SwiSh: Distributed Shared State Abstractions for Programmable Switches

Lior Zeno[⋆] Dan R. K. Ports[†] Jacob Nelson[†] Daehyeok Kim[†] Shir Landau Feibish[‡] Idit Keidar[⋆]
Arik Rinberg[⋆] Alon Rashelbach[⋆] Igor De-Paula[⋆] Mark Silberstein[⋆]

[⋆]Technion [†]Microsoft Research [‡]The Open University of Israel

Abstract

We design and evaluate *SwiSh*, a *distributed shared state management* layer for data-plane P4 programs. SwiSh enables running scalable stateful distributed network functions on programmable switches entirely in the data-plane. We explore several schemes to build a shared variable abstraction, which differ in consistency, performance, and in-switch implementation complexity. We introduce the novel *Strong Delayed-Writes (SDW)* protocol which offers consistent snapshots of shared data-plane objects with semantics known as r-relaxed strong linearizability, enabling implementation of distributed concurrent sketches with precise error bounds.

We implement strong, eventual, and SDW consistency protocols in Tofino switches, and compare their performance in microbenchmarks and three realistic network functions, NAT, DDoS detector, and rate limiter. Our results show that the distributed state management in the data plane is practical, and outperforms centralized solutions by up to four orders of magnitude in update throughput and replication latency.

1 Introduction

In recent years, programmable data-plane switches such as Intel's Tofino, Broadcom's Trident, and NVIDIA's Spectrum [9, 33, 60] have emerged as a powerful platform for packet processing, capable of running complex user-defined functionality at Tbps rates. Recent research has shown that these switches can run sophisticated network functions (NFs) that power modern cloud networks, such as NATs, load balancers [40, 57], and DDoS detectors [45]. Such data-plane implementations show great promise for cloud operators, as programmable switches can operate at orders of magnitude higher throughput levels than the server-based implementations used today, enabling a massive efficiency improvement.

A key challenge remains largely unaddressed: realistic data center deployments require NFs to be *distributed over multiple switches*. Multi-switch execution is essential to correctly process traffic that passes through multiple network paths, to tolerate switch failures, and to handle higher throughput. Yet, building distributed NFs for programmable switches is challenging because most of today's NFs are *stateful* and need their state to be consistent and reliable. For example, a DDoS detector may need to monitor traffic coming from multiple locations via several switches. However, it cannot be implemented by routing all traffic through a single switch since it is inherently not scalable. Instead, it must be implemented in a distributed manner. Furthermore, in order to detect and mitigate an attack, a DDoS detector must aggregate per-packet source statistics across all switches in order to correctly identify super-spreaders sending to too many destinations. Similarly, in multi-tenant clouds, per-user policies, such as rate limiting, cannot be implemented in a single switch because user's VMs are often scattered across multiple racks, so the inter-VM traffic passes through multiple switches.

Distributed state management is, in general, a hard problem, and it becomes even harder in the context of programmable data-plane switches. In the "traditional" host-based NF realm, several methods have been proposed to deal with distributed state. These include remote access to centralized state storage [39] and distributed object abstractions [77], along with checkpoints and replication mechanisms for fault tolerance [64, 71]. Unfortunately, few of these techniques transfer directly to the programmable switch environment. These switches have the capability to modify state on *every* packet, allowing them to effectively implement stateful NFs. However, distributing the NF logic across multiple switches is extremely challenging as it requires synchronizing these frequent changes under harsh restrictions on computation, memory and communication.

Existing systems that implement NFs over multiple switches do so by designing ad hoc, application-specific protocols. Recent work on data-plane defense against link flooding [36], argues for data-plane state synchronization among the switches, but provides no consistency guarantees. While applicable in this scenario, it would not be enough in other applications, as we discuss in our analysis (§4). A more common solution, usually applied in network telemetry systems,

is to periodically report the per-switch state to a central controller [1,6,18,25,26,29,49,78]. Such systems need to manage the state kept on each switch and to determine when and how the central controller is updated – navigating complex tradeoffs between frequent updates leading to controller load and communication overhead versus stale data leading to measurement error. In contrast to these approaches, we seek a solution that supports general application scenarios *without relying on a central controller in failure-free runs*, while allowing all switches to take a *consistent action as a function of the global state*, e.g., to block a suspicious source in the DDoS detector.

We describe the design of such a general distributed shared state mechanism for data-plane programs, **SwiSh**. Inspired by distributed shared memory abstractions for distributed systems [41,48], SwiSh provides several replicated shared variable abstractions with different consistency guarantees. At the same time, SwiSh is tailored to the needs of NFs and co-designed to work in a constrained programmable switch environment.

Our analysis reveals three families of NFs that lend themselves to efficient in-switch implementation, with distinct consistency requirements. For each family we explore the triple tradeoff between consistency, performance, and complexity. We design (1) Strong Read-Optimized (SRO): a strongly consistent variable for read-intensive applications with low update rates, (2) Eventual Write-Optimized (EWO): an eventually-consistent variable for applications that can tolerate inconsistent reads but require frequent writes, and (3) Strong Delayed-Writes (SDW): a novel consistency protocol which efficiently synchronizes multi-variable snapshots across switches while providing a consistency and correctness guarantee known as r-relaxed strong linearizability [27].

SDW is ideal for implementing concurrent sketches, which are popular in data-plane programs [12, 13, 24, 30, 35, 38, 51–54, 78, 83]. Unlike eventually consistent semantics, the r-relaxed strong linearizability offered by SDW enables principled analysis of concurrent sketches. This property enables the derivation of precise error bounds and generalizes to different sketch types, such as non-commutative sketches [67].

Implementing these abstractions efficiently in a switch is a challenge, and it involves judicious choice of hardware mechanisms and optimization targets. Our main ideas are: (1) *minimizing the buffer space* due to the scarcity of switch memory, even at the expense of higher bandwidth; (2) using the *in-switch packet generator* for implementing reliable packet delivery and synchronization in the data-plane.

We fully implement all the protocols in Tofino switches and devise reusable APIs for data-plane replication. We evaluate the protocols both in micro-benchmarks and in three real-world distributed NFs: a rate limiter, a network address translator (NAT) and a DDoS detector. Our novel SDW protocol achieves micro-second synchronization latency and offers about four orders of magnitude higher update rates compared to a central controller or SRO. We show that SDW (1) achieves stable 99th percentile replication latency of 6μsec when running on four programmable pipes (two per switch), thus sharing state both among local and remote pipes; (2) scales to 32 switches when executed in a large-scale emulation and fits switch resources even for 4K switches; (3) requires linear number of replication messages in state size which is independent from the number of actual updates to the state.

We show that SDW is instrumental to achieving high performance in applications: the centralized controller fails to scale under growing application load, whereas SDW-based versions show no signs of performance degradation.

This paper extends our workshop paper [82] by introducing the SDW protocol, as well as providing an implementation and evaluation of SRO and EWO.

In summary, this work makes the following contributions:

- Analysis of memory consistency requirements and access patterns of common NFs suitable for in-switch execution,
- Design and implementation of strongly- and eventually consistent shared variables, as well as a new SDW consistency protocol specifically tailored for in-switch implementation, which guarantees consistent snapshots and provably provides r-relaxed strong linearizability which facilitates implementation of concurrent sketches,
- An implementation and evaluation of three distributed NFs on Tofino switches demonstrating the practicality and utility of the new abstractions.

2 Background: Programmable Switches

The protocol independent switch architecture (PISA) [8] defines two main parts to packet processing. The first is the parser which parses relevant packet headers, and the second is a pipeline of match-and-action stages. Parsed headers and metadata are then processed by the pipeline. The small (~10 MB) switch memory is shared by all pipeline stages. Often, switches are divided into multiple independent pipes [34], each serving a subset of switch ports. From the perspective of in-switch applications, the pipes appear as different switches, so stateful objects are not shared between them.

PISA-compliant devices can be programmed using the P4 language [73]. P4 defines a set of high-level objects that consume switch memory: tables, registers, meters, and counters. While tables updates require control-plane involvement, all other objects can be modified directly from the data-plane.

A data-plane program processes packets, and then can send them to remote destinations to the control-plane processor on the switch, or to the switch itself (called *recirculation*).

Switches process packets atomically: a packet may generate several local writes to different locations, and these updates are atomic in the sense that the next processed packet will not see partial updates. Single-row control-plane table updates are atomic w.r.t. data-plane [74]. These properties allow us to implement complex distributed protocols with concurrent state updates without locks.

Although not a part of PISA, some switches add packet generation support. Packet generators can generate packets directly into the data-plane. For example, the Tofino Native Architecture (TNA) [34] allows generation of up to 8 streams of packets based on templates in switch memory. The packet generator can be triggered by a timer or by matching certain keys in recirculated packets.

3 Motivation

The Case for Programmable Switches as NF Processors. The modern data center network incorporates a diverse array of NFs beyond simple packet forwarding. Features like NAT, firewalls, load balancers, and intrusion detection systems are central to the functionality of today's cloud platforms. These functions are stateful packet processing operations, and today are generally implemented using software middleboxes that run on commodity servers, often at significant cost.

Consider an incoming connection to a data center service. It may pass through a DDoS detection NF [3, 58], which blocks suspicious patterns. This service is stateful; it collects global traffic statistics, e.g., to identify "super-spreader" IPs that attempt to flood multiple targets. Subsequently, traffic may pass through a load balancer, which routes incoming TCP connections to multiple destination hosts. These are stateful too: because subsequent packets in the same TCP connection must be routed to the same server (a property dubbed per-connection consistency), the load balancer must track the connection-to-server mapping. Both DDoS detectors and load balancers are in use at major cloud providers [19,61], and handle a significant fraction of a data center's incoming traffic. Implemented on commodity servers, they require large clusters to support massive workload.

