# RedPlane: Fast and Consistent Fault Tolerance for Stateful In-Switch Applications

*Daehyeok Kim, Jacob Nelson, Dan Ports, Vyas Sekar, Srinivasan Seshan*

## Abstract

Many recent efforts have demonstrated the performance benefits of running datacenter functions (e.g., NATs, load balancers, key-value stores, consensus protocols) on programmable switches. However, a key missing piece is dealing with 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 *replicated data plane* for stateful in-switch applications providing location independent, fault tolerant state storage. This provides in-switch applications consistent access to their state, even if the switch they run on fails or traffic is rerouted to a new location. We address key challenges in devising new correctness abstractions for in-switch applications and a practical, provably correct replication protocol. Our evaluations show that RedPlane incurs negligible overhead and enables end-to-end applications to rapidly recover from switch failures.

## 1 Introduction

Today's data center network switches are no longer simple stateless packet forwarders. They are capable of implementing many of the increasingly sophisticated network functions that underlie modern cloud networks – NATs, firewalls, and load balancers, to name a few [4, 24, 37]. More radically, armed with new programmable switching platforms, there have been attempts to employ in-network computation to accelerate distributed applications [30, 44, 48], and cloud providers have started deploying such platforms in their production networks [5].

Such stateful processing in network switches leads to a new challenge: fault tolerance. Classic network designs followed the end-to-end principle [43], keeping state on the end hosts to enable fate sharing. When switches are stateless, recovering from their failure is simply a matter of finding a new communication path. But this era is long past: today's data center is rife with stateful middleboxes and network functions that challenge this paradigm. For example, the failure of a switch running a load balancer may cause the loss of its forwarding state, breaking thousands of active connections.

We are forced to reconsider fault tolerance for in-switch processing – something that has so far been done in ad hoc, application-specific ways. In this paper, we offer a generic mechanism for recovering in-switch applications after failure. Data center networks are already engineered with redundant network paths [19, 42, 47]. Our goal 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 a uniquely difficult problem because of its scale: a data center switch can process up to 4.8 *billion* packets per second [11], each of which may update state. While classic distributed systems techniques like checkpointing and active replication have been applied successfully to software middleboxes [40, 46], they do not translate well to the in-switch deployment model. Not only do programmable switch data planes operate at far higher throughput (Tbps rather than Gbps), they also have more constraints on compute and storage. As we will describe later, existing approaches to fault tolerance either suffer from poor performance or consume excessive switch resources.

In this paper, we introduce *RedPlane*, a *replicated data plane* for stateful in-switch applications, which provides an abstraction of location independent, fault tolerant state storage. An application retains consistent access to its state, even if the switch it runs on fails or traffic is rerouted to a new location. RedPlane achieves this through a data plane centric replication mechanism that continuously replicates state updates to an external state store. This state store is implemented using traditional servers, allowing it to use relatively cheap DRAM rather than expensive switch memory. Note that running entirely in the data plane channel is key to keep up with the switch's full processing speed.

Since we need to implement replication mechanisms in a switch data plane, where the traffic rate is high but memory and other resources are limited, RedPlane departs from classical replication systems in a few key ways:

- We observe that network packet processing permits a weaker consistency model than general replication systems, as networks are already permitted to be lossy.
- We build a lightweight sequencing protocol to ensure that state updates are processed in the correct order, even though many of the usual mechanisms (e.g., TCP connections) are unavailable on the switch data plane.
- Memory constraints mean that we cannot buffer output until the corresponding state updates are durably recorded, RedPlane uses the network itself as temporary storage by piggybacking packet contents on coordination messages.
- We use a lease-based state management protocol to provide consistency without coordinating on every state access, and to migrate state between different switches as needed.

We implement a prototype of RedPlane in P4 [7] and showcase that with RedPlane APIs, developers can make their P4 program fault-tolerant with few modifications. We evaluate it

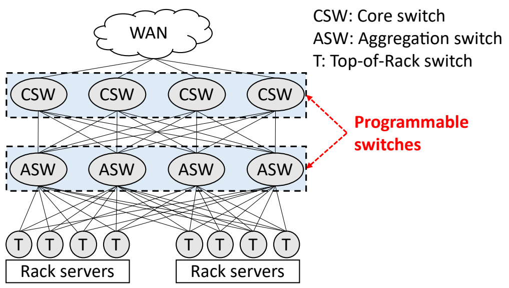


*Figure 1: Example deployment of programmable switches in a data center.*

with various application benchmarks in our testbed consisting of two Tofino-based programmable switches, four regular switches, and 10 commodity servers. Our evaluation results show that under failure-free operation, RedPlane does not incur per-packet latency overhead for read-intensive applications (e.g., NAT) and even for write-intensive applications (e.g., per-flow counter), the latency overhead is $\approx 8\ \mu s$. When a switch fails, RedPlane can recover end-to-end TCP throughput within a second by accessing the correct state.

**Contributions and Roadmap.** In summary, we make the following contributions:

- We formally define a new consistency model called lossy linearizability for in-switch applications (§4).
- We design a state replication protocol that provides lossy linearizability, prove its correctness, and verify it with TLA+ model checking [28] (§5).
- We demonstrate its feasibility and practicality by implementing RedPlane components on switch ASICs, integrating in-switch applications with minimal modifications, and running experiments on a real testbed (§6, §7).

## 2 Background and Motivation

### 2.1 In-Network Acceleration in Data Centers

In-network processing has flourished in recent years, as a natural convergence of the demand for new network functionality from data center operators to support sophisticated virtual networks and the commercial availability of programmable switch platforms. Programmable switches are used for classic middlebox functionality like NATs and firewalls and also advanced load balancers [24,37], monitoring systems [4,20], and other network functions. Programmable switches also serve as in-network accelerators for storage systems [23, 31, 35], database systems [30, 48], and other applications [44].

