TUS: A Transactional Update Service for SDN Applications

Qi-An Fu and Wenfei Wu
Tsinghua University

ABSTRACT

In this paper, we design a network update service for SDN applications to transactionally update the network. We name our system TUS, which is a layer sitting between the SDN controller and SDN applications. SDN applications can call TUS's interfaces to achieve atomicity, consistency, isolation, and durability guarantee for network updates, which eases the application's programming. TUS provides APIs for consistent update, uses logs for atomic update, applies optimistic concurrency control (OCC) for inter-update isolation, and checks each asynchronous rule update for durability. These design choices are decided by network unique characteristics and challenges — OCC can isolate network updates from volatile network states and checking asynchronous rule update is also used in implementing consistency guarantee in multi-phase network update. We prototype TUS, validate it consistency guarantee and failure recovery effectiveness, and also measure the overhead introduced by TUS logs.

1 INTRODUCTION

Under the paradigm of Software-Defined Networking (SDN), a network can be updated flexibly to satisfy dynamic network requirements and workloads, which usually involves multiple devices (e.g., setup a path). A network update should satisfy several properties to guarantee the network in a correct state. For example, the update procedure should be consistent (e.g., loss-freedom [25], congestion-freedom [10, 18]); concurrent updates should be logically isolated or conflict-resolved [2, 27]; and the SDN framework should make an update event to be reliable/durable in case of system crash [7]. Combining individual properties, an SDN update should be atomic, consistent, isolated, and durable (ACID), and we call a network update to be *transactional* if it satisfies ACID properties.

APSys 2018, Jeju Island, South Korea
 978-1-4503-6006-7/18/08…$15.00
DOI: 10.1145/3265723.3265728

In this paper, our goal is to abstract the applications' ACID requirements as a separate *transactional update service*, and build an layer sitting between the SDN controller and SDN applications to provide this service. We name our system TUS. TUS takes SDN updates from multiple applications, devises the ACID execution of simultaneous updates, and finally commits the updates to the network.

Referring to the individual existing network solutions above [7, 27] as well as the experience in designing database transactions, ACID can be achieved by separate mechanisms — the consistent update algorithms above can be integrated into SDN applications and submitted to the SDN controller; the SDN controller should have concurrency control to achieve isolation; a log module can be used to replay updates during failure recovery for atomicity; and persistent data storage can guarantee durability. However, network updates and network states have their own unique characteristics differently from database transactions and data records, which requires TUS to be wisely crafted.

First, some network states are volatile, which are hardly isolated from network updates. The volatile nature comes from the data plane events (e.g., flow/byte counting) instead of control plane commands, which is a significant difference from persistent storage system underlying a database. If an update is computed based on a volatile state, when the update is committed, the volatile state may invalidate the update if its value changes. TUS derives a lazy-validation mechanism from the optimistic concurrency control [16]: it performs each update on a local copy, validates whether network changes during the update computation; if there is no changes, it commits the local copy to the actual network if no, and otherwise cancels the update and returns failure.

Second, (some) consistent network updates are required to be executed in multiple phases, and thus, TUS should support this new semantic in interfaces, execution, and failure recovery. For example, two-phase update [25] is proposed to avoid transient packet loss when setting up a (new) path for a flow [25]; multiple-phase update is proposed to avoid transient link congestion when lively migrating multiple flows' paths [10, 18]. All updates from the SDN controller to switches are asynchronous, which adds special difficulty to confirm that all updates within a phase are complete. TUS adds a new interface `tx.barrier()` for applications to divide an update in phases; a barrier is implemented by stop

and wait for active confirmation (read and check what is written) from device updated at the barrier; and the failure recovery module also incorporates the barrier operation by recording it in logs and submitting it to execution engine.

Combining the classical database design for ACID and the two customizations for network updates, we propose TUS with the following design and features. (1) The northbound APIs to SDN applications can express start and commit of an update as well as the read/write and phase operations with in an update (for *consistency*). (2) TUS uses a log module to record network update events, and it can replay/rollback updates during a failure recovery (*atomicity*). (3) To perform updates on volatile network states, TUS adopts Optimistic Concurrency Control (OCC) [16] instead of locks, which is the only feasible solution for volatile data update (*isolation*). (4) Each individual rule update to a switch is confirmed by reading and checking the rule again to guarantee that the asynchronous update actually completes (durability).

