SwiShmem: Distributed Shared State Abstractions for Programmable Switches

Lior Zeno
Technion
liorz@campus.technion.ac.il

Dan R. K. Ports
Microsoft Research
dan@drkp.net

Jacob Nelson
Microsoft Research
Jacob.Nelson@microsoft.com

Mark Silberstein
Technion
mark@ee.technion.ac.il

ABSTRACT

Programmable switches provide an appealing platform for running network functions (NFs), such as NATs, firewalls, and DDoS detectors, entirely in data plane, at staggering multi-Tbps processing rates. However, to be used in real deployments with a complex multi-switch topology, one NF instance must be deployed on each switch, which together act as a single logical NF. This requirement poses significant challenges in particular for stateful NFs, due to the need to manage *distributed shared NF state* among the switches. While considered a solved problem in classical distributed systems, data-plane state sharing requires addressing several unique challenges: high data rate, limited switch memory, and packet loss.

We present the design of *SwiShmem*, the first *distributed shared state management* layer for data-plane P4 programs, which facilitates the implementation of stateful distributed NFs on programmable switches. We first analyze the access patterns and consistency requirements of popular NFs that lend themselves for in-switch execution, and then discuss the design and implementation options while highlighting open research questions.

CCS CONCEPTS

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

KEYWORDS

Programmable switches, Programmable networks, Distributed state management, Network function virtualization

ACM Reference Format:
Lior Zeno, Dan R. K. Ports, Jacob Nelson, and Mark Silberstein. 2020. SwiShmem: Distributed Shared State Abstractions for Programmable Switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20), November 4–6, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3422604.3425946

HotNets '20, November 4–6, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8145-1/20/11…$15.00
https://doi.org/10.1145/3422604.3425946

1 INTRODUCTION

In recent years, programmable data-plane switches [3, 33] 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 implement the advanced network functions (NFs) that power modern cloud networks, such as NATs, load balancers [19, 32], and DDoS detectors [25]. These in-switch 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, potentially heralding a massive efficiency improvement.

Unfortunately, a key challenge remains largely unaddressed: while prior work shows the potential of programmable switches to execute NFs efficiently, they primarily do so on *a single switch*. Realistic data center deployments require NFs to be distributed over multiple switches: to handle higher throughput than a single switch can support, to tolerate switch failure, and to process traffic that passes through many network paths. A firewall monitoring incoming data center traffic, for example, cannot be implemented by routing all traffic through a single switch. Network functions must therefore be implemented in a distributed manner.

Building distributed NFs for programmable switches is challenging because most of today's NFs are *stateful* and demand consistency and reliability. If a load balancer assigns a connection to a particular destination, subsequent packets for that connection must be routed accordingly – even if they arrive at different switches or if the original switch fails.

Distributed state management is, in general, a hard problem, and it becomes even harder in the context of programmable switches. In the "traditional" software-based NF realm, several methods have been proposed to address the challenges of distributed state. These include remote access to centralized state storage [17] and distributed object abstractions [46], along with fault tolerance mechanisms [37, 42]. Unfortunately, few of these ideas transfer directly to the programmable switch environment, which faces three daunting new challenges: (1) it must be able to handle packets at line rate, greatly limiting available per-packet computation; (2) it faces even stricter storage requirements, with only ∼10 MB state available from the data-plane; and (3) it lacks mechanisms for durable storage or reliable communication (even TCP is unavailable).

This paper argues that generic abstractions for state management can greatly simplify the design of distributed in-switch network function applications. Our goal is to provide a "one big switch" abstraction for stateful NFs that enables developers to write a program

that appears to run on a single reliable switch, even though it actually executes across multiple switches. We believe that providing general abstractions, rather than ad-hoc application-specific solutions, is essential to the adoption of this technology in production environments.

We describe the design of such a generic distributed shared state mechanism, SwiShmem. Inspired by distributed shared memory abstractions for distributed systems [20, 28], SwiShmem provides replicated shared registers in a way that is tailored to the needs of NFs. It supports strong consistency for read-intensive registers and eventual consistency for frequently-written registers, two modes which we argue capture the needs of most existing NFs that lend themselves to efficient in-switch implementation.

