Swing State: Consistent Updates for Stateful and Programmable Data Planes

Shouxi Luo* Hongfang Yu
University of Electronic Science and Technology of China

Laurent Vanbever
ETH Zürich

ABSTRACT

With the rise of stateful programmable data planes, a lot of the network functions that used to be implemented in the controller or at the end-hosts are now moving to the data plane to benefit from line-rate processing. Unfortunately, stateful data planes also mean more complex network updates as not only flows, but also the associated states, must now be migrated consistently to guarantee correct network behaviors. The main challenge is that data-plane states are maintained at line rate, according to possibly runtime criteria, rendering controller-driven migration impossible.

We present Swing State, a general state-management framework and runtime system supporting consistent state migration in stateful data planes. The key insight behind Swing State is to perform state migration entirely within the data plane by piggybacking state updates on live traffic. To minimize the overhead, Swing State only migrates the states that cannot be safely reconstructed at the destination switch.

We implemented a prototype of Swing State for P4. Given a P4 program, Swing State performs static analysis to compute which states require consistent migration and automatically augments the program to enable the transfer of these states at runtime. Our preliminary results indicate that Swing State is practical in migrating data-plane states at line rate with small overhead.

CCS Concepts

•Networks → Network architectures; Programmable networks; Network management;

*Work performed while interning at ETH Zürich.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

SOSR'17, April 3–4, 2017, Santa Clara, CA, USA

© 2017 ACM. ISBN 978-1-4503-4947-5/17/04...$15.00

DOI: http://dx.doi.org/10.1145/3050220.3050233

Keywords

Network updates; Software-Defined Networking; P4; Stateful programmable data planes.

1. INTRODUCTION

By enabling stateful applications to run directly *in* the data plane, at line rate, programmable data planes [9, 23, 8, 29, 28, 16, 24] have recently emerged as a promising research area.

Yet, despite making SDNs more powerful, maintaining states in the data plane also calls for new consistent update mechanisms as it prevents traditional update techniques from working, and this, for three main reasons. First, the fact that data-plane states can be updated at line rate—at speeds that can reach Tbps [5]—prevents any software-based controller from consistently moving states from one device to another. Inconsistent migration is a problem for any data-plane application that requires strong-consistency network-wide. Examples of such applications include stateful firewalls tracking dynamic flow characteristics (e.g., low-level TCP states [30]) or anomaly detection applications [22]. Second, even ignoring states dynamism, the exact set of states to be migrated may actually be unknown to the controller, preventing it from performing the migration in the first place. Indeed, the states location in memory can differ from device to device according to runtime factors (e.g. a hash computed on packet headers) that are invisible to the controller. Third, data-plane states can be shared across multiple flows, forcing these flows to be migrated at the same time to avoid inconsistency. Again, the exact flows to migrate can depend on runtime factors that are invisible to the controller.

This work We present Swing State, a general migration framework for stateful data planes. Swing State addresses the above challenges by consistently moving states from one device to another entirely within the data plane, at line rate. The key idea is to have each packet record the state values it reads at the source data plane, carry them to the destination device (through piggybacking), and override the memory locations it reads there. Once the corresponding states in the source

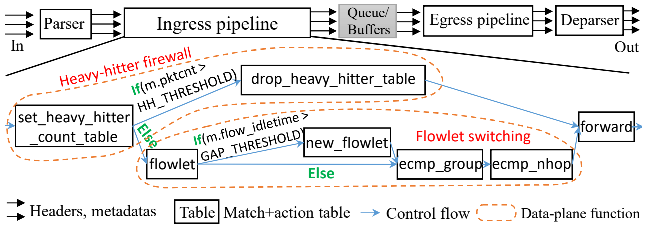

Figure 1: Abstract data-plane model used by P4 with a showcase of how data-plane functions are implemented.

and the destination devices are synchronized, flows can be migrated using any existing network update techniques such as [26]. Swing State is generic and enables to consistently shift data-plane states pertaining to any P4 program, without human intervention.