We implement TUS on a Python based SDN controller Ryu [1]. Our evaluation demonstrates the feasibility of TUS: TUS can implement a two-phase update easily and correctly with few packet loss; TUS guarantees the right behavior during the failure recovery test; and also TUS only introduces acceptable overhead to the SDN control plane.

In this paper, we make the following contributions.

- We design a holistic service for SDN updates name TUS, which guarantees atomicity, consistency, isolation, and durability.
- For consistency guarantee, we enrich the semantics of update service and its corresponding APIs, which especially supports multi-phase network updates.
- For isolation guarantee, after analyzing the read/write and (non-)volatile nature of network states, we adopt optimistic concurrency control (OCC) as the design choice, which makes concurrency control of network update possible and efficient.
- We prototype TUS, and preliminary evaluation shows its feasibility in terms of functionality and performance.

2 BACKGROUND

2.1 Network Update in SDN

SDN Architecture. A typical SDN controller sits between SDN applications and network switch fabrics. It maintains a *network information base (NIB)*, which is an overview of the whole network. NIB maintains the following information: the network topology (switches and their connections) and individual switch states (flow tables, and per-rule statistics) (e.g., Onix [15]). The SDN controller communicates with switches by OpenFlow protocol [21], where it can send commands to update switch rules and also pull switch states (e.g., statistics, rules). Although it takes a few communication rounds for the controller to sync up the switch states and the NIB (eventual consistency), we assume the NIB stands for the whole network states in this paper.

Multiple SDN applications run as plug-in modules on top of the SDN platform: they read network states from NIB, make their own network update decisions (independently), and submit their updates to the NIB and the underlying network. We illustrate a few examples of network updates and analysis their requirements.

Example 1: Path Initialization. During a path initialization, a controller application aims to set up a path for a flow from the source to the destination via the path <A, B, C, D> (Figure 1).

ACID should be considered for the update: the rules to set up the path should guarantee no loops and no black holes are introduced (consistency); if the control plane crashes during the update, the recovery process should guarantee the all or none of the rules of the path are set up (atomicity); the path should take effect after set up (durability).

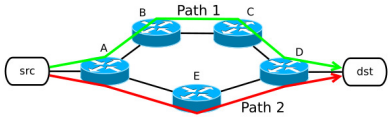


Figure 1: An example of path setup and migration

Example 2: Online Path Migration. Assume multiple flows are traversing a path <A, B, C, D>, and the network controller would move several flows of them to the new path <A, E, D> (Figure 1). To migrate a specific flow f, the 2-phrase update would modify the flow rules in E (adding a rule of f), A (modifying the rule of f to D), B and C (deleting the rule of f) sequentially [25].

In addition to the considerations in Example 1, this case has more requirements: since there are running flows, transient loss during update should be avoided (consistency); concurrent updates should not conflict each others (isolation, e.g., two updates reserve bandwidth on the same link or one update deletes another's rules).

Example 3: Path Backup. Assume a controller would set up a backup path for an existing path in case of path failure, it would read the workload on existing paths, and choose one with the least load and set up backup rules on it. In this process, the controller reads the statistics on existing paths, and computes a path with the least work load, and writes rules into switches on the path.

In this case, the isolation property is more difficult to guarantee. Because the statistics on the path is volatile — they are updated by data plane packet processing, and the computation in the control plane may be based on a stale value. Thus, there should be a mechanism to validate the isolated execution of this update.[1]

[1]It also depends on the application's logic whether the stale value of a state matters, thus, the application should be given flexibility to decide whether to enable the isolation validation.

2.2 ACID Requirements for SDN Update

We propose to abstract the ACID requirements from various SDN updates, and *wrap up and provide the mechanisms for ACID as a service*, thus, the trouble of integrating ACID requirements in SDN applications can be saved. We name this service TUS. (1) TUS takes updates from applications as input; it is the applications' responsibility to compute a consistent update but TUS should provide APIs for the applications to express the semantics of a consistent update. (2) TUS guarantees the execution of an update to be atomic, isolated, and durable. We elaborate each properties as follows.

