Software-Defined Networks as Databases

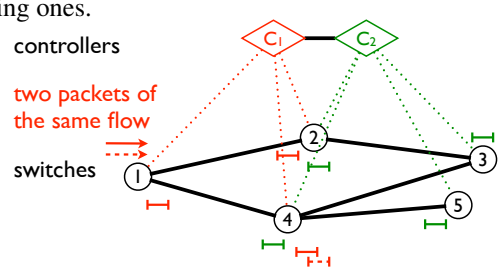
Anduo Wang* Wenchao Zhou‡ Brighten Godfrey* Matthew Caesar*
*University of Illinois at Urbana-Champaign ‡Georgetown University

1 Introduction

In software-defined networks (SDN), the separation of the control and data-plane moves the concurrency control from the data-plane to a separate, now logically centralized controller program. As a result, despite its intention to simplify programming, the separation forces the programmer to deal with a spectrum of concurrent events (e.g. execution of controller programs, in-flight packets), a task that is notoriously challenging and error-prone. It is not even clear what concurrency problems the programmer shall account for. Although early stage works propose specific correctness conditions and point solutions[4, 1], a comprehensive study is still lacking.

Most existing work focuses on the concurrency problem we call **atomicity**[1], which concerns one single network-wide update transaction. We use network-wide transaction (or transaction) to refer to a logical network operation that consists of potentially multiple switch-level updates. An atomicity failure scenario is shown in the figure (the red transaction spanning over switches $1, 2, 4$) when in-flight packets during the transaction are processed by a mixture of switches with rules before or after the transaction. In addition to atomicity, we identify concurrency problems arising from multiple transactions, which we call the **consistency** and **isolation**. To the best of our knowledge, they are *not* addressed in existing works. Section 3 will connect atomicity, consistency, and isolation to the well-studied ACID transactional semantics in databases literature [3]. Here, we introduce their intuition by examples:

- **Consistency.** Consider a load balancing controller program that instructs a switch (4) to forward a flow through a randomly yet uniquely chosen server (4's next-hop set to either 3 or 5). Two packets of the same flow that arrive in a short window could trigger two concurrent updates (the red transactions of solid and dashed intervals), causing two conflicting rules that forward the flow through different next-hops, thus violating consistency. Such consistency is not automatically enforced in today's SDN subsystem, but is manually handled by programmers.
- **Isolation.** Interleaving concurrent transactions may further complicate the problem (the red and green transactions by C_1, C_2) — the interleaving could leave the switches to apply updates in different orders. If updates are not commutative (e.g. updates to firewall and load-balancer), it can lead to inconsistent processing of flows. Note that interleaving transactions is desirable even with one single controller e.g. an on-going transaction in a data-center involving thousands of switch updates should not block incoming ones.



This paper seeks a generic approach to deal with concurrency, including the above problems. We observe a close connection between the SDN data-plane and a rich body of work on *distributed databases*, and use this to build a general model (Section 2) to reason about concurrency in the SDN data-plane. We then identify relevant properties (Section 3) guaranteeing correctness, and adapt the database concept of *linearizability* (Section 4) to allow concurrency in SDN while preserving correctness. The key contribution of this paper is to develop this general approach, which leads directly to algorithms that can in future work be built in systems.

2 Concurrency model

This section builds a general model for SDN by drawing connection to databases, shown in the following analogy.

*A concurrent system is a collection of **sequential processes** that perform **concurrent computation** on **shared objects**.*

*A SDN network is a collection of distributed **switches** that independently **update their switching rules** over the set of **shared forwarding path attributes***

The essence is to map the notions of objects, sequential processes, and concurrent computation, to the networking entities of forwarding path attributes, switches, and switch-level rule updates. The key intuition is inspired by the observation that networks provide services through forwarding paths, not the individual switches or links that set up the paths. We argue that the behavior of a network can be modeled by the services provided on the forwarding paths (i.e., the associated **attributes** such as reachability, bandwidth, way-points), and each packet, as it flows through the network, leads to a distinct view of the network state. We use path objects to refer to these forwarding paths and the associated attributes. This allows us to view a network as a distributed database, where each (partially replicated) database copy corresponds to one path object. When the path objects observed by all the flows are consistent (i.e.

