

ZENITH: Towards A Formally Verified and Highly-Available Control Plane

Pooria Namyar*

University of Southern California

Arvin Ghavidel*

University of Southern California

Mingyang Zhang

Google

Harsha V. Madhyastha

University of Southern California

Srivatsan Ravi

University of Southern California

Chao Wang

University of Southern California

Ramesh Govindan

University of Southern California

Abstract

Today, large-scale software-defined networks use microservice-based controllers. Bugs in these controllers can reduce network availability by making the data plane state inconsistent with the high-level intent. To recover from such inconsistencies, modern controllers periodically reconcile the state of all the switches with the desired intent. However, periodic reconciliation limits the availability and performance of the network at scale. We introduce ZENITH, a microservice-based controller that *avoids inconsistencies* by design rather than always relying on recovery mechanisms. We have formally verified ZENITH’s specifications and have proved that it ensures the network state will eventually be consistent with intent. We automatically generate ZENITH’s code from its specification to minimize the likelihood of errors in the final implementation. ZENITH’s guarantees and abstractions also enable developers to independently verify SDN applications and ensure end-to-end safety and correctness. ZENITH resolves inconsistencies 5x faster than today’s designs and significantly improves availability.

CCS Concepts

- Networks → Network reliability; Network manageability; Control path algorithms.

Keywords

Software Defined Networking, Formal Methods, Availability

ACM Reference Format:

Pooria Namyar, Arvin Ghavidel, Mingyang Zhang, Harsha V. Madhyastha, Srivatsan Ravi, Chao Wang, Ramesh Govindan. 2025. ZENITH: Towards A Formally Verified and Highly-Available Control Plane. In ACM SIGCOMM 2025 Conference (SIGCOMM ’25), September 8–11, 2025, Coimbra, Portugal. ACM, New York, NY, USA, 25 pages. <https://doi.org/10.1145/3718958.3750533>

1 Introduction

SDN controllers are essential for managing modern data centers and wide-area networks [19, 28, 33, 34]. They simplify network

*Equal contribution



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGCOMM ’25, Coimbra, Portugal

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1524-2/25/09

<https://doi.org/10.1145/3718958.3750533>

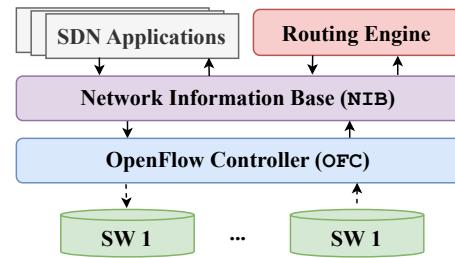


FIGURE 1: A microservice-based architecture similar to Orion [19] and ONOS [12].

management by providing high-level interfaces for developers to build applications. These applications communicate their *intended* network state to the controller, either to achieve a particular management objective (e.g., removing a switch from the network) or to react to network events (e.g., switch failures). The controller must (1) program the switches to match the high-level intent specified by applications, and (2) report the correct state of switches to applications. As a result, the controller’s performance, scalability, and reliability can significantly impact cloud applications.

Microservice-based designs. Early controllers were monolithic (e.g., Onix [33]). In contrast, modern large-scale proprietary (e.g., Orion [19]) and open-source controllers (e.g., OpenDaylight [4] and Onos [12]) adopt a microservice-based design. They consist of a collection of independently deployable microservices that collaborate to achieve controller functionality. A microservice-based design has two advantages: it can scale to networks with thousands of switches [51] and support over a million updates per second [19]. It also permits high feature velocity by allowing multiple teams to concurrently develop and deploy features [19].

Figure 1 shows a controller [19, 33] comprising two microservices: an OpenFlow Controller (OFC) and a Routing Engine (RE). OFC programs flow entries into switches. RE schedules network operations. They communicate through a Network Information Base (NIB). SDN applications [9, 42, 43, 56] also function as independent microservices in this design.

Internally, each microservice contains several components. For example, the OFC consists of a monitoring server, a worker pool, and event handlers. Components can execute concurrently and communicate through message passing or shared memory. Concurrent execution permits high operation throughput by leveraging multiple cores in modern server-class machines.

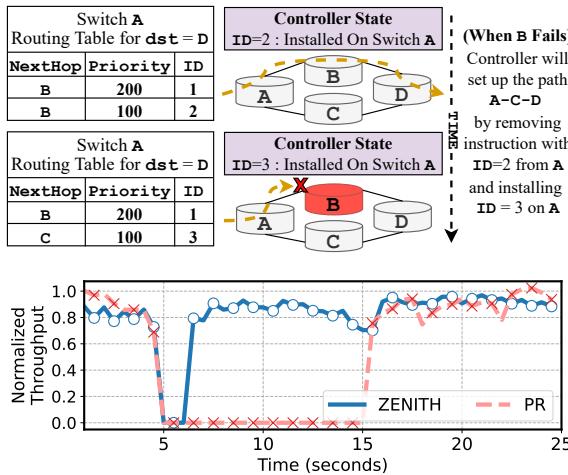


FIGURE 2: (Above) **Example inconsistency** between the control and data planes causing packet drops, see §G. (Below) **Periodic Reconciliation (PR) reduces availability** by waiting for the next scheduled reconciliation to find and resolve the inconsistency.

1.1 State Inconsistency

Both monolithic and distributed designs can suffer from *inconsistency* between the control and data planes. For example, if the controller mishandles component or switch failures, the data plane state may not reflect the operator’s desired intent. This inconsistency can result in blackholing traffic or routing it along congested paths. Inconsistency is a root cause of large failures at Google [24] that are reportedly difficult to diagnose. Many of these failures last for tens of minutes, extending network unavailability. As a result, state inconsistency poses a direct threat to availability budgets and remains a key obstacle in achieving high availability at hyperscalers [24].

Certain state inconsistencies are unavoidable. For example, after an application specifies intent, the switch state cannot be updated instantaneously due to processing and networking delays. During this interval, the state is inconsistent. Similarly, components in modern controllers (Figure 1) do not share fate: switches or controller components can fail independently of each other. The intent becomes consistent only when controller components have recovered and the controller has correctly processed switch events.

However, state inconsistency can arise from incorrect design or implementation as well. These can be *avoided* by fixing the controller or the application. To understand the types and prevalence of avoidable inconsistencies, we analyzed the commit history of OpenDayLight [4] (ODL, a widely-used open-source microservice-based SDN controller). In §A, we summarize 26 instances of ODL commits that fix inconsistencies resulting from incorrect controller or application designs. We briefly describe two examples here.

Networking Events. When an event such as switch failure occurs, the controller must update its own view of network state and notify applications. In ODL, separate threads handle switch failure and recovery. In one incident [5], a switch experienced a brief failure followed by a rapid recovery. Due to a race condition between these threads, the controller processed the recovery event before the failure event, leading to an incorrect perception of the switch

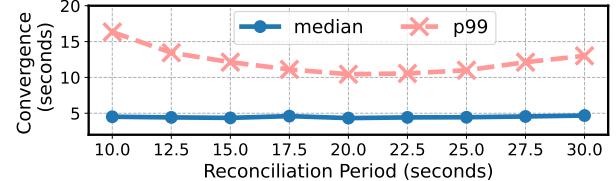


FIGURE 3: The impact of periodic reconciliation on tail convergence increases as the reconciliation period decreases.

status. The controller believed the switch was down while it was actually operational.

Component or microservice interactions. In ODL, multiple threads schedule route updates in response to switch failures. In one incident [6], two switches disconnected within a short timeframe. The first thread began calculating new routes without considering the second failure, while the second thread started computing routes based on both failures. The first thread ultimately updated the network after the second thread, but the second thread’s updates to the NIB took precedence. This resulted in the applications believing the correct routes were installed (based on the NIB state), even though the routes were actually incorrect.

State inconsistency can reduce network availability. Inconsistency can cause traffic to be dropped or routed inefficiently on congested paths [31, 45]. Figure 2 shows a simplified example. Switches A and D communicate via B. As a result of state inconsistency (described in §G), the controller is unaware that A has a high-priority flow entry for D with B as the next hop. The adverse impact of this *hidden* entry becomes evident only when B fails. To get A to begin forwarding traffic to D via C, the controller replaces the low-priority entry (with ID 2) with a new entry that has C as the next hop. Instead of using the new entry, switch A continues to use the hidden entry, which results in blackholed traffic. This example is not hypothetical; §6 describes how we reproduced this inconsistency in ODL.

1.2 Consistency Recovery in Modern Controllers

Modern controllers [4, 12, 19] recover from avoidable state inconsistencies using a mechanism called *periodic reconciliation* (or PR). A PR controller periodically (every 30s in Orion [19]) retrieves all flow state from every switch, compares it with the locally stored intent, and updates inconsistent entries. Although PR can eventually detect and remove the hidden flow entry of Figure 2, it has several drawbacks.

PR can still impact availability. Figure 2 shows that, after the controller updates switch A’s table, throughput remains zero until PR reconciles the hidden entry. In complex networks with greater path diversity and multi-path routing, such inconsistencies may not result in blackholed traffic but can still degrade throughput (see Figure 14).

More frequent reconciliation is not effective. One cannot improve availability by simply reducing the reconciliation period. Figure 3 shows the time it takes for the network to converge to the desired intent for different reconciliation periods on a testbed with 200 switches; see Figure 11 and §6.1 for a description of the

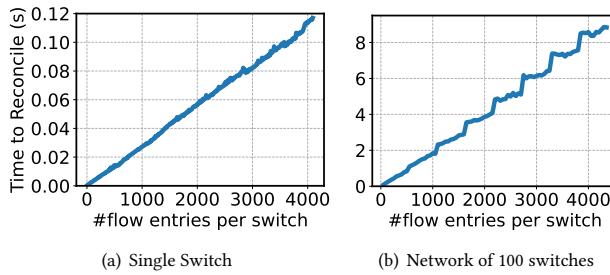


FIGURE 4: It takes longer to reconcile the state as the topology and the flow tables grow. (a) a Cumulus switch (SN2100) running OVS, and (b) a topology with 100 switches.

methodology. More frequent reconciliations increase the likelihood of network updates colliding with reconciliation cycles. Hence, reconciliation itself becomes a dominant source of tail latencies.

PR scales poorly with network size. Reconciliation time increases with network and flow table size. We show this in three ways. First, on a single Cumulus SN2100 switch, the time to complete reconciliation grows by 9× from 13 ms to 117 ms when table size grows by 8× (from 512 entries to 4096 entries, Figure 4(a)). Second, in a network with multiple switches, the controller can read the state of all switches in parallel, but updating the NIB with the received updates is the bottleneck. For example, in an otherwise unloaded network of 100 switches on the Sphere testbed [7], reconciliation time increases by an order of magnitude from 831 ms to 8.58 s when the number of entries in every switch’s flow table increases from 500 to 4000 (Figure 4(b)). Third, the tail of the convergence time distribution increases with the network size. 99th percentile convergence time of PR is 4.2× larger than the median on a testbed with 350 switches (Figure 11). This is again because convergence is delayed when network updates coincide with reconciliation.

Operators must, therefore, configure longer reconciliation periods to limit overhead in large networks. These delays adversely impact network availability.

1.3 ZENITH: Preventing Inconsistency

Given hyperscalers’ quest for stringent availability targets [24], we assert that *recovering* from inconsistencies is no longer sufficient. Availability budgets are increasingly tight: five-nines availability allows for only 5 minutes of downtime per year. A single SDN controller bug that introduces inconsistency, even if it occurs once before being fixed [11, 18, 38, 48], can lead to an outage that exceeds this availability budget.

In this paper, we explore a *proactive* approach that seeks to *prevent inconsistency* by design. We present ZENITH, a microservice-based controller which is architecturally similar to Orion [19], ONOS [12], and ODL [4], but with a key distinction: it is *formally-verified* to ensure eventual consistency between intent and switch state under a wide range of switch, link, component, and microservice failures. ZENITH thus prevents inconsistency for all modeled failures and only needs to fall back to consistency recovery in rare cases involving unmodeled failures. As such, ZENITH enables data plane state to converge to intent faster, improving availability.

Designing and implementing such a controller is challenging for three reasons. First, modern controller designs are complex and incorporate significant concurrency to achieve high operation throughput; specifying and verifying complex concurrent software is non-trivial [44]. Second, both the controller and SDN applications must be correct for the correctness property to hold end-to-end. However, in a microservice-based design, SDN applications evolve independently of the controller core. Finally, manually translating a specification into an implementation can be error-prone, voiding any guarantees inherent in the verified specification. In addressing these challenges, we make three contributions.

Contributions. ZENITH-core, our core controller, captures the component-wise decomposition of modern microservice-based controllers and presents a simple abstraction for SDN applications in the form of a directed cyclic graph (DAG) of operations on the network (§3). A DAG captures dependencies between operations on network state to ensure *hitless* updates (*i.e.*, no traffic impact [19]).

Our first contribution is a specification of ZENITH-core in TLA+ [35] (one of the largest known TLA+ specifications to date). We use the TLC model-checker [54] to verify that this specification ensures eventual consistency between the desired DAG and network state. To scale model checking, we develop an aggressive suite of optimizations that leverage symmetry, commutativity, and abstraction. Finally, enabled by the conciseness of the DAG abstraction, we *prove the correctness* of ZENITH-core using the TLA+ proof system [16]. To our knowledge, no prior work has developed a provably correct controller.

To ensure end-to-end correctness, ZENITH allows developers to formally specify and verify their SDN applications (ZENITH-apps, for short). However, verifying an application alongside the entire core is expensive and time-consuming. Our second contribution is a technique to verify ZENITH-apps *independently* of the core. This is made possible by the succinctness of the DAG abstraction (§4).

Our third contribution is NADIR, a tool to automatically generate code from ZENITH’s specifications (§5). NADIR borrows ideas from PGo [25] and has comparable correctness guarantees. Unlike PGo, NADIR supports microservice-based designs.

We evaluate ZENITH (§6) on a large testbed and show that it enables the data plane state to converge to the intent more than 5× faster at the 99th percentile compared to a controller that uses periodic reconciliation. Without our optimizations, our specification is too big to be model checked even within a day; with them, it completes in 3 seconds. Our code is available at <https://github.com/USC-NSL/ZENITH>.

Ethics. This work does not raise any ethical issues.

2 ZENITH Overview

Avoiding inconsistency requires writing controller and application code that (1) is correct and robust to races and (2) can correctly handle a wide range of single and concurrent failures despite network delays and switch state uncertainty. Writing such code is hard since programmers have for long had trouble reasoning about concurrent execution and the impact of failures. Fuzzing and testing can help find some bugs arising from incorrect implementation but may not be able to uncover subtle logical errors triggered under specific and complex circumstances [44]. Using a strongly consistent design (§7)

ZENITH	Role	Orion [19]	ONOS [12]	ODL [4]	
NIB	—	A logically centralized in-memory database that stores the network state, shares the state with different components, and is a central point for communication between microservices.	NIB	Store	MD-SAL
OFC	Monitoring Server*	A set of workers that communicate with switches.	OFE	OF Controller	OF Plugin
	Topo Event Handler	Updates topology and routing state in NIB based on OpenFlow events.	Topo Mangaer		Topo/SW Manager
	OF Worker*	A set of workers that translate high-level OPs to protocol-dependent flow messages.	OFE		OF Plugin
DE	DAG Scheduler	Schedules a new DAG on one of the Sequencers and ensures stale DAGs are deleted properly.	RE	Requires Standalone App	Requires Standalone App
	Sequencer*	A set of workers that ensure OPs are installed in the order DAG enforces.		Provider/ Listener	Provider/ Listener
	NIB Event Handler	Produces/Consumes events for/from NIB and is familiar with NIB semantics.		Runtime Environment	Runtime Environment
DE/OFC	Watchdog	Monitors all the submodules and restarts them if they fail.			

