Elastic Slicing in Programmable Networks

Turkovic, B.; Nijhuis, S.H.; Kuipers, F.A.

Citation (APA)
Turkovic, B., Nijhuis, S. H., & Kuipers, F. A. (2021). Elastic Slicing in Programmable Networks. In K. Shiomoto, Y.-T. Kim, C. E. Rothenberg, B. Martini, E. Oki, B.-Y. Choi, N. Kamiyama, & S. Secci (Eds.), *Proceedings of the 2021 IEEE Conference on Network Softwarization: Accelerating Network Softwarization in the Cognitive Age, NetSoft 2021* (pp. 115-123). Article 9492528 (Proceedings of the 2021 IEEE Conference on Network Softwarization: Accelerating Network Softwarization in the Cognitive Age, NetSoft 2021). IEEE. https://doi.org/10.1109/NetSoft51509.2021.9492528

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Elastic Slicing in Programmable Networks

Belma Turkovic, Sjors Nijhuis, and Fernando Kuipers

Delft University of Technology, the Netherlands

Abstract—The concept of network slicing enables operators to provision multiple virtual networks on top of a single (shared) physical infrastructure. Adding elasticity to slicing, i.e., the ability to on-demand provision/release dedicated network resources, improves resource utilization. However, efficiently allocating and scaling slice resources, while maintaining state consistency, is challenging. Especially with P4-programmable network devices that process packets at Tbps speeds, controller-driven scaling of network functions would be too time-consuming, and data-plane scaling is needed.

In this paper, we address this need, by developing a custom scaling protocol and framework that can consistently, with negligible delay, scale network slices and functions transparently to the slice end-users. We compare, via emulation and experiments on programmable hardware, our approach to state-of-the-art scaling techniques and demonstrate significant slice resource utilization improvements and scaling duration reductions.

I. INTRODUCTION

Since their introduction, Software-Defined Networking (SDN) and Network Functions Virtualization (NFV) have enhanced network flexibility, reconfigurability, and agility [7], [15]. Moreover, when combined, they support offering Quality-of-Service (QoS) through the concept of network slicing [5], [18].

Network slicing assumes that virtual networks, each tailored to different service needs, are created on top of a shared physical infrastructure. Each of these virtual networks consists of virtual nodes and virtual links. As can be seen in Fig. 1, every virtual link represents a path with reserved network resources (e.g., bandwidth) in the physical network. Moreover, those links connect virtual nodes, representing Network Functions (NFs), that provide a specific network functionality. Stateful NFs, such as firewalls or heavy-hitter detection, require knowledge of the previously processed packets to function correctly, while stateless NFs, such as routing, do not [27].

As traffic volumes are unpredictable and generally change over time, a static slicing solution either over-provisions resources or does not guarantee a certain QoS [20]. A slicing solution that supports elasticity, i.e., the ability to automatically scale the assigned network resources to match the current traffic volumes, would solve this problem. We discern two ways of scaling: (1) vertical scaling, in which slice resources are scaled up/down at the NF(s) and reserved bandwidth increased/decreased on the virtual link(s) (Fig. 1a), and (2) horizontal scaling, in which a new NF is deployed/removed and a portion of the traffic redirected by creating/removing virtual links connected to the slice, to reduce the load on the existing NF (Fig. 1b). This way, service providers only

pay for the resources they use (pay-per-use model), end-users get their requested level of QoS level, and network providers can support multiple services simultaneously using fewer resources.

Scope & Motivation. In this paper, we consider elastic slicing (both vertically and horizontally) in the context of P4-programmable data-planes [3]. On the one hand, NFs primarily responsible for packet forwarding (e.g., firewalls, NAT, monitoring) might benefit from hardware acceleration by being offloaded to (P4) programmable switches. On the other hand, programmable hardware can process packets at Tbps speeds [11]. Hence, NF-state may change very frequently, making controller-driven scaling and traditional NFV frameworks (focused on migrating software-maintained states) too time-consuming. Moreover, even if controllers could keep up, migrating an NF, which potentially maintains hundreds of state variables per flow [13], would overload the controller, thereby prolonging the scaling time and leading to state inconsistencies. Yet, up-to-date state information is crucial for the correct functioning of many NFs.