Many of these applications are *stateful*: they maintain application-specific state on the switch, which they use to determine how to process packets. In this paper, we focus primarily on *hard state* applications, where a loss of state disrupts network or application functionality. Consider an in-switch NAT, where the state consists of an address translation table, and a pool of available ports. Losing this state would

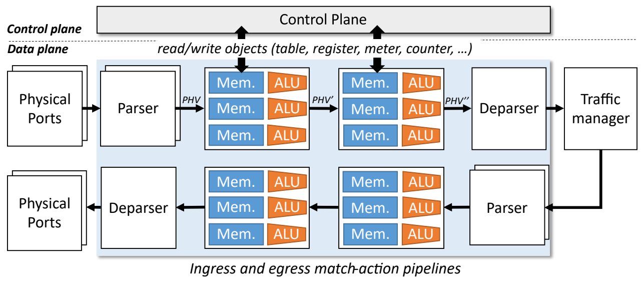

Control Plane
Control plane
Data plane
read/write objects (table, register, meter, counter, ...)
Physical Ports
Parser
PHV
Mem.
ALU
PHV'
PHV''
Deparser
Traffic manager
Ingress and egress match-action pipelines

*Figure 2: Programmable pipelined switch architecture.*

make it impossible to forward packets for existing connections. Table 1 gives more examples of stateful applications and their access patterns.

Other applications maintain only soft state in the switch and use in-switch state as a cache [23], or deal with failures via application-level recovery protocols [30, 31] or by re-executing application-level operations [44, 48]. For such applications, loss of switch state leads to performance degradation (e.g., increased latency) but not fundamental correctness issues. While these applications are outside our scope, they can still benefit from our work; e.g., achieving improved performance during failure or simplifying their core designs by relying on the intrinsic recovery provided by RedPlane.

**Network model.** We consider a deployment model where programmable switches are installed into the network fabric to support in-switch applications. In a typical data center (Figure 1), this would correspond to the core or aggregation layer switches. In-switch processing deployed at this layer can service requests for an entire cluster (vs. being deployed at the top-of-rack switch layer). We consider a network where packets are routed using traditional layer-3 protocols such as Equal-Cost Multi-Path routing (ECMP) and the Border Gateway Protocol (BGP), which is the common approach in large-scale production data center networks [10, 19, 29]. This means that packets may take different paths between layers – so an in-switch application must run as a set of multiple programmable switches deployed at the same layer, e.g., all switches in the aggregation layer. It also provides resilience in the event of failures; packets can be routed along another path to bypass a failed switch.

**Primer on programmable switches.** Programmable switch architectures used today, e.g., the Barefoot Tofino [6], use memory to provide a variety of stateful object abstractions. These include packet headers, metadata, tables, registers, meters, and counters. These objects can be classified into two categories: transient and persistent. Transient objects are those only accessible during a single packet's processing, including packet headers and metadata. Persistent objects are all other types of objects, including tables, registers, meters, and counters. Remaining available across the processing of multiple packets, they store application state, such as the address translation table in the NAT example above. Our goal

| Access patterns | Example applications | Keys | Values | Impact of switch failure |
|---|---|---|---|---|
| Read-intensive | Network Address Translator (NAT) | IP 5-tuple | (IP addr, port) pair | End-to-end connection broken |
| | Load balancer | IP 5-tuple | IP address | End-to-end connection broken |
| | Stateful firewall | IP 5-tuple | Connection state | End-to-end connection broken |
| Write-intensive | Account usage monitor in cellular networks | User ID | Usage in bytes | Inaccurate accounting |
| | Sketch-based heavy-flow detection | Indexes | Sketches | Misdetection |
| Mixed read/write | In-network sequencer | Session ID | Sequence number | Incorrect sequencing |
| | Generic key-value store | Key | Values | Losing key-value pairs |

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

in this paper is to allow these persistent objects to survive switch failures, permitting applications to continue executing without loss of state. Figure 2 illustrates where the stateful objects are allocated in a switch pipeline and how the control (CPU) and data plane (ASIC) access them. In the ingress and egress match-action pipeline (the data plane), global objects are allocated in each stage (denoted as Mem.) and accessed by packets via ALUs. These objects are also accessible by the control plane via the control plane APIs.[1]

## 2.2 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 hardware failures are prevalent. For example, in Microsoft's data center, 29% of customer-impacting incidents are related to hardware failures including ASIC failure, fiber cuts, or power failures [33], and in Facebook's data center, 26% of incidents are related to switch failures [36].

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 [34], causing traffic to be rerouted via a redundant path by routing protocols such as BGP and ECMP [10, 19, 29]. The effect of this is that traffic that previously traversed one switch might start being processed through a different one, where the appropriate application state is unavailable. In the absence of this state, application processing can fail – for example, lacking the proper translation table entries, the NAT cannot forward packets for open connections. This eventually forces end hosts to establish new connections and can seriously degrade end-to-end performance. Table 1 provides more examples of how switch failures impact correctness or performance of applications.

## 2.3 Existing Approaches and Limitations

An ideal fault-tolerance solution for in-switch applications should have the following four properties:

- **Correctness**: Informally, when a switch fails, and packets are routed to alternative switches, an application must be able to access correct state that has been updated or read by the latest packet processed that is output by the failed switch (or alternate path).[2]
- **Performance**: Under failure-free operation, overhead for per-packet latency should be low (say, a few tens of $\mu s$).
- **Low resource overhead**: It should not consume switches' limited computation and storage resources excessively.
- **Location independence**: It must allow a packet to update and/or read state regardless of the location of a switch where the packet is routed.