TABLE 1: Each component’s roles in ZENITH. The last three columns show the corresponding components in existing large-scale controllers with the same responsibilities. In some cases, we do not know the individual components inside the microservice. For example, Orion [19] does not describe the details of RE. For these, we only mention the high-level responsible service. (* = worker pool)

may help, but given the high throughput requirements of modern controllers, is likely to be impractical.

Key Ideas. We develop a model-checked formal specification of an SDN controller that provably achieves *eventual consistency* between intent and switch state. Our approach is inspired by work at large content providers that has demonstrated the feasibility of formally specifying and verifying practical systems [44]. ZENITH has two additional properties that bring it within the realm of feasibility. First, developers can formally specify and verify SDN applications *independently* of the controller. This preserves the ability to decouple application development from controller evolution (§1). Second, ZENITH *automatically* generates code, avoiding errors that might arise in manually implementing specifications.

ZENITH Overview. ZENITH has three components:

ZENITH-core (§3) is a microservice-based SDN controller that receives *intent* from applications. It ensures that (1) the network state will eventually align with the specified intent and (2) applications will eventually receive a consistent network state.

ZENITH-apps (§4). To develop a correct SDN application in ZENITH, users (1) develop a formal specification of their application and (2) define the desired correctness properties for their application. ZENITH verifies the application specification *independently* of ZENITH-core, yet ensures that the final specification is safe and will result in end-to-end correctness when used with ZENITH-core. This step is optional, and developers can also choose to build applications that use ZENITH-core without verifying them, at the expense of end-to-end correctness guarantees.

NADIR (§5). ZENITH specifications use an imperative language called PlusCal [54]. NADIR automatically generates ZENITH code from PlusCal specifications.

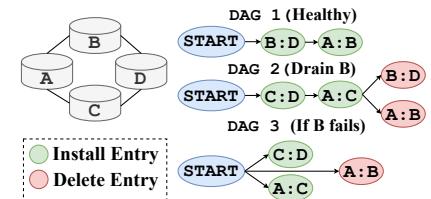


FIGURE 5: Example dependency graphs to route traffic from A to D. For brevity, we omit the source and destination in our notation.

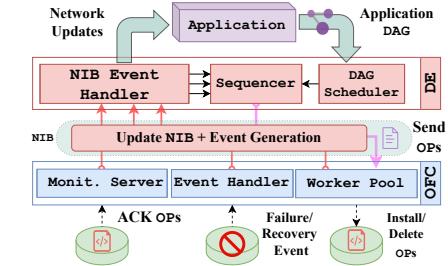


FIGURE 6: The ZENITH-core components and operation.

3 ZENITH-core

ZENITH-core’s functional decomposition resembles existing controllers, such as Orion [19], ONOS [12], and ODL [4] (Table 1). In this section, we describe how we derive a formal specification of ZENITH-core and verify it over a wide range of scenarios (Table 3).

3.1 An Abstraction for Application Intent

SDN applications respond to network events (e.g., failures) or changes in high-level intent (e.g., draining a switch) by specifying operations (OPs) on one or more switches. An OP can trigger a topology change (e.g., by disabling a port), or modify switch state (e.g., by changing a table entry). To ensure hitless updates [19], OPs often have dependencies [30, 47]: an OP may need to precede or follow one or more OPs on the same switch or across multiple switches.

ZENITH succinctly captures such dependencies using a directed acyclic graph or DAG of OPs. For example, Figure 5 shows one DAG for a healthy network, one for draining switch B, and one for recovering from B’s failure. Assume there is a single flow from A to D and the notation A : C denotes an instruction to route that flow’s traffic from A to C. If an application wants to “drain switch B”, the controller must route the traffic via C instead of B. A naive solution might install A : C and C : D in parallel. However, if A : C is installed *before* C : D, switch C will drop packets destined to D until C : D is installed. To prevent this, a controller must enforce an *order*, installing C : D before A : C.

The DAG abstraction allows us to: (a) precisely specify *correctness conditions* for an eventually consistent controller (§3.3), (b) verify ZENITH-core independent of ZENITH-apps (§3.6) and (c) prove the correctness of ZENITH-core (§3.8). Without it, these would have been difficult, if not impossible.

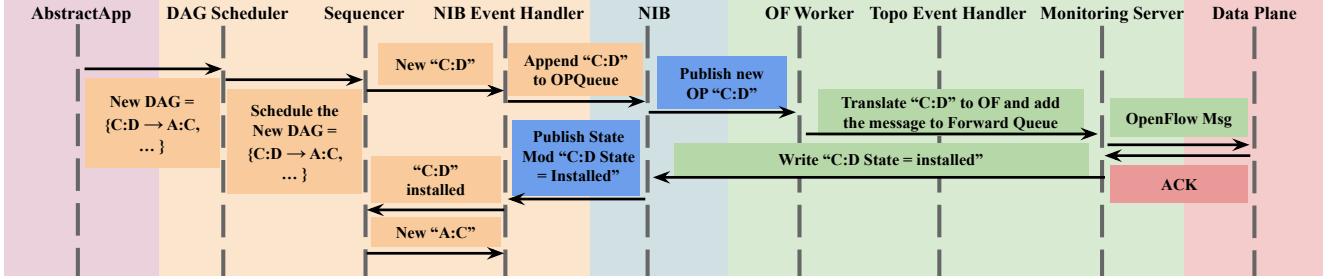


FIGURE 7: A simplified view of how ZENITH’s components interact to install the second DAG in Figure 5. For readability, this example simplifies the interactions. (Green: OFC, Blue: NIB, Orange: DE, Purple: AbstractApp, §3.6)

3.2 ZENITH-core Components

ZENITH-core consists of three main components (similar to existing microservice-based controllers [4, 12, 19]), each with one or more sub-components (Figure 6 and Table 1).

Network Information Base (NIB) is an in-memory database [33] that stores network state and facilitates inter-module communication. *DAG Engine* (DE) receives DAGs from applications and notifies them about new events. It includes a *DAG Scheduler* that processes DAG requests from applications, a pool of *Sequencers* that process OPs while enforcing DAG dependencies, and a *NIB Event Handler* that processes events from the NIB. The *OpenFlow Controller* (OFC) manages communication with switches. It has a *pool of workers* that translate OPs to protocol-dependent messages, a *Monitoring Server* that communicates with switches, and an *Event Handler* that processes switch failure and recovery.

Figure 7 shows how these components interact when ZENITH-core receives the second DAG in Figure 5. The state of each OP is recorded in the NIB, and the NIB Event Handler generates updates about the status of OPs for both Sequencer and other applications. Once Sequencer notices that C:D is done, it schedules A:C. The Sequencer’s output (e.g., C:D) is processed by one of the workers in the Worker Pool. Workers convert OPs into OpenFlow messages and send them to switches through Monitoring Server.

After a switch installs C:D, it sends an acknowledgment (ACK) to OFC. The Monitoring Server collects these ACKs and notifies the NIB that C:D is done. Sequencer is eventually notified of this event and can resume its operation by submitting the next OP (Figure A.7 has a more complex example).

3.3 ZENITH-core Correctness

Respecting dependencies in a DAG can be tricky, especially under failures. For example, when draining B in Figure 5, suppose ZENITH-core receives the ACK for C:D, but immediately fails and loses state about the ACK. Upon recovery, it will not install A:C as its dependency appears unsatisfied. In this case, C:D is installed in the network, but the controller has no record of it, leading to an inconsistency between network state and intent.

Furthermore, imagine switch B fails, and an application requests the controller to apply the third DAG in the middle of installing the first DAG and while OP A:B is still *in-flight*. To be correct, the controller must explicitly remove A:B. Otherwise, the OP A:B might be installed after the third DAG is complete, overwriting A:C and blackholing traffic.

Term	Definition
OP	Protocol-agnostic flow instruction
DAG	A graph to describe order of OPs
$\mathcal{I} = \{r\}$	set of all OPs where r is a single OP
$\mathcal{D} =$	Data plane state
$\{\mathcal{T}_d,$	Topology state
$\mathcal{G}_d\}$	Sequence of OPs
$\mathcal{C} =$	Control plane state
$\{\mathcal{T}_c,$	Controller’s perceived topology
$\mathcal{R}_c\}$	Controller’s perceived routing state
\mathcal{P}	The intended DAG.

TABLE 2: Terminology.

To avoid such cases, we define several *correctness conditions* for ZENITH-core. Below, we provide a formal description of these correctness conditions using TLA+ syntax [54]. These correctness conditions rely on the notation introduced below and listed in Table 2.

Control and Data Plane State (Figure 8). ZENITH-core takes DAGs from applications and converts them to the *desired network state*, which describes OPs performed on specific switches.

At any instant, the data plane state (\mathcal{D}) consists of: (1) *switch topology state* (\mathcal{T}_d), the current health status of each switch and its ports, and (2) *switch routing state* (\mathcal{G}_d), the sequence of OPs installed on each switch. The controller also maintains *its view* of the topology state (\mathcal{T}_c) and every switch’s routing state (\mathcal{R}_c), which can differ from the data plane due to delays and failures.

Correctness and Eventual Consistency. To ensure ZENITH-core’s *correctness*, it is sufficient to verify three conditions:

① **ZENITH-core never violates DAG OP dependencies.** We verify every OP is installed after its predecessor in the DAG, stated formally as:

$$\text{CorrectDAGOrder} \triangleq \forall(r_1, r_2) \in \mathcal{P}.\text{Edges} :$$

$$\text{firstInstall}(\mathcal{G}_d, r_1) < \text{firstInstall}(\mathcal{G}_d, r_2)$$

where *firstInstall* returns the time of an OP’s first installation. We consider only the first time an OP is installed, for two main reasons. First, an OP may be redundantly installed multiple times due to delays or inconsistencies, but only the initial installation determines whether the DAG ordering is respected. Second, in the event of a switch failure, an OP might be lost even though its successors have already been installed elsewhere – this is not considered a DAG violation, and the controller cannot prevent it.

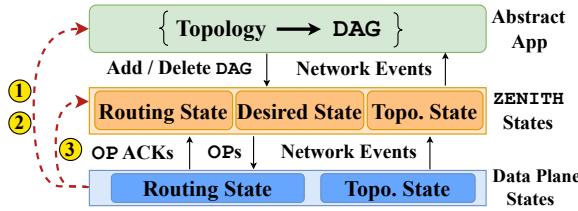


FIGURE 8: Correctness conditions for ZENITH-core.

- ② Switch routing state eventually matches intended DAGs.** We verify that all the OPs in the target DAG are eventually processed in the switches and remain installed thereafter:

$$\text{CorrectDAGInstalled} \triangleq \diamond \square (\forall r \in \mathcal{I} : r \in \mathcal{P} \rightarrow r \in \mathcal{G}_d)$$

where $\square P$ enforces invariant P at every step (*always* operator), and $\diamond P$ indicates that P must hold at least once (*eventual* operator). In combination, $\diamond \square P$ is an *eventual always* operator, meaning that P will eventually become true and will continue to hold true afterward. The invariant P validates that switch routing state conforms to the target DAG and that all OPs from the DAG have been successfully installed on the switch. In practice, the target DAG can change anytime by SDN applications based on the data plane state. We describe how we incorporate this in §3.6.

- ③ ZENITH-core's view of routing state eventually matches each switch's routing state.** We ensure the controller has, eventually, the correct view of the routing state and remains consistent thereafter.

$$\text{CorrectRoutingState} \triangleq \diamond \square (\forall r \in \mathcal{I} : r \in \mathcal{R}_c \Leftrightarrow r \in \mathcal{G}_d)$$

A controller that respects these conditions will eventually be consistent and correctly handle scenarios like those described above. We encode these conditions as safety ① and liveness ②③ properties. ②③ also ensure that the controller's view of the desired state – derived from DAGs – eventually matches its view of the routing state. We list other requirements that help debug our design and avoid unnecessary operations in §B.

3.4 Developing the ZENITH-core Specification

We specified ZENITH-core components and correctness properties in TLA+ [54], then systematically model-checked ZENITH-core:

- (1) We started with a single monolithic controller incorporating all components and verified its correctness in the absence of failures.
- (2) Next, we decomposed ZENITH-core into DE and OFC, but without introducing concurrency within each microservice. We verified this in the absence of failures.
- (3) Finally, we incrementally decomposed each microservice into components. For example, we added the Worker Pool to the OFC and verified the specification with failures; then added the NIB Event Handler to the DE and verified again, continuing this process until we had a verified microservice-based controller.

At each step, the model checker revealed specification errors, accompanied by a counterexample trace. We used these traces to iteratively fix the specification. We uncovered 83 specification errors, which we taxonomize in §3.9.

LISTING 1: A part of our monolithic specification that forwards OPs to switches.

```

1  fair process monolithicWorkerPool
2  variables OPToS = NADIR_NULL; begin
3  ControllerThread: while TRUE do
4      FIFOGet(OPQueueNIB, OPToS); /* Read an OP object from queue
5      if isSwitchHealthy(OPToS) then /* Check if sw is healthy
6          ForwardOP: controllerSendOP(OPToS);
7          UpdateNIBSend: FIFOPut(NIBEventQueue, SentEvent(OPToS));
8      else /* Report failure if switch is dead
9          UpdateNIBFail: FIFOPut(NIBEventQueue, FailEvent(OPToS));

```

To complete our description of how we verified ZENITH-core, we explain: (a) our model of delay and failures (§3.5), (b) how we verified ZENITH-core independently of ZENITH-apps (§3.6), and (c) how we scaled model-checking (§3.7).

Before doing so, we illustrate the process of specifying ZENITH-core using the PlusCal language in TLA+. Listing 1 shows a portion of our *initial* monolithic specification (step (1) above). We define it as a separate PlusCal *process* called *monolithicWorkerPool* for readability. This process defines an independent thread of execution and is responsible for sending OPs to switches. For clarity, we color *keywords* dark blue, *constants* purple, *Operator/Macro calls* cyan, *global variables* orange, and *process labels* red. Each process must have a unique identifier and may define and initialize its own local variables. The constant *NADIR_NULL* is a reserved name for NADIR (§5) that mocks a typical null-like value. The *monolithicWorkerPool* contains a *ControllerThread* that repeatedly reads an OP from a queue (Line 4), sends it to the switch (Line 6), and updates NIB state about the OP (Line 7).

3.5 Modeling Causes of Inconsistency

ZENITH-core's correctness relies on how its components cope with delays and failures. When verifying ZENITH-core's specification, it is unnecessary to model switches with full fidelity in order to capture these causes of inconsistency. Instead, our switch abstraction (*AbstractSW*) faithfully captures how failures and delays in switch-controller interactions impact the controller (see compositional verification in §3.7). Listing 2 shows our model of the switch.

Communication Delays. We capture non-deterministic delays in controller-switch communications [45] using two queues per switch: SWInQ and SWOutQ (Line 4). *AbstractSW* receives messages via SWInQ and responds to the controller via SWOutQ. The model checker can arbitrarily delay queue reads, modeling the variability of communication latency.

Switch Operations. As in other SDN controllers [12, 19, 33, 45], *AbstractSW* exports an OpenFlow [40]-like interface: it notifies the controller of status changes and acts upon flow programming events. In addition, *AbstractSW* is not Byzantine: if it acknowledges an OP, it has completed that OP correctly.

