. The application also passes the GT ID as an additional argument.
3. The TM parses the call and translates it into one or more OpenFlow read commands, each stamped with the GT ID (see Section 6.2.3 for details). For each read command, i.e. each switch, the TM inserts a new entry in its internal table used to track the progress of the transaction. The table stores the information

about the exchanged commands in the transaction, such as their type, target SW ID, GT ID and LW GT ID. The LW GT ID is set to null at this moment, it will be updated later. Note that there is a possibility that the TM locally checks in the internal table whether any of the the switches involved in the current transaction is already locked by another transaction and, if yes, the TM can reject the transaction. In that case, the transaction processing would end at this step and the read commands would not be passed to the RMs.

4. The RM receives a read, reads the resource and then generates a response stamped with the GT ID from the read and the LW GT ID, which was stored internally.

5. The TM receives the response from the RM, parses the GT ID and uses it to find the correct entry in its internal table. Then it retrieves the LW GT ID from the response and updates the entry. It finally parses the response and delivers the requested data to the application.

6. The application formulates network update and calls `transactionalUpdate(...)` on the TM to execute it. It supplies its GT ID too.

7. The TM reads the GT ID from the update and sends a `Vote` to the involved RMs as the first message of the 2PC exchange. The `Vote` includes both the GT ID and the LW GT ID of the target switch (as retrieved from the TM internal table), as well as the actual network update.

8. On the reception of `Vote`, each RM locally locks the switch and sets the lock flag of the switch to the incoming GT ID. The switch thus becomes unavailable for other transactions - it discards all the commands that do not belong to the current transaction. The RM then compares the incoming LW GT ID with the stored one. If the two differ, the switch was meanwhile updated and the received update is based on an obsolete network state, hence the RM declares it invalid and rejects it. Otherwise, the RM applies the changes to the staging area (this does not affect the data packets traversing the switch). If this is successful, the RM sends a `Confirm` message to the TM (without activating the change). Otherwise, it rejects the update, sends `Reject` to the TM and unlocks the resource.

9. If the TM receives confirmations from all the RMs of the current update, it starts the commit phase by sending a `Commit` message to them. If at least one RM rejected the request or did not reply within the expected time, the TM aborts the update by sending `Rollback` to the RMs that replied with `Confirm`.

10. On `Commit`, the RMs activate their staging areas and update the LW GT ID of the switch, whereas on `Rollback`, they discard them. In both cases, they send

`Finished` back to the TM and release the locks. If the TM receives `Finished` from each switch involved, it considers the transaction as successful. Otherwise, if `Finished` did not arrive from each switch involved after certain wait period, the TM performs the rollback.

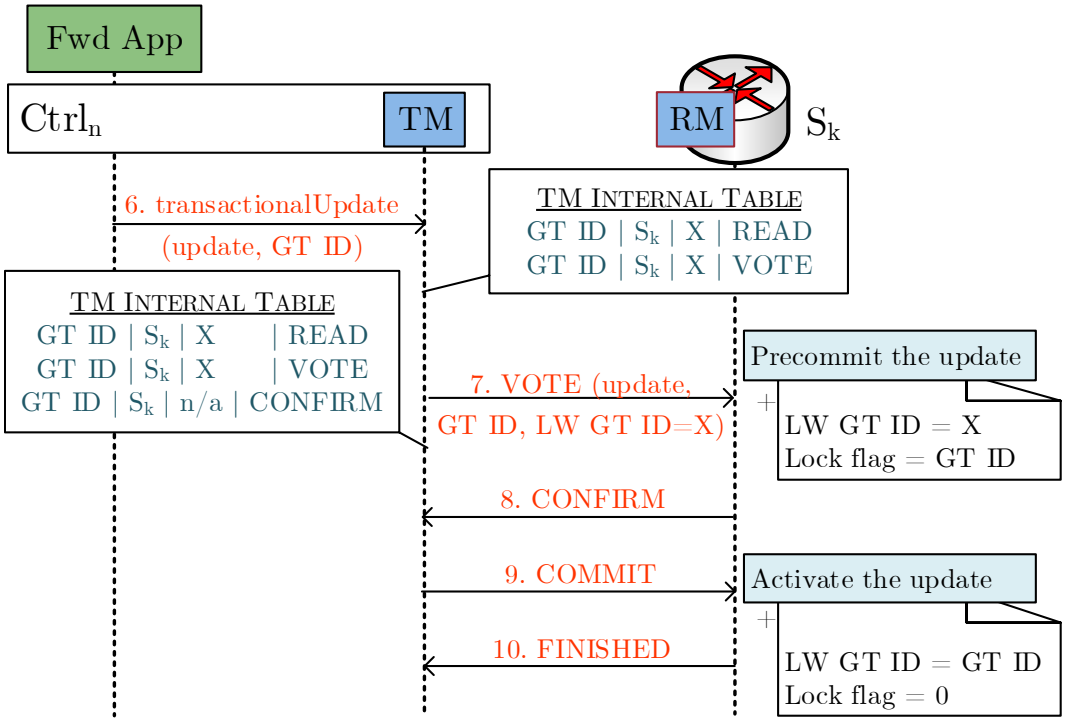

Figure 6.4: TM-RM message exchange for a successful (commit) transactional network update, steps 6-10.

6.2.2.4 Deadlock Handling

Deadlocks can happen in FitSDN only when processes involved in the 2PC exchange fail or cannot communicate. The scenario in which two or more transactions request and hold mutually blocking locks is not a problem, as their TMs will detect this by receiving at least one `Reject`, after which they both rollback the ongoing transactions.

If a 2PC coordinator, i.e. TM, fails just before deciding on the outcome of a transaction, the RMs may stay blocked forever, waiting for the `Commit` or `Rollback` from the TM. Similarly, if an RM fails just before sending its vote response to the TM, the TM cannot decide whether to commit or rollback the transaction and other RMs involved in the update will stay blocked forever.

To resolve the deadlocks, FitSDN uses timeouts in TMs and in RMs. A TM starts the timer for each 2PC primitive it sends. If the timer expires before the response arrives,

the TM concludes that the involved RM has failed and aborts the entire transaction. Similarly, an RM starts the timer, whenever it locks its resource. If the timer expires, the RM concludes that the TM has failed and discards the pending update.

6.2.3 Implementation

Our proof-of-concept implementation is based on OpenFlow. Our TM is a new module in the Floodlight controller [41] and the RM is an extension of the OpenVSwitch (OVS) [107]. TM and RM communicate in our implementation via the unchanged OpenFlow. This is to say that we did not modify OpenFlow to include ACP or transaction related messages, but instead we used existing OpenFlow messages for that purpose, just modifying how the endpoints interpret them.

Similar holds for the WBI. In FitSDN, the controllers exchange the unchanged OpenFlow, i.e. we do not implement any new protocol for the WBI. The proxy is a new module in the Floodlight controller, which relays messages received over the WBI to the target switch using the existing SBI and vice versa. More details on this follow.

6.2.3.1 T-SBI Implementation

We use the same set of OpenFlow messages with a specific command and tableId combination, as described in the Section 4.3.3.1. The only difference is that `Vote` message in FitSDN must carry the LW GT ID. We set the pad field of `Vote` to LW GT ID.

6.2.3.2 TM Implementation

Our TM is a new core service of Floodlight. It offers the three API calls to the applications, `startTransaction`, `read` and `transactionalUpdate`, interprets them and creates appropriate OpenFlow messages to send to the RMs. The TM implements its internal table shown in Figure 6.3 and Figure 6.4 as a Java HashMap in our implementation. It uses the map to track the progress of the transaction, where it adds a new entry for each message sent to RMs. The `transactionalUpdate()` can be called specifying the maximum number of installation attempts, MAX_N_INST. When an update installation fails, the TM retries after a random period of time, but at most MAX_N_INST times. To implement this, the TM maintains another HashMap, which stores the GT ID and number of installation attempts N_INST, initially set to 0 and incremented for each installation attempt. If it exceeds MAX_N_INST, the TM considers the update unsuccessful. The TM finally informs the application about the transaction outcome and removes the transaction related entries from the HashMaps.

The TM implements the interface IOFMessageListener of Floodlight, to be notified when the controller receives an OpenFlow message. Specifically, it registers to receive network

state read responses, e.g. `OFPMP_FLOW_STATS` and the T-SBI messages that it can receive from RMs, i.e. `OFPT_BUNDLE_CONTROL` and `OFP_ERROR_MESSAGE`, representing `Confirm`, `Reject` and `Finished`. SDN applications must register to use the TM. To register, the application creates the instance of the TM in the initApp() method. Strictly speaking, this renders the transactional updates unusable to old applications. However, this is just our current implementation choice.

6.2.3.3 RM Implementation

We extend OVS to include the following additions. The switch must:

- set and hold the last write global transaction ID (LW GT ID) as described,
- stamp the read responses with the GT ID and LW GT ID,
- set and hold the lock flag as necessary,
- reject lock attempts in locked state,
- reject the updates if based on obsolete state (when the LW GT ID in `Vote` differs from the current LW GT ID),
- perform updates in the staging area and,
- activate them on `Commit`.

We implement the LW GT ID and lock flags as 4-byte unsigned integers and add it to the ofproto, an OVS library that implements an OpenFlow switch. When transiting the state to locked, the switch sets the lock flag to the GT ID, which it reads from the xid field of the `Vote`. We use a mutex to provide exclusive access to the lock flag. So if concurrent `Vote` messages arrive, only one of them can successfully set the lock flag. To update the staging area, we split the existing bundle commit phase in two (method do_bundle_commit() in ofproto). In the first, the switch creates the staging area with a new version number, equal to the transaction ID, and installs the update in it. Note that we do not require any change in case of failure; the standard OVS code suffices. In case of success, our modified switch does not activate the new version immediately; instead, it holds back the activation, but it sends a `OFPBCT_COMMIT_REPLY` to the TM.

The RM implementation extends ofproto with two additional calls to process the messages from the second phase of 2PC, transaction_commit() and transaction_rollback(). They are called through the OpenFlow message handlers of OVS, handle_flow_mod(), that in turn is called from the main message handler of ofproto, handle_openflow(). We extend handle_openflow() to include a check if the lock flag is set. If yes, the switch discards all messages whose xid differs from the lock flag. The checks if the

received message should be interpreted as `Commit` or `Rollback` is implemented in handle_flow_mod(). In the case of a `Commit`, the switch activates the new version equal to the GT ID. Since this procedure is a single write operation on a pointer, we consider it atomic and assume that it cannot fail. The switch also updates the LW GT ID flag to the GT ID. In case of `Rollback`, the switch discards the staging area. Finally, the switch clears the lock flag.

6.2.3.4 WBI Implementation

We use the existing logic for SBI implementation from the Floodlight, namely classes OFSwitchManager and OFChannelHandler. OFSwitchManager bootstraps a new Netty server on port 6666 to handle the connections towards the controllers. In OFChannelHandler we add the method processOFMessageWestbound() to dispatch the received messages to the higher orders of control. Proxy Implementation: Our proxy is a new core service of Floodlight. It listens to the messages received over the WBI and from the switches. Therefore, similarly to the TM, the proxy implements the interface IOFMessageListener, to be notified when the controller receives an OF message. It registers to receive receive network state read responses, e.g. `OFPMP_FLOW_STATS` and the T-SBI messages.

6.2.4 Code Availability

FitSDN is free software, available to download from github [40].

6.3 Evaluation Results

We now evaluate FitSDN performances and compare them to ONOS and SCL. Throughout the evaluation, we use a simple SDN application for constrained flow path allocation, e.g. 1 Mbps for a flow. A series of requests to set up a flow between two switches arrive in the network as a Poisson process with parameter λ $1/sec$. The requests are processed by the controller instance of the originating switch (exactly one instance controls each switch). If a request is initially rejected in FitSDN, e.g. a switch is already locked, we use a backoff mechanism to reschedule it after a random delay, drawn from the exponential distribution with mean 0.01. We do not limit the number of processing attempts of a request, i.e. the controller instance reschedules its processing until it succeeds. We use the following three performance metrics:

- Request processing time: The time from the moment when the controller instance receives a request until the moment when it pushes the update to the network.

- Path setup delay: The time from the moment when the controller instance receives a request until the moment when the last concerned switch successfully installs the flow.
- End-2-end delay: The time from the moment when a source sends a data packet up to the moment when the destination receives it.

In Subsection 6.3.1 we report the request processing time of a single Floodlight, distributed ONOS and SCL controllers, and FitSDN. We use simulations for these tests rather than the existing implementations, as they are much easier to scale up to larger cluster sizes. It is important to emphasize that we do not simulate the overall controller operation for any of the observed controllers, but only the processing of one specific event. We increase the request arrival rate and whenever it reaches the processing limit of the controller (new requests start accumulating in the controller's queues and the queues are continuously increasing) we add a new computing resource to run in parallel to the existing one(s), i.e. a new thread in the Floodlight or a new controller instance in ONOS, SCL and FitSDN. To simplify our simulations, we assume that each instance of the distributed controller runs as a single-threaded server. This is a fair assumption, as we want to evaluate the effect of distribution on the observed SDN controllers. Ideally, we would like to observe that adding computing resources to the network can decrease the request processing time on average.

Finally, in Subsection 6.3.2 we report on the path setup delay and end-2-end delay of the implementations of ONOS, SCL and FitSDN for a small cluster size.

6.3.1 Request Processing Time

Figure 6.5 shows the request processing time when the network is controlled by centralized Floodlight and distributed ONOS, SCL and FitSDN controllers (referred to as "clusters"), and the request arrival rate varies from 200 to 3000 s^{-1}. The controlled network is random with the total of 72 switches and 180 links. We run 100 independent experiments for each of the 4 scenarios and measure in each the processing time of 1000 flow setup requests and calculate the 95% percentile thereof. This way, we end up with 100 values for each scenario.

We observe from the results that introducing new threads in Floodlight effectively reduces the request processing time. Each sharp drop shown in Figure 6.5 for Floodlight represents the moment when the request arrival rate reached the system capacity and a new thread was added. However, the same does not hold for the SCL cluster. The reason for this is that adding a new computing resource to SCL does not split the load; to reach the same state, each instance has to process each request from the network.

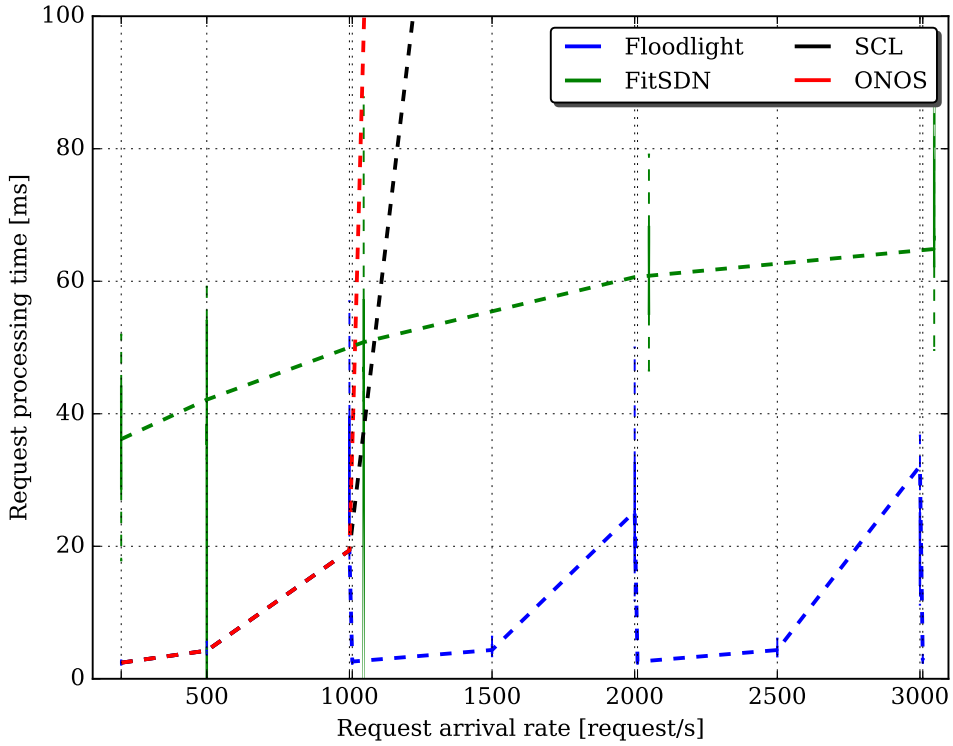

Figure 6.5: The 95th quantile of the measured sample of request processing times. For each point on the x axis we plot the total of 100 values of 95th quantiles measured in our simulation runs.

Also, adding new instances in ONOS does not improve the request processing time, as the benefits are offset by an increasing delay of the consensus protocol itself.

Unlike that, adding computing resources to the FitSDN cluster helps, as the the request processing time remains low. We observe a mild, sub-linear increase of the delay processing time, indicating that FitSDN scales well with the network load. Figure 6.6 plots the number of used FitSDN instances in each observed point from Figure 6.5. We see that the FitSDN cluster size scales linearly with the input load.

In an attempt to precisely understand how a new instance in FitSDN reduces the request processing time, we fix the request arrival rate at 2000 requests/sec and measure the processing time for different cluster sizes. (To support this rate, the cluster must have at least 13 instances.) Figure 6.7 shows that two new instances in the FitSDN cluster can reduce the processing time on average to around 50% of the initial value, measured for the minimal cluster size. The rate of reduction gets smaller as the cluster grows.

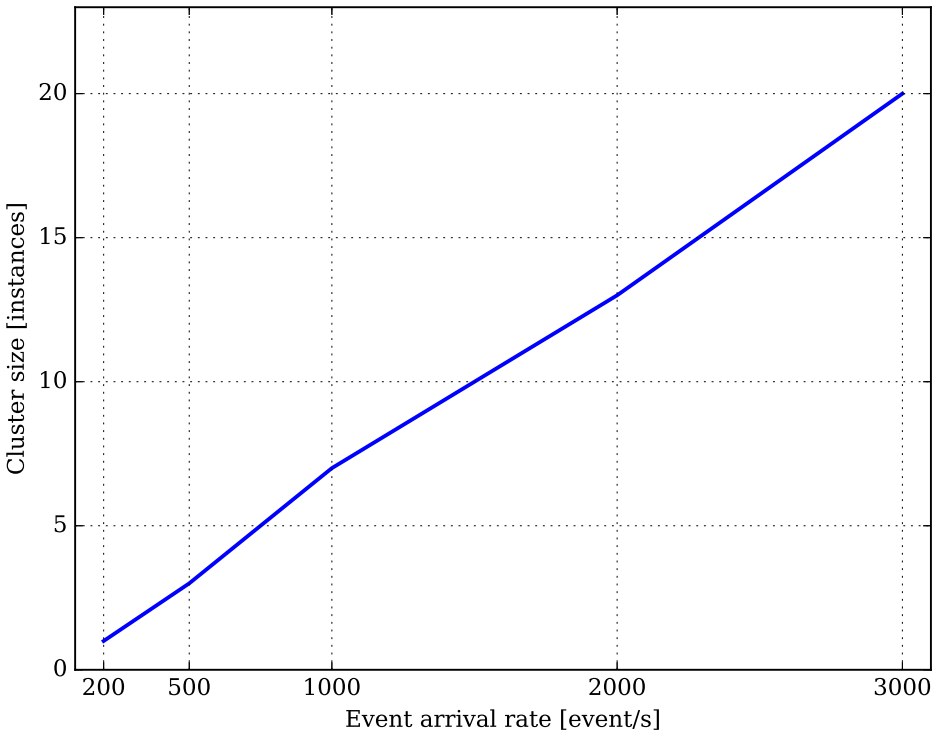

Figure 6.6: Minimal size of FitSDN cluster to process the given arrival rate.

Finally, we would like to understand the impact of the network size on the processing delay. Therefore, we now include the larger random networks, with 156 and 272 nodes into the picture. It is however important to recognize at this point that the bigger networks have more switches with users. To achieve fair comparison, we must therefore scale up the update arrival rate for the larger networks. It thus becomes $3000 \times 156/72 \approx 6500$ for the 156 nodes network and $3000 \times 272/72 \approx 11340$ for the 272 network. We must also scale up the cluster size to be able to process the new arrival rates. Each arrival rate increase of 1000 requests/sec requires adding 8 new instances in the controller cluster. Hence, to support the arrival rate of 6500 and 11340 requests/sec we use clusters with sizes 47 and 88.

We show in the Figure 6.8 the 95th percentile of the processing time of 10000 flow setup requests for each tested network. We observe from the Figure 6.8 that the 95the percentile increases just slightly as we scale up the network size. The explanation is as follows. As the network size grows, the length of shortest paths in the network grows as well. Because of that, each individual update takes longer, which increases

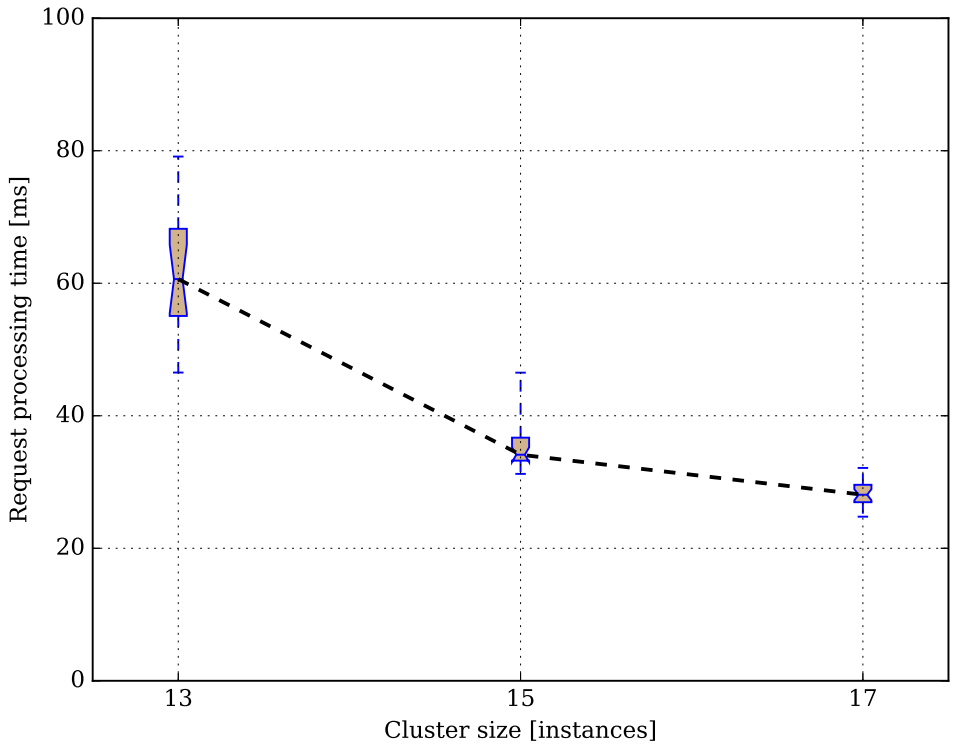

Figure 6.7: Request processing time for $\lambda = 2000$ requests/sec in FitSDN cluster of size 13, 15 and 17.

the probability that two network updates overlap in time. Therefore, the flow setup rejection rate increases and more updates require more installation attempts. However, the individual update increase is not long enough to deteriorate the overall performance. We therefore claim that FitSDN scales well with respect to the network size.

Another factor that may impact FitSDN performance is the network topology. As the topology is getting more centralized we might expect deterioration of the performance, because more requests will target the same node(s). We performed tests to understand to what degree is this harming FitSDN and, similar to our findings from Section 4.4.4 and Annex A, we found that the distribution of betweenness centrality of nodes [96] and the distribution of the shortest path lengths affect the overall performance of FitSDN. As long as the underlying topology has even distribution of the betweenness centrality and reduced share of very long shortest paths between nodes, we observe the same trends shown in Figures 6.5 to 6.8. The performance slightly deteriorates for more centralized networks (e.g. [95, 59]), but our main claim remains valid even in this case:

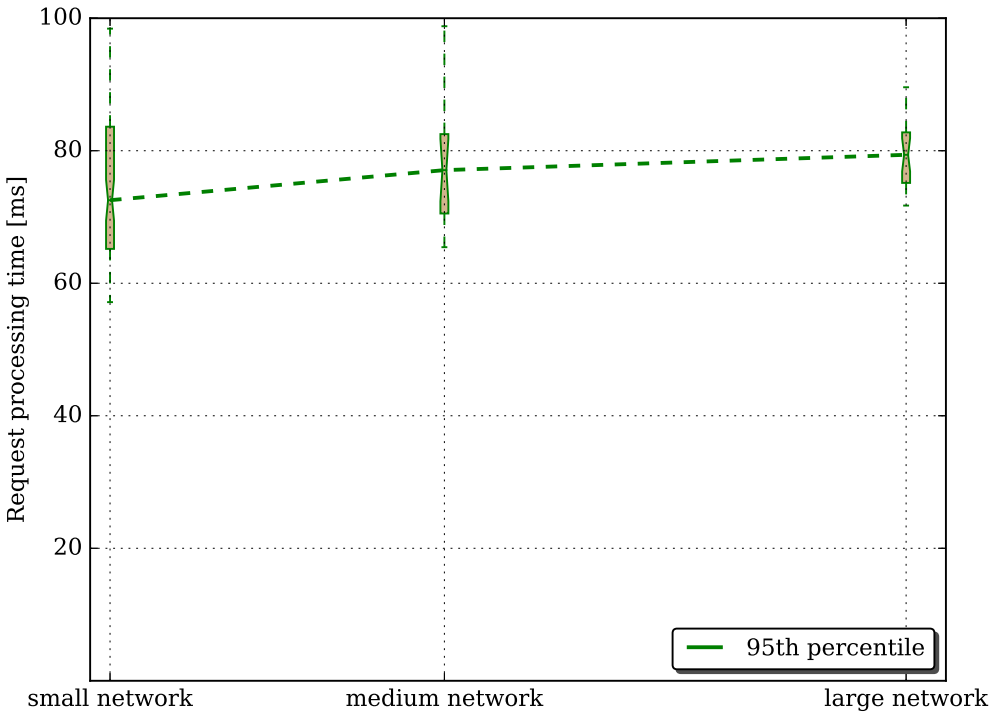

Figure 6.8: The 95th percentile of the processing delay for the small, medium and large network. The update request arrival rates and cluster sizes were scaled up accordingly.

FitSDN efficiently uses available computing resources to scale well with network sizes and request update rates.

6.3.2 End-2-end Delay and Path Setup Delay of the FitSDN, SCL and ONOS Implementation

We now compare FitSDN to SCL and ONOS in a small and controlled, but realistic deployment. We take ONOS v1.11.2, our own mock implementation of SCL and our FitSDN implementation and, for each, create a cluster of three controllers, each running on a separate Huawei RH2288H server with 48 Intel CPUs and 1.5TB of RAM. We run our data plane in Mininet. Our test network has a linear topology of 9 switches, each running OpenVSwitch, v2.8.0. Each controller controls three switches. The results, averages of 10 repetitions, are shown in Table 6.1.

As expected, SCL has the smallest end-2-end delay along the 9-hop path, as its instances do not coordinate their updates. FitSDN is slightly worse, while staying within the general SDN model, unlike SCL. ONOS, which also supports general programming,

performs substantially worse, around 4 times worse in the median. This is due to ONOS inability to coordinate the installation of the flow rules, i.e. uncoordinated sequence of PACKET_OUTs and PACKET_INs to and from the switches along the path, as our Wireshark traces confirm. In more detail, the ONOS instance first writes the flow rules pushed by the application to its flow rule store; once there, they get automatically pushed to the switches, out of control of the application. In addition, ONOS does not provide acknowledgments to the application confirming installation of the flows in the data plane. If now the PACKET_OUT hits the first switch on the path before the flow rules are installed, a new PACKET_IN can emerge and the whole update cycle can repeat, essentially ending up with a double bandwidth reservation. In our experiments, we observed this behavior for each network update.

The path setup delay has no PACKET_OUTs and thus enables us to isolate the effect of consensus in ONOS.

System	E2E Delay			Path Setup Delay		
	Median	Min	Max	Median	Min	Max
ONOS	48.3ms	46ms	66ms	29ms	27ms	32ms
SCL	5.2ms	6.6ms	8.1ms	3.5ms	5ms	6.5ms
FitSDN	11.2ms	12.5ms	15ms	9ms	10ms	12ms

Table 6.1: End-2-end and path setup delay on a 9-hop linear topology controlled by ONOS, SCL and FitSDN.

6.4 Conclusion

We now briefly discuss how FitSDN responds to the requirements from Section 5.3. In previous Section we evaluated FitSDN performance (REQ 2.1), therefore the focus here is on the three remaining requirements.

REQ 2.2 System Administration. FitSDN is by design resistant to failures of the controller instances or the transport network interconnecting them. State updates of the eventually consistent data stores, e.g. mastership and topology stores, are disseminated using gossip protocol that is highly robust to failures. Besides, the connectivity between the instances is not a prerequisite for operation of a single FitSDN instance. In the extreme case, if all transport network that interconnects the instances in FitSDN cluster completely crashes, each instance will remain operational within its own subdomain.

FitSDN inherits easy scaling of the gossip protocol. Each newly launched FitSDN instance must contain the configuration file, with the information about the instance's initial contact point [78], i.e. already operational instance in the FitSDN controller

cluster. The new instance advertises itself to its initial contact point, which gossips this event to other members of the cluster. The initial contact point gossips also in the opposite direction: it passes the information about the current cluster setup and replicated stores, i.e. mastership and topology, to the new node. After certain period of time, the gossiping will result with the convergence of the cluster information in all operational FitSDN instances.

Alternatively, instead of having to insert the information about the initial contact point, the instances can build the so called resource interconnection layer proposed in [33], which operates on the principle of ID-based structured routing. In that case, each instance has unique identifier and the instances run self-stabilizing protocol inspired by linearization algorithm [102] that creates a routing overlay, in which each instance maintains the connection towards its direct neighbor in the ID space. When a new instance is added to the cluster, it randomly selects its own ID and starts the protocol to find out its neighbor in the ID space. This would enable fully autonomous initial bootstrapping, integration and recovery from link failures, such that they should not require human or central entity intervention.

REQ 2.3 Application Development. In FitSDN application developer does not have to consider various artefacts of the controller distribution, such as consistency of the network state and concurrency exceptions. The developer is also relieved from having to verify if the network state update pushed from the controller is indeed mapped to the network infrastructure. This task in FitSDN is delegated to the TM.

Consider, for example, the application that installs a path with bandwidth guarantees in FitSDN. The application first receives a PACKET_IN, which it interprets as a flow request. It reads the source and the destination IP from it and retrieves the shortest path between them from the network topology store. If there are several shortest paths, it randomly selects one of them. We next show in Listing 6.1 the pseudocode of the application part that installs the new path. (We assume usage of the try-catch block like in Java, and that the developer's choice is to have specialized exception handling).

```
1 long GT_ID = tm.startTransaction();
2 try {
3   availableBW = tm.read(Resource.BW, path, GT_ID);
4   if(availableBW >= requestedBW) {
5       try {
6            tm.transactionalUpdate(path, requestedBW, GT_ID);
7       } catch (UpdateExceptionType ue) {
8            process(ue);
9       }
```

```
10          sw.sendPacketOut(extractDataPacket(packet_in));
11      }
12  } catch (Exception e) {
13      process(e);
14  }
```

Listing 6.1: The code of the application for path installation with bandwidth guarantees in FitSDN

The application logic first calls the TM procedure `startTransaction()` to obtain the GT ID (line 1). Afterwords, it calls the TM procedure `read()` to retrieve the available bandwidth along the path. It passes the GT ID as a parameter to this call (line 3). From the developer's point of view, passing the GT ID makes the only difference when reading the network state in FitSDN compared to the same operation on state of the art controllers, such as ONOS or Onix - this way, the developer in FitSDN specifies to which transaction the read belongs. If the available bandwidth along the path is greater than what the application requires, i.e. requestedBW in our example, the program execution proceeds with the path installation. The application logic calls the TM procedure `transactionalUpdate()` and passes the update to it, i.e. the path and the requested bandwidth (line 6). The GT ID is also passed in this call to specify to which transaction the update belongs. When the path is successfully installed, the application pushes the data packet, which arrived in the initial PACKET_IN, to the network (line 10). This is done by encapsulating the data packet in PACKET_OUT.