SwiShmem introduces new replication protocols which are optimized for the programmable switch environment. These protocols follow two main design principles: (1) **minimizing the necessary buffer space** due to the scarcity of switch memory, (2) exploiting **Tbps network bandwidth** available for inter-switch communication. Thus, while they largely inherit the basic mechanisms employed in traditional distributed systems, such as chain replication, the protocols explicitly trade ample network bandwidth for in-switch memory space.

In summary, this work makes the following contributions:

- Analysis of the memory consistency requirements and access patterns of common NFs suitable for in-switch execution
- New distributed protocols that provide SwiShmem, an in-switch distributed shared register abstraction to facilitate shared state management across multiple switches
- The implementation considerations for SwiShmem on real programmable switches

2 BACKGROUND: PISA SWITCHES

The protocol-independent switch architecture (PISA) [2] is the standard for programmable data-plane switches. PISA 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 ($\sim$ 10 MB) switch memory is split between pipeline stages.

PISA compliant devices are programmed using the P4 language. P4 defines a set of high-level objects that consume switch memory: tables, registers, meters and counters. While registers, meters, and counters can be modified directly from the data-plane, tables require control-plane to perform update. A data-plane program processes packets which can be sent to remote destinations, to the control-plane processor on the switch, or to the switch itself (called *recirculation*).

Switches process packets atomically: if a packet generates multiple local writes to different locations, these updates are atomic in the sense that the next processed packet will not see an intermediate view on the state. This property allows implementing complex distributed protocols with concurrent state updates without locks (for example, chain replication with pending writes (§6.1)).

3 MOTIVATION

3.1 The Case for Programmable Switches as Network Function Processors

The modern data center network incorporates a diverse array of network functions beyond simple packet forwarding. Features like network address translation (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, for example, a stateful load balancer, which aims to uniformly route TCP connections among multiple destination hosts. It receives incoming connections, assigns them to a destination server, and forwards encapsulated traffic to that server. Because it is essential for subsequent packets in the same TCP connection to be routed to the same server (a property dubbed per-connection consistency), the load balancer must track the connection-to-server mapping. Load balancers like this are in widespread use at major cloud providers [7, 34], and handle a significant fraction of a data center's incoming traffic. Implemented on commodity servers, they require large clusters to support their massive workload.

Programmable data-plane switches offer an appealing alternative to commodity servers for implementing network functions at lower cost. Researchers have shown that new programmable pipelines can be used to implement many types of network functions. For data center operators, the benefit is clear: 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 [7], 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 only slightly higher than a single server, but can process *several hundred times* as many packets.

3.2 Distributed Switch Deployments

Although prior research has focused on showing that NF functionality can be implemented on a single switch [25, 32], realistic data center deployments universally require multiple switches. We see two possible deployment scenarios. The NF processing can be placed in switches in the network fabric. In order to capture all traffic, the load balancer application would need to run on all possible paths, e.g., by being deployed to every core switch or every aggregation switch. Alternatively, a dedicated cluster of switches (perhaps located near the ingress point) could be used to serve purely as NF accelerators. Both are inherently distributed deployments: they require multiple switches in order to (1) scale out, (2) tolerate individual switch failures, and (3) capture traffic across multiple paths.

The challenge of a distributed NF deployment comes from the need to manage the state shared among the NF instances. Processing a packet at one switch may require reading or updating state that is 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.

Shared state is a necessity for distributed stateful processing. In some cases, it may suffice to shard state between switches – for

example, to store the load balancer's connection mapping only on the switch that assigned it, on the assumption that future packets for that flow will be processed by the same switch. This approach, however, does not work for NFs that require global state, e.g., to add a per-client rate limiter to the load balancer. It also falls short if a flow is routed through a different switch, something that may occur in various failure scenarios – or in the normal case, if recent proposals for adaptive routing [10, 14] or multi-path TCP [8, 36] are adopted.

To support our goal of a "one big switch" abstraction, SwiShmem provides a shared state mechanism capable of supporting global state: any object can be read or written from any switch. SwiShmem transparently replicates state updates to other switches for fault tolerance and remote access.

