RedPlane: Enabling Fault-Tolerant Stateful In-Switch Applications

Daehyeok Kim§‡
Jacob Nelson‡, Dan Ports‡, Vyas Sekar§, Srinivasan Seshan§

§Carnegie Mellon University  ‡Microsoft
Programmable networks are stateful

Classical switches

<table>
<thead>
<tr>
<th>Match</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP addr/prefix</td>
<td>Forward (port)</td>
</tr>
</tbody>
</table>

“Stateless” packet processing

Programmable “data plane” switches

Stateful in-switch applications:
network functions, monitoring, accelerating distributed systems

<table>
<thead>
<tr>
<th>Match</th>
<th>Stateful action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IP addr, port)</td>
<td>NAT (IP addr., port)</td>
</tr>
</tbody>
</table>

Programmable switching ASICs

⇒ “Stateful” packet processing
Problem: Switch failure

Switch failures are prevalent \[1, 2\]

Other stateful apps suffer from the same problem!

Flow state does not exist! ➔ Connection broken 😞

<table>
<thead>
<tr>
<th>Match (Flow ID)</th>
<th>NAT action (IP, Port)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10.0.0.1, 4321)</td>
<td>(192.168.10.1, 1234)</td>
</tr>
</tbody>
</table>

Strawman solutions

S1: Checkpoint-recovery

- Requires a custom routing policy
- Updates can be lost

S2: Chain replication among switches

- Mismatch between the control and data plane performance ➔ miss some state / packet drops
- Consumes additional resources

- Requires a custom routing policy

Our work: RedPlane

- App code + RedPlane P4 API
- APIs that allow easy integration with apps
- Correct state replication entirely in the data plane
- Inexpensive replicated state store on commodity servers

One big fault-tolerant switch abstraction!
Outline

RedPlane motivation

RedPlane design

Results
RedPlane design overview

Developer

App code

P4 Compiler

RedPlane-enabled app

RedPlane protocol

RedPlane-enabled app

RedPlane protocol

External state store

P4 API

+RedPlane
Challenge 1: Correct replication in the data plane

Strawman: strict correctness used in server-based replicated systems

Buffer a packet until the state is replicated (exactly-once semantics)

Ensure replication messages are delivered in order and reliably (Linearizability)
Challenge 1: Correct replication in the data plane

Strawman: strict correctness used in server-based replicated systems

- Expensive to buffer entire packets 😞
- Expensive to realize reliable transport in the switch data plane 😞
Linearizable mode: Relaxed correctness

**Insight:** End-to-end network apps already tolerate lossy networks!

**Our approach:** Linearizability-based relaxed correctness

- In-order and reliable message delivery
  - Provides linearizability

- Permitting some input/output packet loss
  - No need to buffer entire packets
Basic RedPlane protocol: Realizing the linearizable mode in the data plane

Example: per-flow packet counter

1. Sends a state initialization request

Piggyback an output packet instead of buffering locally!
Basic RedPlane protocol: Realizing the linearizable mode in the data plane

Example: per-flow packet counter

1. Sends a state initialization request

2. Receives an ACK & initializes the local state
Basic RedPlane protocol: Realizing the linearizable mode in the data plane

Example: per-flow packet counter

1. Sends a state initialization request
2. Receives an ACK & initializes the local state
3. Replicates the updated state
Basic RedPlane protocol:
Realizing the linearizable mode in the data plane

Example: per-flow packet counter

1. Sends a state initialization request
2. Receives an ACK & initializes the local state
3. Replicates the updated state
4. Receives an ACK & releases the output packet
Inconsistency due to unreliable channel

Problem: state in the switch and state store can be inconsistent due to out-of-order requests or request packet loss
Our approach: A simple UDP-based transport with sequencing and lightweight retransmission

Commits the latest requests only based on a sequence number

Buffers only RedPlane header by leveraging packet mirroring & truncating feature in ASIC
Challenge 2: Transparent to routing policies

A switch failure or recovery can cause routing traffic to another switch.

Ensure that a packet accesses the correct state irrespective of the location of the current switch.
Accessing stale state

**Problem:** A packet may access state during failover or recovery.

![Diagram showing a link failure between two switches and an external state store, indicating that local state is still alive.](image)
Accessing stale state

Problem: A packet may access state state during failover or recovery
Accessing stale state

**Problem:** A packet may access stale state during failover or recovery.

Diagram:
- Two RedPlane-enabled apps
- Two switches (Switch-1 and Switch-2)
- External state store
- Reads stale state!
Lease-based state ownership management

Our approach: For a given flow, ensuring only one switch processes packets at a time using leases.
Lease-based state ownership management

Our approach: For a given flow, ensuring only one switch processes packets at a time using leases.
Lease-based state ownership management

Our approach: For a given flow, ensuring only one switch processes packets at a time using leases
Challenge 3: Handling high traffic volume

Switch data plane operates at up to a few billion packets per second

High performance overhead 😞

Unable to keep up with high replication request rate 😞
Challenge 3: Handling high traffic volume

Switch data plane operates at up to a few billion packets per second

- RedPlane-enabled app
- External state store
- Lease-based state management allows local reads for read-centric apps
- “Bounded-inconsistency mode” for write-centric applications
Putting it all together

RedPlane provides a fault-tolerant state store abstraction to applications

Sequencing and lightweight retransmission mechanism (Correctness)

Lease-based state management (Routing agnostic, Performance)

Linearizability-based correctness definition (Correctness)
Bounded-inconsistency for write-centric applications (Correctness, Performance)
Outline

RedPlane motivation

RedPlane design

Results
Implementation

Developer

P4 code

P4 modules

Six switches (2 programmable, 4 regular) and 10 commodity servers

P4 Compiler

RedPlane-enabled app

Various P4 applications for evaluation

+RedPlane P4 API

External state store

Replicated state store on servers in C++

RedPlane protocol

P4 Compiler
How does RedPlane affect application latency?

State initialization overhead (once per flow)

CDF

Latency (μs)

Switch-NAT (w/o FT)
RedPlane-NAT
Server FT-NAT

6X
How much BW overhead does RedPlane add?

- NAT: 99.8%
- Firewall: 99.9%
- Load balancer: 99.9%
- EPC-SGW: 91.4%
- Sync-Counter: 25.6%

BW consumption (%)

- Original packets
- RedPlane requests
- RedPlane responses
How fast the connectivity can be recovered?

![Graph showing throughput over time with labels for baseline (no failure), failure, and failure + RedPlane conditions. The graph highlights instances where Switch-1 failed and Switch-1 recovered.]
Other results

Throughput of RedPlane-enabled applications

Low switch resource overhead of reliable replication protocol

Less than 13% of switch ASIC resource usage

Model checking for RedPlane protocol by using TLA+
Future directions

Better support for write-centric apps

Supporting non-partitionable states

Automatically enabling fault-tolerance with compiler/language support

Next generation switch architectures for fault-tolerance
Conclusions

Switch failures can affect the correctness of stateful in-switch apps

**RedPlane provides a fault-tolerant state store abstraction**
- Linearizability-based practical correctness definition for in-switch apps
- Bounded inconsistency mode for write-centric apps
- Sequencing and lightweight retransmission for reliable replication
- Lease-based state ownership management

Offers fault tolerance with minimal performance and resource overhead
- No per-packet latency overhead for read-centric apps
- End-to-end connectivity is recovered within a second

[github.com/daehyeok-kim/redplane-public](http://github.com/daehyeok-kim/redplane-public)