The same application part in ONOS has at least twice as many lines of code. We show the pseudocode in Listing 6.2. Without any guarantees that the network update is indeed mapped to the network infrastructure, the developer itself must verify it the network state corresponds to what is expected. In our example the update is pushed to the network (line 7) and the verification of the network state is performed immediately afterwords (line 8 to line 17). If the update is not installed after the maximum waiting time, which is set to 20 ms in our example (line 1), the exception is thrown (line 13). This exception is processed in the catch block (line 17 to line 29), in which the method for uninstallation of the partially installed update is called (line 19). Note that the uninstallation process can also fail, which is a well-known issue in SDN [73]. Therefore, the application logic must verify if the uninstallation process was successful. For example, the application developer can specify the maximum waiting time until the uninstallation process should be complete. If the time is exceeded, a new exception is thrown (line 24). When such exception occurs, the network administrator can inspect the network element where the partial update was not uninstalled and try to solve the issue manually.

```
1  int maxWaitTime = 20;
2  int waitTimeMillis = 1;
3  if (hasEnoughBandwidth(path, requestedBW)) {
4     try {
5      allocateBandwdith(singlePath, bandwdith);
6      try {
7         installPath(singlePath);
8         int currentInstallWaitTime = 0;
9         while (!pathInstalled(singlePath)) {
10          Thread.sleep(waitTimeMillis);
11          currentInstallWaitTime += waitTimeMillis;
12          if (currentInstallationWaitTime > maxWaitTime) {
13             throw new Exception("MaxWaitingTimeExceeded");
14          }
15        }
16        sw.sendPacketOut(extractDataPacket(packet_in));
17     } catch {
18        uninstallPartiallyInstalledPath(singlePath);
19        int currentUninstallWaitTime = 0;
20        while (!pathUninstalled(singlePath)) {
21          Thread.sleep(waitTimeMillis);
22          currentUninstallWaitTime += waitTimeMillis;
23          if (currentUninstallWaitTime > maxWaitTime) {
24            throw new Exception("MaxWaitingTimeExceeded");
25          }
26        }
27        deallocateBandwdith(singlePath, bandwdith);
28     }
29   } catch() {
30     System.err.println("BandwidthAllocationFailure!");
31   }
32 }
```

Listing 6.2: The code of the application for path installation with bandwidth guarantees in ONOS

The application developer in HyperFlow and SCL encounters even more complexities, as the applications must be deterministic and guarantee idempotent updates from each of the controller instances. This means that the application developer must not introduce any randomness in the application logic. Consider the example where the application retrieves several shortest paths with the same cost from the topology store. The application logic cannot simply randomly select one of them and proceed with the flow set up. If so, the application instances hosted on different controller instances could all

choose different paths. As a result, there would be several paths installed in the network for the same flow, which can easily cause various data plane inconsistencies. Instead, the developer must create a specific criterion on ordering the retrieved shortest paths and selecting the one with specific order number. This is the only way to guarantee that all application instances hosted in the SCL controller cluster select the same path and issue the idempotent update.

REQ 2.4 SDN Model Richness. FitSDN does not constrain the SDN model in any way (e.g. by requesting idempotent network updates only), neither does it eliminate important use cases by permitting only a small range of input parameters (e.g. level of dynamicity). The only constraint imposed by FitSDN is its extension of the switch (RM). Given the relative simplicity of the RM and the benefits of this design, we believe that it is a good design choice. Besides, the development of OpenFlow testifies of similar trends in the past (e.g. bundles, time synchronization, etc). Our implementation shows that the change introduced in OVS switches requires around 500 lines of code.

Note that FitSDN, by using switches as storage media, does not just shift failure points from the controller to the switch. While FitSDN regulates how data on the switch can be accessed, any switch only holds the flow table data relevant to the flows traversing it, which is an operational requirement in any SDN. As this data is only pertinent to that switch, a failure of the switch is also equivalent to current data plane failures and must result in flow rerouting.

We now add FitSDN to the Table introduced in the Section 5.3.

REQ		*Tightly coupled* Onix [2010] ONOS [2014]	*Loosely coupled* HyperFlow [2010] SCL [2017]	**FitSDN [2019]**
2.1	System Performance	×	✓✓	✓✓
2.2	System Administration	×	✓✓	✓✓
2.3	Application Development	✓	×	✓✓
2.4	SDN Model Richness	✓✓	×	✓✓

Table 6.2: Assessment of the state of the art distributed flat SDN controllers and FitSDN against the requirements REQ 2.1 to REQ 2.4.

6.5 Key Contributions of This Chapter

This chapter addressed the research question RQ3 and its challenges C5 and C. The key contributions are as follows:

fitSDN. We designed and implemented a novel Flexible Integrated Transactional SDN (FitSDN), which eliminates the replication of critical data in the controller and rather maintains it in the switches and accesses it using transactional mechanisms. FitSDN is easy to manage: different failure situations are easy to handle and the controller scales out naturally in the sense that it does not require existing instance reconfiguration. It does not constrain the SDN model in any sense and delivers a transactional API to the application developers that is a comfortable abstraction to relieve the developer from having to foresee and watch for a broad range of possible exceptions in the distributed system. This finally lets the SDN application developers to focus on the application logic and thus tackles the challenge C5.

Performance analysis of FitSDN. We evaluated FitSDN in simulation and emulation environment and compared its performance with the performance of single-headed Floodlight controller and representatives of tightly and loosely coupled distributed controllers, namely ONOS and SCL. Our performance evaluation demonstrates that FitSDN, unlike current state of the art distributed controllers, scales well with network sizes and input load owing to its seamless integration of additional compute resources, i.e paying no price of distribution as such. FitSDN rather behaves as a set of single-image controllers that share their load. Adding new computing resources to the FitSDN reduces the request processing time, which does not apply for ONOS and SCL. Our measurements show that FitSDN achieves around 4 times faster responses to network events compared to ONOS. As expected, FitSDN does not outperform SCL in this category, as SCL does not coordinate updates but at the same time strongly constrains the SDN model, which is not the case with FitSDN. With this analysis we tackled the challenge C6.

After tackling the challenges C5 and C6 we can answer the research question RQ3, in which we can conclude that transactional access to a switch enables solving the SDN dilemma; our novel distributed SDN controller FitSDN achieves ease of application development and administration, as well as good performances.

6.6 Statement on Author's Contributions

This chapter is based on the publications "FitSDN: Flexible Integrated Transactional SDN" by Maja Curic, Zoran Despotovic, Artur Hecker and Georg Carle, published at LCN 2019 [29].

The author contributed to the study presented in the LCN paper and the adapted sections in this chapter. The author designed the FitSDN. The author implemented the

FitSDN and carried out the overall evaluation process. Finally, the author contributed to the discussion and analysis of the obtained results.

In the following we describe changes between this chapter and study on FitSDN presented in [29]. All of the content from the LCN 2019 paper is present in this dissertation. We add more details regarding the implementation of FitSDN in Section 6.2. In Section 6.3 we add the analysis of the impact of the network size on the processing delay in FitSDN. Finally, in Section 6.4 we add more detail discussion on how FitSDN fulfills the requirements that we pose as a challenge for the distributed SDN controller.

Part IV

Summary and Conclusion

Chapter 7

Conclusion

7.1 Results from research questions

In the following section we present results from the research questions as described in detail in Section 1.2.

RQ1: How does SDN improve flexibility of networks?

The goal of the RQ1 was to investigate the possibility to implement procedures from legacy networks as SDN applications and to check whether the use of dynamic programmability of SDN can optimize their network utilization. We select Mobile Core Network (MCN) as example network. To answer this research question we had to tackle the following challenge:

- C1: Investigate the possibility to implement a set of the procedures from LTE EPC in SDN-based MCN.

We now present the result of the tackled challenge.

CHALLENGE C1

To tackle the challenge C1 we focused on the location management (LM) procedures from legacy MCN, namely UE state management and paging. We show that it is possible to design and implement LM as SDN application. In addition, we show that by using programmability of SDN we can dynamically reduce the signaling load generated by these applications and adapt it flexibly to network conditions or operator requirements. Therefore, our SDN-based location management outperforms its legacy counterpart in LTE EPC.

RQ2: How can SDN coordinate concurrent network-wide updates in a scalable manner and streamline application development?

The goal of the RQ2 was to investigate the possibility to achieve a scalable built-in mechanism for coordination of concurrent updates in SDN and therefore relieve the SDN application developer from having to solve the issues in SDN that occur due to concurrency. To answer this research question we had to tackle three following challenges:

- C2: Investigate semantics of conflicts that can occur when multiple control applications simultaneously update the SDN network.
- C3: Model a novel SDN architecture that avoids concurrency issues by design.
- C4: Evaluate the performance of the novel SDN architecture when it uses pessimistic concurrency control.

We now present the result of each of the tackled challenges.

CHALLENGE C2

To tackle the challenge C2 we needed to collect and analyze concurrency conflicts that occur in SDN due to uncoordinated network updates. Without coordination, the commands from concurrent updates that target the same set of switches can be interleaved in time. This can lead to partial update installations or incorrect network behavior. The state of the art controllers, both centralized and distributed, do not provide any integrated mechanism to prevent concurrency issues, but rather transfer this burden to the SDN application developer. The developer therefore must use complex dedicated try-test-retry logic on every network-wide update, without actual guarantees of success. Besides, this can lead to longer, possibly wildly varying, network update installation delays. Finally, we draw a parallel to the DBMS systems and come to the conclusion that the conflict-serializable execution of the updates in SDN is a prerequisite for their correctness. The SDN would profit from a support for transactional network update logic with ACID properties, where network-wide updates are atomic, consistent, isolated and durable.

CHALLENGE C3

In the challenge C3 we first investigated the mechanisms that are used in DBMS to coordinate concurrent updates. We analyzed pessimistic and optimistic concurrency control as mechanisms that enforce different isolation levels for the updates, as well as the protocols that enforce atomicity. We then designed, implemented and evaluated the transactional SDN architecture, that parallels that of the DBMS, with transactional

manager (TM), which runs as a new service in the SDN controller and implements 2PC as Atomic Commit Protocol and resource manager (RM), which runs in the switch and uses simple locking mechanism to enforce the strictest level of isolation between the updates. The application developer uses the API provided by the TM to install updates in isolated and atomic manner.

CHALLENGE C4

In the challenge C4 we performed the extensive performance analysis of our transactional SDN architecture. Our results have shown that, although the resource locking can cause rejection of the concurrent updates and hence deteriorate the overall rate of successfully installed updates, enforcing multiple update installation attempts or increasing network redundancy results in almost 100% successfully installed updates. With the small path redundancy, the flow setup latency stays reasonable low: notably, we can set up almost all flows that arrive at a rate as high as $1000s^{-1}$, while providing to each update a delay of only few milliseconds.

RQ3: How can we solve the SDN dilemma and achieve a distributed SDN design that appeals to both SDN application developers and administrators?

The goal of the RQ3 was to investigate the possibility to distribute the SDN controller, without sacrificing the application development or system administration efforts. The state of the art controllers come with the number of design choices, such as network state replication, that are appealing either to the application developer or the system administrator, but never both. To answer this research question we needed to tackle following challenges:

- C5: Investigate the possibility to eliminate the replication of critical network state in controller instances.
- C6: Evaluate the performance of the novel distributed SDN controller and compare it to the state of the art controllers.

We now present the result of each of the tackled challenges.

CHALLENGE C5

To tackle the challenge C5, we built upon the transactional SDN, which we proposed while tackling previous research question RQ2, and designed and implemented Flexible, Transactional and Integrated SDN controller (FitSDN). The controller instances in FitSDN do not replicate any network state that requires strong consistency. Instead, such network state is integrated in the network elements. For this to work without conflicts, the controller instances have transactional access to the switches. FitSDN

adds TM and RM to the standard SDN model and incorporates them to the controller and the switch, respectively. The network update in FitSDN is divided in two phases. In the read phase, the state queried by the application is retrieved directly from the network. In the write phase the application uses the call for installation of the update with transactional semantics. The RM in FitSDN validates if the write is based on the stale read and uses locking during the write phase to prevent interleaving of writes from multiple updates. To demonstrate both the feasibility and the ease of realization of our proposal, we provide open source extensions to the state of the art controller (Floodlight) and switch (OVS), which we successfully test in a Mininet environment.

CHALLENGE C6

To tackle the challenge C6 we evaluated FitSDN in simulation and emulation environments and compared its performance to the state of the art controllers, standalone Floodllight, ONOS and SCL. We used our simulation to measure the request processing time as observed by the controller. We vary the request arrival rate and analyze the effects of adding new computing resources to the observed controllers. Our results show that adding new computing resources to FitSDN can maintain the request processing time low, even for the high request arrival rates. We did not observe the same for the state of the art controllers ONOS and SCL. Besides, we show that FitSDN scales well with network sizes as well. We compared our implementation of FitSDN with ONOS and SCL in the emulation. We measured path installation time and end-2-end delay observed by the users when there is no concurrency in the network and events arrive sequentially on the controller. Our results show that FitSDN cannot achieve faster response to network events compared to SCL, as SCL does not coordinate the updates in any way and constraints the general SDN model. However, FitSDN achieves around 4 time faster responses to network events compared to ONOS.

7.2 Future Work

In this dissertation we demonstrated the possibility to introduce transactional network updates in SDN and enable easy and intuitive APIs that permit the SDN application developer to focus on developing application logic, without being hindered by internals of the controller design. Although our thorough performance evaluation of transactional updates with locking-based concurrency control in the environment with centralized controller shows good scalability and low impact on the flow setup latency, and the analysis in the environment with distributed controller indicates that our novel controller built around transactional access to the switch even outperforms popular state of the art consensus-based controllers, there is still potential for future research.

In the future work, we plan to investigate the complexity and performance of other schedulers in RMs, most notably optimistic concurrency control techniques such as commitment ordering or serialization graph testing and possibly, of mixed strategies. Another promising possibility is the use of more fine grained resources, e.g. switch ports or flow tables, rather than the switch as a whole. Both should be able to improve the possible operational concurrency and, hence, improve the overall system throughput.

Finally, we plan to investigate, if our proposal can benefit from the "power of two choices" concept [92], where a network update would propose two alternatives, affecting completely or partially orthogonal resources, and successful installation means that only one of them does not conflict with any other concurrent installation.

Part V

Annex

Annex A

Extended Performance Evaluation of Transactional SDN

A.1 Impact of Network Topology on Performance of Transactional SDN

This section presents further evaluation of our transactional SDN architecture. In general, we generate a series of network update requests with transactional semantics and study how the network handles them. Specifically, we observe the following two quantities as our KPIs:

- Success rate: The fraction of requests that have been successfully processed.
- Request completion time (delay): The time elapsed from the moment the TM receives a request until the moment the last switch in the request receives the `Commit`. We measure the latencies of successful requests only.

Ideally, we would like to have a high success rate, close to 100%, and small and evenly distributed delays. All that, to the highest degree possible, irrespective of parameters of the environment, such as the request arrival rate and the underlying network size, etc. We will show in the following that this is in fact possible.

However, we constrain ourselves to network update requests with very specific semantics, i.e. our updates set up flows between randomly selected endpoints in the network. Thus

each update will touch switches along a path in the network.[1] The reason for focusing on those is that they are most critical for SDN, as in a typical SDN deployment the controller may be exposed to very high flow installation or update rates.

A.1.1 Evaluation environment

To have as accurate evaluations as possible, we use a rather complex evaluation environment that mixes simulations and Mininet based emulation. In essence, we simulate the full system operation, but learn and set up its relevant properties with help of our emulation, previously explained in Section 4.4.3.

We evaluate the transactional SDN on four different topologies, summarized in Table A.1 and described next. We characterize the topologies by two quantities that, as will be seen shortly, turn out to be critical for the performance: distribution of the shortest path length between vertices and distribution of *betweenness centrality* across vertices [96]. The latter is a measure of the extent to which a vertex in a graph lies on the shortest paths between other vertices.

Hierarchical topology. We use network topologies generated from [95, 59], representing backhauls of carrier networks. They are hierarchical and consist of three layers: access, aggregation, and core. They contains k switches at the core and k pods of size k switches, i.e. a total of k^2 switches, at the aggregation. Switch connectivity degree in aggregation and core pods is m. $k/2$ switches of each aggregation pod are connected to the access, with five access switches per each aggregation switch. The rest of $k/2$ aggregation switches from the pod are connected to the core, each of which is connected to n core switches.

***r*-dimensional (*r*-d) torus topology.** An r-dimensional torus consists of $N = k_1 \times k_2 \times ... \times k_r$ switches, k_i representing the number of switches in dimension i. Each switch is identified by a tuple $(a_1, ..., a_r)$, where a_i represents its position along dimension i. A direct link between switches $(a_1, ..., a_r)$ and $(b_1, ..., b_r)$ exists if there is only one j such that $a_j = (b_j \pm 1) \, mod \, r$ and $a_i = b_i$ for all $i \neq j$ [35]. Therefore, each switch has the same degree $d = 2r$ and the same betweenness centrality.

Scale-free topology. Our scale-free topology, generated as per [13], contains a total of k switches whose connectivity follows a power-law distribution, i.e. the probability that a switch has n connections equals $P(n) = n^{-\gamma}$. This results in many sparsely and only few highly connected switches. The latter have high betweenness centrality.

[1] We therefore use the terms flow and path to mean the same thing here.

Random topology. We generate a random topology with n vertices based on the $G(n, p)$ Erdős-Rényi model [37]. Each edge is included in the graph with probability p independent of other edges. The random topology serves us in these evaluations rather as a theoretical baseline. Real world networks are rarely random.

The specific parameters used to generate instances of four different topologies are shown in Table A.1. We generate three instances of the random and the scale-free topology, representing small, medium and large network with 72, 156 and 272 switches. For the other two, the torus and the hierarchical topology, we in addition vary the connectivity (2-d, 3-d torus, $n = 2$ and $n = 3$ for the hierarchical) and end up with six instances of each. We make sure that the topologies are comparable with respect to their size (number of switches and number of links).

In the torus, scale-free and random network, each switch has users attached to it. In contrast, in the hierarchical network users are attached only to access switches, while aggregation and core switches perform routing only. The number of users per switch (i.e. access switches for the hierarchical network and all switches for the other networks) is kept constant, irrespective of the topology instance.

The total number of users thus changes with network size. Each user sends communication requests to other, randomly chosen users, independently of the others and the same rate as the others. Thus, if the network has n switches that host users, each hosting k users which send requests at a rate of x requests/sec then the network controller (our TM, precisely) sees the combined request rate of $n \times k \times x$.

A.1.2 Evaluation results

We begin by showing the properties of the evaluated topologies. We first focus on the small instances, with 72 nodes. The other topology instances will be used later. Figure A.1 depicts the histogram of the shortest path lengths between all pairs of nodes, while Figure A.2 depicts the CDF of betweenness centrality of nodes. The main takeaway from these figures is that: 1) the random, 2-d torus, 3-d torus and hierarchical with $n = 3$ have pretty even distributions of betweenness centrality, whereas the scale-free and hierarchical with $n = 2$ have skewed distributions; 2) the random, torus topologies and scale-free have small shortest paths, the hierarchical topologies does not.

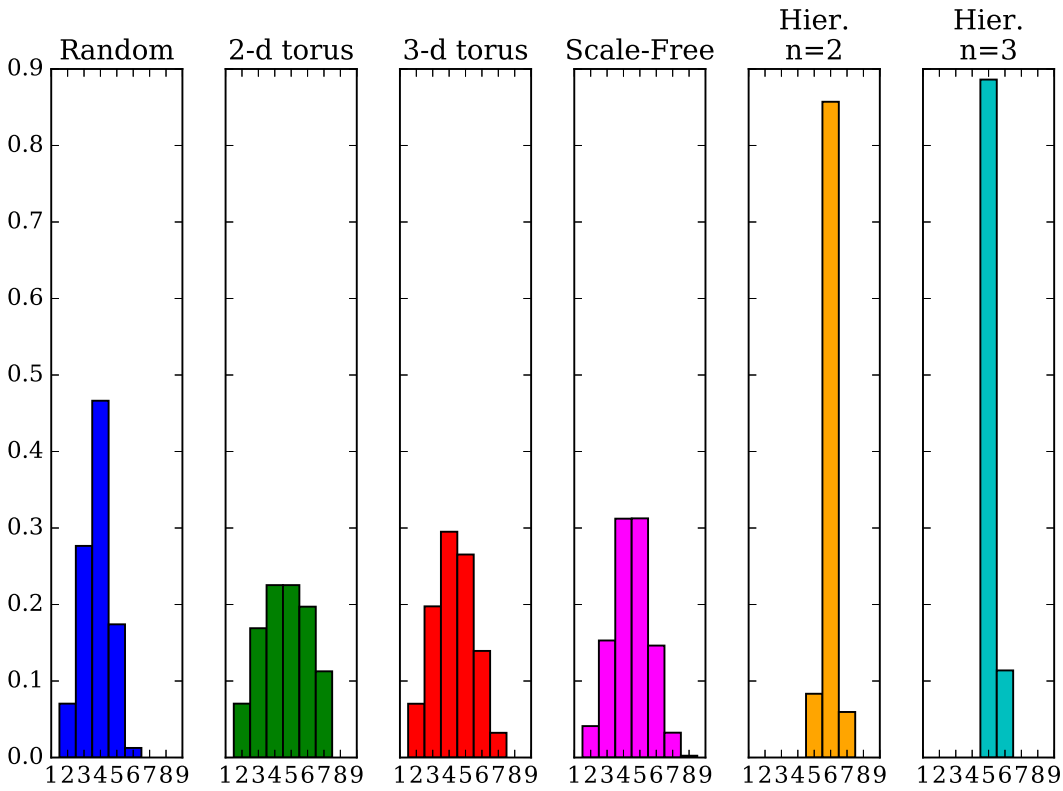

Figure A.1: PDF of the shortest path lengths for the small topology instances with 72 nodes.

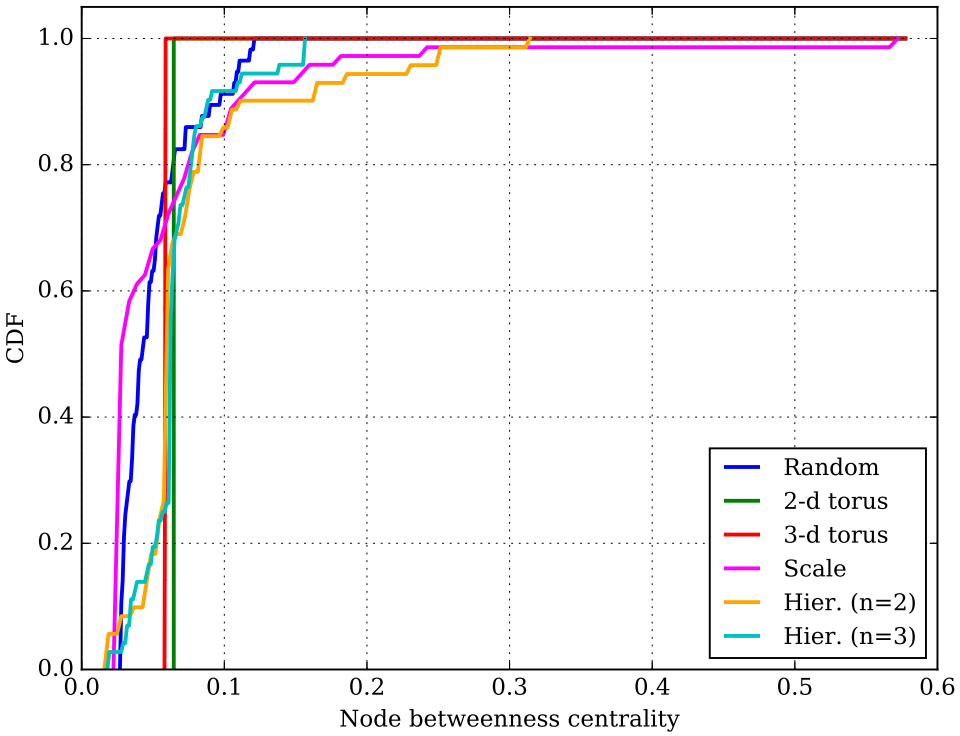

Figure A.2: CDF of betweenness centrality of nodes for the small topology instances with 72 nodes.

	Random		**Torus**	**Scale-free**	**Hierarchical**	
		$r = 2$	$r = 3$		$n = 2$	$n = 3$
SMALL NETWORK						
Switches	72	72	72	72	72	72
Parameters	$p = 0.07$	$k_1 = 8, k_2 = 9$	$k_1 = 3, k_2 = 4, k_3 = 6$	-	$k = 8$,$m = 3$	$k = 8$,$m = 3$
Links	180	144 + 36*	216	105	187	218
MEDIUM NETWORK						
Switches	156	156	168†	156	156	156
Parameters	$p = 0.0225$	$k_1 = 12, k_2 = 13$	$k_1 = 4, k_2 = 6, k_3 = 7$	-	$k = 12$,$m = 3$	$k = 12$,$m = 3$
Links	434	312 + 78*	504	233	404	489
LARGE NETWORK						
Switches	272	272	288†	272	272	272
Parameters	$p = 0.018$	$k_1 = 16, k_2 = 17$	$k_1 = 4, k_2 = 8, k_3 = 9$	-	$k = 16$,$m = 3$	$k = 16$,$m = 3$
Links	678	544 + 136*	864	425	733	861

* Links added along one dimension to obtain similar number of links as in the random topology.
† Chosen parameters result with slightly larger number of switches, but shortest path lengths remain comparable with those from other topologies.

Table A.1: Parameters used for topology generation.

The load in our simulations is a series of requests to set up flows between randomly selected endpoints in our networks. We run each experiment with 1000 flows from the users. The paths and thus the switches to lock, are determined as the shortest paths between the corresponding switches (if multiple shortest paths exist, one is randomly selected). The requests are modeled as a Poisson process with rate λ. We vary λ from 10 to $1000s^{-1}$.

Figure A.3 presents the success rate as a function of the request arrival rate, without and with backoffs. When a request is initially rejected, we use a backoff mechanism to reschedule it a bit later. Our current backoff simply delays a rejected request for a random time, drawn from the exponential distribution with mean 0.01. If a request does not succeed after 4 attempts in total, we simply drop it. Each point in the graph represents the median success rate over 50 independent experiments and its corresponding 95% confidence interval.

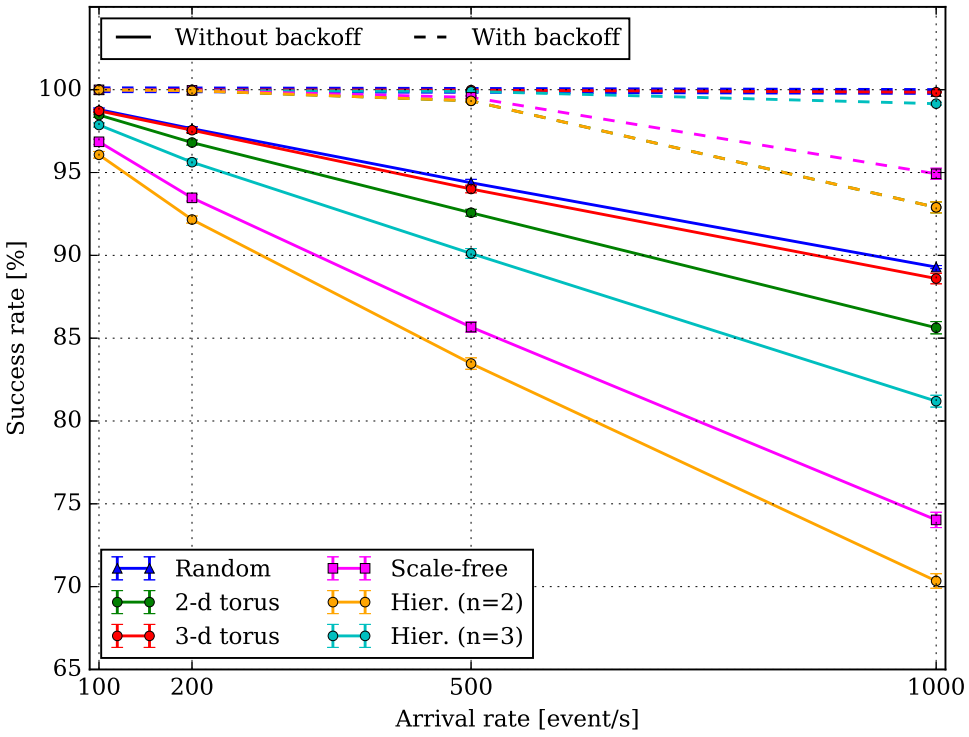

Figure A.3: Success rate (topology instances with 72 nodes).

We observe the sharpest success rate decrease for the hierarchical topology with $n = 2$. Remember that this hierarchical topology has both long shortest paths and a skewed

betweenness centrality distribution. So a single request requires locking more switches (see the lock durations from Figure 4.5), while at the same time the network cannot accommodate too many simultaneous requests. These two together lead to a collapse of the success rate for high request arrival rates.

This is due to the constrains that flows must traverse the core layer, which is why core switches have high betweenness centrality. Therefore, probability that two concurrent flows overlap in core switches is high. Besides, the shortest paths between pairs in the hierarchical topology are the longest from all four tested topologies – they have at least 5 hops. Longer paths imply longer locking duration (Figure 4.5), during which switches are unavailable for subsequent flow installation attempts, which additionally reduces the success rate.

The scale-free topology exhibits similar behavior. Recall that the scale-free topology has nice shortest path lengths but pretty bad distribution of betweenness centrality, i.e. only a few nodes with very high betweenness centrality values. So this indicates that the betweenness centrality is more important than the shortest path lengths.

This appears valid also if we compare the success rate of the hierarchical topology with $n = 3$, on the one hand, and the hierarchical with $n = 2$ and scale-free, on the other. As Figure A.1 and Figure A.2 indicate, the additional links between the aggregation and the core layer when $n = 3$ make the betweenness centrality distribution of the hierarchical topology less skewed, but the shortest paths are still long. However, the improved betweenness centrality was sufficient for the hierarchical topology with $n = 3$ to outperform both, the hierarchical with $n = 2$ and the scale-free, with 12% and 8% more successfully installed updates respectively.

Similarly, the 2-d torus has slightly longer shortest paths than the scale-free but much better distribution of betweenness centrality. The result is the 12% (6% in case of backoffs) increase of the success rate. A change of the same order of magnitude is visible when we compare the scale-free with the 3-d torus or random topology.

Figure A.3 also indicates that backoffs do increase the success rate. For $\lambda = 1000$ requests/sec backoffs bring additional 10% to 20% of satisfied requests. We observe that the success rate is essentially 100% for topologies without nodes with high betweenness centrality. On the other hand, the success rate stays below 95% for the scale-free and hierarchical topology with $n = 2$.