Consistency. Before/after/during a network update, the network should not violate certain properties (e.g., loss-freedom, congestion-freedom). In the SDN architecture, the update is computed by the SDN application and submitted to the SDN controller. TUS should provide APIs to express the start/commit of an update and read/write/phase operations with in an update.

Isolation. (1) There may be simultaneous updates from different SDN applications. While executed simultaneously, the final results should be equivalent to that they are executed sequentially in a serial order. Thus, concurrency control between updates are needed. (2) A network update may also be simultaneous with data plane updates, in which case data plane events cannot be stalled to wait for the control plane updates. Because such states (e.g., flow statistics) flow from the switch to the NIB and then to the SDN application; in this case, there should be a rollback mechanism for the control plane to cancel updates if its volatile states vary.

Atomicity. An update involves multiple rules in multiple switches, and they need to be executed in an all-or-none manner. Otherwise, a partially executed update would leave the network with errors. For example, if a half of a flow's path is set up, there would be black holes or unused rule in the switches.

Durability. Once an update is committed to the network, it should be always functional in its lifetime, being resistant to failure and crashes in SDN controller and applications. Like a persistent storage, switch rules can be considered as durable in switches' forwarding tables, but TUS should guarantee the individual switch update is committed to the forwarding table.

2.3 Challenges and Design Choices

Strawman Solution. We insert an enhancement layer between SDN applications and the SDN controller. The enhancement layer has northbound APIs to accept network updates from applications, devises and executes the update to the network. Such an SDN transactional update service layer could use a log module to record each update so as to guarantee *atomicity*; it has interfaces to support the semantics of *consistent* update; it can adopt concurrency control (e.g., lock) to achieve *isolation* between updates; since an switch update is asynchronous, TUS can read and check what is written to guarantee the update is *durable*. But when combing these designs into one framework, the following two network specific challenges should be considered.

Challenge 1: Isolating network updates from data plane events is difficult. When a network update is in progress, it is possible that network states in the data plane changes (e.g., statistics for packet processing), causing simultaneous network updates based on them to be invalid. In database systems, conflict resolution is usually performed by using locks to exclude simultaneous operations (read/write) on the same object. However, this is not preferred in networks, because if a state change is driven by data plane events, it flows from the data plane to the control plane, and it is hard to add a lock from the control plane to exclude data plane events. In existing network operating systems [4, 15], the designers usually propose the SDN applications would handle the conflict by themselves.

Solution. We categorize network states into two kinds — *persistent states and volatile states*. Persistent states are read/written by the control plane (e.g., flow rules); and volatile states are updated by data plane events and is read only to the control plane (e.g., flow statistics). Since volatile states cannot be modified by the controller (read only), we adopt a lazy validation approach: after the application completes a network update, it would re-check whether the related states are changed during the update, if no, it would commit the update, and otherwise cancel it.

Table 1: States and Concurrency Control

States	Lock	OCC
Persistent States	Yes	Yes
Read-only Volatile States	No	Yes
Read-Write Volatile States	No	No

Table 1 shows the categorization of states and possible ways for concurrency control (CC), where "yes" means the CC method can handle the category of states in its row, and "no" means it cannot do so. For persistent states, both lock and optimistic concurrency control (OCC, i.e., update to a new copy, validate the copy, and merge with the original data copy) [16] can achieve correct concurrency control; for read-only volatile states, only OCC works; and for read-write volatile states, none of lock and OCC works[2]. Fortunately, network states are persistent states and read-only volatile states, which leave us the design choice of OCC.

[2]If a state is varying by itself internally, there is no way to stop it varying when an external write is committed to it (either by lock or OCC).

Challenge 2: Network applications need to express semantics of executing an update in phases, which requires TUS to have interfaces and internal implementation to support this new semantics. The two-phase update in Example 2 (§2.1) is an example of such a requirement. However, an update to an SDN switch is usually implemented in an asynchronous fashion (e.g., Ryu controller [1]); to make things worse, the communication delays from the controller to different switches are usually different (depending on the network conditions and the switch states). Thus, even switch updates are issued in an expected order, there is no guarantee they are completed in the same order. sou **Solution.** TUS introduce a new operator `barrier()` for SDN applications, which divides switch updates into batches, expressing the semantics of phases in a network update. In the underlying implementation, TUS would read the updated switch states in the same batch to confirm their completion; for the log module, when a multi-phase update is replayed, the replay algorithm would also apply this read and check mechanism between phases.