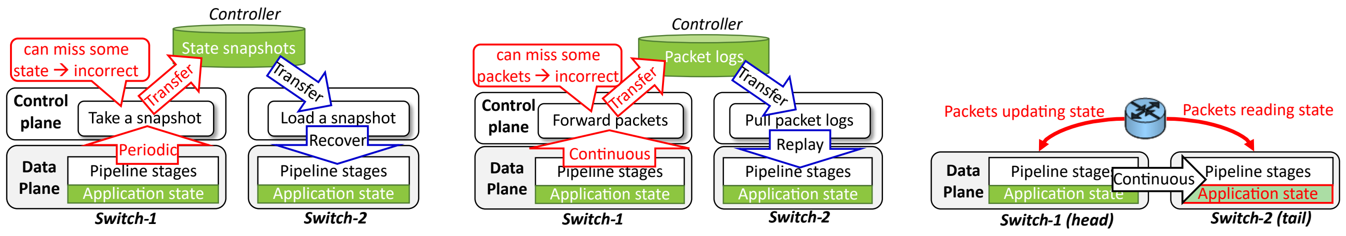
Given these requirements, we can now visit classical fault tolerance mechanisms [17, 39, 49] and mechanisms tailored for network middleboxes [40, 46]. At a high level, these approaches can be categorized into three classes: (1) checkpoint-recovery, (2) rollback-recovery, and (3) state replication. All prior work targets server-based implementations, and it is not obvious if we can translate them to the switch context. In what follows, we explore this question, and show that seemingly natural adaptations of these approaches to the switch environment fail to meet our requirements.

**Checkpoint-recovery.** Checkpointing approaches periodically snapshot application state (e.g., an address translation table in NAT) and commit it to stable storage (e.g., [40]). When a failure occurs, the latest snapshot is populated on a backup node (an alternative switch in our context). Figure 3a illustrates a candidate implementation on switches using the controller to store snapshots. To achieve a consistent snapshot, the switch must pause data plane execution and buffer packets during the snapshot period. This is challenging because of the mismatch between the data traffic rate (Tbit/s) and the data-to-control plane bandwidth (Gbit/s), and packets may be lost during snapshot. More fundamentally, checkpointing has a tradeoff between consistency and performance: frequent snapshots have a high performance overhead, but infrequent snapshots mean that the saved state snapshot may be out of data when needed for recovery.

**Rollback-recovery.** This approach, previously used for middleboxes [46], 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 (Figure 3b). In principle, this approach can guarantee correctness if every packet is synchronously logged and replayed after a failure. However, as with checkpointing, the

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

[2] We will define correctness more precisely in §4



*(a) Checkpoint-recovery: The switch control plane periodically snapshots data plane state and commits it 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 the data plane memory using chain replication. Packets must be routed to the correct chain node depending on their operation.*

*Figure 3: Existing approaches.*

| Property | Checkpoint | Rollback | Replication |
|---|---|---|---|
| Correctness | ✗ | ✗ | ✓ |
| Performance | ✗ | ✗ | ✓ |
| Resource overhead | ✓ | ✓ | ✗ |
| Location independence | ✓ | ✓ | ✗ |

*Table 2: Existing fault-tolerance approaches that can be potentially applied for in-switch applications.*

mismatch between the switch data and control plane's processing speed will result in many packets being missed and thus be incorrect. In terms of performance, logging and replaying packets increases per-packet latency to a few tens of *ms* for both normal and failover operation.

**State replication among switch data planes.** Consider a state machine replication approach using chain replication [49], but applied to switch data planes. 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 approach achieves correctness because state updates are never lost. Although packets need to be forwarded through multiple switches, this is done entirely on the data plane, so it can function at high speed. However, 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 does not provide location independence 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.** Table 2 summarizes our discussion and shows that none of the existing approaches satisfy all four properties. Our analysis also indicates two fundamental insights that inform our design: (1) Involving the switch control plane involvement significantly increases latency and limits the throughput on failure-free operation. (2) While switch data-plane-based approaches can provide correctness with good performance, they incur significant switch resource overhead and may not be location-independent.

## 3 Overview

Our goal is to design a fault tolerance solution that satisfies all four properties – correctness, performance, low resource overhead, and location independence. To this end, we present RedPlane, which provides an abstraction of location-independent, fault-tolerant state storage for stateful in-switch applications. RedPlane continuously replicates state updates so that they 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 processor is not required for state replication, avoiding the issues with the checkpointing and rollback-recovery approaches of §2.3. 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 §2.3.

As shown in Figure 4, RedPlane provides a set of APIs implemented in P4 [7], a programming language to implement stateful programs on programmable switches, and exposes modules that can be used in P4 programs. This allows 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 data plane, which continuously replicates state updates to the state store through the data plane.

### 3.1 Challenges

While replicating state updates through the data plane to an external state store is conceptually simple, realizing this idea in practice presents some difficult challenges:

1. **Handling high traffic volume.** Switches operate at immense traffic volumes: a programmable switch may process up to 4.8 billion packets per second [11], whereas a server-based system would be hard-pressed to handle more than a few million. If each packet requires interacting with

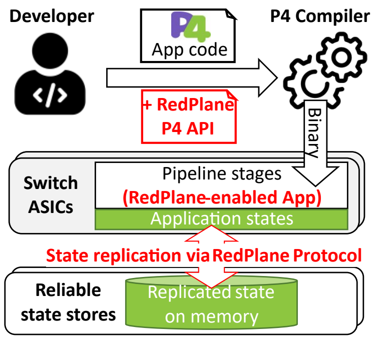


*Figure 4: Overview of RedPlane, highlighting new extensions that extend traditional workflows for in-switch applications*

a state store built from commodity servers, its capacity will rapidly be exceeded. We must find a way to minimize communication with the state store.