Figure A.4 depicts the CDF of the path setup delay, the other metrics we were interested in. The results shown hold for $\lambda = 1000$ requests/sec, backoffs were on. We observe that for the topologies without nodes with high betweenness centrality, the 2-d, 3-d torus

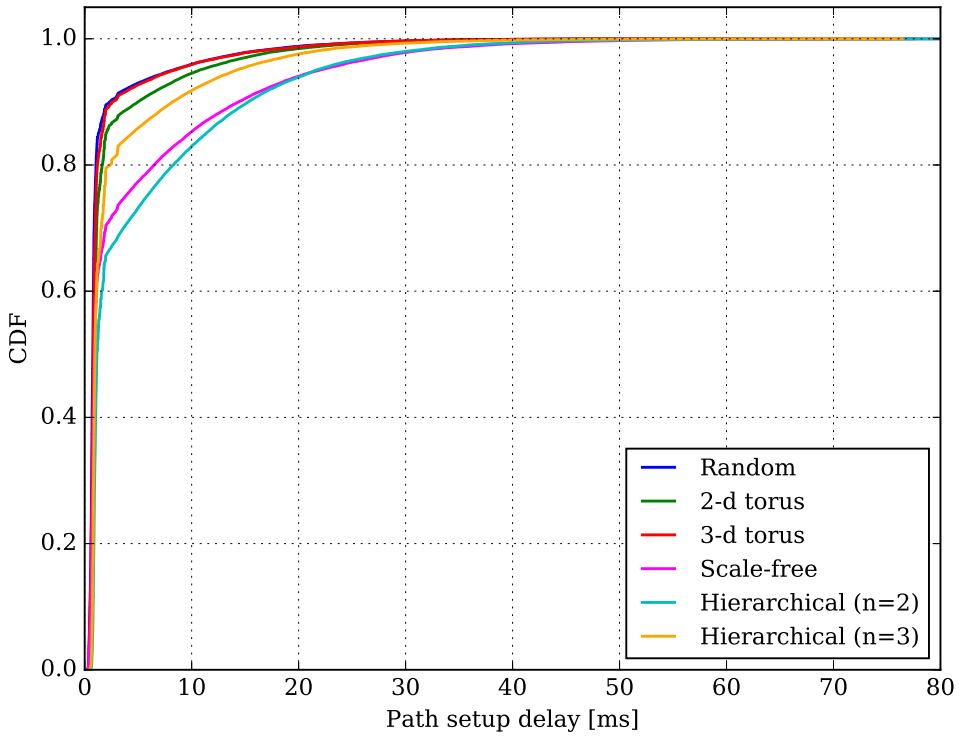

Figure A.4: CDF of the path setup delay, 3 reattempts and arrival rate of 1000 requests/sec (topology instances with 72 nodes).

and the random, the delay stays below 6ms for around 90% of the requests and that the 95th percentile of the delay is below 10ms. For the other tested topologies, with more skewed distributions of betweenness centrality, the 95th percentile is significantly higher, it reaches 25ms. For all tested topology instances, the tail of distribution originates from the updates which were installed after backoffs, through reattempts. To provide more insights into this, we present in Figure A.5 the histograms with normalized frequencies of requests installed in their first attempt and after 1, 2 or 3 backoffs. We observe that over 80% of the updates were successfully installed at the first attempt for the random and torus networks. This ratio decreases as the betweenness centrality distribution becomes more skewed. In the hierarchical network with $n = 3$, around 75% of the updates were installed at the first attempt. Finally, this ratio is significantly lower for scale-free and hierarchical topology with $n = 2$, where it reaches maximum of 62% and 57%, respectively.

In an attempt to understand the impact of the network size on the performance, we now include the larger networks, with 156 and 272 nodes (168 and 288 for 3-d torus),

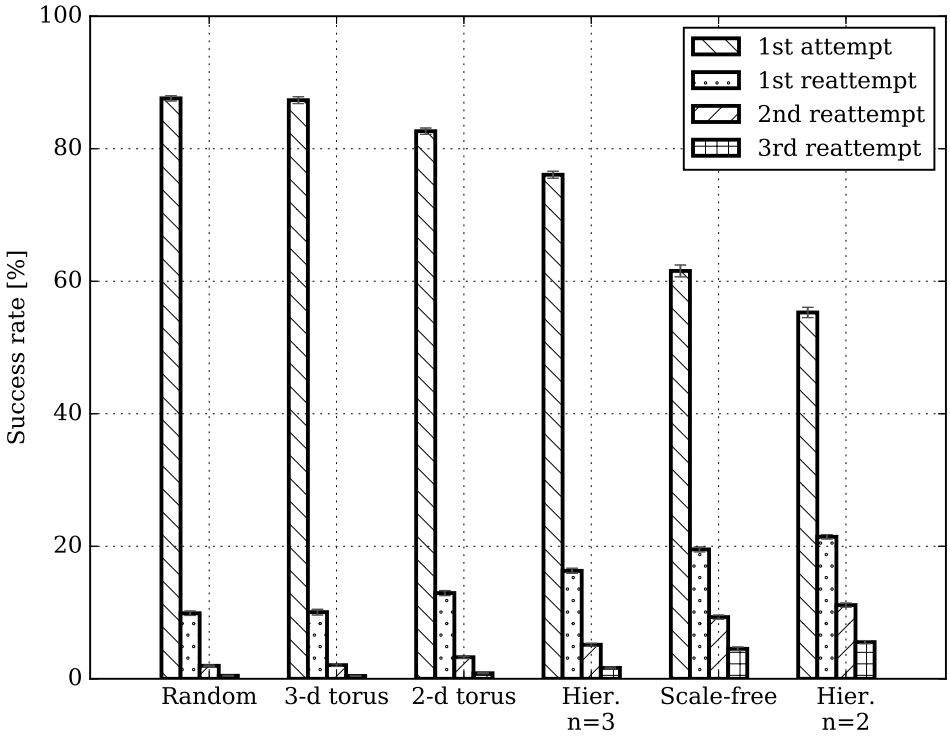

Figure A.5: Histogram of updates that were: satisfied after the first attempt, satisfied after 1, 2 and 3 reattempts, rejected. The results hold for the same setting as in Figure A.4. Topologies with 72 nodes.

into the picture. While the findings just presented for the small networks continue to (qualitatively) hold for both medium and large networks, we are now interested in their cross-comparison. The paths between switches are longer in larger networks, which is why the locking duration of the switches is longer (see Figure 4.5). Therefore, we expect deterioration of the measured KPIs. The most severe deterioration is expected for the highest update arrival rates. That is why we focus on those rates only. It is however important to recognize at this point that the bigger networks have more switches with users. To achieve fair comparison, we must therefore scale up the update arrival rate for the larger networks. It thus becomes $1000 \times 156/72 \approx 2170$ for the 156 nodes network and $1000 \times 272/72 \approx 3780$ for the 272 network. For 3-d torus these values are $1000 \times 168/72 \approx 2333$ for the 168 nodes network and $1000 \times 288/72 \approx 4000$ for the 288 network.

Figure A.6 presents simultaneously the evolution of the success rate (vertical axis on the left) and delay (vertical axes on the right) with the network size, when the update arrival rates are set to 1000, 2170 and 3780 requests/sec for the three shown network

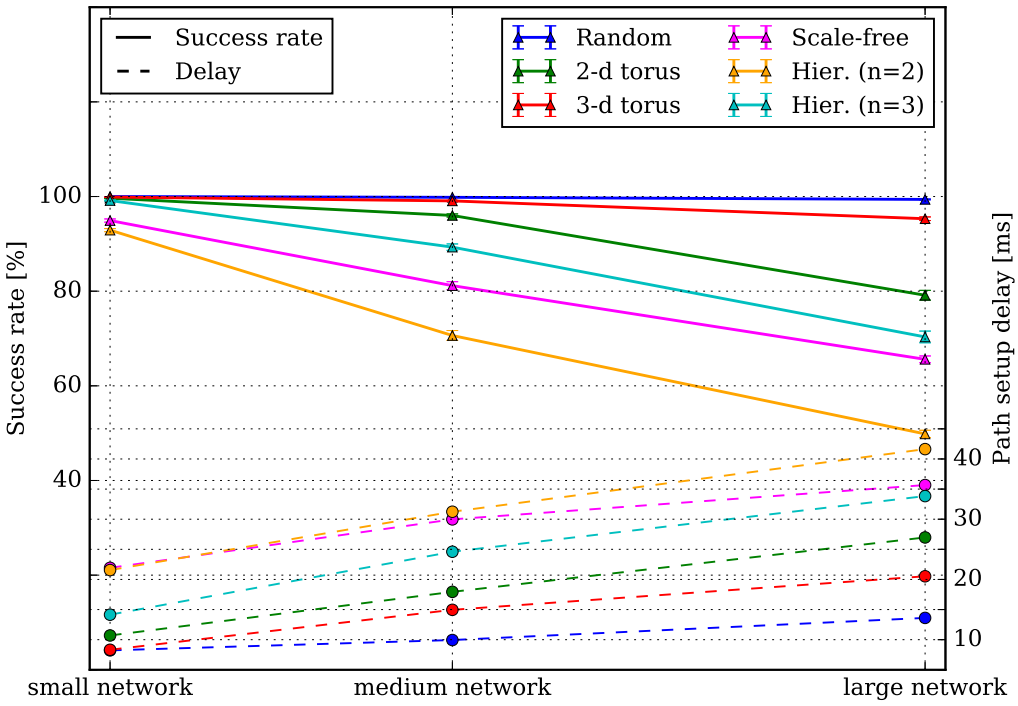

Figure A.6: The success rate (left vertical axes) and the 95th percentile of the path setup delay (right vertical axes) for the small, medium and large network. The update request arrival rates were scaled up accordingly.

sizes, respectively. As mentioned above, the last two arrival rates differ for the instances of 3-d torus network, and we set them to 2333 and 4000 requests/sec for medium and large network instances, respectively. In the upper part of Figure A.6, we see that the success rate drops sharply for each increase in the network size. In that sense, we can claim that the transactional SDN does not scale well for the hierarchical, the scale-free and the 2-d torus network. The 2-d torus performs the best among them in the large network with around 80% of the updates successfully installed. We observe the worst value of the success rate for the hierarchical network with $n = 2$, less then 50% of the updates are installed successfully. However, the random topology and the 3-d torus continue to perform well and remain almost unaffected by the network size increase. The success rate drops by 1 to 2% for each increase in the network size. The explanation of this is as follows. As the network size grows, shortest paths in the network grow as well. It is exactly this that makes the 2-d tours fail for large networks. Each individual update takes longer, now long enough to deteriorate the overall performance. But the

3-d torus rectifies this, it adds more links, which reduces the share of very long shortest paths between nodes, so we almost regain the old performance. Figure A.7 depicts the histograms of the shortest path lengths between all pairs of nodes for 2-d and 3-d large torus networks. We observe that around 30% of the shortest paths in the 2-d torus have 10 hops and more. In the 3-d torus this percentage decreases to around 5%.

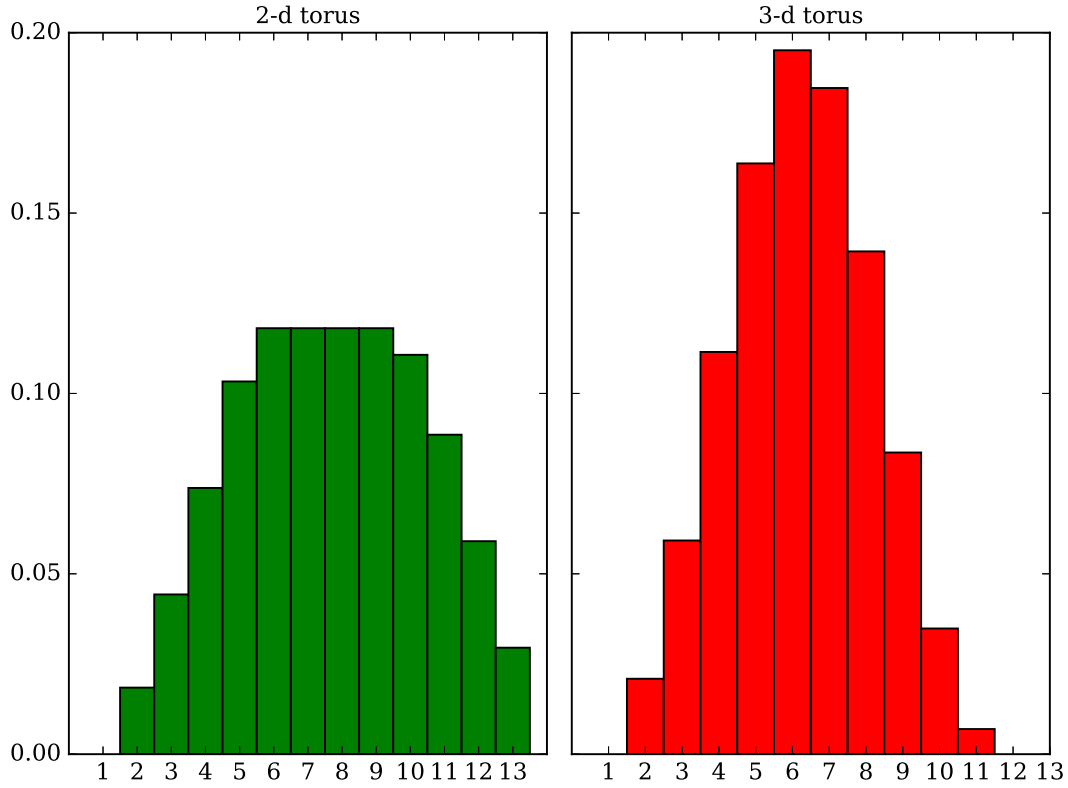

Figure A.7: The distribution of the shortest path lengths between all pairs of nodes in the 2-d and 3-d torus network with 272 and 288 nodes, respectively.

The bottom part of the Figure A.6 plots the 95th percentile of the path setup delay for different topology sizes. The 95th percentile in the scale-free, torus and hierarchical networks is around 10ms higher for each increase in the network size. The smallest increase in the 95th percentile of path setup delay is observed for the random and the 3-d torus network: around 2ms and 6ms, respectively. The increase in the path setup delay mainly relates to the updates which were installed after backoffs, through reattempts. Besides, the increase in the path setup delay is inevitable in the larger networks due to longer shortest paths.

Our simulations show that the transactional SDN can achieve very high success rate and low delays at the same time, with absolute values that are acceptable in practical

deployments. However, the performance depends on two topological characteristics – the distribution of lengths of the shortest paths between vertices as well as the distribution of the betweenness centrality of the vertices. The shorter are the paths in the network, the shorter are the periods when the switches involved in the updates are locked and hence, the better is the performance of the transactional SDN. Besides, the fewer nodes with high betweenness centrality means fewer bottlenecks where updates will conflict with each other, which results with better performance. Nevertheless, the impact of the distribution of betweenness centrality is much more dominant – the success rates in topologies where few nodes have high betweenness centrality is deteriorated even when the shortest path lengths are fairly short. With that in mind, it is necessary to be careful about the choice of the topology on which the transactional SDN is deployed. Scale-free topologies are never a good choice. A hierarchical topology can be a good choice only if redundant links are available. It would be ideal to use random topologies since they combine both topological characteristics that guarantee good performance of the transactional SDN. However, as these are hard to achieve in practice, due to physical limitations, the r-dimensional torus appears as more than a satisfactory replacement.

List of Figures

List of Tables

Bibliography

[1] Daniel Abadi. „Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story“. In: *Computer* 45.2 (2012), pp. 37–42. ISSN: 0018-9162. DOI: 10.1109/mc.2012.33.

[2] *ACIDRepo*. Web Page. 2018. URL: https://github.com/OVSacid/ACIDRepo.git.

[3] *Active Honeypot*. Web Page. URL: https://community.arubanetworks.com/t5/Active-Honeypot/ct-p/ActiveHoneypot.

[4] *ADARA Comet SDN Virtual WAN Optimizer – Trial*. Web Page. URL: https://community.arubanetworks.com/t5/ADARA-Comet-SDN-Virtual-WAN/ct-p/ADARACometSDNVirtualWANOptimizerTrial.

[5] *ADARA Hercules SDN Orchestration System - Trial*. Web Page. URL: https://community.arubanetworks.com/t5/ADARA-Hercules-SDN-Orchestration/ct-p/ADARAHerculesSDNOrchestrationSystem.

[6] Yehuda Afek, Anat Bremler-Barr, Shir Landau Feibish, and Liron Schiff. „Sampling and Large Flow Detection in SDN“. In: *ACM SIGCOMM Computer Communication Review* 45.4 (2015), pp. 345–346. ISSN: 0146-4833. DOI: 10.1145/2829988.2790009. URL: https://doi.org/10.1145/2829988.2790009.

[7] Hassan Ali-Ahmad, Claudio Cicconetti, Antonio de la Oliva, Vincenzo Mancuso, Malla Reddy Sama, Pierrick Seite, and Sivasothy Shanmugalingam. „An SDN-Based Network Architecture for Extremely Dense Wireless Networks“. In: *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*. Series An SDN-Based Network Architecture for Extremely Dense Wireless Networks. 2013, pp. 1–7. ISBN: ISBN. DOI: 10.1109/SDN4FNS.2013.6702534.

[8] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. „An analysis of network-partitioning failures in cloud systems“. In: *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018*. Year, pp. 51–68. DOI: 10.5555/3291168.3291173.

[9] Vaibhav Arora, Tanuj Mittal, Divyakant Agrawal, Amr El Abbadi, and Xun Xue. „Leader or majority: Why have one when you can have both? improving read scalability in raft-like consensus protocols“. In: (). DOI: 10.5555/3154580.3154594.

[10] Osama Arouk, Adlen Ksentini, and Tarik Taleb. „Group Paging-Based Energy Saving for Massive MTC Accesses in LTE and beyond Networks“. In: *IEEE Journal on Selected Areas in Communications* 34.5 (2016), pp. 1086–1102. DOI: 10.1109/JSAC.2016.2520222.

[11] Miloud Bagaa, Tarik Taleb, and Adlen Ksentini. „Efficient Tracking Area Management Framework for 5G Networks“. In: *IEEE Transactions on Wireless Communications* 15.6 (2016), pp. 4117–4131. ISSN: 1536-1276. DOI: 10.1109/TWC.2016.2535217.

[12] Jyotirmoy Banik, Marco Tacca, Andrea Fumagalli, Behcet Sarikaya, and Li Xue. „Enabling Distributed Mobility Management: A Unified Wireless Network Architecture Based on Virtualized Core Network“. In: *2015 24th International Conference on Computer Communication and Networks (ICCCN)*. Series Enabling Distributed Mobility Management: A Unified Wireless Network Architecture Based on Virtualized Core Network. 2015, pp. 1–6. ISBN: ISBN. DOI: 10.1109/ICCCN.2015.7288404.

[13] Albert-Laszlo Barabasi, Réka Albert, and Hawoong Jeong. „Mean-Field Theory for Scale-Free Random Networks“. In: *Physica A: Statistical Mechanics and its Applications* 272 (2000), pp. 173–187. DOI: 10.1016/S0378-4371(99)00291-5.

[14] Daniel M. Batista, Gordon Blair, Fabio Kon, Raouf Boutaba, David Hutchison, Raj Jain, Ramachandran Ramjee, and Christian Esteve Rothenberg. „Perspectives on software-defined networks: interviews with five leading scientists from the networking community“. In: *Journal of Internet Services and Applications* 6.1 (2015), p. 22. ISSN: 1867-4828 1869-0238. DOI: 10.1186/s13174-015-0035-3. URL: https://doi.org/10.1186/s13174-015-0035-3.

[15] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, and William Snow. „ONOS: towards an open, distributed SDN OS“. In: *Proceedings of the third workshop on Hot topics in software defined networking - HotSDN '14*. Series ONOS: towards an open, distributed SDN OS. 2014, pp. 1–6. ISBN: ISBN. DOI: 10.1145/2620728.2620744.

[16] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. „OpenState: programming platform-independent stateful openflow applications inside the switch“. In: *ACM SIGCOMM Computer Communication Review* 44.2 (2014),

44–51. ISSN: 0146-4833. DOI: 10.1145/2602204.2602211. URL: https://doi.org/10.1145/2602204.2602211.

[17] *BlueCat DNS Director*. Web Page. URL: https://community.arubanetworks.com/t5/BlueCat-DNS-Director-Beta/ct-p/BlueCatDNSDirectorBeta.

[18] Wolfgang Braun and Michael Menth. „Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices“. In: *Future Internet* 6.2 (2014), pp. 302–336. ISSN: 1999-5903. DOI: 10.3390/fi6020302.

[19] Kai Bu, Xitao Wen, Bo Yang, Yan Chen, Erran Li Li, and Xiaolin Chen. „Is every flow on the right track?: Inspect SDN forwarding with RuleScope“. In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pp. 1–9. DOI: 10.1109/INFOCOM.2016.7524333.

[20] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. „Software transactional networking: Concurrent and consistent policy composition“. In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*. Series Software transactional networking: Concurrent and consistent policy composition. 2013, pp. 1–6. ISBN: ISBN. DOI: 10.1145/2491185.2491200.

[21] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. „Ethane: Taking control of the enterprise“. In: *ACM SIGCOMM computer communication review*. Vol. 37. Series Ethane: Taking control of the enterprise. ACM New York, NY, USA, 2007, pp. 1–12. ISBN: ISBN. DOI: 10.1145/1282380.1282382.

[22] Balakrishnan Chandrasekaran, Brendan Tschaen, and Theophilus Benson. *Isolating and Tolerating SDN Application Failures with LegoSDN*. Conference Paper. 2016. DOI: 10.1145/2890955.2890965. URL: https://doi.org/10.1145/2890955.2890965.

[23] Yangyang Chen and Xavier Lagrange. „Analysis and improvement of mobility procedures for mobile relays in LTE networks“. In: *2015 IEEE 26th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pp. 1769–1774. DOI: 10.1109/PIMRC.2015.7343585.

[24] Inc. Cisco System. *OpFlex: An Open Policy Protocol*. Report. 2014. URL: https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-731302.pdf.

[25] *Configuring Networks and Devices with Simple Network Management Protocol (SNMP)*. Web Page. 2003. URL: https://tools.ietf.org/html/rfc3512.

[26] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, and Peter Hochschild. „Spanner: Google's globally distributed database“.

In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), pp. 1–22. ISSN: 0734-2071. DOI: 10.1145/2491245.

[27] Lenore J. Cowen. „Compact Routing with Minimum Stretch“. In: *Journal of Algorithms* 38.1 (2001), pp. 170–183. ISSN: 01966774. DOI: 10.1006/jagm.2000.1134.

[28] Maja Curic, Georg Carle, Zoran Despotovic, Ramin Khalili, and Artur Hecker. „SDN on ACIDs“. In: *2nd Cloud-Assisted Networking Workshop, CAN 2017*. Year, pp. 19–24. DOI: 10.1145/3155921.3155924.

[29] Maja Curic, Zoran Despotovic, Artur Hecker, and Georg Carle. „FitSDN: Flexible Integrated Transactional SDN“. In: *44th Annual IEEE Local Computer Networks Symposium on Emerging Topics in Networking, LCN Symposium 2019*. Year, pp. 1–9. DOI: 10.1109/LCNSymposium47956.2019.9000677.

[30] Maja Curic, Zoran Despotovic, Artur Hecker, and Georg Carle. „Transactional Network Updates in SDN“. In: *2018 European Conference on Networks and Communications, EuCNC 2018*. Year, pp. 203–208. DOI: 10.1109/EuCNC.2018.8442793.

[31] *Defense Flow*. Web Page. URL: https://community.arubanetworks.com/t5/Defense-Flow/ct-p/DefenseFlow.

[32] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. „Epidemic algorithms for replicated database maintenance“. In: *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing - PODC '87*. Series Epidemic algorithms for replicated database maintenance. 1987, pp. 1–12. ISBN: ISBN. DOI: 10.1145/41840.41841.

[33] Zoran Despotovic, Xun Xiao, Ramin Khalili, Maja Curic, and Artur Hecker. „Dynamic and Scalable Control as a Foundation for Future Networks“. In: *Emerging Automation Techniques for the Future Internet*. Advances in Wireless Technologies and Telecommunication. IGI Global, 2019. Chap. chapter 8, pp. 208–230. ISBN: 9781522571469 9781522571476. DOI: 10.4018/978-1-5225-7146-9.ch008.

[34] Christian Deyerl and Tobias Distler. „In Search of a Scalable Raft-based Replication Architecture“. In: *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data - PaPoC '19*. Series In Search of a Scalable Raft-based Replication Architecture. 2019, pp. 1–7. ISBN: ISBN. DOI: 10.1145/3301419.3323968.

[35] Jose Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection networks*. Morgan Kaufmann, 2003. ISBN: 1558608524.

[36] Anwar Elwalid, Cheng Jin, Steven Low, and Indra Widjaja. „MATE: MPLS adaptive traffic engineering“. In: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*. Vol. 3. Series MATE: MPLS adaptive traffic engineering. IEEE, 2001, pp. 1300–1309. ISBN: ISBN. DOI: 10.1109/INFCOM.2001.916625.

[37] Paul Erdős and Alfréd Rényi. „On the evolution of random graphs“. In: *Publ. Math. Inst. Hung. Acad. Sci* 5.1 (1960), pp. 17–60.

[38] *Experiment F - Flow Subsystem Burst Throughput*. Web Page. 2018. URL: https://wiki.onosproject.org/display/ONOS/1.14%3A+Experiment+F+-+Flow+Subsystem+Burst+Throughput.

[39] Jelte Fennema. *Bug in the raft paper and proposed solution*. Web Page. 2017. URL: https://groups.google.com/forum/#!topic/raft-dev/JEtBYaPpHXo.

[40] *FitSDNRepo*. Web Page. 2019. URL: https://github.com/fitSDN/RepoMain.git.

[41] *Floodlight OpenFlow Controller – Project Floodlight*. Web Page. URL: https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview.

[42] Gianluca Foddis, Rosario G. Garroppo, Stefano Giordano, Gregorio Procissi, Simone Roma, and Simone Topazzi. „LTE traffic analysis for signalling load and energy consumption trade-off in mobile networks“. In: *IEEE International Conference on Communications, ICC 2015*. Vol. 2015-September, pp. 6005–6010. DOI: 10.1109/ICC.2015.7249279.

[43] Klaus-Tycho Foerster, Juho Hirvonen, Stefan Schmid, and Jukka Suomela. „On the Power of Preprocessing in Decentralized Network Optimization“. In: *2019 IEEE Conference on Computer Communications, INFOCOM 2019*. Vol. 2019-April. Year, pp. 1450–1458. DOI: 10.1109/INFOCOM.2019.8737382.

[44] Klaus-Tycho Foerster and Stefan Schmid. „Distributed Consistent Network Updates in SDNs: Local Verification for Global Guarantees“. In: *2019 IEEE 18th International Symposium on Network Computing and Applications (NCA)*. Series Distributed Consistent Network Updates in SDNs: Local Verification for Global Guarantees. IEEE, 2019, pp. 1–4. ISBN: ISBN. DOI: 10.1109/nca.2019.8935035.

[45] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. „Frenetic: A network programming language“. In: *ACM Sigplan Notices* 46.9 (2011), pp. 279–291. ISSN: 0362-1340.

[46] Qi-An Fu and Wenfei Wu. „TUS: A Transactional Update Service for SDN Applications“. In: *Proceedings of the 9th Asia-Pacific Workshop on Systems*. Association for Computing Machinery, 2018, pp. 1–8. DOI: 10.1145/3265723.3265728. URL: https://doi.org/10.1145/3265723.3265728.

[47] Yonghong Fu, Jun Bi, Kai Gao, Ze Chen, Jianping Wu, and Bin Hao. „Orion: A Hybrid Hierarchical Control Plane of Software-Defined Networking for Large-Scale Networks“. In: *2014 IEEE 22nd International Conference on Network Protocols*. Series Orion: A Hybrid Hierarchical Control Plane of Software-Defined Networking for Large-Scale Networks. IEEE, 2014, pp. 569–576. ISBN: ISBN. DOI: 10.1109/icnp.2014.91.

[48] Hector Garcia-Molina. *Database systems: the complete book*. Pearson Education India, 2008. ISBN: 813170842X.

[49] Matteo Gerola, Francesco Lucrezia, Michele Santuari, Elio Salvadori, Pier Luigi Ventre, Stefano Salsano, and Mauro Campanella. „ICONA: A Peer-to-Peer Approach for Software Defined Wide Area Networks Using ONOS“. In: *2016 Fifth European Workshop on Software-Defined Networks (EWSDN)*. Series ICONA: A Peer-to-Peer Approach for Software Defined Wide Area Networks Using ONOS. IEEE, 2016, pp. 37–42. ISBN: ISBN. DOI: 10.1109/ewsdn.2016.12.

[50] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. „Understanding network failures in data centers: measurement, analysis, and implications“. In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 350–361. ISSN: 01464833. DOI: 10.1145/2043164.2018477.

[51] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. „Evolve or die: High-availability design principles drawn from googles network infrastructure“. In: *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference - SIGCOMM '16*. Series Evolve or die: High-availability design principles drawn from googles network infrastructure. 2016, pp. 58–72. ISBN: ISBN. DOI: 10.1145/2934872.2934891.

[52] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. „A clean slate 4D approach to network control and management“. In: *ACM SIGCOMM Computer Communication Review* 35.5 (2005), 41–54. ISSN: 0146-4833. DOI: 10.1145/1096536.1096541. URL: https://doi.org/10.1145/1096536.1096541.

[53] Roch A. Guerin and Ariel Orda. „QoS routing in networks with inaccurate information: theory and algorithms“. In: *IEEE/ACM transactions on Networking* 7.3 (1999), pp. 350–364. ISSN: 1063-6692. DOI: 10.1109/90.779203.

[54] Isabelle Guis. *The SDN Gold Rush To The Northbound API*. Web Page. 2012. URL: https://www.sdxcentral.com/articles/contributed/the-sdn-gold-rush-to-the-northbound-api/2012/11/.

[55] Jordan Halterman. *Transaction Management in ONOS*. Web Page. 2018. URL: https://groups.google.com/a/onosproject.org/forum/#!msg/onos-dev/cg5NyJsyxUA/SrsUvraIBgAJ.

[56] 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*. Series Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. 2012, pp. 19–24. ISBN: ISBN. DOI: 10.1145/2342441.2342446.

[57] Ekram Hossain, Mehdi Rasti, Hina Tabassum, and Amr Abdelnasser. „Evolution toward 5G multi-tier cellular wireless networks: An interference management perspective“. In: *IEEE Wireless Communications* 21.3 (2014), pp. 118–127. ISSN: 1536-1284. DOI: 10.1109/mwc.2014.6845056.

[58] Heidi Howard and Jon Crowcroft. „Coracle: evaluating consensus at the internet edge“. In: *ACM SIGCOMM Computer Communication Review* 45.5 (2015), pp. 85–86. ISSN: 01464833. DOI: 10.1145/2829988.2790010.

[59] Michael Howard. „Using carrier Ethernet to backhaul LTE“. In: *Infonetics Research White Paper* (2011).

[60] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. „ZooKeeper: Wait-free Coordination for Internet-scale Systems“. In: vol. 8. Series ZooKeeper: Wait-free Coordination for Internet-scale Systems. 2010. ISBN: ISBN. DOI: 10.5555/1855840.1855851.

[61] IETF. *NETCONF Configuration Protocol*. Web Page. 2011. URL: https://tools.ietf.org/html/rfc6241.