Swing State achieves consistent data-plane state migration in three consecutive steps. First, prior to deploying a P4 program, Swing State automatically analyzes it to figure out: *(i)* which states require live migration (because they cannot be safely reconstructed from the traffic); and *(ii)* which flow headers can update them at runtime. Second, Swing State augments the P4 program to enable the live migration of these states. Third, upon a state migration request pertaining to a set of flows, Swing State configures the source switch to piggyback the relevant state values onto the corresponding traffic. The destination switch then decapsulates these values and overrides its own states accordingly. Once the states are synchronized, Swing State lets the source temporarily mirror the relevant traffic to the destination and notifies the controller that it can safely reroute the flows.

Novelty While consistent network updates has been the topic of extensive research (e.g., [31, 26, 15, 32]), we are not aware of any technique ensuring per-packet consistency in the presence of data-plane states. Also, with respect to consistent state migration initiatives in the context of Network Function Virtualization [13], middleboxes [25], or network controllers [14, 27, 33], the key novelty of Swing State is that it works at line rate, over hardware-based data planes. In contrast, previous works focused on migrating software-maintained states that are up to orders of magnitude less dynamic.

Contributions To sum up, our main contributions are:

- Swing State, a general state-management framework alongside with a runtime system which enables live state migrations in any P4-enabled network (§2);
- A static analysis algorithm which automatically identifies the states requiring live migration in a P4 program (§3), together with an augmentation procedure to actually support live migration at runtime (§4);
- An efficient state synchronization procedure (§5);
- An implementation of Swing State along with a preliminary evaluation assessing its feasibility (§6).

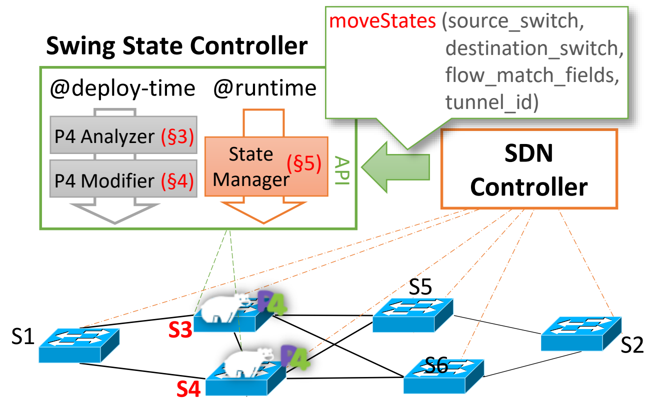

Figure 2: Swing State architecture. Using the `moveStates` primitive, SDN controllers can instruct the Swing State runtime to consistently migrate the states maintained for one or more flows from one switch to another.

2. MOTIVATION

In this section, we first explain with a simple example how P4 program leverages data-plane states and why it is hard to shift them around (§2.1). We then describe the core principles behind Swing State (§2.2).

2.1 Background

Stateful P4 data planes In P4-enabled switches, data-plane states (stored in registers[1]) reside in the device's ingress or egress pipelines and are maintained by actions. P4 developers construct data-plane functions by defining match+action tables, along with control flows, and header parsers/deparsers.

As an illustration, P4 enables to easily implement a stateful firewall which automatically drops the traffic originating from heavy hitters and load-balances the rest. Figure 1 depicts one possible implementation [4]. It involves two tables connected with a conditional control flow. The first table, `set_heavy_hitter_count_table`, counts the packets in each TCP flow (`m.pktcnt`), before caching the result in `hh_pktcnt` by hashing the packet's header (see Line 18–23 in Figure 3). If this count is larger than `HH_THRESHOLD`, the packet goes to the `drop_heavy_hitter_table` where it is dropped; otherwise, it goes to the reset ingress pipelines where it is load-balanced according to the flowlet it belongs to.