3.3 The Case for Data-plane Replication

Control-plane mechanisms are the common practice of replicating the switch state [1, 4, 22, 31]. 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 [22, 48]. SwiShmem proposes instead to replicate the switch data plane.

Data plane replication makes it possible to support applications that read or modify stateful variables on every packet. This is the new capability of programmable data-plane switches that makes it possible to implement more sophisticated network functions than traditional control-plane SDN. As we will see in §4, these applications use state in diverse ways. Some are read-mostly; others update state on every packet. Some require strong consistency between switches to avoid exposing inconsistent states to applications (e.g., a network sequencer for accelerating storage systems [27]); others can tolerate weak consistency (e.g., measurement or monitoring applications that already provide approximate results [25, 47]). Managing this state in a programmable data-plane switch requires a new approach: replication protocols that run in the control plane cannot operate at this rate, so a control-plane solution would cause significant gaps between replicas, potentially affecting normal application behavior and the system availability. SwiShmem 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 because it 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 acknowledgements entirely in the data-plane for our strongly-consistent replication protocol (§6.1).

3.4 Challenges of In-Switch Replication

Making switch state updates visible and fault tolerant is the purview of a replication protocol. This is one of the classic problems in server-based distributed systems, with many such protocols having been extensively studied in a general setting [24, 43, 45], as well as specifically targeting distributed execution of network functions [17, 42]. Applying them to the programmable switch domain presents new challenges due to the high processing throughput but low memory capacity of the switches and unreliable connection between them.

To understand these challenges, consider Chain Replication [45]. This model organizes replicas into a chain. Write requests are processed first by the head of the chain, which sequences the requests, and then are propagated along the chain. Once the write request propagates to the tail, an acknowledgement is generated and sent to the client. Reads are sent to the tail, which always returns the latest completed write. Thus, it provides linearizability [13]. While this protocol works well for servers, several limitations of programmable switches make direct implementation problematic:

Unreliable communication. To ensure that updates are applied to all replicas in the same order, chain replication relies on TCP to provide a reliable, in-order communication channel. Programmable switches cannot run TCP in the data-plane.

Blocking requests. Strongly consistent replication protocols require that an update is recorded by other replicas before its results can be externalized. For NFs, this means that the output packet must be buffered until the write is acknowledged by other switches. Programmable switches lack the memory to buffer packets for any significant amount of time.

Memory usage. Replication protocols often include optimizations or design choices that assume large amounts of memory are available, something that is very much not the case on a programmable switch. For example, each server in chain replication tracks the set of outstanding write operations in order to optimize failure recovery: if the chain is broken, only these operations need to be replayed. On a switch, using precious memory for this purpose, if even possible, is unlikely to be the right tradeoff.

4 APPLICATIONS

Global shared registers with linearizable consistency guarantees are a strong programming abstraction, but they come with a heavy performance cost. Strongly consistent replication protocols require blocking operations to communicate with remote replicas, a proposition that is costly in general and, as discussed previously, particularly difficult on switches. SwiShmem provides different levels of consistency targeted at the needs of NF applications.

To support this design, we study the access patterns and consistency requirements of NF applications that have been built on PISA applications. Table 1 summarizes the results. We observe two common patterns: read-intensive workloads that can tolerate expensive writes, and write-intensive workloads that can tolerate inconsistencies. Below, we describe several in-switch applications and how they use state.

4.1 Read-intensive NFs

Network Address Translators (NATs) share the connection table among the NF instances. The table is queried on every packet, but only updated when a new connection is opened; table rows require strong consistency, otherwise leading to broken client connections in case of multi-path routing or switch failure. NATs generally also manage a pool that tracks unassigned ports; however, different port ranges can be assigned to different switches to avoid sharing this state.

	Application	State	Write frequency	Read frequency	Consistency
Read-intensive	NAT	Translation table	New connection	Every packet	Strong
	Firewall	Connection states table	New connection	Every packet	Strong
	Intrusion prevention system (IPS)	Signatures	Low	Every packet	Weak
	L4 load-balancer	Connection-to-DIP mapping	New connection	Every packet	Strong
Write-intensive	DDoS detection	Sketch	Every packet	Every packet	Weak
	Rate limiter	Per-user meter	Every packet	Every window	Weak

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

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.