AbstractSW supports (1) installing a new rule, (2) deleting an existing rule, (3) returning the routing table, and (4) changing the controller role (important for failover). We model these as a fair process [35] (swMainProcess, Line 5) – a fair process is a concurrent

LISTING 2: Our *AbstractSW* model for a single switch.

```

1 variables /* Switch global variables
2   SWState = ..., /* State (routing table, ...)
3   HealthStatus = ...,
4   SWInQ = ..., SWOutQ = ..., /* Queues from/to the controller
5   fair process swMainProcess
6   variables ingressPkt = NADIR_NULL; begin
7   SwitchSimpleProcess: while TRUE do
8     await SwitchIsHealthy(HealthStatus);
9     FIFOGet(SWInQ, ingressPkt);
10    OP: PerformOP(SWState, ingressPkt); /* no-op if switch is down
11    ACK: AckOP(SWOutQ, ingressPkt); /* Send ACK if necessary
12  /* Note lack of `fair` prefix for failure and recovery processes
13  process swFailure begin
14  SwitchFailureProcess : while TRUE do
15    await SwitchIsHealthy(HealthStatus);
16    SetStateLoss(SWState); /* Can be None/Partial/Complete
17    SwitchFailure();
18  process swRecovery begin
19  SwitchResolveFailureProcess : while TRUE do
20    await ~SwitchIsHealthy(HealthStatus);
21    ResolveFailure();

```

unit of execution that would eventually take a specific step if the step remains continuously enabled (a property that holds in practice). *swMainProcess* internally calls a function *PerformOP* (Line 10) that takes the current switch state and the next request as input and alters the switch state accordingly. If needed, it also responds to the request by sending a message to the controller (Line 11). *PerformOP* is only enabled if the switch is healthy (Line 8).

Switch Failures can be triggered by many factors [22, 24, 43], such as faulty hardware or software bugs. Rather than modeling the root causes, we capture their impact. This simplifies abstraction as different root causes can have similar outcomes (e.g., reboot after an outage, or a kernel panic).

We capture failures along two dimensions. (1) State loss: a switch may lose all of its state (including the routing table and any ongoing requests from the controller), some of its state (retaining the routing table but losing all or part of ongoing requests from the controller), or none of its state after a failure. (2) Duration: a failure may be permanent or transient, with recovery occurring after a non-deterministic period. Together, this approach can capture a broad range of failure scenarios (see Table 3).

To model this, we introduce two processes, *swFailure* (Line 13) and *swRecovery* (Line 18). *swFailure* forces a healthy switch to fail, while the *swRecovery* restores a failed switch. We deliberately made the failure processes unfair – the model checker explores both executing or not executing these processes. This helps express transient failures (failure followed by recovery), permanent ones (*swRecovery* simply is not executed), as well as no-failure scenarios (both processes do nothing).

Controller Failures. We model failures at the level of both microservice and components within a microservice (e.g., the failure of a worker in a Worker Pool). We conservatively assume that the

failed component or microservice loses all of its state. The model checker can decide to fail a component or microservice at any step.

Fallback to Consistency Recovery. The model checker verifies ZENITH-core’s correctness under *any combination* of message delays, and switch and controller failures (Table 3). An implementation of this specification ensures that the switch state will eventually become consistent with intent. It removes all sources of avoidable inconsistency (\$1), thereby increasing availability.

However, as with any formal specification, the correctness of an implementation of the ZENITH specification is contingent upon: (a) the failure scenarios we model, (b) the assumptions we make about switches, and (c) how faithfully the implementation captures the specification. Violating any of these conditions may lead to inconsistencies. For example, we have assumed that switches install and then acknowledge an OP. If a switch bug violated this assumption, controller state would be inconsistent with switch state.

If such a violation were to occur in practice, we can update our *AbstractSW* model to reflect them and model-check ZENITH-core. In addition, a practical ZENITH-core implementation can include a component that recovers consistency, using periodic reconciliation, when such bugs manifest. This recovery component will likely need to fix inconsistencies very rarely relative to a PR-based controller, since ZENITH-core prevents most forms of inconsistency by design.

3.6 Verifying ZENITH-core without Apps

In practice, SDN applications are developed independently of, and sometimes long after, the controller. It would be tedious to re-verify ZENITH-core each time a new application is developed or an existing one changes. Fortunately, the DAG abstraction allows verifying ZENITH-core independently of application.

To do this, we verify the controller together with an *AbstractApp* which operates over a simple test topology. It contains a set of pre-defined DAGs: one consistent with the original topology, and others that correspond to specific events (e.g., a switch or a port failure). Unlike a real ZENITH-app, *AbstractApp* does not include logic for generating DAGs. It simply reacts to data plane events by deleting the current DAG and installing a new one consistent with the updated topology.

When model-checking ZENITH-core together with *AbstractApp*, the correctness conditions in §3.3 ensure that: (a) the data plane will never have a routing state corresponding to a deleted DAG and (b) the controller correctly notifies applications of data plane events. These properties guarantee that *control plane topology state* eventually matches the *switch topology state*, and enable ZENITH-apps to be verified independently of ZENITH-core (§4).

3.7 Scaling Model Checking

ZENITH’s specification is 8.6K lines, larger than that of other model-checked systems such as S3 and EBS at Amazon [44] (Table A.1). ZENITH also includes multiple concurrent components (Figure 7) and handles a broad class of failures (Table 3). This leads to a state-space explosion [37], rendering verification infeasible. To overcome this, we adopt several well-known scaling techniques, which are all

	Scenario	Details	Example cause
DP	Complete Permanent	Switch completely fails and never recovers.	Hardware Bug
	Complete Transient	Switch completely fails, but it recovers after some time. As part of the failure, the switch loses all its state (including its routing tables in TCAM).	Transient Power Outage
	Partial Transient	One or more components of a switch (e.g., ASIC) fail for a period of time. The sw may lose some of its states (e.g., buffers), but the TCAM state remains.	CPU overload
CP	Partial	Each of the components inside a microservice can fail independently and lose their local state. Eventually, a Watchdog would detect and restart the process.	Software Bug
	Complete	OFC, DE, and NIB can completely fail. In this case, ZENITH fails over to another instance of these services.	Power Outage
MO	Traffic Engineering	It sequences and schedules the operations necessary to reroute the traffic.	New demands
	SW Drain/Undrain	It sequences and schedules the operation to drain the traffic from a switch or to reinstate the switch in the network.	Software Upgrade
	Planned Failover	It receives requests for failing over some microservices and ensures they are done without any disruption to the network or to other components.	Maintenance
	Concurrent	We also allow for these failures and operations to happen concurrently (e.g., a switch failure during a management operation)	SW Software bug during SW Drain

TABLE 3: ZENITH is robust to different failures and management operations. (DP=Data Plane, CP=Control Plane, MO=Management Op)

sound: if the specification after applying these techniques is correct, the initial specification is correct too.

Compositional verification. Verifying certain aspects of a system does not require modeling every component in full detail. Consider two interacting components, A and B. To verify the correctness of A, we can abstract the impact of B on A as B_{abs} and verify a system consisting of A and B_{abs} . For this approach to be sound, B_{abs} must be an over-approximation of B [14, 15]. For example, when verifying ZENITH-core alongside a switch under complete failure, we can over-approximate the switch with a single component that either installs the OP and sends back an ACK or fails.

Partial order reduction. We can reduce the state space substantially by identifying sets of execution steps (potentially in different components) that are independent of (or *commute* with) each other and forcing a particular order of execution [8, 23, 37, 46]. This approach is sound since the ordering between independent execution steps results in the same final state. For example, a step in which a component modifies the internal state is independent of any step in any other component or microservice because these cannot access the internal state. Thus, we enforce a single order when writing a local variable in a component but allow component interleavings when writing a local variable in a microservice. We use locks and labels in TLA+ to exploit commutativity.

Symmetry reduction. Workers in Worker Pool have identical roles, and any of them can handle an incoming task without affecting controller correctness due to symmetry [13]. Thus, we can soundly eliminate redundant states by assigning tasks deterministically (e.g., always selecting the available worker with the lowest ID).

3.8 Proving Correctness

Even with these optimizations, parallelized model checking ZENITH-core in the presence of a single switch failure and recovery for a 16-node topology and DAGs with a total of 15 OPs takes about 2 hours. Thus, model checking ZENITH-core does not scale to arbitrary DAGs and topologies. Moreover, model-checking for a few topologies and DAGs does not guarantee ZENITH correctness for all possible inputs.

To obtain such a guarantee, we have developed a proof (§F) of correctness for ZENITH-core. This uses the TLA+ proof system [16] to show that ZENITH-core is correct for (a) any topology and DAG, (b) for the failures we model, and (c) under the switch behavior assumption we make. At a high-level, our proof works as follows. First, we identify several properties and prove by contradiction that, if ZENITH-core is not correct, at least one of these properties must be violated. Second, we use the TLA+ proof system to prove that each of these properties holds for our ZENITH-core specification. Prior proofs (e.g., for Paxos correctness [36]) have used this methodology.

3.9 A Taxonomy of Specification Errors

While verifying ZENITH-core, we encountered several subtle specification errors, which we summarize in this section (see §C for a detailed taxonomy and §G for an example TLA+ trace). The description references Listing 3, the final specification of the WorkerPool. Listing 1 describes the initial specification.

State Management Errors. The state of an ongoing operation can be spread across multiple microservices, their components, or switches. We have found three common approaches to ensure robust state management. Where possible, these apply fixes that minimally impact performance since high operation throughput is important for modern controllers.

Careful ordering of operations. Many components (e.g., Listing 1) have to take some actions (e.g., forward C:D) and update some shared state (e.g., update NIB) to reflect the action. If component (A) performs the action before updating a relevant shared state (Lines 6 and 7 in Listing 1), other parts of ZENITH-core may progress and update the same shared state before (A) does, causing an inconsistency. In these cases, we first update the state and *then* perform the actions. This implicitly serializes future updates to the same state (Lines 9 and 10 in Listing 3).

State recording and crash recovery. The above fix can be fragile. Suppose the component fails after updating the state but before taking the action. When it restarts, it might mistakenly assume that the action has completed, causing the controller to wait indefinitely

LISTING 3: The final WorkerPool specification

```

1 fair process WorkerPool
2 variables OPToS = NADIR_NULL; begin
3 StateRecovery: /* State recovery logic, executed on startup
4   WorkerPoolStateRecovery(workerPoolState);
5 ControllerThread: while TRUE do
6   AckQueueRead(OPQueueNIB, self, OPToS);
7   workerPoolState[ self ] := OPToS; /* Record state
8   if isSwitchHealthy(OPToS.sw) then /* Check if sw is healthy
9     UpdateNIBSend: FIFOPut(NIBEventQueue, SentEvent)(OPToS);
10    ForwardOP: controllerSendOP(OPToS);
11   else /* Report failure if switch is dead
12     UpdateNIBFail: FIFOPut(NIBEventQueue, FailEvent)(OPToS);
13   RemoveOPFromQueue:
14     workerPoolState[ self ] := NADIR_NULL; /* Clear state
15   AckQueuePop(OPQueueNIB, self);

```

(e.g., for an OP to get installed). To fix this, each component must devise a state machine to track the progress of actions. For example, the component responsible for forwarding an OP to a switch should first record it as in-progress (Line 7), update the shared state (Line 9), perform the action (Line 10), and then mark the action as done (Line 14). Now, if the controller fails, it can read the state and recover from the crash upon restart (Line 4).

Event Processing. Many components use an event-based processing model where they receive and process events from other components. The WorkerPool (Listing 3) processes OPs from an OP-Queue and sends them to switches. If it crashes after dequeuing the event but before completely updating the state, the event is lost. To avoid this, our specification reads the head of the queue (Line 6), processes the event, and removes the event once its processing is complete (Line 15).

State machine design errors. Multiple components can track the state of DAG or OP installation, and each component may contribute to a different transition in the state machine. For example, we keep the state of OPs in the NIB. One component can mark the OP as DONE after receiving an acknowledgment from the switch. Another can reset the OP’s state to NONE when a switch recovers from a failure. We illustrate two sources of errors in such designs.

Designing a correct state machine. In the above example, it is possible for the two components to receive an OP acknowledgment and a switch recovery event at the same time. This can happen during a short switch failure/recovery event, which creates ambiguity: Was the OP installed before the switch failure or after its recovery? In such cases, it is better to be conservative and assume the OP was *not* installed.

Accounting for delays in operations. Ignoring delays in OP installation is another source of error. Suppose a controller starts draining a switch, then receives a request to undrain the same switch. If the controller assumes that the drain completes instantly and starts the undrain prematurely, the operations can be executed

in the wrong order. We account for these delays by adding transition states to the state machine.

Directed Reconciliation. When a switch recovers (§3.5), the controller does not know the extent of state loss. To mitigate this uncertainty, we can: (a) wipe out and reprogram the switch or (b) identify and resolve inconsistencies with the corresponding switch. Unlike periodic reconciliation, which queries all the switches, approach (b) uses *directed reconciliation*, which reads the state only from the switches known to be potentially inconsistent. The optimal approach depends on the severity of state loss; for small inconsistencies, directed reconciliation is faster. Such tradeoffs are harder to determine at the specification level, so we develop and evaluate (§6) both versions, with and without directed reconciliation.

4 ZENITH-apps

SDN applications request ZENITH-core to add or remove DAGs. However, the apps themselves can be error-prone, which can also impact network availability [11, 49].

We now describe how developers can specify and formally verify apps in ZENITH to ensure that correctness guarantees apply end-to-end and not just to ZENITH-core. This step is optional: a developer can instead modify an existing application to use the DAG abstraction without verification. In this case, end-to-end correctness may be violated – the application may schedule an incorrect DAG.

Developer view. ZENITH developers can (1) formally specify their applications in PlusCal [1] and (2) define the correctness properties of their applications. They can use TLA’s model checker to iteratively fix errors until they obtain a verified specification. They can then use NADIR (§5) to generate the application’s code. §E describes the PlusCal specification of a ZENITH-app that drains switches.

Guaranteeing end-to-end correctness requires jointly verifying ZENITH-app and -core. However, model-checking the entire specification can be time-consuming and hinder app development. Fortunately, ZENITH-core’s design allows developers to verify ZENITH-apps *independently of the core*.

Independently verifying ZENITH-apps. ZENITH-core guarantees (§3.3) that (a) a DAG submitted by an app is eventually installed in the data plane, absent intervening modifications, and (b) the NIB’s view of topology and routing state is eventually consistent with that of the data plane. Given this, we can guarantee end-to-end correctness by ensuring ZENITH-apps: (1) correctly and safely react to the events (that they are guaranteed to receive from ZENITH-core) by recomputing the DAGs consistent with current network state and (2) correctly add the DAGs as input to ZENITH-core.

To do this, we verify ZENITH-apps with an *AbstractCore*, rather than the full ZENITH-core specification. *AbstractCore* maintains a list of DAGs and delivers the arbitrary network events generated by the model-checker to the app. The app processes these events, determines whether they invalidate an existing DAG, deletes the DAG, and installs a new one consistent with the current state.

ZENITH-app designers can verify correctness for two types of invariants: (1) App-specific invariants – e.g., ensuring that a drain app does not disable more than 25% of the network’s capacity [51, 56]; and (2) DAG correctness invariants – e.g., ensuring no traffic flows over a drained switch.

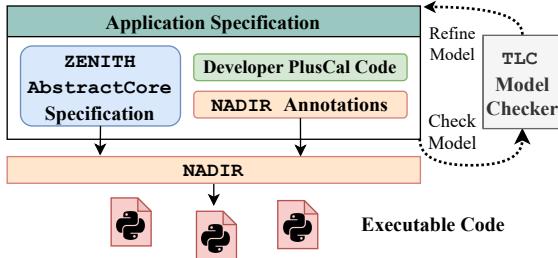


FIGURE 9: Input and output of NADIR. A TLA+/PlusCal specification is directly turned into executable code.

Other than drain, we also developed and verified specifications for a traffic engineering (TE) app, and one that executes OFC planned failover. We show that independent verification significantly reduces the time-to-verify these apps (§6).

5 Generating Code

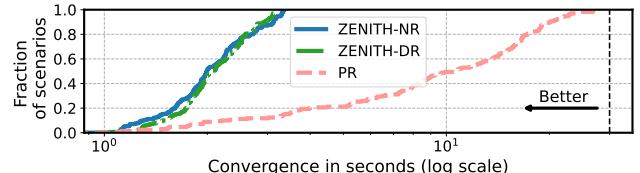
We have developed NADIR, a tool that automatically generates executable code from ZENITH-core and ZENITH-app specifications. It takes as input a PlusCal [54] specification. It outputs an Abstract Syntax Tree (AST) for the specification. Then, given a pre-defined runtime code library for a specific programming language (Python, in our case), NADIR generates executable code matching the specification for that language.