Update scenario Suppose that switch S3 (Figure 2) runs the aforementioned application and that flows from S1 to S2 (crossing S3) have been flagged as heavy. Now consider that we need to reboot S3 (e.g., to perform a firmware update). To avoid impacting the traffic, we want to move flows away from S3 to S4. Yet, simply

[1]`register` is one kind of state residing in data plane. Other state types include `counter` and `meter`. Unlike registers though, they are more akin to write-only objects as they can only be referenced in special primitive actions [23]. Thus, this paper focuses on register states.

```
1 #define REG_SIZE 8192
2 field_list l4_fields {
3   ipv4.srcAddr;
4   ipv4.dstAddr;
5   ipv4.protocol;
6   tcp.srcPort;
7   tcp.dstPort;
8 }
9 register hh_pktcnt{
10   width: 16;
11   instance_count: REG_SIZE;
12 }
13 field_list_calculation l4_hash {
14   input { l4_fields;}
15   algorithm : crc16;
16   output_width : 16;
17 }
18 action set_hh_count() {
19   //m is an user-defined metadata
20   modify_field_with_hash_based_offset(m.flow_id,
       0, l4_hash, REG_SIZE);
21   m.pktcnt = hh_pktcnt[m.flow_id] + 1;
22   hh_pktcnt[m.flow_id] = m.pktcnt;
23 }
24 //this table only has a default action
25 table set_heavy_hitter_count_table {
26   actions { set_hh_count; } size: 0;
27 }
```

Figure 3: An implementation of `set_heavy_hitter_count_table` (see Figure 1), written in P4 v1.1 [1].

```
register lasttime {//used to detect new flowlets
  width: 32; instance_count: REG_SIZE;
}
register flowlet_id {//used for ecmp hashing
  width: 16; instance_count: REG_SIZE;
}
action lookup_flowlet_map() {
  m.flow_idletime = intrinsic_metadata.
      ingress_global_timestamp-lasttime[m.flow_id];
  lasttime[m.flow_id] = intrinsic_metadata.
      ingress_global_timestamp;
  m.flowlet_id = flowlet_id[m.flow_id];
}
//compute inter-packet gap and update lasttime
table flowlet {
  actions { lookup_flowlet_map; } size: 0;
}
action update_flowlet_id() {
  m.flowlet_id = m.flowlet_id + 1;
  flowlet_id[m.flow_id] = m.flowlet_id;
}
table new_flowlet {
  actions { update_flowlet_id; } size: 0;
}
```

Figure 4: An example implementation of `flowlet` and `new_flowlet` shown in Figure 1, written in P4 v1.1 [1].

shifting flows from S3 to S4 (e.g. using [26]) would cause the runtime states stored in `hh_pktcnt` to be lost, allowing traffic that should be dropped to go through.

This example sheds light on two fundamental questions regarding consistent data-plane states migration:

What to migrate? Not all states require consistent migrations: some functions can automatically recover their states from live traffic. This is for instance the case for our implementation of flowlet detection (Figure 4) which records each flow's last reference time and current flowlet ID in register `lasttime` and `flowlet_id`. Migrating these states is not necessary as they can be reconstructed nearly immediately at the destination.

How to migrate? The simplest way to migrate states is to request the control plane to export the states from the source device to the destination device (e.g., similarly to [13]).

Unfortunately, simply migrating states via the control plane does not work because of: *(i)* the speed at which states can be updated (Tbps in the new generation of programmable data-planes [5]); and *(ii)* the flexible support of state references which makes it hard, if not impossible, to infer the exact states location at runtime.

The latter problem results from the fact that many applications (e.g. [6, 13, 25, 28]) reference per-flow states using a hash of the corresponding packet headers. As different switches might use different set of inputs for hashing or have different capacity/size for the register arrays, the resulting state location can end up being device-specific. For instance, the destination switch might employ a less-specific input for hash calculation (e.g., based on 4 tuples instead of 5) and have a larger `REG_SIZE`, in which case each flow's state location (e.g., Line 20, Figure 3) would shift. Even worse, the fact that P4 supports the use of runtime data (e.g., action parameters) as input to the hash functions can make it impossible for the control plane to infer the exact reference to use for accessing each state at compilation time.