2. **Tolerating unreliable communication.** Communication between the switch and state store takes place via an unreliable communications channel. This is hardly a new problem for distributed systems, which are accustomed to asynchronous, lossy networks. However, the traditional solutions cannot readily be applied in the switch environment: a switch data plane does not have the ability to establish TCP connections, nor can it buffer significant amounts of traffic. We must find a new way to ensure that state updates are reliably and consistently recorded.
3. **Supporting location independence.** An in-switch application may be distributed over multiple switches – recall the example of a switch cluster accelerating NAT translation. RedPlane must allow instances to operate concurrently with separate state. At the same time, one switch's state may be needed on another; e.g., when a link failure or change in load balancing causes traffic to be routed to a different switch. RedPlane must be able to transparently migrate that state, e.g., to make the translation table entry available when packets for a particular connection are processed by a different NAT instance.

## 3.2 Key Ideas

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

1. **Practical consistency models for switch state.** Traditional definitions of linearizability [21] provide a strict correctness guarantee for replicated systems, but also impose onerous coordination requirements. We observe that two properties of the network environment and of in-switch applications allow us to safely relax these requirements. First, networks are already lossy, so our implementation mechanisms themselves can impose a small probability of packet loss without impacting correctness. We formalize this through a definition of lossy linearizability (§4.2). Second, in-switch applications generally organize their state on a per-flow basis. We formalize this through a per-flow lossy linearizability model (§4.3). This allows us to partition state storage around flows, and enables our state migration mechanism to support location independence.
2. **Lightweight sequencing and retransmission.** To cope with the unreliable communication channel between the switch data plane and state stores without 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 (§5.2).
3. **Lease-based state ownership.** To reduce the frequency with which the switch must coordinate with the state store, RedPlane uses a lease-based mechanism where the state store grants temporary ownership of certain state to a switch (§5.3). This approach allows us to avoid coordination with the state store for packets which read but do not modify switch state. At the same time, our mechanism ensures that all state updates are durably recorded before any of their effects are externalized, providing the appropriate correctness guarantees. The lease-based ownership mechanism also serves as the means by which state is migrated between switches to support location independence.

## 4 Consistency for In-Switch Applications

Informally, RedPlane provides the abstraction of "one big fault tolerant switch" – the behavior of the system is indistinguishable from the same application running on a single switch that never fails. This section makes precise that definition of correctness. RedPlane's semantics are based on linearizability, a standard correctness condition for concurrent systems. However, we observe that the standard definition is stricter than what is expected by a network, as lossy networks provide no delivery guarantees. RedPlane uses a relaxed definition of *lossy linearizability* which captures these semantics, and enables important protocol optimizations.

### 4.1 Preliminaries

In our replicated state machine approach [27, 45], we model a stateful switch program, where the output and next state are determined entirely by the input and current state:

**Definition 1 (Stateful program).** A stateful program $P$ is defined by a transition function $(I,S) \to (O^*,S')$ 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 [13], their compilers assign state to pipeline stages in a way that makes packet processing appear atomic [12]. We assume that our target applications are deterministic, as described in §2.

The gold standard for replicated state machine semantics is single-system linearizability [21]. That is, that the observed execution matches a sequential execution of the program that respects the order of non-overlapping operations. Rephrased in terms of packet processing, this means:

**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.

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

## 4.2 Lossy Linearizability

The previous definition requires that every input packet be processed and its corresponding output delivered.[3] Implementing this level of reliability over unreliable channels is challenging. However, it is also a stricter requirement than networks provide: networks can be lossy and do not guarantee delivery. Thus, we explicitly permit messages to be lost, as applications are designed to be resilient to such losses. Formally:

**Definition 4** (**Lossy linearizability**). A history $H$ is a lossy-linearizable execution of program $P$ if we can construct a linearizable history $H'$ by adding output events $O$ and removing input events $I$ from $H$ (otherwise preserving $H$'s order).

This definition captures two scenarios that can already happen in networked systems. First, it is possible for an input packet to neither update the program state nor produce an output packet (the events $I$ in $H$ but not $H'$). This corresponds to a packet that was dropped before reaching the switch. Second, it is possible for an input packet to update the switch state but produce no visible output packet. This corresponds to a packet that was received and processed, but the output packet dropped immediately after leaving the switch (these are the output events $O$ not observed in $H$).

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 there is message loss during communication between a switch and the state store. In these scenarios, an input packet or its output may be lost. This definition expresses that this is similar to packet loss that already occurs in the network.[4]

[3] The concept of linearizability comes from the concurrency literature [21], so it has no notion of failure or operations that do not complete.

[4] Of course, dropping too many packets is undesirable for performance reasons; such loss events are rare.

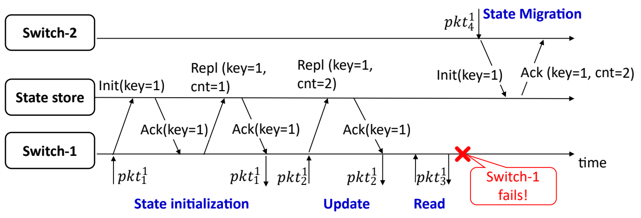


*Figure 5: Basic workflow of RedPlane state replication protocol. Repl means a state replication request. $pkt_n^f$ indicates $n^{th}$ packet of a flow $f$.*

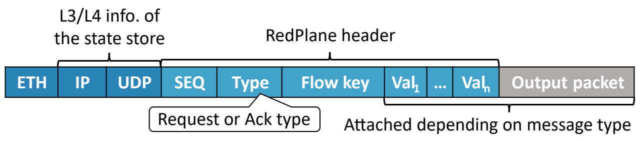


*Figure 6: RedPlane state replication protocol packet format.*

## 4.3 Per-flow Lossy 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. For example, each translation table entry in a NAT is tied to a specific flow. For many applications, per-flow state is the only state that needs to be consistent or fault tolerant. In some cases, this is because the application only has per-flow state. In other cases, global state exists but can tolerate weaker consistency – for example, traffic statistic counters can tolerate some error. RedPlane generally provides consistency for per-flow state, and can optionally provide consistency for global state. More precisely:

**Definition 5** (**Per-flow (lossy) linearizability**). A history $H$ is per-flow (lossy) linearizable if, for each flow $f$, the subhistory $H_f$ for the packets in flow $f$ is (lossy) linearizable.