This is difficult, as specifications are too abstract for direct conversion to executable code. For example, PlusCal does not require specifying types of global and local variables, of inputs and outputs, as well as the entry point of each process. Therefore, prior work, PGo [25], which generates code for TLA+ specifications requires specifications to be written in a new programming language, Modular PlusCal, which includes constructs for type and entry point specifications. But, requiring a new language would likely hinder adoption. More importantly, PGo generates code for concurrent threads of execution that share a single address space, so is unsuitable for a microservice-based design like ZENITH.

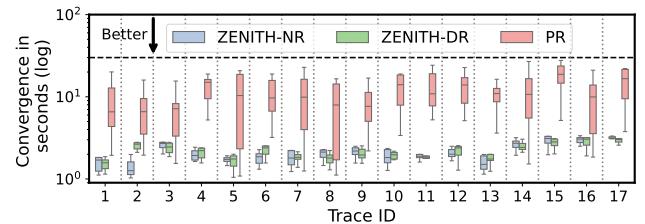
To overcome PGo’s shortcomings, NADIR requires developers of ZENITH-apps to annotate their PlusCal specifications with type information. These annotations enable NADIR to convert the specification into an AST. After the programmer verifies their PlusCal specification of an app, they must re-verify their specification after they annotate it (Figure 9).

Second, in ZENITH, all persistent state is in the NIB. In our specification, global variables are fully persistent and must survive failures; local variables have no persistence. Therefore, NADIR automatically generates code to ensure persistence and serialized access for global variables stored in the NIB and accessed by other microservices. NADIR analyzes the AST to distinguish between operations on local and global variables.

NADIR-generated code preserves the specification’s correctness guarantees under the following conditions: (a) the PlusCal specification is verified to be correct with NADIR annotations, (b) the implementation of synchronization primitives and the runtime library is correct. These guarantees are similar to those that PGo provides.



(a) CDF of Convergence Time



(b) Individual Example Traces

FIGURE 10: Our ZENITH-core implementation is robust to the TLA+ traces and converges faster than PR. The dashed lines show the reconciliation period (=30s, as in Orion [19]).

6 Evaluation

Our evaluations demonstrate that ZENITH-core converges faster and scales better than PR-based controllers. ZENITH maintains higher throughput in practical scenarios such as WAN failures [34]. Our scaling optimizations (§3.7) are crucial for verifying ZENITH-core (§3.6). Decoupling app from core also speeds up verification by a few orders of magnitude (§4).

Experiment setup. We use the implementation of ZENITH that NADIR generated from the fully verified specifications (§5). All results described in this section use the Sphere (formerly Merge) testbed [7], which can materialize hundreds of VMs running Open vSwitch. On these, we evaluate synthetic, data center, and WAN topologies of varying scales.

Metrics. Most experiments measure *convergence time*, the time between when DAG installation commences and when the controller certifies in the NIB that the data plane has converged to the state corresponding to the DAG. Some experiments also measure the *throughput* achieved by flows during the DAG installation. Other results quantify the *time-to-verify* specifications and the *complexity* of the specifications (§6.3).

Trace Replay: Switch and Component Failures. We run ZENITH and each baseline on the set of TLA+ traces obtained during the process of developing the ZENITH-core specification (§3.4). When TLA+ detects a safety or liveness violation, it produces a trace of the steps needed to generate that violation. We developed a *Trace Orchestrator* (TO) which enforces the execution of a trace by blocking modules from proceeding until the trace demands it. It enforces which blocked module should be allowed to take a step in the trace and which failure to be injected into which component at what step. For each trace, a correct implementation of ZENITH-core should converge, while a PR controller will incur an inconsistency that will be resolved by periodic reconciliation. Thus, these experiments have two goals: (1) to validate the correctness

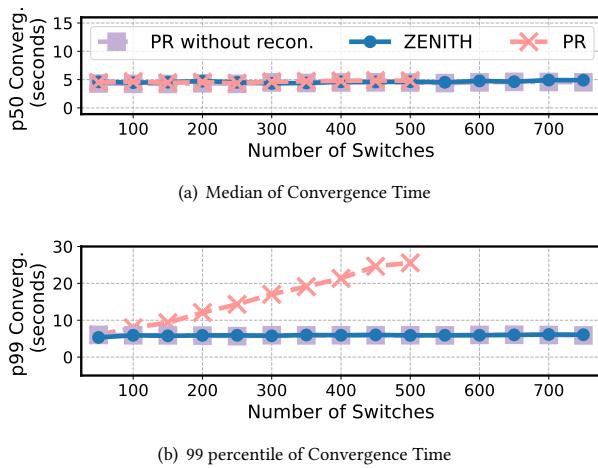


FIGURE 11: ZENITH scales better than PR. ZENITH’s median and 99p remain the same with the network size. PR’s tail convergence increases by up to 5× due to reconciliation interfering with convergence. We confirm this by evaluating a controller with the same implementation as PR that does not do the reconciliation. PR is unable to scale beyond 500 nodes since it fails to converge within the reconciliation interval.

of NADIR-generated code and (2) to compare convergence time differences between ZENITH and PR for the corresponding trace.

TO does not scale to large topologies. To evaluate on a larger network, we disable TO and instead randomly induce switch and component failures. This tests ZENITH’s ability to scale to larger topologies as well as further validates NADIR.

Comparison Baselines. We consider two variants of ZENITH: (a) without any reconciliation (Zenith-NR) and (b) with only directed reconciliation (Zenith-DR, §3.9). The PR controller we use is a simplified version of ZENITH-core that is robust to concurrency errors but relies on periodic reconciliation to be correct under switch or component failures. Some experiments also use a PR variant in which the controller preemptively reconciles switch state when a switch comes up (PRUp). Our reconciliation implementation follows open-source controllers such as ONOS [3], and follows the description in Orion [19].

In one experiment, we compare ZENITH against an open source controller (ODL). The complexity of ODL precludes a more complete evaluation; many of our experiments use trace orchestration, which requires changes to controller code.

6.1 Evaluating ZENITH-core

ZENITH-NR converges an order of magnitude faster than PR. Fig. 10(a) shows the distribution of convergence times for ZENITH-core and PR across 170 different runs over 17 TLA+ traces (10 runs per trace). These traces trigger either inconsistencies between data and control plane or state update races and cause deadlocks in a microservice component. PR includes a timeout, much shorter than the PR interval, to resolve such deadlocks.

On average, PR takes 11.2s to converge in these scenarios, whereas ZENITH-NR takes 2.11s (5.3× lower). At the 99th percentile (p99), PR converges within 26.8s, whereas ZENITH-NR does so within 3.3s in all cases (8.1× lower).

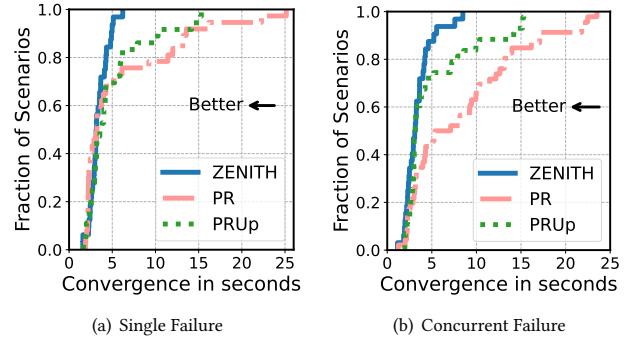


FIGURE 12: ZENITH has better tail convergence on random switch failures, measured on a 300-node topology.

Fig. 10(b) shows the boxplot for individual sample traces. ZENITH has lower variation compared to PR for most of the traces. PR’s convergence depends on the timing of failures relative to the reconciliation. When the failures occur just *after* the reconciliation, PR must wait a full round.

ZENITH-NR and ZENITH-DR have comparable convergence. TO only works with small and shallow DAGs, thus the two systems show virtually identical results. Unless otherwise stated, ZENITH-NR is the system used in what follows.

Convergence time on large-scale topologies. To evaluate ZENITH’s performance at scale, we obtained 750 physical nodes on the Sphere testbed [7] and configured them to match the KDL topology from the Topology Zoo [32], the largest one in that collection. The rest of this subsection reports results of experiments on topologies of different sizes, where each is a subgraph of KDL.

Even in the absence of failures, PR tail convergence time scales poorly (Figure 11). In a network with no failures, network policy changes and management operations trigger DAG installation. To mimic this, we run 5-minute-long experiments where we repeatedly install a new DAG and measure how long it takes for the controller to install it. We schedule a new DAG only after the controller converges on the previous one. Each DAG only updates a portion of the topology (*i.e.*, 5 switches). We repeat each experiment 10 times.

ZENITH’s median convergence time is comparable to that of PR across all topology sizes (Figure 11). However, unlike ZENITH, PR’s 99p latency increases as the network scales. This is because PR must retrieve and process switch state, which delays DAG installation. We confirmed this by disabling reconciliation in PR; for such a controller (which is not robust to failures), 99p convergence is unaffected by topology size. In our experiment, PR fails to converge within the 30-second reconciliation interval once the network size exceeds 500 nodes. In contrast, the topology size *does not affect* ZENITH’s tail convergence.

With random switch failures, PR tail behavior degrades significantly (Fig. 12). We compare how ZENITH handles switch failures against PR and PRUp (we use PRUp only in this experiment since it differs from PR on handling switch recovery). For this and the next experiment, we used a 300-node subgraph of the KDL topology. Fig. 12(a) shows convergence under randomly generated

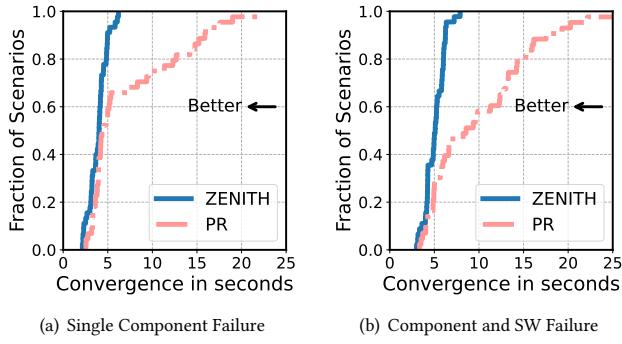


FIGURE 13: ZENITH’s tail convergence is lower during random component failure, measured on a 300-node topology.

switch failures, with at most one failure at a time. ZENITH has the same median as PR and PRUp, but its tail convergence time is much lower ($4.1\times$ lower p99 convergence). A single switch failure triggers a DAG update but does not typically cause state inconsistencies. Thus, PR and PRUp often converge as fast as ZENITH. At the tail, however, both PR and PRUp encounter inconsistencies that only reconciliation can resolve.

In Figure 12(b), we study convergence under concurrent failures. We set the average failure inter-arrival time to be shorter than the convergence time. ZENITH’s median convergence is unaffected, but PR and PRUp perform worse. PR’s and PRUp’s median convergence is $2.5\times$ and $1.5\times$ worse than ZENITH. At tail (p99), PR and PRUp’s convergence is $2.8\times$ and $1.9\times$ worse than ZENITH. PRUp fares better than PR due to the additional reconciliation upon switch recovery, but both still fail to manage in-flight OPs correctly at the tail.

ZENITH converges faster under component failure (Fig. 13).

Under single failures, ZENITH’s median is $1.9\times$ and its p99 convergence is $3.4\times$ lower than PR. With concurrent failures, ZENITH’s benefit is $2.0\times$ on median and $3.2\times$ at the tail.

6.2 Evaluating ZENITH-apps

Traffic Engineering (TE). We quantify how much ZENITH (TE + core) improves flow throughput. We use the B4 WAN topology [29], which we created in our testbed. We compare against a TE app on PR and on ODL [4].

Experiment setup. We first let the network run for a few seconds with traffic between selected sources and destinations. Then, we force one of the switches to completely fail. This triggers a fast local recovery (at $t = 8$) where the impacted sources detect the failure and migrate their corresponding traffic to a predefined backup path [34]. Local recovery helps maintain the connection but can push the traffic to a path with lower available capacity (caused by congestion or fiber errors [52]). Switch failure triggers DAG installation around $t = 16$. Before this DAG is fully installed, TE notices the congested link and schedules another DAG installation. PR fails to handle overlapping DAG installations and creates an inconsistency, which it later resolves using reconciliation. ODL behaves similarly (Figure A.2).

ZENITH’s TE maintains higher throughput than PR’s TE (Figure 14). In ZENITH, throughput improves as soon as TE schedules a

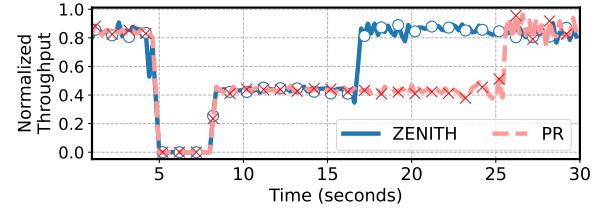


FIGURE 14: ZENITH maintains higher throughput compared to PR on the 12-node B4 topology in our testbed (see also §D.1).

new DAG (at $t=16$). However, PR fails to maintain consistency and has to wait for reconciliation (at $t = 26$ s). This means the traffic is running at a lower throughput for 10s. Overall, ZENITH exhibits $1.47\times$ higher throughput than ODL, and $1.23\times$ than PR.

OFC Planned Failover. Fig. 15 shows the convergence times during planned OFC failover. We use TO to replay 5 of the traces that exhibit inconsistencies. Fig. 15(a) shows the distribution of convergence times across 50 runs of these traces. ZENITH has a bounded and small convergence time. Compared to PR, it reduces convergence time significantly: on average, it is $2.3\times$ faster, and its p99 convergence is $3.8\times$ lower. Moreover, the variance with ZENITH is much smaller (Fig. 15(b)) than with PR for the same reason as in Fig. 10(b).

Drain/undrain. On a Fat-tree with background traffic at $\sim 80\%$ load, we drain an aggregate switch at $t=20$ and undrain it at $t=40$. Figure 16 shows the aggregate throughput of impacted traffic normalized by the link capacity. ZENITH is able to keep the throughput consistently high, with only a slight throughput decrease when the switch is drained, because of reduced network capacity.

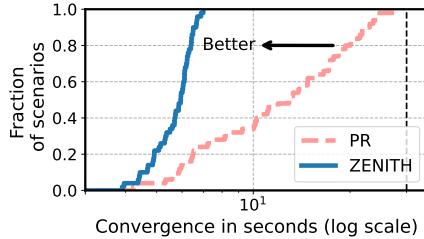
6.3 Quantifying the ZENITH Specification

Decoupling apps from core. We verified drain/undrain both with the full specification of ZENITH-core and with AbstractCore (§4). We found that decoupling and using AbstractCore reduces the verification time by more than $100\times$ (from 30 m to 2 s). Other apps can also be verified quickly: TE in 6 s, Failover in 3 s.

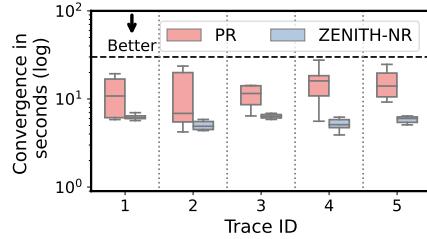
Scaling techniques from §3.7 (Table 4). We apply multiple scaling techniques to verify ZENITH-core, with much larger specifications than prior work. To quantify the impact of each technique, we measure the verification time under a single switch failure that cause a transition from a DAG of size 2 to a DAG of size at most 3 (involving up to 5 OPs).¹ Without any of our techniques, TLA+ explores more than 200M states and crashes after 30 hours due to memory exhaustion. Applying symmetry reduces the time to under 11 hours and 82M distinct states. Applying compositional verification on top of symmetry further reduces the runtime by $7.6\times$ and the number of states by $7.5\times$. The final runtime with all the optimizations, including partial order reduction, is 3s, and TLA+ only needs to verify 12K distinct states. Our optimizations also reduce the diameter (number of steps on the longest trace).