Fortunately, programmable switches come with monitoring features that enable the data-plane to report the exact QoS the packets experienced while being processed [14], [26]. And, in contrast to centralized approaches, programmable switches allow us to offload time-sensitive actions from the central controller to the data-plane. When combined with the advanced monitoring features, we can quickly detect and react to changing traffic conditions [20].

Contributions & Outline. *We present an elastic network-slicing framework for P4-programmable network devices that economizes on slice resource utilization, while maintaining state consistency and at low scaling time.*

We split our framework into two components: (1) a central controller and (2) a data-plane component. With its global view of the network, the central controller is responsible for long-term network management, such as the (de)allocation of NFs and route calculation. The data-plane component is deployed directly on the switches and only has a local network view but fast reaction time and accurate traffic information. Therefore, it is responsible for reacting to time-sensitive operations, such as load monitoring (Sec. II-A), state transfer, and virtual link configuration (Sec. II-B).

In Sec. III, we evaluate our framework, both through emulation as well as via experiments on programmable hardware, by comparing it to traditional (controller-driven) approaches. Our experiments show that only by having an "intelligent" data-plane, on-time scaling can be achieved.

(a) Vertical scaling. Whenever a new flow joins or the volume of an existing one increases, resource reservations on all the virtual links are increased.

(b) Horizontal scaling. When NF3 is no longer able to process all the incoming traffic without any QoS degradation, and/or bandwidth assigned on virtual links NF2-NF3 and NF3-NF2 cannot be scaled up, a new NF3 instance NF3' is spawned and two new virtual links (NF3'-NF2 and NF3'-NF4) created (Scale-Out). Similarly, if enough resources to process all the incoming traffic are present on one of the paths, two virtual nodes are merged into one (Scale-In).
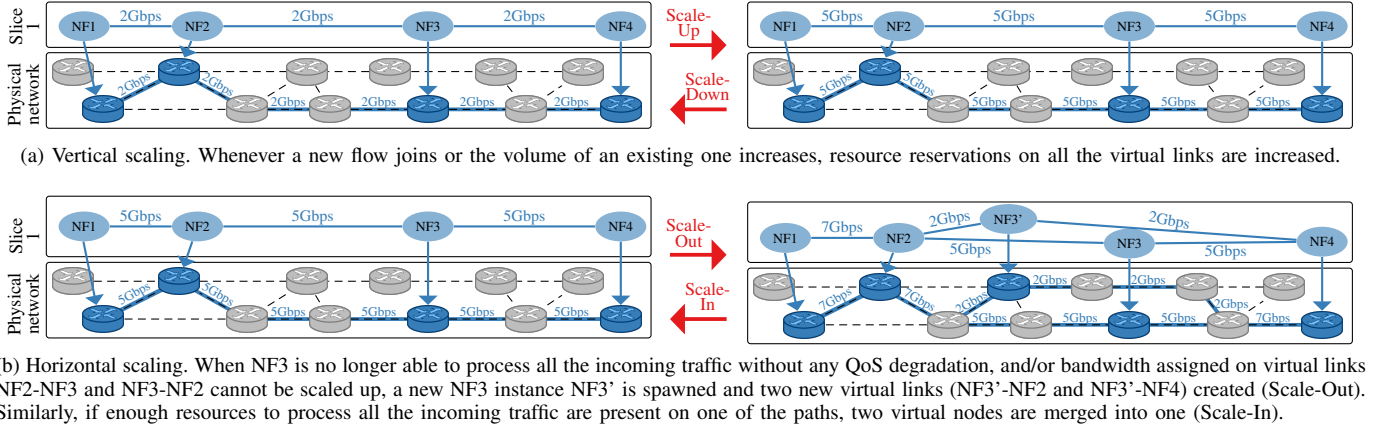
Fig. 1: Different types of scaling.