For programs that use only per-flow state, per-flow (lossy) linearizability is indistinguishable from global (lossy) linearizability, because linearizability is a *local* (i.e., composable) property [21]. The benefit of correctness on a per-flow level is that it means synchronization between states associated with different flows are not required. As we will see 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: 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 state for the other flows.

## 5 RedPlane Design

Now, we describe the RedPlane protocol that achieves lossy linearizability. We begin with an overview of the protocol and explain how we address practical challenges.

## 5.1 Basic Design

As shown in Figure 4, RedPlane largely 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. We use a per-flow counter as an example application.

To simplify exposition, in this section we start with some simplifying assumptions: that there is no packet loss or reordering between switches and the state store, that switches do not fail with in-transit messages, and that packets for a flow are routed to only one switch at a time. We eliminate 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 in our implementation by an IP 5-tuple, though any other key is possible. Each state store shard is replicated using conventional mechanisms. We do not innovate here, and many existing key-value stores will suffice. Our prototype is a simple in-memory storage server that uses chain replication [49] with a group size of 3.

### 5.1.2 Basic replication protocol

**State initialization and migration.** In RedPlane, an application replicates state updates to the state store by exchanging protocol messages formatted as shown in Figure 6. The IP and UDP headers contain the address information of a state store (if it is a request) or a switch (if it is a response) so that each message can be routed to an intended destination. The RedPlane header consists of a sequence number, a message type, and a flow key. Depending on a message type, it can also include flow state and an output packet. We will discuss these fields shortly.

When the application receives a packet that belongs to a flow it has never seen before, it sends a *state initialization request* to the state store. It identifies the proper 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 state has never existed on any switches that run the application or (2) the flow state has existed on a failed switch, and a packet for the flow is 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. In case (2), since the state store already has the flow state, it sends a response containing the latest state.

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 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 this overhead is minimal.

**Reading or updating state.** Once the state has been initialized, the application can read the counter value directly. When it updates the counter, 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.

**Packet buffering.** When the application updates the counter (or any other 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 consistency. 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 as a form of delay line memory – trading off network bandwidth, which is plentiful on a switch, for data plane memory, which is scarce.

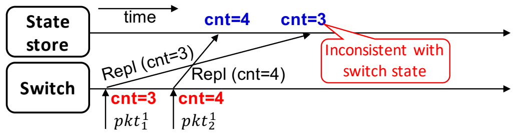
Note that it is possible to receive packets that read state when there are in-flight replication requests for the state. In this case, the packets are buffered in the same way through the network (but without RedPlane headers) until a switch receives a response for the latest replication request.

Overall, Figure 5 illustrates how RedPlane-enabled application initializes, updates, reads, and migrates state (during failover) via our replication protocol. While our basic design provides correctness under the simplified assumption, we find that in more realistic environments, it may not be able to guarantee correct behavior. In the following sections, we describe potential problems, challenges, and how we extend the basic design to address them.

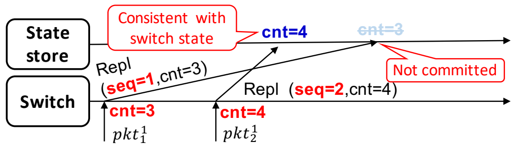
## 5.2 Sequencing and Retransmission

To guarantee correctness, replication requests must be *successfully* delivered and replicated *in order* at the state store. However, successful in-order delivery is not guaranteed in a best-effort network between switches and the state store.

Figure 7a 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 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.

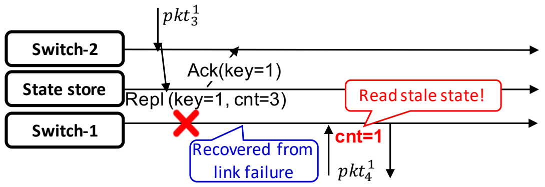


*(a) Out-of-order delivery of requests causes inconsistency.*

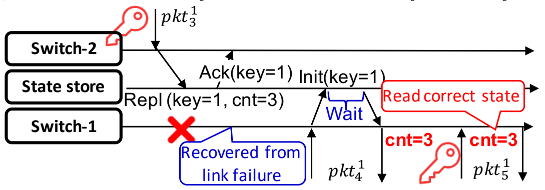


*(b) Request sequencing serializes replication requests.*

*Figure 7: 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.*



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



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

*Figure 8: Consistent state access for multiple switches.*

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, 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* [31], 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, as shown in Figure 7b.

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 packet mirroring capability of switch ASICs. When it sends a replication request, it makes a copy of the request using the egress-to-egress mirroring. Then, the mirrored request is buffered at the egress queue and it is retained until the switch receives a response with the same or a higher sequence number.

As discussed previously, buffering is challenging on a switch due to memory limitations. RedPlane buffers only state updates – not the piggybacked output packet. This reduces the amount of data that needs to be buffered. 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 lossy linearizability 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.

## 5.3 Lease-based State Ownership

What if multiple switches attempt to process packets for a particular flow at the same time? The protocol in §5.2 will not be correct in this case, when there are concurrent accesses to the same state. Figure 8a illustrates why. After Switch-1 has a link failure (but does not lose its state), packets are routed to an alternate, Switch-2. If Switch-1 recovers, a packet may read its old state, a violation of linearizability.

One approach to solving this problem would be to serialize all operations, including reads, through the state store – i.e., to not store state on switches at all. While correct, this approach suffers from extremely poor performance; it would negate the benefit of using the switch as an accelerator entirely.