Error and Specification Complexity. Our TLA+ traces from ZENITH-core have a median length of 56 steps, with a maximum of 110 and a minimum of 21 (Figure A.6 shows the distribution). This

¹Verification in the presence of switch recovery or larger DAG sizes is only feasible with our full suite of optimizations.



(a) CDF of Convergence Time



(b) Individual Example Traces

FIGURE 15: ZENITH improves the convergence during planned OFC failover.

Optimizations	Time	#Distinct States	Diameter
None	> 30h	> 200M	–
Sym	10h 43m	82M	393
Sym/Com	1h 25m	11M	302
Sym/Comm/Part	3s	12K	109

TABLE 4: Our optimizations reduce the verification time from more than 30 hours to 3 s. (Sym = Symmetry, Com = Compositional verification, Part = Partial order reduction)

indicates the subtlety of the errors we encountered in the process of developing the specification.

In §D.2, we quantify the specification complexity for four components after verifying under 1) switch partial failure, 2) controller partial failure, 3) switch and controller partial failure, 4) switch complete permanent failure, and 5) and 6) switch complete transient failure (with and without reconciliation). We measure complexity using the Henry-Kafura information flow metric [27], which captures the complexity of each component and the information flow between components. In short, Sequencer is the most complex component of ZENITH-core as it has to correctly undo DAG installation after a complete switch failure before installing a new one. Monitoring Server is also complex since it must reason about partial transient switch failures. Finally, ZENITH-DR is more complex than ZENITH-NR since tracking directed reconciliation adds complexity.

7 Related Work

Network Controller Designs. Onix [33] and ONOS [12] motivate the need for a distributed software-defined control plane and provide mechanisms for distributed agreement for consistent log replication. Orion’s [19] eventually consistent network controller replaced Onix’s monolithic design with a microservice-based design, but requires periodic reconciliation. ZENITH, unlike Orion, is a model-checked controller specification robust to transient and permanent failures of both controller and switch components.

SDN Consistency Specifications. ZENITH avoids reconciliation by using formal methods to ensure controller correctness. Reconciliation can also be avoided using higher-overhead strong synchronization among control plane microservices and data plane components (e.g., distributed transactions [17, 31], consistent updates [47], or causally consistent updates using network event structures NES [39]). Other work studies customizable consistency properties for network updates [21, 30, 57] that determine safety

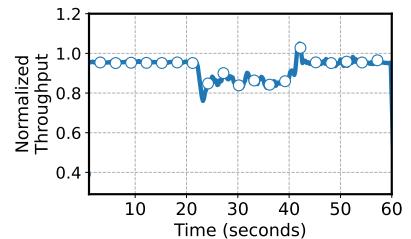


FIGURE 16: ZENITH maintains high throughput when draining (t=20) and undraining (t=40) a switch in Fat-tree.

and liveness conditions that a controller must satisfy, but typically require distributed protocols that may not scale well.

Networked Systems Verification. Verification tools have been used to test network configurations [20, 55]. SDN controller platforms, however, pose a new challenge due to network dynamics of the interaction between control and data planes. In prior work, TLA+ has been used to verify the network topology in the SDN context [50], but no prior work has explored verification of microservice-based SDN controllers.

Distributed Systems Verification. Prior work [26] has verified distributed systems by modeling them as state machines (e.g., a Mealy machine). Most closely related to our work is Anvil [53], which uses this approach to implement a formally verified distributed systems controller. There is no straightforward transformation for the existing Anvil API to describe a microservice-based SDN controller and its interaction with a switch. Moreover, ZENITH contains complex and coupled interacting microservices, which may not be amenable to a classic Mealy-machine specification.

8 Conclusions and Future Work

ZENITH avoids periodic reconciliation using a systematically model-checked specification. It exhibits an order of magnitude improvement in tail convergence when inconsistencies arise. A suite of optimizations enables us to verify its large and complex specification, and its ability to independently verify ZENITH-apps helps preserve the benefits of microservices-based controllers.

ZENITH takes a first step toward building a correct-by-construction controller, but much work remains. Future directions include supporting other switch-configuration protocols [20], incorporating other SDN applications such as those that bridge legacy protocols like BGP, and scaling verification. Another direction is to extend NADIR to generate code for these new components and to evaluate and optimize its overhead.

Our approach can be used to explore alternative microservice decompositions, helping identify configurations that optimize convergence time. Future work can also explore verifying performance properties of controllers [10, 41].

Acknowledgments. We thank the anonymous reviewers for their insightful comments. This material is based upon work supported by the U.S. National Science Foundation under grants CNS-1901523 and 2330066. Pooria Namyar was supported by the Google PhD Fellowship.

References

- [1] [n. d.]. A High-Level View of TLA+. <https://lamport.azurewebsites.net/tla/high-level-view.html>. ([n. d.]).
- [2] [n. d.]. MongoDB. <https://www.mongodb.com/>. ([n. d.]).
- [3] [n. d.]. ONOS Network Topology State Management. <https://wiki.onosproject.org/display/ONOS/Network+Topology+State>. ([n. d.]).
- [4] [n. d.]. OpenDaylight. <https://www.opendaylight.org/>. ([n. d.]).
- [5] [n. d.]. OpenDayLight Incident 1. <https://git.opendaylight.org/gerrit/c/controller/+32352?usp=search>. ([n. d.]).
- [6] [n. d.]. OpenDayLight Incident 2. <https://git.opendaylight.org/gerrit/c/bgpcep/+33697?usp=search>. ([n. d.]).
- [7] [n. d.]. The Sphere Testbed Platform. <https://sphere-project.net/>. ([n. d.]).
- [8] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal Dynamic Partial Order Reduction. *SIGPLAN Not.* (2014).
- [9] Omid Alipourfard, Jiaqi Gao, Jeremie Koenig, Chris Harshaw, Amin Vahdat, and Minlan Yu. 2019. Risk Based Planning of Network Changes in Evolving Data Centers. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery.
- [10] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. 2021. Toward Formally Verifying Congestion Control Behavior. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery.
- [11] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Moily Sagiv, Michael Schapira, and Asaf Valadarsky. 2014. VeriCon: towards verifying controller programs in software-defined networks. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Association for Computing Machinery.
- [12] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. 2014. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*. Association for Computing Machinery.
- [13] Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. 1998. Symmetry Reductions in Model Checking. In *Computer Aided Verification, 10th International Conference, CAV*.
- [14] Edmund M. Clarke, Orna Grumberg, and David E. Long. 1994. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.* (1994).
- [15] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. 1989. Compositional Model Checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89)*.
- [16] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. 2012. TLA+ Proofs. (2012). arXiv:[1208.5933](https://arxiv.org/abs/1208.5933)
- [17] Huynh Tu Dang, Daniela Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15*. ACM.
- [18] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. 2016. SDNRacer: concurrency analysis for software-defined networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. Association for Computing Machinery.
- [19] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. 2021. Orion: Google's Software-Defined Networking Control Plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*.
- [20] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association.
- [21] Klaus-Tycho Förster, Ratul Mahajan, and Roger Wattenhofer. 2016. Consistent Updates in Software Defined Networks: On Dependencies, Loop Freedom, and Blackholes. In *IFIP NETWORKING*.
- [22] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. *SIGCOMM Comput. Commun. Rev.* (2011).
- [23] Patrice Godefroid. 1990. Using Partial Orders to Improve Automatic Verification Methods. In *Computer Aided Verification, 2nd International Workshop, CAV '90*.
- [24] Ramesh Govindan, Ima Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2016. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA.
- [25] Finn Hackett, Shayan Hosseini, Renato Costa, Matthew Do, and Ivan Beschastníkh. 2023. Compiling Distributed System Models with PGo. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. Association for Computing Machinery.
- [26] Chris Hawblitzel, Jon Howell, Manos Kapritos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. Association for Computing Machinery.
- [27] S. Henry and D. Kafura. 1981. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering* (1981).
- [28] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. 2018. B4 and after: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google's Software-Defined WAN. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery.
- [29] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hözle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a Globally-deployed Software Defined Wan. *SIGCOMM CCR* (2013).
- [30] Xin Jin, Hongjiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. 2014. Dynamic Scheduling of Network Updates. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. Association for Computing Machinery.
- [31] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. 2015. Ravana: Controller Fault-Tolerance in Software-Defined Networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. Association for Computing Machinery.
- [32] Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications* (2011).
- [33] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. 2010. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association.
- [34] Umesh Krishnaswamy, Rachee Singh, Paul Mattes, Paul-Andre C Bissonnette, Nikolaj Bjørner, Zahira Nasrin, Sonal Kothari, Prabhakar Reddy, John Abel, Srikanth Kandula, Himanshu Raj, Luis Irún-Briz, Jamie Gaudette, and Erica Lan. 2023. OneWAN is better than two: Unifying a split WAN architecture. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association.
- [35] Leslie Lamport. 1994. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.* (1994).
- [36] Leslie Lamport. 2011. Byzantizing Paxos by Refinement. In *Proceedings of the 25th International Conference on Distributed Computing*. Springer-Verlag.
- [37] Jeffrey F. Lukman, Huan Ke, Ceser A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongs, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery.
- [38] Roman May, Ahmed El-Hassany, Laurent Vanbever, and Martin Vechev. 2017. BigBug: Practical Concurrency Analysis for SDN. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. Association for Computing Machinery.
- [39] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černý. 2016. Event-driven Network Programming. In *SIGPLAN Notices*.
- [40] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* (2008).
- [41] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, Umesh Krishnaswamy, Ramesh Govindan, and Srikanth Kandula. 2024. Finding Adversarial Inputs for Heuristics using Multi-level Optimization. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*.
- [42] Pooria Namyar, Behnaz Arzani, Srikanth Kandula, Santiago Segarra, Daniel Crankshaw, Umesh Krishnaswamy, Ramesh Govindan, and Himanshu Raj. 2024. Solving Max-Min Fair Resource Allocations Quickly on Large Graphs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*.
- [43] Pooria Namyar, Arvin Ghavidel, Daniel Crankshaw, Daniel S. Berger, Kevin Hsieh, Srikanth Kandula, Ramesh Govindan, and Behnaz Arzani. 2025. Enhancing Network Failure Mitigation with Performance-Aware Ranking. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association.
- [44] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardonoff. 2015. How Amazon web services uses formal methods. *Commun. ACM* (2015).
- [45] Aurojit Panda, Wenting Zheng, Xiaohu Hu, Arvind Krishnamurthy, and Scott Shenker. 2017. SCL: Simplifying Distributed SDN Control Planes. In *14th USENIX*

- Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association.
- [46] Doron A. Peled. 1993. All from One, One for All: on Model Checking Using Representatives. In *Computer Aided Verification, 5th International Conference, CAV '93*.
 - [47] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David 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)*. Association for Computing Machinery.
 - [48] Natali Ruchansky and Davide Proserpio. 2013. A (not) NICE way to verify the openflow switch specification: formal modelling of the openflow switch using alloy. *SIGCOMM Comput. Commun. Rev.* (2013).
 - [49] Leonid Ryzhyk, Nikolaj Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B. Terry, and George Varghese. 2017. Correct by Construction Networks Using Stepwise Refinement. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*.
 - [50] Vadym Shkarupyo and Olga Polska. 2018. The approach to SDN Network topology verification on a basis of Temporal Logic of Actions. In *2018 14th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET)*.
 - [51] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hözle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Sigcomm '15*.
 - [52] Rachee Singh, Monia Ghobadi, Klaus-Tycho Foerster, Mark Filer, and Phillipa Gill. 2017. Run, Walk, Crawl: Towards Dynamic Link Capacities. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets '17)*. Association for Computing Machinery.
 - [53] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. 2024. Anvil: Verifying Liveness of Cluster Management Controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association.
 - [54] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model checking TLA+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 54–66.
 - [55] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. 2014. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association.
 - [56] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C. Mogul, and Amin Vahdat. 2019. Minimal Rewiring: Efficient Live Expansion for Clos Data Center Networks. In *Proc. 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2019)*.
 - [57] Wenzuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P. Brighten Godfrey. 2015. Enforcing Customizable Consistency Properties in Software-defined Networks. In *NSDI*. 73–85.

Appendix

Appendices are supporting material that have not been peer-reviewed.

A State Inconsistency in a Real-World Controller

We study the commit history of OpenDaylight [4] to understand the root causes of state inconsistency in modern controllers. Our analysis focuses on bugs that met at least one of the following criteria:

- Are enabled by a race condition between threads within the controller.
- Are enabled by a race condition between one or more controller threads and a device being managed by the controller.
- Resulted in a deadlock among a set of threads in the controller.

To ensure relevance and impact, we only consider bugs that were fixed and committed into a released version of the controller. Additionally, we limited our analysis to modules that have direct counterparts in our controller design:

- **OpenFlow Plugin**, which corresponds to ZENITH’s Worker Pool.
- **BGP And PCEP Plugin**, which is a large and complex plugin that implements functionality similar to both the DE and Worker Pool in ZENITH.
- **NETCONF Plugin**, which is similar to the Monitoring Server and manages the distribution of updates from devices to the controller.

Outcome Of Each Bug. The commit histories include detailed steps for reproducing each bug. The bugs we selected fall into three classes:

- **State Inconsistency:** The bug causes a mismatch between the state in a data-plane element and the state recorded in the control plane.
- **Exception:** The bug triggers an exception in the thread that observes the result of the bug. If this exception causes a component to crash or requires the client to retry the operation, it can delay state consistency.
- **Deadlock:** Threads may enter a deadlock while waiting for one another to complete operations involving controller state or the network. These situations can also lead to state inconsistency.

Taxonomy Of Bugs. We identify a total of 26 race conditions. From this set:

- **9 bugs** result in state inconsistencies between the controller and the data plane, requiring reconciliation to restore correctness.
- **7 bugs** lead to some form of deadlock, with varying consequences:
 - **2 traces** traces block a connection until it times out, causing certain operations (e.g., statistics collection) to fail during that period.
 - **1 trace** trace prevents a connection from being properly cleaned up in the NIB.
 - **4 traces** prevent any response to the caller, potentially delaying operations until a timeout occurs.

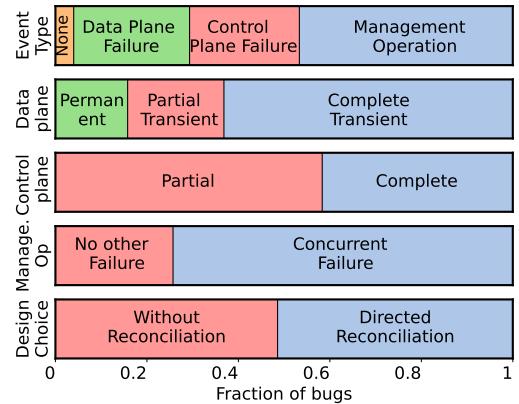


FIGURE A.1: Classification of Specification Errors.

- **10 bugs** cause exceptions within the controller.

ODL developers addressed these issues through a combination of strategies: changing the execution order (**5 bugs**); enforcing correct synchronization (**20 bugs**); and changing the implementation (**1 bug**).

B Other Controller Requirements

We also check for two conditions that help us debug our design and avoid unnecessary operations (formalisms omitted for space).

(Safety) Unnecessary OP installation. This check ensures that the controller installs an OP at most once on the target switch. However, in some cases (e.g., when a controller cannot determine if a switch received an in-flight OP before it failed or after it recovered), we cannot avoid duplicate OPs; in these cases, we relax this constraint.

(Safety) Concurrency Violation. Modern controllers use worker pools to perform certain types of tasks: e.g., translating an OP to an OpenFlow message or communicating with a switch. Any worker can work on a task taken from a task queue. This design leverages concurrency to improve throughput. We add a constraint to ensure that no two workers can work on the same task at the same time.