II. ELASTICITY FRAMEWORK

To support elasticity, we propose a hierarchical framework (Fig. 2a) consisting of:

1) A central controller (CC) that, with its global overview of the available network resources and existing traffic flows, determines the network's long-term behavior. It is responsible for initializing new slices, finding the most appropriate locations to place NFs during scaling, and guiding the data-plane component.
2) The data-plane component (DPC) that, based on the central controller's input and measured traffic conditions, performs all latency-sensitive tasks. It is responsible for load monitoring, state transfer, and flow rerouting.

To support the information exchange between different data-plane components running at different switches, we implemented a custom *slice management (SM) protocol*, shown in Fig. 2a. During each scaling process (horizontal or vertical), the DPCs exchange information and update the slice (e.g., reroute flows, transfer state, adjust bandwidth) using the SM header.

Fig. 2b illustrates the processes to react to overload. DPC processes are shown in green and involve all latency-sensitive tasks. For example, a DPC continuously monitors the slice and quickly detects traffic changes. In contrast to a controller-driven approach, it makes this decision on a per-packet basis, informing the CC only when scaling is needed, and does not depend on a monitoring interval or the control-plane's speed. If detected, the CC (whose processes are shown in blue) first attempts the easier of the two: vertical scaling. If insufficient resources are available at that path, the CC tries to reroute a flow to a path that would have enough resources or, ultimately, scales-out. Since scaling-out is the most involved of these three situations, Fig. 2 explains the involved interactions in more detail. It is essential to notice that, while the CC indicates which flows to reroute (green-blue blocks in Fig. 2b), the process of rerouting and state-transfer is completely offloaded to the data-plane (Fig. 2). Therefore, route/state inconsistencies, due to one switch receiving a controller update (e.g., a new route) before others or after the state has changed, are avoided.

During under-load detection, the processes are similar (but reversed). As before, upon detection, DPCs inform the CC, which either scales vertically (scaling-down) or horizontally (scaling-in). However, in contrast to the previous example, during under-load, if enough resources are available on one path so that another one can be merged into it, the framework will initiate the scaling-in (horizontally) independently of the possibility to scale-down (vertically, see Sec. II-A).

How much bandwidth to assign? We decided to use an auto-tuned value ΔBw, which represents the step in which we increase/decrease reserved bandwidth. If ΔBw is set low, the reserved bandwidth is increased/decreased in tiny steps. While this increases resource efficiency, it also leads to instability and frequent scaling processes for dynamic traffic. If ΔBw is high, resource efficiency reduces, and scaling will occur infrequently. Our solution initially assigns a low value to ΔBw to preserve the slice resources. After each scaling event, ΔBw is increased by a factor k, and a timer is started. If another scaling event occurs within this timer, the same process is repeated. This process continues until an interval is encountered in which no scaling occurred, which causes ΔBw to be reset to its initial value. This way, we start scaling conservatively, only assigning small chunks of bandwidth to the slice. However, if we detect a high increase, we quickly build-up ΔBw to reduce the number of scaling events.

Since our framework relies on offloading latency-sensitive tasks to the DPC, we will explain two main tasks: (1) load monitoring and (2) virtual link configuration, state transfer, and flow rerouting (as well as all involved modules) in more detail.