3 DESIGN

We first give the architectural overview of TUS showing how ACID are guaranteed, and then detail the design of each part.

3.1 TUS Architecture and Workflow

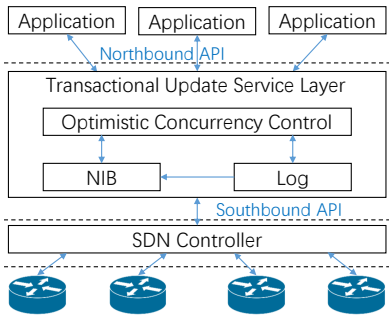


Figure 2: TUS architecture and deployment

TUS is a new layer sitting between the SDN controller and SDN applications, providing transactional update services to SDN applications. It has a NIB inside, maintaining the following information: (1) the network topology, including switches, links (pairs of switches), and link capacity and bandwidth allocation; (2) routing rules in each switch in the format of <switch, match, action, statistics>. In NIB, the switch, match, and action data are *persistent* states, and the statistics states are *volatile* states.

TUS have three modules inside. (1) The concurrency control (CC) module has northbound APIs (i.e., the APIs between TUS and SDN applications) to the SDN applications, which are *consistent* update interfaces. The CC module takes simultaneous network updates from multiple applications; it creates a shadow copy of network states for each network update and merges (or cancel) them finally, guaranteeing *isolation*. (2) The NIB has the network states, providing network information to the CC module. The NIB also has the southbound APIs (i.e., the APIs between TUS and the SDN controller) with the controller to read/write network states, where it uses asynchronous write with active confirmation (read and check after write) to make sure that each switch update is *durable*. (3) There is a log module guaranteeing *atomicity*. In TUS execution time, the log module records network updates in a persistent storage (a file on a disk); if TUS crashes and reboots, the log module would replay transactions that have completed lazy validation (§3.3) and cancel ones that have not.

3.2 TUS Interfaces and Examples

TUS has northbound APIs in Table 2. The life cycle of an update consists of four stages — READ, VALIDATION, WRITE, and INACTIVE. A network update starts by calling `tx=transaction()`, and then the CC module would record this new transaction and mark its state as READ. The application of the update would read and write network states by `tx.read()` and `tx.write()`. The read operations would make a copy of the read stats in the transaction's space (into a "read set"), and write operations would be recorded locally as well (into a "write set"). When an update is complete, it calls `tx.commit()` to start the execution of the update. The `tx.commit()` first marks the update into VALIDATION phase, where it validates whether there exists concurrent access conflicts; if there is no concurrent access conflict, it marks the update into WRITE phase, where all write operations of the transactions are executed; after the write phase, the transaction is marked as INACTIVE. As discussed in §2.3, TUS introduces `tx.barrier()` to express the semantics of executing write operation in phases. And TUS also has a "VOLATILE" parameter in `tx.commit()`, which gives the application flexibility whether it enables concurrency control on volatile states.

Algorithm 1 shows how to use TUS APIs to program a network update. The example is a 2-phase update in §2.1. The update transaction starts first, then a path is computed, the steps of update are submitted to TUS by `tx.write()` and `tx.barrier()` (including updating each hop and a barrier for phases). Finally, `tx.commit()` validates the update and write the update to NIB. If the validation fails, the application is notified and take its own actions on failure handling .

3.3 Concurrency Control

The persistent states in NIB are flow forwarding rules (match /action), which is read/written by SDN applications; while

Table 2: TUS interfaces and their operations

Interface	Behaviors in CC	Record in the log
`tx=transaction()`	start a transaction in "READ" stage	<tx_ID, START>
`tx.read(Switch, Match, "ACTION"/"STAT")`	read a state and record in read set	<tx_ID, READ, Match, "ACTION"/"STAT">
`tx.write(Switch, Match, Action)`	Record the actions in write set	<tx_ID, WRITE, Match, Action>
`tx.commit(VOLATILE)`	mark tx in "VALIDATION" stage	<tx_ID, VALIDATION, VOLATILE>
	mark tx in "WRITE" stage, and execute tx	<tx_ID, WRITE>
	mark tx in "INACTIVE" stage	<tx_ID, INACTIVE>
`tx.barrier()`	Record a "barrier" in tx space	<tx_ID, BARRIER>