2.2 Overview

To support consistent and live network updates in stateful data planes, we propose Swing State, a general state-management framework and runtime system offering one main primitive: `moveStates` (Figure 2). At its core, Swing State adopts the novel idea of automatically identifying the states requiring consistent migration then letting each packet/flow *itself* move the state values it has read by leveraging the programmability of data plane. With Swing State, devices can perform live migrations of data-plane states without freezing traffic nor the rule updates made by the control plane.

Swing State capability of moving states at runtime means that developers can write P4 applications without having to care about migration. At deploy time, Swing State *Analyzer* analyses their P4 program to infer which states: *(i)* require migrations; and *(ii)* are shared between multiple flows meaning they should be treated as a whole. Based on this analysis, Swing State *Mod-*

ifier automatically augments the program to support live state migrations. By using the augmented program, reconfigurable devices automatically support consistent and live migration for data-plane states.

Once the SDN controller wants to migrate a set of flows f from one device to another, the *State Manager* first checks whether this migration is safe based on its state analysis. In case f shares critical states with others, or it uses device-specific states (see below), the *State Manager* raises an alert along with remarks. Otherwise, the *State Manager* configures the source and destination devices to migrate data-plane states. Once all required states have been migrated, the *State Manager* notifies the controller that it can update f's paths safely.

3. STATIC ANALYSIS

In this section, we describe how Swing State analyzes P4 programs to identify state types (§3.1) along with the corresponding flow spaces that use them (§3.2).

3.1 State taxonomy

We classify P4 data-plane states along two dimensions: *(i)* their usage (soft vs hard); and *(ii)* whether their values are location-dependent or not.

Property 3.1 (Soft state vs Hard state)**.** *A P4 state is soft if its value is computed from, or maintained depending on, random variables, such as the time stamps of events triggered by packets (e.g., arrive, enqueue, dequeue, or leave), the occupancies of queues, meter values, etc. Otherwise, the state is considered as hard.*

Soft states are typically used for optimization purposes in congestion control algorithms, scheduling, and active queue management [28]. As the values of soft states are essentially random, the data-plane functions tolerate inconsistency by design. As an example, in Figure 4, the states stored in `lasttime` and `flowlet_id` are soft as they depend on packet arrival times.

In contrast, hard states are maintained deterministically and explicitly (e.g., according to a state machine) and cannot easily be recovered from live traffic. As an example, the packet count stored in `hh_pktcnt` used by heavy-hitter detection (Figure 3) is hard. Other examples include security applications, such as stateful firewalls or anomaly detection, whose state machines depend on few key observations (e.g., TCP state tracking [30]) that only happen once in the lifetime of a flow. The mapping between virtual IPs (VIPs) and direct IPs (DIPs) found in any network load-balancer [11, 12] is another example of hard state which is usually set at the beginning of the connection.

From an update viewpoint (see Table 1), only hard states require to be migrated as soft states can be reconstructed at the new location of the flow, at the price of a slightly less efficient (but still, correct) network.

Property 3.2 (Location dependency)**.** *A P4 state is location-dependent if its value is device-specific, such as port id, local time stamps of events triggered by packets, or the current occupancy of a queue.*

Location	Dependent	Independent
Soft	No migration (e.g. [2])	No migration (e.g. [3])
Hard	Data-plane migration with transfer function	Direct migration (e.g. [19, 10, 20, 21, 4])

Table 1: Only hard states require to be consistently migrated as soft states can directly be reconstructed at the target device. Location-dependent states further required to be transformed to ensure compatibility.

Some hard states only make sense locally and/or depend on the network topology. To avoid correctness issues, these hard states therefore need to be "translated" to the corresponding state representation used at the target device. For instance, consider a switch running a stateful application which builds a list of MAC addresses authorized to send traffic on each physical port. Shifting flows crossing this switch to another one requires to move the corresponding decisions, while adapting the references to the physical ports to corresponding ones on the target switch.