[62] Madan Jampani. *Update in a multipartitioned raft implementation (2-phase commit)*. Web Page. 2016. URL: https://groups.google.com/a/onosproject.org/forum/#!msg/onos-dev/bkgXaYUBoCE/Jfm2a6NYAgAJ.

[63] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. „Dynamic scheduling of network updates“. In: *ACM SIGCOMM Computer Communication Review* 44.4 (2014), pp. 539–550. ISSN: 0146-4833.

[64] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. „Walking the tightrope: Responsive yet stable traffic engineering“. In: *ACM SIGCOMM Computer Communication Review* 35.4 (2005), pp. 253–264. ISSN: 0146-4833.

[65] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. *CacheFlow: Dependency-Aware Rule-Caching for Software-Defined Networks*. Conference Paper. 2016. DOI: 10.1145/2890955.2890969. URL: https://doi.org/10.1145/2890955.2890969.

[66] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. „Ravana: Controller fault-tolerance in software-defined networking“. In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research -*

SOSR '15. Series Ravana: Controller fault-tolerance in software-defined networking. 2015, pp. 1–12. ISBN: ISBN. DOI: 10.1145/2774993.2774996.

[67] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. „Real time network policy checking using header space analysis“. In: *10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013*. Year, pp. 99–111. DOI: 10.5555/2482626.2482638.

[68] Wolfgang Kellerer, Arsany Basta, Peter Babarczi, Andreas Blenk, Mu He, Markus Klugel, and Alberto Martinez Alba. „How to Measure Network Flexibility? A Proposal for Evaluating Softwarized Networks“. In: *IEEE Communications Magazine* 56.10 (2018), pp. 186–192. ISSN: 0163-6804 1558-1896. DOI: 10.1109/mcom.2018.1700601.

[69] Ramin Khalili, Zoran Despotovic, and Artur Hecker. „Flow setup latency in SDN networks“. In: *IEEE Journal on Selected Areas in Communications* 36.12 (2018), pp. 2631–2639. ISSN: 0733-8716.

[70] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. „Veriflow: Verifying network-wide invariants in real time“. In: *Proceedings of the first workshop on Hot topics in software defined networks - HotSDN '12*. Series Veriflow: Verifying network-wide invariants in real time. 2012, pp. 15–27. ISBN: ISBN. DOI: 10.1145/2342441.2342452.

[71] Hyojoon Kim and Nick Feamster. „Improving network management with software defined networking“. In: *IEEE Communications Magazine* 51.2 (2013), pp. 114–119. ISSN: 0163-6804.

[72] 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: *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010*. Year, pp. 351–364. DOI: 10.5555/1924943.1924968.

[73] Ayaka Koshibe and Bob Lantz. *Why are flows stuck in PENDING_ADD state?* Web Page. 2017. URL: https://wiki.onosproject.org/display/ONOS/FAQ#FAQ-WhyareflowsstuckinPENDING_ADDstate?.

[74] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. „Software-defined networking: A comprehensive survey“. In: *Proceedings of the IEEE* 103.1 (2014), pp. 14–76. ISSN: 0018-9219.

[75] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. „What You Need to Know About SDN Flow Tables“. In: *Passive and Active Measurement*. Ed. by Jelena

Mirkovic and Yong Liu. Springer International Publishing, pp. 347–359. ISBN: 978-3-319-15509-8.

[76] Maciej Kuzniar, Peter Peresini, Marco Canini, Daniele Venzano, and Dejan Kostic. *A SOFT way for openflow switch interoperability testing*. Conference Paper. 2012. DOI: 10.1145/2413176.2413207. URL: https://doi.org/10.1145/2413176.2413207.

[77] Maciej Kuzniar, Peter Peresini, and Dejan Kostić. „Providing Reliable FIB Update Acknowledgments in SDN“. In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies - CoNEXT '14*. Series Providing Reliable FIB Update Acknowledgments in SDN. 2014, pp. 415–422. ISBN: ISBN. DOI: 10.1145/2674005.2675006.

[78] Avinash Lakshman and Prashant Malik. „Cassandra: a decentralized structured storage system“. In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40. ISSN: 0163-5980.

[79] Leslie Lamport. „Paxos made simple“. In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[80] Bob Lantz, Brandon Heller, and Nick McKeown. „A network in a laptop: rapid prototyping for software-defined networks“. In: *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks - Hotnets '10*. Series A network in a laptop: rapid prototyping for software-defined networks. 2010, pp. 1–6. ISBN: ISBN. DOI: 10.1145/1868447.1868466.

[81] Zohaib Latif, Kashif Sharif, Fan Li, Md Monjurul Karim, Sujit Biswas, and Yu Wang. „A comprehensive survey of interface protocols for software defined networks“. In: *Journal of Network and Computer Applications* (2020), p. 102563. ISSN: 1084-8045.

[82] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. „Logically centralized? State distribution trade-offs in software defined networks“. In: *Proceedings of the first workshop on Hot topics in software defined networks - HotSDN '12*. Series Logically centralized? State distribution trade-offs in software defined networks. 2012, pp. 1–6. ISBN: ISBN. DOI: 10.1145/2342441.2342443.

[83] Geng Li, Yichen Qian, Chenxingyu Zhao, Y. Richard Yang, and Tong Yang. „DDP: Distributed Network Updates in SDN“. In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. Series DDP: Distributed Network Updates in SDN. IEEE, 2018, pp. 1468–1473. ISBN: ISBN. DOI: 10.1109/icdcs.2018.00150.

[84] *Link Layer Discovery Protocol and MIB*. Web Page. 2002. URL: https://www.ieee802.org/1/files/public/docs2002/lldp-protocol-00.pdf.

[85] Priya Mahadevan, Dmitri Krioukov, Marina Fomenkov, Bradley Huffaker, Xenofontas Dimitropoulos, and Amin Vahdat. „Lessons from three views of the Internet topology“. In: *arXiv preprint cs/0508033* (2005).

[86] Clarissa Cassales Marquezan, Zoran Despotovic, Ramin Khalili, David Perez-Caparros, and Artur Hecker. „Understanding processing latency of SDN based mobility management in mobile core networks“. In: *2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*. Series Understanding processing latency of SDN based mobility management in mobile core networks. 2016, pp. 1–7. ISBN: ISBN. DOI: 10.1109/PIMRC.2016.7794937.

[87] James McCauley, Aurojit Panda, Martin Casado, Teemu Koponen, and Scott Shenker. „Extending SDN to large-scale networks“. In: *Open Networking Summit* (2013), pp. 1–2.

[88] Jedidiah McClurg, Hossein Hojjat, and Pavol Černý. „Synchronization Synthesis for Network Programs“. In: *Computer Aided Verification*. Lecture Notes in Computer Science. 2017. Chap. Chapter 16, pp. 301–321. ISBN: 978-3-319-63389-3 978-3-319-63390-9. DOI: 10.1007/978-3-319-63390-9_16.

[89] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černý. „Event-driven network programming“. In: *ACM SIGPLAN Notices* 51.6 (2016), pp. 369–385. ISSN: 0362-1340.

[90] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. „OpenFlow: enabling innovation in campus networks“. In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746.

[91] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. „Network function virtualization: State-of-the-art and research challenges“. In: *IEEE Communications surveys & tutorials* 18.1 (2015), pp. 236–262. ISSN: 1553-877X.

[92] Michael Mitzenmacher. „The power of two choices in randomized load balancing“. In: *IEEE Transactions on Parallel and Distributed Systems* 12.10 (2001), pp. 1094–1104. ISSN: 10459219. DOI: 10.1109/71.963420.

[93] Tal Mizrahi and Yoram Moses. „Software defined networks: It's about time“. In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. Series Software defined networks: It's about time. IEEE, 2016, pp. 1–9. ISBN: ISBN. DOI: 10.1109/infocom.2016.7524418.

[94] Matthew Monaco, Oliver Michel, and Eric Keller. „Applying operating system principles to SDN controller design“. In: *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks - HotNets-XII*. Series Applying operating system

principles to SDN controller design. 2013, pp. 1–7. ISBN: ISBN. DOI: 10.1145/2535771.2535789.

[95] Ron Nadiv and Tzvika Naveh. „Wireless backhaul topologies: Analyzing backhaul topology strategies“. In: *Ceragon White Paper* (2010), pp. 1–15.

[96] Mark E. J. Newman. „A measure of betweenness centrality based on random walks“. In: *Social networks* 27.1 (2005), pp. 39–54. ISSN: 0378-8733.

[97] Thanh Dang Nguyen, Marco Chiesa, and Marco Canini. „Decentralized Consistent Updates in SDN“. In: *Proceedings of the Symposium on SDN Research*. Series Decentralized Consistent Updates in SDN. 2017, pp. 21–33. ISBN: ISBN. DOI: 10.1145/3050220.3050224.

[98] *Novel radio link concepts and state of the art analysis*. Report. 2013. URL: https://cordis.europa.eu/docs/projects/cnect/9/317669/080/deliverables/001-METISD22v1pdf.pdf.

[99] David Nowoswiat. *Managing LTE Core Network Signaling Traffic*. Web Page. 2013. URL: https://www.nokia.com/blog/managing-lte-core-network-signaling-traffic/.

[100] Diego Ongaro. *Bug in single-server membership changes*. Web Page. 2015. URL: https://groups.google.com/forum/#!msg/raft-dev/t4xj6dJTP6E/d2D9LrWRza8J.

[101] Diego Ongaro and John Ousterhout. „In search of an understandable consensus algorithm“. In: *2014 USENIX Annual Technical Conference, USENIX ATC 2014*. Year, pp. 305–319. DOI: 10.5555/2643634.2643666.

[102] Melih Onus, Andrea Richa, and Christian Scheideler. „Linearization: Locally self-stabilizing sorting in graphs“. In: *9th Workshop on Algorithm Engineering and Experiments and the 4th Workshop on Analytic Algorithms and Combinatorics*. Year, pp. 99–108. DOI: 10.5555/2791188.2791198.

[103] Aurojit Panda, Colin Scott, Ali Ghodsi, Teemu Koponen, and Scott Shenker. „CAP for networks“. In: *HotSDN 2013 - Proceedings of the 2013 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. Year, pp. 91–96. DOI: 10.1145/2491185.2491186.

[104] Aurojit Panda, Wenting Zheng, Xiaohe Hu, Arvind Krishnamurthy, and Scott Shenker. „SCL: Simplifying distributed SDN control planes“. In: *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017*. Year, pp. 329–345. DOI: 10.5555/3154630.3154657.

[105] Peter Pereini, Maciej Kuzniar, and Dejan Kostić. „OpenFlow Needs You! A Call for a Discussion about a Cleaner OpenFlow API“. In: *2013 Second European Workshop on Software Defined Networks*. Series OpenFlow Needs You! A Call

for a Discussion about a Cleaner OpenFlow API. IEEE, 2013, pp. 44–49. ISBN: ISBN. DOI: 10.1109/ewsdn.2013.14.

[106] Peter Peresini, Maciej Kuzniar, and Dejan Kostic. „Dynamic, Fine-Grained Data Plane Monitoring With Monocle“. In: *IEEE/ACM Transactions on Networking* 26.1 (2018), pp. 534–547. ISSN: 1063-6692 1558-2566. DOI: 10.1109/tnet.2018.2793765. URL: https://doi.org/10.1109/TNET.2018.2793765.

[107] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. „The design and implementation of open vSwitch“. In: *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015*. Year, pp. 117–130. DOI: 10.5555/2789770.2789779.

[108] Prashant Pillai, Kandeepan Sithamparanathan, Giovanni Giambene, Miguel Ángel Vázquez, and Paul Daniel Mitchell. *Wireless and Satellite Systems*. Springer, 2018. ISBN: 331976571X.

[109] Phillip Porras, Steven Cheung, Martin Fong, Keith Skinner, and Vinod Yegneswaran. „Securing the Software Defined Network Control Layer“. In: *Proceedings 2015 Network and Distributed System Security Symposium*. Series Securing the Software Defined Network Control Layer. 2015. ISBN: ISBN. DOI: 10.14722/ndss.2015.23222.

[110] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. *SIMPLE-fying middlebox policy enforcement using SDN*. Conference Paper. 2013. DOI: 10.1145/2486001.2486022. URL: https://doi.org/10.1145/2486001.2486022.

[111] Rapeepat Ratasuk, Nitin Mangalvedhe, Amitava Ghosh, and Benny Vejlgaard. „Narrowband LTE-M System for M2M Communication“. In: *2014 IEEE 80th Vehicular Technology Conference (VTC2014-Fall)*, pp. 1–5. ISBN: 1090-3038. DOI: 10.1109/VTCFall.2014.6966070.

[112] Rapeepat Ratasuk, Athul Prasad, Zexian Li, Amitava Ghosh, and Mikko A. Uusitalo. „Recent advancements in M2M communications in 4G networks and evolution towards 5G“. In: *2015 18th International Conference on Intelligence in Next Generation Networks*, pp. 52–57. DOI: 10.1109/ICIN.2015.7073806.

[113] Yoav Raz. „The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment“. In: 92 (), pp. 292–312. DOI: 10.5555/645918.672337.

[114] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. „Abstractions for network update“. In: *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication - SIGCOMM '12*. Association for Computing

Machinery, 323–334. DOI: 10.1145/2342356.2342427. URL: https://doi.org/10.1145/2342356.2342427.

[115] Jennifer Rexford, Albert Greenberg, Gisli Hjalmtysson, David A Maltz, Andy Myers, Geoffrey Xie, Jibin Zhan, and Hui Zhang. „Network-wide decision making: Toward a wafer-thin control plane“. In: *Proc. HotNets*, pp. 59–64.

[116] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. „Canopus: A scalable and massively parallel consensus protocol“. In: *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. Series Canopus: A scalable and massively parallel consensus protocol. 2017, pp. 426–438. ISBN: ISBN. DOI: 10.1145/3143361.3143394.

[117] Peter Rost, Albert Banchs, Ignacio Berberana, Markus Breitbach, Mark Doll, Heinz Droste, Christian Mannweiler, Miguel Puente, Konstantinos Samdanis, and Bessem Sayadi. „Mobile network architecture evolution toward 5G“. In: *Infocommunications Journal* 9.1 (2017), pp. 24–31. DOI: 10.1109/MCOM.2016.7470940.

[118] Ermin Sakic, Fragkiskos Sardis, Jochen W. Guck, and Wolfgang Kellerer. „Towards adaptive state consistency in distributed SDN control plane“. In: *2017 IEEE International Conference on Communications (ICC)*. Series Towards adaptive state consistency in distributed SDN control plane. IEEE, 2017, pp. 1–7. ISBN: ISBN. DOI: 10.1109/icc.2017.7997164.

[119] Malla Reddy Sama, Siwar Ben Hadj Said, Karine Guillouard, and Lucian Suciu. „Enabling network programmability in LTE/EPC architecture using OpenFlow“. In: *2014 12th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt)*. Series Enabling network programmability in LTE/EPC architecture using OpenFlow. 2014, pp. 389–396. ISBN: ISBN. DOI: 10.1109/wiopt.2014.6850324.

[120] Liron Schiff, Stefan Schmid, and Petr Kuznetsov. „In-band synchronization for distributed SDN control planes“. In: *ACM SIGCOMM Computer Communication Review* 46.1 (2016), pp. 37–43. ISSN: 0146-4833.

[121] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, H. B. Acharya, Kyriakos Zarifis, and Scott Shenker. „Troubleshooting blackbox SDN control software with minimal causal sequences“. In: *ACM SIGCOMM Computer Communication Review* 44.4 (2015), pp. 395–406. ISSN: 0146-4833. DOI: 10.1145/2740070.2626304.

[122] Sakir Sezer, Sandra Scott-Hayward, Pushpinder Kaur Chouhan, Barbara Fraser, David Lake, Jim Finnegan, Niel Viljoen, Marc Miller, and Navneet Rao. „Are we ready for SDN? Implementation challenges for software-defined networks“. In: *IEEE Communications Magazine* 51.7 (2013), pp. 36–43. ISSN: 0163-6804.

[123] Anees Shaikh, Jennifer Rexford, and Kang G. Shin. „Evaluating the impact of stale link state on quality-of-service routing“. In: *IEEE/ACM Transactions On Networking* 9.2 (2001), pp. 162–176. ISSN: 1063-6692.

[124] Alexander Shalimov, Dmitry Zuikov, Daria Zimarina, Vasily Pashkov, and Ruslan Smeliansky. „Advanced study of SDN/OpenFlow controllers“. In: *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia on - CEE-SECR '13*. Series Advanced study of SDN/OpenFlow controllers. 2013, pp. 1–6. ISBN: ISBN. DOI: 10.1145/2556610.2556621.

[125] Michael Stonebraker. *Why Enterprises Are Uninterested in NoSQL*. Web Page. 2010. URL: https://cacm.acm.org/blogs/blog-cacm/99512-why-enterprises-are-uninterested-in-nosql/fulltext.

[126] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M. Frans Kaashoek, and Robert Tappan Morris. „Flexible, wide-area storage for distributed systems with wheelfs“. In: *6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009*. Year, pp. 43–58. DOI: 10.5555/1558977.1558981.

[127] Maja Sulovic, Clarissa Cassales Marquezan, and Artur Hecker. „Towards Location Management in SDN-based MCN“. In: *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. Series Towards Location Management in SDN-based MCN. IEEE, 2017, pp. 1097–1102. ISBN: ISBN. DOI: 10.23919/inm.2017.7987443.

[128] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007. ISBN: 0132392275.

[129] *TechM Smart Flow Steering*. Web Page. URL: https://community.arubanetworks.com/t5/TechM-Smart-Flow-Steering/ct-p/TechMSmartFlowSteering.

[130] David Tennenhouse, Jonathan Smith, W. Sincoskie, David Wetherall, and Gary Minden. „A survey of active network research“. In: *IEEE Communications Magazine* 35.1 (1997), pp. 80–86. ISSN: 0163-6804. DOI: 10.1109/35.568214.

[131] David L. Tennenhouse and David J. Wetherall. „Towards an active network architecture“. In: *ACM SIGCOMM Computer Communication Review* 37.5 (2007), pp. 81–94. ISSN: 0146-4833. DOI: 10.1145/1290168.1290180.

[132] *The NOX Controller*. Web Page. 2014. URL: https://github.com/noxrepo/nox.

[133] Amin Tootoonchian and Yashar Ganjali. „HyperFlow: A Distributed Control Plane for OpenFlow“. In: *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*. Vol. 3. INM/WREN'10. San Jose, CA: USENIX Association, 2010, p. 3. DOI: 10.5555/1863133.1863136.

[134] Daniel Turner, Kirill Levchenko, Alex C. Snoeren, and Stefan Savage. „California fault lines: understanding the causes and impact of network failures“. In: *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM - SIGCOMM '10*. Series California fault lines: understanding the causes and impact of network failures. 2010, pp. 315–326. ISBN: ISBN. DOI: 10.1145/1851182.1851220.

[135] *User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)*. Web Page. 2002. URL: https://tools.ietf.org/html/rfc3414.

[136] *VMware NSX*. Web Page. URL: https://avinetworks.com/vmware-nsx/.

[137] Wen Wang, Wenbo He, Jinshu Su, and Yixin Chen. „Cupid: Congestion-free consistent data plane update in software defined networks“. In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. Series Cupid: Congestion-free consistent data plane update in software defined networks. IEEE, 2016, pp. 1–9. ISBN: ISBN. DOI: 10.1109/infocom.2016.7524420.

[138] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001. ISBN: 0080519563.

[139] Yanwei Xu and Tiantian Ren. „Analysis of the New Features of OpenFlow 1.4“. In: *Proceedings of the 2nd International Conference on Information, Electronics and Computer*. ISBN: 978-90-78677-99-4. DOI: 10.2991/icieac-14.2014.17.

[140] Volkan Yazıcı, Ulas Kozat, and M. Sunay. „A new control plane for 5G network architecture with a case study on unified handoff, mobility, and routing management“. In: *IEEE Communications Magazine* 52.11 (2014), pp. 76–85. ISSN: 0163-6804. DOI: 10.1109/mcom.2014.6957146.

[141] Soheil Hassas Yeganeh and Yashar Ganjali. „Beehive: Simple distributed programming in software-defined networks“. In: *Proceedings of the Symposium on SDN Research - SOSR '16*. Series Beehive: Simple distributed programming in software-defined networks. 2016, pp. 1–12. ISBN: ISBN. DOI: 10.1145/2890955.2890958.

[142] Soheil Hassas Yeganeh, Amin Tootoonchian, and Yashar Ganjali. „On scalability of software-defined networking“. In: *IEEE Communications Magazine*. Vol. 51. Ieee, Year, pp. 136–141. ISBN: 0163-6804. DOI: 10.1109/MCOM.2013.6461198.

[143] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. *A Survey on Network Troubleshooting*. Report TR12-HPNG-061012. Stanford University, 2012.

[144] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. „Automatic Test Packet Generation“. In: *IEEE/ACM Transactions on Networking*

22.2 (2014), pp. 554–566. ISSN: 1063-6692 1558-2566. DOI: 10.1109/tnet.2013.2253121.

[145] Peng Zhang, Cheng Zhang, and Chengchen Hu. „Fast Data Plane Testing for Software-Defined Networks With RuleChecker“. In: *IEEE/ACM Transactions on Networking* 27.1 (2019), pp. 173–186. ISSN: 1063-6692 1558-2566. DOI: 10.1109/tnet.2018.2885532.

[146] Wei Zhao, Haihua Shen, Feng Zhang, and Huazhe Tan. „Adaptive Power Optimization for Mobile Traffic Based on Machine Learning“. In: *2019 IEEE 23rd International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pp. 500–505. DOI: 10.1109/CSCWD.2019.8791501.

[147] Yuqing Zhu, Philip S. Yu, and Jianmin Wang. „RECODS: Replica consistency-on-demand store“. In: *29th International Conference on Data Engineering, ICDE 2013*. Year, pp. 1360–1363. DOI: 10.1109/ICDE.2013.6544944.

Appendix

Published Articles in Original Format

Due to copyright restrictions, we emphasize that all publications in this Section are accepted versions and not published versions. The publications are reprinted with the permissions of the publisher.

Towards Location Management in SDN-based MCN

Maja Sulovic, Clarissa Cassales Marquezan, Artur Hecker
European Research Center, Huawei Technologies, Munich, Germany
Email: {maja.sulovic, clarissa.marquezan, artur.hecker}@huawei.com

***Abstract*—One of the key functionalities in EPC (Evolved packet Core) is Mobility Management (MM), which provides procedures for service continuity as the mobile devices (UE) move across the network. In a quest for more flexibility of MM, recent studies suggested to map MM procedures to SDN applications in an SDN controller. So far, these proposals investigated and showed how path switching during handover, one of the procedures of MM, is achieved in such an SDN-based Mobile Core Network (MCN). However, Location Management (LM) procedures, such as paging, are equally important. In this paper, we address this gap. We notably design and implement SDN-based UE state management and paging procedure for MCN. Our design only uses the data from the SDN controller, such as UE connectivity and flow information, and defines a new set of UE states: deregistered, idle, and active. We introduce new, dynamically configurable UE inactivity timers that regulate the transitions among these states. With this, we implement a purely SDN-based paging procedure capable of bringing a UE into a connected state on request, e.g. to deliver the downlink traffic. The experiments with our implementation on Floodlight SDN controller under different network loads and configuration settings for UE inactivity timers in Mininet demonstrate the practical feasibility of an SDN-based LM. Further, we show how different UE inactivity timer values can flexibly regulate the amount of paging. All in all, our results suggest that, leveraging the programmability provided by SDN, operators can flexibly install and use LM SDN apps with different settings to tailor the amount of paging according to the specific UE or user application activity patterns and its needs.**

I. INTRODUCTION

In the ongoing design of the 5G mobile networks, flexibility is believed to be the key feature to support the expected variety of use cases and massive amounts of mobile devices. Achieving such flexibility while keeping the complexity low is a challenge for the next generation Mobile Core Networks (MCN) [1], [2]. The state of the art suggests Software Defined Networking (SDN) and Network Function Virtualization (NFV) as key enablers for 5G MCN [3], [4]. Current approaches can be divided in two groups. The first encloses proposals that maintain almost all EPC control signaling and entities [5], [6], [7]. They use virtualized EPC entities as VNFs (Virtual Network Functions), and the SDN controller to flexibly manage the interconnections of the latter. The other group includes solutions changing the signaling and the entities of the control plane. In this case, the services of the EPC are maintained, but the implementation uses SDN Applications as basic service elements [8], [9], [10]. While both approaches can achieve the flexibility, the latter has an important advantage of leveraging the SDN infrastructure for a broader set of functions, hence removing the need for an additional NFV infrastructure or for non-trivial coordination between the SDN and the NFV subsystems. Our view on the evolved MCN in this paper is aligned with the solutions in the second group.

The native MCN functionality is mobility management (MM), including the prominent handover management for UEs with active sessions, i.e. in "connected" state. In the current LTE EPC (Evolved Packet Core), the Mobility Management Entity (MME) and the basestations (eNB) are responsible for handling UE mobility. To spare battery resources, UEs regularly become idle. The transition from connected to idle state in LTE depends on a static UE inactivity timer preconfigured in each eNB in the network [11]. With this, whenever an eNB detects inactivity of a UE, it signals this event to the MCN. On this event, the MME orders a partial release of data plane resources for that UE and changes its state from connected to idle.

Hence, another set of crucial MM procedures is location management (LM), employing tracking area (TA) updates and paging to track the attachment point of idle UEs. Here, the TA update allows the MCN to track the approximate attachment point of idle UEs. Now, to deliver downlink data to idle UEs, paging must be performed within the latest tracking area to switch the UE state to connected and to retrieve its precise attachment point. LTE defines one state transition mechanism to fit all UE and application types in the network, which may not be optimal as discussed in [12]. Yet, paging represents an important load: while LM in the overall LTE signaling traffic accounts for up to one third of LTE signaling [13], TA updates account for 4.9% and paging represents more than 28% of the overall MME load. Still, previous research did not investigate, how such procedure can be provided in an SDN-based MCN.

To address this gap, we design and implement an SDN-based state management and paging procedure for MCN. Our idea is to use the data available at the SDN controller to design a new set of UE states. Then, we introduce new, dynamically configurable UE inactivity timers that regulate the transitions among such states. Overall, our approach achieves the flexibility by enabling the configuration of these timer values per UEs and applications; what is more, it supports dynamic changes of the configured values in runtime, e.g. in response to network events and conditions.

We implemented our ideas on the Floodlight SDN controller and used Mininet to run tests under different network loads. When all UEs have the same inactivity timer value, our experiments show that: i) the higher the UE activity, which we control with the inter-arrival time of session requests, the lower is the number of page messages that need to be triggered; ii) a variation of 1s in a scenario of high UE activity (e.g., changing from 1s to 2s the inter-arrival time for session requests) has a large effect in the increase of paging to be triggered (e.g., it shows an increase of 40% of paging triggered), while when this variation occurs in low UE activity, the increase in the amount of paging to be triggered increases by only 5%. As our goal was not to improve the paging procedure per se, these results,

which prove a comprehensible system response, underline the practical feasibility of our approach. What is more, they show that different UE inactivity timer values effectively regulate the amount of paging. Combined with the flexibility of the SDN paradigm, where instances of various SDN apps could be used for different groups of UEs, distinct user application classes, etc., our main contribution is a system that allows an operator to precisely tailor the overall paging in the system to the needs and policies. We believe that such fine tuning is key for 5G, where much more devices of different types are expected.

This paper is organized as follows. Next session discusses related work. Section III presents the technical backgrounds. Section IV introduces the design of our SDN-based location management solution. Section V presents emulation results of the solution. Finally, in Section VI, we conclude.

II. RELATED WORK

Several approaches addressed optimization of location management signaling in wireless networks. Bagaa et al. [14] proposed the implementation of a framework for efficient tracking area list management (ETAM). ETAM tunes the trade-off between tracking area update (TAU) and paging signaling messages. It finds the optimal distribution of Tracking Areas (TAs) in the form of Tracking Area Lists (TALs) and assigns TALs to the UEs, according to their mobility patterns or geographical distribution. The authors have shown that optimization of TALs in the network achieves better balance between the amount of triggered location updates and paging procedures. As it is dedicated to TALs, we regard this work as complementary to ours. Arouk et al. [15] acknowledged the need to provide different mechanisms of paging in 5G networks. They argue that the increase in data traffic volumes in 5G could cause congestion in both AN and MCN, which would further introduce intolerable delays, packet loss, or even service unavailability. They proposed an optimization of current 3GPP Group Paging (GP), which reduces the amount of signaling overhead in RAN. Similar ideas can be combined with our approach. Most of previous proposals for MCN on SDN are focused on handover management and path switching solutions. For instance, Marquezan et al, [8] proposed an SDN-based MCN for path switching using reactive and proactive design choices for handling the MCN signaling when a handover occurs. Ali-Ahmad et al. [16] also addressed handover management and proposed introduction of novel mechanisms that must be both distributed, in order to avoid bottlenecks, and offered dynamically, to reduce the signaling load and improve the overall performance. They propose a distributed MM to manage two types of handover events: when a UE moves within one local area, the handover event can be managed solely by the local controller, and when a UE attaches to the access point in another local area, the handover event must be managed by two local and one regional controllers. While similar SDN designs could also host our LM SDN apps, the authors did not considers LM in their work. Sama et al. [12] is one of the few proposals to consider LM in SDN. They define a new set of messages for current LTE procedures such as attachment and session setup. They also consider the flow expiry mechanism of OpenFlow switches in order to determine, whether a UE is in idle of connected states. They provide analytical means to calculate the signaling, and one of their formulae indicates the signaling load for session setup, for the precise case when the RAN bearer expires but not the MCN bearer. Therefore, this work is dedicated to the RAN idle modes and does not specify how to page a UE when the MCN bearers expire as well. While this approach fits the LTE "always on" principle, 5G is expected to support other types of services and devices.

III. BACKGROUND

In this session, we first discuss how UE state management and paging procedure work in LTE. Then, we introduce the basic concepts of the Mobility Management SDN Application (MMA) [8], which we used as a basis for the contribution in this paper.

A. Location Management in EPC

In LTE, the MME must know the exact attachment point of the UE in order to provide mobile service. The UE is not required to constantly stay connected. At the same time, the MME will not page the UE in the entire network on downlink data delivery. Instead, 3GPP partitions the network in so-called paging or tracking areas (TA). The UE can effectively move within the TA without sending anything, while the MME only has to flood the known TA when paging a UE [11].

The procedures defined within EPS Mobility Management (EMM) protocol and EPS Connectivity Management (ECM) protocol define different states for a UE upheld by the MME, as illustrated in Figure 2. The EMM states result from the procedures like Attach and Tracking Area Update. UE can be in one of the two EMM states, EMM-DEREGISTERED or EMM-REGISTERED. EMM-DEREGISTERED implies that the UE is not attached to the network; MME does not have routing or location information. After a successful attachment, the state of the UE from MMEs perspective changes to EMM-REGISTERED and the MME knows the location of the UE at least to the accuracy level of the TA (dependinig on the ECM status). The ECM states describe the signaling connectivity between the UE and the EPC. UE can be in one of two ECM states, ECM-IDLE and ECM-CONNECTED. When UE attaches to the network, MME determines which Serving and PDN Gateways (SGW and PGW) will serve the UE and creates an always-on GTP tunnel between them, dedicated to that UE (S5 bearer). MME also establishes the tunnel between the eNodeB and SGW (S1 bearer) and the default radio barrier between the UE and the eNB. With the established S1, S5 and radio bearers, the UE is in ECM-CONNECTED state, and MME knows its exact location to the accuracy of the serving eNB. After a certain period of UE's inactivity (detected by the expiration of the RRC inactivity timer at the eNB, usually in range of few seconds to few tens of seconds), the eNB sends a "UE Context Release" message to the MME. The MME then requests the SGW and the eNB to release the S1 bearer and radio resources dedicated to that UE, respectively. The MME switches the state of the UE to ECM-IDLE and considers its location to be known to the accuracy level of TA. The tunnel between the SGW and PGW remains established, when the UE's state is switched to ECM-IDLE (hence the name "always on").