A. Load monitoring

Since the scaling procedure is relatively time-consuming and requires many network updates, frequent scaling would lead to network instability and sub-optimal resource utilization. To infer the best times for horizontal scaling, for each P4 NF, we deployed two two-rate three-color meters: *growth*
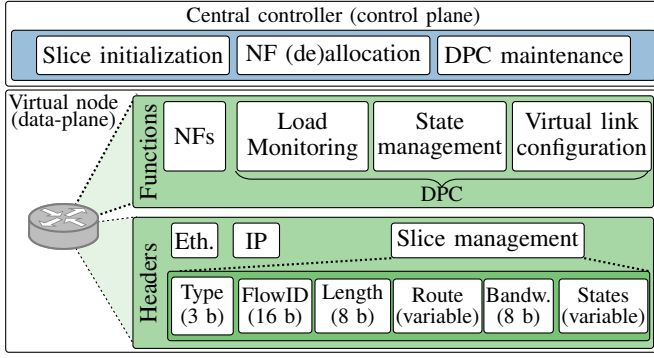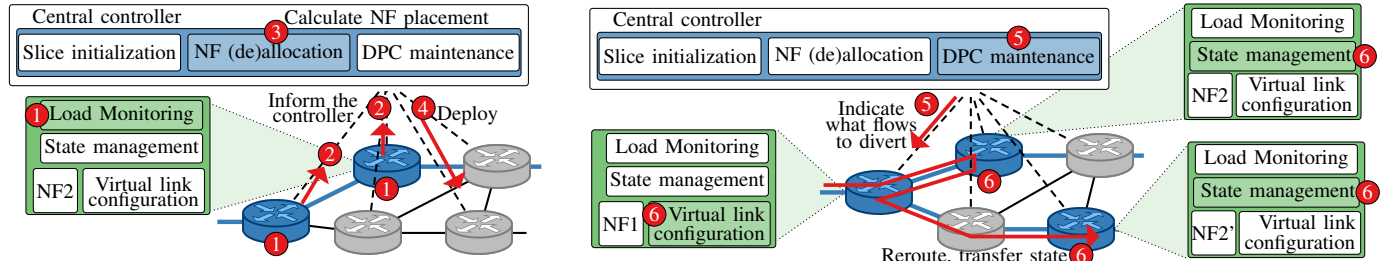
(a) Hierarchical design of the slicing framework.
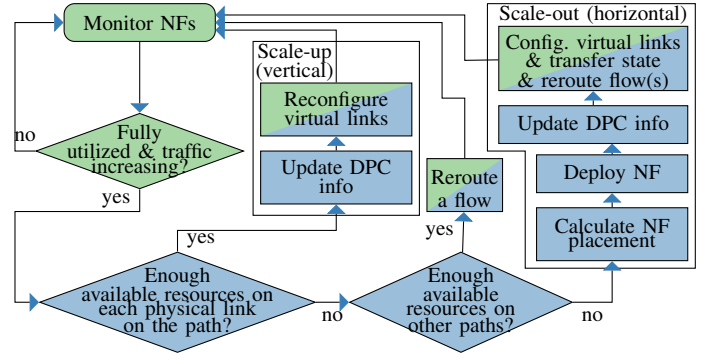
(b) Flow chart illustrating the process of scaling-out and -up.

(c) Interactions between components during scaling-out. DPCs continuously track the slice's utilization (step 1, Sec. II-A). If they detect overload, they notify the CC (step 2). The CC subsequently determines the switch where NF2' will be deployed on and the route to connect the node to the rest of the slice (step 3). Next, the CC deploys the new NF (step 4) and, upon its completion, informs NF1 to divert a portion of its traffic to the newly deployed NF2' (step 5). When a packet of this flow is received, the state from NF2 is transferred to NF2', and a virtual link is created using the Slice Management Protocol (step 6, Sec II-B).