Swing State requires the developers to write specific transfer functions [27] to migrate hard and location-dependent states. While writing transfer functions requires detailed knowledge of the application and the topology, most of the location-dependent states encountered in practice are soft and therefore require no migration (nor transfer functions). Indeed, *none* of the stateful P4 programs we analyzed [3, 4, 10, 18, 19, 20, 22, 28] required transfer functions.

3.2 Flow-space dependencies

If multiple flows read and write the same state, they should be migrated together, as doing otherwise could cause inconsistent forwarding. Swing State needs therefore to be aware of the flow space using each state.

Given a P4 state, the flow space that uses it is given by all the expressions used in the control flow selection, action lookup, and the index calculation. Unfortunately, P4 also supports state indexes to be computed from runtime factors such as any other register values and action parameters (e.g., [18]), which makes it impossible to *precisely* figure out at compilation time which set of packets share any given state.

To address this problem, Swing State considers a generalized flow space when runtime inputs are used (similarly to [17]). Specifically, Swing State abstracts away runtime inputs and only considers the subpart directly parsed from the packet and employed by the state's reference/index calculation. As an example, the precise input space of `lasttime`'s reference (Figure 4) implicitly involves `m.pkt <= HH_THRESHOLD`, the expression employed by the control flow (Figure 1). Overlooking it to treat values in `lasttime` as per-flow states, indexed by the unmodified 5-tuple, results in a conservative answer.

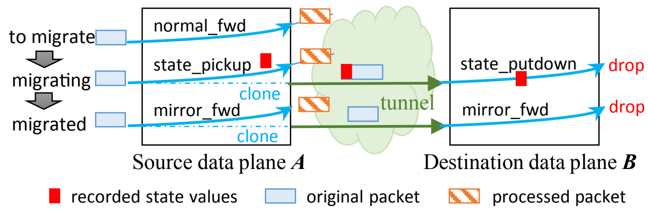

Figure 5: The source and destination devices cooperate to migrate states with different forwarding modes.

Performing the analysis For each P4 program, Swing State *Analyzer* first analyzes how its match+action tables are connected and how each employed action is implemented. It then builds a Directed Acyclic Graph (DAG) to capture the control and data dependencies among the involved header and metadata fields. Finally, it infers the types of each state along with the set of flow spaces modifying them. For this, it leverages the names of P4 pre-defined metadata fields [23, 1]. As an illustration, soft states are attached to P4 metadata fields such as `ingress_global_timestamp`, `deq_timedelta`, and `deq_qdepth`; while location-dependent states typically uses P4 metadata fields such as `ingress_port`.

While relying on P4 metadata fields for inferring types is simple, it comes with two drawbacks. First, metadata fields are platform-dependent and therefore requires expert knowledge to ensure correctness. Second, the analysis is sometimes not precise as some inferred hard states could actually be treated as soft should we know the high-level applications intent (as such, it is conservative). Here, enabling P4 developers to annotate how states should be handled from a migration viewpoint would certainly be helpful; we leave this for future work.

4. MAKING P4 STATES "SWINGABLE"

In this section, we describe how Swing State augments P4 programs to enable runtime migration. We focus on migrating *hard* states involved in the ingress pipeline (where most of the processing lies).

4.1 Forwarding modes

Swing State achieves live state migration by introducing four types of local forwarding modes:

1. **normal_fwd:** the default mode in which the data plane forwards the packet normally.
2. **state_pickup:** a mode appearing only at a migration's source device in which the data plane: *(i)* forwards the packet normally, while recording the used state values; and *(ii)* makes a clone of the original packet, encapsulates it with the recorded values, then tunnels it to the destination device.
3. **state_putdown:** a mode appearing only at a migration's destination in which the data plane: *(i)* decapsulates the packet to get its original header and state values; *(ii)* processes the original header normally while overriding each state before reading it; and *(iii)* drops the processed packet at the end.
4. **mirror_fwd:** a mode having different meanings for a migration's source and destination devices. In this mode, the source device forwards the packet normally while tunneling a clone to the destination, while the destination processes the decapsulated packet normally and drops.

