RedPlane: Enabling Fault-Tolerant Stateful In-Switch Applications

Daehyeok Kim[⋆†], Jacob Nelson[†], Dan R. K. Ports[†], Vyas Sekar[⋆], Srinivasan Seshan[⋆]
[⋆]Carnegie Mellon University, [†]Microsoft

Abstract

Many recent efforts have demonstrated the performance benefits of running datacenter functions (*e.g.,* NATs, load balancers, monitoring) on programmable switches. However, a key missing piece remains: fault tolerance. This is especially critical as the network is no longer stateless and pure endpoint recovery does not suffice. In this paper, we design and implement RedPlane, a fault-tolerant state store for stateful in-switch applications. This provides in-switch applications consistent access to their state, even if the switch they run on fails or traffic is rerouted to an alternative switch. We address key challenges in devising a practical, provably correct replication protocol and implementing it in the switch data plane. Our evaluations show that RedPlane incurs negligible overhead and enables end-to-end applications to rapidly recover from switch failures.

CCS Concepts

• **Networks** → **Programmable networks**; **In-network processing**; • **Hardware** → **Emerging technologies**; • **Computer systems organization** → **Availability**.

Keywords

Programmable switches, Programmable networks, Fault tolerance, State replication

ACM Reference Format:
Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, Srinivasan Seshan . 2021. RedPlane: Enabling Fault-Tolerant Stateful In-Switch Applications. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–27, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3452296.3472905

1 Introduction

Today's data center switches are no longer simple stateless packet forwarders. They implement sophisticated network functions, such as NATs, firewalls, and load balancers [6, 39, 55] and accelerate distributed applications [45, 47, 67, 73, 78]. Cloud service providers have even started deploying them in production networks [7].

Such stateful processing in switches leads to a new challenge: fault tolerance. Classic network designs followed the end-to-end principle [66], keeping critical state only on the end hosts. This enabled a fate-sharing approach to reliability [27]; when switches are stateless, recovering from their failure simply entails finding a new communication path. Stateful in-switch applications [7] challenge this paradigm; *e.g.,* the failure of a switch running a load balancer may cause the loss of its forwarding state, breaking thousands of active connections. While data center networks are engineered with redundant network paths [34, 65, 71] to provide fault tolerance at the routing layer, there are no capabilities for recovering in-switch state after failure.

Thus, we need to reconsider fault tolerance for in-switch processing – something previously done in ad hoc, application-specific ways. Our goal in this paper is to ensure that, after a failure and reroute, the same application state becomes available at the replacement switch, without degrading performance and while remaining transparent to end hosts.

Making switch state fault tolerant is uniquely challenging because of the scale and resource constraints involved. Techniques like checkpointing and active replication, which have been applied to software middleboxes [63, 70], are designed for server-based systems. These techniques rely on obtaining a consistent snapshot of state and buffering output until state updates are durably recorded to other servers. However, a switch's high packet processing speed (a few *billion* packets/second [13, 14, 19]) and its limited compute and storage capabilities make it infeasible to translate these techniques to the switch context.

In this paper, we introduce *RedPlane*,[1] a fault-tolerant state store for in-switch applications. RedPlane provides APIs for developers to (re)write their stateful P4 programs and make them fault-tolerant. This allows an application to retain consistent access to its state, even if the switch it runs on fails or traffic is rerouted to an alternative switch. RedPlane achieves this through a data plane centric replication mechanism that continuously replicates state updates to an external state store implemented using DRAM on commodity servers. Note that running entirely in the data plane channel is key to keeping up with the switch's full processing speed.

Realizing this high-level idea in practice entails several challenges. First, traditional notions of strict correctness with linearizability and exactly-once semantics for operations require reliable communication and output buffering. However, this is infeasible on the switch data plane due to its limited capabilities. Second, at the traffic volumes the switch data plane needs to process, naïvely requiring per-packet coordination with the server-based state store imposes severe performance overheads. Last, routing decisions when a switch fails could be unpredictable. Thus, we must be able to transparently migrate the relevant state between two switches regardless of the routing decisions.

We address these challenges with the following key ideas:

- Based on the requirements of in-switch applications, we define two practical correctness models. First, based on our observation that network applications are already resilient to packet loss, we define a strict consistency mode by explicitly adopting

SIGCOMM '21, August 23–27, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8383-7/21/08.
https://doi.org/10.1145/3452296.3472905

[1] The name denotes a *replicated data plane.*

the standard definition of linearizability [36], which permits operations that do not complete while still providing strong consistency. Second, for write-centric applications (*e.g.,* monitoring using sketches [28]) that can tolerate approximate results, we propose a relaxed consistency mode that allows some state to be lost after a failure, but bounds the inconsistency with lower overheads.

- Instead of buffering packets using limited switch resources, we use the network itself and state store's memory as temporary storage by piggybacking packet contents on coordination messages.
- To enable reliable state replication, we build a lightweight sequencing and retransmission protocol that ensures state updates are processed in the correct order, without requiring complex protocols (*e.g.,* TCP) in the switch data plane.
- To avoid overheads due to frequent coordination with the state store, we propose a lease-based state ownership protocol [33, 49, 57] to provide correctness without coordinating on every state access and migrate ownership between different switches as needed.

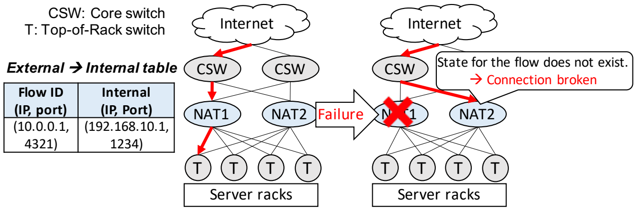
We design the RedPlane protocol that realizes our consistency modes, prove its correctness, and confirm this using a TLA+ model checker [18]. We implement a prototype of RedPlane in P4 [11] and C++/Python, and show that different types of applications can be fault tolerant using it. We evaluate it with various applications in our testbed consisting of two Tofino-based programmable switches, four regular switches, and 10 servers. Our evaluation results show that under failure-free operation, RedPlane has negligible per-packet latency overhead for read-centric applications like NAT, and less than 8 μs overhead even for the worst case. When a switch fails, RedPlane can recover end-to-end TCP throughput within a second by accessing the correct state.

2 Background and Motivation

In-network processing has flourished in recent years, as a natural convergence of the demand for sophisticated network functionality from data center operators and the commercial availability of programmable switch platforms [8, 13, 14]. Programmable switches are used for classic middlebox functionality [39, 55], monitoring [6, 35], DDoS defense systems [76, 77] and accelerating other networked systems [38, 45–47, 52, 67, 73, 78].

These applications are *stateful*; *i.e.,* state on the switch determines how to process packets. In this paper, we focus primarily on *hard state* applications, where a loss of state disrupts network or application functionality.[2] An example is an in-switch NAT, where the key state is an address translation table. Losing this state would make it impossible to forward packets for existing connections.

Network model. We consider a deployment model where programmable switches are installed into the network fabric such that all traffic to be processed by an in-switch application traverses one of the programmable switches. This could be achieved in several different ways, depending on the network architecture. In a typical data center architecture (Fig. 1), this could be achieved by using the switches on all core or all aggregation-layer switches.[3] All traffic entering or leaving a cluster, for example, would traverse one of these switches. Alternatively, an operator might deploy a cluster of programmable switches as dedicated "NF accelerators", explicitly routing traffic through them; this approach is similar to how software load balancers [30, 60] are deployed today.



Flow ID (IP, port)	Internal (IP, Port)
(10.0.0.1, 4321)	(192.168.10.1, 1234)

Figure 1: Impact of switch failures on in-switch NATs.

State partitioning. We assume that application state is partitionable using some key derived from the packet header, and that each packet's processing uses only state from the associated partition. In many cases, such as for the NAT example, the key will be the IP 5-tuple, and, hence, we use "partition" and "flow" interchangeably. However, other applications may use different partitioning, *e.g.,* partitioning on VLAN ID to detect heavy-hitter flows for a particular tenant.

We also assume that the network is configured to provide best-effort affinity such that packets from the same partition usually arrive at the same switch. Standard layer-3 routing protocols such as Equal-Cost Multi-Path routing (ECMP) provide this property when they are configured to use the partition key as their hash key.

Primer on programmable switches. Programmable switch architectures used today, *e.g.,* Intel Tofino [14], use a limited amount of on-chip memory (*e.g.,* SRAM and TCAM) to provide a variety of stateful object abstractions, including tables, registers, meters, and counters. Applications can use these to keep state across multiple packets, such as the address translation table in the NAT example above. In the ingress and egress match-action pipeline, objects are allocated in each stage and accessed by packets via ALUs. These objects are also accessible by the switch control plane through the ASIC-to-CPU PCIe channel which has a limited bandwidth (O(10 Gbps)) compared to the ASIC's per-port bandwidth (O(100 Gbps)). In addition, the ASIC provides other built-in functionality such as packet replication, recirculation, and mirroring for more advanced packet processing.[4]

2.1 Impact of Switch Failures

Switches can fail, either by a switch failing entirely (a fail-stop model), or by individual links losing their connectivity. Measurement studies in production data centers have shown that such switch failures are prevalent. For example, in Microsoft's data center, 29% of customer-impacting incidents are related to hardware

[2] Other applications maintain only soft state in the switch and provide their own failure recovery mechanisms. These are not the focus of our work, though RedPlane could perhaps help simplify their design or improve recovery performance.

[3] In principle, RedPlane could be deployed on top-of-rack (ToR) switches, but it is potentially less useful. If each rack has one ToR switch, and it fails, connectivity to the servers in that rack is lost. RedPlane can restore the switch state onto a different rack, but depending on the application that may not be useful. However, if there are two ToR switches per rack, RedPlane would be useful.

[4] While we use Tofino-based programmable switches for our work, we believe our design can be implemented on other programmable switch ASICs since hardware capabilities leveraged in RedPlane's switch data plane (*e.g.,* packet mirroring) are general features supported by most switch ASICs.

State access	Applications	Impact of switch failure
Read-centric	NAT	Connection broken
	Stateful firewall	Connection broken
	Load balancer [55]	Connection broken
	SYN flood defense [77]	Dropping valid packets
Write-centric	Super-spreader detection [72]	Inaccurate detection
	Heavy-flow detection [53]	Inaccurate detection
Mixed-read/write	SGW in EPC [69]	Active session broken
	In-network sequencer [46]	Incorrect sequencing
	Per-object routing [47, 78]	Choosing wrong servers
	In-network key-value store	Losing key-value pairs

Table 1: Examples of stateful in-switch applications and impact of switch failures.

failures including ASIC failure, fiber cuts, or power failures [50], and in Facebook's data center, 26% of incidents are related to switch failures [54].

Switch failures can impact stateful applications in two ways. If a switch fails entirely, all application state it held is lost. Beyond that, a link failure or the failure of a *different* switch can impact many paths in the network [51], causing traffic to be rerouted [22, 34, 43]. Traffic that previously traversed one switch might be routed to a different one, where the appropriate state is unavailable. In the absence of this state, application processing can fail. For example, as illustrated in Fig. 1, lacking the proper translation table entries, the NAT cannot forward packets for existing connections, breaking open connections *en masse*. Indeed, this is a serious problem – software-based stateful load balancers at cloud providers implement complex failover mechanisms [30, 60].

Beyond the conventional NFs (*e.g.,* NATs, load balancers, firewalls), there are several in-switch applications (shown in Table 1) that exhibit complex state access patterns. For example, many applications that are designed to enforce QoS policies (*e.g.,* rate limits) employ streaming algorithms (*e.g.,* sketching) to capture characteristics of traffic such as heavy-hitters [53, 72]. Switch failures lead them to make inaccurate decisions as the statistical data is lost. Such applications update state (*e.g.,* sketches) on every packet, so we call them *write-centric*. In contrast, many conventional NFs and DDoS defense systems (*e.g.,* SYN proxy) [76, 77] are *read-centric*.