C Classification of Specification Errors

This section taxonomizes the dimensionality of specification errors.

Plane. We attribute a specification error to data, control, or management plane, depending on the location of failure that triggered the error (first row of Figure A.1). For example, if an error occurs when a switch fails, we attribute it to the data plane. For concurrent failures, we attribute the error to the component or microservice whose robustness we were examining. For example, if a data plane failure triggers a specification error during partial failure of a management app (e.g., drain or undrain), we would attribute the error to the management operation because it was not able to handle the data plane failure correctly.

Only 4% of our specification errors occur when designing a controller in the absence of failures. This illustrates the difficulty of ensuring robustness to failure. Control and data plane failures account for the same number of errors, but management operations

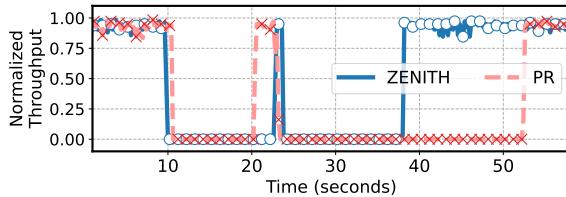


FIGURE A.2: ZENITH maintains higher throughput compared to ODL [4] on the B4 topology in our Testbed (Note that ODL is based on Java whereas ZENITH is based on Python and hence, we expect ODL to perform the same operations much faster).

account for twice this number. This makes intuitive sense: management operations need to be carefully sequenced to avoid any impact on customers. In practice, most (e.g., 70% from Google [24]) failures also occur during management operations.

Failure type. We now look at how specification errors in each plane are distributed across partial, transient, and concurrent failures.

Data plane failures. A large fraction of data plane specification errors occur due to transient failures. Designing a correct controller to handle transient failures is particularly challenging, as it requires the controller to frequently change its routing strategy. If the recovery is quick, the controller must adjust its routing in the middle of the previous change (for switch down event) and also deal with many in-flight operations. Among transient failures, complete failures are harder to manage since they cause the switch to completely lose its routing state.

Control plane failures. These occur as a result of component failures (partial) or complete microservice failures (which requires failover). Understandably, partial failures account for a majority of the errors, since correct specifications have to account for the loss of component internal state. Complete failures also account for a large fraction (nearly 40%) of the errors, since designing a correct failover is hard.

Management operations. Errors in this category have two main reasons. First, correctly orchestrating microservices to sequence management operations is inherently difficult. Second, errors occur when failures happen during the execution of these operations. Such failures are harder to handle correctly because they can occur at any point in the middle of a complex sequence. For instance, while verifying the robustness of an OFC planned failover — originally designed and verified to be correct in the absence of failures — we uncovered 26 additional specification errors, representing more than a 5× increase compared to the failure-free case.

D Extended Evaluation

In this section, we present an extended evaluation of ZENITH.

D.1 Throughput Experiments

We have also conducted a similar experiment to Figure 14, in which we compare against an open-source controller, OpenDaylight (ODL) [4] (Figure A.2). In this experiment, a complete switch failure and a partial transient failure occur concurrently. Our ODL DE app fails to clean up state, resulting in traffic being blackholed until reconciliation resolves the inconsistency. For the same set of failures, ZENITH recovers faster. In Figure A.2, ZENITH’s failure detection

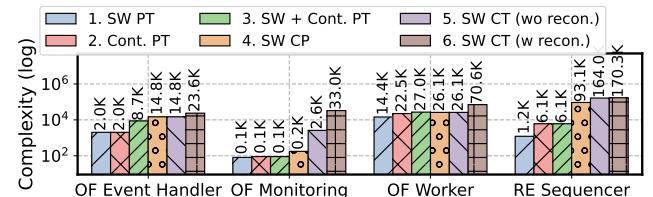


FIGURE A.3: Specification complexity. (PT=Partial Transient, CP=Complete Permanent, CT=Complete Transient)

System	Line Count
S3	804 PlusCal
DynamoDB	939 TLA+
EBS	102 PlusCal
AWS Internal Lock Manager	223 PlusCal, 318 TLA+
ZENITH (no failover)	1.8K PlusCal, 4.9K TLA+
ZENITH (with failover)	2.1K PlusCal, 6.5K TLA+

TABLE A.1: ZENITH’s specifications are much larger than prior work. For ZENITH, we show the numbers for two of our specifications (one that contains our failover logic and one that does not). The numbers for other systems are from [44].

time is set to match that of ODL, so it takes longer to recover from the failure than in Figure 14.

D.2 Specification Complexity

Figure A.3 shows the software complexity of the specifications of four of the components after verifying ZENITH-core for (1) switch partial failure, (2) controller partial failure, (3) switch and controller partial failure, (4) switch complete permanent failure, and (5) and (6) switch complete transient failure (with and without reconciliation). We measure complexity using the Henry-Kafura information flow metric [27], which captures the complexity of each component and the information flow between components.

Sequencer is most complex since it must sequence OPs in the DAGs and manage the transition from one DAG to another in response to network events, while considering all in-flight OPs. Sequencer complexity increases significantly after verifying switch complete permanent failures, since we had to introduce extra mechanisms to correctly manage transitions between DAGs.

Monitoring Server complexity increases after verifying switch complete transient failures. To deal with these failures, Monitoring Server needs to check acknowledgments at the granularity of flows instead of OPs. This is because DE schedules some OPs to install new flows when the switch fails. Later, when the switch recovers, these flows need to be cleaned up using some other OPs. As a result, we not only need to keep track of the OPs but also their actions.

E Specification of a ZENITH App

We present the PlusCal specification of an SDN application to illustrate both the process and the practicality of this approach. The application schedules switch updates to *drain* a switch (*i.e.*, remove it from the network) given the current topology and active paths in the network. Operators require drains to be *hitless*; they should not result in traffic drops caused by incomplete network state.

Our specification of this *hitless drain* app is on the order of a hundred lines of PlusCal and TLA+. Cloud providers like Amazon

LISTING 4: Drain Application Process

```

1 process drainer \in {{app0} \X {DRAIN_APP}}
2 variables
3   /* Process local variables
4   currentRequest = NADIR_NULL, currentTopology = NADIR_NULL,
5   nodeToDrain = NADIR_NULL, currentPaths = {},
6   currentOPs = {}, endpoints = {},
7   pathsAfterDrain = {}, drainPathSetOPs = {},
8   drainedDAG = {}, nextDAGID = 1;
9 begin
10 DrainLoop:
11 while TRUE do
12   /* Wait until a new drain request comes in
13   FIFOGet(ManagerInterface, currentRequest);
14   /* Read the contents of the request
15   InspectDrainRequest(currentRequest, currentTopology,
16   currentPaths, nodeToDrain, currentOPs);
17 ComputeDrain:
18   /* First, get the set of endpoints that we need to
19   /* keep connected during/ after drain
20   endpoints := getPathSetEndpoints(currentPaths) \
21   {nodeToDrain};
22   /* Compute the new paths
23   pathsAfterDrain := ShortestPaths(
24     endpoints,
25     (currentTopology.Nodes \ {nodeToDrain}),
26     RemoveNodeFromEdgeSet(nodeToDrain,
27     currentTopology.Edges));
28   /* Compute the new DAG and OPs that implement the paths
29   call ComputeDrainDAG(pathsAfterDrain, currentOPs,
30   drainPathSetOPs, drainedDAG);
31 SubmitDAG:
32   /* Generate an ID for this DAG and send it to the core
33   FIFOPut(DAGEventQueue, [id -> nextDAGID, dag -> drainedDAG]);
34   nextDAGID := nextDAGID + 1;
35 end while;
36 end process;

```

have written larger specifications (up to 800 lines) and used them to verify several deployed systems [44].

Preliminaries. For clarity, we color **keywords** dark blue, **constants** purple, **Operator/Macro calls** cyan, **global variables** orange, **Procedure calls** brown, and **process labels** red. The constant **NADIR_NULL** is a reserved name for NADIR that mocks a typical null-like value.

We specify each application as a process. A process is an independent thread of execution in PlusCal and must have a unique identifier. Listing 4 shows the process for the **drainer** and is identified by the tuple «**app0**, **DRAIN_APP**». Each process can define and initialize its own local variables (**currentRequest** in Line 4).

Each process can also access and modify *global variables*. Listing 4 uses two global variables: **ManagerInterface** (Line 13) and **DAGEventQueue** (Line 33). These variables capture communication between specifications, using message queues between processes. For example, **drainer** uses the **DAGEventQueue** to submit its final DAG to **ZENITH-core**. Similarly, management software (not shown as part of this

specification) can use **ManagerInterface** to submit their drain requests to the **drainer**.

LISTING 5: Drain Application Global Variable Declarations

```

1 variables
2   /* ----- CORE GLOBAL VARIABLES -----
3   /* Queue of DAGs from applications to the controller
4   DAGEventQueue = <>>>;
5
6   /* ----- APPLICATION GLOBAL VARIABLES -----
7   /* This is the queue of drain requests .
8   /* A drain request has 4 parts :
9   /* - The current topology (i.e. set of running switches and
10  /* links)
11  /* - The set of paths active in the network (i.e. current
12  /* routing configuration in the network)
13  /* - The set of OPs that implement the paths in the previous set
14  /* - The node index to drain
15  /* DrainRequestQueue = <>>>

```

Each PlusCal process consists of a sequence of atomic steps, defined between labeled blocks. In Listing 4, the code between the label **ComputeDrain** (Line 17) and Line 30 is an atomic step that computes the DAG that drains the network.

Drainer Application Logic. Given a drain request, the drain application (Listing 4) performs the following steps:

- (1) Get the set of endpoints (Line 20) that must *remain* connected and be able to exchange traffic both during *and* after the drain procedure.
- (2) Compute the set of viable paths for each pair of endpoints, assuming the target node for draining has been completely removed from the network (Line 23).
- (3) Generate a DAG that installs the new paths and safely removes the old ones without causing traffic drop (Line 29).
- (4) Submit the DAG to the **ZENITH-core** for installation (Line 33).

Logic for Generating DAG. Central to the drain application is generating a DAG that can safely install the new paths and remove the old ones. Listing 6 shows the logic for this part, encapsulated in a procedure named **ComputeDrainDAG**. In PlusCal, procedures are a series of atomic steps (*i.e.*, statements between labels) that are executed by a process (Line 29 in Listing 4).

The **ComputeDrainDAG** procedure constructs a DAG that ensures new paths are installed and fully operational before removing the old ones. This guarantees that the drain is hitless, even during the transition. To achieve this, the procedure (1) ensures all the OPs that remove the old paths are installed after all the OPs that install the new ones (Line 38), and (2) assigns higher priority to the new paths (Lines 13 and 24). As a result, once the new paths are installed, switches fully shift their traffic onto them, making it safe to remove the old paths.

Operator and Macro Calls. We use the operator **HighestPriorityInOPSet** (Line 13 in Listing 6) as an example to illustrate how operators are specified in Listing 7. This operator computes the maximum priority among all the OPs in the previous paths. It is possible to specify this in PlusCal using logical quantifiers, which makes it simpler. However, we implement it using TLA+ recursion, as it speeds up model checking.

LISTING 6: A Procedure to Compute the Drain DAG

```

1 procedure ComputeDrainDAG(newPaths, previousOPs, newOPs, newDAG)
2 /* A procedure can define its own local variables , just like a process
3 variables nextPriority = NADIR_NULL,
4     newOPsStartingIndex = NADIR_NULL, newOPSet = {},
5     newDAGEdgeSet = {}, newDAGNodeSet = {},
6     currentPath = NADIR_NULL, currentPathResult = {};
7 begin
8 ComputePriority:
9     /* The new OPs MUST be installed with a higher priority than
10    /* previous paths (otherwise, there is no way to guarantee the
11    /* drain is hitless ). To do this , sift through previous OPs and
12    /* get the highest priority in it .
13    nextPriority := HighestPriorityInOPSet(previousOPs) + 1;
14    /* Each hop in a new path will have a new OP associated with it .
15    /* Each OP needs an index, and so will start at the end of
16    /* the current OP list for indexing.
17    newOPsStartingIndex := Cardinality (previousOPs);
18
19 ComputeNewPathsDAG:
20 while Cardinality (newPaths) > 0 do
21     /* Pick one of the new paths
22     currentPath := (CHOOSE p \in newPaths: TRUE);
23     /* Compute the OPs and the ordering that implements the new path
24     currentPathResult := ComputeSinglePathDAG(currentPath,
25                                                 newOPsStartingIndex,
26                                                 nextPriority );
27     /* Combine the results and remove the processed path from the set
28     newOPSet := newOPSet \cup currentPathResult.ops;
29     newDAGEdgeSet := newDAGEdgeSet \cup currentPathResult.edges;
30     newDAGNodeSet := newDAGNodeSet \cup currentPathResult.nodes;
31     newPaths := newPaths \ {currentPath};
32 end while;
33
34 CleanupPreviousOPs:
35     /* To cleanup the previous OPs, add the deletion instruction for
36     /* all the previous OPs at the end of the DAG (i.e. attach them
37     /* to all the leaves of the DAG)
38     newDAG := ExpandDAG([v -> newDAGNodeSet, e -> newDAGEdgeSet],
39                         GetDeletionOPs(previousOPs));
40 end procedure;

```

LISTING 7: An Operator to Compute the Highest Priority

```

1 RECURSIVE _HighestPriorityInOPSet(, _)
2 _HighestPriorityInOPSet (setOfOPObjects, priority ) ==
3     IF Cardinality (setOfOPObjects) = 0
4         THEN priority
5     ELSE
6         LET
7             currentOPObject == CHOOSE x \in setOfOPObjects: TRUE
8             currentPriority ==
9                 IF priority < currentOPObject. priority
10                    THEN currentOPObject.priority
11                    ELSE priority
12         IN
13             _HighestPriorityInOPSet (
14                 setOfOPObjects \ {currentOPObject},
15                 currentPriority )
16     HighestPriorityInOPSet (setOfOPObjects) ==
17         _HighestPriorityInOPSet (setOfOPObjects, 0)

```

We omit the specifications of other **Operator/Macro calls** for brevity. For example, **ShortestPaths** uses a recursive breadth-first search to compute shortest paths between specified pairs. The macros **FIFOPut** and **FIFOGet** specify queue operations.