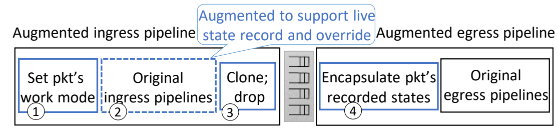

Figure 6: The data-plane/pipeline augmentations made by Swing State (the modification to parsers is omitted).

Figure 5 illustrates how Swing State synchronizes the data-plane states at the source and destination devices by changing the forwarding mode assigned to packets from `normal_fwd` to `state_pickup` to `mirror_fwd`. We describe the process in more details in §5.

4.2 Program augmentations

Swing State automatically augments P4 programs to support these forwarding modes (Figure 6). The key insight is to let each (stateful) action itself *record* and *override* the read state values for each packet. To achieve this, Swing State inserts code snippets just before each read access to hard states. For instance, consider the stateful action `set_hh_count` in Figure 3: `hh_pktcnt` tracks flows' packet counts where that number of the flow in process is `hh_pktcnt[m.flow_id]`. Swing State generates metadata field `_SS_m.hh_pktcnt_0` for the reading of `hh_pktcnt[m.flow_id]` at Line 5 (Figure 7).

If a packet is in the mode of `state_pickup`, the augmented `set_hh_count` would cache the observed value (Line 4) and set metadata `_SS_m.stateful` to 1 (Line 7), indicating the packet being processed is using states. Then the augmented pipelines would: *(i)* make a clone of the original packet (③, Figure 6); *(ii)* encapsulate the clone with this cached state value; and *(iii)* tunnel it to the target data plane (④, Figure 6). In case of `state_putdown`, the augmented parser and ingress pipeline decapsulate the received clone and cache the state values in `_SS_m.hh_pktcnt_0` (①, Figure 6) which is then used by `set_hh_count` to overwrite the value of `hh_pktcnt[m.flow_id]` (Line 3, Figure 7); finally, this clone packet gets dropped (③, Figure 6).

5. MANAGING STATE MIGRATION

Migrating states from switch A to B for a given flow space f includes 4 steps.

1. Configure B to accept states destined to it. Swing State employs specific tags (i.e., `tid`) to identify concurrent migration tasks. Upon receiving an encapsulated packet, B's ingress pipeline, i.e., ① in Figure 6,

```
1 action set_hh_count() {
2   modify_field_with_hash_based_offset(m.flow_id,
      0, l3_hash, REG_SIZE);
3   hh_pktcnt[m.flow_id] =
3       (_SS_m.pkt_wmode == state_putdown) ?
3      _SS_m.hh_pktcnt_0 : hh_pktcnt[m.flow_id];
4   _SS_m.hh_pktcnt_0 = hh_pktcnt[m.flow_id];
5   m.pktcnt = hh_pktcnt[m.flow_id] + 1;
6   hh_pktcnt[m.flow_id] = m.pktcnt;
7   _SS_m.stateful = (_SS_m.pkt_wmode
7      == state_pickup) ? 1 : _SS_m.stateful;
8 }
```

Figure 7: Swing State *Modifier* inserts Line 3 (in pink) to enable state overrides, and inserts Line 4 and 7 (in lime) to enable state records, by using *ternary operators* [1].

first checks whether this packet carries state values. If so, ① decapsulates the packet to get the original header, caches the carried state values in pre-defined metadata fields (e.g., `_SS_m.hh_pktcnt_0`), and sets this packet's work mode as `state_putdown`. The actions then overwrite the states read by this packet.

2. Activate A to emit f's states. To let A's data plane record f's states, Swing State inserts match+action rules into ①, so that f's packets would be marked as `state_pickup`. If some state values have been recorded during ② (i.e., `_SS_m.stateful==1`), ③ will clone this packet to egress pipeline via primitive action `clone_i2e`, then ④ encapsulates this clone with the recorded state values and delivers them to B via tunnel `tid`. From now on, all the state values used by f would be automatically synchronized/mirrored to its target data plane, B.