Programmable data-plane switches offer an appealing alternative to commodity servers for implementing NFs at lower cost. Researchers have shown that they can be used to implement many types of NFs. For data center operators, the benefit is a major reduction in the cost of NF processing. Whereas a software-based load balancer can process approximately 15 million packets per second on a single server [19], a single switch can process 5 *billion* packets per second [33]. Put another way, a programmable switch has a price, energy usage, and physical footprint on par with a single server, but can process *several hundred times* as many packets.

Distributed Switch Deployments. Prior research focused on showing that NFs can be implemented on a single switch [45, 57]. However, realistic data center deployments universally require multiple switches. We see two possible deployment scenarios. The NF can be placed in switches in the network fabric. For example, in order to capture all traffic, the load balancer would need to run on all possible paths, e.g., by being deployed on every core switch or every aggregation switch. Alternatively, a cluster of switches (perhaps located near the ingress point) could be used to serve as NF accelerators. Both are inherently distributed deployments: they require multiple switches in order to (1) scale out, (2) tolerate switch failures, and (3) capture traffic across multiple paths.

The challenge of a distributed NF deployment stems from the need to manage the global state shared among the NF instances, which is inherent to distributed stateful applications. Specifically, packet processing at one switch may require reading or updating variables that are also accessed by other switches. For example, the connection-to-server mapping recorded by the load balancer must be available when later packets for that connection are processed – even if they are processed by a different switch, or the original switch fails. Similarly, a rate limiter would need to track and record the total incoming traffic from a given IP, regardless of which switch is processing it.

SwiSh provides a shared state mechanism capable of supporting global state: any global variable can be read or written from any switch. SwiSh transparently replicates state updates to other switches for fault tolerance and remote access. In case of state locality, only a subset of the switches would replicate that state [82].

The Case for Data-Plane Replication. Control-plane mechanisms are commonly used for replicating the switch state [7, 11, 43, 56]. However, the scalability limitations of this approach have been well recognized, and several recent works focus on improving it by distributing the control-plane logic across a cluster of machines or switches [43, 81]. SwiSh proposes instead to replicate the state in the data plane.

Data plane replication enables supporting distributed NFs that read or modify switch state on *every packet*. This new capability of programmable data-plane switches allows implementations of more sophisticated data-plane logic than traditional control-plane SDN.

As we will see in §4, applications use state in diverse ways. Some are read-mostly; others update state on every packet. Some require strong consistency among switches to avoid exposing inconsistent states to applications (e.g., a distributed NAT must maintain correct mappings to avoid packet loss), while others can tolerate weak consistency (e.g., rate limiters that already provide approximate results [63]). SwiSh provides replication mechanisms for different classes of data that operate at the speed of the switch data-plane.

At the same time, data-plane replication offers an opportunity to build a more efficient replication mechanism without additional control-plane processing servers. Furthermore, data-plane replication can take advantage of unique programmable hardware characteristics that are not available in a traditional control-plane. For example, the atomic packet processing property enables a multi-location atomic write to the shared state. We leverage this feature to enable fast processing of acknowledgments entirely in the data-plane for our strongly-consistent replication protocol (§6.1).

Control-plane replication is not enough. Managing a globally shared state in a programmable data-plane switch requires

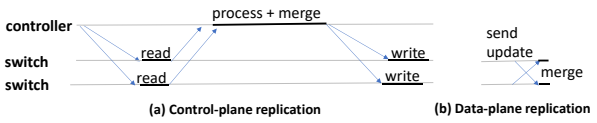


Figure 1: Data-plane vs. control-plane replication

a new approach: replication protocols that run in the control-plane cannot operate at this rate at scale.

Figure 1 shows the cycle performed by a controller to synchronize between switches, and contrasts that with data-plane replication. The controller periodically queries the switches, collects information, processes it, and sends the updates back. Merely reading and updating the register states in switches is quite slow. We measured an average latency of 507msec to read a sketch with 3 rows each with 64K 4-byte registers from the on-switch control-plane;[1] updates are similar. This latency limits the rate at which the data can be retrieved from switches.

Moreover, the central controller may become the bottleneck quite quickly. For example, recent work on DDoS detection that used a central controller to query switches reported a maximum update rate of once in 5 seconds [53] because it could not accommodate faster updates.

In contrast, data-plane replication reads from and writes to registers much faster: we measured 486μseconds to read the same sketch from the data-plane, which is over three orders-of-magnitude faster than the control-plane access. Further, in-switch processing time is negligible as well.

These properties make data-plane replication an obvious choice for building stateful distributed NFs.

4 Application Consistency Requirements

We study the access patterns and consistency requirements of a few typical NF applications that have been built on PISA switches. Table 1 summarizes the results.

We identify three families of consistency requirements:

1. **Strong consistency**: Workloads cannot tolerate inconsistency between switches – a read must see a previous write. These are usually read-intensive workloads that can tolerate infrequent, but expensive writes;
2. **Weak (eventual) consistency**: Mixed read/write workloads tolerate arbitrary inconsistency;
3. **Bounded-delay consistent snapshots**: Mixed read/write workloads that tolerate inconsistency for a bounded time – a read must see all but a bounded number of previous writes, yet require that *all switches* read from a consistent state. These requirements are typical for sketches.

Below, we describe how these consistency requirements arise in several in-switch applications.

[1] We use BfRt API (C++) and average over 100 iterations.

4.1 Strong Consistency

Network Address Translators (NATs) share the connection table among the NF instances. The table is queried on every packet, but updated when a new connection is opened; table rows require strong consistency, or it may lead to broken client connections in case of multi-path routing or switch failure. Also, NATs usually manage a pool of unassigned ports; however, the pool can be partitioned among the switches into non-overlapping ranges to avoid sharing.

Stateful firewalls monitor connection states to enforce context-based rules. These states are stored in a shared table, updated as connections are opened and closed, and accessed for each packet to make filtering decisions. Like the NAT, the firewall NF requires strong consistency to avoid incorrect forwarding behavior.

L4 load balancers [57] assign incoming connections to a particular destination IP, then forward subsequent packets to the appropriate destination IP. Per-connection consistency requires that once an IP is assigned to a connection, it does not change, implying a need for strong state consistency.

Observation 1. These workloads require strong consistency, but they update state infrequently, making a costly replication protocol more tolerable. Moreover, most of these examples use switch tables that should be modified *through the control-plane*, naturally limiting their update rate. For example, the NAT NF uses control-plane to update the connection table. We leverage this observation when designing the replication protocol for this class of NFs.

4.2 Weak (Eventual) Consistency

Rate limiters restrict the aggregated bandwidth of flows that belong to a given user. The application maintains a per-user meter that is updated on every packet. The meters are synchronized periodically to identify users exceeding their bandwidth limit and to enforce restrictions. Maintaining an exact network-wide rate across all switches would incur a very high overhead and is therefore unrealistic. So rate limiters can tolerate inconsistencies, but the meters must be synchronized often enough [63] to minimize discrepancy.

Intrusion prevention systems (IPS) [47] monitor traffic by continuously computing packet signatures and matching against known suspicious signatures. If the number of matches is above a threshold, traffic is dropped to prevent the intrusion. This application can tolerate transient inconsistencies: it is acceptable for a few malicious packets to go through immediately after signatures are updated.

Observation 2. Some NFs tolerate weakly consistent data, potentially affording simpler and more efficient replication protocols. However, as we will describe next, other functions may defer the writes to be once in a window, but do require to have a consistent view of prior writes among all the switches.

	Application	State	Write frequency	Read frequency
Strong consistency	NAT	Translation table	New connection	Every packet
	Firewall	Connection states table	New connection	Every packet
	L4 load-balancer	Connection-to-DIP mapping	New connection	Every packet
Weak consistency	Intrusion prevention system	Signatures	Low	Every packet
	Rate limiter	Per-user meter	Every packet	Every window
Bounded delay consistent snapshot	DDoS detection	Sketch	Every sampled packet	Every packet
	Microburst detection	Sketch	Every packet	Every window

Table 1: NFs classified by their access pattern to shared data and their consistency requirements.

4.3 Bounded-Delay Consistent Snapshots

We assign mixed read/write applications that use data sketches to this class. Data sketches are commonly used in data-plane programs [12, 13, 24, 30, 35, 51, 52, 54, 78]. They are probabilistic data structures that efficiently collect approximate statistics about elements of a data stream.

Below we consider two examples of sketch-based NFs.

Microburst detection identifies flows that send a lot of data in a short time period. ConQuest [13] is a recent sketch-based system for a single switch, which uses a sliding window mechanism composed of a group of Count-Min sketches (CMS) [14]. At most one sketch is updated on every packet.

DDoS detection [45] requires tracking the frequency of source and destination IPs using a CMS with bitsets [80]. The sketch is updated on every packet, but sampled periodically to trigger an alarm when IP frequencies cross a threshold.

Strongly consistent read-optimized protocols are too costly for such workloads due to their write-intensive nature. Fortunately, because a data sketch is inherently approximate, it does not require strong consistency – it is acceptable for a query to miss some updates. Moreover, sketches are typically *stream-order invariant* [67], meaning that the quantity they estimate (such as number of unique sources, heavy hitters, and quantiles) does not depend on the packet order.

At the same time, sketches generally cannot tolerate weak consistency either. With no guarantee of timeliness, sketches might be useless. A DDoS attack might be over by the time it is detected. Moreover, the attack might be detected at one location much earlier than it is detected at another, leading to an inconsistent response. Furthermore, sketches have known error bounds (see [15] and others). These bounds are violated if updates are arbitrarily delayed [27, 66], making it hard to reason about the impact of sketch errors on the application.

Observation 3. Sketches require a *bounded-delay consistent snapshot* consistency level. Formally, it provides *r-relaxed strong linearizability* (Appendix A), which supports sketch applications with provably bounded error. Intuitively, *r*-relaxed strong linearizability guarantees that accesses to shared data are equivalent to a sequential execution, except that each query may "miss" up to *r* updates. SwiSh supports this consistency level using its novel Strong Delayed-Write (SDW) protocol, which provides a consistent snapshot of the sketch at all the replicas, while delaying reads until such a snapshot is constructed.