Algorithm 1: 2-phase update

```
tx = Transaction()
firstHop, otherHops = GetHops(PathB)
for each hop in otherHops do
    tx.write(hop)
end
tx.barrier()
tx.write(firstHop)
if tx.commit() failed then
    Do something
end
```

the volatile states are *read-only volatile* statistics, which is only updated by data plane packet processing and can only be read by the SDN controller. For volatile states, locks cannot be applied for concurrency control, because the states are varying by themselves. The CC module adopts the lazy validation from OCC [2] for volatile state related updates, and to make all states operations uniform, persistent states are also concurrency controlled in the same way.

The traditional Optimistic Concurrency Control (OCC) divides `tx.commit()` into two stage: a validation stage and a write stage, both of which are in critical section across all updates. When an transaction c is validated, any other updates (called p) who (1) have not completed write phase when c starts (READ stage) and (2) have completed write phase when c starts validation (VALIDATION stage) would be involved in c's validation. If p writes some network states that are read by c, then c is invalid and the shadow copy of c is discarded; otherwise c is valid and all the write operations in c are merged into NIB.

In TUS, we customize the validation in two ways for network state update. First, when validating whether c's read set is polluted by other updates' write set, TUS also checks whether the volatile states in the read set vary by themselves (i.e., data plane events, by comparing the value in read set with the instantaneous value in the NIB). Second, since some applications have relaxed requirements into the isolation, the `tx.commit()` has a boolean parameter "VOLATILE" to decide whether the volatile states are considered in c's VALIDATION stage.

3.4 Failure Recovery

The SDN controller or SDN applications may crash in runtime; then the network operator would reboot them or perform failover to a new instance. In this duration, the log module would replay or cancel updates to guarantee *atomicity* of each transaction.

As the lifetime of a transaction is READ, VALIDATION, WRITE, and INACTIVE, and all stages and operations of a transaction is recorded in logs on a persistent storage, the log module scans the log sequentially, and the following actions are taken for each updates.

(1) All updates that are in READ and INACTIVE stages are discarded.
(2) All updates that are in WRITE stage are replayed and marked as INACTIVE.
(3) All updates that are in VALIDATION stage are copied to CC module and executed with concurrency control.

4 IMPLEMENTATION AND EVALUATION

We implement TUS based on the Ryu SDN controller [1], and the implementation has about 1000 lines of code. These experiments in evaluation were carried out on a machine with Intel I7 CPU and 48 GB memory. We use mininet [17] to emulate the network topologies in these experiments. We demonstrate the consistency guarantee and failure recovery functionalities of TUS; and we measure the overhead introduced by TUS to network updates.

4.1 Consistency Guarantee

2-phase update. We show that TUS can guarantee the consistency in network updates. We design an online path migration in Figure 1 and implement the algorithm in Algorithm 1 in TUS. The `tx.barrier()` is not used in the first setting of this experiment (i.e., before update to the 1st hop) and used in the second. We measure the packet loss in both settings and repeat the experiments for 100 times. The result is in Table 3. We can see that without `tx.barrier()`, packet loss is significant— about 1022 on average for each path update;

with `tx.barrier()`, the number of packet drop reduces to about 10, in which case the drop is not caused by transient inconsistent routing (black holes) but by buffer overflow when locking a switch's forwarding table to update rules.

Table 3: Packets loss with and without barrier

setup	with barrier	without barrier
packets loss count	9.5 ± 2.7	1021.6 ± 651.2
packets loss percentage	0.01%	1.20%

4.2 Failure Recovery

We show the effectiveness of the failure recovery function of TUS (the log module). We still use the topology in Figure 1 and measure the traffic throughput on path 1 and path 2 in the experiment (Figure 3).

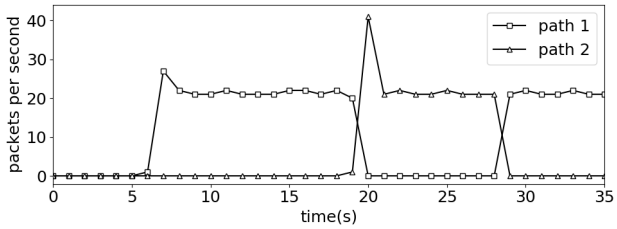


Figure 3: Failure recovery

Initially, there is no routing rules in switches. At $t \approx 7$ s, we reboot the controller with a log <(path 1, WRITE), (path 1→2, START)>, indicating path 1 is in WRITE stage and a migration from path 1 to path 2 in READ stage; after reboot path 1 setup is complete and path migration is discarded (traffic appearing on path 1).