3. Wait for incoming packets. As states are piggybacked on traffic, Swing State waits for matching traffic to trigger the migration process.

4. Activate `mirror_fwd` for f on A. After Step 3, all f's state values in A and B have been synchronized. Swing State then configures A's ① to set f's work mode as `mirror_fwd`; f's incoming packets will be mirrored to B. Then B processes them as normal and drops.

After Step 4, all states involving f have been migrated and the flow can safely be moved from A to B.

6. PRELIMINARY EVALUATION

We have successfully used Swing State to analyze and augment the P4 application shown in Figure 1.

Implementation As the API of a P4 data plane is automatically generated from its code, managing the actual state migration (using the match+action rules described in ①, ③, and ④) is relatively straightforward. Also, since the two current versions of P4, 1.0.2 and 1.1.0, are not syntactically incompatible, both the analysis and augmentations are performed on the JSON-formed Intermediate Representations (IR) outputted by the P4 front-end compiler and which is designed to be consistent across different versions [7].

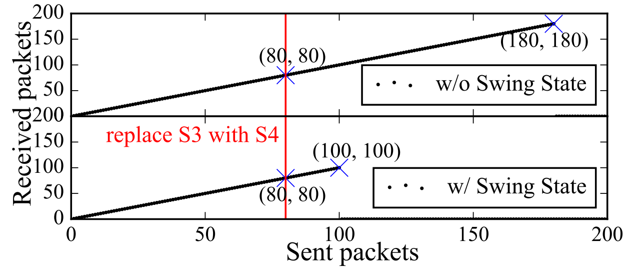

Figure 8: Without Swing State, the flow's packet counts used by heavy-hitter firewall would get lost (i.e., values in `hh_pktcnt`), resulting in allowing packets that should be dropped (the threshold of dropping is 100).

Case study We check whether the augmented P4 application supports consistent network updates by reproducing the example of Figure 2 (moving flows from switch S3 to S4). We set the threshold of heavy hitter to 100 and let S1 send packets to S2 via a TCP connection. Figure 8 shows how the number of received packets (at S2) changes when the network is updated with and without Swing State. With Swing State, the values of packet counts get migrated correctly; thus, the stateful firewall works perfectly as no update happened.

Limitations & future work While promising, the current (preliminary) version of Swing State is limited and requires future work. First, Swing State is not designed to deal with: *(i)* packet re-ordering and loss; and *(ii)* inconsistent hash collisions between different hash implementations. In rare cases, these issues might lead to inconsistent migrations. Second, Swing State does not currently support state merging operations should the source and the target switch have states in common. Finally, our current implementation of Swing State can mirror multiple times the same state value, resulting in wasted bandwidth. A possible solution here is to filter duplicates at the source using a bloom filter.

7. CONCLUSION

This paper introduced Swing State, a general framework for migrating data-plane states for programmable switches. By directly piggybacking state values on traffic, Swing State migrates data-plane states without freezing the traffic nor control-plane updates. We implemented a Swing State prototype and showed that it can automatically analyze and augment P4 programs as well as successfully perform a live migration of the states pertaining to a heavy-hitter firewall.

Acknowledgments

We would like to thank our shepherd, Jeongkeun Lee, for his invaluable help in improving the paper and the anonymous reviewers for their constructive feedback. Shouxi Luo was supported by the China Scholarship Council during his stay at ETH Zürich.

8. REFERENCES

[1] Official extension to P4 v1.1 spec. https://github.com/p4lang/tutorials/tree/master/p4v1_1/simple_router.

[2] P4 OpenState applications. https://github.com/OpenState-SDN/openstate.p4.

[3] P4 SIGCOMM 2015 Tutorial Exercise 2: Implementing TCP flowlet switching. https://github.com/p4lang/tutorials/tree/master/SIGCOMM_2015.

[4] P4 SIGCOMM 2016 Tutorial: Implementing Heavy Hitter Dectection. https://github.com/p4lang/tutorials/tree/master/SIGCOMM_2016.