Another group of applications have *mixed-read/write* state access patterns, typically with much less frequent updates than write-centric applications. One example in this category is NFs in the packet core for cellular networks (*e.g.,* Evolved Packet Core (EPC) for LTE) [17]. Packet core NFs such as a serving gateway (SGW) route users' data traffic from user devices to the Internet and vice versa based on per-user states (*e.g.,* forwarding state), which are updated when the control plane receives signaling messages (*e.g.,* device attached). To cope with the increasing volume of signaling traffic [4, 10],[5] there have been recent efforts to accelerate the control plane functions by offloading them to the programmable data plane [5, 9, 61, 69]. For example, a SGW running on a switch maintains per-user tunnel endpoint IDs (TEIDs) to route packets, and this state is updated by signaling messages and read by data packets that are encapsulated with TEIDs. Thus, when a switch fails, since the SGW loses the state, it cannot forward packets for users, disrupting active connections. Affected users need to re-establish connections after the failure [21], increasing the service latency. Other applications that route requests in application-specific ways (*e.g.,* for databases [78] or key-value stores [47]) also fall into this category since they require state updates on every write (but not read) request.

2.2 Existing Approaches and Limitations

We now examine classical fault tolerance mechanisms [32, 58, 74] and mechanisms tailored for network middleboxes [63, 70]. At a high level, these approaches can be categorized into three classes: (1) checkpoint-recovery, (2) rollback-recovery, and (3) state replication. All prior work targets server-based implementations. In what follows, we discuss why natural adaptations of these approaches to the switch environment fail to ensure correct behavior during failures.

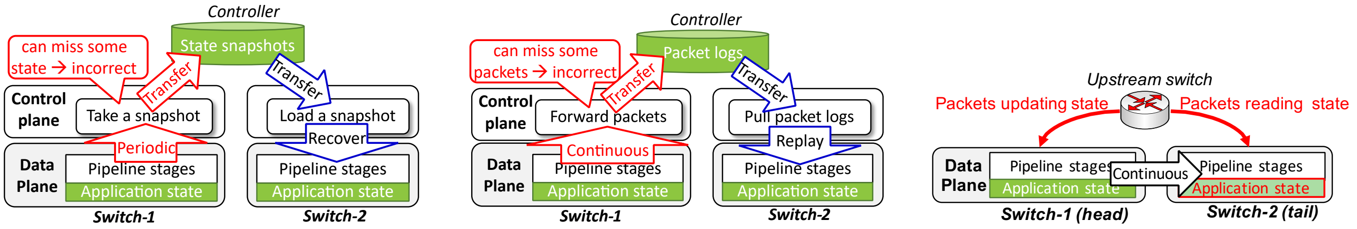
Checkpoint-recovery. Checkpointing approaches periodically snapshot application state (*e.g.,* an address translation table in NAT) and commit it to stable storage (*e.g.,* [63]). When a failure occurs, the latest snapshot is populated on a backup node (*i.e.,* an alternative switch in our context). Fig. 2a illustrates a candidate implementation on switches using an external controller to store snapshots via the switch control plane. To achieve a consistent snapshot, data plane execution must be paused and packets buffered during the snapshot period. Limited data-to-control plane bandwidth in modern switch architectures makes this impractical.

Rollback-recovery. This approach, previously used for software middleboxes [70], logs every packet to stable storage and replays the traffic logs on a new device after failure to reconstruct application state. A natural implementation is sending every packet to the switch control plane, which logs it to the controller (Fig. 2b). In principle, this approach can guarantee correctness if every packet is synchronously logged and replayed after a failure. However, the mismatch between the data traffic rate (*Tbps*) and the data-to-control plane bandwidth (*Gbps*) will result in many packets being dropped and will, thus, be incorrect.

State replication among switch data planes. Consider a state machine replication approach using chain replication [74], but applied to switch data planes (Fig. 2c). Packets are forwarded through a sequence of switches, each of which updates its state and forwards the packet to the next switch in a chain. Only once the packet has reached the tail of the chain is it forwarded on its way to its destination. This is done entirely on the data plane, so it can function at high speed. This approach achieves correctness *only if* state updates are not lost. However, the state updates are delivered over an unreliable channel, and since the switch data plane cannot effectively support reliable transport protocols (*e.g.,* TCP) updates could be lost or reordered, violating correctness. Also, using one switch to replicate another switch's state makes poor use of data plane-accessible switch memory – the most costly and limited resource. It also requires changes to the routing policy of the network since a packet needs to be explicitly routed to a specific switch in the chain depending on whether the packet updates state or not.

Takeaways. From the above discussion, we see two key takeaways. First, approaches that rely on the switch control plane must consider the mismatch between control and data plane speeds. Second,

[5] Despite the growth, it is expected that signaling traffic rate is still much lower than that of data traffic (*e.g.,* 5% of data traffic [56]).



(a) Checkpoint-recovery : Switch control plane periodically snapshots and commits state to the controller. During this time, all packets must be buffered.

(b) Rollback-recovery : Each packet is forwarded to the control plane and logged by the controller.

(c) State replication : Switches replicate state to data plane memory using chain replication. Packets must be routed to the correct chain node.

Figure 2: Highlighting why adapting existing approaches for fault tolerance fails for hardware switches.

while switch data-plane-only approaches can provide good performance, they suffer three shortcomings: (a) incurring significant switch resource overhead; (b) making it difficult to reason about correctness due to unreliable communication channels between switch data planes; and (c) they may additionally constrain routing policies.

3 RedPlane Overview

Our goal is to design a fault tolerance solution that provides the following four properties:

- **Correctness**: Switch failure should be transparent to applications: clients should not see state that would not be possible in the absence of a failure.
- **Performance**: Under failure-free operation, overhead for per-packet latency should be low (say, a few tens of μs).
- **Low resource overhead**: It should not consume switches' limited compute and storage resources excessively.
- **Transparency to routing policies**: That is, we must allow a packet to update and/or read state regardless of the location of a switch where the packet is routed.

To this end, we present RedPlane, which provides an abstraction of fault-tolerant state storage for stateful in-switch applications. RedPlane provides an illusion of "one big fault-tolerant switch" – the behavior is indistinguishable from the same application running on a single switch that never fails. To achieve this, RedPlane continuously replicates state updates which can be restored without loss after a failure.

RedPlane takes a state replication approach with two defining characteristics: (1) the switch's state replication mechanism is implemented entirely in the data plane, and (2) state storage is done through an *external state store*, a reliable replicated service made up of traditional servers. Property (1) means that the switch's control plane is not required for state replication, avoiding the issues with the checkpointing and rollback-recovery approaches of §2.2. Property (2) means that the replicated state is stored in commodity server DRAM, a relatively low cost storage medium compared to switch data plane memory. This avoids the high resource overhead of the state replication approach discussed in §2.2.

While the idea of using servers' memory as an external store is similar to recent work on TEA [42], it is important to note that TEA does not tackle fault tolerance. It focuses on the problem of resource augmentation to enable a switch to retrieve state stored in memory of servers. Furthermore, that design can only utilize servers *directly attached* to a top-of-rack switch. As such, their design does not tackle fault tolerance or provide provable correctness in the scenarios when multiple switches can access the store.

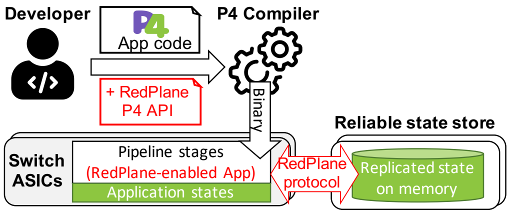


Figure 3: RedPlane overview highlighting extensions to traditional workflows for in-switch applications

RedPlane provides a set of APIs (Fig. 3) implemented in P4 [11], a language to specify data plane programs on programmable switches, to allow developers to easily integrate RedPlane with their stateful P4 applications. Once developers (re)write their applications using RedPlane APIs, the P4 compiler generates a binary of RedPlane-enabled applications loaded to the switch, which continuously replicates updates to the state store through the data plane.

Scope and limitations: In this work, our focus is on enabling fault tolerance for stateful applications with partitionable hard state, where a loss of state disrupts network or application functionality, shown in Table 1. Applications only with non-partitionable state (*e.g.,* global counter) are beyond the scope of this work. Also, we assume that global state in an application (*e.g.,* a port pool in NAT) is sharded across and managed by state store servers. Other applications that need soft-state (*e.g.,* in-network caches or ML accelerators) do not require fault tolerance, but may benefit from RedPlane.

3.1 Challenges

While replicating state updates through the data plane to an external state store seems appealing, realizing this idea in practice presents some challenges:

C-1. Providing correct replication in the data plane while tolerating unreliable communication. Traditional server-based replicated systems aim to provide strict correctness by ensuring not just linearizability but also that each operation is executed exactly once even in the presence of dropped or retransmitted messages [44, 49, 57]. To do so, they build on reliable communication channels like TCP. However, the switch data plane cannot support reliable communication, nor can it buffer significant amounts of traffic.

C-2. Handling high traffic volume. Switch data plane operates at immense traffic volumes (up to a few billion packets per second [13, 19, 23]), in contrast to server-based systems handling a few million. If each packet that reads or updates state requires interacting with a server-based state store, the servers' capacity will rapidly be exceeded. It will also incur significant performance overhead.

C-3. Being transparent to routing policies. A switch failure, recovery, or network routing change could cause traffic flows originally processed at a switch S1 to be routed to a different switch S2. However, since the routing decisions may be unpredictable, we cannot make assumptions on S2 or presuppose what backup routes will be taken. That is, we must be able to transparently migrate the relevant state from S1 to S2 irrespective of the location of S2. For instance, we need to make the NAT table entry available when packets for a particular connection are processed by a different instance.

3.2 Key Ideas

To tackle these challenges, we build on four key ideas:

I-1. Practical correctness for switch state (§4). We define two correctness models based on the requirements of in-switch applications. The first, a strict consistency mode, is based on linearizability [36]. Because we observe that network applications are already designed to tolerate packet loss, we explicitly adopt the standard definition of linearizability, which permits operations that do not complete while still providing strong consistency for those that do. Second, since many write-centric applications (*e.g.,* monitoring using sketches [28]) accept approximate results, we propose a relaxed consistency mode that allows some state to be lost after a failure, but bounds the inconsistency.

I-2. Piggybacking output packets (§5.1). Instead of buffering output packets using limited switch resources, we use the network itself as temporary storage by piggybacking packet contents on coordination messages.

I-3. Lightweight sequencing and retransmission (§5.2). To cope with the unreliable communication channel between the switch data plane and the state store with low resource overhead, we employ a sequencing mechanism for protocol messages and devise a lightweight switch-side retransmission mechanism by repurposing the switch ASIC's packet mirroring feature.

I-4. Lease-based state ownership (§5.3). To reduce the frequency with which the switch must coordinate with the state store, especially for applications with read-centric and mixed-read/write workloads, we adopt a lease-based mechanism inspired by prior work [33, 49, 57]. This allows us to avoid coordination with the state store for packets which need to read but do not modify state. At the same time, we ensure that all state updates are durably recorded before any of their effects are externalized, guaranteeing linearizability. This mechanism also serves as the means by which state is migrated between switches to support the transparency.

Taken together, these high-level ideas address the aforementioned challenges. First, the linearizability-based consistency model coupled with the piggybacking and lightweight sequencing and retransmission mechanism allows to replicate state reliably and correctly (C-1). Second, the relaxed consistency and lease-based state ownership help cope with high traffic volume (C-2). Lastly, the lease-based state ownership makes RedPlane transparent to routing policies (C-3).

4 Correctness Model

RedPlane provides two levels of consistency, which applications can choose between based on their requirements. A *linearizable* mode provides strict guarantees, making the system indistinguishable from a single fault-tolerant switch. Because this has a high overhead for write-centric applications due to frequent coordination with the state store, RedPlane also offers a *bounded-inconsistency* mode that permits some state updates to be lost on switch failure, but guarantees a consistent view of switch state.