**Our approach.** To allow switches to process state reads without contacting the state store every time, RedPlane ensures that only one switch can process packets for a given flow at a time. It does this using leases, a classic mechanism for managing cached data in file systems [18] and replicated systems [32, 38]. Figure 8b illustrates this. If a packet wants to access 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 also send explicit lease renewal requests. Our prototype does so every 0.5 seconds.

## 5.4 Protocol Correctness

RedPlane's replication protocol provides per-flow lossy linearizability. 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 lossy 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 linearizbility 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 our protocol to verify it (Appendix B).

**Weak consistency mode.** Some applications, such as ones that use approximate or statistical data structures (i.e. sketches [14]), do not require strict consistency. To better support these applications, RedPlane supports a weak consistency mode. In this mode, when a packet updates state, RedPlane releases the output packet without waiting until the state is replicated. This mode trades correctness for performance. In our implementation, we use this mode to implement a heavy-hitter detector and an asynchronous-counter.

## 6 Implementation

**Data plane.** We implement RedPlane's data plane components in P4-16 [7] (≈1068 lines of code) and expose them as a library of P4 control blocks [7, §13], which form the RedPlane APIs. As shown below, developers can make the state of applications fault-tolerant by using the API. We compile RedPlane-enabled applications to the Barefoot Tofino ASIC [6] with P4 Studio [8]. 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 packet mirroring and header truncation capability of the switch ASIC which allows us to buffer only a request header packet of each packet by truncating an original payload.

**Control plane.** We implement the switch control plane in Python. The main capability needed here is to initialize and migrate (if available) state for the data plane by processing corresponding responses forwarded by the data plane component. Since this is not performance critical, we opt for a modular implementation using Python.

**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++ running on Linux servers. We wanted a kernel-bypass UDP transport library; while systems like DPDK or Netmap would suffice, we were already familiar with Mellanox's Verbs API [1], and so we built our own (called `raw_transport`) based on `IBV_QPT_RAW_PACKET` queue pairs. This is the same API used to implement Mellanox's DPDK driver. To ensure reliability in the presence of server failures, we implement chain replication [49] on a group of state store servers located at different racks so that each state updates are replicated to three servers.

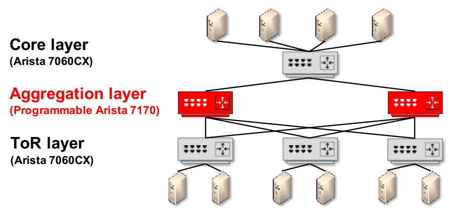


*Figure 9: Three-layer network testbed for experiments.*

**Applications.** To demonstrate the applicability of RedPlane, we implement various in-switch applications in P4 described below. Simplified P4 code for NAT is in Appendix A.

*NAT:* The NAT implementation uses RedPlane to implement a fault-tolerant per-flow 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.

*Firewall:* The stateful firewall adds fault-tolerance to a per-flow TCP connection state table using RedPlane. Its state is updated when a TCP connection is established from an internal network.

*Load balancer:* The load balancer maintains a per-flow 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.

*Per-flow counter:* This application counts a number of packets for each flow and maintains it as its state. RedPlane supports both synchronous update (i.e., strict consistency) and asynchronous update.

*Heavy-hitter (HH) detector:* We implement a heavy-hitter detector using count-min sketches [14]: there are 3 sketches, each consisting of 64 slots. Since sketches are an approximate data structure, they do not require strict consistency. Thus, we use asynchronous update mode. Unlike the above applications, this one issues three update requests for each sketch, and the state store *merges* updates for each slot in the sketches.

*Simple in-switch key-value store:* We also implement a simple in-switch key-value store to evaluate the performance of RedPlane-enabled applications vs. read/update ratios.

## 7 Evaluation

We evaluate RedPlane on a testbed consisting of six commodity switches (including two programmable ones) and servers (Figure 9) using both real data center network packet traces and synthetic packet traces. Our key findings are:

- In failure-free operation, RedPlane induces no per-packet latency overhead for applications that are read-intensive or

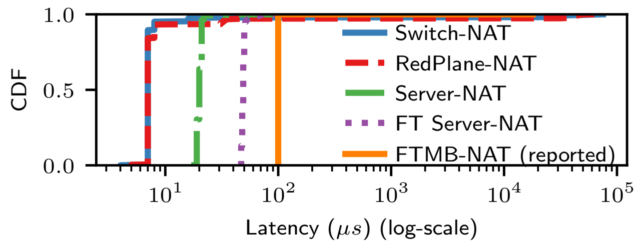


*Figure 10: End-to-end RTT when RedPlane-NAT processes packets vs. other approaches.*

replicate state asynchronously. For write-intensive applications, RedPlane incurs 8 $\mu s$ per-packet overhead (§7.1).

- For failure-free operation, the throughput of read-intensive applications are not degraded. For write-intensive applications, the throughput is bottlenecked by state store performance, but adding servers scales the throughput (§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 as shown (Figure 9). The aggregation layer has two 64-port Arista 7170 Tofino-based programmable switches [11] (red in the figure), 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 of aggregation layer switches fails. There are four servers connected to the core switch, which we use to emulate hosts outside the data center. Each ToR switch has two servers connected; we use one as a state store and the other as an end host. 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).

## 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 use publicly available packet traces from a real data center and enterprise network [2, 3]. The packet sizes vary (64–1500 bytes) in the real traces. We replay traces using our custom packet generator built based on our `raw_transport` library described in §6.

**Overhead of RedPlane** As an exemplar application, we evaluate the per-packet latency for a NAT application in RedPlane[5] and compare it with baseline implementations: (1) NAT written in P4 without fault-tolerance (Switch-NAT), (2) NAT implemented on a CPU server without fault-tolerance

[5]We choose NAT to compare it with a NAT performance reported in a prior work [46].