NADIR Type Annotations. All of the previous listings describe a specification that can be model-checked with TLC. However, NADIR cannot process them as-is, since they lack the necessary type annotations. The type annotations for Listing 4 are the following:

LISTING 8: Drain Application Type Annotations for NADIR

```

1 STRUCT_SET_RC_DAG == [
2     v: SUBSET NADIR_INT_ID_SET,
3     e: SUBSET (NADIR_INT_ID_SET \X NADIR_INT_ID_SET)]
4 STRUCT_SET_DAG_OBJECT == [
5     id: NADIR_INT_ID_SET,
6     dag: STRUCT_SET_RC_DAG]
7 STRUCT_SET_OP_OBJECT == [
8     priority : Nat,
9     sw: SW,
10    op: NADIR_INT_ID_SET]
11 STRUCT_SET_TOPOLOGY == [
12     Nodes: SUBSET Nat,
13     Edges: SUBSET (Nat \X Nat)]
14 STRUCT_SET_DRAIN_REQUEST == [
15     topology: STRUCT_SET_TOPOLOGY,
16     paths: Seq(Nat),
17     node: Nat,
18     ops: SUBSET STRUCT_SET_IR_OBJECT]
19 /* This aggregator defines the name of structs during code generation
20 NadirStructSet == ("StructRCDAG" :-> STRUCT_SET_RC_DAG)
21           ("StructDAGOobject" :-> STRUCT_SET_DAG_OBJECT)
22           ("StructOPObject" :-> STRUCT_SET_OP_OBJECT)
23           ("StructTopology" :-> STRUCT_SET_TOPOLOGY)
24           ("StructDrainRequest" :->
25             STRUCT_SET_DRAIN_REQUEST)
25 /* The type annotation. It is also an invariant of the specification
26 TypeOK == /\ NadirFIFO(
27             STRUCT_SET_DAG_OBJECT, DAGEventQueue)
28             /\ NadirFIFO(
29                 STRUCT_SET_DRAIN_REQUEST, DrainRequestQueue)
30             /\ NadirLocalVariableTypeCheck(
31                 NadirNullable(STRUCT_SET_DRAIN_REQUEST),
32                 currentRequest)
33             /\ NadirLocalVariableTypeCheck(
34                 NadirNullable(STRUCT_SET_TOPOLOGY),
35                 currentTopology)
36             /\ NadirLocalVariableTypeCheck(
37                 SUBSET Seq(Nat), currentPaths)
38             /\ NadirLocalVariableTypeCheck(
39                 NadirNullable(Nat), nodeToDrain)
40             /\ NadirLocalVariableTypeCheck(
41                 SUBSET STRUCT_SET_OP_OBJECT, currentOPs)
42             /\ NadirLocalVariableTypeCheck(
43                 SUBSET Nat, endpoints)
44             /\ NadirLocalVariableTypeCheck(
45                 SUBSET Seq(Nat), pathsAfterDrain)
46             /\ NadirLocalVariableTypeCheck(
47                 SUBSET STRUCT_SET_OP_OBJECT, drainPathSetOPs)
48             /\ NadirLocalVariableTypeCheck(
49                 SUBSET STRUCT_SET_RC_DAG, drainedDAG)
50             /\ NadirLocalVariableTypeCheck(
51                 Nat, nextDAGID)

```

The type annotation describes the type of all the variables defined within the specification for NADIR. Users can introduce C-like structs to NADIR. For example, Line 7 describes what an OP object looks like. It is a struct that has 3 fields:

- `priority` determines the priority of this OP. TLA+ uses `Nat` to describe the set of natural numbers, so `priority` is just a number as well.
- `sw` represents the corresponding switch. It is defined as a member of the constant set `SW`, which itself is defined as part of ZENITH-core. `SW` is the set of all the switches that we would ever have in the network.
- `op` is a unique identifier. The set `NADIR_INT_ID_SET` is a special name reserved by NADIR. It means that this value is a *pointer* to some other struct. In our case, this struct would be an OpenFlow packet during runtime.

F Proof of ZENITH-core Correctness

We now **prove** that our ZENITH-core specification satisfies the three correctness conditions listed in §3.3. The proof proceeds in two steps. First, we identify a set of properties and prove these properties are sufficient to ensure a controller specification is correct (§F.2). Then, we use the TLA proof system to show that ZENITH-core's specification satisfies these properties and is therefore correct (§F.3).

F.1 Assumptions

We make the following assumptions:

- A1. Switch and controller processes are at least weakly fair [35].
- A2. NIB operations are atomic and consistent. The NIB never loses its state or undergoes any failure of any kind.
- A3. Switches acknowledge OPs if and only if they have completed them correctly. Furthermore, switches process requests one at a time and completely and correctly wipe the TCAM if requested by the controller and eventually generate failure/recovery events.
- A4. The NIB and the network are initially consistent.

We discuss why these assumptions are practical/necessary.

A1: Weak fairness asserts that if a process is continuously able to take a certain action, then the action must occur eventually. This is a standard assumption for distributed systems [53, 54]. We also assume switches are weakly fair, which ensures that switches *eventually* acknowledge messages and produce events in case of failure.

A2: NIB implementations that use modern replicated database systems (e.g., MongoDB [2]) provide atomicity, consistency, and durability.

A3 ensures that ZENITH can always converge to the dataplane state by (1) starting from some known state that is consistent with the switch state (e.g., an empty TCAM), and (2) processing acknowledgments from the switches. In practice, some switches may *aggregate* acknowledgments (e.g., OpenFlow), we disallow this for simplicity.

A4 formalizes the need for a known starting state of the system.

F.2 Proof of Correctness

We first state several properties, then use these to prove that any controller that satisfies these properties must be correct. In §F.3,

we use the TLA+ proof system [16] to show ZENITH-core follows these properties.

Terminology. We say an OP is *in-flight* when it has been emitted by the controller and is in transit to the corresponding switch. An OP is *deemed in-flight* when the controller believes it is still in transit, regardless of its actual status. We similarly differentiate between an *installed* OP and an OP *deemed installed*.

Properties:

- P1. Each process is *always eventually* able to process a request.
- P2. The controller processes an OP if and only if all its preceding OPs in the DAG have been installed and the controller is aware of this.
- P3. The controller can successfully install a single OP on a healthy switch. It marks the OP as in-flight before sending it to the switch, and always updates the NIB upon receiving the corresponding ACK.
- P4. The controller guarantees (1) in-order delivery of OPs to each switch, and (2) in-order processing of OP ACKs from each switch.
- P5. The controller treats every DAG the same, regardless of its size and structure.
- P6. Upon detecting the recovery of a switch from failure, the controller schedules OPs to clean up the switch state, and subsequently removes the routing state from the NIB upon receiving the ACK. These OPs go through the same pipeline as the rest of the OPs and follow property P4.
- P7. Upon detecting a failure, the controller does not update the state of any affected OPs in the NIB and does not send any new OPs until the switch has recovered and the cleanup operation has been acknowledged. The instruction to clear a switch is an exception, and the controller can send it at any time.
- P8. The controller eventually detects switch failure/recovery and updates the NIB to reflect the health status of the switch.

Before proceeding, we reiterate the correctness conditions we impose on ZENITH-core:

- ① ZENITH-core never violates DAG OP dependencies.
- ② Switch routing state eventually matches intended DAGs.
- ③ ZENITH-core's view of routing state eventually matches each switch's routing state.

Single DAG. We first consider the case where a single application wants to install a single DAG, and all involved switches remain healthy throughout the process.

THEOREM F.1. *A controller that satisfies P1 – P8 is correct under any single DAG.*

PROOF. We prove this by contradiction. Suppose the controller violates at least one of the three correctness properties (§3.3). One of the following cases must occur:

Case 1: The controller violates DAG OP dependencies (①). This means the controller processes an OP even though at least one of its preceding OPs in the DAG has not been installed. This contradicts property P2 of the controller and, therefore, cannot occur.

Case 2: There exists an OP such that the controller either (i) fails to install it (violating ②), or (ii) installs it but is unaware

of its installation (violating (3)) – despite all OP dependencies being satisfied. By properties P1 and P2, the controller will eventually begin processing this OP once its dependencies are satisfied. At that point, the OP can be treated as an independent DAG (i.e., a subgraph of size 1). Failure to install such an OP or to update the NIB contradicts property P3 or P5. Therefore, it cannot occur.

Since both cases lead to contradictions, the controller must be correct if it satisfies P1 – P8. \square

DAG transitions. We now consider the case where a single application issues multiple updates to the DAG, and all switches involved in the final DAG remain healthy once the final DAG is scheduled, but other switches may fail and recover.

In this setting, correctness means that:

- The controller installs the final DAG, respecting the correctness properties in §3.3;
- Its view of the routing state *eventually* reflects both the final DAG and any OPs from earlier DAGs that were not explicitly cleaned up by subsequent updates or switch recovery;
- At any point in time, there must not exist an OP associated with a healthy switch that is in-flight or installed, but not deemed as either by the controller; and
- The controller eventually detects switch failure/recovery and updates the NIB to reflect the health status of the switch (property P8).

The latter two conditions ensure that the application has full visibility into the network state and can decide whether to retain or remove any installed or in-flight OP when updating the DAG.

THEOREM F.2. *At any point in time, a controller that satisfies P1 – P8 correctly reflects the status of all the OPs; that is, there must not exist an OP associated with a healthy switch that is in-flight or installed, but not deemed as either by the controller.*

PROOF. We prove this by contradiction. Suppose such an OP exists. We need to analyze two cases based on the state of the corresponding switch:

Case 1: The switch remains healthy. By property P3, the controller marks an OP as in-flight before sending it to the switch. This means the OP is deemed in-flight. If the switch remains healthy, the controller continues to deem the OP as in-flight or installed, unless it is explicitly overwritten by another OP from a subsequent DAG. In that case, the controller updates the NIB only after the new OP is installed and its ACK is received (property P3). This ensures that the period during which the controller deems the OP as in-flight or installed is at least as long as the period during which it is actually in-flight or installed.

Case 2: The switch fails and then recovers. By property P7, the controller does not send any new OPs after detecting the failure and until after the switch is cleaned up. Therefore, the controller must have sent the OP before detecting the failure. By property P7, the controller does not modify the state of such OPs upon detecting the failure. After the switch recovers, properties P4 and P6 ensure that the switch receives the cleanup OP after all in-flight OPs, guaranteeing that the switch processes it only after completing

those earlier operations. Consequently, the controller receives the ACK for the cleanup OP only after the switch has been fully cleaned up. This sequencing ensures that the controller continues to deem an OP as in-flight or installed for at least as long as it is actually in-flight or installed on the switch.

This means such an OP cannot exist, completing the proof. \square

REMARK. *Theorem F.2 guarantees that the switch is completely empty before the controller begins sending new OPs. This follows from the fact that the controller only issues new OPs after receiving the clean-up acknowledgment and after removing the switch state in NIB. If the switch were not completely empty, there should exist an OP present in the switch that the controller does not deem as in-flight or installed, contradicting Theorem F.2.*

THEOREM F.3. *A controller that satisfies P1 – P8 is correct under any arbitrary sequence of DAG transitions.*

PROOF. We prove this by contradiction. Suppose the controller violates at least one of the correctness properties. One of the following cases must occur:

- Case 1: The controller fails to install the final DAG and update the NIB. This directly contradicts Theorem F.1, which guarantees that the controller can correctly install any individual DAG and update the NIB.
- Case 2: The switch state does not reflect the final DAG since a stale OP from a previous DAG overwrites an OP from the final DAG on a switch. By property P4, OPs from the final DAG are delivered to each switch after any OPs from earlier DAGs. Consequently, OPs from the final DAG will overwrite any prior state, and the final switch state will always respect the final DAG. Note that the switch may retain some OPs from previous DAGs if the final DAG does not explicitly overwrite or delete them. However, this does not violate correctness, as the application is responsible for correctly scheduling the DAGs.
- Case 3: The controller's view of the routing state does not eventually match the switch state because the controller overwrites the state of an OP from the final DAG to reflect the installation of a stale OP. This contradicts property P4, which guarantees both in-order installation of OPs and in-order processing of OP ACKs from each switch. Therefore, this cannot exist.
- Case 4: The controller's view of the routing state does not eventually match the switch state because the controller does not correctly reflect the state of an OP from an earlier DAG that was not cleaned up by subsequent DAGs. Such an OP should belong to a switch that has remained healthy since the installation of the OP (property P6). Since the switch has remained healthy, the controller should have eventually received the ACK and should have updated the NIB (property P3). Thus, this case also can not happen.

All these cases lead to a contradiction. Combined with property P8 and Theorem F.2, this establishes that a controller satisfying Properties P1 – P8 guarantees correctness across arbitrary DAG transitions. \square

Concurrent DAGs. In cases where multiple applications submit concurrent DAGs, a controller that satisfies properties P1 – P8 remains correct as long as the DAGs do not contain conflicting OPs (e.g., one DAG installs a routing entry while another attempts to delete the same entry). In the absence of such conflicts, we can treat each DAG independently, and our proof still applies. If the DAGs do contain conflicting OPs, we require applications to de-conflict outside ZENITH-core. It may also be possible to design controllers that are correct in the presence of conflicting DAGs, but this will require first revisiting correctness conditions in the presence of conflicts.

F.3 Proof of Properties

To complete our proof of ZENITH-core correctness, we must show that the listed properties P1 – P8 hold for our specification of ZENITH-core. We do this by writing a proof for each property² in the TLAPS [16] language, and machine-checking the proof for our PlusCal specification of ZENITH-core. In the paragraphs below, we provide a natural language description of the proof.

Note About TLAPS: At the time of writing, TLAPS does not support certain constructs. Most notably, it cannot directly prove certain temporal properties with the current implementation of the toolbox. We address this limitation by introducing *history variables* into the specification. These auxiliary variables record the events that happen for each behavior. For example, to prove that queue operations preserve the order of insertion, we construct a history variable that records the sequence of dequeued objects. Although history variables can make the specification unwieldy, they allow TLAPS backends to work. Our use of these variables is inspired by prior work [36].

Terminology. We say an OP is "in progress" from the instant it is scheduled by the Sequencer until either the Sequencer receives a notification that it has been installed or the corresponding switch fails.

We will now describe how ZENITH-core satisfies each of the properties P1 – P8.

P1: For the OFC processes, property P1 follows from assertion A1 (*i.e.*, weak fairness). This is because our OFC specifications contain no AWAIT statements except for blocking when awaiting new requests. If queue operations and particularly switch queue operations are fair and an unprocessed request exists, it will eventually appear on a queue and unblock the corresponding OFC process.

The proof is more involved for the Sequencer. Unlike OFC processes, the Sequencer has an additional AWAIT statement. It waits for the acknowledgment of previously issued OPs before it can schedule new OPs with satisfied dependencies. To prove that the Sequencer never deadlocks, we must show that this AWAIT statement eventually unblocks until the Sequencer is done with the DAG.

To prove this property, we state an *Inductive Invariant* of ZENITH:

THEOREM F.4. *For each non-empty DAG being processed, at least one of the following is true at all times:*

²The one exception is P5, discussed below.

- (1) *The DAG is finished AND the DAG is stable (meaning that there are no in-progress OPs that overwrite any operation associated with it).*
- (2) *There exists at least one schedulable OP (i.e., an OP with satisfied dependencies that has not yet been processed by the Sequencer).*
- (3) *There exists at least one in-progress OP associated with the current DAG.*
- (4) *A switch that has an associated OP in the DAG fails.*

PROOF. We will prove this by induction. The theorem holds for the initial state according to A4 and the definition of a DAG. The DAG is non-empty and thus must have at least one root node without any dependencies. This OP is our first schedulable OP and clearly satisfies the second object of the disjunction that we stated.

Assume that the invariant has held up to some particular state, we now show that it must hold for the next state as well by contradiction. Assume that all of the statements have failed to hold, thus it must be the case that all the associated switches are healthy, and the following conjunction is true:

- There are no schedulable or in-progress OPs; ZENITH-core and the switches have processed every single one of them.
- The DAG is not finished *OR* an OP that overwrites it is still in-progress.

Since there are no in-progress OPs, we can refine the second case to assert that the DAG has failed to finish. At the same time, there are no schedulable OPs. So, it must be the case that there exists an OP in the DAG that has been in progress but has not finished, which contradicts the assertion that all in-progress OPs have been processed. \square

Using Theorem F.4, we can show that DE never deadlocks. As established earlier, Weak Fairness guarantees that OFC eventually sends all in-progress OPs, and as we later demonstrate (proof of P3), it does so correctly. As a result, all in-progress OPs will eventually be installed. Therefore, it must eventually be the case that either the DAG is finished and stable, or there exists a schedulable OP. In both cases, the Sequencer's AWAIT statement unblocks, allowing it to either keep processing the DAG by scheduling a new OP or announcing the DAG complete.

REMARK. *It may seem that combining the fact that neither the OFC or DE ever deadlock with the statement of Theorem F.4, would imply that eventually a DAG would have to be finished and stable because all in-progress and schedulable OPs eventually finish.*

This is only true if no switch associated with the DAG ever fails. Otherwise, any in-flight OPs heading toward it would fail too. This prompts the Sequencer to schedule it again eventually, and hence ZENITH-core will keep trying to send an OP to a switch that is dead. Clearly, this DAG cannot finish, and hence the applications must change the DAG.