If a downlink data needs to be delivered to a UE in ECM-IDLE state, the SGW buffers the data from PGW and

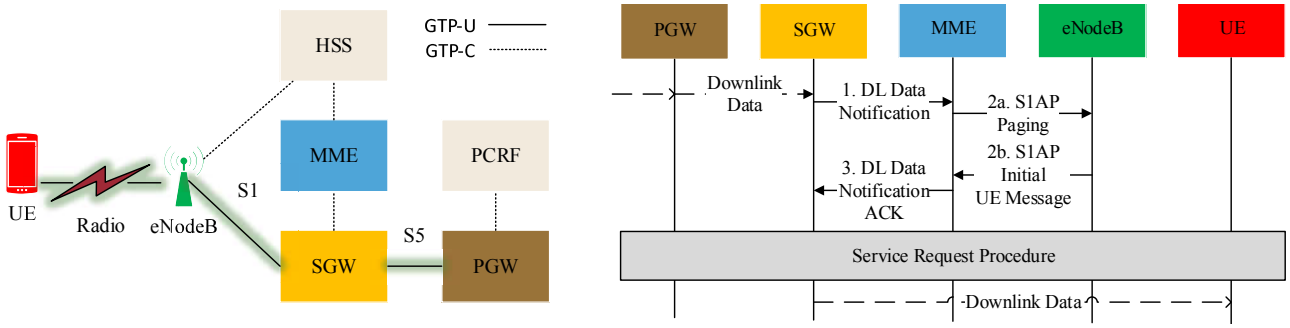

Fig. 1: Downlink data session establishment in LTE

sends a "DL Data Notification message" to the MME (step 1). The MME starts paging the UE, by asking all eNBs of the TA to broadcast a paging message for that UE (step 2a). When the UE receives this paging request, it sends paging response message (step 2b and 3) and starts Service Request Procedure to establish radio and S1 bearers. After receiving paging response, the MME will switch the state of the UE to ECM-CONNECTED. The buffered downlink data packets will now be delivered from SGW to the UE. The call flow is shown in Figure 1.

B. SDN Based Mobility Management Application

In [8], the authors proposed a proactive (MMP) mobility management design. Their solution is solely based on SDN to handle changes at the MCN level. This assumes that eNBs support the OpenFlow protocol. When a UE needs to be handed over to another eNB, the eNB will send a message to the controller (using a PACKET_IN) to signal the controller and MMP that a handover is happening. The MMP pre-installs flowrules, which match UEs data traffic in the neighboring eNBs, proactively, i.e. before the handover event occurs. However, the MMP does not consider location management and how specific UE states can influence the downlink data delivery. We adopted the MMP as the baseline architecture for the UE LM, state management and paging procedure proposed in this paper.

IV. Proposal of Location Management in 5G MCN

Our design leverages three mechanisms. First, we rely on the Flow Information Base (FIB) from the previous work [8]. Originally, this FIB on the SDN controller stored information about all UE flows, deployed flow matches and the current UE attachment point. We extended it to store flow expiration timers that control session activity. Second, we only use the native capabilities of spec-conform OpenFlow switches. OpenFlow defines the IDLE_TIMEOUT field in a flow entry as the period of time in seconds without a single packet matching that flow entry; after that time the flow entry expires and must be removed from the flow table. The IDLE_TIMEOUT field of each flow entry is set to the value of previously mentioned flow expiration timer from FIB. A flow entry also has a flag field (i.e., OFPFF_SEND_FLOW_REM), which indicates to the OF switches to send flow removal notifications towards the SDN controller, using the message OFP_FLOW_EXPIRED. Third, we used the SDN controller data about network devices. We extended this data to store information about the UE tracking area and UE state switching timers, which are essential for the paging procedure, as discussed in Section IV-A. We combine the timers that control the UE states with flow (session) expiry timers and flow entry removal notifications to indicate, when a UE needs to change its state.

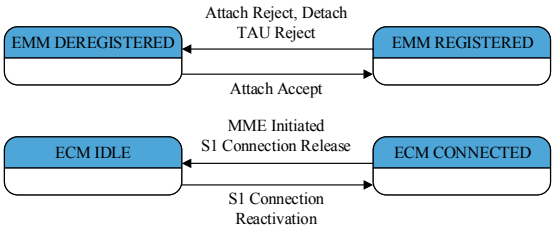

Fig. 2: UE state transition diagram in LTE

Our architecture is depicted in Figure 3. We defined two new SDN applications: UE State App and SDN Paging App. The former manages the state transitions for each UE, while the latter performs the paging procedure when necessary. The other three modules, FIB, Mobile Service State Information Manager (MSSIM), and MMP are from [8], extended in this work. The MSSIM interprets a message from the OF Switch (i.e., a PACKET_IN) against the data in the FIB to decide, which event is associated with the message. We extended this module to identify the need to trigger paging, when the destination UE is in idle state. The MMP was extended to set up active flows using the flow expiry information from the FIB. One of the key features of our solution is the possibility to program both the timers that regulate flow expiration and the timers that regulate the UE state switching. Hence, our solution goes beyond the current design in LTE, in which one model fits all types of devices and service needs.

In our design, we assume a correctly dimensioned SDN layer. For approaches how to build this, see the comprehensive SDN survey [17]. Hence, in our design, as per [18], we make sure that the proposed SDN apps do not introduce global locks and states. Indeed, in our design, any added state is strictly per UE, therefore allowing to add new application instances for (groups of) UEs as necessary, which solves scalability.

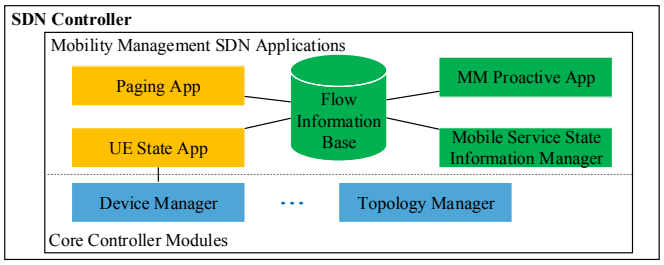

Fig. 3: Proposed Architecture

A. UE State Management

The proposed UE state management enables three states as illustrated in Figure 4: ACTIVE, IDLE or DEREGISTERED. UE with at least one active flow installed in the network is in state ACTIVE. Its access point is known to the accuracy of access switch and the port number, hence state ACTIVE corresponds to the state ECM-CONNECTED from LTE. The UE State App registers to receive flow entry expiration notifications sent by the OF switches. Every time a flow expiry notification arrives, the UE State App will mark the expired flow in FIB as inactive and check the remaining active flows of the source and destination UE. If there are no more active flows originating or terminating in any of these UEs (or in one of them), the UE State App starts the timer T_IDLE for that UE. If new PACKET_IN message arrives, indicating new originating or terminating session on the UE, whose timer T_IDLE is counting, the UE State App will stop the timer, reset its value and forward the PACKET_IN to the MMP to establish flow path. Otherwise, if the timer T_IDLE expires, the UE transitions from ACTIVE to IDLE. The attachment point of a UE in IDLE state is considered to be known to the accuracy of the 5G TA (TA in LTE), which corresponds to the state ECM-IDLE in LTE. If a new PACKET_IN requires establishment of a new flow path towards a UE in IDLE state, the MSSIM will interact with the Paging App to trigger paging and bring UE to ACTIVE state. Finally, a UE transits from IDLE state to DEREGISTERED, if its inactivity period exceeds the predefined period T_DEREGISTER ($\gg$T_IDLE), also controlled by the UE State App. Such UE is considered to be detached from the network and will not be paged. State DEREGISTERED corresponds to the state EMM_DEREGISTERED in LTE. A UE has to register to the network to change its state from DEREIGSTERED to ACTIVE.

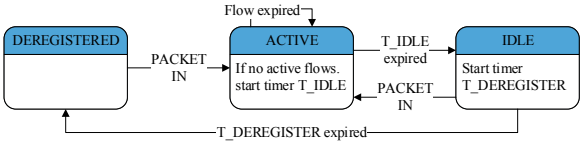

Fig. 4: UE state transition diagram in SDN

B. Paging procedure in SDN-based MCN

The overall steps for performing paging are illustrated in Figure 5. The trigger for paging (step 1) is PACKET_IN analyzed by the MSSIM application. Then, the Paging App starts the paging procedure by sending PACKET_OUT messages (step 2) to all access switches that belong to the 5G TA, where the UE in IDLE state is registered. Each PACKET_OUT message encapsulates the paging message that targets the UE in IDLE state. To achieve this, we set the destination port in all PACKET_OUT messages to ALL. Each switch in 5G TA de-capsulates the paging message and floods all ports with it, except the ingress port. The paging message reaches all UEs registered in 5G TA (step 3.a), and only the one that is actually being paged sends paging response (step 3.b). The access switch will, upon reception of the paging response message, issue PACKET_IN towards the controller (step 4). The controller internally passes the message to MSSIM that recognizes this PACKET_IN as paging response and forwards the message to UE State App, which updates UE's current attachment point and last seen timestamp. At this point, the UE State App changes the UE state back to ACTIVE and resets its T_IDLE timer. Once active, the SDN Apps can establish flow paths towards such UE.

C. Implementation

We implemented our solution, addressing SDN-based state management and paging procedure, using the open source Floodlight SDN controller (version 1.0). We emulated our solution on a Mininet platform extended to support mobility. This platform has limitations related to the emulation of the radio signaling. For this reason, we could not implement the paging at the radio side. Instead, we created paging request and response messages that could be transmitted from the APs (which in our platform are OF Switches). The paging request message is a UDP packet targeting a certain closed port at the UE in IDLE state. When the UE in IDLE state receives this UDP packet, it sends back a response message "ICMP UDP Port Unreachable". This message is the paging response in our solution. When AP (OF Switch) receives paging response, it sends a PACKET_IN to the SDN controller. When this PACKET_IN arrives at the controller, the Device Manager will update its internal database with UE's attachment point. The PACKET_IN message is further passed to the UE State App to update the UE state to ACTIVE and reset the T_IDLE timer.

V. EVALUATION

In this section, our goal is to demonstrate the practical feasibility of the SDN-based LM approach. Note that we are not aiming to evaluate the LM procedures per se (i.e., paging). As our design is SDN-based, it is relatively easy to implement other paging logics, such as discussed in Section II.

Instead, we pursue two goals: first, we want to know, whether our SDN-based LM works as expected. Second, and more importantly, we want to show that the parameters of our proposal effectively regulate the amount of paging in the system. To do this, we study the impact of two parameters on the system paging: the frequency of PACKET_IN messages and the T_IDLE timer value. The evaluated metric is the percentage of downlink deliveries that triggered paging. The

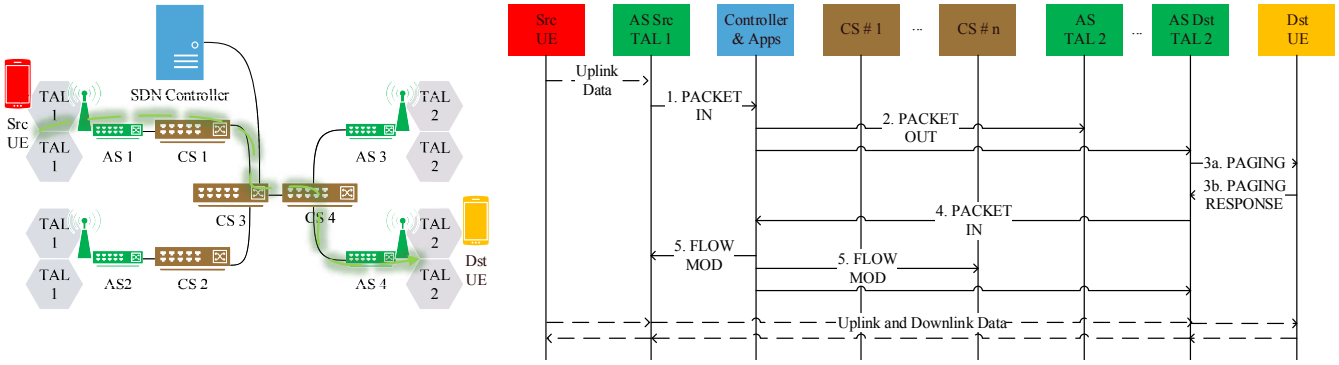

Fig. 5: Downlink data session establishment in SDN

testbed for the evaluation is a two-layered (access and core) Mininet topology. Parameters used in the emulation are shown in Table I.

TABLE I: Simulation parameters

Number of access switches	10
Number of core switches	50
Number of hosts per access switch	5
Number of sessions	250
Gamma distribution shape	2
Session duration	10s
Session expiry timer IDLE_TIMEOUT	5s
T_IDLE	5s

In our emulation, each host in the network starts communication with other randomly chosen host, whereby the time between two subsequent communication requests from one host is random and follows exponential distribution. Each new session request results with PACKET_IN on the SDN controller, so their arrival follows the same distribution. Since all hosts in the network communicate simultaneously and independently, the inter-arrival time of all PACKET_IN messages on the SDN controller (triggered by all subscribers) will be a sum of independent exponentially distributed random variables and, as such, it will follow a two-parameter gamma distribution (PDF and mean value as in Eq. 1 and Eq. 2, respectively). By varying the parameters of the gamma distribution, shape k and scale θ, it is possible to simulate more or less frequent arrival of PACKET_IN messages on the SDN controller. These inter-arrival times corresponds to the different overall UE activity in the network, where less and more frequent arrival implies lower and higher overall activity of UEs, respectively.

$$f(x;k,\theta) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-\frac{x}{\theta}} \; x \geq 0 \, and \, k, \theta > 0 \qquad (1)$$

$$E[x] = k\theta \qquad (2)$$

The goal of the first test is to determine, how the overall UE activity in the network impacts the amount of paging requests. In scenarios with increased UE activity (higher frequency of PACKET_IN arrivals), the probability that a randomly chosen host at a randomly chosen time has at least one active flow (i.e., it is in the ACTIVE state) will also be increased. Hence, the probability that such chosen UE will be paged before setting up the flow path for a received PACKET_IN is accordingly lower. Figure 6 shows the results of the experiments for this scenario. We varied the load of the network and observed the percentage of paging triggered. The x-axis represents the average PACKET_IN inter-arrival time in the network and y-axis represents the percentage of PACKET_INs that triggered paging before the establishment of the flow path between the two communicating UEs. The first observation is that the higher the UE activity in the network, i.e., the lower the inter-arrival time, the lower is the amount of triggered paging. This is aligned with our expectations. For instance, when average inter-arrival time of PACKET_IN messages is set to 1s (shape 0.5 and scale 2), the percentage of paging is around 32%. If the mean inter-arrival time is reduced and set to 2s, the percentage of PACKET_INs that triggered paging almost double. The increase in the mount of paging remains, as we increase the inter-arrival time (which in its turn, reduces the UE activity in the network); nevertheless, this increase in the percentage has a lower trend of growth. The results show non-linear dependency on the overall UE activity, where the small difference, when

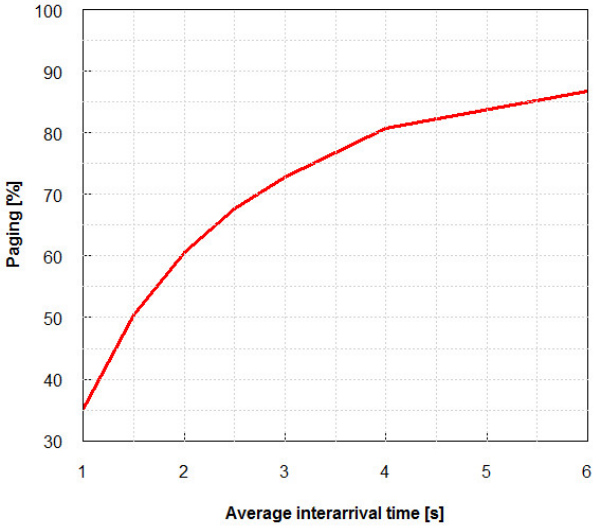

Fig. 6: Percentage of paging depending from the network load

there is high UE activity can significantly increase the amount of triggered paging requests, but the same variations with low UE activity does not generate the same increase in triggered paging requests. We can conclude that the proposed solution can implement UE state aware paging procedure as in LTE today. However, the observed non-linear behavior confirms that an approach, where the timers can be programmed is able to optimize the amount of signaling produced by the paging procedure.

The second test investigates the impact of timer T_IDLE value on the amount of paging requests. The tests have been carried out using three scenarios with different UE activity the inter-arrival time between two subsequent PACKET_IN messages in the network was modeled using gamma(0.5, 2), gamma(0.75, 2) and gamma(1, 2), which corresponds to average inter-arrival time equal to 1s, 1.5s and 2s, respectively. Values used for the timer T_IDLE are 2, 5, 10, 20 and 30 seconds. The results are depicted in Figure 7. As expected, the increase of the inactivity timer reduces the amount of triggered paging requests. Moreover, there exists approximately linear relationship between T_IDLE timeout value and the percentage of PACKET_INs that triggered paging. When the T_IDLE value is increased of 5 seconds, we observe an average reduction of 4% in the amount of triggered paging requests in all tested scenarios. This experiment shows that dynamic programmability of T_IDLE in the proposed approach unfolds the possibility to tailor paging curve according to the needs of each type of device, application, or any other MCN operator requirement.

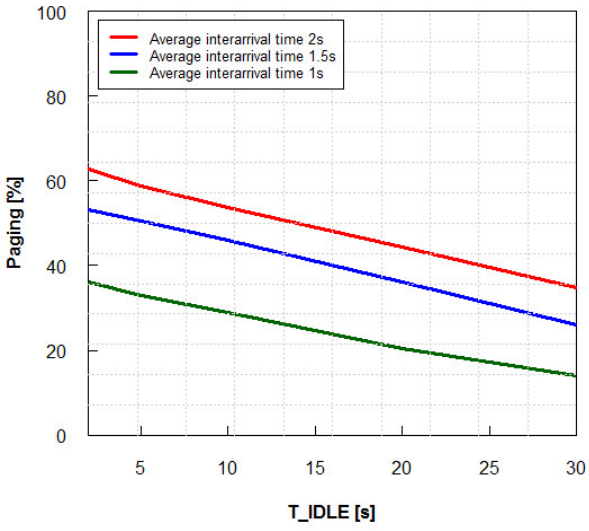

Fig. 7: Paging percentage depending on the value of T_IDLE

VI. CONCLUSION AND FUTURE WORK

This paper shows that it is possible to design and implement an SDN-based UE state management and paging procedures for MCN. Furthermore, it also shows that the programmability offered by SDN can improve the performance of paging procedures, reducing signaling load by enabling the capability of tailoring the need to trigger paging to UE types, applications or other MCN operator requirements. As a future work, we plan to investigate solutions for dynamically defining UE inactivity timer values based on different network performance metrics.

REFERENCES

[1] S. E. Elayoubi et al., "5G service requirements and operational use cases: Analysis and METIS II vision," IEE EuCNC, Athens, pp. 158-162., June 2016.

[2] P. Rost et al., "Mobile network architecture evolution toward 5G," in IEEE Communications Magazine, vol. 54, no. 5, pp. 84-91, May 2016.

[3] M. R. Sama, L. M. Contreras, J. Kaippallimalil, I. Akiyoshi, H. Qian and H. Ni, "Software-defined control of the virtualized mobile packet core," in IEEE Communications Magazine, vol. 53, no. 2, pp. 107-115, Feb. 2015.

[4] J. Mueller, Y. Chen, B. Reichel, V. Vlad and T. Magedanz, "Design and implementation of a Carrier Grade Software Defined Telecommunication Switch and Controller," IEEE Network Operations and Management Symposium (NOMS), Krakow, pp. 1-7, May 2014.

[5] P. Ameigeiras, J. J. Ramos-munoz, L. Schumacher, J. Prados-Garzon, J. Navarro-Ortiz and J. M. Lopez-soler, "Link-level access cloud architecture design based on SDN for 5G networks," in IEEE Network, vol. 29, no. 2, pp. 24-31, March-April 2015.

[6] D. Wang, L. Zhang, Y. Qi and A. U. Quddus, "Localized Mobility Management for SDN-Integrated LTE Backhaul Networks," IEEE VTC Spring, Glasgow, pp. 1-6, May 2015.

[7] J. Wen and V. O. K. Li, "Data prefetching to reduce delay in software-defined cellular networks," IEEE PIMRC, Hong Kong, pp. 1845-1849, August 2015.

[8] C. C. Marquezan, Z. Despotovic, R. Khalili, D. Perez-Caparros and A. Hecker, "Understanding processing latency of SDN-based mobility management in mobile core networks," IEEE PIMRC, Valencia, Spain, pp. 1-7, June 2016.

[9] J. Banik, M. Tacca, A. Fumagalli, B. Sarikaya and L. Xue, "Enabling Distributed Mobility Management: A Unified Wireless Network Architecture Based on Virtualized Core Network," ICCCN, Las Vegas, NV, pp. 1-6, August 2015.

[10] V. Yazıcı, U. C. Kozat and M. O. Sunay, "A new control plane for 5G network architecture with a case study on unified handoff, mobility, and routing management," in IEEE Communications Magazine, vol. 52, no. 11, pp. 76-85, November 2014.

[11] 3GPP TS 123 401 V11.3.0, 2012. [Online]. Available: http://www.3gpp.org/dynareport/23401.htm

[12] M. R. Sama, S. Ben Hadj Said, K. Guillouard and L. Suciu, "Enabling network programmability in LTE/EPC architecture using OpenFlow," WiOpt, Hammamet, pp. 389-396, May 2014.

[13] Nowoswiat and G. Milliken, "Managing LTE Core Network Signaling Traffic," http://insight.nokia.com/managing-lte-core-network-signaling-traffic, July 2013.

[14] M. Bagaa, T. Taleb and A. Ksentini, "Efficient Tracking Area Management Framework for 5G Networks," in IEEE Transactions on Wireless Communications, vol. 15, no. 6, pp. 4117-4131, June 2016.

[15] O. Arouk, A. Ksentini and T. Taleb, "Group Paging-Based Energy Saving for Massive MTC Accesses in LTE and Beyond Networks," in IEEE Journal on Selected Areas in Communications, vol. 34, no. 5, pp. 1086-1102, May 2016.

[16] Ali-Ahmad, H., Cicconetti, C., De la Oliva, A., Mancuso, V., Sama, M.R., Seite, P. and Shanmugalingam, S., "An SDN-based network architecture for extremely dense wireless networks." IEEE SDN4FNS, November 2013.

[17] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, Steve Uhlig, "Software-Defined Networking: A Comprehensive Survey," Proceedings of the IEEE, vol. 103, No. 1, Jan 2015.

[18] S. H. Yeganeh, A. Tootoonchian, Y. Ganjali, "On Scalability of Software-Defined Networking," IEEE Communications Magazine, vol 51, issue 2, February 2013.

SDN on ACIDs

Maja Curic
Huawei ERC, Munich, Germany
maja.curic@huawei.com

Georg Carle
TU München, Germany
carle@in.tum.de

Zoran Despotovic
Huawei ERC, Munich, Germany
zoran.despotovic@huawei.com

Ramin Khalili
Huawei ERC, Munich, Germany
ramin.khalili@huawei.com

Artur Hecker
Huawei ERC, Munich, Germany
artur.hecker@huawei.com

ABSTRACT

Software-defined networks (SDN) do not guarantee coherent network operations, when uncoordinated SDN applications concurrently update the network forwarding state. As this problem has not so far received considerable attention, in this paper, we introduce transactional network-wide update support in SDN. We notably design and implement a new SDN service complementing state of the art proposals to achieve atomicity, isolation and durability of network updates in any typical SDN setup. The experiments with our implementation of the ACID service in FloodLight and OpenVSwitch demonstrate the practical feasibility of our proposal and good scalability to different network loads and sizes.

CCS CONCEPTS

• **Networks → Programmable networks**;

KEYWORDS

SDN, transactional updates, atomicity, isolation, durability

ACM Reference format:
Maja Curic, Georg Carle, Zoran Despotovic, Ramin Khalili, and Artur Hecker. 2017. SDN on ACIDs. In *Proceedings of CoNEXT '17, Incheon, Korea, December 12–15, 2017,* 7 pages.
DOI: TBA

1 INTRODUCTION

Previous research in Software-Defined Networks (SDN) has addressed the problem of consistent network updates [10, 14, 16, 19]. However, while consistency addresses the correctness of traffic policy changes, it does not deal with the execution completeness and with concurrency of network-wide updates. While these issues could be mitigated, to an extent, by preparing a monolithic SDN application from an overall network policy [3], that model does not suit a vision of SDN, where network administrators can add 3rd party applications from app stores like [1] to the controller. In particular, the latter vision is very relevant for so-called *cloud-assisted networking*, where deployable network functions can use SDN controller's north-bound interface to set up their respective network paths (e.g. combined NFV/SDN environments).

CoNEXT '17, Incheon, Korea

DOI: TBA

Consider, e.g., two SDN control apps, *Gaming App* and *Security App*, independently developed to set up suitable data paths for the end-users and provide native protection against attacks, respectively. Running on the controller, these apps could request network updates shortly after another [9]. If the selected data paths at least partly overlap (e.g. target the same switch *S0*), a conflict might occur: the rules installed say by *Security App* can break the reasons (e.g. load), for which *Gaming App* identified *S0* as suitable, or they could be obsoleted by the subsequent update by *Gaming App* (e.g. in case of match-field overlap). Also, the changes applied until this moment by *Security App* could conflict with the changes applied by *Gaming App* elsewhere. Hence, leaving such updates uncoordinated can lead to an incomplete execution or, worse, to an incoherent traffic forwarding configuration, thus failing to satisfy a single application.

Complementing related work on the consistency of SDN updates, in this paper, we design, implement and evaluate a novel mechanism to support atomicity, isolation and durability of SDN network updates. Inspired by distributed database management systems [4], our contribution is to add ACID properties [5] to SDN:

- We introduce a new mechanism to support ACID properties of SDN network-wide updates. It features network-wide atomicity, network-wide isolation with the strictest level, durability for the installed rules and works with distributed control planes and any SDN application.
- We show the practical feasibility of our proposal by implementing it in FloodLight and OpenVSwitch. Our design is sound, as the implementation reuses many of the existing provisions and requires only minimal changes.
- Finally, through a performance evaluation of our implementation in networks of different scales, we quantitatively show the practical relevance of our approach.

In the follow-up, we first provide a motivation example for our work. Next, we review relevant work from the previous SDN research. We then state our assumptions and requirements. As the state of the art fails to fulfill the latter, we present the design and implementation of the novel ACID SDN service. Then, we present the performance evaluation results of our implementation in the well-known Mininet testbed. We wrap up by a conclusion and an outlook to our future work in this area.

2 MOTIVATION

To explain the need for transactional updates in SDN, consider the scenario from Figure 1 that depicts a simple network with seven switches, a gaming server and two end users, *User 1* in green and *User 2*, in blue. Let us assume that *User 1* is a gamer, about to

access the gaming server. The network is controlled by a controller that runs two applications: a *Security App* and a *Gaming App*. The latter provides ultra-low latency (ULL) paths from the gamers in the network to the *Gaming Server*. The *Security App* monitors the traffic and takes appropriate measures to protect the network, whenever it detects sufficient evidence for an attack; such appropriate measures include migrating valid flows to a new path and possibly blocking unauthorized flows and ports.

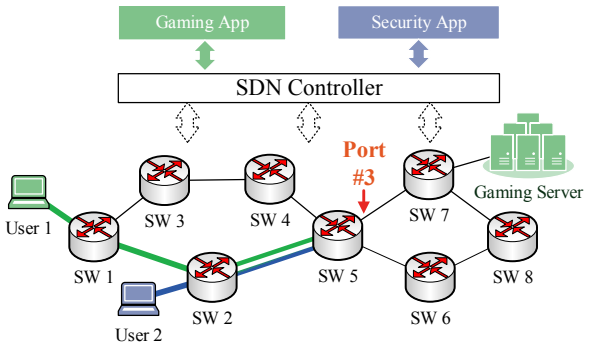

Figure 1: Example of non-atomic and uncoordinated network updates: The security application shuts down port 3 on switch 5 at approximately the same time, when the ULL application sets up a path that goes over that same port.

Assume *User 1* starts the gaming application; this generates a network event causing the SDN *Gaming App* to trigger installation of low latency path from *User 1*; hence, the *Gaming App* fetches all alternative paths between source and destination, checks state of the relevant switches and decides to install the path over available switches *SW1*, *SW2*, *SW5* and *SW7* to the *Gaming Server*. Assume that this process starts from switch *SW7*, as it is the closest to the controller. Assume now *User 2* triggered a DDOS attack a while ago consisting of a complex changing flow pattern and that, in response to that, the *Security App* has just decided to block *Port 3* on switch *SW5* in response; as the *Gaming App* did not finish installing its flows yet, *Security App* simply closes the port. Hence, we are facing a conflict: on the one hand, the *Gaming App* works based on an obsolete state of the network (at the moment of its check, the switch *SW5* looked available). As a result, *User 1* wil not be able to get the service; on the other hand, the *Security App* could not know that new valid flow rules were about to be installed on switch *SW5*. In addition, even if the *Gaming App* eventually realizes that it cannot go over *Port 3* of switch *SW5*, it has to clean up the artifacts it has possibly created in the network up to the moment it learned this.

There are several takeaways in this example. First, note that the problem is not due to authorization: while we can easily make sure that closing entire ports requires higher authorization levels (e.g. only available to the security app), this would not resolve the illustrated problem; it is actually the concurrency of both applications, unaware of each other and yet referring to the same resources, that produces a cumulated undesired effect of a network not delivering the service, yet supersed with unnecessary, partial or simply wrong forwarding rules. Second, the example shows that, while partial updates are useless for network-wide applications, there is actually no explicit confirmation of the network level success. Third and finally, the example also illustrates the substantially increased development effort: without any confirmation or indication of success or failure, a developer would have to use dedicated try-test-retry logic on every network-wide update, without actual guarantees of success; the testing part is an additional burden, producing longer, yet still unguaranteed, network update installation delays. Similarly, while it would be nice for developers to clean up unnecessary rules in case of partial failures during network updates, it is actually tricky to figure out, when this situation occurs.

3 RELATED WORK

OpenFlow v1.4 has introduced the concept of bundles. A bundle groups related state changes on a switch and executes them in an atomic and isolated manner. Bundles can be pre-validated on switches and committed by the controller. This way, network-wide changes across multiple switches can be synchronized to a certain extent. However, the OpenFlow specification indicates that bundle execution may fail during commit. Therefore, bundles cannot provide network-wide isolation and atomicity, when concurrent, network-wide updates affect multiple switches, as in the example above.

Several proposals consider execution ordering of commands within one update to achieve consistency properties. Reitblatt et al. [16] propose per-packet consistency of updates, which guarantees that each packet is always processed by a single network configuration. Jin et al. [9] propose a mechanism that aims to achieve the congestion freedom by considering the network state at the moment of the update. Cupid [19] improves the update time of [9] by enforcing parallelism for commands that do not interfere with each other. However, these proposals do not solve concurrency issues in SDN.