4.1 Preliminaries

By default, RedPlane provides *linearizability* [36], a correctness condition for concurrent systems. We model a stateful in-switch program as a state machine, where the output and next state are determined entirely by the input and current state:

Definition 1 (Stateful in-switch program). A stateful program P is defined by a transition function $(I, S) \rightarrow (O^*, S')$ that takes an input packet and the current state, and produces zero, one, or multiple output packets, along with a new state.

To simplify the definitions below, we will assume that each input packet p produces exactly one output packet $P(p)$; it is straightforward to extend them to the zero- or many-output case. This implies that the program's behavior is determined entirely by the sequence of input packets, and in particular that it is deterministic and that packets are processed atomically. Although switch architectures are pipelined designs that process multiple packets concurrently [26], their compilers assign state to pipeline stages in a way that makes packet processing appear atomic [24].

The gold standard for replicated state machine semantics is single-system linearizability [36]. That is, that the observed execution matches a sequential execution of the program that respects the order of non-overlapping operations. To adapt linearizability for in-switch programs, we first redefine a history in terms of packet processing:

Definition 2 (History). A history is an ordered sequence of events. These can be either input events I_p, in which a packet p is received at a RedPlane switch, or output events O_p in which the corresponding output packet is output by a RedPlane switch.

Note that it is possible for there to be input events I_p without the corresponding output O_p, if the processing of p is still in process or due to a failure. We discuss this in depth next.

4.2 Linearizable mode

Definition 3 (Linearizability for a stateful in-switch program). A history H is a linearizable execution of a program P if there is a reordering S of the input events in H such that (1) the value for each output event O_p that exists in H is given by running P on the input events in S in sequence, and (2) if O_x precedes I_y in H then I_x precedes I_y in S.

Here, S is the apparent sequential order of execution.

Linearizability is fundamentally a safety property, not a liveness one: it specifies what output values are acceptable, but does not guarantee that all operations complete. It is possible for a packet to be received and (1) update switch state, but produce no output, or (2) neither update switch state nor produce any output. Definition 3 reflects this: a packet with an input event but no output event can still appear in the sequential order S. If it precedes the processing of other packets, then they see the effects of its state update. If it appears at the end of S, it has no visible effect on system state.

While these anomalies comport with the definition of linearizability, most replicated systems aim to provide a stronger property: that every operation is executed exactly once and returns its result to the client. Ensuring this requires several protocol-level mechanisms: typically, clients retry requests that do not receive a response, and replicas keep state to detect duplicate requests and resend the responses without executing them twice [44, 49, 57]. As we see (§5.2), these techniques are not feasible in our environment.

Accordingly, RedPlane takes a different approach: it *explicitly permits* these two types of anomalies. While this may seem surprising, it matches the semantics of modern networks. The two cases correspond to a packet being lost (1) between the RedPlane switch and its destination or (2) between the source and the RedPlane switch, respectively. Network applications must already tolerate lossy networks, so they are resilient to such losses.

Relaxing the definition of correctness enables a tractable implementation. By not requiring the system to achieve complete reliability, our protocol may drop packets during failover, or if messages between a switch and the state store are lost. In these scenarios, an input packet or its output may be lost. Of course, dropping too many packets is undesirable for performance reasons; such loss events are rare.

4.3 Per-flow Linearizability

In most in-switch programs, some or all state is associated with a particular flow – a subset of traffic identified by a unique key, *e.g.,* an IP 5-tuple, VLAN ID, or an application-specific object ID. For example, each translation table entry in a NAT is tied to a specific flow based on an IP 5-tuple. For many applications, per-flow state is the only state that needs to be consistent or fault tolerant – either because there is no global state, or because global state can tolerate weaker consistency, *e.g.,* traffic statistic counters that need not be precise. RedPlane generally provides consistency for per-flow state (consistency for global state is optional):

Definition 4 (Per-flow linearizability). A history H is per-flow linearizable if, for each flow f, the subhistory H_f for the packets in flow f is linearizable.

As long as programs use only per-flow state, per-flow linearizability is *the same* as global linearizability, because linearizability is a *local* (*i.e.,* composable) property [36]. The benefit of operating on a per-flow level is that it means synchronization between states associated with different flows are not required. As we show in §5, this allows RedPlane to distribute execution of a program across multiple switches: each has the state associated with certain flows, and can process packets for those flows. This matches the way many applications are deployed in practice, *e.g.,* a NAT will be deployed to a cluster of switches, using ECMP for load balancing. Because this load balancing is done on a per-flow granularity, each switch is responsible for performing translation for a subset of flows, and does not need access to the translation table for the other flows.

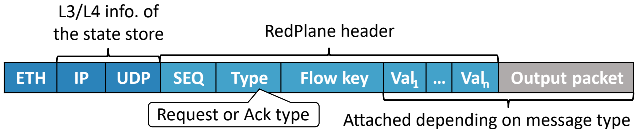


Figure 4: RedPlane state replication protocol packet format.

4.4 Bounded-inconsistency mode

RedPlane's linearizable mode uses a synchronous replication protocol (§5.1), which can induce high overhead for write-centric applications. However, we observe that many write-centric applications in programmable switches operate in contexts where approximate results are acceptable, *e.g.,* monitoring using sketches [28] or Bloom filters [25]. For these applications, RedPlane offers a *bounded-inconsistency* mode that has lower overhead.

In this mode, RedPlane takes periodic snapshots of data plane state and replicates them asynchronously. This means that upon switch failure, the most recent state updates can be lost. However, RedPlane ensures that the system recovers to a consistent state from within a time interval ϵ. RedPlane's consistent snapshot mechanism ensures that the state after recovery reflects an actual state of the system, which simplifies reasoning about the correctness of complex data structures.[6] In §5.4, we describe how we address key challenges in realizing this mode in RedPlane.

5 RedPlane Design

Now, we describe the RedPlane protocol that realizes our linearizable and bounded-inconsistency modes. We begin with an overview of the protocol and explain how we address practical challenges.

5.1 Basic Design

As shown in Fig. 3, RedPlane consists of (1) an external state store built on commodity servers and (2) a RedPlane-enabled application running on the switch data plane. In this section, we describe how the components work together via the state replication protocol.

For clarity of exposition, we start with simplifying assumptions: there is no packet loss or reordering between switches and the state store, switches do not fail while messages are in transit, and packets for a flow are routed to only one switch at a time. We revisit these assumptions in §5.2 and §5.3.

5.1.1 *External state store:*

The external state store is an in-memory key-value storage system. We partition it across multiple shards by flow – identified by an IP 5-tuple or other key. Each state store shard can be replicated using conventional mechanisms and we do not seek to innovate here as many existing key-value stores meet our needs (*e.g.,* [16, 48, 59]). Specifically, our prototype is a simple in-memory storage server implemented in C++ that uses chain replication [74] with a group size of 3.

[6] Although the bounded-inconsistency mode may affect properties of some approximate data structures (*e.g.,* no false negatives in Bloom filters), since it bounds the inconsistency within ϵ, developers or network operators can easily reason about the potential inconsistency.

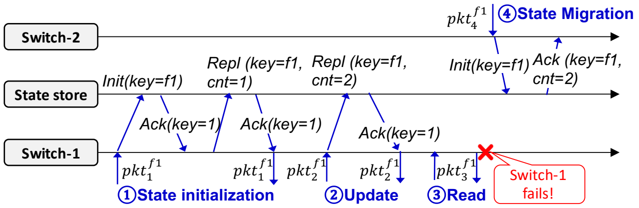


Figure 5: Basic workflow of RedPlane state replication protocol. "Repl" indicates a state replication request. pkt_n^f indicates n^{th} packet of a flow f.

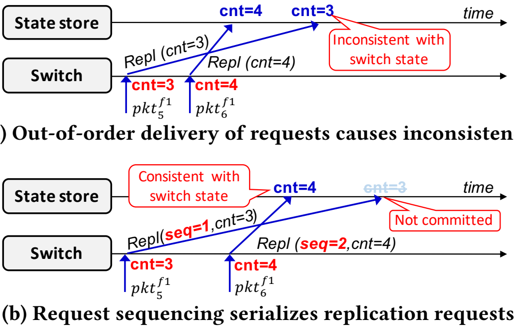
5.1.2 *Basic replication protocol:* A RedPlane-enabled application replicates state updates to the state store by exchanging protocol messages formatted as shown in Fig. 4. It uses standard UDP and IP headers to address messages to the state store or the switch using their respective IP addresses. The RedPlane header consists of a sequence number, a message type, and a flow key. Depending on the message type, it can also include flow state and an output packet. We will discuss these fields shortly. Note that we assign an IP address to each RedPlane switch and use it for routing requests and response packets between state store servers and RedPlane switches. This works with general L3 routing protocols including ECMP and BGP.

As an illustrative example to help understand the protocol, we consider a per-flow counter application shown in Fig. 5, This application updates or reads the state for each packet. In the example, there are two switches and a state store. We have multiple packets in each flow f, with the n^{th} packet denoted as pkt_n^f. This example illustrates a case where the Switch-1 initially handles $f1$, but after its failure, the flow is rerouted to the Switch-2.

State initialization or migration (Step ① or ④ in Fig. 5). When the application receives a packet that belongs to a flow it has never seen before (*e.g.,* pkt_1^{f1}), it needs to send a *state initialization request.* It identifies the corresponding state store server by hashing the flow key (*e.g.,* IP 5-tuple), and looking up the corresponding server IP and UDP port from a preconfigured table.

There are two possible cases: (1) the flow is new and so has no state, or (2) the flow state previously existed on a failed switch, and a packets for that flow are now being routed to a switch on an alternative path (*i.e.,* failover). In case (1), upon receiving the request, the state store initializes its storage for the state and sends a response back to the switch (**Step ①**). In case (2), since the state store already has the flow state, it sends a response containing the latest state (**Step ④**).

Upon receiving the response, the application installs the returned state into the corresponding switch memory. For stateful memory registers, this can be done entirely in the data plane. On the Tofino architecture, updates to match tables or certain other resources need to be done through the switch control plane. In this case, RedPlane routes the processing through the control plane. This can introduce additional latency (we measure this in §7.1). However, many in-switch applications already require a control plane operation on a new flow (*e.g.,* to install a new translation mapping in a NAT), in which case the added overhead is minimal.



(a) Out-of-order delivery of requests causes inconsistency.

(b) Request sequencing serializes replication requests.

Figure 6: Serializing out-of-order requests with sequencing. Counter values (cnt) in red and blue indicate the state on the switch and the state store, respectively.

Reading or updating state (Step ② or ③ in Fig. 5). Once the state has been initialized, the application can read the state value (*i.e.,* the counter in our example) directly (**Step ③**). When it updates the state (*i.e.,* the counter value), RedPlane sends a *replication request* with the new value to the state store. This message is generated entirely through the data plane. The state store applies the update, and sends a *replication reply* message (**Step ②**).

Piggybacking output packets. When the application updates the state, RedPlane should not allow an output packet to be released until the state has been recorded at the state store – otherwise, the update could be lost during a switch failure, violating correctness. This requires the output packet to be buffered until the replication reply is received.

Unfortunately, the switch data plane does not have sufficient memory to buffer packets in this way (and various other constraints on how memory can be accessed make it unsuitable for storing complete packet contents). RedPlane instead piggybacks the packet onto its replication request message, and the state store returns it in its reply. When the reply is received, RedPlane decapsulates and releases the packet. In effect, this uses the network and the memory on the state store as a form of delay line memory – trading off network bandwidth, which is plentiful on a switch, for data plane memory, which is scarce.

Note that it is possible to receive packets that read state when there are in-flight replication requests for the state. In this case, the packets are buffered in the same way through the network (with a special RedPlane request type) until a switch receives a response for the latest replication request.