P2: Each OP goes through a pipeline of modules in order to reach the designated switch. In particular, DE begins scheduling an OP through the Sequencer module, and then hands it over to the Worker Pool in OFC, which will then transmit it to the switch if and only if the NIB records the switch as healthy at the time of inspection. We will prove below (as part of property P4) that this pipeline keeps the scheduling order, and as such, to prove P2, it suffices to assert that

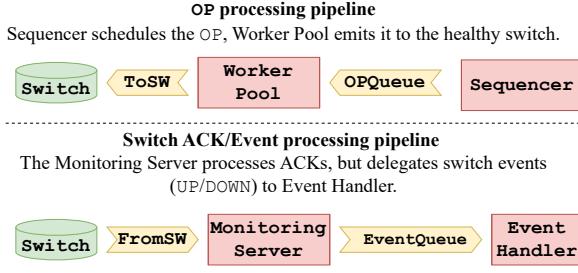


FIGURE A.4: ZENITH-core processing pipeline.

DE only begins scheduling an OP, if and only if its dependencies in the DAG were recorded as satisfied. (Note that P2 is an end-to-end assertion about ZENITH-core, but using P4, we are showing that it can be satisfied by only proving a property about one particular module, the DE).

This requirement is satisfied *verbatim* in the Sequencer module, where the set of schedulable OPs at each instant is defined as the set of all OPs that: (a) are a member of the current DAG, (b) are not in-progress or installed, and (c) have all of their dependencies finished. Clearly, this satisfies our requirement, and combined with our proof of P4, P2 is satisfied as well.

P3: P3 is closely related to P1. At its core, it requires that:

- If OFC has received the request to install an OP, and the switch remains healthy, then eventually the OP *must* always be installed.
- If an unprocessed acknowledgment exists, then the controller must eventually process it.
- The OFC must mark the OP as in-flight *before* actually emitting it towards the switch.

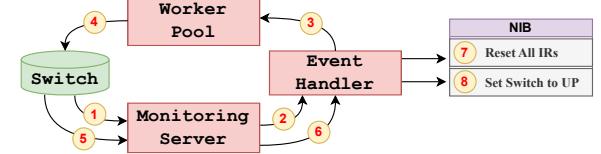
We can refer to Figure A.4 to formulate the first two requirements in TLA⁺ terms:

- If the length of `OPQueue` is greater than 0, then the Worker Pool action is enabled.
- If the length of `FromSW` is greater than 0, then the Monitoring Server action is enabled.

Both of these follow immediately from A1. If any of these two requirements fail, then the action that dequeues the OP or ACK from the associated queue has not been taken, despite being continuously enabled, violating Weak Fairness. Thus, the actions that send OPs to working switches, and collect ACKs in OFC are enabled when needed. We may finally invoke A3 in order to make sure switches process OPs correctly and return the ACK once some OP has been received.

We now only need to show the correctness of steps taken individually by modules in OFC. The main requirements in order to satisfy P3 would be the following:

- The Monitoring Server must always record any ACKed OP as finished, regardless of its current state. Per A3, switch only sends correct information (although it can be stale).
- The Worker Pool must record the OP as in-flight before sending it (as stated directly in P3). This is required since not doing so may put the Worker Pool in a race with the Monitoring Server in case the ACK arrives very quickly.



Transitioning a switch from DOWN to UP

- ① The switch emits a keep alive as a recovery event to the Monitoring Server.
- ② The Monitoring Server forwards the event to the Event Handler.
- ③ Event Handler issues a `CLEAR_TCAM` request by putting it on the `OPQueue`.
- ④ The Worker Pool eventually forwards the instruction to the switch.
- ⑤ The switch handles the instruction and clears the TCAM of any OPs. The ACK is forwarded to the Monitoring Server.
- ⑥ Event Handler receives the ACK via Monitoring Server.
- ⑦ Event Handler now first resets all OPs associated with the recovered switch.
- ⑧ Event Handler finally marks the switch as `UP`, completing the process.

FIGURE A.5: Procedure for returning a previously dead switch per P5.

Both of these requirements can be satisfied *verbatim*, as they only describe individual module steps. Listing 3 shows how this can be done for Worker Pool.

P4: Part (2) is an assertion of a queue property (in particular, it is the assertion that `FromSW` is actually a queue). The specification of a queue is a standard TLA⁺ construct and we make use of it in our specification of ZENITH as well. Part (1) is slightly more complicated, since OPs traverse 2 queues to reach a switch, those being `OPQueue` and `ToSW`. If the Worker Pool was single threaded and processed only one object at a time, then the requirement would be trivial.

Assuming queues guarantee that dequeue ordering agrees with enqueue ordering, ZENITH satisfies the second requirement for a multi-threaded Worker Pool by ensuring the following:

- Each individual thread in Worker Pool works on only a single OP at a time.
- Each thread in Worker Pool only works on OPs destined to a predefined and fixed set of switches. Essentially, switches are *consistently sharded* among threads such that each switch maps exactly to one thread in the Worker Pool.

P5: This property is difficult to express with the tools provided by TLAPS. We have verified by inspection that ZENITH-core components do not distinguish DAGs by size or shape. Specifically, we inspected each boolean expression in OFC and Sequencer, and verified they only reference individual OPs, not the DAG as a whole.

P6 and P7: P6 describes the procedure to return a previously disabled switch to the topology. In Figure A.5, we detail the procedure as demanded by the statement of P6. Similarly, P7 describes the

procedure for handling switch failure. Both properties can be satisfied verbatim in the specification, but we find it helpful to highlight why these steps are necessary.

First, it is crucial to note that the `CLEAR_TCAM` request in step 3 *MUST* traverse the Worker Pool (*i.e.*, the Topo Event Handler cannot forward the instruction directly to the switch). Not doing so would enable a race between the OPs sent by the Worker Pool, and the `CLEAR_TCAM` instruction sent by Topo Event Handler.

Second, the requirement of P7 would mean that the controller must check the switch state in the NIB before attempting to forward any OP to it. In our specification, Worker Pool handles this responsibility. Combined with atomicity of NIB writes per A2, the Worker Pool would prevent forwarding OPs to a switch as soon as the Event Handler suspends it. This explains why the `CLEAR_TCAM` instruction is exempt from the same rule.

These observations, combined with P4, imply that the switch receives the `CLEAR_TCAM` instruction after all the in-flight OPs, making sure that the TCAM is empty in the end (as discussed in the remark for Theorem F.2).

P8: The Event Handler is the only module that may change the health status of a switch, thus P8 can be satisfied by making sure that:

- ① If the Event Handler receives a switch failure event from the Monitoring Server, it immediately sets the switch state to DOWN.
- ② The Event Handler sets the switch state to UP only after resetting all OPs for the recovering switch (*i.e.*, the step ordering in Figure A.5 is honored).

If the Event Handler obeys these rules, then P8 must inductively hold as follows; Per A4, the property clearly holds for the initial state of the controller. Now assume that there exists some next state where the property is violated. For this to be the case, all switch failure/recovery events must have been processed and there must exist a switch such as S^* whose actual state is not in consensus with what has been recorded on the NIB and continuously remains so. Naturally, this would allow for only 2 cases:

Case 1: S^* is dead, yet the NIB records it as UP. Per A3, the switch must have generated a failure event and per P1, the controller must have eventually forwarded it to the Event Handler. If the switch remains continuously DOWN on the NIB, then the Event Handler must have failed to set the state to DOWN, violating ①.

Case 2: S^* is healthy, despite being recorded as DOWN. Since all events have been processed, the recovery event for the switch must have already been received by the Event Handler and the cleanup procedure per Figure A.5 must have been initiated as well. The switch has remained healthy, and per P1, it must have eventually received the request to clear the TCAM, and per A3, it must have executed and then acknowledged it. Invoking P1 again would mean that the ACK must have been received by the Event Handler. Thus, if the switch remains continuously DOWN on the NIB, then the Event Handler must have violated ② and failed to set the switch state to UP.

Note that the request to clear the switch or its acknowledgment may be lost, if and only if the switch has failed again after recovery.

In this case, the property still remains true, as the state still remains as DOWN in the end.

G Example of a TLA+ trace

Figure A.8 shows an example trace that is a simplified version of one of our TLA+ traces that caused a violation of consistency requirements in our design of ZENITH. As described in §1, such inconsistencies can lead to packet drops and, potentially, blackholed traffic.

In this example, a switch (SW-A) fails but recovers after a short duration (*e.g.*, transient power outage). Both failure and recovery events propagate through the controller until they reach *AbstractApp*. In response, *AbstractApp* schedules a new DAG that installs a new flow rule (OP1) on the recovered switch. The OP1 is eventually forwarded and installed on SW-A, and the Sequencer stops working on it as the DAG is successfully in place.

During all these steps, Topo Event Handler was computing all the necessary changes in response to the switch recovery event. This includes resetting the state of all the OPs that the controller observes as in flight at that point (which includes OP1). This process rewrites the state of the NIB and causes an inconsistency. NIB thinks the OP is not installed while the OP is in fact installed on SW-A. **This is an example of an inconsistency leading to the hidden flow entry in Figure 2.**

Note that even controllers that reconcile the state with recovered switches (*i.e.*, switch up reconciliation) are not robust to this trace as the reconciliation can happen before the OP1 is installed.

The TLA+ trace that demonstrates this inconsistency involves 64 steps and 3 switches, close to the median trace length (Figure A.6).

Solution (careful ordering of operations within a component as discussed in §3.9). In the initial design leading to this trace, Topo Event Handler first updates the state of the topology and then modifies the state of OPs. This was correct under SW Partial Transient and SW Complete Permanent failures (Table 3). However, verifying this approach under SW Complete Transient Failure triggered this issue, and we fixed it by changing the order of operations (first, change the state of OPs and then update the topology).

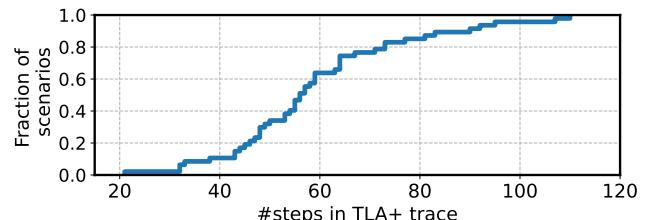


FIGURE A.6: The number of steps in TLA+ traces that caused some safety/liveness violation during our design of ZENITH.

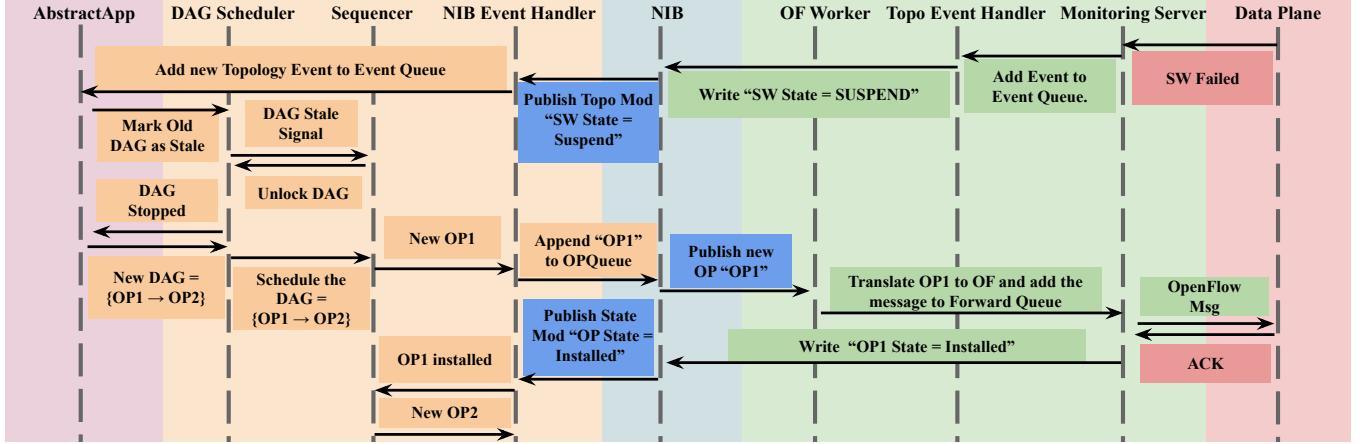


FIGURE A.7: A simplified illustration of how ZENITH’s components interact when a switch fails. For readability, this example simplifies the interactions. (Green: OFC, Blue: NIB, Orange: DE, Purple: AbstractApp, §3.6)

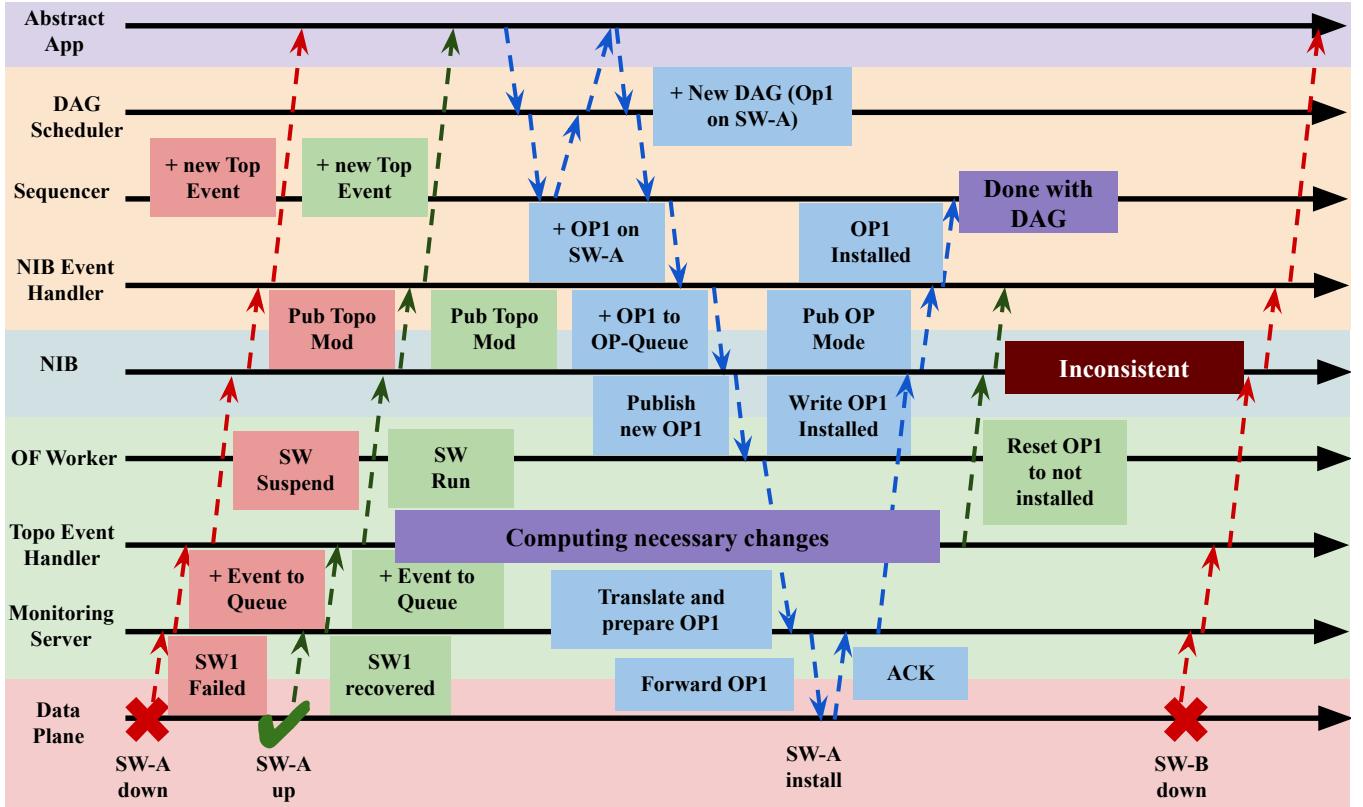


FIGURE A.8: An example trace from TLA+ that causes inconsistency between control and data plane, leading to packet drops. We simplified the trace significantly to improve its understandability. The original TLA+ trace for this bug consists of 64 steps and involves 3 switches.