Several mechanisms for coordination of concurrent updates in SDN have been suggested. Schiff et al. [17] propose a synchronization framework for coordination of distributed updates, when multiple controllers concurrently update a switch state. They stamp the switch with the installed policy identifier and add primitives for reading and updating the stamp value. An SDN application can install the policy, only if it knows the current value of the stamp - this guarantees that the switch state was not changed meanwhile by another controller. The update works as follows: an application composes a bundle with the stamp update primitive at the beginning, followed by the update commands. While this work provides single-switch update atomicity and isolation, it does not yield the same properties for updates involving multiple switches.

Driven by a motivation similar to ours, Mizrahi et al. [12] introduced the concept of *scheduled bundles* (meanwhile incorporated in OpenFlow v1.5). It allows the commands in a bundle to be executed at a pre-determined time. This work is closest to our requirements, as it can serialize the execution of concurrent updates. However, the chosen approach comes with the cost of the perfect time synchronization between network elements, and any synchronization incorrectness may cause wrong schedules. Moreover, as switch update times depend on hardware capabilities, control load and the

nature of the updates [9], bundle commit end times may not be synchronized, although the bundle commit start times are, making packets on the fly during the update be processed by an intermediate state. Finally, the enforcement of such scheduled bundles requires a central entity to schedule all updates, which leads to a bottleneck and a single point of failure. In a distributed control plane, controller instances require a protocol and an overhead to agree on the execution times of their updates.

4 ASSUMPTIONS AND REQUIREMENTS

As the previous work (cf. Section 3) fails to address situations like in the example in Section 1, we here formulate our assumptions and requirements on a new method.

Scalability: As per state of the art (cf. [14]), we want the new mechanism to cover smaller SDN with a single central controller, bigger scale decentralized, hierarchical or replicated, controllers and fully distributed SDN setups.

Synchronization: Unlike the work in [12], we want to relax the strict time synchronization requirement for controllers and switches. Indeed, networks are often used for time synchronization and should be able to survive without it, especially during bootstrapping or after outages. Besides, simple network devices often do not store complex notions of local time [8]. Note that similar arguments led to a non-reliance on external time sources during the design of SNMPv3 [20]. This choice means that we cannot easily use any "do this after that" type of protocols, as messages sent in an order might arrive in a different order at the affected recipients.

Network Use: We make no assumptions on traffic arrival, distribution or on the type of applications supported. Notably, we would like the mechanism to be SDN-application agnostic. This means that we are not interested in resolving conflicts in application's own semantics (e.g. if an application creates a loop).[1] For mechanisms resolving semantic conflicts in SDN network policies, we refer an interested user to the policy-analysis tools such as, e.g., Frenetic [3].

Resource Granularity: For the purpose of this work, we assume that switch-level granularity is sufficient. In our future work, we will explore, whether and to which extent, finer resource granularity (e.g. at the port or at flow table level) can improve the performance.

Full ACID: The focus of this work is on the correct execution of possibly concurrent network-wide updates, which could occur in any time, order, quantities and places affecting any resource sets. To enable consistency, the suggested method should support integration with at least some of the previously proposed consistency methods [16] [9] [19].

5 OUR PROPOSAL

5.1 Design Choices

Our approach in this paper is to adapt relevant mechanisms from the distributed database management systems (DBMS) to the SDN area. Our proposal for isolation is motivated by the concurrency control method from databases known as two-phase lock protocol (2PL), which serializes events by utilizing locks over the data. During the transaction's execution, it locks the specific data and blocks other transactions from accessing it. Similarly, our proposal for atomicity is motivated by the two-phase commit protocol (2PC), a form of an atomic commit protocol (ACP), which coordinates all the processes that participate in a distributed atomic transaction on whether to commit to or to abort (roll back) the transaction. The protocol starts with the commit-request phase (or voting phase), in which a transaction coordinator prepares the participating processes to take the necessary steps (to vote) for either agreeing to or rejecting the suggested transaction. In the subsequent commit phase the coordinator decides, whether to execute or to abort the transaction based on the received votes. 2PL and 2PC are well studied and, together, form a de facto standard to achieve global serializability across database systems [6]. Each of these two basic protocols exists in many concrete variants, achieving slightly different properties at different costs (see e.g. [4, 5] for an overview).

In this paper we develop an SDN-specific ACP and use SS2PL schedules at switches, which provides strict global serializability [15]. This is a conscient design choice with consequences: as strict global serializability resolves all concurrency (more precisely, in our case, it only supports concurrency on disjoint resource sets), it might induce a relatively high cost. However, it also provides the strongest isolation guarantees and can therefore fulfill the expectations of all applications. In a commercial deployment, similarly to DBMS, we would expect the support for different isolation levels, to best trade off the requirements of different application classes against costs. In this paper however, we concentrate on the most rigorous support and evaluate the costs for SDN in Section 6. In our future work, we might explore alternative approaches, possibly providing weaker concurrency classes.

Finally, to achieve durability, we disallow explicit self-expiring rules in the resources; instead, we use a coordinated combination of the methods above to achieve a correct network-wide effect.

5.2 Architecture

We propose a new, so-called ACID Service for SDN, capable of executing actions requested by SDN applications as network-wide transactional updates. Its architecture is shown in Figure 2 and contains the three main parts:

(1) *ACID Transaction Manager (ATM):* implemented as a new module in SDN controller, it gets all requests from locally running SDN applications. It then handles the voting and locking of the affected remote resources;
(2) *ACID Resource (AR):* represents a resource, i.e. in this paper an SDN switch. We realize it as a simple switch modification capable of interpreting incoming vote requests and of handling local resources in the SS2PL fashion;
(3) *ACID SDN Protocol (ASP):* the ACP, i.e. in our view, a protocol between ATM and ARs on the SDN southbound interface. We here exclusively use the OpenFlow protocol without any changes, but we change the meaning and interpretation of these messages in the end-points (for details cf. Table 2).

All network status and update requests from the SDN applications requiring transactional properties are dispatched to the ACID Service, i.e. to an ATM. Note that our architecture supports several controllers, such that there can be several ATMs. The ARs

[1] This can be extended to several applications: for instance, if one application installs a policy, and the next application removes it immediately after, is fine from our point of view.

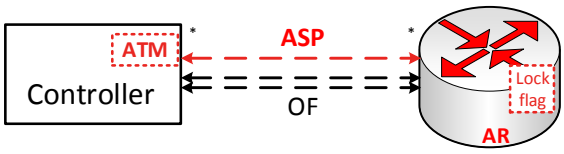

Figure 2: ACID Service Architecture

affected by the network update can be derived dynamically by the ATM from the network update request. Preliminary, management-level assignments of resources to ATMs are also possible; in this case, they should be reflected in a corresponding SDN application deployment to the controllers with ATMs.

Our architecture requires modifications of switches and controllers. If limited to a single controller deployment, with some constraints, the ACID Service can be implemented as a monolithic controller module, without changes on the switch (i.e. the voting/locking could be done virtually in the controller resource tables); depending on the round trip time (RTT) between the ATM and ARs, this might yield a faster transaction execution and, possibly, a higher transaction throughput. However, this solution would not be able to fulfill our scalability requirement. Besides, having the freedom to place many ATMs actually permits to optimize locality of ARs to the ATM, such that we believe that the performance cost would be similar to what is shown in our evaluation (cf. Section 6). The proposed solution is therefore more general, but permits to achieve a similar performance.

The ACID Service performs as follows:

(1) Upon the reception of an application update request, ATM sends an inquiry for a specific action execution (e.g. a flow rule installation) to all switches affected by the update. This message acts as ACP vote request to AR. We call this message `Vote-Lock`.
(2) On the reception of `Vote-Lock`, as per SS2PL, AR locally locks the resource, so it is not usable for other requests (e.g. from another ATM or application) until further notice from the initiating ATM. AR tries to apply all requested changes to the *staging area* []. If this is successful, the AR confirms to ATM using `Confirm`. Otherwise, the AR sends `Reject` and immediately unlocks.
(3) If all concerned ARs confirmed to process the proposed update (hence locking their resources), the ATM triggers the execution of the update by sending `Commit` to all ARs. If, however, at least one of the ARs rejected the request (e.g. because it is already locked, or e.g. because the update contradicts the existing rules) or does not reply within the expected time, then the ATM aborts the network update by sending `Rollback` to all ARs that replied with `Confirm`.
(4) Following `Commit`, the ARs activate the staging area. Following `Rollback`, ARs, i.e.switches, discard the staging area. In both cases, the ARs send `Finished` back to ATM and locally release the locks.

5.3 Implementation

We implemented the proposed architecture in an SDN network by introducing a new core service in the FloodLight SDN controller [2] and by making appropriate modifications to the OpenVSwitch SDN switch [18]. We change all applications to use the new core service on the controller. Instead of directly installing the network update, the application uses the API of ATM, which installs the update in the network in the transactional manner. A brief description of changes introduced at all SDN layers is shown in Table 1.

ASP Implementation: We implement the ASP using existing OpenFlow messages, as per Table 2. We implement `Vote-Lock` primitive as an OpenFlow bundle [18] with a lock request at the beginning (i.e., `OFPT_FLOW_MOD` with specific command and tableId combination), followed by the commands from the update. Primitive `Commit` in our implementation is a `OFPT_FLOW_MOD` message with command `OFPFC_DELETE` and tableId set to 255. The implementation details for the rest of the ASP primitives are shown in the Table 2.

ATM Implementation: We implement ATM as a new core service in the Floodlight controller. ATM assigns a unique, non-zero identification number (ID) of 4-byte length to each update. To ensure different IDs for updates from different ATMs (i.e., different controllers), we use the first byte to identify the ATM. This limits the number of ATMs to 255. The following 3 bytes are used to identify the updates from one ATM. This supports up to about 16 million concurrent updates from a single ATM. ATM populates the `xid` field of the ASP primitives with the respective update ID.

Figure 3 shows a typical message flow diagram with an active ACID Service. If at least one of the switches rejects the update or does not reply within the expected time, ATM starts the rollback procedure, by sending out `Rollback` to all switches that confirmed the update (Figure 4).

AR Implementation: In OpenVSwtich, we only require the following changes — the switch must a) set and hold the locking flag as necessary b) try updates in the staging area and c) activate it on `Commit`. We implement the locking flag as a 4-byte unsigned integer and add it to the `ofproto`, an OVS library that implements an OpenFlow switch. When transiting to state `locked`, the switch sets the lock flag to the update ID, which it reads from the `xid` field of the `Vote-Lock` message. We enforce mutex lock to limit the concurrent attempts of setting the locking flag. Hence, if the switch supports bundle parallelism and concurrent `Vote-Locks` arrive, only one of them can successfully set the lock flag.

To try to update the staging area, we split the existing bundle commit phase in two. In the first, the switch creates the staging area with a new version number and installs the update in it. Note that in case of failure, we do not require any change; the standard OVS code suits our purposes. In case of success, our modified switch does not activate the new version immediately; instead, it holds back the activation, yet it informs the ATM about the the installation outcome using standard `OFPBCT_COMMIT_REPLY`. Only in the second phase, i.e. after `Commit`, the switch activates that new version. Since this procedure is a single write operation on a pointer, we consider it atomic and assume that it cannot fail.

6 EVALUATION

We now evaluate the performance of the proposed solution under different network workloads and sizes. Our testbed is a Mininet [11] / Floodlight [2] environment running OpenFlow v1.4, including

Table 1: ACID Service Implementation summary

SDN Application	In our current implementation, all applications must be registered to the ACID Service. The application formulate all network status or update request using a new API primitive
SDN Controller	A new core service implements ATM and provides an API to the application layer. The latter adds the method *installNetworkUpdate()*, which executes the requests in transactional fashion
OpenFlow Protocol	No changes required
SDN Switch	Equipped with a new flag that holds the lock data (lock flag). The switch acts as AR and supports the proposed ASP.

Table 2: Protocol implementation summary

Primitive	OpenFlow message	Command	Type	Code	TableId
Vote-Lock	OFPBCT_OPEN_REQUEST	-	-	-	-
	OFPT_FLOW_MOD	OFPFC_MODIFY_STRICT	-	-	255
	(Network Update)	-	-	-	-
	OFPBCT_COMMIT_REQUEST	-	-	-	-
Reject	OFP_ERROR_MESSAGE	-	OFPET_BUNDLE_FAILED	OFPBFC_MSG_FAILED	-
Confirm	OFPT_BUNDLE_CONTROL	-	OFPBCT_COMMIT_REPLY	-	-
Commit	OFPT_FLOW_MOD	OFPFC_DELETE	-	-	255
Rollback	OFPT_FLOW_MOD	OFPFC_DELETE_STRICT	-	-	255
Finished	OFP_ERROR_MESSAGE	-	OFPET_FLOW_MOD_FAILED	OFPBRC_UNKNOWN	-

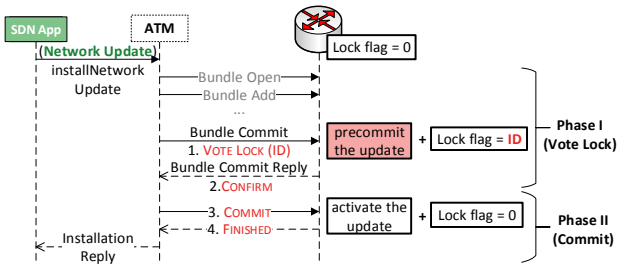

Figure 3: ACID Service Message Flow (Network update installation successful)

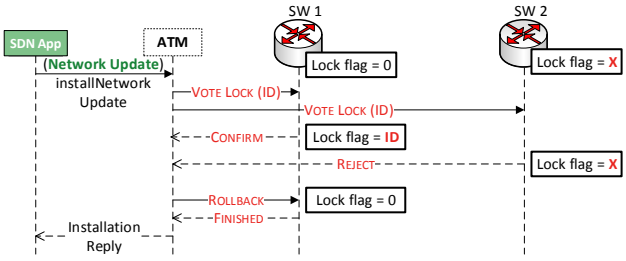

Figure 4: ACID Service Message Flow (Vote Lock phase partial failure)

our modifications to OVS and Floodlight (§5.3). We use network topologies generated from [7, 13]. They are hierarchical and consist of three layers (access, aggregation, and core), with K switches in the core, K pods of K switches each in the aggregation, and K^2 switches at the access. $K/2$ switches of each aggregation pod are connected to the access, with two access switches per each of these switches, and $K/2$ of them are connected to the core, each of which is connected to two core switches. The degree of connectivity in the core and within aggregation pods is 3. We consider no redundancy in the access part, e.g. each access switch is connected to only one aggregation switch. However, this is not a limitation, as we do not apply our reservation mechanism on access switches.

There is one user attached to each access switch, generating flows toward other users in the network. Flows arrive according to a Poisson process with parameter λ requests/sec. The source-destination pairs of the flows are selected randomly. The new flows are processed by an application running on top of a centralized controller, which determines the set of switches that are affected by each of these flows and sends the control updates to these switches. We implement a simple Floodlight application, which sets up the shortest paths between sources and destinations of the flows (if multiple shortest paths are available for a flow, one is picked randomly), and we assume that if the forwarding paths of two flows overlap, and if these paths are installed simultaneously, conflict would occur and both these installations would be unsuccessful (to mimic situations similar to what is discussed in Section 2 with *Security App* and *Gaming App*). Hence, our application needs to invoke ACID service at the controller to guarantee atomicity, isolation, and durability of its updates. Note that during these evaluation neither our SDN application nor the ATM are bottlenecks, as they process concurrent flow requests in parallel, and as our controller runs on a server with a lot of memory and CPU capacity (Huawei RH2288H server with 48 Intel CPUs and 24x 32GB of RAM). We run Mininet on a separate server, with a similar configuration, and interconnect both via unused Gigabit Ethernet link.

We measure the ratio of the flows that has been successfully processed by the application using ACID, referring to as *success rate*. The rejected requests, i.e. flows the paths of which are not installed due to the conflict, can be rescheduled by applying some sort of back-off mechanism, improving the success rate. In this paper, we show the results without applying any such mechanism and leave this topic to future studies. The parameters of our study are the flow arrival rate, λ, and the network-size. We vary the arrival rate from 16.7 to 100 requests/sec and consider three different topologies:

- a small topology ($K = 4$) – 16 access, 16 aggregation, and 4 core switches which has a network diameter of 6.
- a medium size topology ($k = 8$) – 64 access, 64 aggregation, and 8 core switches which has a network diameter of 7.
- a large topology ($k = 12$) – 144 access, 144 aggregation, and 12 core switches which has a network diameter of 8.

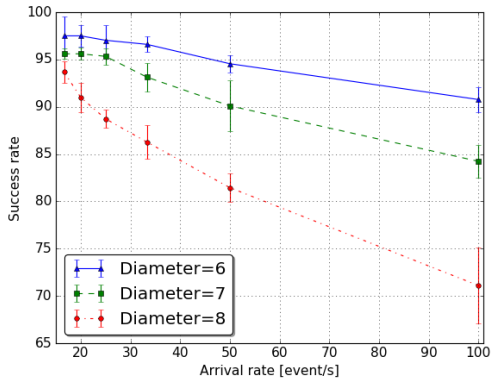

Figure 5: Success rate as a function of λ. The results are shown for different network sizes. We observe a sub-linear relationship between success rate and λ.

We show the success rate as a function of arrival rate for different network sizes in Figure 5. Each point in the graph represents the average success rate over 10 independent experiments and its corresponding 95% confidence interval. We run each experiment until a total of 300 flows are generated by the users.

We observe from the results that as λ increases the success rate decreases. This is expected, as the probability that two reservation requests collide with each-other increases as λ increases. The interesting observation is that this decrease is sublinear, or at most linear, which means that we can scale to larger rates, without observing significant drops in success rate. We also observe that the success rate decreases, as we increase the network size. This is expected as the average number of switches that must be updated per request is larger for larger network, increasing the possibility that two concurrent updates collide with each-other. Note that without our service, conflicts could not be detected, which could cause more severe issues in the network than just rejecting an installation request.

7 CONCLUSION AND FUTURE WORK

In this paper we propose a novel mechanism to achieve transactional network-wide updates in SDN (atomicity, isolation and durability). The proposed ACID service is easy to implement, does not require any complex provisions on switches and scales well to higher traffic arrival rates and larger networks. All in all, we show that transactional network-wide updates are practicable in SDN.

Yet, we mainly see this work as a mere first step towards the general adoption of transactional updates in SDN. Indeed, there are a number of ways to build on this result and come up with even more scalable solutions. For example, we plan to investigate possibilities of using more fine grained resources, e.g. switch ports or flow tables, rather than the switch as a whole. That would reduce the probability of conflict and thus improve the overall system throughput. Furthermore, we plan to design and evaluate more general methods to achieve ACID properties, most notably the commitment ordering of which SS2PL schedules are known to be a strict subset, and compare them to the SS2PL-typical locking presented in this paper, so that we can get a full picture of how transactional updates in SDN can perform.

REFERENCES

[1] HP Enterprise Aruba. 2017. SDN App Store. http://community.arubanetworks.com/t5/SDN-Apps/ct-p/SDN-Apps. (2017).
[2] Project Floodlight. 2017. Floodlight OpenFlow Controller. (2017). http://www.projectfloodlight.org/floodlight/
[3] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. 2011. Frenetic: A Network Programming Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 279–291. https://doi.org/10.1145/2034773.2034812
[4] H. Garcia-Molina, J. D. Ullman, and J. Widom. 2008. *Database Systems: The Complete Book* (2 ed.). Prentice Hall Press.
[5] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.). Morgan Kaufmann Publishers Inc.
[6] The Open Group. Distributed Transaction Processing: The XA Specification *(1991)*. http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf
[7] M. Howard. Using carrier Ethernet to backhaul LTE. White Paper, Infonetics Research *(Feb. 2011)*.
[8] IETF. 2003. RFC3512, Configuring Networks and Devices with SNMP. https://tools.ietf.org/html/rfc3512. (2003).
[9] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. 2014. Dynamic Scheduling of Network Updates. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 539–550. https://doi.org/10.1145/2619239.2626307
[10] N. P. Katta, J. Rexford, and D. Walker. 2013. Incremental Consistent Updates. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*. ACM, New York, NY, USA, 49–54.
[11] B. Lantz, B. Heller, and N. McKeown. 2010. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX)*. ACM, New York, NY, USA, Article 19, 6 pages. https://doi.org/10.1145/1868447.1868466
[12] T. Mizrahi and Y. Moses. 2016. Software defined networks: It's about time. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2016.7524418
[13] Ron Nadiv and Tzvika Naveh. 2010. Wireless Backhaul Topologies: Analyzing Backhaul Topology Strategies. White Paper. (August 2010).
[14] T. D. Nguyen, M. Chiesa, and M. Canini. 2017. Decentralized Consistent Updates in SDN. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. ACM, New York, NY, USA, 21–33. https://doi.org/10.1145/3050220.3050224
[15] Yoav Raz. The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Mangers Using Atomic Commitment *(ACM VLDB '92)*.
[16] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. 2012. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*. ACM, New York, NY, USA, 323–334.
[17] L. Schiff, S. Schmid, and P. Kuznetsov. 2016. In-Band Synchronization for Distributed SDN Control Planes. *SIGCOMM Comput. Commun. Rev.* 46, 1 (Jan. 2016), 37–43. https://doi.org/10.1145/2875951.2875957
[18] Open vSwitch. 2017. (2017). http://www.openvswitch.org/
[19] W. Wang, W. He, J. Su, and Y. Chen. 2016. Cupid: Congestion-free consistent data plane update in software defined networks. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 1–9.
[20] IETF Network WG. 2002. RFC3414, User-based Security Model (USM) for SNMPv3. https://tools.ietf.org/html/rfc3414. (2002).

Transactional Network Updates in SDN

Maja Curic, Zoran Despotovic, Artur Hecker
Huawei ERC, Munich, Germany
Email: {maja.sulovic, zoran.despotovic, artur.hecker}@huawei.com

Georg Carle
TU München, Germany
Email: carle@net.in.tum.de

***Abstract*—In current software-defined networks, network updates are neither atomic nor isolated. As such, when performed concurrently, they can lead to inconsistencies such as two conflicting policies installed in different parts of the network. This paper proposes transactional (i.e. atomic and isolated) network updates, embodied in a simple and clean architecture that parallels that of database management systems and involves a transaction manager (TM) running on the controller and a resource manager (RM) that runs on the switch. We implement these as extensions to a state of the art controller (Floodlight) and SDN switch (OVS). The implementation not only demonstrates the feasibility of the design but also shows the ease of its realization. We also evaluate our design by measuring how much time it takes to push transactional updates and at what throughput under various update arrival rates, for various update types and underlying topologies. We find out that, for example, with small path redundancy, we can setup almost all flows that arrive at a rate as high as $1000s^{-1}$, while providing to each a delay within only few milliseconds.**

I. Introduction

While the software-defined networking (SDN) paves the way to programmatic, event-based network updates, these are not very useful to developers in the current form. Indeed, in the current SDN, there is no guarantee that the network state is not altered, while a control application tries to apply some changes. Flow events arrive at the controller in an uncoordinated fashion, and, dispatched to control applications, could result in mutually conflicting or contradictory statements. This can have a nefast overall effect, resulting in either partial, conflicting or simply incorrect cumulative policies activated in the switches. Drawing a parallel to the well-known ACID properties of database transactions [1], SDN would profit from network updates that are:

- *Atomic*, meaning that the network-wide updates, going beyond one single switch, either succeed completely or not at all.
- *Consistent*, meaning that the network wide invariants and policies should continue to hold as the network state evolves, i.e. constraints imposed by those policies are never violated.
- *Isolated*, meaning that two overlapping operations, occurring at the same time and targeting the same switches, cannot result in unexpected effects. In essence, isolation eliminates inconsistencies that result from the concurrency as such and not from the semantics of individual operations. For example, if one application blocks a flow type, and some other application simultaneously allows this same flow type, then, with isolation, whoever comes last should win, i.e. the flow type cannot be blocked on a set of switches and allowed on another.
- *Durable*, meaning that the effect of a transaction survives indefinitely, unless it is changed by another transaction.

978–1–5386–1478–5/18/$31.00

Our standpoint is that SDN should institutionalize support for these services instead of transferring the burden to the developer, or simply neglect them, as it is the case now. Of the above four properties, only consistency has so far received sufficient attention [2] [3] [4]. We argue that atomicity, isolation and durability are equally important. This holds not only for a "distributed control plane" [5], in which multiple control applications that reside on potentially distinct controllers simultaneously update the network, but also for the most typical SDN deployment, namely that of a single SDN controller running a set of independent control applications.

We analyze the semantics of conflicts that can happen when multiple control applications simultaneously update the network and come up with the conclusion that so called *conflict serializable* update schedules can eliminate them. In database parlance, we permit only update schedules from the equivalence class of conflict-serializable (CSR) histories [6]. Among many possible scheduling algorithms (schedulers) that provide CSR histories, we consider the *strong two-phase locking* (SS2PL) [6] as the right choice. The main reason is that SS2PL is easy to implement in an SDN switch, as our proof of concept implementation indeed shows, giving us the opportunity to support transactional updates in SDN with a simple and clean architecture that parallels that of database management systems and involves:

- a transaction manager (TM) running on the controller and
- a resource manager (RM) that runs on the switch.

Our TM also coordinates the RMs via an Atomic Commit (AC) protocol (e.g. two-phase commit, 2PC), to achieve global atomicity. Thus our TM and the RMs achieve global serializability as well [7].

With our architecture, a network update holds locks over required switches, which thus become effectively unavailable for the others during the update. Our evaluations focus specifically on understanding if this is limiting and to what degree. We thus test the architecture for various update types, update rates and underlying topologies and find out that, for example, with small path redundancy, we can setup almost all flows that arrive at a rate as high as $1000s^{-1}$, while providing to each a delay within only few milliseconds.

II. Motivation

To see the need for transactional updates in SDN, consider an example network from Fig. 1. It has eight switches (SW 1 to SW 8) and four user nodes (User 1 to User 4). It is controlled by one controller with two SDN applications, the Green App and the Blue App, that compete for a given resource. That can be bandwidth of a link or even flow table of a switch. Let us assume the former, just to make the exposition to follow concrete. Note as well that in the latter case, the two can be literally any applications. So let the bandwidth budget of all links in the network be 1.5 Mbps, while the applications are about to reserve 1 Mbps paths each. If the reservations are concurrent the following outcome can happen: one application has reserved the link SW 2 - SW 5, while the other one holds the link SW 5 - SW 7. So none of them can proceed, both will have to give up their reservations.

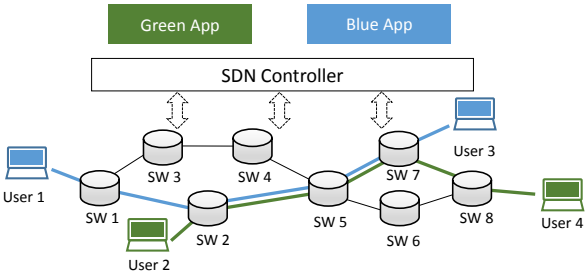

Fig. 1. Example network to illustrate non-atomic and uncoordinated updates.

This issue is an artifact of concurrency, that is inherent to any SDN deployment, centralized or distributed, with one or multiple controllers. See [5], [8] for more examples of the same problem. In concrete terms, we are dealing with an instance of the so called "inconsistent read" [6]. Let x and y denote the available bandwidth of the links SW 2 - SW 5 and SW 5 - SW 7, respectively. Then the description above is equivalent to the following pattern of interleaved read and write operations:[1]

$$r_1(x)r_2(y)r_2(x)w_1(x)r_1(y)w_2(y)w_1(y)w_2(x),$$

where the subscripts denote the two apps. We argue that this schedule is not *conflict serializable* [6] [1], i.e. it cannot be equivalent to any serial schedule. Thus it must be eliminated by appropriate concurrency control mechanisms. After a review of available mechanisms and their suitability to SDN, this paper suggests a full solution to this problem.

III. Related Work

The idea for transactional updates was developed in the area of database management systems (DBMS). In transactional DBMS, a Transaction Manager (TM) coordinates transactions that span multiple Resource Managers (RM). RMs implement a concurrency control mechanism to support concurrent change requests to the *local* resource in a more or less rigorous manner. Depending on this rigor, concurrency control can be pessimistic or optimistic [7] and yields different *isolation levels*, progressively accepting possible conflicts in favor of more concurrency and, hence, better performance. Unfortunately, as our examples show, conflicts can be a real problem for some SDN applications; hence, in the following, we concentrate on schedulers, which fulfill high isolation requirements.

Two-phase locking (2PL) is a pessimistic mechanism. An RM uses 2PL to grant locks over data to a single transaction, forcing the other transactions to wait if they want to access the same data. In its basic form, 2PL guarantees serializability. Two additional variants of 2PL: S2PL (strict 2PL) and SS2PL (strong 2PL) provide recoverability and cascadelessness, respectively. *Commitment-ordering* (CO) schedulers are optimistic. They serialize the execution of the transactions by building and analyzing a transactions' precedence graph. This generally gives better performance than the pessimistic schedulers, but comes at an additional cost, complexity of algorithms that the RM has to run.

When transactions span multiple RMs, it might be necessary to provide *global* serializability across the RMs. If all RMs are synchronized and share common info, e.g. timestamps, this can be done with a timestamp-ordering protocol. Alternatively, a TM that uses an Atomic Commit Protocols (ACP), such as 2PC, and coordinates RMs that provide a concurrency control mechanism such as SS2PL will just do [7].

In SDN, OpenFlow v1.4 has introduced the concept of bundles. A bundle groups related state changes on a switch and executes them in an atomic and isolated manner. Bundles can be pre-validated on switches and committed by the controller. However, the OpenFlow specification indicates that bundle execution may fail during commit. Therefore, bundles cannot provide network-wide isolation and atomicity, when concurrent, network-wide updates affect multiple switches.

Recently, Schiff et al. [5] proposed a synchronization framework for coordination of distributed updates, in which multiple controllers concurrently update a switch. They stamp the switch with the installed policy identifier and add primitives for reading and updating the stamp value. An SDN application can install the policy, only if it knows the current value of the stamp - this guarantees that the switch state was not changed meanwhile by another controller. The update works as follows: an application composes a bundle with the stamp update primitive at the beginning, followed by the update commands. Using bundle for installation together with stamping achieves serializability and recoverability, but only if such updates affect a single switch. It remains unclear, how to extend this to a network-wide update.

Driven by a motivation similar to ours, Mizrahi et al. [9] introduced the concept of *scheduled bundles* (meanwhile incorporated in OpenFlow v1.5). It allows the commands in a bundle to be executed at a pre-determined time. As such, this work is closest to ours, as it can serialize the execution of concurrent updates and, hence, create a serial history. However, the chosen approach comes at the cost of perfect time synchronization between network elements, and

[1]Reads do not happen "physically". They rather exist logically, as the apps tacitly assume that there is enough bandwidth.

any synchronization incorrectness may cause wrong schedules. Moreover, as switch update times depend on hardware capabilities, control load and the nature of the updates [8], bundle commit end times may not be synchronized, even when the bundle commit start times are, making packets on the fly during the update be processed by an intermediate state. Finally, the enforcement of such scheduled bundles requires a central entity to schedule all updates, which leads to a bottleneck and a single point of failure. In a distributed control plane, controller instances require a protocol and an overhead to agree on the execution times of their updates.

Canini et al. [10] have a focus on a similar problem. They propose transactional middleware for semantic composition of distributed updates. It intercepts the updates, serializes them and modifies when necessary, to prevent overriding the existing network configuration. As the middleware processes the updates sequentially, it must be single-threaded, which impedes the benefits of the control plane distribution. Besides, the proposal does not provide technical implementation details [10] or an analysis of the middleware design complexity.