5 SwiSh Abstractions

SwiSh provides the abstraction of shared variables to programmable switches. This section describes the interface and the types of semantics it offers for shared data.

System model. We consider a system of many switches, each acting as a replica of shared state. Switches communicate via the network, and we assume a standard failure model: packets can be dropped, duplicated and arbitrarily re-ordered, and links and switches may fail. Since switches are comprised of multiple independent pipes with per-pipe state (§2), we consider a pipe rather than a switch, a node in the protocol. We use the terms pipe and switch interchangeably.

Data model. The basic unit of shared state is a *variable*, associated with a unique key, which exposes an API for updating the variable (potentially using general read-modify-write functions), and reading it. The API is thus available on all switches, and variables are read and updated through a distributed protocol. SwiSh supports three types of variables which have different semantics and are accessed through different protocols:

1. *Strong Read-Optimized (SRO)* variables provide strong consistency (linearizability);
2. *Eventual Write-Optimized (EWO)* variables have low cost for both reads and writes, but provide only eventual consistency;
3. *Strong Delayed-Writes (SDW)* variables provide strong consistency (linearizability), but expose writes (even to the local replica) only after their values have been synchronized across the replicas.

We require that, no matter which semantics are used, all variables eventually converge to a common state. To this end, we require that variables be *mergeable*. We consider two merging policies: LWW as a general method, and Conflict-Free Replicated Data Types (CRDTs) as specialized mergeable data types that implement common data structures that are used in NFs. A general way to merge variables is to assign

an order to updates and apply a last-writer-wins (LWW) policy. The merge function applies an update if and only if its version number is larger than the local one. Unique version numbers can be obtained by using a switch ID as a tie breaker in addition to a timestamp attached to each write request.

In some cases, updates can be merged systematically. These are discussed in the literature of Conflict-Free Replicated Data Types (CRDTs), which offer *strong eventual consistency* and *monotonicity* [69]. Monotonicity prevents counter-intuitive scenarios such as an increment-only counter decreasing.

Counters are a natural application for this technique, as they are common in NFs (§4) and have a straightforward CRDT design. An increment-only counter can be implemented by maintaining a *vector* of counter values, one per switch. To update a counter, a switch increments its own element; to read the result, it sums all elements. To merge updates from another switch, a switch takes the largest of the local and received values for each element. Further extensions support decrement operations [69].

Variables may be used to store different data types, such as array entries, read/write variables, sets, and counters. They are implemented using appropriate stateful P4 objects.

6 In-Switch Replication Protocols

Below we assume that switches do not fail; we relax this assumption in §6.4.

6.1 Strong-Read Optimized (SRO)

The SRO protocol is based on chain replication [76], as shown in Figure 2a, adapted to an in-switch implementation with the following key difference: instead of contacting the tail for its latest version and keeping multiple versions per variable, we forward reads to pending writes to the tail.

SRO provides per-variable linearizability [28], because writes are blocking and reads concurrent to writes are processed by the tail node. Its write throughput is limited by the need to send packets through the control plane.[2] Note, however, that many read-intensive NFs already require control plane involvement for their updates, such as NATs, firewall and load balancers [57].

A variation of this protocol, used in many systems, including CRAQ [72] and ZooKeeper [31], reduces the read latency by performing local reads, yet offers weaker semantics [46].

6.2 Eventual Write-Optimized (EWO)

Both variants of the read-optimized protocol have a high write cost. Because supporting both strong consistency and frequent updates is fundamentally challenging, we offer relaxed-consistency variables. This is acceptable for many write-intensive applications, as discussed in §4.

Reads from EWO variables are performed locally, and writes are applied asynchronously. That is, when a switch receives a packet P that modifies state, it modifies its local state, emits any output packet P' immediately, and asynchronously sends a write request to all other switches (Figure 2b). A more sophisticated version can employ batching to avoid flooding the network with updates, and instead send the write request after accumulating several updates.

Unlike SRO, we do not delegate the problem of reliable write delivery to the control plane because it does not scale for write-intensive workloads. Instead, switches periodically synchronize each EWO variable from the data plane. This design choice avoids expensive buffering and re-transmission logic in the data-plane.

Periodic synchronization overcomes the issue of lost packets. As updates to EWO variables are idempotent, packets can be arbitrarily duplicated with no effect. Finally, due to updates being commutative, packet reordering has no effect.

We note that this protocol is simple, but it leads to inconsistent replicas and would incur high bandwidth overheads. With over-subscribed links [23], excessive replication traffic would only worsen the congestion. The following protocol overcomes these limitations.

6.3 Strong Delayed-Writes (SDW)

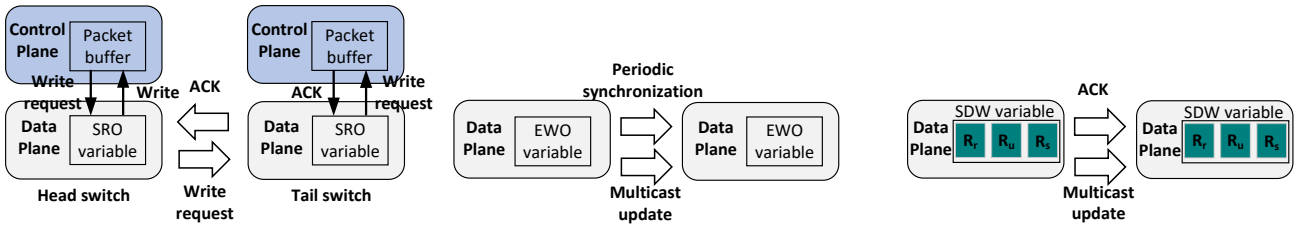
As explained in §4, certain NFs tolerate inconsistencies among switches, but require state convergence within a bounded time. For such NFs, SwiSh offers *strong delayed-writes* (SDW) variables, ensuring semantics known as r-relaxed strong linearizability [27]. These semantics guarantee that every read of a variable observes all but a bounded number of updates. If the variable is used to store a data sketch, then r-relaxed strong linearizability often directly implies error bounds on the sketch's estimate [67].

SwiSh batches updates into windows, and synchronizes window advancement (Figure 2c). The complete protocol and its analysis appear in Algorithm 1 in Appendix A; below is an informal overview.

To distribute a variable R, each switch maintains three objects holding copies of R: R_u, R_r and R_s. At any given time, R_u is updated, R_r is queried, and R_s is synchronized (merged) across switches. The objects' roles are switched in a round-robin manner on window advancement.

All switches run the same protocol. At the start of a window, all switches send the contents of R_s to all the others. Any (local) update is applied to R_u, and any query is executed on R_r. Once a switch receives R_s from all other switches, and furthermore receives ACKs from all other switches that they received its R_s, it advances to the next window.

[2] NetChain [37] implements chain replication entirely in the data plane. The difference is that NetChain is a service and clients are responsible for retrying operations. Our switches are effectively the "clients" and must buffer output packets and retry requests.



(a) SRO: Based on chain replication. Relies on control-plane for packet buffering.

(b) EWO: Updates are broadcast. Switches periodically send their state for reliability.

(c) SDW: Updates are sent in rounds. Switches advance to the next round after receiving ACKs and updates from others.

Figure 2: A high-level overview of in-switch replication protocols.

On window advancement, the objects are rotated, so R_u becomes the new synchronization variable R'_s, R_r is merged into R_s and then cleared – it becomes the new update object R'_u, and the synchronized buffer R_s becomes the new read buffer R'_r. Thus, after the synchronization of window w completes, R'_u is empty and ready to accumulate updates of window $w+1$, R'_r reflects all updates that occurred in all switches in all windows up to $w-1$, and R'_s reflects all updates done in windows up to $w-1$ in all switches, as well as local updates done in window w.

Crucially, as we prove in Appendix A, this protocol *guarantees* that a query in some window w sees all updates occurring in all windows $\leq w-2$. We also prove that, by bounding the number of updates in a window to B, every query sees all but at most $2NB$ updates that occur before it, where N is the number of switches.

Multi-variable snapshots. Another advantage of the window protocol is that it allows applications to take *consistent snapshots* [59] over a collection of SDW variables by advancing the window simultaneously for all of them. This means that we can support multi-variable queries (for instance, collecting an array of counters as used in a CMS), and ensure that all queries see update batches in a consistent order. Thus, given two updates u_1 and u_2 occurring in different switches, it is impossible for a query at one switch to see a state reflecting only u_1 (and not u_2) while a query at another switch sees only u_2 (and not u_1).

6.4 Handling Failures

We now consider fail-stop switch failures. We assume that a central controller can detect which switches have failed.

SRO. When a switch fails, the chain becomes partitioned. First, we reconnect the chain by bypassing the failed node; if the failed switch is the head, the second node in the chain assumes its responsibility. This follows the standard chain replication protocol. A new switch is added to the end of the chain. It starts to process writes, but does not replace the tail until the data transfer to it is complete. This requires control plane involvement.

The control plane on one of the switches takes a snapshot of its state, and then resends all pending write requests through the normal data plane protocol. These writes contain the sequence number at the time of the snapshot to prevent overwriting newer values with old ones. Once the new switch has acknowledged all writes, it replaces the tail.

EWO. Because live replicas regularly synchronize their entire state, this synchronization protocol is inherently robust to switch and link failures. The failed switch is removed from the multicast group. Once a new switch replaces the faulty one, it is added to the multicast group, and begins serving reads after obtaining an initial view of the shared state.

SDW. The protocol inevitably stalls once a failure occurs (i.e., the local window ids stop increasing). Denote the maximum window at a correct switch at the time of the failure by $wmax$. The difference between the local window ids at each pair of switches is at most one. Thus, every stalled switch is in window $wmax$ or $wmax-1$.

