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.

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.

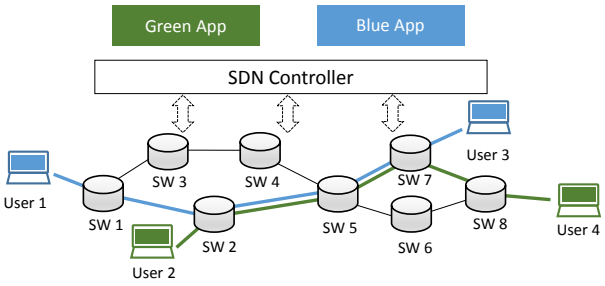

Fig. 1. Example network to illustrate non-atomic and uncoordinated updates.

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.

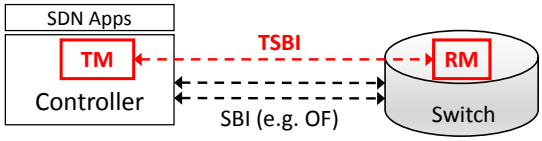

Fig. 2. Transactional SDN architecture.

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.

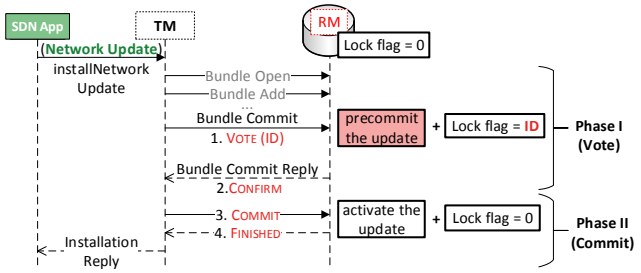

Fig. 3. Message flow of a typical transactional network update.

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 vote-locks of each individual switch. So when an application sends vote-locks 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-lock 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.

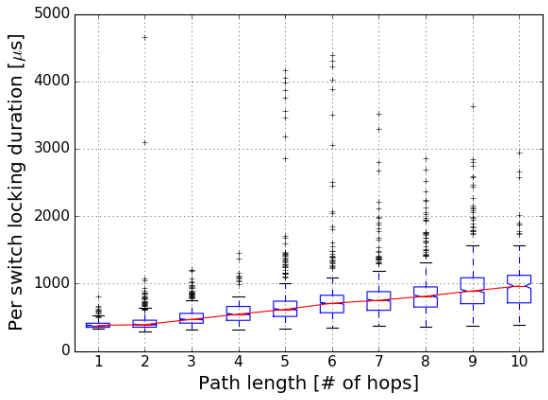

Fig. 4. CDFs of vote-lock duration for groups of up to 10 switches.

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 $1000s^{-1}$.

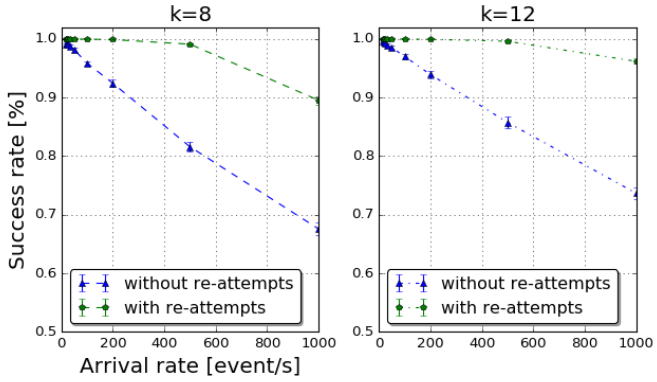

Fig. 5. Success rate.

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 of around 75% of the updates is below 6ms and that the 95-

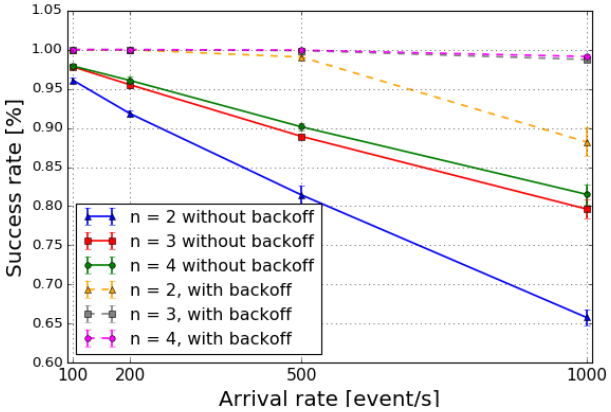

Fig. 6. Success rate.

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.

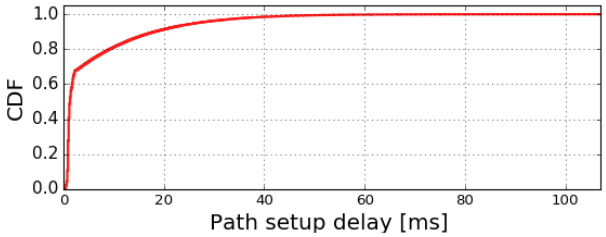

Fig. 7. CDF of the path installation delay – network with k=12, 3 installation re-attempts and arrival rate of 1000 requests/sec.

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 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 plane to investigate if our proposal can benefit from "power of two choices" concept, where network update consists of two alternatives, affecting completely or partially orthogonal resources, and successful installation means that only one of them does not conflict with another concurrent installation.

Acknowledgement

This work has received funding from the European Unions 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.