At $t \approx 20$ s, we reboot the controller with a log <(path 1, INACTIVE), (path 1→2, WRITE), (path 2→1, START)>. After reboot, we observe traffic changes path from path 1 to path 2, which indicates that the update "path 1→2" takes effect.

At $t \approx 28$ s, we reboot the controller with a log <(path 1, INACTIVE), (path 1→2, INACTIVE), (path 2→1, WRITE)>, and observe traffic migrates from path 2 to path 1. This indicates that the final update "path 2→1" takes effect.

4.3 Overhead

We measure the overhead introduced by TUS's log module. In the experiment, we install a number of rules in a few phases to a switch. We tune the total number rules and number of rule per phase, and compare the total rule installation time with and without TUS's log module. The result is shown in Figure 4.

We conclude that two factors influencing the total rule install time— barrier wait time and file I/O time in log module. If there is one rule per phase, the barrier wait time dominates the total time (e.g., Figure 4a shows no significant difference between the rule update with and without log). But if all rules are updated in one phase (Figure 4c), the difference of total time is caused purely by the log file I/O, where the total increases from 3.8s to 19.9s for 10000 rules update (1.6 ms for each rule). Figure 4b shows a case in between, where the enabled log module causes 22% performance degradation when 10 rules are updated per phase.

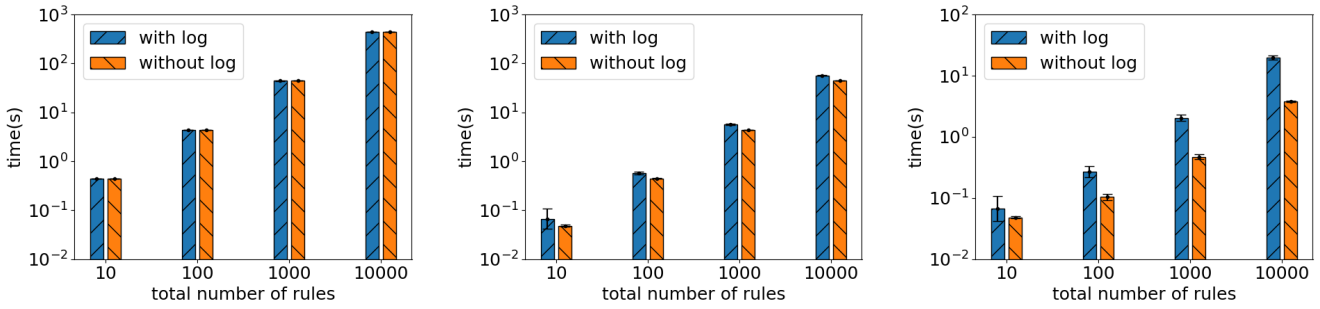
5 RELATED WORK

ACID for network update. ACID properties have been discussed individually in the existing literature, and TUS is a system providing all four properties and complements their insufficiency in each aspect. For *atomicity*, LegoSDN [7] improves system reliability by decoupling the SDN platform and SDN applications as independent processes, and it can guarantee the atomicity of all messages "in flight" between processes (by replaying them when a failure happens). In TUS, the atomicity is guaranteed at the granularity of "network updates" (e.g., a path setup), whose messages (e.g., one switch update rule) include the ones already processed, in flight, and being generated; and this granularity requires the design of "transactions" in the log module.

For *consistency*, Reitblatt et al. [25] propose a two-phase update algorithm to update a path, which avoids packet loss due to transient black holes during update; zUpdate[18] proposes a multi-phase update algorithm to update multiple paths, and avoids transient congestion on links; later Dionysus[10] proposes to use dependency graphs to incremental update new flows' paths instead of re-computes all paths, which saves the computation and update overhead. Canini et al. [6] propose algorithms to reduce tag complexity (i.e., the number of transient rules) in a multi-phase update. TUS provides APIs to describe these consistent algorithms; and compared with transactions in database systems, it also adds APIs to support phase-based transaction (i.e., `tx.barrier()`).