[5] The World's Fastest & Most Programmable Networks. Barefoot Networks White paper, 2016.

[6] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *SIGCOMM*, New York, NY, USA, 2016. ACM.

[7] A. Bas. Enabling fast P4 development with bmv2. P4 Workshop 2016.

[8] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch. *SIGCOMM CCR*, 44(2):44–51, Apr. 2014.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM CCR*, 44(3):87–95, July 2014.

[10] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at Network Speed. In *Symposium on SDN Research*, SOSR '15, New York, NY, USA, 2015. ACM.

[11] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI*, pages 523–535, Santa Clara, CA, USA, Mar. 2016. USENIX Association.

[12] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud Scale Load Balancing with Hardware and Software. In *SIGCOMM*, New York, NY, USA, 2014. ACM.

[13] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *SIGCOMM*. ACM, 2014.

[14] S. Ghorbani, C. Schlesinger, M. Monaco, E. Keller, M. Caesar, J. Rexford, and D. Walker. Transparent, Live Migration of a Software-Defined Network. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 3:1–3:14, New York, NY, USA, 2014. ACM.

[15] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving High Utilization with Software-driven WAN. In *SIGCOMM*. ACM, 2013.

[16] E. J. Jackson, M. Walls, A. Panda, J. Pettit, B. Pfaff, J. Rajahalme, T. Koponen, and S. Shenker. SoftFlow: A Middlebox Architecture for Open vSwitch. In *USENIX ATC*, pages 15–28, Denver, CO, USA, June 2016. USENIX Association.

[17] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In *NSDI*, pages 239–253, Santa Clara, CA, Mar. 2016. USENIX Association.

[18] J. Klomp. P4 VPN Authentication; Authentication of VPN Traffic on a Network Device with P4. Technical report, University of Amsterdam, July 2016.

[19] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *OSDI*, pages 467–483, GA, 2016. USENIX Association.

[20] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A Better NetFlow for Data Centers. In *NSDI*, Santa Clara, CA, USA, Mar. 2016. USENIX Association.

[21] Y. Li, R. Miao, C. Kim, and M. Yu. LossRadar: Fast Detection of Lost Packets in Data Center Networks. In *CoNEXT*, pages 481–495, New York, NY, USA, December 2016. ACM.

[22] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *SIGCOMM*, New York, NY, USA, 2016. ACM.

[23] The P4 Language Consortium. *The P4 Language Specification*, version 1.1.0 edition, January 2016.

[24] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *NSDI*, pages 117–130, Berkeley, CA, USA, 2015. USENIX Association.

[25] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *NSDI*, Lombard, IL, USA, 2013. USENIX Association.

[26] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, New York, NY, USA, 2012. ACM.

[27] K. Saur, J. Collard, N. Foster, A. Guha, L. Vanbever, and M. Hicks. Safe and Flexible Controller Upgrades for SDNs. In *Proceedings of the Symposium on SDN Research*, SOSR '16, New York, NY, USA, 2016. ACM.

[28] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet transactions: High-level programming for line-rate switches. In *SIGCOMM*, New York, NY, USA, 2016. ACM.

[29] H. Song. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. HotSDN '13, New York, NY, USA, 2013. ACM.

[30] B. Tschaen, Y. Zhang, T. Benson, S. Banerjee, J. Lee, and J.-M. Kang. SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining. In *IEEE SDN-NFV Conference*, 2016.

[31] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure. Seamless network-wide IGP migrations. In *ACM SIGCOMM 2011 conference*, pages 314–325. ACM, 2011.

[32] S. Vissicchio and L. Cittadini. Flip the (flow) table: Fast lightweight policy-preserving sdn updates. In *INFOCOM*. IEEE, 2016.

[33] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford. Virtual Routers on the Move: Live Router Migration As a Network-management Primitive. In *SIGCOMM*, pages 231–242, New York, NY, USA, 2008. ACM.