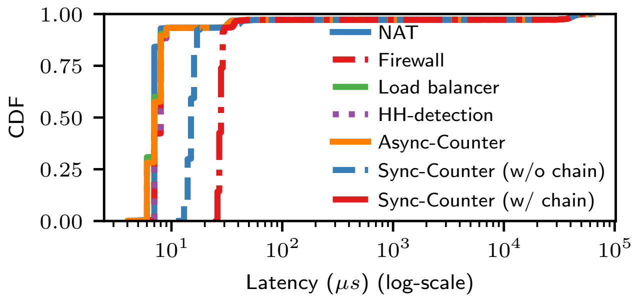


*Figure 11: End-to-end RTT for RedPlane-enabled applications. All applications have chain replication enabled for the state store. For Sync-Counter, we also show its overhead without the chain replication.*

(Server-NAT), (3) NAT implemented on a server with fault-tolerance (FT Server-NAT), and (4) FTMB-NAT which uses rollback-recovery for server-based middleboxes [46].[6]

Figure 10 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 $\mu s$ and 8 $\mu s$, 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 (33 *ms* and 42 *ms*, 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. We see that server-based versions (FT Server-NAT and FTMB-NAT) have 7 – 14× higher median latency vs. 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 Figure 11, RedPlane-enabled NAT, firewall, load balancer, heavy-hitter (HH) detection, and asynchronous counter all have the same 8 $\mu s$ median latency, which is identical to that without fault-tolerance. This is due to their state access patterns: the NAT, firewall, and load balancer all update state only when a new flow is created. For HH-detection and Async-Counter, although they update the state for every packet, since state replication is performed asynchronously, it does not affect the latency. On the other hand, since Sync-Counter updates state and replicate updates *synchronously* for every packet, it adds an additional latency of 20 $\mu s$ to every packet. As mentioned in §6, our implementation uses chain replication to make the state store reliable among servers, the chain replication latency also contributes

[6]We use the latency reported in the original FTMB paper [46] since we were not able to get its full implementation.

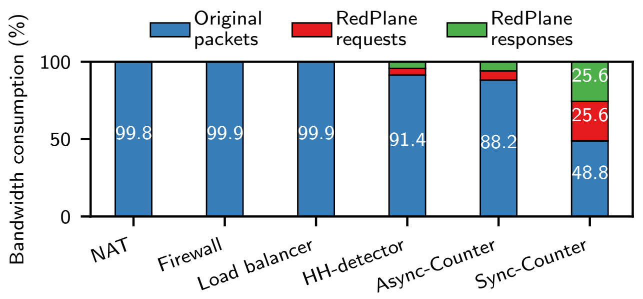


*Figure 12: Bandwidth overhead of RedPlane.*

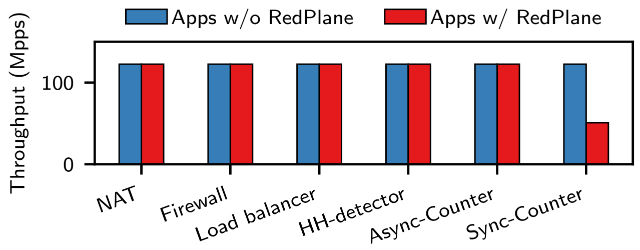


*Figure 13: Impact of RedPlane on data plane throughput of RedPlane-enabled applications.*

to the per-packet latency. To break down this overhead, we did the same measurement without chain replication, and we see that the latency overhead becomes 8 $\mu s$, meaning that chain replication adds 12 $\mu s$ latency overhead.

## 7.2 Bandwidth Overheads

**Additional bandwidth needed:** We instrument each application to count the number of bytes it sends and receives, including both original packets and protocol message packets. Figure 12 shows the ratio of bytes sent for each type to the total number of bytes. For read-intensive 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 the applications which asynchronously replicate state updates for every packet, HH-detector and Async-Counter, RedPlane consume 8.6% and 11.8% additional bandwidth, respectively. Sync-Counter shows higher bandwidth overhead (51.2%) because RedPlane requests and responses contain both headers and original payload. This result implies that in an extreme case where an application replicates state update synchronously for every packet, achieving fault-tolerance is expensive.

**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 bytes packets from three servers, one from each rack, to one of servers attached to the core switch as fast as possible using our custom packet generator built based on `raw_transport`. 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. Figure 13 shows the throughput of each application with and without RedPlane. Obviously, applications achieve the maximum throughput without RedPlane. With RedPlane, read-intensive (NAT, firewall, and load balancer) applications and applications that replicate state updates asynchronously (HH-detector and Async-Counter) can achieve the same throughput as their non fault-tolerant counterparts. However, 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-intensive applications as they are also affected by the performance of the state store.

**Varying update ratios:** While most of existing in-switch applications are read-intensive 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 (if the packet updates the state). 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. Figure 14 shows that as the update ratio increases, the throughput degradation depends on the number of state store servers and by adding more servers, we can achieve higher throughput.

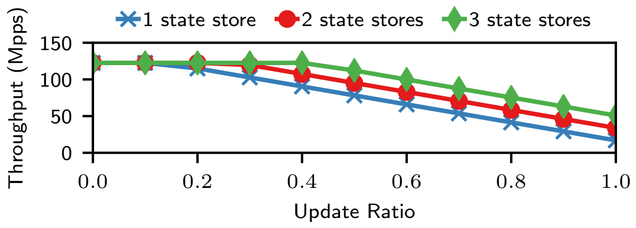


*Figure 14: Impact of update ratio on data plane throughput of RedPlane-enabled applications.*

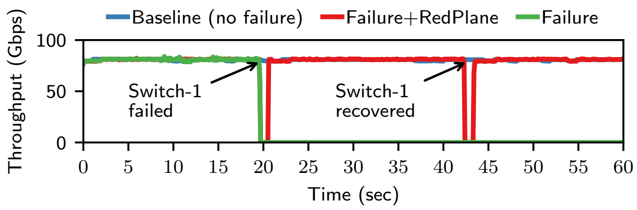


*Figure 15: End-to-end throughput changes during failover and recovery with and without RedPlane.*

## 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` [9] on two servers, one attached to a core switch (server) and the other attached to a ToR switch

| 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 3: Switch ASIC resources used by RedPlane.*

(client), and measure TCP throughput changes between them over 60 seconds. The client creates 4 parallel TCP connections to achieve the maximum throughput (≈81 Gbps). We use NAT as an application 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).