For *isolation*, Athens[2] designs the mechanism to decide which update should be executed, while TUS adopts first-commit-first-execute (which can be switched if needed). And statesman[27] proposes to use pre-defined rules (e.g., last writer wins or priority) to resolve conflict updates. Both solutions assume network state changes are from control plane updates; while in TUS, we make a categorization of network states, and conclude that due to the volatile nature of some network states, OCC[16] is the only choice for concurrency control. For *durability*, we test and found the switch rule update of Ryu[1] is not guaranteed to succeed, and thus, we implement a read-again-after-write to acknowledge the execution of a rule update.

Other Complementary work. In transactional network update systems, the goal is to "propose correct update plans and execute them correctly". While due to the complicated



(a) 1 rule per phase (b) 10 rules per phase (c) All in one phase

Figure 4: Overhead Measurement

nature of network management and applicable scope of these systems, there are some complementary solutions to guarantee networks in the correct states. Anteater and HSA[13, 20] assume that "network states may be wrong", and propose algebra to check network invariants (i.e., correctness about blackhole-freedom, loop-freedom); and Veriflow and NetPlumber[12, 14] are two further solutions to incrementally check updated network states. With a similar goal that "network states may be wrong", VMN[24] verifies the correctness of stateful networks with network functions. There also exist some network emulation tools such as BUZZ and Mikado[8, 30], which actually emulate the execution of network processing packets to find out "possible network state/policy errors". NDB and OFRewind[9, 28] are network debugging tools, which are used to find network problems when "a network error or failure already happens"; NDB captures all runtime traces for later analysis, and OFRewind replays recorded traces for operators to debug. These systems either detect that the network is in a bad state OR block a rule from being installed on a switch if it breaks operator specified semantics. TUS, on the other hand, implement correct (i.e., consistent) update plans and execute them in the correct ways (i.e., atomic, isolated, and durable). Within the applicable scope of TUS (§6), we advocate using TUS to correctly manage networks; and out of TUS's scope (e.g., non-switching rule management), these complementary solutions would cover the insufficiency of TUS.

6 DISCUSSION

Scope of application. The application of TUS has two prerequisites. First, the conflicts between updates can be clearly resolved. In TUS, conflict resolution is done by comparing the match field (flow headers) in rules, while a network may have other configurations whose conflicts are not easy to detect. For example, the priority setting of a flow may cause violation on end-to-end latency, whose causal relationship, however, is hard to identify. Second, recently, more network functions are introduced into networks, which contains variable states. These states could be read-write states (Table 1 in § 2.3), which, however, cannot be handled by locking and OCC. Complementary solutions that "check whether a network is correct after an update is proposed/deployed" can be used to handle errors in NF states (See §5).

Open questions. In TUS, one assumption is that network update commands from TUS's southbound APIs is executed directly to the network, which, however, does not always hold. For example, if a switch has its own intelligence to write rules (e.g., P4 switch [5]), this self-intelligence may disturb update from SDN controller; in other examples, there are several network enhancement component sitting between the controller and the network [3], e.g., TCAM optimizer [11, 19, 26, 29] and application composer [22, 23], they may also influence the execution of TUS update rules, causing the transactional update rules to be invalid. Our current standpoint is that the coexistence of transactional updates and network enhancements should be carefully designed to avoid conflict (or simply disable one of them).

7 CONCLUSION AND FUTURE WORK

To conclude, we build a holistic network update service for SDN application, named TUS. TUS abstracts the ACID requirements and provides the service to SDN applications, which eases the programming in SDN applications. TUS provides consistent update interfaces, uses a log to record updates guaranteeing atomicity, applies optimistic concurrency control for inter-update isolation, and uses read and check after written to guarantee the durability of each rule update. This design also overcomes the challenges of expressing multi-phase update semantics and executing updates with volatile states. Our preliminary implementation and evaluation show that TUS can implement consistent update and perform failure recovery, and has limited overhead. In the future, we would test TUS in more complete settings, including all combinations of workload, network topology, and scenarios (e.g., failure, recovery), with the metric of functional correctness, performance and overhead.