We reconcile the states of the surviving switches as follows: a controller reads the states of all switches. It collects the state of R_r in some switch that is in window $wmax$ and sends it to all switches that are in window $wmax-1$ (if any), so they advance to window $wmax$. The controller merges all the R_s objects to yield the most up-to-date state for window $wmax+1$ and broadcasts it to all switches, thus updating their R_s objects to the merged state. Then it removes the failed switch from the multicast group and the switches resume the protocol from window $wmax+1$.

Adding a new replica is a two-stage process: increasing the expected number of ACKs on correct switches and making sure that all switches are in the same window, which stalls window progression, followed by adding the new replica to the multicast group of each correct switch. The new switch begins serving reads after the current window completes.

We note that during the recovery the updates to the live switches are not lost, but rather accumulated in local switch

replicas R_u. These updates are then synchronized during the recovery. Thus, this protocol is not time critical and can be implemented in control-plane without adding code to the resource-constrained data-plane.

7 Design

We explain the messaging mechanism shared by all protocols, and then describe the SDW design. SRO and EWO closely follow their descriptions in §6.

7.1 Replication Message Exchange

Packet format. Switches exchange replication packets, updates, and acknowledgments with each other to replicate state. Replication packets are IP packets; therefore, by assigning an IP per switch, these packets can be routed using standard L3 routing protocols. Besides Ethernet and IP headers, each packet includes a single bit indicating whether the packet is an update/write request or an ACK, the keys and values accessed by the write, and, in SRO, also a sequence number. For example, in an SRO NAT implementation, the keys are the source IP and source port, and the values are the translated IP and port. In an SDW DDoS application, the keys are sketch indices and the values are counter increments.

Reliable delivery. A major challenge in data-plane replication is ensuring delivery of replication packets. Current switches do not provide enough control over internal switch buffers to store and retransmit a packet from the data-plane.

We identify two cases that require buffering. First, there are *replication* packets generated by each switch as part of the replication protocol. Such packets must be reliably delivered in SDW. Second, there are *write* packets that are received from external sources (not from a switch) and update the NF state in a switch. In SRO these packets cannot be externalized until the updated state is synchronized among the switches.

We handle these two cases separately. For SDW replication packets we keep the state being replicated at the application level until acknowledged, instead of buffering the packet. Then an *ACK-check* packet is *periodically* generated by the packet generator. If the sent replication packet has not yet been acknowledged by other switches, the ACK-check triggers its retransmission. Here we use the recirculation trigger for the packet generator to initiate a batch of packets at once.

In SRO, the packets themselves must be buffered since their content is not reproducible by the switch. Buffering in the data-plane is an open problem and we leave it for future work. However, since most NFs that use SRO would require the updates to be performed via the control-plane anyway (Observation 1, §4.1), we relay the reliable delivery to the control-plane of the switch that receives the write packet. The cost of buffering and retransmission is negligible, as we show in §9.2. Future switches might enable table updates in the data-plane, motivating data-plane buffering mechanisms to avoid control-plane involvement in replication.

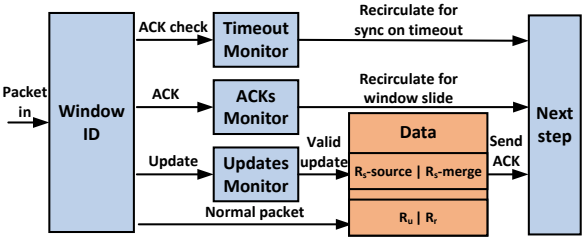


Figure 3: SDW high-level design. Blue boxes are reusable P4 control blocks, while the orange box is application-dependent.

Packet duplication and reordering. SRO replication packets are shipped with a sequence number allowing each replica to apply updates in order and to reject updates with sequence numbers lower than those already processed. In EWO, updates are idempotent and monotonic so detecting duplication and reordering is already a part of the merging process. We explain the SDW implementation in detail below.

7.2 Strong Delayed-Writes (SDW)

As presented in Figure 3, the data structure used in SDW is organized as two register arrays, each of which holds a 32-bit pair. At any given time, one window is designated for reading and writing, namely, its register arrays used as the R_u and R_r objects in the SDW protocol. The other window is used in the sync operation. The sync object R_s is divided into two register values, one, denoted R_s-*merge*, receives data from other switches, while the other, called R_s-*source*, holds the local state as sent to other switches at the beginning of the window. This separation is important to allow retransmissions (§7.1).

Synchronization. The alternating window structure enables SwiSh to ensure that the R_r in each window are consistent across all switches. Each time synchronization is initiated, the content of the R_s-source register is sent to all the switches, and the content received from all of the switches is merged in the R_s-merge register. Note that each switch also receives (and hence merges) its own update. Once all the updates are received, the content of R_s-merge is identical across the switches, so the synchronization for that window is finished.

Unfortunately, a full sketch cannot be read while processing a single packet, so we send the sketch column by column. For simplicity, we first explain handling of a single-column sketch, and then discuss the complete implementation.

Each switch maintains two bitmaps: one, to track ACKs that other switches received its updates, and the other, that it received all updates from them. If an update was lost, the sketch is retransmitted.

Window advancement. The last update to complete the bitmaps signifies the completion of the sync round for the

switch. The switch advances the window ID, swaps the roles of the registers, and starts a new sync process again. During this swap the following arrays are swapped: R_s-merge swaps with R_r, and R_s-source swaps with R_u.

Ready phase. Because round advancement is a local event, the switches do not advance their windows in lock-step. Thus, a switch may receive an update for the *next* window, which will be dropped and retransmitted later. Buffering such updates would significantly increase the memory footprint. Instead, we introduce the *ready phase*. Once a switch advances its local window it broadcasts a *ready* packet to all the rest. A switch starts broadcasting its updates only after it receives *ready* packets from all other switches (existing bitmap can be reused for tracking). This phase ensures that an update will not be sent to a switch that is not yet ready to merge it. Ready packets are retransmitted upon timeout, though in the experiments we did not encounter such cases. In our evaluation (§9.2) we show that the *ready* phase is critical to achieving predictable replication latency.

Multi-column sketches. Ideally, each switch should track each column being synchronized separately, to filter duplicates and retransmit lost updates. This solution would be too memory-consuming and would limit the sketch size, however. We make two optimizations. First, for R_s-merge, we retain the original bit-per-switch tracking, so a switch sends an ACK only when a full sketch was received. Thus, we always retransmit a full sketch. Second, we maintain a counter per switch which tracks the index of the next column to be updated. Only updates that match this counter are accepted. This approach is correct: it handles duplicates and packet reorders. However, while it is efficient for duplicates, it would lead to sketch retransmission if packets are reordered. We assume that this is a rare event, however, because IP routing in data centers usually maintains the same path for a given flow.

We implement both approaches. The bitmap-per-column implementation allows using sketches with 3 rows and 64K entries per-row and can scale up to 32 switches. The counter-per-switch implementation can scale to 4K switches for the same sketch size.

Note that changing the communication pattern from an all-to-all to an aggregation tree, e.g. as in SwitchML [68], may also reduce the per-switch state but at the cost of increasing replication latency.

Register initialization. There is no way to iterate over all the registers and reset them. Instead, we piggyback initialization on the first write and use a single bit in each register to determine whether the register is initialized. These bits are reset during the processing of sync packets.

Reducing replication bandwidth. Recall that SDW is used for a collection of variables, stored in register arrays, over which queries can take consistent snapshots. Our current implementation of the sync protocol exchanges a full state snapshot (including all variables) rather than only the ones that were updated. The challenge for selective updates is that the switches send a varying number of packets in each window (due to hardware limitations, the state does not fit in one packet), and so the destination does not know when to acknowledge the state receipt. To overcome this challenge, switches count the number of updates that they send in a window and piggyback this number on the last update.

Recovery. The recovery protocol follows the algorithm mentioned above (§6.4), but also considers the ready phase and sends ready packets to allow switches to make progress before removing the faulty switch from the replica group. SDW does not rely on a centralized controller in failure-free runs. However, as writes are not lost upon switch failures, recovery is kept off of the critical-path and is not time-sensitive. Therefore, we chose to offload the recovery protocol to a centralized controller which frees switch resources.

8 Implementation

We expound the implementation of SRO and EWO, and then we describe the distributed NFs implemented on top of SwiSh. Last, we provide additional implementation details and limitations.

8.1 Strong Read-Optimized (SRO)

We run the replication protocol in the control-plane logic. Write packets (packets that modify state) are forwarded to the control plane, which subsequently generates a write request forwards it to the head of the chain.

The way write requests are handled depends on the storage type where the data is stored in the switch. If the data can be modified only from the control-plane, then write requests must be processed by the control-plane at each switch in the chain. Otherwise, write requests can be processed directly in the data-plane. We implement reading from tail by tunneling the reading packets through the tail switch to its destination with an outer IP header (similar to IP-in-IP). While a write is pending, the key is flagged as "read-from-tail", causing subsequent reading packets to be sent to the tail.

8.2 Eventual Write-Optimized (EWO)

The EWO logic uses the following types of packets: (a) Regular packets from applications – read and write to the shared state. (b) Update packets – sent when the local state changes. The recipient merges these updates with its local state. (c) Generated packets – for reliable message delivery. Because each register array can only be accessed once per packet, if the state consists of an array, we generate one packet per array entry. If we maintain multiple register arrays, they can be accessed by a single packet.

Reads are local, while writes require sending an update to other switches. To broadcast updates, we use egress-to-egress mirroring to create a truncated copy of the original

write packet. We use the multicast engine to create a copy of the update packet for each switch in the replica group. Each copy is then modified to carry the updated values.

The application state each switch maintains depends on the particular data structure. For example, to implement a shared counter, each switch maintains a vector of counters, one per switch in the replica group. On the other hand, growing only sets and LWW variables do not require sharding.

In order to ensure eventual consistency in the face of lost update packets, a periodic background task is implemented by using the switch's packet generator that iterates over the register array, forming write update packets consisting of the indices and values for each register, and forwarding each one to a randomly-selected switch in the replica group.