Intrusion Prevention Systems (IPS) [26] monitor traffic by continuously computing packet signatures and matching against known suspicious signatures. In case of too many matches, traffic is dropped to prevent the intrusion. This application can tolerate some transient inconsistencies: it is acceptable for a few additional malicious packets to go through immediately after signatures are updated.

L4 load balancers [32], as we have already discussed, assign incoming connections to a particular destination IP, then forward subsequent packets to the appropriate destination IP. Per-connection consistency (PCC) requires that once an IP is assigned to a connection, it does not change, implying a need for strong consistency of application state.

Observation 1. Although these workloads require strong consistency, they update state infrequently, making a costly replication protocol more tolerable. Indeed, most of these examples use switch data structures that must be modified *through the control plane*. We leverage this observation when designing the replication protocol for this class of NFs.

4.2 Write-intensive NFs

DDoS detection [25] requires tracking the frequency of source and destination IPs using approximate sketch data structures. The sketches are updated and read on every packet, triggering an alarm when the analysis of the IP frequencies raises suspicion of the attack. Approximate sketches have been shown to behave correctly under eventual consistency [39].

Rate limiters monitor and 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. Periodically, the meters are read to identify users exceeding their bandwidth limit and enforce restrictions. This application can tolerate some transient inconsistencies: it is acceptable for a few additional packets to go through immediately after the user reaches the bandwidth limit.

Observation 2. The aforementioned write-intensive workloads can tolerate weakly consistent data, permitting more efficient replication protocols. A particularly common use case is a shared *counter*, which is especially well-suited for eventually consistent protocols because it has *commutative* increment operations.

5 SWISHMEM ABSTRACTIONS

SwiShmem provides the abstraction of shared registers 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. They are able to communicate via the network, and we assume a standard failure model: packets can be dropped, and links and switches may fail.

Data model. The basic unit of shared state is a *register*. We begin by assuming that each register is replicated on every switch (we discuss extensions for partitioning and migrating state in §7). SwiShmem registers are read and modified through a replication protocol; a compiler could be used to translate regular P4 register accesses into SwiShmem operations. SwiShmem supports three types of registers which have different semantics and are accessed through different protocols:

(1) *Strong Read Optimized (SRO)* variables provide strong consistency (linearizability)
(2) *Eventual Read Optimized (ERO)* variables provide slightly weaker consistency in exchange for lower latency
(3) *Eventual Write Optimized (EWO)* variables have low cost for both reads and writes, but provide only eventual consistency

6 IN-SWITCH REPLICATION PROTOCOL

We describe here the protocols for each of the three types of variables. Initially, we assume switches do not fail; we relax this assumption in §6.3.

6.1 Read-Optimized Protocols (SRO & ERO)

SRO registers provide linearizability for in-switch applications. The SRO protocol is based on chain replication [45], adapted to an in-switch implementation. Switches are ordered to form a chain. This protocol supports both registers as well as state that can be read (e.g., counters) or written (e.g., tables) only by the control plane.

Writes are handled by processing an input packet P to determine the output packet P' and associated write set Q. Rather than being applied and sent immediately, both P' and Q are forwarded to the control plane, which buffers P' until the write is completed. It then sends a write request to the switch at the head of the chain, and retries it if a timely response is not received.

If Q includes only data-plane accessible registers, in all other switches, the update protocol is processed entirely in the data plane. Otherwise, the update protocol is processed by the control-plane of each switch in the chain. The head assigns a per-key sequence number to the write request to ensure consistent order of concurrent

writes, applies the update locally, and propagates it down the chain. Each switch in the chain applies updates only if they are in order, and forwards them to the next switch in the chain. It also sets a *pending bit* associated with the register, indicating that a write is in progress. Once the write request arrives at the tail switch, the tail sends an acknowledgment to the writer, which can release its output packet, and to the other switches, which can clear their pending bit.

Reads are processed using the local copy of the register, and incur no overhead, as long as the associated pending bit is not set.[1] Otherwise, the input packet P is forwarded to the tail of the chain, and processed there; this guarantees that the latest committed version of the data is used, and also avoids the need to buffer packets.