[1] This corresponds to the per-packet consistency identified in [4].

satisfying user-defined constraints), they offer an illusion of a **one-path network**. From **objects as path attributes**, the **processes as switches** and **concurrent computation as switching rule updates** analogies follow naturally, capturing how the path objects are accessed and updated. We say an execution (a set of transactions) is a **serial** execution if the constituent steps are applied sequentially; otherwise it is a **concurrent** execution permitting transaction interleaving.

These familiar database notions allow us to port the rich literature of database techniques to software-defined networks. The next sections adapt to SDN the **ACID** properties for guaranteeing correctness and the linearizability [2] condition for achieving concurrency.

3 Concurrency correctness: ACID

ACID (atomicity, consistency, isolation, and durability) [3] is a set of properties concerning transactional semantics in databases. We adapt the properties to SDN to guarantee a correct distributed data-plane. We do not consider network failures in this paper and leave durability as future work. **Atomicity** requires a transaction to be applied in an "all-or-nothing" fashion, transforming the data-plane state atomically, exposing no transient states to an external observer. By interpreting "external observer" as packets that are processed (thus observing) by the data-plane, it follows that atomicity requires no packets are processed by a transient data-plane state; **Consistency** requires an update to transform data-plane among only valid states. A data-plane state is valid if all path objects satisfies user-defined constraints; **Isolation** requires updates from a set of concurrent transactions not interfering each other, that is, the execution of the concurrent transactions results in a network state equivalent to one that would be obtained in some serial execution.

4 Achieving ACI(D)-preserving concurrency: linearizability

Linearizability is a correctness condition for achieving concurrency while respecting ACID properties, its connection to networking shown in the following analogy.

*A system is linearizable if it **behaves like** a **serial processor of operations** on a **one-copy database**.*

*A software-defined network is linearizable if it **behaves like** a **consistent serial transaction of network updates** on **one-path network**.*

Here **it behaves like** says that a set of **concurrent transactions is linearizable** if we could construct an equivalent serial execution, which is also called the transaction's **linearization**. **Equivalence** means the effects of the updates in the concurrent transaction is identical to that in the serial execution. That is, the two different executions leave the path objects with the same values. More over, a **consistency function** over the path objects, defines the **consistent** network state, and captures user-defined constraints; whereas an abstraction function maps the consistent network state (collection of partial replication) to an imaginary **one-path network** (an imaginary aggregation of the replication). For example, a set of path objects with consistent destination values (i.e. either the path agrees on the consensus destination, or is not set up yet), the abstraction function generates a "one-path network" that offers reachability to the destination.

The above linearizability notion establishes an equivalence relation between an arbitrary set of concurrent network transactions over a distributed data-plane and its sequential counter-part over a one-path copy. This connection sheds light on the analysis and implementation of concurrent data-plane from a database angle, and it opens the opportunity to manage concurrency problems through well-tested database mechanisms. For example, given a set of concurrent transactions, to achieve an efficient yet correct (ACID properties) data-plane implementation, a naive approach is to start with a reference sequential execution, which is easy to obtain (e.g. via database primary-copy technique) but lacks the desired performance. Then derive from the reference a concurrent but equivalent execution, which offers better performance and is potentially less complex for it permits interleaving and asynchronous switch-level operations. The concurrent execution is obtained from either existing linearizability algorithms that heuristically add concurrency to the reference, or synthesizers (built on top existing linearizability checker) that exhaustively search for equivalent executions with maximal degree of concurrency. For implementation purpose, both the linearizability algorithms and the synthesizers can be built into a **transaction scheduler** sitting in the middle of the controller and the actual data-plane. The transaction scheduler is transparent to the programmer, acting like a virtual network hyper-visor with a shifted interest: allowing the programmers to focus on the control logic of their application by hiding concurrency control, thus automatically improving data-plane performance with correctness guarantee.

References

1. M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. Software transactional networking: Concurrent and consistent policy composition. In *Proceedings of HotSDN*, New York, NY, USA, 2013.
2. M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
3. C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.
4. M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proceedings of SIGCOMM*, New York, NY, USA, 2012.