8.3 Distributed NFs

We prototype three multi-switch NFs. We also prototype a distributed version for all of these NFs built using the protocols in SwiSh. In addition, for two of them we also implement a version that uses a central controller for synchronization.

Network Address Translator (NAT). This application maps internal source IPs to external source IPs. Each switch maintains two translation tables – one that maps (external source IP, external source port) to (internal source IP, internal source port) and another that performs the inverse mapping. We implement a distributed NAT using the SRO protocol. It requires no changes to the data-plane logic.

Super-spreaders detection (DDoS). This application detects source IPs that communicate with more than 1000 unique destination IPs. Inspired by OpenSketch [80], we implement it using a CMS, with a bit set instead of counters. Packets are first sampled based on the (source IP, destination IP) pair. Sampled packets set a single bit in the bitset in each row of the sketch. The bitset is used to estimate the number of unique destinations. Our implementation uses a sketch with 3 rows and 32K 32-bit wide bitsets per row.

We implement two designs based on a central controller. In both, the controller obtains the list of suspicious IPs from each switch, and decides to block IPs if the sum of different destination IPs for that source from both switches exceeds 1000, in which case it inserts an entry to the block list of each switch. However, there are two ways for the controller to obtain this data: (a) pull-based: each switch maintains a gradually growing list of potential IPs to block. The controller periodically *pulls* the delta in the list since the previous pull; (b) push-based: each switch sends a packet when it detects a potential IP to block. For simplicity we mark an IP as suspicious if it sends to more than 500 destinations, and construct the workload to send half of the packets from each source to one switch and another half to the other, thus the implementation works correctly for this case.

The distributed design replicates the sketch using the SDW protocol, each switch unilaterally decides to block a destination according to the replicated sketch, which essentially holds a global view of the network.

Rate limiter. We implement a rate limiter based on the token bucket algorithm [70]. In the single switch design, the controller periodically fetches rate estimations from each switch, calculates the token limit per each user and each switch, and writes it back to the switches. We implement two distributed versions, with EWO and SDW respectively. Switches replicate their own rate estimates for each user, and calculate their limit according to global traffic ratios.

8.4 Implementation Details

We implement SwiSh using $P4_{16}$ [73] and Intel P4 Studio 9.6.0 [32] for Tofino switches. We implement all protocols as described.

API. We expose the building blocks of each protocol's design as P4 control blocks [73]. We then use this API to implement our NF applications (§8.3).

Control Plane. For applications that use SRO variables, we implement the control-plane logic in C++ using the user space packet DMA API (kpkt). For the other protocols, we initialize the switch state using bfrt-python. We also utilize a simple TCP server in C++ for reading register values from the switch for the recovery protocol.

Limitations. Our current implementation does not include the required recovery logic for SRO because it is well-known and in-control plane, thus it does not challenge our design. Although independent to the number of switches in the replica group, the major limitation of replicated NFs is the increase in SRAM usage (×4). We fully implement recovery for SDW.

9 Evaluation

We evaluate the protocols and applications on two Tofino switches (each two pipes) and on 32 switches in an emulator. Our key observations are:

- Control-plane replication is too slow.
- SRO has high latency and low throughput.
- SDW is scalable and replicates large sketches in microseconds.
- For a DDoS detector, SDW responds instantly to an attack, blocking malicious packets, while central controller allows almost 50% of the packets to go through.
- For a rate limiter, SDW and EWO respond instantly to traffic changes, while central controller lags behind.

Setup. We use two machines with Intel Xeon Silver 4216 2.1 GHz CPUs, connected via two EdgeCore Wedge 100BF-32X programmable switches. The server is dual socket with 192 GB RAM. Hyper-threading and power saving are disabled. One machine acts as a traffic generator/consumer; it has two 100G Intel E800 NICs. The other acts as a central controller; it has two 40G NVIDIA ConnectX-4 Lx EN NICs.

Topology. We use the *leaf-spine* topology in which the switches are connected as shown in Figure 4, and run ECMP on one pipe and a NF on the second.

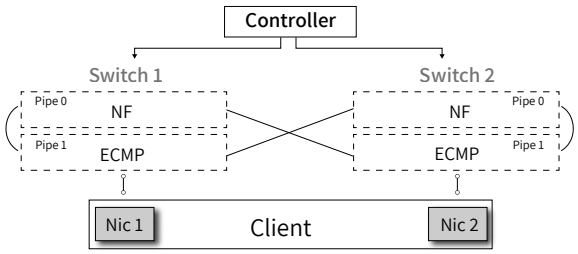


Figure 4: Testbed topology.

Performance measurement methodology. We build a DPDK-based packet generator. We evaluate SwiSh on a real packet trace, CAIDA [10], as well as synthetic workloads. Throughput is measured by the NIC and application-level performance counters. Latency is measured in software.

To measure the performance of the in-switch NFs and protocol implementations, we create a line-rate load (100Gbps, unless stated otherwise) on a single switch port. To validate that the performance obtained via this approach is representative of the switch under load over all its ports, we also run one experiment with *a fully loaded switch running at 2.1 Gpps* (§9.2). We show that the performance is almost the same as with a single port traffic, validating our methodology.

9.1 End-to-end benchmarks

NAT. We replay 10K packets from the CAIDA dataset and measure the per packet latency with and without replication. 21% of the packets are processed by the control plane (update packets), while the rest are processed in the data-plane. Figure 5d shows the latency distribution. SwiSh does not introduce any overheads for read packets, while update packets are taking about twice as long to get processed since they are batched in the control plane until the update is acknowledged by the other switch.

We also compare the throughput of the distributed version with the one on a single switch, while sending 64-byte packets at line rate to a single port. There are *no updates during the test*, as we wait for the handshake to complete. Therefore, both versions achieve line-rate throughput (112 Mpps).

Super-spreader detection. DDoS is configured to detect sources (IPs) that communicate with more than 1K different destinations. We create a trace where packets are sent from different source IPs, each with thousands of different destinations. Each source IP sends 10K packets.

In the experiment we replay a trace where we vary the number of packets that have different source IPs sent per second, while maintaining the absolute transfer rate from each source IP constant. This is a reasonable scenario where an attacker uses a botnet to generate malicious traffic while maintaining the transfer rate of each bot constant.

We compare the number of packets sent by each source IP relative to the number of packets received by the destination IP. Ideally, each source IP should be blocked after the first 1000 packets, therefore the ratio should be about 10%.

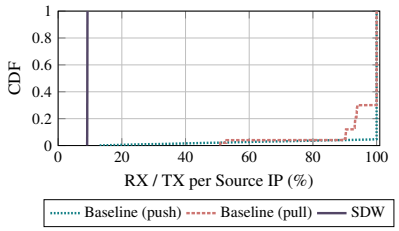
We compare the push and pull baselines with the implementation that uses SDW replication. Figure 5b shows that both versions of the centralized controller are quickly becoming overwhelmed and cannot keep up with processing the updates, failing to block packets. At 1.5K source IPs/second the push baseline breaks down because the push requests to block certain IPs from the switch get dropped at the host, thus their respective IPs are left unblocked. The results were obtained after increasing the socket receive buffers to 25MB.

To validate this result, we run the same workload fixed at 4K source IPs/sec. Figure 5a shows the distribution of the ratio of packets received per source IP across all source IPs. We observe that the pull design manages to block up to 30% of all the source IPs, but for each IP different number of packets leaked. Effectively, the pull design was unable to block traffic from 70% of the source IPs. That is because the controller collects batches of requests and handles them together, thus some source IPs manage to send more than others. However, the push design blocks only 5% of all the source IPs. The SDW-based design, shown as a vertical line at 10%, passes the first 10% of each source (which is our super-spreader detection threshold), and then blocks all the packets as expected.

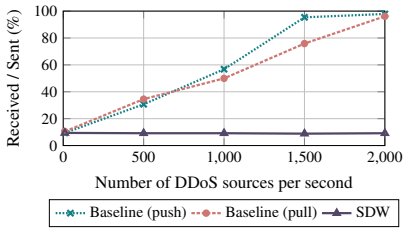
Per-user rate limiter. We set a limit of 2Mpps per-user and configure the rate limiter to re-estimate rates every 1ms.

We create a trace where packets are sent from different source IPs (each source defines a different user) with 40 unique users (sending rate is 2Mpps per user). The trace is comprised of alternating phases with a period of 5s. In even phases, all flows of a specific user are split equally between the two switches. While in odd phases, 90% of each user's flows are routed to one of the spine nodes and the rest 10% are forwarded to the other spine node. These alternations results in immediate changes in the per-user rate estimator that each switch maintains.

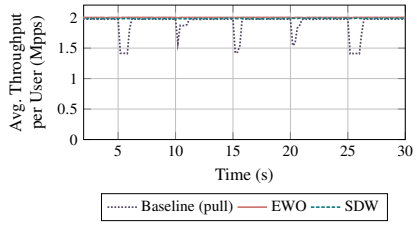
We compare our EWO and SDW protocols with a pull-based baseline and measure the average throughput per user over time. In the first 5 seconds of the experiment, the traffic is balanced so each switch runs at 1Mpps and the controller sets a per-user limit of 1Mpps on each switch. At the 5th second of the experiment, we change phases, and now one switch measures 1.8Mpps and the other switch measures 0.2Mpps. Because each switch was set to limit each user to 1Mpps, the first switch forwards only 1Mpps and the other switch forwards 0.2Mpps resulting in 1.2Mpps aggregate throughput. Figure 5c shows the average received throughput per-user over time at a sampling period of 200 ms. The baseline misses the phase changing point and allows the throughput to reduce to



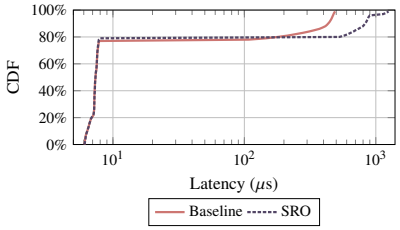
(a) The distribution of the ratio of packets received per source IP across all 4K source IPs.