SRO provides per-register linearizability [13], 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, minimizing the additional cost.

Eventual Read Optimized (ERO). ERO is a variation of SRO that provides eventual consistency by always performing reads locally [43], rather than forwarding them to the tail when there are concurrent writes. This guarantees bounded read latency, and also saves space by eliminating the need for pending bits. Otherwise, the protocol is identical to SRO.

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 registers. This is acceptable for many write-intensive applications, as discussed in §4. These variables support eventual consistency for arbitrary data, and can provide stronger guarantees for certain specific types like counters.

For EWO state, reads are always 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 the other switches. This faces two challenges: (C1) updates may get lost; (C2) the receiving switch must merge the state update with its current state.

Periodic synchronization. Unlike SRO, we cannot 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 register from the data plane. This design choice avoids expensive buffering and re-transmission logic in data-plane. It is a protocol well-suited for programmable switches, as it takes advantage of the large available bandwidth and the small total switch state. Together, these make regular full synchronizations feasible. For example, even if the switches synchronize 10 MB (about the full memory size) every 1 ms, the total bandwidth consumed by the synchronization would constitute $\frac{10MB}{1ms \times 5Tbps} \sim 1\%$ of the total switch bandwidth.

Merging. C2 is harder to solve; the correct way to merge writes is application dependent. SwiShmem offers a default mode based on last-writer-wins, and a special case for counters (and potentially other data types) with stronger convergence semantics.

A generic answer to merging state is to assign an order to updates and apply a last-writer-wins (LWW) policy. In LWW, each register is associated with a version number. The merge function accepts an update from another switch only for the version numbers 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. The timestamp can be a Lamport clock [23] or a real-time clock, which can be synchronized among the switches down to tens of nanoseconds [18]. LWW provides eventual consistency, but until it converges there may be inconsistent behavior as updates propagate through the system.

In some cases, it is possible to merge updates more systematically. These are discussed extensively in the literature of Conflict-Free Replicated Data Types (CRDTs), which offer *strong eventual consistency* and *monotonicity* [41], which avoids counter-intuitive scenarios such as a 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 simply takes the larger of the local and received value for each element. Further extensions support decrement operations [41]. While many other CRDTs have been designed (e.g., sets and their variants), whether they are useful for in-switch NF applications or implementable in a switch data plane is an open question.

6.3 Handling failures

We now consider fail-stop switch failures. We assume that a central controller can detect which switches have failed. We consider two phases: (a) switch failover and (b) recovery.

SRO. When a switch fails, the chain becomes partitioned. Thus, writes cannot be processed. First, we regain connectivity by reprogramming the routing of the failed switch neighbors. In-flight writes that were dropped due to the failure, will eventually timeout and re-sent by the control-plane of the writer switch. This is the same as in the standard chain replication protocol.

To recover, we add a new switch to the end of the chain. The new switch starts to process writes, but does not replace the tail. Some control plane support is needed for the initial data transfer. The control plane on one of the switches takes a snapshot of its shared state, and then uses it to resend the write requests for each value through the normal data plane protocol. These writes contain the sequence number at the time of the snapshot, to prevent overwriting new values with old ones. Once the new switch has acknowledged all writes, it has the latest complete state, and can replace the tail in processing reads.

[1] This read-only optimization is derived from CRAQ [43].

[2] One might ask why this requirement is needed, when NetChain [15] implements chain replication entirely in the data plane. The difference is that NetChain is a service exposed to clients, which are responsible for retrying operations. Our switches are effectively the "clients" and need to be able to buffer output packets and retry requests, something infeasible on the data plane.

EWO. The synchronization protocol is inherently robust to switch and link failures. If a switch fails while broadcasting its updates, any switch that did receive the update can then synchronize the other switches, which will produce the same result. Thus, other than removing the failed switch from the multicast group, no explicit failover protocol is needed.

Recovery is equally simple: we add the new switch to the system by adding it to the multicast group, and wait for the first periodic synchronization mechanism to complete.