Fig. 2: Elastic slicing framework. CC (DPC) modules and tasks are shown in blue (green).
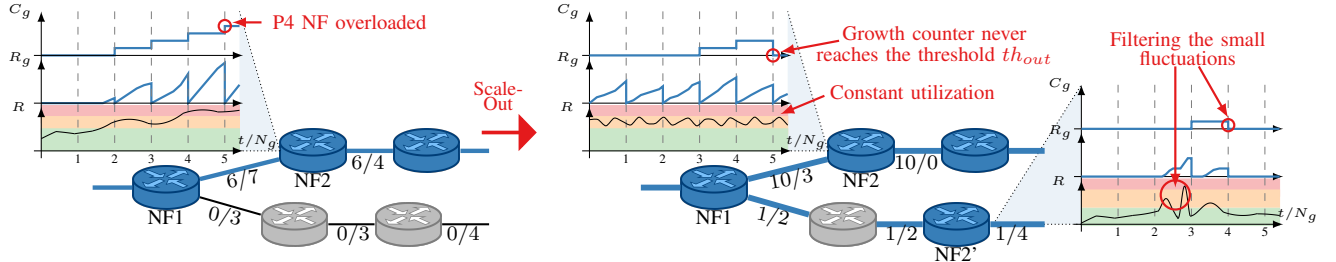
(a) Growth meter. If the traffic volumes increase and more and more packets are colored yellow or red, the growth rate R_g and, consequently, the growth counter C_g increase until C_g reaches the threshold M. At this moment, scaling is initialized. In contrast, small fluctuations in the traffic volumes or large temporary peaks are filtered out.
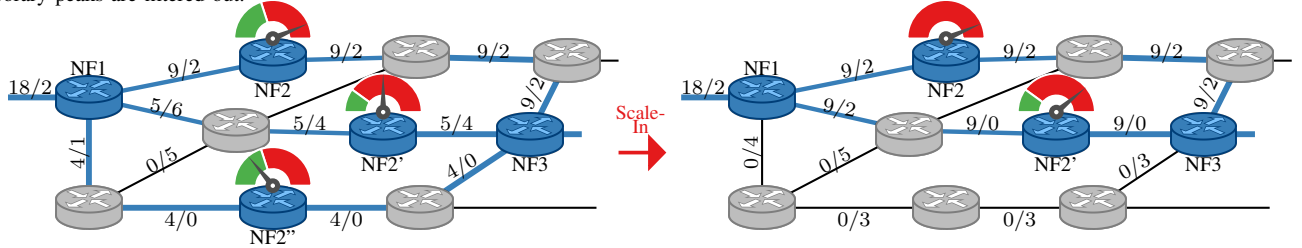
(b) Merging meter. To merge NF2" into NF2' or NF2, either of them needs to have enough resources to take over the traffic processed on NF2" (i.e., 4). DPC maintenance calculates the minimum resources available on the virtual links connecting NF2 and NF2' to NF1 and NF3. Finally, it sets the rate thresholds on NF2" to the maximum of these two: 4. Likewise, the thresholds on NF2 and NF2' are configured to 4 and 2, respectively. As all the packets processed on NF2" are green, i.e., NF2" is processing less traffic than the amount of free resources available on NF2', scaling-in is initiated and the thresholds recalculated to 0 and 2 for NF2 and NF2', respectively.

Fig. 3: Load monitoring. The first number above the link represents the amount of reserved bandwidth, while the second number represents the total amount of bandwidth available on the link (i.e., not assigned to any other slices).

meter to detect that the slice is close to its full capacity, and *decline meter*, to detect that the slice is underutilized.

Growth meter. Scaling-out (and up) is initialized whenever the slice is no longer able to process all the traffic without any degradation. Consequently, the growth meter rate threshold needs to be configured low enough to allow the CC to deploy the new NF without any degradation to any of the flows currently processed in the slice. In this paper, we set them to ΔBW and $2\Delta BW$ less than the reserved bandwidth BW_r. Additionally, to avoid unnecessary NF allocations, too frequent scaling, and instabilities due to traffic fluctuations, the switches ensure that the traffic is increasing continuously.

To do so, we instructed the switches to track three additional metrics: *the growth rate* (R_g), *growth counter* (C_g), and *decline counter* (C_d). The growth rate tracks the number of yellow packets processed in the last N_g packet interval (Fig. 3a). C_g uses R_g to track the overall trend of the slice utilization and is calculated as follows: if the number of yellow packets (the growth rate R_g) is increasing between two subsequent intervals of N packets, or the number of red packets is greater than zero, the C_g is increased by one. However, if R_g is decreasing, C_g is reset to 0. This way, if the slice utilization is continuously increasing (for at least M intervals), scaling will occur after at most $N \cdot M$ packets since the first yellow/red packet (Fig. 3a). Moreover, to detect under-load, each switch tracks C_d, increasing it by one if all the packets in an N-interval are green. Otherwise, it resets it to 0. Consequently, if C_d reaches M, meaning that for the last $N \cdot M$ packets, the slice had excess $2\Delta BW$ bandwidth, the switch will initiate the scaling-down process (by ΔBW).