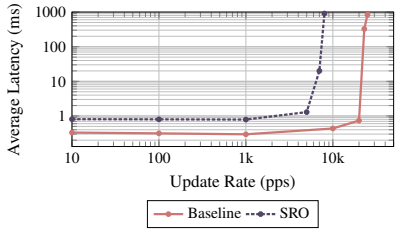
(b) The ratio of received packets for different number of DDoS sources per second.



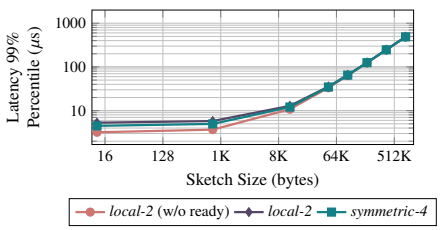
(c) The impact of reaction time on a rate-limiter when changing traffic distribution.



(d) NAT latency distribution with and without replication.



(e) NAT latency with and without replication when sending update packets at a variable rate.



(f) The window advancement period for SDW replication for different state sizes.

Figure 5: End-to-end and analysis results.

below 1.5Mpps, slightly more than the expected value due to sampling. SDW and EWO perform compareably. They both react immediately to traffic changes without throughput drops. EWO eagerly sends 40 updates every 1ms, allowing the other switch to immediately change its limit. SDW replicates such a small state (40 32bit values) under 10 microseconds (5f).

9.2 Analysis

SRO: Update rate. We measure the overhead introduced to update packets with replication. For this experiment we run NAT on spine switches and send update packets that are all processed by the control plane. We measure the per packet latency and report the average. Figure 5e shows that compared to the single switch, in the replicated setting the update rate is reduced by a factor of ×2.5. This is expected since each update packet generates a write request and an ACK that has to be processed on the other switch.

We observe that *the update rate of SRO is limited by the throughput of the control plane on a single switch*. SRO variables cannot sustain more than 20K updates per second (160Kbps). Update latency increases with the update rate because packets are buffered in the control-plane until acknowledged.

SDW: Window advancement latency. We measure the time each window absorbs updates before being advanced. We vary the state size being replicated, so for the smallest sketch the time to advance the window is the upper bound on the replication rate, constrained by the latency of updates between switches. There are no retransmissions in this experiment.

We replicate a sketch with 3 rows and vary each row size to up to 64K counters (total of 768KB in each sketch). We store the global timestamps of the first 10K window increments and read them at the end of the run.

We use the following topologies: (a) *local-2*: two local pipes on a single switch (we measure identical results compared to two remote pipes); (b) *symmetric-4* - four pipes, two in each switch, with a dedicated link between each pipe.

Figure 5f shows the 99th percentile latency to advance the window for each state size. As we see, the window can be advanced as fast as every 3μsec for the skectch of 4 bytes. The current bottleneck is the packet sending rate which, even within the switch (Recirculation), takes a few hundred nanosecs. This window advancement rate implies that the updates become visible after 6μsec (since R_u becomes R_r after two window advancements). For the sketch larger than 1K, the actual replication rate is about *13Gbps* between each pair of switches, which is about *five orders of magnitude* faster than SRO. We note that this rate is limited by the maximum packet rate (~160Mpps) of a single port. This is because replication packets hold only 12 bytes of data, which in turn is due to limited per-packet memory accesses imposed by the hardware. Optimizing the effective bandwidth is left for future work.

We observe negligible increase in the window advancement latency when adding two additional switches. This is because each switch updates all the others concurrently, hence no additional delay. The ready phase adds a constant latency overhead of 2μsec to each replication round.

SDW: Performance under full switch load. We generate

traffic on all switch ports as follows. We saturate a single port using our packet generation machine and let that traffic travel through each port in the switch by connecting ports in a chain and forming a "snake" (a similar methodology was used in NetCache [38]). We reserve ports that are used for replication. We use the symmetric-2 topology. We saturate the switch with 130B packets, each updating the sketch. For a sketch with 64K entries per-row we measure 486 μsec window advancement latency, at a total packet rate of 1.8Gpps. For a sketch size of 1 entry per-row we measure 3.2 μsec window advancement latency at a total packet rate of 2.1Gpps.[3] In both extremes, we could not measure any impact on window advancement latency, which is expected as the switch logic is guaranteed to perform packet processing at a switch line rate.

SDW: Retransmissions and the ready phase. The ready phase ensures that the switches do not send updates after advancing their window before all others advanced to that same window. Without this guarantee, an update from the consequent window that arrives too early will be dropped, and later retransmitted after a timeout. We now show that this phase is essential to avoid retransmissions and maintain low latency when scaling to more switches.

We first run the protocol without the ready phase (Figure 5f, local-2 no ready) on two pipes on the same switch. The protocol runs in lockstep on both of the pipes, so we do not see any update retransmissions. However, with four pipes (symmetric-4 topology) there are many retransmission (not shown in the Figure). For example, we measure an average of 2934 update retranmissions in the first 10K window advancements across 100 runs. We observe a similar behavior in an asymmetric topology four pipes connected using the leaf-spine topology. Adding the ready phase completely eliminates such retransmissions and allows the system to progress effectively as fast as a two-pipes system, with stable latency guarantees.

SDW: Recovery. We measure the total recovery time of the protocol from the time pipes fail to the time the system makes progress, i.e. windows are advancing again. We run four pipes in the 4-symmetric topology that replicate a sketch and shut down random pipes. We disable the failed pipes' ports to other switches in a random order. We repeat this experiment 20 times for each data point, and vary sketch sizes and failure counts. We report the average recovery time.

Figure 6 shows that recovery time is dominated by the time it takes to synchronize the sketches of correct switches. Therefore, recovery time increases as sketch size increases, and decreases as the failure count increases. As expected, for the 3 pipe failures setup, only a single correct switch remains live, thus recovery time is independent from the sketch size.

As explained in §6.4, updates sent to live switches during the recovery are not lost but accumulated, so the recovery time minimization is a secondary goal. Nevertheless, recovery time can be further reduced by applying additional optimizations,

[3] 1-entry per-row requires lower replication load and frees certain resources affording higher packet rate.

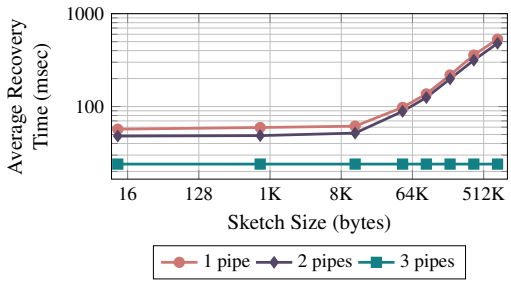


Figure 6: The impact of number of failures and sketch size on the total SDW recovery time.

e.g. parallelizing the currently serial controller-to-switch communication and batching requests, and by writing the logic using a more efficient programming language.

SDW: Scalability. We emulate a large replica group of switches running the SDW protocol by connecting together 32 Tofino model instances running in Docker containers. The switches are connected together via another switch that runs L3 forwarding. We verify that the protocol runs correctly and that there are no update retransmissions.

10 Related Work

In-switch NFs. Previous studies have shown that offloading NFs to programmable switches, such as load balancers [57] and DDoS detectors [45], enables very high performance. However, these projects were designed for a single switch. SwiSh aims to facilitate the deployment of these applications in a distributed fashion. RedPlane [42] enables switch state replication to servers for fault tolerance, but does not support state modification on multiple switches concurrently, as our work does.

In-switch acceleration. Previous works suggested in-switch acceleration for general-purpose applications such as key-value caches [38, 50], replicated key-value stores [37], query processing [24] and aggregations [68, 75]. SwiSh can be useful for such general-purpose applications too. For example, SwiSh could be used to implement the cache invalidation mechanism in DistCache. We note, however, that due to the general-purpose nature of these applications, some of them feature a complex state, and require strong semantics together with frequent updates, which SwiSh does not provide. Such requirements are less common in NFs; thus, we target SwiSh to facilitate the development of distributed NFs.

State management for NFs. State management and fault-tolerance for NFs on servers have been well studied [20, 64, 65, 71, 77]. However, these techniques are infeasible in the context of programmable switches. For example, FTMB [71] suggests a rollback-recovery technique for fault-tolerance in which packets are logged and replayed upon failures. However, due to the high processing rate of the switch, it is impractical to log

every packet to external storage or through the control-plane.

In-switch coordination. NetChain [37] and P4xos [16] implement coordination protocols running in the data plane to provide reliable storage as a network service. We apply data plane replication as an internal building block for NFs, a task for which it is well suited as the data-plane properties (e.g., limitations to ~100 byte objects) are better matched for replicating NF state registers than arbitrary applications.

Distributed network state. Managing distributed network state has been well studied. Onix [43] distributes network-wide state among multiple controllers. DIFANE [81] offloads forwarding decisions to authority switches to alleviate load on the controller and to reduce per-flow memory usage in network switches. Mahajan *et al.* [55] explore consistency semantics during network state updates. While previous works focus on control-plane managed state, SwiSh specifically targets replication of mutable state of data-plane programs.

Distributed network monitoring. Network-wide monitoring requires coordinated, distributed computation across switches [25, 26, 63]. Harrison *et al.* [25, 26] propose a distributed heavy-hitter detection algorithm that combines local counters with a centralized controller. SwiSh can be used to implement similar algorithms without a centralized controller, potentially providing faster response. Ripple [36] replicates state in data-plane for link-flooding defense but does not provide consistency guarantees. Ripple can be implemented using SwiSh. Distributed computation is also needed if the resources of a single switch are insufficient, e.g. Demianiuk *et al.* [18] partition state across switches for flow metric computation.