ACKNOWLEDGMENT

We thank the anonymous reviewers and our shepherd Behnaz Arzani for their insightful comments. This work is supported by Tsinghua University Start-up Grant.

REFERENCES

[1] [n. d.]. Ryu SDN Framework. ([n. d.]). https://osrg.github.io/ryu/

[2] Alvin AuYoung, Yadi Ma, Sujata Banerjee, Jeongkeun Lee, Puneet Sharma, Yoshio Turner, Chen Liang, and Jeffrey C Mogul. 2014. Democratic resolution of resource conflicts between sdn control programs. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 391–402.

[3] Theophilus Benson. 2017. A Call To Arms for Tackling the Unexpected Implications of SDN Controller Enhancements.. In *Proceedings of the First Asia-Pacific Workshop on Networking*. ACM, 15–21.

[4] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. 2014. ONOS: towards an open, distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 1–6.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[6] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. 2015. A distributed and robust sdn control plane for transactional network updates. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*. IEEE, 190–198.

[7] Balakrishnan Chandrasekaran, Brendan Tschaen, and Theophilus Benson. 2016. Isolating and Tolerating SDN Application Failures with LegoSDN. In *Proceedings of the Symposium on SDN Research*. ACM, 7.

[8] Seyed Kaveh Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. 2016. BUZZ: Testing Context-Dependent Policies in Stateful Networks.. In *NSDI*. 275–289.

[9] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks.. In *NSDI*, Vol. 14. 71–85.

[10] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. 2014. Dynamic scheduling of network updates. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 539–550.

[11] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. 2013. Optimizing the one big switch abstraction in software-defined networks. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 13–24.

[12] Peyman Kazemian, Michael Chan, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis.. In *NSDI*. 99–111.

[13] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks.. In *NSDI*, Vol. 12. 113–126.

[14] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. 2012. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 49–54.

[15] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. 2010. Onix: A distributed control platform for large-scale production networks.. In *OSDI*, Vol. 10. 1–6.

[16] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226. https://doi.org/10.1145/319566.319567

[17] Bob Lantz, Brandon Heller, and Nick 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

[18] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. 2013. zUpdate: Updating data center networks with zero loss. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 411–422.

[19] Shouxi Luo, Hongfang Yu, and Lemin Li. 2015. Practical flow table aggregation in SDN. *Computer Networks* 92 (2015), 72–88.

[20] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Godfrey, and Samuel Talmadge King. 2011. Debugging the data plane with anteater. In *ACM SIGCOMM Computer Communication Review*, Vol. 41. ACM, 290–301.

[21] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* 38, 2 (2008), 69–74.

[22] Jeffrey C Mogul, Alvin AuYoung, Sujata Banerjee, Lucian Popa, Jeongkeun Lee, Jayaram Mudigonda, Puneet Sharma, and Yoshio Turner. 2013. Corybantic: towards the modular composition of SDN control programs. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 1.

[23] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. 2013. Composing Software Defined Networks.. In *NSDI*, Vol. 13. 1–13.

[24] Aurojit Panda, Ori Lahav, Katerina J Argyraki, Mooly Sagiv, and Scott Shenker. 2017. Verifying Reachability in Networks with Mutable Datapaths.. In *NSDI*. 699–718.

[25] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for network update. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 323–334.

[26] Myriana Rifai, Nicolas Huin, Christelle Caillouet, Frédéric Giroire, D Lopez-Pacheco, Joanna Moulierac, and Guillaume Urvoy-Keller. 2015. Too many SDN rules? Compress them with MINNIE. In *Global Communications Conference (GLOBECOM), 2015 IEEE*. IEEE, 1–7.

[27] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. 2014. A Network-State Management Service, In ACM SIGCOMM. https://www.microsoft.com/en-us/research/publication/a-network-state-management-service/

[28] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. 2011. OFRewind: enabling record and replay troubleshooting for networks. In *USENIX Annual Technical Conference*. USENIX Association, 327–340.

[29] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. 2010. Scalable flow-based networking with DIFANE. *ACM SIGCOMM Computer Communication Review* 40, 4 (2010), 351–362.

[30] Yifei Yuan, Sanjay Chandrasekaran, Limin Jia, and Vyas Sekar. 2018. Efficient and Correct Test Scheduling for Ensembles of Network Policies. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. USENIX Association.