While our basic design provides correctness under the simplified assumptions, we find that in more realistic environments, it may not be able to guarantee correct behavior. In the following sections, we describe potential challenges, and how we extend the basic design to address them.

5.2 Sequencing and Retransmission

To guarantee correctness, replication requests must be *successfully* delivered and replicated *in order* at the state store. For example, the replication request (**Step ②** in Fig. 5) must be delivered in order. However, successful in-order delivery is not guaranteed in a best-effort network between switches and the state store.

Fig. 6a illustrates why such unreliability in the network can be problematic. We use the same per-flow counter as an example. Each time the counter is incremented, RedPlane sends the new value to the state store. If the state store just processes updates in the order they are received, a reordering could cause a later counter value to be replaced with an earlier one. Request loss can cause a similar issue.

A traditional replication system, like chain replication, might address this by relying on a reliable transport protocol like TCP. Unfortunately, it is not practical to implement a full TCP stack on the switch data plane – if it is possible at all, it would excessively consume data plane resources.

Our approach. Instead of implementing a full-fledged reliable transport on a switch data plane, we choose to build a simpler UDP-based transport with mechanisms that deal with possible packet reordering and loss. First, to handle out-of-order state replication request messages, we employ a mechanism called *request sequencing* [46], which assigns a per-flow monotonically increasing sequence number to each request message. The state store uses this sequence number to avoid applying updates out of order (Fig. 6b).

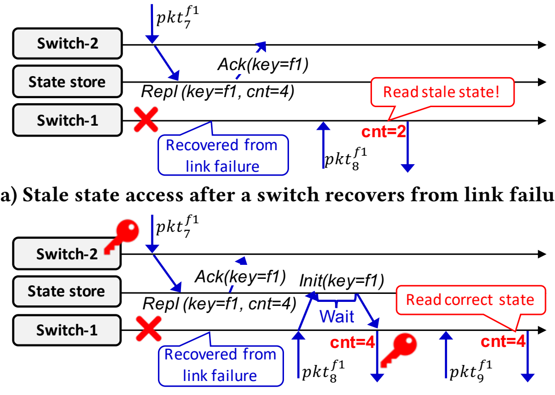
Second, to cope with lost replication requests or responses, we develop a mechanism for *request buffering*. RedPlane buffers replication requests and retransmits them if it does not receive a reply before a timeout. We implement this by repurposing the egress-to-egress packet mirroring capability of switch ASICs. When RedPlane sends a replication request, it mirrors a copy with the current timestamp as metadata. When the mirrored request enters the egress pipeline and it has not been acknowledged by a response with the same or a higher sequence number, RedPlane checks whether the request has timed out by comparing the current timestamp to the timestamp in its metadata. If it has timed out, it resends the request to the state store. Otherwise, it mirrors the request again without ending the request to the state store.

As discussed previously, buffering a full packet payload is challenging on a switch due to memory limitations. Instead, RedPlane buffers *only state updates* (*i.e.,* the RedPlane header) – not the piggybacked output packet by truncating the packet. This reduces the amount of data that needs to be mirrored. A consequence of this is that if a replication request or its response is dropped, the output packet will be lost. This is permitted by our linearizability-based correctness model: it is indistinguishable from the output packet being sent and dropped in the network. The state updates must be retransmitted, however, because subsequent packets processed by the switch may see the new version, and thus it must be durably recorded. We measure the overhead of request buffering in §7.4.

5.3 Lease-based State Ownership

What if multiple switches attempt to process packets for a particular flow at the same time, especially during failover or recovery? The protocol in §5.2 will not be correct in this case, when there are concurrent accesses to the same state. Fig. 7a illustrates why. After Switch-1 has a link failure (but does not lose its state, which is *cnt*=2), packets are routed to an alternate, Switch-2. If Switch-1 recovers, a packet may read its old state, a violation of linearizability.

Our approach. RedPlane ensures that only one switch can process packets for a given flow at a time using *leases*, a classic mechanism for managing cached data in file systems [33] and replicated systems [49, 57]. Fig. 7b illustrates this. If a packet wants to access



(a) Stale state access after a switch recovers from link failure.

(b) Only one switch holds a lease (red key) on state at a time.

Figure 7: Consistent state access for multiple switches.

state, but the state is not available at the switch, it first requests a lease for the flow. The state store grants a lease for a specific time period (1 second in our prototype) only if no other switch holds an active lease on the same flow state. The lease time is renewed each time the switch sends a replication request for that flow; switches that frequently read but infrequently update state can send explicit lease renewal requests. Our prototype does so every 0.5 seconds.

5.4 Periodic Snapshot Replication

As described in §4, RedPlane offers bounded-inconsistency mode for write-centric applications that permit approximate results, *e.g.,* monitoring using sketches [72] or Bloom filters [76]. In this section, we describe how we realize it in the switch data plane.

For such applications, RedPlane replicates *snapshots* of state *asynchronously* and *periodically*. Every T_{snap} seconds, a snapshot of the current state is sent to the state store, while output packets are released without waiting for replication to complete.

However, realizing this approach entirely in the data plane is challenging. While data structures often consist of multiple entries (*e.g.,* slots in sketches), the switch is architected, and the P4 language is designed, to allow access to a single entry per register array per packet. Also, building hardware that could atomically copy entire register arrays would be costly.

To address this challenge, we employ a *lazy snapshotting* approach. We maintain two copies of the data structure that are lazily synchronized with each other. These are interleaved in the switch's register arrays so that each array index contains two entries, one from each copy. Two metadata registers are used to indicate which entry at each index is the *active* copy. The first, a 1-bit flag, is toggled when a snapshot is taken. The second, a 1-bit register array, represents whether that index has been updated since the current snapshot started.

To take a snapshot, we flip the flag and read values from the now-inactive copy. Meanwhile, when packets arrive and update the array, one of two operations occur. The first packet to update an index synchronizes the two copies and then updates the active copy. Later packets simply update the active copy. This allows us to take a consistent snapshot of the entire structure while incoming packets continue to update it. Additional snapshots must wait for the current one to complete. We describe the pseudocode of our mechanism in Appendix A.

Replication is achieved using the switch ASIC's packet generator. We configure it to generate a batch of packets every T_{snap} seconds. To replicate a data structure with n entries, we generate a batch of n packets, each with a unique ID p_i. The ID in each packet is used to address the ith entry in the data structure and copy its value into a RedPlane replication protocol header. Note that while RedPlane asynchronously replicates snapshots, it still guarantees successful replication with its sequencing and retransmission mechanisms.

5.5 Protocol Correctness

RedPlane's replication protocol provides per-flow linearizability defined in §4. Due to space constraints, we give only a brief sketch of the reasoning here. The lease protocol ensures that at most one switch is executing a program for a particular flow at a time. The sequencing, retransmission, and buffering protocol ensure that an output packet is never sent unless the corresponding state update has been recorded and acknowledged by the state store.

During non-failure periods, RedPlane provides per-flow linearizability because the single switch processing packets for a flow operates linearizably, but some output packets may be lost (due to dropped replication traffic with piggybacked messages). After a failover, the new switch receives a state version at least as new as the most recent output packet from the old switch. This satisfies the linearizability requirement that any packet sent after these output packets were observed follow it in the apparent serial order of execution. We also wrote a TLA+ specification of the linearizable mode to model-check the above property (Appendix C).

Our periodic snapshot replication guarantees that the system recovers to a consistent state from within a time bound ϵ (*i.e.,* bounded inconsistency) by tracking the time since the last successful replication; if the time bound is exceeded, an application-specific action may be taken (*e.g.,* dropping further packets or treating the switch as failed).

6 Implementation

Our prototype implementation is available in our repository [20].

Data plane. We implement RedPlane's data plane components in P4-16 [11] (≈1192 lines of code) and expose them as a library of P4 control blocks [11, §13], which form the RedPlane APIs that developers can use to make application state fault tolerant. We compile RedPlane-enabled applications to the Intel Tofino ASIC [14] with P4 Studio 9.1.1 [12]. We implement key functions such as lease request generation, lease management, sequence number generation, and request timeout management, using a series of match-action tables and register arrays. We evaluate the additional resource usage in §7.4. As mentioned in §5.2, we implement request buffering via the mirroring and truncation capabilities of the switch ASIC, which allows us to buffer only the replication protocol data and discard the original payload. We implement a basic sketch that supports lazy snapshotting; developers can modify it to implement similar data structures such as Bloom filters.

Control plane. We implement the switch control plane in Python and C++. Its main function is to initialize and migrate (if available) state for the data plane by processing corresponding responses forwarded by the data plane component.

State Store. Our contribution is in the fault tolerance protocol design and switch components. As such, our state store prototype is built based on readily available libraries and simple implementations. We implement RedPlane's state store in C++ for Linux servers. It uses Mellanox's kernel-bypass raw packet interface [3] for optimized I/O performance. To ensure reliability in the presence of server failures, we implement chain replication [74] using a group of 3 servers located in different racks.

Applications. To demonstrate the applicability of RedPlane, we implement various applications in P4 described below. The simplified P4 code for NAT is available in Appendix B.
(1) NAT: The NAT implementation uses RedPlane to implement a fault-tolerant per-5-tuple address translation table and available port pool. Since the port pool is a shared by different flows, it is sharded across state store servers and managed by them. The state is updated when a TCP connection is established from an internal network.
(2) Firewall: The stateful firewall adds fault-tolerance to a per-5-tuple TCP connection state table using RedPlane. Its state is updated when a TCP connection is established from an internal network.
(3) Load balancer: The load balancer maintains a per-5-tuple server mapping table; we make it fault-tolerant using RedPlane. It also uses a server IP pool, which is shared state. When a new TCP connection is established from an external network, the state is updated.
(4) EPC-SGW: We also implement a simplified serving gateway (SGW) used in cellular networks, a mixed-read/write application. It maintains per-user tunnel endpoint ID state. The state is updated by signaling messages and read by data packets.
(5) Heavy-hitter (HH) detection: We implement a heavy-hitter detector using count-min sketches [28] as an example of write-centric applications; there are 3 sketches, each consisting of 64×32-bit slots indexed by a hash of the IP 5-tuple. We implement separate sketches per VLAN ID, assuming that the network operator wants to enforce different policies for each cloud tenant. Since sketches are an approximate data structure which can be replicated asynchronously, we use periodic snapshot replication.
(6) Per-flow counter: To demonstrate RedPlane's worst-case performance, this application counts packets forwarded for each IP 5-tuple. State is updated for every packet and synchronous replication must be used.

7 Evaluation

We evaluate RedPlane on a testbed consisting of six commodity switches (including two programmable ones) and servers (see Appendix D) using both real data center network packet traces and synthetic packet traces. Our key findings are:

- In failure-free operation, RedPlane adds no per-packet latency overhead for applications that are read-centric or replicate state asynchronously. For write-centric applications in linearizable mode, RedPlane incurs 8 μs per-packet overhead (§7.1).
- In failure-free operation, the throughput of read-centric applications is not degraded. For write-centric applications, the throughput is bottlenecked by state store performance in linearizable

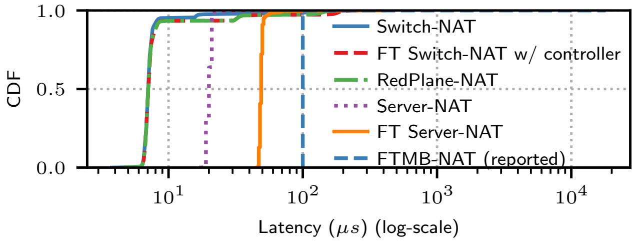


Figure 8: End-to-end RTT when RedPlane-NAT processes packets *vs.* other approaches.