Relaxing consistency for availability. Many systems have traded consistency for increased availability and performance [4, 17, 21, 44, 62, 72, 79]. For example, TACT [79] aims to provide a middle-ground between strong and eventual consistency. However, TACT may block read and write operations to enforce consistency bounds which is unsustainable in the switch environment. Additionally, TACT maintains a single version of the data while SDW maintains multiple versions of the state and seamlessly switches to the up-to-date one as soon as the previous synchronization round is completed. Therefore, the protocol advances as fast as the network conditions allow while providing consistent snapshots to every replica. On the other hand, the combination of dynamic system behavior and consistent snapshots cannot be expressed using TACT's consistency metrics.

11 Conclusions

SwiSh offers a systematic approach to state sharing among programmable switches. We analyze the requirements of in-switch stateful NFs and implement three protocols for data-plane replication. We introduce a novel SDW protocol that achieves high update rate and low update latency, while providing strong consistency guarantees, which are particularly useful for implementing sketches. We show experimentally that data-plane is practical and fast, and achieves orders of magnitude higher performance than the traditional centralized controller designs. We believe that this work will pave the way for building distributed stateful NFs entirely in data-plane.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Dejan Kostic, for their insightful comments and constructive feedback. Lior Zeno was partially supported by the HPI-Technion Research School. We gratefully acknowledge support from Israel Science Foundation (grants 980/21 and 1027/18) and Technion Hiroshi Fujiwara Cyber Security Research Center.

References

[1] Yehuda Afek, Anat Bremler-Barr, Shir Landau Feibish, and Liron Schiff. Detecting heavy flows in the SDN match and action model. *Comput. Networks*, 136:1–12, 2018.

[2] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. Mergeable summaries. *ACM Transactions on Database Systems (TODS)*, 38(4):1–28, 2013.

[3] Amazon Web Services. AWS Shield. https://aws.amazon.com/shield.

[4] Mary Baker and John Ousterhout. Availability in the Sprite distributed file system. In *Proceedings of the 4th workshop on ACM SIGOPS European workshop*, pages 1–4, 1990.

[5] Ziv Bar-Yossef, TS Jayram, Ravi Kumar, D Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 1–10. Springer, 2002.

[6] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, Shir Landau Feibish, Danny Raz, and Minlan Yu. Routing oblivious measurement analytics. In *IFIP Networking*, pages 449–457, 2020.

[7] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6, 2014.

[8] 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. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[9] Broadcom. Trident 3. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series/.

[10] CAIDA. The CAIDA UCSD Anonymized Internet Traces - 2019. https://www.caida.org/catalog/datasets/passive_dataset.

[11] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking Control of the Enterprise. *ACM SIGCOMM computer communication review*, 37(4):1–12, 2007.

[12] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. Catching the Microburst Culprits with Snappy. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, page 22–28, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A. Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *ACM SIGCOMM Conference on Emerging Networking EXperiments and Technologies*, pages 15–29. ACM, 2019.

[14] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, April 2005.

[15] Graham Cormode and S. Muthu Muthukrishnan. Approximating data with the count-min sketch. *IEEE Software*, 29(1):64–69, 2012.

[16] 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.

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[18] V. Demianiuk, S. Gorinsky, S. Nikolenko, and K. Kogan. Robust Distributed Monitoring of Traffic Flows. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–11, 2019.

[19] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, 2016.

[20] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 163–174, New York, NY, USA, 2014. Association for Computing Machinery.

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[22] Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 373–382, 2011.

[23] 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 *Proceedings of ACM SIGCOMM 2009*, Barcelona, Spain, August 2009. ACM.

[24] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-Driven Streaming Network Telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 357–371, New York, NY, USA, 2018. Association for Computing Machinery.

[25] Rob Harrison, Shir Landau Feibish, Arpit Gupta, Ross Teixeira, S. Muthukrishnan, and Jennifer Rexford. Carpe elephants: Seize the global heavy hitters. In *Proceedings of the 2020 ACM SIGCOMM 2020 Workshop on Secure Programmable Network Infrastructure, SPIN@SIGCOMM 2020, Virtual Event, USA, August 14, 2020*, pages 15–21, 2020.

[26] Harrison, Rob and Cai, Qizhe and Gupta, Arpit and Rexford, Jennifer. Network-Wide Heavy Hitter Detection with Commodity Switches. In *Proceedings of the Symposium on SDN Research*, SOSR '18, New York, NY, USA, 2018. Association for Computing Machinery.

[27] Thomas A Henzinger, Christoph M Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 317–328, 2013.

[28] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[29] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. SketchVisor: Robust network measurement for software packet processing. In *ACM SIGCOMM*, pages 113–126, 2017.

[30] Qun Huang, Patrick P. C. Lee, and Yungang Bao. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In *ACM SIGCOMM*, pages 576–590, 2018.

[31] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, 2010.

[32] Intel. P4 Studio. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/p4-suite/p4-studio.html.

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

[34] Intel. Tofino Native Architecture. https://github.com/barefootnetworks/Open-Tofino.

[35] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. Qpipe: Quantiles sketch fully in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 285–291, 2019.

[36] Jiarong Xing and Wenqing Wu and Ang Chen. Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3865–3881. USENIX Association, August 2021.

[37] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, April 2018. USENIX Association.

[38] 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 *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 97–112, Boston, MA, March 2017. USENIX Association.

[40] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proceedings of the 2016 Symposium on SDN Research (SOSR '16)*, Santa Clara, CA, USA, March 2016. ACM.

[41] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 USENIX Winter Technical Conference*, San Francisco, CA, USA, January 1994. USENIX.

[42] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. RedPlane: Enabling Fault-Tolerant Stateful in-Switch Applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 223–244, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A Distributed Control Platform for Large-Scale Production Networks. In *OSDI*, volume 10, pages 1–6, 2010.

[44] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[45] A. C. Lapolli, J. Adilson Marques, and L. P. Gaspary. Offloading Real-time DDoS Attack Detection to Programmable Data Planes. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 19–27, 2019.

[46] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. Composing ordered sequential consistency. *Information Processing Letters*, 123:47–50, 2017.

[47] B. Lewis, M. Broadbent, and N. Race. P4ID: P4 Enhanced Intrusion Detection. In *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–4, 2019.

[48] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[49] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better NetFlow for data centers. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 311–324, Santa Clara, CA, March 2016.

[50] 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 *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 143–157, USA, 2019. USENIX Association.

[51] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *ACM SIGCOMM*, pages 334–350, 2019.

[52] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM*, pages 101–114, 2016.

[53] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, MinlanYu, and Vyas Sekar. Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switche. In *Proc. USENIX Security*, 2021.

[54] Zaoxing Liu, Samson Zhou, Ori Rottenstreich, Vladimir Braverman, and Jennifer Rexford. Memory-efficient performance monitoring on programmable switches with lean algorithms. In *Symposium on Algorithmic Principles of Computer Systems, APOCS*, pages 31–44, 2020.

[55] Mahajan, Ratul and Wattenhofer, Roger. On Consistent Updates in Software Defined Networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, New York, NY, USA, 2013. Association for Computing Machinery.

[56] McKeown, Nick and Anderson, Tom and Balakrishnan, Hari and Parulkar, Guru and Peterson, Larry and Rexford, Jennifer and Shenker, Scott and Turner, Jonathan. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[57] 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 *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[58] Microsoft Azure. Azure DDoS Protection. https://azure.microsoft.com/en-us/services/ddos-protection/.

[59] Robert HB Netzer and Jian Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and distributed Systems*, 6(2):165–169, 1995.

[60] NVIDIA. Spectrum. https://www.nvidia.com/en-us/networking/ethernet-switching/spectrum-sn4000/.

[61] 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. Ananta: Cloud Scale Load Balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 207–218, New York, NY, USA, 2013. Association for Computing Machinery.

[62] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible Update Propagation for Weakly Consistent Replication. *SIGOPS Oper. Syst. Rev.*, 31(5):288–301, oct 1997.

[63] Raghavan, Barath and Vishwanath, Kashi and Ramabhadran, Sriram and Yocum, Kenneth and Snoeren, Alex C. Cloud Control with Distributed Rate Limiting. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, page 337–348, New York, NY, USA, 2007. Association for Computing Machinery.

[64] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. Pico Replication: A High Availability Framework for Middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[65] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 227–240, Lombard, IL, 2013. USENIX.

[66] Arik Rinberg and Idit Keidar. Intermediate value linearizability: A quantitative correctness criterion. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 2:1–2:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[67] Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, Lee Rhodes, and Hadar Serviansky. Fast concurrent data sketches. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pages 117–129, 2020.

[68] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[69] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, page 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[70] S. Shenker and J. Wroclawski. RFC2215: General Characterization Parameters for Integrated Service Network Elements, 1997.

[71] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 227–240, New York, NY, USA, 2015. Association for Computing Machinery.

[72] Jeff Terrace and Michael J. Freedman. Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, page 11, USA, 2009. USENIX Association.

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

[74] The P4.org Architecture Working Group. $P4_{16}$ Portable Switch Architecture (PSA). https://p4.org/p4-spec/docs/PSA-v1.1.0.html.

[75] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating Database Queries with Switch Pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2407–2422, New York, NY, USA, 2020. Association for Computing Machinery.

[76] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation*, OSDI'04, page 7, USA, 2004. USENIX Association.

[77] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic Scaling of Stateful Network Functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 299–312, Renton, WA, April 2018. USENIX Association.

[78] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[79] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. OSDI'00, USA, 2000. USENIX Association.

[80] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, pages 29–42, April 2013.

[81] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. Scalable flow-based networking with DIFANE. *ACM SIGCOMM Computer Communication Review*, 40(4):351–362, 2010.

[82] Zeno, Lior and Ports, Dan R. K. and Nelson, Jacob and Silberstein, Mark. SwiShmem: Distributed Shared State Abstractions for Programmable Switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 160–167, New York, NY, USA, 2020. Association for Computing Machinery.