7 IMPLEMENTATION SKETCH

Implementing SRO. For SRO, we have to deal with state on both the control plane and the data plane. We have two main areas of state overhead. The first is in managing the chain during updates. Each switch has a register array with a sequence number and an in-progress bit per entry. Since this is relatively small, current programmable switches could support over a million entries; however, since these state elements only protect other state updates, multiple keys can share the same sequence number and in-progress bit, reducing state requirements further.

The second state overhead is buffering write packets during state updates to provide strong consistency, but our design uses the control plane to implement that which has ample DRAM capacity.

Note that for mutating packets, the *output* packet is not sent until the writes are acknowledged by the chain. Then, the packet is injected back to the data plane and forwarded to its destination (possibly via the switch at which it originally arrived).

For routing, each switch may store the IP addresses of the head switch as well as its successor and predecessor. Alternatively, write request packet headers may incorporate an IP list of the chain nodes.

Implementing EWO. For EWO, the state is stored only on the data plane in pairs of registers. Each switch contains one register array for each switch in the replica group; each register array stores a version number and a value. Due to the atomicity of packet processing in the switch (§2), the replication protocol can update both the version number and the value atomically. Current programmable switches can support large replica groups with a few tens of thousands of entries, or small replica groups with over a million entries.

For mutating packets that write to counters, the switch updates the version numbers and values in the register array for the local replica, updates the packet and forwards it towards its destination. Then, it uses egress mirroring and the multicast engine to broadcast small write update packets containing only this switch's new version numbers and values to the other switches in the replica group.

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

Bandwidth overhead. Generating write requests for replication consumes available bandwidth which may be substantial especially in write-intensive workloads. Batching write requests may alleviate this issue at the expense of reduced availability and consistency.

8 RELATED WORK

In-switch network functions. Previous studies have shown that offloading NFs to programmable switches, such as load-balancers [32] and DDoS detectors [25], enables very high performance. However, these projects were designed for a single switch. SwiShmem aims to facilitate the deployment of these applications in a distributed fashion.

In-switch acceleration. Previous works suggested in-switch acceleration for general-purpose applications such as key-value caches [16, 29], replicated key-value stores [15], query processing [11] and aggregations [40, 44]. SwiShmem can be useful for such general-purpose applications too. For example, SwiShmem 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 SwiShmem does not provide. Such requirements are less common in NFs; thus, we target SwiShmem to facilitate the development of distributed NFs.

State management for NFs. State management and fault-tolerance for NFs on servers has been well studied [9, 37, 38, 42, 46]. However, these techniques are infeasible in the context of programmable switches. As an example, FTMB [42] suggests a rollback-recovery technique for fault tolerance. Packets are logged and replayed in case of a failure. However, due to the high processing throughput of the switch, it is impractical to log every packet to external storage or through the control plane.

In-switch coordination. NetChain [15] and P4xos [5] implement coordination protocols running in the data plane to provide reliable storage as a network service. Our work shares similar ideas, and we believe that replicated storage as an internal building block for NFs rather than an external service may be an even more compelling application. Their properties (e.g., limitations to $\sim$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 [22] distributes network-wide state among multiple controllers. DIFANE [48] 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.* [30] explore consistency semantics during network state updates. While previous works focus on control-plane managed state, SwiShmem specifically targets replication of mutable state of data-plane programs.

Distributed network monitoring. Monitoring of network-wide properties requires coordinated, distributed computation across switches [12, 35]. Harrison *et al.* [12] propose a distributed heavy hitters detection algorithm that minimizes the communication overheads between the switches and the controller. Switches maintain local counters and use them to trigger updates to a centralized controller. SwiShmem can be used to implement similar algorithms while eliminating the need for a centralized controller, thus potentially providing faster response. Furthermore, distributed computation are needed if the resources of a single switch are insufficient. Demianiuk *et al.* [6] explore distributed flow metric computation on multiple switches while overcoming network noise. It uses an adhoc protocol in which

the bits of each counter are effectively split between switches to achieve larger capacity, and the system efficiently handles overflows. SwiShmem focuses on a different application domain where the state is replicated across switches rather than decomposed.