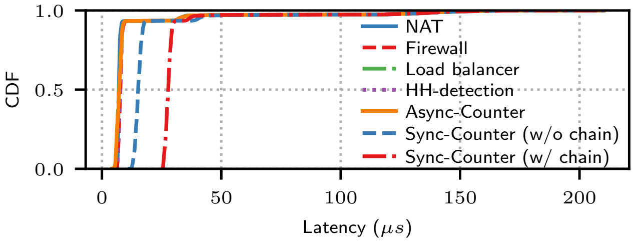


Figure 9: End-to-end RTT for RedPlane-enabled applications. All applications have chain replication enabled for the state store. For Sync-Counter, we also show its overhead without chain replication.

mode, but periodic snapshot replication reduces the overhead. Similarly, RedPlane incurs almost no bandwidth overhead for read-centric applications and small overhead for write-centric in bounded-inconsistency mode even at scale (§7.2).

- After a switch failure, RedPlane-enabled applications access their correct state and recover end-to-end TCP throughput within a second (§7.3).
- RedPlane provides these benefits with little resource overhead as it consumes <14% of ASIC resources (§7.4).

Testbed setup. We build a three-layer network testbed (shown in Appendix D). The aggregation layer has two 64-port Arista 7170 Tofino-based programmable switches [23] running stateful applications written in P4. The core and ToR switches run 5-tuple-based ECMP routing to route packets to end hosts even when one aggregation switch fails. Each ToR switch has two servers connected, and four additional servers attached to the core switch emulate hosts outside the datacenter. The state store runs on one server in each rack. All servers are equipped with an Intel Xeon Silver 4114 CPU (40 logical cores), 48 GB DRAM, and a 100 Gbps Mellanox ConnectX-5 NIC, running Ubuntu 18.04 (kernel version 4.15.0). We repeat each experiment 100 times unless otherwise noted.

7.1 Latency in Normal Operation

First, we evaluate the per-packet latency overhead introduced by RedPlane under failure-free operation for the 5 applications in §6. To measure the processing latency, we have each application send packets back to a sender node and track the RTT of each packet. We replay publicly available packet traces from a real data center and enterprise network [1, 2] to generate 100,000 packets and measure the processing latency of each packet. The packet sizes vary (64–1500 bytes) in the real traces. To evaluate EPC-SGW, we inject a signaling packet for every 17 data packets, following statistics used in previous studies [56, 62].

Overhead of RedPlane. As an exemplar application, we evaluate the per-packet latency for a NAT in RedPlane[7] and compare it with baseline implementations: (1) NAT written in P4 without fault-tolerance (Switch-NAT), (2) NAT written in P4 with controller based fault-tolerance (Switch-NAT w/ an external controller)[8] (3) NAT implemented on a CPU server without fault-tolerance (Server-NAT), (4) NAT implemented on a server with fault-tolerance (FT Server-NAT), and (5) FTMB-NAT which uses rollback-recovery for server-based middleboxes [70].[9] For switch-NAT w/ controller, RedPlane-NAT, and server-NAT, we enable chain replication for the controller, state store, and NAT instances, respectively.

Fig. 8 shows the CDF of the per-packet latency distribution. Compared to Switch-NAT, which is expected to have the lowest latency, RedPlane-NAT shows the same 50th and 90th percentile latency (7 μs and 8 μs, respectively), meaning that there is no overhead. This is because for NATs, packets except for the first packet of each flow only require state (*i.e.,* address translation table) to be read. Both Switch-NAT and RedPlane-NAT show a high 99th percentile latency (110 μs and 142 μs, respectively), mainly due to the overhead introduced by our control plane implementation; in Switch-NAT, the first packet of every flow is forwarded to the switch control plane to create and insert a new entry to the translation table. RedPlane-NAT has additional overhead since it needs to request a lease from the state store before updating state. Switch-NAT with the external controller incurs higher 99th percentile latency (185 μs) due to the communication overhead between the switch control plane and the controller and between controller instances (for chain replication) over the slower management network. Server-based versions (FT Server-NAT and FTMB-NAT) have 7–14× higher median latency compared to the switch-based approaches, as packets need to traverse additional hops in the network and they have inherent performance limitations.

Impact on different applications. Next, we evaluate the per-packet processing latency overhead of different RedPlane-enabled applications. As shown in Fig. 9, RedPlane-enabled NAT, firewall, load balancer, EPC-SGW, and heavy-hitter (HH) detection, all have the same 8 μs median latency, identical to that without fault-tolerance. The NAT, firewall, and load balancer are read-centric and update state only when a new flow is created; EPC-SGW is mixed-read/write, and updates state on signaling packets whose frequency is 5% of data packets. HH detection, although it is write-centric, performs periodic state replication asynchronously, so it does not affect the latency. On the other hand, since Sync-Counter updates state and replicates updates *synchronously* for every packet, it adds an additional latency of 20 μs to every packet. 12 μs of this overhead is due

[7] We choose NAT to compare results with those reported in prior work [70].

[8] We implement a simple external controller to emulate SDN controller-based approaches (*e.g.,* Morpheus [68] and Ravana [40]), which communicates with the switch control plane via a 1 Gbps management channel.

[9] We use the latency reported in the original FTMB paper [70] since we were not able to get its full implementation.

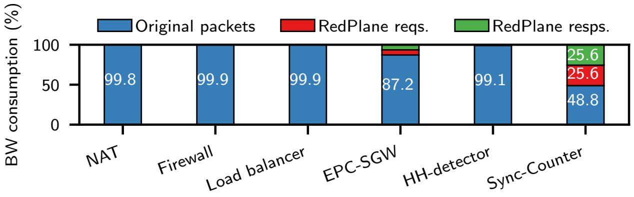


Figure 10: RedPlane replication bandwidth overhead.

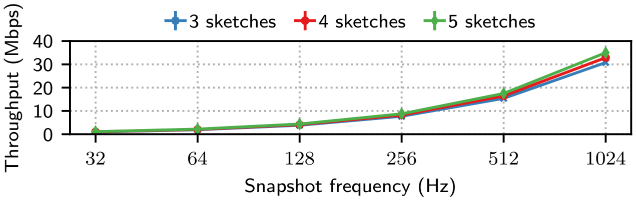


Figure 11: Impact of the frequency of snapshotting on RedPlane-enabled HH-detector.

to the 3-way chain replication used to tolerate state store server failures.

7.2 Bandwidth Overheads

To evaluate network bandwidth overheads, we inject 64-byte packets from three traffic generation servers at≈207.6 Mpps[10] which is the maximum rate that our traffic generator can achieve.

Additional bandwidth consumed. In this experiment, we instrument each application to count the number of bytes it sends and receives, including both original packets and protocol message packets. Fig. 10 shows the ratio of bandwidth used for RedPlane messages to the total traffic. For read-centric applications including NAT, firewall, load balancer, we see that there is almost no bandwidth overhead since RedPlane generates protocol messages only for the first packet of each flow. For EPC-SGW, RedPlane incurs 12.8% overhead since it generates protocol messages for signaling packets, and some of data packets are buffered through the network as described in §5.1. For HH-detector, which asynchronously replicates a snapshot of state for every 1 ms, RedPlane incurs negligible overhead. We also measure the absolute bandwidth overhead for different snapshot frequencies and number of sketches as shown in Fig. 11. For a 1 ms period, it consumes 34.16 Mbps (13.8%). Even with 5 sketches, this is lower than the bandwidth overhead for Sync-Counter (51.2%) because in the latter case RedPlane requests and responses contain both headers and original payload. This result implies that in an extreme case where an application replicates state updates synchronously for every packet, achieving fault-tolerance is expensive. We also analyze the bandwidth overhead at scale (*i.e.,* a topology with more RedPlane switches) for all 6 applications using our analytical model-based simulation, and the result is consistent with Fig. 10 in terms of the percentage overhead.

Throughput impact on applications. In this experiment, we measure the throughput of RedPlane-enabled applications and compare it with the same applications without fault tolerance. We send 64-byte packets from three servers, one from each rack, to one of servers attached to the core switch at ≈207.6 Mpps. In our testbed, the link between an aggregation and a core switch becomes the bottleneck, and we observe that the maximum forwarding rate the aggregation switch can achieve is around 122.5 Mpps. Fig. 12 shows the median throughput of each application with and without RedPlane. Obviously, applications achieve the maximum throughput without RedPlane. With RedPlane, read-centric (NAT, firewall, and load balancer) applications and applications that replicate state updates asynchronously (HH-detector) can achieve the same throughput as their non fault-tolerant counterparts. The RedPlane-enabled EPC-SGW achieves a slightly lower throughput than that of its counterpart, mainly due to some data packets buffered through the network during the replication. The throughput of Sync-Counter becomes nearly half that of its counterpart: we find that it is bottlenecked by the performance of the state store. This suggests that applying a strict consistency mode degrades the throughput of write-centric applications as they are also affected by the performance of the state store.

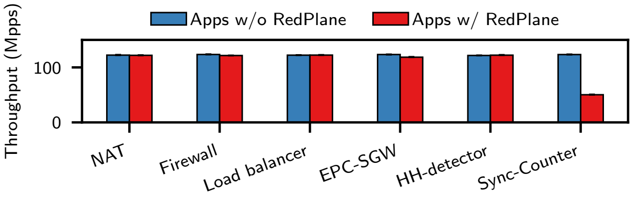


Figure 12: Impact of RedPlane on data plane throughput of RedPlane-enabled applications.

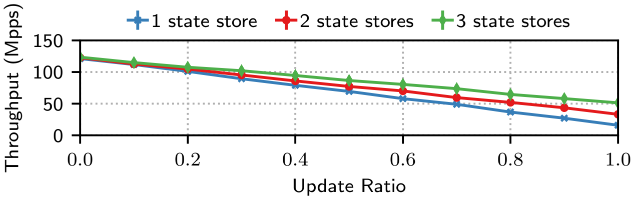


Figure 13: Impact of update ratio on data plane throughput of the RedPlane-enabled key-value store.

Varying update ratios. While most of existing in-switch applications are read-centric or perform asynchronous replication, incurring little overhead, it is important to understand the maximum throughput of applications characterized by different read/write (*i.e.,* update) ratios. For this experiment, we write a simple in-switch key-value store in P4 with RedPlane and generate packets consisting of custom header fields that indicate an operation (read or update), a key, and a value (for updates). We use the same setup as the previous experiment and let each server generate packets based on a predefined update ratio with uniformly distributed random keys. Fig. 13 shows that as the update ratio increases, the throughput degradation depends on the number of state store servers; by adding more servers, we can achieve higher throughput.

7.3 Failover and Recovery

Next, we measure how fast the end-to-end performance can be recovered by RedPlane in the presence of switch failure and recovery. We run `iperf` [15] to measure between two servers, attached to a core switch and a ToR switch respectively. All traffic

[10] Each server generates packets at ≈69.2 Mpps.

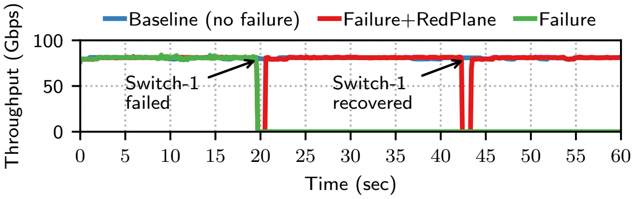


Figure 14: End-to-end throughput changes during failover and recovery with and without RedPlane.

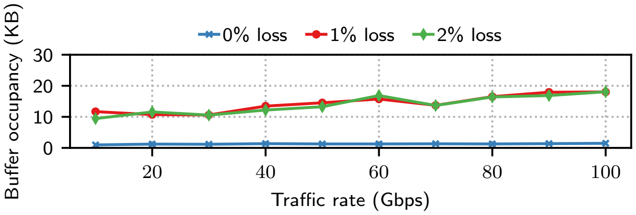


Figure 15: Switch packet buffer occupancy due to request buffering.

passes through a NAT running on the programmable switches. We compare changes in TCP throughput when (1) there is no failure (Baseline), (2) one switch fails without RedPlane (Failure), (3) one switch fails while using RedPlane (Failure+RedPlane).