Figure 15 shows the results. When there is no failure, the throughput is ≈81 Gbps (Baseline). When a switch without RedPlane (`Switch-1`) fails at t=20 seconds, throughput goes to zero (Failure). Here, when the failure happens and the core switch detects the failure, it routes packets to the backup switch (`Switch-2`). Since the NAT on `Switch-2` does not have state for the TCP flows, it drops the packets, eventually breaking the connection. For the same reason, even when `Switch-1` recovers at around 42 seconds, the throughput does not recover. In contrast, RedPlane-enabled NAT successfully maintains high throughput when the switch fails and recovers after a short disruption (0.9 and 1.0 seconds). Analyzing the disruption, we find that there are three main factors: (1) the period between `Switch-1`'s failure and when packets are rerouted to `Switch-2`, (2) the period between the state store receiving a lease request and the lease previously assigned to `Switch-1` expiring (set to 1 second), and (3) latency for initializing the lease and populating migrated state on `Switch-2`. We expect that further optimizations on the switch control plane and the state store can reduce this disruption.

## 7.4 RedPlane Switch ASIC Resource Usage

Table 3 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. SRAM is the most used resource (13.2%); all other usage, including Match Crossbar, Meter ALU, Gateway, VLIW instruction, and hash bits, is less than 10%.[7] 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).

[7]Match Crossbars are used for implementing the 'matching' part of match-action tables. Meter ALUs perform stateful operations on registers. Gateways perform 'if-else' conditions in the control flow.

## 8 Related Work

**In-switch applications.** Recent efforts have shown that offloading to programmable switches enhances performance. For example, offloading the sequencer [31], key-value cache [23, 35], and coordination service [22] improves the performance of distributed systems. However, these applications can lose their state due to switch failures. RedPlane can help them fault-tolerant or simplify their designs.

**Fault-tolerance and state management for NFs.** Fault-tolerance for network functions (NF) or middleboxes has been addressed by prior systems like Pico [40] and FTMB [46]. When an NF instance fails, the state of the failed NF is recovered through checkpoint or rollback recovery on a new NF instance. However it is impractical to apply these approaches to the context of switch data plane due to correctness and performance issues as we discussed in §2.3. Previous work on state management for stateful NFs uses the local or remote storage to manage NF state [16, 41, 50]. However, these APIs target planned state migration rather than unplanned failures, these approaches are not applicable for dealing with failures. Similar work has also been proposed for router migration [25]; again this focuses on planned migration and cannot recover state from failed switches.

**Server memory as external store for switches.** Our approach leverages servers' memory in the data plane, akin to recent work [26]. While the idea of using servers' memory as an external store is similar, this work was designed for a different goal. Their goal is to enable a switch to access memory of servers directly attached to it on the same rack as a "virtual memory". They do not tackle fault tolerance or correctness in the scenarios when multiple switches can access the store.

**Switch-based reliability protocols.** Other recent work runs coordination protocols between switches to build reliable storage [15, 22]. Our goal is conceptually different –to replicate state for in-switch applications rather than provide a networked storage service. That said, some of our ideas for network sequencing are similar [31].

## 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 that provides the abstraction of location independent, fault tolerant state store for in-switch applications. We formally define new consistency model for replicated 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.

## References

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

[2] 2009-M57-Patents packet trace. http://downloads.digitalcorpora.org/corpora/scenarios/2009-m57-patents/net/, 2009.

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

[4] Advanced network telemetry. https://www.barefootnetworks.com/use-cases/ad-telemetry/, 2018.

[5] 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/., 2018.

[6] Barefoot Tofino. https://www.barefootnetworks.com/products/brief-tofino/, 2018.

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

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

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

[10] Alexey Andreyev. 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/, 2014.

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

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

[13] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM*, 2013.

[14] Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *Proc. VLDB Endow.*, 1(2):1530–1541, August 2008.

[15] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, pages 1–13, 2020.

[16] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *ACM SIGCOMM*, 2014.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SOSP*, 2003.

[18] Cary Gray and David Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review*, 23(5):202–210, 1989.

[19] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM*, 2009.

[20] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*, 2018.

[21] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[22] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *USENIX NSDI*, 2018.

[23] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *ACM SOSP*, 2017.

[24] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load

balancing using programmable data planes. In *ACM SOSR*, 2016.

[25] Eric Keller, Jennifer Rexford, and Jacobus E van der Merwe. Seamless bgp migration with router grafting. In *NSDI*, 2010.

[26] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *ACM SIGCOMM*, 2020.

[27] Leslie Lamport. Time, clocks, and ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[28] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

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

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

[31] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say no to paxos overhead: Replacing consensus with network ordering. In *USENIX OSDI*, 2016.

[32] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA, July 2012.

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

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

[35] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *USENIX FAST*, 2019.

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

[37] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *ACM SIGCOMM*, 2017.

[38] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, 2014.

[39] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *ACM SOSP*, 2011.

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

[41] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *USENIX NSDI*, 2013.

[42] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM*, 2015.

[43] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[44] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

[45] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

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

[47] 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. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *ACM SIGCOMM*, 2015.

[48] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating database queries with switch pruning. In *ACM SIGMOD*, 2020.

[49] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *USENIX OSDI*, 2004.

[50] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *USENIX NSDI*, 2018.