9 DISCUSSION & CONCLUSIONS

SwiShmem offers a systematic approach to state sharing among programmable switches. While still in the initial stages of implementation, we believe that most of the ideas outlined here are feasible. However, beyond the technical viability, it opens up a range of novel design opportunities for applying general distributed system principles and protocols while exploiting different trade-offs and optimizing them to memory-constrained switch programs.

One current limitation of SwiShmem is the need for control plane involvement to achieve strongly consistent writes. While in our experience applications that require frequent writes and strong consistency are rare among traditional NFs, some new in-network applications like sequencers [27] have such data. A way to implement buffering and retransmission in the data plane – perhaps achievable with creative use of existing switch features – would enable this support.

In the current proposal, we have assumed that all state can be stored on all switches. This allows the system to scale out in terms of throughput, but not in terms of state. If all state is indeed used by all switches, not much can be done about this. If there is locality, i.e., some state is normally used only by a subset of switches, it would not need to be replicated to all switches. One way to manage this, which we are currently exploring, is to use a central controller that acts as a directory service (in the vein of cache coherence protocols [28]), tracking which switches replicate which state, and migrating data as needed.

We envision that SwiShmem is only the first step toward a broader "one-switch abstraction" which aims to enable automatic transformation of a single-switch program into a distributed one. This goal is not far fetched. For example, several prior works [21] already showed how to find the shared state to elastically scale out the single virtual NFs running on commodity servers. We hope to apply similar ideas for multi-switch scaling of P4-based NFs.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd Ben Leong for their insightful comments and constructive feedback. Lior Zeno was partially supported by the HPI-Technion Research School. This research was supported by Israel Science Foundation grant 1027/18.

REFERENCES

[1] Berde, Pankaj and Gerola, Matteo and Hart, Jonathan and Higuchi, Yuta and Kobayashi, Masayoshi and Koide, Toshio and Lantz, Bob and O'Connor, Brian and Radoslavov, Pavlin and Snow, William and Parulkar, Guru. 2014. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking* (Chicago, Illinois, USA) *(HotSDN '14)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2620728.2620744

[2] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (Hong Kong, China) *(SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 99–110. https://doi.org/10.1145/2486001.2486011

[3] Broadcom. 2020. *Trident 3*. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series/.

[4] Casado, Martin and Freedman, Michael J. and Pettit, Justin and Luo, Jianying and McKeown, Nick and Shenker, Scott. 2007. Ethane: Taking Control of the Enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Kyoto, Japan) *(SIGCOMM '07)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1282380.1282382

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

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

[7] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA, 523–535. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud

[8] Alan Ford, Costin Raiciu, Mark J. Handley, and Olivier Bonaventure. 2013. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824. https://doi.org/10.17487/RFC6824

[9] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 163–174. https://doi.org/10.1145/2619239.2626313

[10] Patrick Geoffray and Torsten Hoefler. 2008. Adaptive routing strategies for modern high performance networks. In *Proceedings of the 16th IEEE Symposium on High Performance Interconnects*. Washington, DC, USA.

[11] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-Driven Streaming Network Telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 357–371. https://doi.org/10.1145/3230543.3230555

[12] Harrison, Rob and Cai, Qizhe and Gupta, Arpit and Rexford, Jennifer. 2018. Network-Wide Heavy Hitter Detection with Commodity Switches. In *Proceedings of the Symposium on SDN Research* (Los Angeles, CA, USA) *(SOSR '18)*. Association for Computing Machinery, New York, NY, USA, Article 8, 7 pages. https://doi.org/10.1145/3185467.3185476

[13] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[14] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. 2020. Contra: A Programmable System for Performance-aware Routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 701–721. https://www.usenix.org/conference/nsdi20/presentation/hsu

[15] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 35–49. https://www.usenix.org/conference/nsdi18/presentation/jin

[16] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 121–136. https://doi.org/10.1145/3132747.3132764

[17] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 97–112. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan

[18] Pravein Govindan Kannan, Raj Joshi, and Mun Choon Chan. 2019. Precise Time-Synchronization in the Data-Plane Using Programmable Switching ASICs. In *Proceedings of the 2019 ACM Symposium on SDN Research* (San Jose, CA, USA) *(SOSR '19)*. Association for Computing Machinery, New York, NY, USA, 8–20. https://doi.org/10.1145/3314148.3314353

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

[20] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. 1994. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating

Systems. In *Proceedings of the 1994 USENIX Winter Technical Conference*. USENIX, San Francisco, CA, USA.