Fig. 14 shows the results. In a network without RedPlane, when Switch-1 fails, packets are rerouted to another switch and dropped, breaking the TCP connections. In contrast, RedPlane-enabled NAT successfully maintains high throughput when the switch fails and recovers after short disruptions (0.9 and 1.0 seconds). This recovery time is affected both by the core switch's failure detection/rerouting time and RedPlane's lease period (set to 1 second here). Control plane and state store optimizations could further reduce this.

7.4 RedPlane Switch ASIC Resource Usage

Packet buffer usage. In this experiment, we evaluate the overhead of our request buffering mechanism (§5.2). Since RedPlane buffers a replication request until it receives a reply corresponding to the request from the state store, it consumes some amount of the switch packet buffer. Since there is no precise way of measuring the buffer usage in real-time, we instead use the queue depth information provided by the switch ASIC to estimate the upper bound of the buffer occupancy.[11] Specifically, we assume a write-centric application where every incoming packet issues a request (*i.e.,* the most demanding scenario). And we let each request packet record its queue depth information to a P4 register in the data plane and read it from the control plane for every second and take the maximum value. We generate packets from a traffic generation server while varying the traffic rate and the request loss rate.[12] Fig. 15 shows the result. When there is no request loss, the buffer occupancy is less than 1.5 KB even at 100 Gbps traffic rate. As we increase the request loss rate, the buffer usage also grows; when the traffic rate is 100 Gbps and ≈2% of requests are lost, our buffering mechanism consumes at most 18 KB, which is acceptable for a given a few tens of MB of the packet buffer in the switch ASIC.

Other ASIC resource usage. We also measure the usage of other ASIC resources consumed by RedPlane data plane for 100K concurrent flows (using the Tofino-P4 compiler's output), expressed relative to each application's baseline usage. Ample resources remain: SRAM is the most used (13.2%), and all others are less than 10% (details in Appendix E). Scaling up concurrent flows would increase only SRAM usage, as it stores per-flow information (lease expiration time, current sequence number, and last acknowledged sequence number).

8 Related Work

In-switch applications. Recent efforts have shown that offloading to programmable switches enhances performance. For example, offloading the sequencer [46], key-value cache [38, 52], and coordination service [37] improves the performance of distributed systems. However, these applications can lose their state due to switch failures. RedPlane can help make them fault-tolerant or simplify their designs.

Fault-tolerance and state management for NFs. Fault-tolerance for NFs or middleboxes has been addressed by prior systems like Pico [63] and FTMB [70]. When an NF instance fails, the state of the failed NF is recovered through checkpoint or rollback recovery on a new NF instance. These approaches cannot be applied directly to the switch data plane (§2.2). Previous work on state management for stateful NFs uses local or remote storage to manage NF state [31, 64, 75]. However, these APIs target planned state migration rather than unplanned failures. Similar work (again, targeting planned migration) has also been proposed for router migration [41].

External memory for switches. Recent work shares our approach of using servers' memory as external storage for switch state [42], but towards a different goal: allow switches to handle state larger than their on-device memory. It does not address fault tolerance or multi-writer consistency.

Switch-based reliability protocols. Other recent work runs coordination protocols between switches to build reliable storage [29, 37]. Our goal is conceptually different – to replicate state for in-switch applications rather than provide a networked storage service – but uses some similar mechanisms, like network sequencing [46].

9 Conclusions

While many recent efforts have demonstrated the potential benefits of running datacenter functions on programmable switches, we argue that there is one critical missing piece in current designs, which is fault tolerance. To address this issue, in this paper, we present RedPlane, which provides a fault tolerant state store abstraction for in-switch applications. We formally define a linearizability-based correctness model for a replicated switch data plane state and build a practical replication protocol based on it. Our evaluation with various stateful applications on a real testbed shows that RedPlane can support fault-tolerance with minimal performance and resource overheads and enable end-to-end performance to quickly recover from switch failures.

Ethics: This work does not raise any ethical issues.

[11] It is a per-packet queue depth measured when a packet is dequeued from the buffer, and the Tofino ASIC provides this information as an intrinsic metadata that can be accessed at the egress pipeline.

[12] We emulate the request loss by dropping requests at a certain probability at the switch.

Acknowledgments

We would like to thank the anonymous SIGCOMM reviewers and our shepherd, Mythili Vutukuru, for their insightful comments and constructive feedback. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by NSF award 1700521. Daehyeok Kim was also supported by the Microsoft Research PhD Fellowship.

References

[1] 2009. 2009-M57-Patents packet trace. http://downloads.digitalcorpora.org/corpora/scenarios/2009-m57-patents/net/.

[2] 2010. Data Set for IMC 2010 Data Center Measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html.

[3] 2011. RDMA Aware Networks Programming User Manual. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[4] 2012. A signaling storm is gathering - Is your packet core ready? https://www.nokia.com/blog/a-signaling-storm-is-gathering-is-your-packet-core-ready/.

[5] 2017. vEPC Acceleration Using Agilio SmartNICs. https://www.netronome.com/media/documents/SB_vEPC.pdf.

[6] 2018. Advanced Network Telemetry. https://www.barefootnetworks.com/use-cases/ad-telemetry/.

[7] 2018. Barefoot Networks Unveils Tofino 2, the Next Generation of the World's First Fully P4-Programmable Network Switch ASICs. https://www.barefootnetworks.com/press-releases/barefoot-networks-unveils-tofino-2-the-next-generation-of-the-worlds-first-fully-p4-programmable-network-switch-asics/..

[8] 2018. Cavium Xpliant Ethernet Switches. https://www.cavium.com/xpliant-ethernet-switch-product-family.html.

[9] 2018. Offloading VNFs to programmable switches using P4. https://wiki.onosproject.org/download/attachments/12420314/p4-vnf-offloading-ons2018.pdf.

[10] 2019. Cisco Visual Networking Index. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-738429.html.

[11] 2019. $P4_{16}$ Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.2.0.html.

[12] 2020. Barefoot P4 Studio. https://www.barefootnetworks.com/products/brief-p4-studio/.

[13] 2020. Cisco Silicon One Q200 and Q200L Processors Data Sheet. https://www.cisco.com/c/en/us/solutions/collateral/silicon-one/datasheet-c78-744312.html.

[14] 2020. Intel Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[15] 2020. iperf(1) - Linux man page. https://linux.die.net/man/1/iperf.

[16] 2020. Redis. https://redis.io/.

[17] 2020. The Evolved Packet Core. https://www.3gpp.org/technologies/keywords-acronyms/100-the-evolved-packet-core.

[18] 2020. The TLA+ Home Page. https://lamport.azurewebsites.net/tla/tla.html.

[19] 2020. Trident4 / BCM56880 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.

[20] 2021. RedPlane Public Repository. https://github.com/daehyeok-kim/redplane-public.

[21] 3GPP. 2012. 3GPP TS 23.007: Restoration procedures.

[22] Alexey Andreyev. 2014. Introducing data center fabric, the next-generation Facebook data center network. https://engineering.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/.

[23] Arista Networks. 2020. Arista 7170 Series. https://www.arista.com/en/products/7170-series.

[24] Barefoot Networks. 2017. Tofino Switch Architecture Specification (accessible under NDA).

[25] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426.

[26] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *ACM SIGCOMM* (2013).

[27] David Clark. 1988. The design philosophy of the DARPA Internet protocols. In *Symposium proceedings on Communications architectures and protocols* (1988). 106–114.

[28] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding Frequent Items in Data Streams. *Proceedings of the VLDB Endowment* 1, 2 (2008).

[29] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé. 2020. P4xos: Consensus as a Network Service. *IEEE/ACM Transactions on Networking* (2020).

[30] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2015. Maglev: A Fast and Reliable Software Network Load Balancer. In *USENIX NSDI* (2015).

[31] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling Innovation in Network Function Control. In *ACM SIGCOMM* (2014).

[32] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *ACM SOSP* (2003).

[33] Cary Gray and David Cheriton. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review* 23, 5 (1989).

[34] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: a scalable and flexible data center network. In *ACM SIGCOMM* (2009).

[35] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven Streaming Network Telemetry. In *ACM SIGCOMM* (2018).

[36] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990).

[37] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX NSDI* (2018).

[38] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM SOSP* (2017).

[39] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable Load Balancing Using Programmable Data Planes. In *ACM SOSR* (2016).

[40] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. 2015. Ravana: Controller fault-tolerance in software-defined networking. In *ACM SOSR* (2015).

[41] Eric Keller, Jennifer Rexford, and Jacobus E van der Merwe. 2010. Seamless BGP Migration with Router Grafting.. In *NSDI* (2010).

[42] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *ACM SIGCOMM* (2020).

[43] Petr Lapukhov, Ariff Premji, and Jon Mitchell. 2016. Use of BGP for routing in large-scale data centers. *Internet Requests for Comments RFC Editor RFC* 7938 (2016).

[44] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John K. Ousterhout. 2015. Implementing linearizability at large scale and low latency. In *ACM SOSP* (2015).

[45] Jialin Li, Ellis Michael, and Dan RK Ports. 2017. Eris: Coordination-free consistent transactions using in-network concurrency control. In *ACM SOSP* (2017).

[46] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *USENIX OSDI* (2016).

[47] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan RK Ports. 2020. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *USENIX OSDI* (2020).

[48] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. In *USENIX NSDI* (2014).

[49] Barbara Liskov and James Cowling. 2012. *Viewstamped Replication Revisited*. Technical Report MIT-CSAIL-TR-2012-021. MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA.

[50] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. 2017. Crystalnet: Faithfully emulating large production networks. In *ACM SOSP* (2017).

[51] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. 2013. F10: A fault-tolerant engineered network. In *USENIX NSDI* (2013).

[52] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *USENIX FAST* (2019).

[53] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *ACM SIGCOMM* (2016).

[54] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. 2018. A large scale study of data center network reliability. In *ACM IMC*.

[55] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *ACM SIGCOMM* (2017).

[56] Ali Mohammadkhan, KK Ramakrishnan, Ashok Sunder Rajan, and Christian Maciocco. 2016. Considerations for re-designing the cellular infrastructure exploiting software-based networks. In *IEEE ICNP* (2016).

[57] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX ATC* (2014).

[58] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast Crash Recovery in RAMCloud. In *ACM SOSP* (2011).

[59] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 33, 3 (2015).

[60] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: Cloud Scale Load Balancing. In *ACM SIGCOMM* (2013).

[61] Larry Peterson, Tom Anderson, Sachin Katti, Nick McKeown, Guru Parulkar, Jennifer Rexford, Mahadev Satyanarayanan, Oguz Sunay, and Amin Vahdat. 2019. Democratizing the network edge. *ACM SIGCOMM Computer Communication Review* 49, 2 (2019).

[62] Zafar Ayyub Qazi, Melvin Walls, Aurojit Panda, Vyas Sekar, Sylvia Ratnasamy, and Scott Shenker. 2017. A High Performance Packet Core for Next Generation Cellular Networks. In *ACM SIGCOMM* (2017).

[63] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. 2013. Pico replication: A high availability framework for middleboxes. In *ACM SoCC* (2013).

[64] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *USENIX NSDI* (2013).

[65] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *ACM SIGCOMM* (2015).

[66] Jerome H Saltzer, David P Reed, and David D Clark. 1984. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)* 2, 4 (1984).

[67] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. (2021).

[68] Karla Saur, Joseph Collard, Nate Foster, Arjun Guha, Laurent Vanbever, and Michael Hicks. 2016. Safe and flexible controller upgrades for SDNs. In *ACM SOSR* (2016).

[69] Rinku Shah, Vikas Kumar, Mythili Vutukuru, and Purushottam Kulkarni. 2020. TurboEPC: Leveraging Dataplane Programmability to Accelerate the Mobile Packet Core. In *ACM SOSR* (2020).

[70] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, et al. 2015. Rollback-recovery for middleboxes. In *ACM SIGCOMM* (2015).