[83] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *the 27th Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

A Theoretical Analysis

The SDW protocol supports stream-order invariant data types like sketches. Variables of this type support three API functions: (1) UPDATE(v) – handling a single addition of element v, (2) QUERY() – returns a value based on the internal state, and MERGE(R') – merges the state of R' with that of the current variable. A requirement of any variable R fitting this model is that the QUERY result depends only on the set of elements that were ingested before it (either by an UPDATE or a MERGE), and not their order. We say that a query *reflects* an update, if the update altered the state before the query executed.

An execution of an algorithm renders a *history* H, which is a series of *invoke* and *response* events of the three API functions. In a *sequential history* each invocation is immediately followed by its response. The *sequential specification* $\mathcal{H}$ of a variable is its set of allowed sequential histories.

A *linearization* of a concurrent execution σ is a history $H \in \mathcal{H}$ such that after adding responses to some pending invocations and removing others, H and σ consist of the same invocations and responses and H preserves the order between non-overlapping operations [28]. If every concurrent execution has a linearization, we say that the variable is linearizable. For randomized variables we require a stronger property, called *strong linearizability*. The qualifier "strong" means that the linearization points are not determined post-facto, which is necessary in randomized variables [22].

A relaxed property of a variable is an extension of its sequential specification to allow for more behaviors. We adopt the notion of r-relaxed strong linearizability from [67], a variant of the relaxation defined by Henzinger et al. [27], brought here for completeness. Intuitively, an r-relaxed variable allows a query to return a result based on all but at most r updates that happened before it.

Definition A.1. A sequential history H is an r-relaxation of a sequential history H', if H is comprised of all but at most r of the invocations in H' and their responses, and each invocation in H is preceded by all but at most r of the invocation that precede the same invocation in H'. The r-relaxation of $\mathcal{H}$ is the set of histories that have r-relaxations in $\mathcal{H}$, denoted $\mathcal{H}^r$.

Our SDW protocol is described in §6.3, and its pseudo-code is presented in Algorithm 1. To prove that Algorithm 1 is r-relaxed strongly linearizable, we first prove a helper lemma:

Lemma 1. *Consider a history H arising from a concurrent execution of Algorithm 1, and some completed update $u \in H$ executed by p_i. Let w be the value of win during u. Update u is reflected by every query q on any p_j, in every window $w' \geq w+2$.*

Proof. Let H be a history arising from a concurrent execution of Algorithm 1, and let $u \in H$ be some completed update executed by p_i. Let w be the value of *win* during the update's execution on p_i.

Update u is added to $objs[w \mod 3]$ on Line 12. On Line 39, $objs[(w+2) \mod 3]$ is broadcast to all switches, specifically to some switch p_j (as p_i retains the update in the same place that is merges received variables, this holds for $j = i$).

The next time p_j advances on Line 35, it enters window $w' = w+2$. Note that the variable that was queried in the previous window $(w'-1)$ is the same variable that reflected u. This variable is the one queried in round w', therefore reflected in round $w' = w+2$.

We now prove by induction that in round $w'' = w'+k$, u is reflected by a query in round w'' on p_j. The base is for $k = 0$, and has been prove.

Assume the hypothesis holds for $w'+l$, we prove for $w'+l+1$. In round $w'+l$, u is reflected by $obj[(w'+l+1) \mod 3]$. On Line 37, p_j merges this variable into $obj[((w'+l+1)+1) \mod 3]$, which is the variable queried in this round.

As this induction is true for all $k \geq 0$, it holds for any $w'' \geq w'$, proving the lemma. □

The following corollary follows directly from Lemma 1:

Corollary 1.1. *Let H be a history arising from a concurrent execution of Algorithm 1, and let $q \in H$ be some query completed by p_i. Let w be the value of win during its execution. Query q reflects all updates occurring in any window $w' \leq w-2$.*

Note: A system where linearizability holds for sub histories including a single query is sometimes called *Ordered Sequential Consistency (OSC)* [46], this is commonly used in systems, e.g., ZooKeeper [31].

Finally, we define the *operation projection* of a history H and a set of operations O as the same history containing only invocations and responses of operations in O. We denote this $H|_O$ Using these formalisms we can prove the following theorem:

Theorem 2. *Consider a history H arising from a concurrent execution of Algorithm 1, and some query $q \in H$. Let U be the set of updates in H. The history of $H|_{U \cup \{q\}}$ is r-relaxed strongly linearizable.*

Proof. Let H be a history arising from a concurrent execution of Algorithm 1, let $q \in H$ be some query by p_i, and let U be the set of all updates in H. Denote $H|_{U \cup \{q\}}$ as H'. We show that H' is r-relaxed strongly linearizable with respect to $\mathcal{H}^r$, for $r = 2NB$. To prove this, we show the existing of two mappings, f and g, such that f maps operations in H' to *visibility points*, and g maps operations in H' to linearization points. Intuitively, visibility points are the time in the execution when an update is visible to a query, i.e., the query reflects the update. Bounding the number of preceding but not yet visible updates gives the relaxation.

We show that (1) $f(H') \in \mathcal{H}$, and (2) $g(H')$ is an r relaxation of $f(H')$. Together, this implies the theorem.

The visibility points ($f(H')$) are as follows:

- For the query, its visibility point is its return.
- For an update returning *false* at time t, its visibility point is t.
- For an update returning *true* at time t, let w be p_i's value of win at time t. The visibility point is the first time after t that p_i's value of win is $win+2$.

Note that in the latter case, the visibility point is after the update returns, so f does not preserve real-time order.

The linearization points ($g(H')$) are as follows:

- An update's linearization point is its return, either *true* or *false*.
- A query's linearization point is its return.

By definition, the linearization points as defined by $g(H')$ aren't decided post-facto – rather the linearization is a predetermined point in the execution.

Consider some update $u \in H'$ executed on some p_j that returns *true*. Let w be p_j's value of win during its execution. Let w' be p_i's value of win during q's execution. We show that if $w \leq w'-2$, then q observes u, and if $w > w'-2$, then q doesn't observe u.

From the definition of Algorithm 1, for any win_i on p_i and win_j on p_j, $|win_i - win_j| \leq 1$.

If $w = w'-2$, then when p_j added u to its local buffers, it did so to $obj[w \mod 3]$. As $|win_i - win_j| \leq 1$, p_j advanced at least 1 window from w. When it did so, it sent $obj[w \mod 3]$ to p_i. In window $w'-1$, p_i merges the update into $obj[w'+1 \mod 3]$. In window w' this same variable is queried, thus q observes u. If $w \leq w'-3$, then the update is merged into some index of the variables array, and is copied over until it is reflected in all 3 of them, and specifically reflected in $obj[w'+1 \mod 3]$ in window w'.

If $w \geq w'-1$, then when p_j added u into its local buffer it did so to $obj[w \mod 3]$. This update is sent to p_i only in window $w+1$, and therefore isn't reflected in $obj[w'+1 \mod 3]$ in window w'.

Therefore, q reflects all updates that return true that happened during any window $w \leq w'-2$. As there are at most B updates that return true in any window, q reflects all but at most $2NB$ updates that precede it in H. Therefore, $g(H')$ is an $2NB$-relaxation of $f(H')$.

As the query returns a value based on the updates that happened before it, and each access to the process local state is down sequentially, q returns a value that reflects all successful updates that happen before it in $f(H')$. Therefore, $f(H') \in \mathcal{H}$. □

Intuitively, every query returns a value reflecting a substream of its preceding and concurrent updates, consisting of all but at most r successful ones. The upper bound r on the number of "missing" updates is of vast importance, without it the drift between one switch and another can grow in an unbounded fashion. For example, consider a counter distributed among two switches running an eventually synchronous algorithm. One switch can increment the counter an arbitrarily large number of times, while the other returns 0 on every query – the promise of eventual synchrony is too weak.

Theorem 2 ensures that every history consisting of a single query and all updates is r-relaxed strongly linearizable, which in many cases preserves some relaxation of the error bounds. For example, Rinberg et al. [67] show that, under a weak adversary, a K-Minimum Value (KMV) θ sketch [5] has an error of at most twice that of the sequential one. Another example is a relaxed Quantiles sketch [2], which has an additive error of $r/n - (r\epsilon)/n$ with some tuning parameter ϵ, where r is the relaxation and n is the stream size. Thus, the impact of the relaxation diminishes as the stream size grows.

Algorithm 1: Algorithm running on switch p_i.

```
1  initialization:
2  win ← 0
3  count ← 0
4  objs ← [obj.init(), obj.init(), o.init()]
5  buf ← {}
6  rcvs ← {}
7  acks ← {}
8  Function Update(v):
9      if count == B then
                                                          // Write variable is full
10         return false
11     else
12         objs [win  mod 3] .update(v)                   // Add to the write variable
13         count ← count + 1
14         return true
15
16 Function Query():
17     return objs [(win + 1)  mod 3] .query()            // Serve query from read variable
18
19 on receive "(o′, w′)" from p_j:
                                                          // Sync
20     if w′ > win then
21         buf ← buf ∪ {(o′, w′)}                         // Buffer messages from future windows
22     else
23         rcvs ← rcvs ∪ {j}
24         objs [(win + 2)  mod 3] .merge(o′)             // Merge into sync buffer
25         send "ack" to p_j
26         check_done()
27
28 on receive "ack" from p_j:
29     acks ← acks ∪ {j}
30     check_done()
31
32 Function check_done():
33     if |rcvs| == n && |acks| == n then
34         count ← 0
35         win ← win + 1                                  // Rotate right
36         o′ ← objs [win  mod 3]
37         objs [(win + 1)  mod 3] .merge(o′)             // Add the updates from window w to the current state
38         objs [win  mod 3] ← o.init()                   // Clear write variable
39         broadcast "(objs [(win + 2)  mod 3] , win)"    // Send sync message
40         rcvs ← {i}
41         acks ← {i}
42         forall (o′, w′) in buf do
                                                          // Handle buffered messages
43             rcvs ← rcvs ∪ {j}
44             objs [(win + 2)  mod 3] .merge(o′)
45             send "ack" to p_j
46         buf ← {}
```