[21] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. 2016. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 239–253. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/khalid

[22] Koponen, Teemu and Casado, Martin and Gude, Natasha and Stribling, Jeremy and Poutievski, Leon and Zhu, Min and Ramanathan, Rajiv and Iwata, Yuichiro and Inoue, Hiroaki and Hama, Takayuki and Shenker, Scott. 2010. Onix: A Distributed Control Platform for Large-Scale Production Networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 351–364.

[23] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[24] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems* 16, 2 (May 1998), 133–169.

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

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

[27] Jialin Li, Ellis Michael, Adriana Szekeres, Naveen Kr. Sharma, and Dan R. K. Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX, Savannah, GA, USA.

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

[29] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies* (Boston, MA, USA) *(FAST'19)*. USENIX Association, USA, 143–157.

[30] Mahajan, Ratul and Wattenhofer, Roger. 2013. On Consistent Updates in Software Defined Networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (College Park, Maryland) *(HotNets-XII)*. Association for Computing Machinery, New York, NY, USA, Article 20, 7 pages. https://doi.org/10.1145/2535771.2535791

[31] McKeown, Nick and Anderson, Tom and Balakrishnan, Hari and Parulkar, Guru and Peterson, Larry and Rexford, Jennifer and Shenker, Scott and Turner, Jonathan. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (March 2008), 69–74. https://doi.org/10.1145/1355734.1355746

[32] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/3098822.3098824

[33] Barefoot Networks. 2020. *Tofino*. https://barefootnetworks.com/products/brief-tofino/.

[34] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: Cloud Scale Load Balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (Hong Kong, China) *(SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 207–218. https://doi.org/10.1145/2486001.2486026

[35] Raghavan, Barath and Vishwanath, Kashi and Ramabhadran, Sriram and Yocum, Kenneth and Snoeren, Alex C. 2007. Cloud Control with Distributed Rate Limiting. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Kyoto, Japan) *(SIGCOMM '07)*. Association for Computing Machinery, New York, NY, USA, 337–348. https://doi.org/10.1145/1282380.1282419

[36] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. 2011. Improving Datacenter Performance and Robustness with Multipath TCP. In *Proceedings of ACM SIGCOMM 2011*. ACM, Toronto, ON, Canada.

[37] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. 2013. Pico Replication: A High Availability Framework for Middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, California) *(SOCC '13)*. Association for Computing Machinery, New York, NY, USA, Article 1, 15 pages. https://doi.org/10.1145/2523616.2523635

[38] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. 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)*. USENIX, Lombard, IL, 227–240. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/rajagopalan

[39] Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, and Hadar Serviansky. 2019. Fast Concurrent Data Sketches. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 207–208. https://doi.org/10.1145/3293611.3331567

[40] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. 2019. Scaling Distributed Machine Learning with In-Network Aggregation. *CoRR* abs/1903.06701 (2019). arXiv:1903.06701 http://arxiv.org/abs/1903.06701

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

[42] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 227–240. https://doi.org/10.1145/2785956.2787501

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

[44] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. 2020. Cheetah: Accelerating Database Queries with Switch Pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2407–2422. https://doi.org/10.1145/3318464.3389698

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

[46] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic Scaling of Stateful Network Functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 299–312. https://www.usenix.org/conference/nsdi18/presentation/woo

[47] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation* (Lombard, IL, USA), Nick Feamster and Jeffrey C. Mogul (Eds.). 29–42.

[48] Yu, Minlan and Rexford, Jennifer and Freedman, Michael J. and Wang, Jia. 2010. Scalable Flow-Based Networking with DIFANE. In *Proceedings of the ACM SIGCOMM 2010 Conference* (New Delhi, India) *(SIGCOMM '10)*. Association for Computing Machinery, New York, NY, USA, 351–362. https://doi.org/10.1145/1851182.1851224