[71] 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 Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *ACM SIGCOMM* (2015).

[72] L. Tang, Q. Huang, and P. P. C. Lee. 2020. SpreadSketch: Toward Invertible and Network-Wide Detection of Superspreaders. In *IEEE INFOCOM 2020* (2020).

[73] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. 2020. Cheetah: Accelerating Database Queries with Switch Pruning. In *ACM SIGMOD*.

[74] Robbert van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *USENIX OSDI* (2004).

[75] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic Scaling of Stateful Network Functions. In *USENIX NSDI* (2018).

[76] Jiarong Xing, Wenqing Wu, and Ang Chen. 2021. Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries. In *USENIX Security* (2021).

[77] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. 2020. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *ISOC NDSS* (2020).

[78] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan RK Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proceedings of the VLDB Endowment* 13, 3 (2019).

Note: Appendices are supporting material that has not been peer-reviewed.

A Details of Lazy Snapshotting

Algorithm 1: Lazy snapshotting

```
/* 1-bit variable indicating the current active buffer */
active_buffer ⟵ 0;
/* array of 1-bit variables indicating which buffer has has lastly been
   updated for a certain slot */
last_updated_buffer[0...REGISTER_SIZE] ⟵ 0;
/* two copies of the replicated data structure (e.g., a sketch in this
   example) */
pair<int,int> sketch [0...REGISTER_SIZE] ⟵ 0;
Upon receiving a packet (pkt):
/* is this the first pkt of a snapshot read burst? */
if pkt.type = SNAPSHOT_READ and pkt.index = 0 then
    /* if so, swap the active buffer */
    active_buffer ⟵ swap_active_buffer();
else
    /* if not, get the current active buffer */
    active_buffer ⟵ get_active_buffer();
/* which buffer was lastly updated for this index? */
last_updated_buffer_for_index ⟵ update_last_updated_buffer(pkt.index,active_buffer);
/* for a regular packet */
if pkt.type = SKETCH_UPDATE then
    /* is this the first time this buffer has been touched since we took
       a snapshot? */
    if active_buffer ≠ last_updated_buffer_for_index then
        /* if so, copy data from the inactive buffer before updating */
        if active_buffer = 0 then
            pkt.result ⟵ copy_update_and_read_buffer_0(pkt.index, pkt.update);
        else
            pkt.result ⟵ copy_update_and_read_buffer_1(pkt.index, pkt.update);
    /* if not, some other packet has touched this buffer since we took a
       snapshot, so just do update */
    else
        if active_buffer = 0 then
            pkt.result ⟵ update_and_read_buffer_0(pkt.index,pkt.update);
        else
            pkt.result ⟵ update_and_read_buffer_1(pkt.index, pkt.update);
/* for a snapshot read packet */
else if pkt.type = SNAPSHOT_READ then
    pkt.update = 0; /* is this the first time this buffer has been touched
       since we took a snapshot? */
    if active_buffer ≠ last_updated_buffer_for_index then
        /* if so, copy data from the inactive buffer before updating */
        if active_buffer = 0 then
            pkt.result ⟵ copy_update_and_read_buffer_0(pkt.index, pkt.update);
        else
            pkt.result ⟵ copy_update_and_read_buffer_1(pkt.index, pkt.update);
    /* if not, some other packet has touched this buffer since we took a
       snapshot, so just do read */
    else
        if active_buffer = 0 then
            pkt.result ⟵ update_and_read_buffer_1(pkt.index, pkt.update);
        else
            pkt.result ⟵ update_and_read_buffer_0(pkt.index, pkt.update);
```

Algorithm 1 shows the pseudocode for lazy snapshotting described in §5.4. We implement this logic in P4 to provide a basic sketch with 64×32-bit slots. As explained in §6, we implement count-min sketches using three of this sketch.

B P4 Skeleton Code of RedPlane-enabled Application

As mentioned in §6 of our paper, we expose RedPlane APIs as modules in P4. Fig. 16 illustrates how the P4 implementation of RedPlane-enabled NAT looks like. Developers need to include the P4 file of RedPlane core APIs (line 1) and the P4 file of their original application code (line 2). Lines highlighted in red shows initialization and the use of the RedPlane ingress and egress control block instances (line 5, 9, 20, and 24). And the lines highlighted in bold blue indicates modules of the original NAT program (line 6 and 11). Since NAT does not update state in the data plane (i.e., read-centric), no modification is needed to their original P4 implementation. Other applications (firewall, load balancer, HH-detector, etc.) introduced in the paper can be implemented in a similar way.

```
1   #include "redplane_core.p4" // RedPlane core API
2   #include "nat.p4"           // developer’s NAT program
3
4   control Ingress (headers hdr, metadata meta) {
5       RedPlaneIngress() redplane_ingress;
6       NAT_Ingress() nat_ingress;
7       L3_Routing() l3_routing;
8       apply {
9           redplane_ingress.apply(hdr, meta);
10          if (meta.is_normal_pkt == true) {
11              nat_ingress.apply (hdr, meta);
12          }
13          if (meta.is_normal_pkt == true ||
14              meta.is_piggybacked == true) {
15              l3_routing_ingress.apply (hdr, meta);
16          }
17      }
18  }
19  control Egress (headers hdr, metadata meta) {
20      RedPlaneEgress() redplane_egress;
21      apply {
22          if (meta.is_redplane_req == true ||
23              meta.is_redplane_ack == true) {
24              redplane_egress.apply(hdr, meta);
25          }
26      }
27  }
28
29  Pipeline(
30     IngressParser(),
31     Ingress(),
32     IngressDeparser(),
33     EgressParser(),
34     Egress(),
35     EgressDeparser()
36  ) pipe;
37
38  Switch(pipe) main;
```

Figure 16: The main part of P4 implementation of RedPlane-enabled NAT.

C TLA+ Specification of RedPlane Protocol

We write a TLA+ specification of RedPlane protocol to model-check its correctness.

```
MODULE redplane_protocol
EXTENDS Integers, Sequences, TLC, FiniteSets
CONSTANTS NULL, SWITCHES, LEASE_PERIOD, TOTAL_PKTS


VARIABLES query, request_queue, SwitchPacketQueue, RemainingLeasePeriod,
owner, up, active, AliveNum, global_seqnum, pc


Exists(val) ≜ val ≠ NULL
RequestingSwitches ≜ {sw ∈ SWITCHES : Exists(query[sw]) ∧ query[sw].type = "request"}

VARIABLES switch, q, seqnum, round, upSwitches, sent_pkts

vars ≜ ⟨query, request_queue, SwitchPacketQueue, RemainingLeasePeriod,
owner, up, active, AliveNum, global_seqnum, pc, switch, q, seqnum,
round, upSwitches, sent_pkts⟩

ProcSet ≜ {"StateStore"} ∪ (SWITCHES) ∪ {"LeaseTimer"} ∪ {"pktgen"}

Init ≜  Global variables
∧ query = [sw ∈ SWITCHES ↦ NULL]
∧ request_queue = ⟨⟩
∧ SwitchPacketQueue = [sw ∈ SWITCHES ↦ 0]
∧ RemainingLeasePeriod = [sw ∈ SWITCHES ↦ 0]
∧ owner = NULL
∧ up = [sw ∈ SWITCHES ↦ TRUE]
∧ active = [sw ∈ SWITCHES ↦ FALSE]
∧ AliveNum = Cardinality(SWITCHES)
∧ global_seqnum = 0

∧ switch = NULL
∧ q = NULL

∧ seqnum = [self ∈ SWITCHES ↦ 0]
∧ round = [self ∈ SWITCHES ↦ 0]

∧ upSwitches = {}
∧ sent_pkts = 0
∧ pc = [self ∈ ProcSet ↦ CASE self = "StateStore" → "START_STORE"
□self ∈ SWITCHES → "START_SWITCH"
□self = "LeaseTimer" → "START_TIMER"
□self = "pktgen" → "START_PKTGEN"]

START_STORE ≜  ∧ pc["StateStore"] = "START_STORE"
```

```
$\land\ pc' = [pc\ \text{EXCEPT}\ ![\text{"StateStore"}] = \text{"STORE\_PROCESSING"}]$
$\land\ \text{UNCHANGED}\ \langle query,\ request\_queue,\ SwitchPacketQueue,$
$RemainingLeasePeriod,\ owner,\ up,\ active,$
$AliveNum,\ global\_seqnum,\ switch,\ q,\ seqnum,$
$round,\ upSwitches,\ sent\_pkts\rangle$

$STORE\_PROCESSING\ \triangleq\ \land\ pc[\text{"StateStore"}] = \text{"STORE\_PROCESSING"}$
$\land\ \text{IF}\ request\_queue \neq \langle\rangle$
 $\text{THEN}\ \land\ switch' = Head(request\_queue)$
$\land\ request\_queue' = Tail(request\_queue)$
$\land\ q' = query[switch']$
$\land\ \text{IF}\ q'.lease\_request = \text{"new"}$
 $\text{THEN}\ \land\ \text{IF}\ owner \neq NULL$
 $\text{THEN}\ \land\ pc' = [pc\ \text{EXCEPT}\ ![\text{"StateStore"}] = \text{"BUFFERING"}]$
 $\text{ELSE}\ \land\ pc' = [pc\ \text{EXCEPT}\ ![\text{"StateStore"}] = \text{"TRANSFER\_LEASE"}]$
 $\text{ELSE}\ \land\ \text{IF}\ q'.lease\_request = \text{"renew"}$
 $\text{THEN}\ \land\ pc' = [pc\ \text{EXCEPT}\ ![\text{"StateStore"}] = \text{"RENEW\_LEASE"}]$
 $\text{ELSE}\ \land\ pc' = [pc\ \text{EXCEPT}\ ![\text{"StateStore"}] = \text{"START\_STORE"}]$
 $\text{ELSE}\ \land\ pc' = [pc\ \text{EXCEPT}\ ![\text{"StateStore"}] = \text{"START\_STORE"}]$
$\land\ \text{UNCHANGED}\ \langle request\_queue,\ switch,\ q\rangle$
$\land\ \text{UNCHANGED}\ \langle query,\ SwitchPacketQueue,$
$RemainingLeasePeriod,\ owner,\ up,\ active,$
$AliveNum,\ global\_seqnum,\ seqnum,\ round,$
$upSwitches,\ sent\_pkts\rangle$

$TRANSFER\_LEASE\ \triangleq\ \land\ pc[\text{"StateStore"}] = \text{"TRANSFER\_LEASE"}$
$\land\ query' = [query\ \text{EXCEPT}\ ![switch] = [type \mapsto \text{"response"}]\ @@\ ([last\_seqnum \mapsto global\_seqnum])]$
$\land\ RemainingLeasePeriod' = [RemainingLeasePeriod\ \text{EXCEPT}\ ![switch] = LEASE\_PERIOD]$
$\land\ owner' = switch$
$\land\ pc' = [pc\ \text{EXCEPT}\ ![\text{"StateStore"}] = \text{"START\_STORE"}]$
$\land\ \text{UNCHANGED}\ \langle request\_queue,\ SwitchPacketQueue,\ up,\ active,$
$AliveNum,\ global\_seqnum,\ switch,\ q,\ seqnum,$
$round,\ upSwitches,\ sent\_pkts\rangle$

$BUFFERING\ \triangleq\ \land\ pc[\text{"StateStore"}] = \text{"BUFFERING"}$
$\land\ request\_queue' = Append(request\_queue,\ switch)$
$\land\ pc' = [pc\ \text{EXCEPT}\ ![\text{"StateStore"}] = \text{"STORE\_PROCESSING"}]$
$\land\ \text{UNCHANGED}\ \langle query,\ SwitchPacketQueue,\ RemainingLeasePeriod,$
$owner,\ up,\ active,\ AliveNum,\ global\_seqnum,$
$switch,\ q,\ seqnum,\ round,\ upSwitches,\ sent\_pkts\rangle$

$RENEW\_LEASE\ \triangleq\ \land\ pc[\text{"StateStore"}] = \text{"RENEW\_LEASE"}$
$\land\ global\_seqnum' = q.write\_seq$
$\land\ query' = [query\ \text{EXCEPT}\ ![switch] = [type \mapsto \text{"response"}]\ @@\ ([last\_seqnum \mapsto global\_seqnum'])]$
$\land\ RemainingLeasePeriod' = [RemainingLeasePeriod\ \text{EXCEPT}\ ![switch] = LEASE\_PERIOD]$
```