A number of works considers ordering of the execution time of the commands within one network update and proposes mechanisms that provide different update properties. Reitblatt et al. [11] proposed two-phase update protocol to achieve per-packet consistency, which guarantees that each packet traversing the network is never processed by a mix of two configurations. Jin et al. [8] proposed Dionysus, a mechanism for congestion free network updates. It requires a rule update that brings a new flow to a link to occur after an update that removes an existing flow if the link cannot support both flows simultaneously. However, these proposals are not solving concurrency issues in SDN.

IV. Design Choices

We briefly review available solutions to atomicity and isolation and analyze their suitability to SDN.

To apply the transactional DBMS architecture to SDN, we wonder where should the TM and the RMs run? While it is fairly clear that the TM should be a module in the SDN controller, there are a number of options for RMs.

First, one can place all RMs locally within the SDN controller, where they would act as images of the switches. With this, we are free to select any concurrency control in RMs, optimistic ones are equally good as the pessimistic, even though they are more complex. However, we have a problem if we go beyond the single controller deployment: it is hard to maintain switch images in several controllers that simultaneously update the network.

At the other extreme, we can incorporate the RM in the switch. That is least constraining regarding the target deployment. The downside is that we need to extend the switch. As the CO schedulers need to maintain the precedence graph, buffer the received transaction requests and reorder their respective commits, it might not be easy to implement an optimistic mechanism within a switch. On the contrary, the pessimistic ones appear feasible, as the RMs should only run a 2PL scheduler and implement the cohort side of the 2PC[2].

Third, we can place RMs in middleware, in an attempt to eliminate cons of the above two extremes. But then we need a new protocol that RMs could use to coordinate updates across their controlled domains.

In this paper we integrate the RM within the switch. We see that as a reasonable first step towards integrating transactional updates with SDN. Other options will be investigated in our future work.

We now briefly consider integration possibilities of our proposed architecture with some of the existing consistency mechanisms. In the proxy-based architectures, e.g. VeriFlow [3] or NetPlumber [4], TM would run as the core service in the proxy. Once VeriFlow or NetPlumber verified the update, they dispatch it to the network using the TM's API call. The same applies for integration with consistency enforcing mechanism(s) that are integrated in the controller, e.g. SE-Floodlight [13]. We do not consider integration with mechanisms such as Dionysus [8] and Cupid [2] since these assume specific application types and network states, which opposes to our own assumptions and requirements to achieve SDN-application agnostic mechanism.

V. Our Proposal

A. Architecture

Fig. 2 shows our high level architecture. The additions to the standard SDN models are shown in red. These include the transaction manager (TM) that runs in the SDN controller, the resource manager (RM) running in the switch and the transactional SBI (TSBI), an extension of the southbound interface for the TM-RM communications. The TM gets update requests from SDN apps and communicates with the required RMs on behalf of them, i.e. it handles the transactions. Our RM essentially transforms the SDN switch into a transaction-aware medium with well defined states that can be accessed and changed only according to transactional semantics (e.g. failed transactions can be undone).

The architecture obviously applies to the single controller SDN deployment. It besides holds for deployments with multiple controllers in which each controller, i.e. each TM, is associated with all switches of the network.

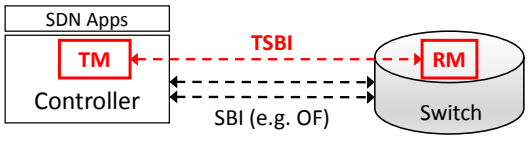

Fig. 2. Transactional SDN architecture.

As an illustration, let's go through a typical sequence of steps in a transactional network update cycle (cf. Fig. 3):

1) Upon the reception of a network update request from an SDN app, the TM sends a so-called `Vote` message to

[2]We assume that potential TM failures can be solved with existing proposals for fault-tolerant SDN controllers, e.g. [12].

the RMs involved in the update. The resource (switch) interprets this message as both transaction initiation and an ACP vote.

2) On the reception of `Vote`, as per SS2PL, each RM locally locks the resource, so it becomes unavailable for other requests (e.g. from another TM or application) until further notice from the initiating TM. Every RM tries to apply all requested changes to its respective *staging area*. If this is successful, the RM sends a `Confirm` message back to the TM. Otherwise, the RM sends `Reject` and immediately unlocks the resource.
3) If the TM got confirmations from all the RMs in the current update, it starts the commit phase by sending a `Commit` message to them. If at least one RM rejected the request (e.g. it is already locked) or did not reply within the expected time, the TM aborts the update by sending `Rollback` to the RMs that replied with `Confirm`. Thus, even if the TM receives `Confirm` from all but a single switch, it will release the obtained locks and start anew. This is a possible improvement area and will be investigated in a future work.
4) On `Commit`, the RMs activate their staging areas, whereas on `Rollback`, they discard these. In both cases, they send `Finished` back to the TM and release the locks.

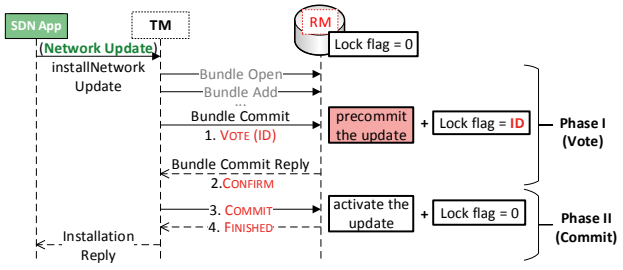

Fig. 3. Message flow of a typical transactional network update.

To achieve durability, we disallow explicit self-expiring rules in the resources; instead, we use a coordinated combination of the methods above to achieve a correct network-wide effect.

Note that we merge the transaction initiation and the beginning of the atomic commit protocol, which might limit the range of supported transactions to write updates only, i.e., we rule out the read requests. We believe that such updates are sufficient, because of the OpenFlow limitations: indeed, as per OpenFlow v1.4 and later, the Read-State messages can only retrieve a very limited information from the switch (e.g. values of counters), most of which is data plane driven; this data is not available for transaction management, i.e. the transactions that write on them are not under OpenFlow control. As we do not aim to prevent the situations, where SDN applications might read network information that becomes obsolete only a moment later due to the data plane changes and rather concentrate on resolving potential conflicts from other, concurrent updates, we disregard "read before write". Merging the locks and the ACP votes permits to reduce the time of the switch lock, which is a performance-related parameter, the implications of which we analyze and show in Section VI.

B. Implementation

In our PoC implementation, the TM is a new module in the Floodlight controller [14], while the RM is an extension of the OVS [15]. Both implementations are available for download [16]. The two communicate via the unchanged OpenFlow. More details on this follow.

TSBI Implementation: We implement the TSBI using existing OpenFlow messages, with specific combination of values for certain fields (i.e., `command`, `type`, `code` and `table_id`). For example, `Commit` in our implementation is a `OFPT_FLOW_MOD` message with `command` and `table_id` fields set to `OFPFC_DELETE` and 255, respectively. We implement `Vote` primitive as an OpenFlow bundle [15] with a transaction initiation primitive at the beginning, followed by the commands from the update.

TM Implementation: Our TM is a new core service of Floodlight. It implements the interface *IOFMessageListener*, to be notified when the controller receives an OF message. It registers to receive only those OF messages, which in our implementation are interpreted as the TSBI primitives sent by RM (`Confirm`, `Reject` and `Finished`). The other messages are irrelevant to the TM.

Similarly, to use the TM, SDN applications must register. They then use the exposed method *transactionalUpdate()* to push their updates in a transactional manner. The method parses the update, detects the affected switches and starts the update. Each update gets a unique ID. We maintain an update table, indexed by the ID, that stores the full update info such as the update itself, its state (including e.g. pending `Votes`, reattempt counter, etc.). We transfer these IDs in the exchanged messages and use them to retrieve the update state and decide on the state transition. The apps can also specify the maximum number of update installation attempts `MAX_N_INST`, that is used to limit the number of retries in case of update installation failures.

RM Implementation: We extend OVS such that it can a) set and hold a lock flag as necessary, b) perform updates in the staging area and c) activate them on `Commit`. We implement the lock flag as an integer and add it to the `ofproto`, an OVS library that implements an OpenFlow switch. When transiting the state to `locked`, the switch sets the lock flag to the update ID, which it reads from the `xid` field of the `Vote`. We use a mutex to provide exclusive access to the lock flag.

To update the staging area, we first change the method `do_bundle_commit()` in `ofproto` such that it creates the staging area with a new version number, equal to the update ID, and installs the update in it. We then modify the `handle_flow_mod()` method such that it accepts only messages whose `xid` matches the lock flag. It then activates (discards) the staging area following a `Commit` (`Rollback`). Since the activation is a single write on a pointer, we consider it atomic and assume that it cannot fail. Finally, the switch clears the lock flag.

VI. Evaluation

A. Simulation environment

Whereas our previous work [17] demonstrated our initial design and feasibility of its implementation, we provide here a comprehensive set of evaluations that permit us to judge on the performance of our transactional SDN. To have as accurate evaluations as possible, we use a rather complex evaluation environment that mixes simulations and Mininet based emulation. In essence, we simulate the full system operation, but learn and set up its relevant properties with help of our emulation. Our simulator is event driven and its key steps are as follows. We first generate a sequence of updates from SDN applications which arrive at the TM in the SDN controller at a specific rate. Each update is a path setup request that has to be atomic. Our simulations maintain a data structure that tracks the locks of each individual switch. So when an application sends vote requests to the switches $s_1, \ldots, s_k$, we change the state of $s_1, \ldots, s_k$ to "locked" and also indicate the times when the locks start and stop, as well as the ID of the request that holds the locks. Thus we can easily see if a switch is free or not when a new request attempts to lock it and record that as a vote failure.

An obvious question here is: what reservation (lock) times do we indicate for the switches? This is where we make use of our emulation. We create a linear Mininet network with the diameter k between 1 and 10 (networks with larger diameters are hard to expect [18]) and low network load (to avoid any congestion on the SDN controller), send `Votes` to each of the k switches and measure the times until they send `Finished` each. We plot these times as CDFs for each $k = 1, \ldots, 10$ in Fig. 4. So, to determine the lock times, we simply see how many switches k a request should lock, pick the CDF for that k and generate k random times from that CDF, one for each switch involved in the update.

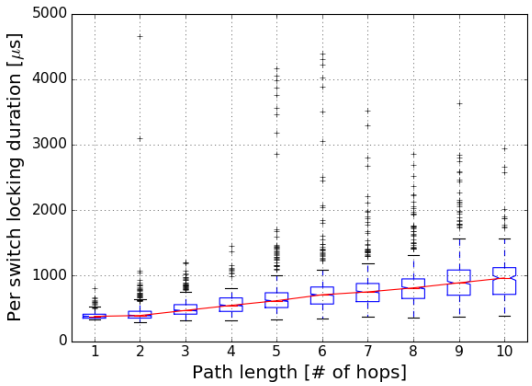

Fig. 4. Locking duration per switch measured over our Mininet/Floodlight testbed. The results are shown for different path lengths.

B. Simulation results

The load in our simulations is a series of requests to setup paths between randomly selected access switches (see below) of our simulated networks. The paths, and thus the switches to lock, are determined as the shortest paths between the corresponding access switches (if multiple shortest paths exist, one is randomly selected). The requests are modeled as a Poisson process with rate λ. We vary λ from 10 to 1000s^{-1}.

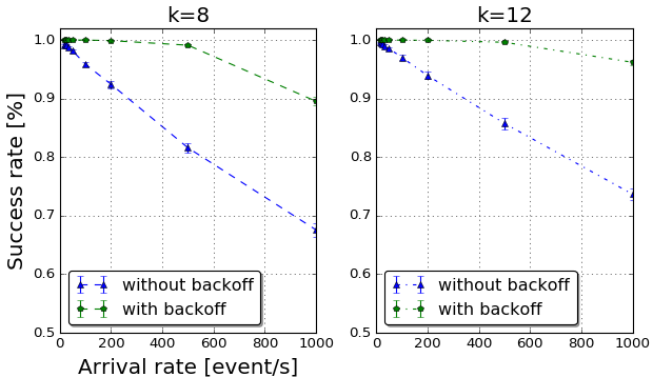

Fig. 5. Success rate for different network sizes and request arrival rates.

We use network topologies generated from [19], [20], representing backhauls of carrier networks. They are hierarchical and consist of three layers: access, aggregation, and core. The total size of the network is controlled with one parameter, k, that coincides with a number of topological properties of the network [19], [20]. In particular, we use two different topologies, with $k = 8$ and 12, termed *medium* and *large*. They have 160 access, 64 aggregation, and 8 core switches, respectively, 360 access, 144 aggregation, and 12 core switches.

Fig. 5 presents the success rate as a function of the request arrival rate, without and with back-offs. When a request is rejected, our (rather basic) back-off mechanism reschedules it again after a random time T_{BACK_OFF}, drawn from the exponential distribution with mean 0.01. We limit the number of installation reattempts to 3. Each point in the graph represents the median success rate over 10 independent experiments and its corresponding 95% confidence interval. We run each experiment with 10000 flows from the users.

Without reattempts, blue line in Fig. 5, the success rate decreases when λ increases. The decrease is sublinear as there are no back-offs to effectively increase the arrival rate and drive the system to a complete collapse. At the same time, back-offs significantly increase the success rate, e.g. in the large network and 1000 requests/sec, less than 5% of the requests were not installed. It is this result, supported by the results shown in Fig. 6 and discussed shortly, that makes us believe that our architecture is viable in SDN networks.

As we lock switches along whole paths, we expect better performance for networks with more redundancy. We achieve that by providing additional links between the aggregation and the core. While the previous experiment had on average $n = 2$ core switches connected with each aggregation pod, we now increase that value to 3 and 4. Fig. 6 shows how the success rate goes up, approximately 15% without, and almost reaching 100% with reattempts.

Figure 7 depicts the CDF of the path setup time. The results shown hold for $k = 12$ (large network), $n = 2$ and $\lambda = 1000$ requests/sec, backoffs were on. We observe that the setup time

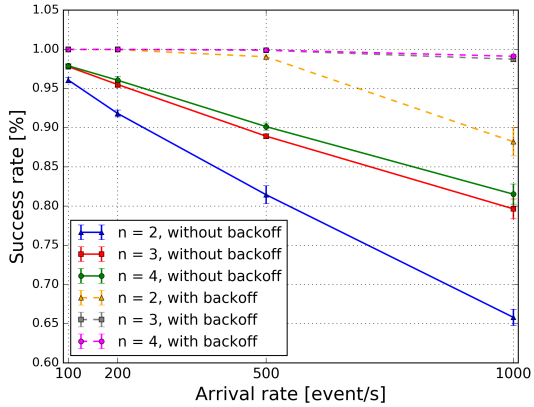

Fig. 6. The effect of (path) diversity: Success rate for different values of the core-aggregation connectivity parameter n and different request arrival rates. Medium size network, k = 8, in all cases.

of around 75% of the updates is below 6ms and that the 95-percentile of the path setup time is below 30ms. The long tail of distribution is related to the updates which were installed after backoffs, through reattempts.

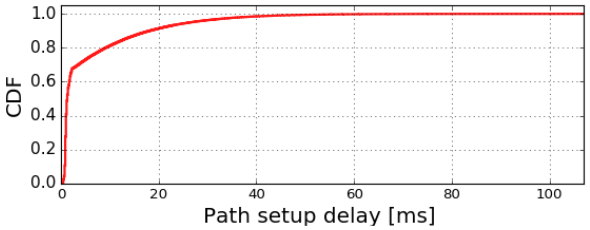

Fig. 7. CDF of the path installation delay. The results hold for the large network, k=12, 3 installation re-attempts and arrival rate of 1000 requests/sec.

VII. Conclusion and Future Work

In this paper we propose a novel service to achieve transactional network-wide updates in SDN (atomicity, isolation and durability), based on 2PC protocol and SS2PL concurrency control mechanism from transactional DBMS. The proposed service is easy to implement, does not require any complex provisions on switches and scales well to higher traffic arrival rates and larger networks. We show that transactional network-wide updates are practicable in SDN. Our simulation results show that, although the resource locking can cause rejection of the concurrent updates and hence deteriorate the overall rate of successfully installed updates, enforcing multiple update installation attempts or increasing network redundancy can result with almost 100% successfully installed updates.

In our future work, we plan to investigate the complexity and performance of other schedulers, most notably of commitment ordering techniques and, possibly, of mixed strategies. Another promising possibility is the mentioned use of more fine grained resources, e.g. switch ports or flow tables, rather than the switch as a whole. Both should reduce the probability of conflict and thus improve the overall system throughput.

Acknowledgement

This work has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 671551.

References

[1] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008.

[2] W. Wang, W. He, J. Su, and Y. Chen, "Cupid: Congestion-free consistent data plane update in software defined networks," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, April 2016, pp. 1–9.

[3] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 49–54.

[4] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 99–112.

[5] L. Schiff, S. Schmid, and P. Kuznetsov, "In-band synchronization for distributed sdn control planes," *SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 1, pp. 37–43, Jan. 2016.

[6] G. Weikum and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[7] Y. Raz, "The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource mangers using atomic commitment," in *Proceedings of the 18th International Conference on Very Large Data Bases*, ser. VLDB '92, San Francisco, CA, USA, 1992, pp. 292–312.

[8] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 539–550.

[9] T. Mizrahi and Y. Moses, "Software defined networks: It's about time," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, April 2016, pp. 1–9.

[10] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "Software transactional networking: Concurrent and consistent policy composition," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 1–6.

[11] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 323–334.

[12] N. Katta, H. Zhang, M. Freedman, and J. Rexford, "Ravana: Controller fault-tolerance in software-defined networking," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: ACM, 2015, pp. 4:1–4:12.

[13] P. A. Porras, S. Cheung, M. W. Fong, K. Skinner, and V. Yegneswaran, "Securing the software defined network control layer." 2015.

[14] Floodlight, "Floodlight OpenFlow Controller – Project Floodlight." [Online]. Available: http://www.projectfloodlight.org/floodlight/

[15] O. vSwitch. http://www.openvswitch.org/.

[16] "Acidrepo," https://github.com/OVSacid/ACIDRepo, 2018.

[17] M. Curic, G. Carle, Z. Despotovic, R. Khalili, and A. Hecker, "Sdn on acids," in *Proceedings of the 2Nd Workshop on Cloud-Assisted Networking*, ser. CAN '17, New York, NY, USA, 2017, pp. 19–24.

[18] P. Mahadevan, D. V. Krioukov, M. Fomenkov, B. Huffaker, X. A. Dimitropoulos, K. C. Claffy, and A. Vahdat, "Lessons from three views of the internet topology," *CoRR*, vol. abs/cs/0508033, 2005.

[19] R. Nadiv and T. Naveh, "Wireless backhaul topologies: Analyzing backhaul topology strategies," White Paper, Ceragon, August 2010.

[20] M. Howard, "Using carrier ethernet to backhaul lte," White Paper, Infonetics Research, February 2011.

FitSDN: Flexible Integrated Transactional SDN

Maja Curic, Zoran Despotovic, Artur Hecker
Huawei ERC, Munich, Germany
{maja.sulovic, zoran.despotovic, artur.hecker}@huawei.com

Georg Carle
TU München, Germany
carle@net.in.tum.de

***Abstract*—Extremely popular with the research community, the SDN has indeed matured and mostly resolved the original performance doubts. The next frontier is to make it popular with the end users, i.e. the administrators and the developers, without sacrificing the performance. Recent distributed SDN proposals still come with a number of design decisions that appear limiting with respect to the ease of administration, application development, and performance. For instance, opting for strong consistency in the controller design results in a complex network administration and can severely impact the performance; choosing the eventual consistency delegates the real problems to application developers: while this design makes the administration lighter, to get the expected behaviour and performance, it requires developers to have profound distributed systems understanding. We argue that this is a bad tradeoff. Consequently, we design, implement and evaluate a novel Flexible Integrated Transactional SDN (FitSDN) that achieves the ease of development and administration, as well as good performances. In a nutshell, FitSDN eliminates the replication of critical network state in the controller instances and taps switch flow tables under a transactional semantics instead. FitSDN enables simple and intuitive APIs that permit SDN application developers to focus on the actual application logic. Comparing FitSDN to a popular distributed controller, FitSDN achieves around 4 times faster responses to network events.**

***Index Terms*—Transactional SDN, distributed SDN.**

I. Introduction

SDN emerged as a paradigm that has the power to change networking. However, even after years of immense research and thousands of published papers, SDN has not established itself as a technology that network operators readily embrace to solve their problems. While the available SDN solutions, e.g. first generation controllers (Pox [1] or Floodlight [2]) and switches (OVS [3]) did a good job of demonstrating the concept, SDN still has a long way to go to become production-ready. To deserve that label, SDN must collect highly positive scores along each of the following four dimensions:

- Performance, meaning not only short delays in responding to network events, but also high availability and resilience, as well as the ability to scale well with network sizes and geographical footprints.
- System administration, meaning acceptably low involvement of human administrators in the course of both normal and exceptional system operation (e.g. failures).
- Application development, meaning rich APIs available to the developers to easily and quickly develop their applications, as well as supporting the trial and error model that would attract the critical mass of developers.
- SDN model richness, meaning that range of possible tasks to perform in the network, i.e. types of events to handle, should be as wide as possible, at least matching what OpenFlow [4] sets as a benchmark.

Judging by the recent papers, SDN researchers are embracing this view. Specifically, there seems to be a consensus that a *distributed* controller design is key to push SDN to the next level of maturity. Whereas there is a striking harmony with respect to requirements and key architectural principles of the sought distributed controller, there is quite some disagreement with respect to details of its realization. Thus, all agree that a highly scalable and resilient distributed SDN controller must run as a set of instances, that these instances can be added or removed without disrupting the system and that they should work together to create what appears to the applications as a single, logically centralized controller. However, there is dissonance when it comes to the level and type of interaction among the controller instances. The proponents of the *tight coupling*, such as ONOS [5] and Onix [6], advocate using strong consistency protocols, e.g. Raft [7] and Paxos [8], to coordinate the operation of instances. These require coordination prior to performing any action in the network and are, as such, pessimistic, i.e. try to avoid problems. The other camp advocates *loosely coupled* systems, such as HyperFlow [9] and SCL [10], in which controller instances run mechanisms for eventual consistency, such as publish/subscribe messaging [11] or gossip protocol [12].

Our paper takes the viewpoint that both camps fail along at least two of the said four dimensions. Roughly, the strong consistency camp fails to provide satisfactory controller performance and presents complicated systems that are hard to manage, while the eventual consistency camp has severe problems with respect to application development and unacceptably limits the underlying SDN model. In our opinion, the root of the problem lies in the need to *replicate* the network state in the controller instances, inherent to both camps. With that assumption, the two designs are essentially two sides of the same coin, and, despite plenty of appealing architectural choices they make, they cannot arrive at satisfactory solutions regarding all four categories.

In this paper, we propose to eliminate state replication in the controller instances. Instead of coordinating the replicas in controller instances, our network features transactional access to allow controller instances to perform their tasks atomically and in isolation from each other on the unique network state. More concretely, we propose a novel Flexible Integrated Transactional SDN (FitSDN), a distributed SDN controller built around the transactional access to switches. We present

our design and details of controller, switch, protocol extensions with support for backward compatibility (e.g. in case of deployed hardware switches). Further, we discuss our open source implementation based on the state of the art controller (FloodLight) and switch (OVS). We show that FitSDN scores well along all four critical dimensions introduced above. In particular, FitSDN controller performs almost without any deterioration due to the controller distribution per se; it rather behaves as a set of single image controllers that share their load. With this design, FitSDN is easy to manage: different failure situations are easy to handle and the controller scales out naturally in the sense that it does not require existing instance reconfiguration. What is more, FitSDN delivers a transactional API to the app developers that is a comfortable abstraction to relieve the developer from having to foresee and watch for a broad range of possible exceptions in the distributed system. FitSDN does not constrain the SDN model in any sense.

These improvements do not come for free, however. The price to pay is an increased complexity of the network element, i.e. the switch. In our design, the switch needs to host a module that checks each controller command and arranges its processing according to transactional semantics. There exists a range of algorithms to do this, which trade off performance for complexity. We show that our current choice, namely locking, nicely hits a sweet spot between the extremes: it is fairly easy to implement, while it brings satisfactory performance. Thus, we conclude that the said price is not unacceptably high.

The paper starts in Section II with the background, in which we analyze the main available papers along the four set dimensions. We present our design in Section III and its evaluation in Section IV. We conclude in Section VI. Finally, in Section VII, we provide the link to the source code of FitSDN.

II. Background

We spend some time in this section to understand how the state of the art distributed controllers behave with respect to the four established criteria.

System performance. In tightly coupled systems, the controller instances run strong consensus protocols such as Raft to agree on the order of network state updates. Raft designates a node, called Raft leader, to coordinate all read and write requests. Each write request in a Raft cluster requires several exchanges between the leader and the other nodes, Raft followers. These exchanges introduce extra delay in responding to network events, i.e. require time that can be prohibitively high, especially when the cluster is geographically distributed [10], [13]. The followers in a plain Raft cluster do not serve any read requests, but instead forward them to the leader, which can overload the leader and lead to under-utilization of the followers [14], [15]. To address this, ONOS partitions its network state (precisely, its "resource store") into a set of shards and runs a small (with three nodes only), separate Raft cluster for each shard. Now, when an update spans multiple shards, ONOS uses the two-phase commit (2PC) protocol [16] to coordinate operation of the involved Raft clusters. That is, however, another source of delay, a fact also known to ONOS developers [17], [18].[1]

In contrast, loosely coupled systems dispense with strong consensus protocols. At a first glance, this permits a controller instance to quickly respond to a network event, as it does not need to coordinate with others. However, as is the case with SCL and HyperFlow, each instance should locally reprocess every event from the network in order to arrive at the correct network state. But therein lies a serious performance penalty, as such a strategy scales badly.

System administration. Strong consensus protocols are complex and exhibit problems, as soon as they get out of their comfort zone and face failures. One should keep in mind that they are susceptible to failures of both running nodes (processes) and the transport network interconnecting them, as they require a full mesh connectivity between the nodes. There is a well-documented list of failure scenarios, in which Raft (or, at least, the available implementations thereof) fails at providing liveness and safety and leaves the system in a deadlock state, such as oscillating leadership [20], permanent split brain [21] and serious performance degradation [22], which can only be solved by a network administrator's intervention. It is important to stress that these failure scenarios occur already in Raft clusters with five or even less instances, while the probability of a failure normally grows with the cluster size. For example, the studies of network-partitioning failures in Google's B4 network [23], CENIC network [24] and data center networks [25] show that they can occur once a week and may require minutes to hours to be repaired. Particularly interesting insights offer Alquraan et al. [26], showing that 21% of such failures in the transport network of the consensus clusters leave the cluster in a lasting erroneous state that persists even after the partition heals, and finally require intervention of the network administrator.

ONOS, with its "sharding" strategy, makes things even more problematic. Namely, every instance in ONOS is a Raft leader for a certain network state partition; thus, each and every node failure in ONOS will trigger an error-sensitive leader election procedure. Even worse, a failure of two out of three nodes that constitute a default Raft cluster means that a majority of instances in a cluster have failed, bringing that cluster to a state that requires human intervention.

SCL and Hyperflow tolerate failures much better, as failing nodes or the network do not render the whole controller unusable. In particular, failures in the transport network are not a problem for SCL, because it uses a gossip protocol to disseminate network events among the instances. Gossip protocols are normally robust to network disruptions [12]. This is less the case for HyperFlow, as it relies on a publish/subscribe system for that purpose: these are less robust to network failures. Failing nodes are not a problem either: SCL connects all switches to all instances, so failure of an instance, or of a modest fraction of these, is not an issue. In

[1]This coordination delay is the reason, why the current ONOS implementation maintains certain parts of the network state (also called data stores) as eventually consistent, even though they semantically require strong consistency [19].

turn, this strategy represents a severe management problem, as it requires a setup and maintenance of a fairly complex control network that is supposed to provide the said switch-controller connectivity.

Application Development. The existing SDN controllers (all, not only distributed) do not provide an API for exception handling capable of informing the application about the update installation outcome [27], [28]. It is then the application developer who has to address this e.g. by verifying if the new network state matches the requested change and bring the network to the previous state in case it does not. (Note that this again requires verification of the new state.) This complicates the life of a developer, as it becomes necessary to know controller internals [29], a fact that hardly helps a wider SDN adoption.

Loosely coupled distributed controllers complicate the matter even further. For example, when an SCL instance reads from its (locally maintained) network state, it may not be aware of the effects of possible more recent writes, just performed on another instance. Such stale reads can lead to conflicts that have been long known in the database management systems (DBMS) as lost update, inconsistent read and dirty read [16]. Interestingly, SCL tends to eliminate such problems by constraining the ways the network should be used or, again, by transferring the problem to the developer.

	Tightly coupled	**Loosely coupled**
System Performance	×	✓✓
System Administration	×	✓✓
Application Development	✓	×
SDN Model Richness	✓✓	×

TABLE I
THE COMPARISON BETWEEN THE STATE OF THE ART PROPOSALS OF THE DISTRIBUTED SDN CONTROLER

SDN Model Richness. Networks controlled by a loosely coupled distributed controller, SCL and HyperFlow in particular, are subject to a number of severe constraints in order to guarantee error-free operation. The first is the maximum acceptable level of dynamicity. For example, a HyperFlow cluster with ten instances limits the controlled network to send up to 1000 events per second. SCL, similarly, cannot handle high event rates from the network. To guard against them, SCL proposes a range of mechanisms that proactively set up paths in the network: TeXCP [30] and MATE [31] and methods for the switches to choose between these paths. The generality of these solutions is fairly questionable.

The second constraint is that SCL and HyperFlow support only deterministic SDN applications. Namely, all instances are presumed to process all network events equally and at the same time. This is to ensure that all instances converge to the same state over time. However, this calls for strictly idempotent network updates, or requires additional distributed machinery to check, if an update is being repeated or not.

In contrast, tightly coupled distributed controllers do not impose such constraints. They are designed to be able to converge whatever is the request arrival order and semantics.

Table I summarizes the above discussion.

III. OUR PROPOSAL: FITSDN

A. Design and Architecture

FitSDN is Flexible. The high-level FitSDN architecture in Figure 1 shows multiple controller instances, $C1$ to $C3$, in a network divided into non-strict subdomains. In a classical OpenFlow and e.g. in ONOS, the division must be strict, i.e. each switch must be controlled by exactly one controller instance (so-called master). FitSDN supports this mode for compatibility: if, e.g. due to connectivity or policy constraints, direct access to switches in some (sub)domains is not possible, FitSDN controller instances enable such access by featuring simple proxies, which implement a stripped-down westbound interface (WBI). The information, which proxy to use to reach a remote switch is available in the FitSDN mastership store of the requesting instance (see below). For example, switches $S21, S22, \ldots, S2k_2$ form a subdomain exclusively controlled by instance $C2$, so $C1$ would need to send requests to it to update e.g. $S21$.

In FitSDN per se, all controller instances remain equal at any moment of operation, and several instances can be active for one switch (cmp. the OFPCR_ROLE_EQUAL option in OpenFlow 1.5). Our design aims at increasing the degrees of freedom for the switches to connect to any number of controllers at will: on the one hand, in contrast to the classical OpenFlow SDN, in FitSDN all connected controller instances can have full control to the switch (e.g. $S1k_1$ is controlled by both $C1$ and $C2$). The mechanisms proposed in this paper resolve the related control conflicts automatically. On the other hand, there is no requirement to connect all switches to all controllers, like e.g. in SCL, to converge to the same network state. We use a different state distribution to achieve that. With this flexibilty, FitSDN improves the resilience and simplifies the administration compared to state of the art controllers: the addition of a new controller instance requires launching the instance and configuring some switches to use it, regardless of previous control relationships.