$\land\ owner' = switch$
$\land\ pc' = [pc\ \textsc{except}\ ![\text{“StateStore”}] = \text{“START\_STORE”}]$
$\land\ \textsc{unchanged}\ \langle request\_queue,\ SwitchPacketQueue,\ up,\ active,$
$AliveNum,\ switch,\ q,\ seqnum,\ round,\ upSwitches,$
$sent\_pkts\rangle$

$statestore\ \triangleq\ START\_STORE \lor STORE\_PROCESSING \lor TRANSFER\_LEASE$
$\lor\ BUFFERING \lor RENEW\_LEASE$

$START\_SWITCH(self)\ \triangleq\ \land\ pc[self] = \text{“START\_SWITCH”}$
$\land\ \lor\ \land\ (up[self] \land SwitchPacketQueue[self] > 0)$
$\land\ active' = [active\ \textsc{except}\ ![self] = \textsc{true}]$
$\land\ \textsc{if}\ RemainingLeasePeriod[self] = 0$
$\textsc{then}\ \land\ pc' = [pc\ \textsc{except}\ ![self] = \text{“NO\_LEASE”}]$
$\textsc{else}\ \land\ pc' = [pc\ \textsc{except}\ ![self] = \text{“HAS\_LEASE”}]$
$\lor\ \land\ pc' = [pc\ \textsc{except}\ ![self] = \text{“SW\_FAILURE”}]$
$\land\ \textsc{unchanged}\ active$
$\land\ \textsc{unchanged}\ \langle query,\ request\_queue,\ SwitchPacketQueue,$
$RemainingLeasePeriod,\ owner,\ up,$
$AliveNum,\ global\_seqnum,\ switch,\ q,$
$seqnum,\ round,\ upSwitches,\ sent\_pkts\rangle$

$NO\_LEASE(self)\ \triangleq\ \land\ pc[self] = \text{“NO\_LEASE”}$
$\land\ query' = [query\ \textsc{except}\ ![self] = [type \mapsto \text{“request”}]\ @@\ ([lease\_request \mapsto \text{“new”}])]$
$\land\ request\_queue' = Append(request\_queue,\ self)$
$\land\ pc' = [pc\ \textsc{except}\ ![self] = \text{“WAIT\_LEASE\_RESPONSE”}]$
$\land\ \textsc{unchanged}\ \langle SwitchPacketQueue,\ RemainingLeasePeriod,$
$owner,\ up,\ active,\ AliveNum,\ global\_seqnum,$
$switch,\ q,\ seqnum,\ round,\ upSwitches,$
$sent\_pkts\rangle$

$WAIT\_LEASE\_RESPONSE(self)\ \triangleq\ \land\ pc[self] = \text{“WAIT\_LEASE\_RESPONSE”}$
$\land\ query[self].type = \text{“response”}$
$\land\ seqnum' = [seqnum\ \textsc{except}\ ![self] = query[self].last\_seqnum]$
$\land\ query' = [query\ \textsc{except}\ ![self] = NULL]$
$\land\ pc' = [pc\ \textsc{except}\ ![self] = \text{“HAS\_LEASE”}]$
$\land\ \textsc{unchanged}\ \langle request\_queue,\ SwitchPacketQueue,$
$RemainingLeasePeriod,\ owner,\ up,$
$active,\ AliveNum,\ global\_seqnum,$
$switch,\ q,\ round,\ upSwitches,$
$sent\_pkts\rangle$

$HAS\_LEASE(self)\ \triangleq\ \land\ pc[self] = \text{“HAS\_LEASE”}$
$\land\ seqnum' = [seqnum\ \textsc{except}\ ![self] = seqnum[self] + 1]$
$\land\ query' = [query\ \textsc{except}\ ![self] = [type \mapsto \text{“request”}]\ @@\ ([lease\_request \mapsto \text{“renew”},\ write\_seq \mapsto seqnum'[se$

$\land$ $request\_queue' = Append(request\_queue, self)$
$\land$ $pc' = [pc$ EXCEPT $![self] =$ "WAIT_WRITE_RESPONSE"$]$
$\land$ UNCHANGED $\langle SwitchPacketQueue, RemainingLeasePeriod,$
$owner, up, active, AliveNum, global\_seqnum,$
$switch, q, round, upSwitches, sent\_pkts\rangle$

$WAIT\_WRITE\_RESPONSE(self) \triangleq$ $\land$ $pc[self] =$ "WAIT_WRITE_RESPONSE"
$\land$ $query[self].type =$ "response"
$\land$ $Assert(seqnum[self] = query[self].last\_seqnum,$
"assertion failed."$)$
$\land$ $query' = [query$ EXCEPT $![self] = NULL]$
$\land$ $active' = [active$ EXCEPT $![self] =$ FALSE$]$
$\land$ $SwitchPacketQueue' = [SwitchPacketQueue$ EXCEPT $![self] = SwitchPacketQueue[self] - 1]$
$\land$ $pc' = [pc$ EXCEPT $![self] =$ "START_SWITCH"$]$
$\land$ UNCHANGED $\langle request\_queue,$
$RemainingLeasePeriod, owner, up,$
$AliveNum, global\_seqnum, switch,$
$q, seqnum, round, upSwitches,$
$sent\_pkts\rangle$

$SW\_FAILURE(self) \triangleq$ $\land$ $pc[self] =$ "SW_FAILURE"
$\land$ IF $AliveNum > 1 \land up[self] =$ TRUE
THEN $\land$ $up' = [up$ EXCEPT $![self] =$ FALSE$]$
$\land$ $AliveNum' = AliveNum - 1$
$\land$ $query' = query$
ELSE $\land$ IF $up[self] =$ FALSE
THEN $\land$ $up' = [up$ EXCEPT $![self] =$ TRUE$]$
$\land$ $query' = [query$ EXCEPT $![self] = NULL]$
$\land$ $AliveNum' = AliveNum + 1$
ELSE $\land$ TRUE
$\land$ UNCHANGED $\langle query, up, AliveNum\rangle$
$\land$ $pc' = [pc$ EXCEPT $![self] =$ "START_SWITCH"$]$
$\land$ UNCHANGED $\langle request\_queue, SwitchPacketQueue,$
$RemainingLeasePeriod, owner, active,$
$global\_seqnum, switch, q, seqnum, round,$
$upSwitches, sent\_pkts\rangle$

$switch\_(self) \triangleq START\_SWITCH(self) \lor NO\_LEASE(self)$
$\lor$ $WAIT\_LEASE\_RESPONSE(self) \lor HAS\_LEASE(self)$
$\lor$ $WAIT\_WRITE\_RESPONSE(self) \lor SW\_FAILURE(self)$

$START\_TIMER \triangleq$ $\land$ $pc[$"LeaseTimer"$] =$ "START_TIMER"
$\land$ $owner \neq NULL$
$\land$ IF $RemainingLeasePeriod[owner] > 0 \land active[owner] =$ FALSE
THEN $\land$ $RemainingLeasePeriod' = [RemainingLeasePeriod$ EXCEPT $![owner] = RemainingLeasePeriod[owne$

```
$\land\ owner' = owner$
 ELSE $\land$ IF $RemainingLeasePeriod[owner] = 0$
 THEN $\land\ owner' = NULL$
 ELSE $\land$ TRUE
$\land\ owner' = owner$
$\land$ UNCHANGED $RemainingLeasePeriod$
$\land\ pc' = [pc$ EXCEPT $![$"LeaseTimer"$] =$ "START_TIMER"$]$
$\land$ UNCHANGED $\langle query, request\_queue, SwitchPacketQueue, up,$
$active, AliveNum, global\_seqnum, switch, q,$
$seqnum, round, upSwitches, sent\_pkts\rangle$

$expirationTimer \triangleq START\_TIMER$

$START\_PKTGEN \triangleq \land\ pc[$"pktgen"$] =$ "START_PKTGEN"
$\land$ IF $sent\_pkts < TOTAL\_PKTS$
 THEN $\land\ AliveNum \geq 1$
$\land\ upSwitches' = \{sw \in SWITCHES : up[sw]\}$
$\land\ \exists\, sw \in upSwitches' :$
$SwitchPacketQueue' = [SwitchPacketQueue$ EXCEPT $![sw] = SwitchPacketQueue[sw] + 1]$
$\land\ sent\_pkts' = sent\_pkts + 1$
$\land\ pc' = [pc$ EXCEPT $![$"pktgen"$] =$ "START_PKTGEN"$]$
 ELSE $\land\ pc' = [pc$ EXCEPT $![$"pktgen"$] =$ "Done"$]$
$\land$ UNCHANGED $\langle SwitchPacketQueue, upSwitches,$
$sent\_pkts\rangle$
$\land$ UNCHANGED $\langle query, request\_queue, RemainingLeasePeriod,$
$owner, up, active, AliveNum, global\_seqnum,$
$switch, q, seqnum, round\rangle$

$packetGen \triangleq START\_PKTGEN$

$Next \triangleq statestore \lor expirationTimer \lor packetGen$
$\lor\ (\exists\, self \in SWITCHES : switch\_(self))$

$Spec \triangleq \land\ Init \land \Box[Next]_{vars}$
$\land\ \mathrm{WF}_{vars}(statestore)$
$\land\ \forall\, self \in SWITCHES : \mathrm{WF}_{vars}(switch\_(self))$
$\land\ \mathrm{WF}_{vars}(expirationTimer)$
$\land\ \mathrm{WF}_{vars}(packetGen)$


$AtLeastOneAliveSwitch \triangleq$
$\land\ AliveNum \geq 1$
$\land\ \exists\, sw \in SWITCHES : up[sw] =$ TRUE

$SingleOwnerInvariant \triangleq$
```

$\forall\, sw \in SWITCHES :$
$sw \neq owner \Rightarrow RemainingLeasePeriod[sw] = 0$

$Liveness \;\triangleq$
$\lor \forall\, sw \in SWITCHES :$
$(query[sw] \neq NULL \land query[sw].type = \text{“request”}) \leadsto$
$owner = sw$

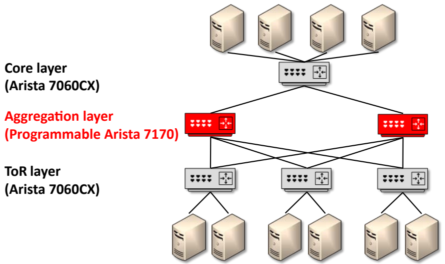


Figure 17: Three-layer network testbed for experiments.

Resource	Additional usage
Match Crossbar	5.3%
Meter ALU	8.3%
Gateway	9.9%
SRAM	13.2%
TCAM	11.8%
VLIW Instruction	5.5%
Hash Bits	3.7%

Table 2: Switch ASIC resources used by RedPlane.

D Testbed

We evaluate RedPlane on a testbed consisting of six commodity switches (including two programmable ones) and servers, as shown in Fig. 17.

E Detailed Switch ASIC Resource Utilization

Table 2 shows the additional switch ASIC resource consumption of RedPlane for 100K concurrent flows (using the P4 compiler's output), expressed relative to each application's baseline usage. Overall, there are ample resources remaining to implement other functions along with RedPlane. RedPlane uses TCAM to implement acknowledgment processing and request timeout management, which need range matches. In terms of scale vs. number of concurrent flows, only the SRAM usage would increase proportional to the number of flows as it stores per-flow information (lease expiration time, current sequence number, and last acknowledged sequence number).