FitSDN is Integrated. FitSDN adopts the notion of data stores from ONOS and organizes the network state into a set of independent stores, among which mastership, network topology and resource stores are the most important. They are denoted by symbols M, T and R in Figure 1. In contrast to prior art, the resource store in FitSDN is globally unique, i.e. it is *not* replicated in the controller instances. To realize this, FitSDN integrates the resource store within the network elements and maintains it as a distributed transactional database.The resource store includes the content of the flow tables, meters, bands, etc. In short, it comprises any data that an application can possibly modify in a switch. This is the part of the network state that pertains to network programming as such and contains data that change most often.

Whereas it is essential to integrate the resource store, i.e. not replicate it, FitSDN does replicate the mastership and the topology stores at the controller instances. In FitSDN, both can operate under eventual consistency, while the mastership store in ONOS is strongly consistent. We believe this is not necessary: if a controller instance, due to the eventual

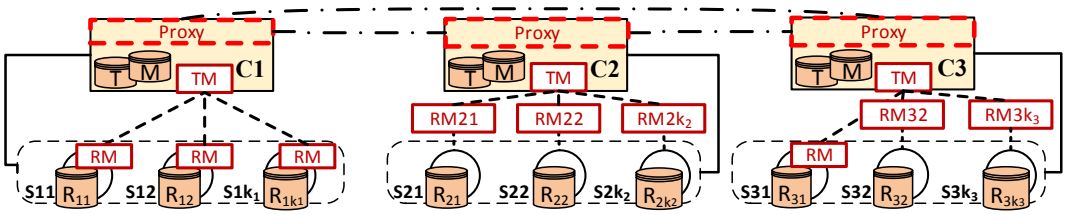

Fig. 1. The distributed transactional control plane of FitSDN.

consistency of its mastership store, were to see multiple other instances as possible masters of a switch, it could still relay a command over all of them. Only the instance that is the true master of the switch would relay the command to the switch, the others would discard it.

FitSDN is Transactional. FitSDN enforces transactional access to the resource store. A controller instance can *atomically* update a set of switches (irrespective of whether they fall into the same domain or not) and in *isolation* from other instances, which might be simultaneously updating them. To keep the state in the switches consistent in presence of concurrent updates, and to avoid problems such as lost update, dirty and inconsistent read [16], the total schedule of reads and writes over a switch must be serializable and recoverable. To achieve this, we adopt and adapt the prior work in the area of DBMS. To this end, FitSDN extends the controller with a transaction manager (to coordinate updates over multiple switches) and the switch with a resource manager (to turn the switch into a transaction-aware medium). These are depicted as boxes labeled TM, and RM respectively in Figure 1. The RM can run any concurrency control algorithm. As it is implemented on the switch, this in itself presents an interesting design question. In this paper, we opt for a rather simple locking mechanism, which can be easily implemented even in hardware switches. We show that in spite of this simplicity, it achieves good performance.

Notice that some RMs are depicted in Figure 1 outside the respective switches (e.g. RM21, RM32). This is to demonstrate that FitSDN is backward compatible, i.e. it can support hardware switches that do not have RMs. Thus, if it is not possible to run the RM in its managed switch, the RM can run as a switch image in the controller. Figure 1 also summarizes various interfaces that FitSDN requires. The dashed lines present the transactional TM-RM interface. The dotted lines present the RM-switch interface, for switches without an RM. The solid lines between the instances and the domains present a southbound interface (SBI) such as OpenFlow (to keep the figure simple, we show only one line per domain). The dash-dotted lines between instances present the WBI, for the instances to reach switches in remote domains. All these extensions are described in detail in §III-B.

B. FitSDN Transactional Architecture

Figure 2 shows FitSDN transactional architecture. The additions to the standard SDN models, shown in red, include the transaction manager (TM) that runs in each SDN controller instance, the resource manager (RM) that runs in each switch, the transactional SBI (T-SBI), an extension of the southbound interface for the TM-RM communications, and a transactional API (T-API), which is exposed to the applications. In more detail:

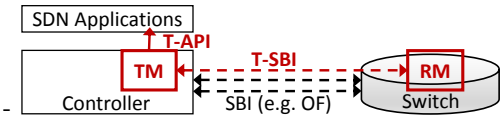

Fig. 2. FitSDN transactional architecture.

1) The TM is a module in the SDN controller instance that handles transactions: it acts as a bookkeeper, tracking their progress and communicating with the required RMs. It provides the API to the applications, to be used for processing of network updates that require transactional semantics. When installing an update across multiple switches, the TM acts as the atomic commit protocol (ACP) coordinator. The controller instances run independent TMs, which do not require any coordination, except that the IDs they assign to transactions should be globally unique (described shortly).
2) The RM controls the resource, i.e. the SDN switch, and transforms it into a transaction-aware medium with well-defined states that can be accessed only according to transactional semantics. The essential part of RM is a concurrency control mechanism (algorithm) that schedules concurrent read and write operations over the resource so as to achieve a desired level of their isolation. There are plenty of available mechanisms to select from (e.g. 2PL [16], ESGT [16], CORCO [32]), and finding the best one for a given setting is an interesting research question per se. In this paper, we select (implementation wise) the simplest one that just locks the resource so as to enforce sequential access to it. RM also acts as a participant of the ACP, formulating its replies to the ACP coordinator based on the state of its (algorithm to control the) resource.
3) The T-SBI is an extension of the SDN southbound interface that enables the TM-RM message exchange. This paper assumes that the TMs use ACP, such as 2PC, to coordinate the RMs.
4) There are three T-API calls relevant for the application:
   - `startTransaction()`, to signal the beginning of a transaction.
   - `read(...)`, to retrieve parts of the network state.
   - `transactionalUpdate(...)`, to update the network, i.e. bring a set of switches to the desired state (both are the call parameters).

C. FitSDN Operation

While the general and the transactional FitSDN architecture, described so far, apply to any FitSDN instantiation, the details we are about to provide now pertain to one specific instantiation, namely the one that uses locking for concurrency control in the RM.

FitSDN locking mechanism adheres to a rather simple principle: when a request arrives at a switch, its RM tries to grant the lock over the switch to the requesting transaction. As simple as it sounds, it still has a number of implications, in particular for understanding the notion of transaction and choice of identifiers that are exchanged between the TM and RM, as well as stored at RM.

Transactions in FitSDN: FitSDN views each transaction as a two phase processing: in the first phase, an application reads from the network state. These reads can be interleaved or followed by application-specific processing (e.g. path calculation). In the second phase, the application updates the network state, i.e. writes to the appropriate stores, most commonly the resource store. FitSDN enforces atomicity of the second phase only. So either all writes succeed or none. In contrast, reads are not atomic, neither on their own, nor together with writes.

The reason for this decision lies in the use of locking as concurrency control in the RM. Precisely, duration of locks affect the performance, so FitSDN tries to reduce lock times as much as possible. Potential conflicts that this design may cause, e.g. a transaction reading a switch that another transaction changes a moment later, are solved by holding in the switch the ID of the transaction that updated it last. More details about this follow shortly.

IDs in FitSDN: There are two types of IDs that are critical in FitSDN, the global transaction ID (GT ID) and the last write global transaction ID (LW GT ID). The TM stamps all commands, reads and writes, within one transaction with the global transaction ID (GT ID) before sending them to the RMs. The GT ID is globally unique at the network level. The RM reads the GT ID from the received commands and always includes it in its response to the TM. This way, the TM can identify the transaction that the response belongs to. FitSDN follows a simple way to achieve GT ID uniqueness: it assigns IDs to TMs and enforces their uniqueness. Each TM, in turn, assigns an ID to each transaction it handles, which is unique within its own scope and appends it to its own ID. The only purpose of the `startTransaction()` call is in fact to generate the transaction ID.

When the RM receives a write request, it stores the GT ID from it locally (but persistently). The RM thus identifies the transaction that updated the switch, i.e. the resource store, last. We call this ID last write GT ID (LW GT ID). It essentially identifies the current resource store state. Whenever replying to a read, the RM sends back the LW GT ID, along with the GT ID retrieved from the read. The TM uses the LW GT ID to associate the writes from the second phase of the transaction with the correct network state reads. In addition to the GT ID, the TM adds the LW GT ID to each write. The RM rejects writes that carry obsolete LW GT IDs.

A typical transaction cycle: To put all this in use, we now go step by step through a typical transaction processing cycle, cf. Figure 3. For the sake of simplicity we assume that the transaction updates switches directly connected to a controller instance.

1) The application receives a network event notification that requires a reliable response and calls `startTransaction()` to receive a GT ID.
2) The application calls a `read(...)` on the TM to retrieve the needed data. The example from Figure 3 assumes that this data is bandwidth allocated over a link. The application also passes the GT ID as an additional argument.
3) The TM parses the call and translates it into one or more OpenFlow read commands, each stamped with the GT ID (see §III-E for details). For each read, i.e. each switch, the TM inserts a new entry in its internal table used to track the progress of the transaction. The table stores the information about the exchanged commands in the transaction, such as their type, target SW ID, GT ID and LW GT ID. The LW GT ID is set to null at this moment, it will be updated later.
4) The RM receives a read, reads the resource and then generates a response stamped with the GT ID from the read and the LW GT ID, which was stored internally.
5) The TM receives the response from the RM, parses the GT ID and uses it to find the correct entry in its internal table. Then it retrieves the LW GT ID from the response and updates the entry. It finally parses the response and delivers the requested data to the application.

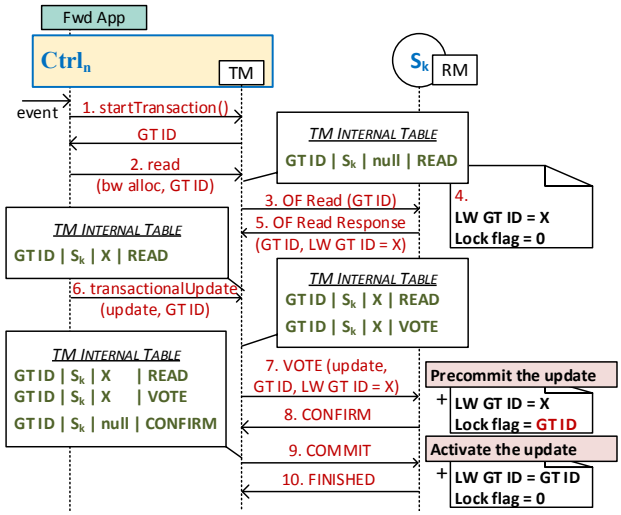

Fig. 3. A typical transaction cycle.

6) The application formulates its network update and calls `transactionalUpdate(...)` on the TM to execute it. It supplies its GT ID too.
7) The TM reads the GT ID from the update and sends a `Vote` to the involved RMs as the first message of the 2PC exchange. The `Vote` includes both the GT ID and the LW GT ID of the target switch (as retrieved from the TM internal table), as well as the actual network update.

8) On the reception of `Vote`, each RM locally locks the switch and sets the lock flag of the switch to the incoming GT ID. The switch thus becomes unavailable for other transactions. The RM then compares the incoming LW GT ID with the stored one. If the two differ, the switch was meanwhile updated and the received update is based on an obsolete network state, hence the RM declares it invalid and rejects it. Otherwise, the RM applies the changes to the staging area (this does not affect the data packets traversing the switch). If this is successful, the RM sends a `Confirm` message to the TM (without activating the change). Otherwise, it rejects the update, sends `Reject` to the TM and unlocks the resource.
9) If the TM receives confirmations from all the RMs of the current update, it starts the commit phase by sending a `Commit` message to them. If at least one RM rejected the request or did not reply within the expected time, the TM aborts the update by sending `Rollback` to the RMs that replied with `Confirm`.
10) On `Commit`, the RMs activate their staging areas and update the LW GT ID of the switch, whereas on `Rollback`, they discard them. In both cases, they send `Finished` back to the TM and release the locks.

D. Deadlock Handling

Deadlocks can happen in FitSDN only when processes involved in the 2PC exchange fail or cannot communicate. The scenario in which two or more transactions request and hold mutually blocking locks is not a problem, as their TMs will detect this by receiving at least one `Reject`, and release the locks.

If a 2PC coordinator fails just before deciding on the outcome of a transaction, the participants may stay blocked forever. Similarly, if a participant fails just before sending its vote response to the coordinator, the coordinator cannot decide whether to commit or rollback the transaction.

To resolve such deadlocks, FitSDN uses timeouts in TMs and in RMs. A TM starts a timer for each 2PC primitive it sends. If the timer expires before the response arrives, the TM aborts the entire transaction. Similarly, an RM starts a timer, whenever it locks its resource, and discards the pending update, when it expires.

E. Implementation

Our proof-of-concept implementation is based on OpenFlow, Floodlight [2] and OpenVSwitch [3]. We extend OpenFlow (or, it is more appropriate to say we use it in a particular way) to implement our T-SBI, Floodlight to include our TM and OpenVSwitch (OVS) to include our RM. The TM is a new core service in the Floodlight controller, while, as just hinted, TM and RM communicate in our implementation via unchanged OpenFlow 1.4 or higher. To include ACP or transaction-related messaging, we used the existing OpenFlow messages, just modifying how the endpoints interpret them. For example, we implement `Vote` as an OpenFlow bundle [3] with a transaction second phase init message at the beginning (i.e. `OFPT_FLOW_MOD` with a specific command and tableId combination), followed by the messages from the update. In `Reject`, `Commit`, `Rollback` and `Finished`, we set the `xid` field to GT ID. In `Vote` and `Confirm`, we set the `bundle_id` to GT ID. We set the `pad` field of `Vote` to LW GT ID.

Similar holds for the optional WBI, which also uses unchanged OpenFlow. To enable message relaying, we run a proxy (on an agreed port) in each instance. The proxy parses messages received from other instances and delivers them to the target switch using the existing SBI.

TM in Floodlight implements the `IOFMessageListener` interface, to be notified when the controller receives an OpenFlow message. Specifically, it registers to receive network state read responses and the TSBI messages. SDN apps must register to use the TM. To register, the app creates the instance of the TM in the `initApp()` method. Strictly speaking, this renders the transactional updates unusable to old apps, but this is just our current implementation limitation.

We extend OVS to include necessary RM-related data structures such as lock flag, LW GT ID, etc. by changing OVS `ofproto` library. Similar extensions exist for the staging area updates and the processing of 2PC messages.

F. Discussion

We now briefly discuss, how FitSDN responds to the requirements set in the introduction. We will spend the whole next section to evaluate FitSDN performance, therefore the focus here is on the last three.

System Administration. Failures of FitSDN instances or of the transport network interconnecting them cannot render the FitSDN cluster nonoperational. State updates of the eventually consistent data stores, e.g. mastership and topology stores, are disseminated using gossip protocol that is highly robust to failures. FitSDN instance stays operational in its own subdomain, regardless of its connectivity to other instances in the cluster.

Application Development. In FitSDN, app developer does not have to consider artefacts of the controller related to the distribution. These are delegated to the TM. Consider, e.g. an application installing a path with bandwidth guarantees. In FitSDN that should be done as follows (the code assumes the try-catch block like in Java, and that the developer's choice is to have specialized exception handling for updates only):

```
try {
  availableBW = tm.read(Resource.BW,path,GT_ID);
  if(availableBW >= requestedBW)
      try {
           tm.transactionalUpdate(path,
              requestedBW,GT_ID);
      } catch (UpdateExceptionType ue) {
           process(ue); }
} catch (Exception e) {
  process(e); }
```

SDN Model Richness. FitSDN does not constrain the SDN model in any way (e.g. by requesting only idempotent network updates), neither does it eliminate important use cases by

permitting only a small range of input parameters (e.g. level of dynamicity). The only constraint imposed by FitSDN is its extension of the switch (RM). Given the relative simplicity of the RM and the benefits of this design, we believe that it is a good design choice. Besides, the development of OpenFlow testifies of similiar trends in the past (e.g. bundles, time synchronization, etc).

Note that FitSDN, by using switches as storage media, does not just shift failure points from the controller to the switch. While FitSDN regulates, how data on the switch can be accessed, any switch only holds the flow table data relevant to the flows traversing it, which is an operational requirement in any SDN. As this data is only pertinent to that switch, a failure of the switch is also equivalent to current data plane failures and must result in flow rerouting.

IV. Evaluation

We now evaluate FitSDN performances and compare them to ONOS and SCL. Throughout the evaluation, we use a simple SDN application for constrained flow path allocation, e.g. 1 Mbps for a flow. A series of requests to set up a flow between two switches arrive in the network as a Poisson process with parameter λ $1/sec$. The requests are processed by the controller instance of the originating switch (exactly one instance controls each switch). If a request is initially rejected in FitSDN, e.g. a switch is already locked, we use a backoff mechanism to reschedule it after a random delay, drawn from the exponential distribution with mean 0.01. We do not limit the number of processing attempts of a request, i.e. the controller instance reschedules its processing until it succeeds. We use the following three performance metrics:

- Request processing time: The time from the moment when the controller instance receives a request until the moment when it pushes the update to the network.
- Path setup delay: The time from the moment when the controller instance receives a request until the moment when the last concerned switch successfully installs the flow.
- End-to-end delay: The time from the moment when a source sends a data packet up to the moment when the destination receives it.

In §IV-A1 we report the request processing time of a single Floodlight, distributed ONOS and SCL controllers, and FitSDN. We use *simulations* for these tests rather than the existing implementations, as they are much easier to scale up to larger cluster sizes. We increase the request arrival rate and whenever it reaches the processing limit of the controller (new requests start accumulating in the controller's queues and the queues are continuously increasing) we add a new computing resource to run in parallel to the existing one(s), i.e. a new thread in the Floodlight or a new controller instance in ONOS, SCL and FitSDN. To simplify our simulations, we assume that each instance of the distributed controller runs as a single-threaded server. This is a fair assumption, as we want to evaluate the effect of distribution on the observed SDN controllers. Ideally, we would like to observe that adding computing resources to the network can decrease the request processing time on average.

Finally, in §IV-A2 we report on the path setup delay and end-to-end delay of the *implementations* of ONOS, SCL and FitSDN for a small cluster size.

A. Evaluation Results

1) Request Processing Time: Figure 4 shows the request processing time when the network is controlled by centralized Floodlight and distributed ONOS, SCL and FitSDN controllers (referred to as "clusters"), and the request arrival rate varies from 200 to 3000 sec^{-1}. The controlled network is random with the total of 72 switches and 180 links. We run 100 independent experiments for each of the 4 scenarios and measure in each the processing time of 1000 flow setup requests and calculate the 95% percentile thereof. This way, we end up with 100 values for each scenario.

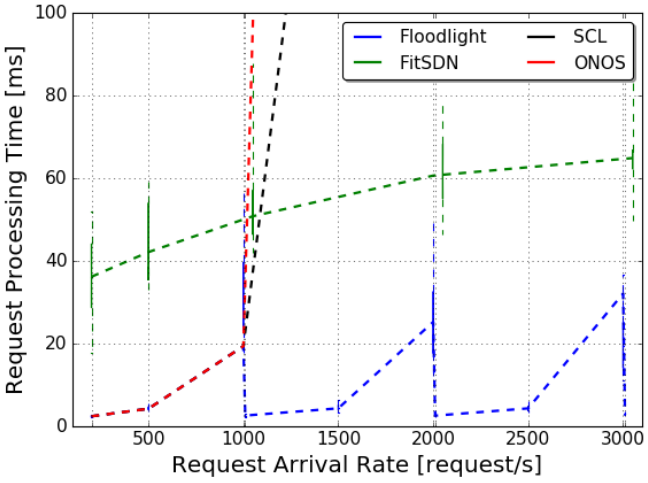

Fig. 4. The 95th quantile of the measured sample of request processing times.

We observe from the results that introducing new threads in Floodlight effectively reduces the request processing time. Each sharp drop shown in Figure 4 for Floodlight represents the moment when the request arrival rate reached the system capacity and a new thread was added. However, the same does not hold for the SCL cluster. The reason for this is that adding a new computing resource to SCL does not split the load; to reach the same state, each instance has to process each request from the network. Also, adding new instances in ONOS does not improve the request processing time, as the benefits are offset by an increasing delay of the consensus protocol itself.

Unlike that, adding computing resources to the FitSDN cluster helps, as the the request processing time remains low. We observe a mild, sub-linear increase of the delay processing time, indicating that FitSDN scales well with the network load. Figure 5 plots the number of used FitSDN instances in each observed point from Figure 4. We see that the FitSDN cluster size scales linearly with the input load.

In an attempt to precisely understand how a new instance in FitSDN reduces the request processing time, we fix the request arrival rate at 2000 requests/sec and measure the processing time for different cluster sizes. (To support this rate, the cluster must have at least 13 instances.) Figure 6 shows that two new

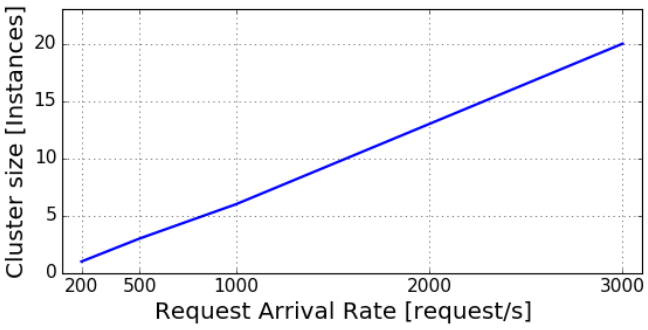

Fig. 5. Minimal size of FitSDN cluster to process the given arrival rate.

instances in the FitSDN cluster can reduce the processing time on average to around 50% of the initial value, measured for the minimal cluster size. The rate of reduction gets smaller as the cluster grows.

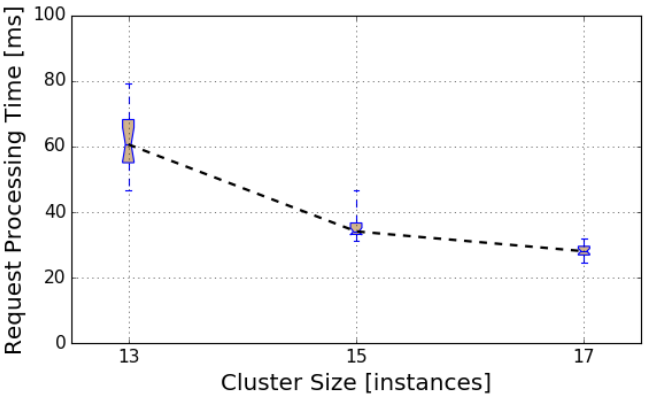

Fig. 6. Request processing time for λ = 2000 requests/sec in FitSDN cluster of size 13, 15 and 17.

2) End-2-end Delay and Path Setup Delay of the FitSDN, SCL and ONOS Implementations: We now compare FitSDN to SCL and ONOS in a small and controlled, but realistic deployment. We take ONOS v1.11.2, our own mock implementation of SCL and our FitSDN implementation and, for each, create a cluster of three controllers, each running on a separate Huawei RH2288H server with 48 Intel CPUs and 1.5TB of RAM. We run our dataplane in Mininet. Our test network has a linear topology of 9 switches, each running OpenVSwitch, v2.8.0. Each controller controls three switches. The results, averages of 10 repetitions, are shown in Table II.

As expected, SCL has the smallest end-to-end delay along the 9-hop path, as its instances do not coordinate their updates. FitSDN is slightly worse, while staying within the general SDN model, unlike SCL. ONOS, which also supports general programming, performs substantially worse, around 4 times worse in the median. This is due to ONOS inability to coordinate the installation of the flow rules, i.e. uncoordinated sequence of PACKET_OUTs and INs to and from the switches along the path, as our Wireshark traces confirm. The path setup delay has no PACKET_OUTs and thus enables us to isolate the effect of consensus in ONOS.

System	E2E Delay			Path Setup Delay		
	Median	Min	Max	Median	Min	Max
ONOS	48.3ms	46ms	66ms	29ms	27ms	32ms
SCL	5.2ms	6.6ms	8.1ms	3.5ms	5ms	6.5ms
FitSDN	11.2ms	12.5ms	15ms	9ms	10ms	12ms

TABLE II
END-TO-END AND PATH SETUP DELAY ON AN 9-HOP LINEAR TOPOLOGY CONTROLLED BY ONOS, SCL AND FITSDN

V. Related Work

Section II discussed related SDN controllers at length. This section therefore deals only with SDN papers that have transactional aspects in their focus.

Schiff et al. [33] proposed a synchronization framework that enforces atomicity and isolation of the distributed single-switch updates. It stamps the switch with the installed policy identifier and adds primitives for reading and updating the stamp. To update the configuration of the switch, an SDN application must know its current stamp value — this guarantees that the switch was not updated meanwhile by another controller instance. However, it remains unclear, how to extend this to a network-wide update. Canini et al. [34] addressed the latter and proposed transactional middleware for semantic composition of distributed network-wide updates. It intercepts the updates, serializes them and modifies when necessary. As the middleware processes the updates sequentially, it must be single-threaded, which impedes the benefits of the control plane distribution. In our previous work [35], [36] we proposed a new SDN architecture that achieves transactional (i.e. atomic and isolated) network-wide updates using the switch as a transaction-aware medium. As such, it does not face the aforementioned problem of the middleware-based solution. However, the proposal assumes connectivity of each switch with each instance of the distributed controller and therefore cannot be used for multi-domain SDN deployments, where each instance controls only a subset of the switches. Besides, it disregards "read before write" and limits the range of supported transactions to write updates only.

To summarize, in contrast to our paper, none of these papers offers a complete, working SDN architecture. They are rather a few scattered elements of a big puzzle, which we brought together, added missing pieces and designed, implemented and tested a working and scalable system.

VI. Conclusion

In this paper we present FitSDN, a novel distributed SDN design that is scalable, easy to manage and program. Its key design idea is synchronization of the controller instances over the switches, thus reducing data replication and eliminating the need for heavy consensus protocols between the instances. To achieve this, FitSDN uses the network as a storage with transactional semantics. With this, FitSDN also enables a reliable network update API that lets the SDN application developers focus on the application logic.

Our performance evaluation demonstrates that FitSDN, unlike current state of the art distributed controllers, scales well with network sizes and input load owing to its seamless

integration of additional compute resources, i.e paying no price of distribution as such.

In our future work, we plan to evalute alternative resource manager algorithms for concurrency control in the switches.

VII. Availability

FitSDN is free software, available to download from
1) https://github.com/fitSDN/RepoMain.git.

References

[1] POX, "POX SDN Controller." [Online]. Available: https://github.com/noxrepo/pox

[2] Floodlight, "Floodlight OpenFlow Controller – Project Floodlight." [Online]. Available: http://www.projectfloodlight.org/floodlight/

[3] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch," in *USENIX NSDI'25*.

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, Mar. 2008.

[5] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "Onos: Towards an open, distributed sdn os," in *HotSDN '14*. New York, NY, USA: ACM, 2014.

[6] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," in *USENIX OSDI'10*, 2010.

[7] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX ATC'14*, 2014.

[8] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, December 2001.

[9] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *INM/WREN'10*, 2010.

[10] A. Panda, W. Zheng, X. Hu, A. Krishnamurthy, and S. Shenker, "Scl: Simplifying distributed sdn control planes," in *USENIX NSDI*, 2017.

[11] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris, "Flexible, wide-area storage for distributed systems with wheelfs," in *USENIX NSDI'09*, 2009.

[12] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *ACM PODC'87*. ACM, 1987.

[13] E. Sakic and W. Kellerer, "Response time and availability study of raft consensus in distributed sdn control plane," *IEEE Transactions on Network and Service Management*, vol. 15, no. 1, March 2018.

[14] V. Arora, T. Mittal, D. Agrawal, A. El Abbadi, X. Xue, Z. Zhiyanan, and Z. Zhujianfeng, "Leader or majority: Why have one when you can have both? improving read scalability in raft-like consensus protocols," in *Proceedings of the 9th USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'17. USENIX Association, 2017.

[15] S. Rizvi, B. Wong, and S. Keshav, "Canopus: A scalable and massively parallel consensus protocol," in *ACM CoNEXT'17*, 2017.

[16] G. Weikum and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[17] "Update in a multipartitioned raft implementation (2-phase commit)," https://groups.google.com/a/onosproject.org/d/msg/onos-dev/bkgXaYUBoCE/Jfm2a6NYAgAJ, Jan 2016.

[18] "Transaction management in onos, mar 2018," https://groups.google.com/a/onosproject.org/d/msg/onos-dev/cg5NyJsyxUA/SrsUvraIBgAJ.

[19] "Flow subsystem burst throughput," https://wiki.onosproject.org/display/ONOS/1.14%3A+Experiment+F+-+Flow+Subsystem+Burst+Throughput, Aug 2018.

[20] H. Howard and J. Crowcroft, "Coracle: Evaluating consensus at the internet edge," *SIGCOMM Comput. Comm. Rev.*, vol. 45, no. 4, 2015.

[21] D. Ongaro, "Bug in single-server membership changes," https://wiki.onosproject.org/pages/viewpage.action?pageId=12419345, Tech. Rep., 2015, raft-dev post 07/09/2015.

[22] J. Fennema, "Bug in raft paper and proposed solution," https://groups.google.com/forum/#!topic/raft-dev/JEtBYaPpHXo, Tech. Rep., 2017, raft-dev post 25/10/2017.

[23] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or die: High-availability design principles drawn from googles network infrastructure," in *SIGCOMM '16*. ACM, 2016.

[24] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage, "California fault lines: Understanding the causes and impact of network failures," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, Aug. 2010.

[25] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, Aug. 2011.

[26] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An analysis of network-partitioning failures in cloud systems," in *OSDI'18*, 2018.

[27] M. Kuzniar, P. Peresini, and D. Kostić, "Providing reliable fib update acknowledgments in sdn," in *ACM CoNEXT'14*, 2014.

[28] P. Peresini, M. Kuzniar, and D. Kostic, "Openflow needs you! a call for a discussion about a cleaner openflow api," in *2013 Second European Workshop on Software Defined Networks*, Oct 2013.

[29] "Stuck in pending add state, apr 2017," https://wiki.onosproject.org/display/ONOS/FAQ#FAQ-WhyareflowsstuckinPENDING_ADDstate?

[30] S. Kandula, D. Katabi, B. Davie, and A. Charny, "Walking the tightrope: Responsive yet stable traffic engineering," in *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '05. ACM, 2005.

[31] A. Elwalid, C. Jin, S. Low, and I. Widjaja, "Mate: Mpls adaptive traffic engineering," in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, vol. 3, April 2001.

[32] Y. Raz, "The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource mangers using atomic commitment," in *VLDB '92*, San Francisco, CA, USA, 1992.

[33] L. Schiff, S. Schmid, and P. Kuznetsov, "In-band synchronization for distributed sdn control planes," *SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 1, Jan. 2016.

[34] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "Software transactional networking: Concurrent and consistent policy composition," in *ACM HotSDN '13*, New York, NY, USA, 2013.

[35] M. Curic, Z. Despotovic, A. Hecker, and G. Carle, "Transactional network updates in sdn," in *2018 European Conference on Networks and Communications (EuCNC)*, June 2018, pp. 203–208.

[36] M. Curic, G. Carle, Z. Despotovic, R. Khalili, and A. Hecker, "Sdn on acids," in *Proceedings of the 2Nd Workshop on Cloud-Assisted Networking*, ser. CAN '17. New York, NY, USA: ACM, 2017.