Merging meter. To be able to scale-in, an NF should process less traffic than the maximum available on the other NFs implementing the same functionality. As illustrated in Fig. 3b, the CC configures the rate thresholds by calculating the maximum traffic volume that any of the other NFs can take over. To avoid too frequent scaling and instabilities, the switches track an additional metric, *the merging counter* C_m. This metric tracks the number of all-green intervals (in the same way C_d did), and, whenever it reaches M, scaling-in is initialized. Finally, the CC readjusts the thresholds (Fig. 3b).

Processing at the CC. To filter the requests belonging to the same event (e.g., scaling-up detected at multiple switches), we implemented a back-off mechanism that, every time a scaling request is received, checks if other requests were received in at least the last $N \cdot M \cdot MSS/BW_{reserved}$ seconds.

### B. Virtual link configuration, state transfer and flow rerouting

To avoid the state and route inconsistencies associated with a controller-driven approach, our framework reroutes the flows, transfers their state, and updates the resource allocations in the data-plane (while processing data-packets), as illustrated in Fig. 4. To fill in the Slice Management header, it relies on the central controller's updates (in particular, the DPC maintenance module) indicating, among other things, what flow to divert to the new NF. For example, if the controller

decided to reroute B (Fig. 4), it informs NF1 of this. NF1, after receiving the first packet of flow B forwards it to the original NF (NF2) to pick up its state (Fig. 4a). While the original packet is processed further along the path (thus avoiding unnecessary delay), a small copy is sent back to NF1, and, afterwards, NF2'. Each switch on the new virtual link updates its forwarding rules (e.g., output port stored in a register array) and resource allocation, as this packet passes through (Fig. 4b). Since it is not possible to update the bandwidth allocations in P4, we decided to generate a digest to a local digest listener, while processing this header. Consequently, this local listener issues the command to update the reservations. When this packet reaches NF1, all subsequent packets (of flow B) are diverted to NF2, and the new virtual link is established. Finally, the same process is used to create the other virtual link (connecting NF2' to the rest of the slice). Moreover, to account for packets that were already forwarded towards NF2 (before NF1's rule was updated), we instruct NF2 to send small updates to NF2', in the same way the first packet was sent, but with a different type (to prevent unnecessary forwarding rules and bandwidth allocation updates (Fig. 4d)).
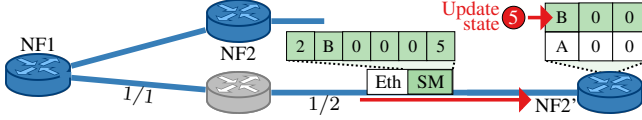
Scaling-in. To avoid the dependency on the incoming packets, we decided to change the migration procedure during scaling-in by transporting one flow's state per received packet destined for NF2'. Thus, in contrast to the example shown in Fig. 4, we use a sequential index to read all the active states of the states table and append it to the SM header. Two situations can occur. First, the data packet can belong to a flow whose state is still present at NF2. In this case, a small copy to transfer the state of the flow indicated by the sequential index is sent to NF2, while the original packet is forwarded further along the path (after updating its state). Second, the packet can belong to a flow whose state was already transferred to NF2. In this case, in addition to the state information of the flow indicated by the sequential index, we also forward the data-packet back to NF2. Upon reception, NF2 updates the state from the SM header as well as the state belonging to the data-packet. Further, while processing these transfer packets, switch NF1 adjusts its forwarding rule for the flow in the SM packet to point to NF2 (instead of NF2'). However, this rule is only put into affect after the state table was transferred (i.e., $table\_size$ packets were sent to NF2').

Difference with SwingState. In our state migration, we leverage the idea from SwingState to transport the packets in the data-plane. However, in contrast to SwingState, we avoid the dependency on the incoming traffic patterns, which could lead to a very long scaling-in process and, consequently, overhead for infrequent flows. Moreover, we combine the state transfer with rerouting and significantly reduce the number of updates that need to be sent. To do so, we assume that during horizontal scaling, NFs implementing the same function execute the same P4 program (i.e., state tables have the same size), and that the same hash function is used for index calculation (assumption not needed for SwingState). In scenarios in which this assumption does not hold, the controller can
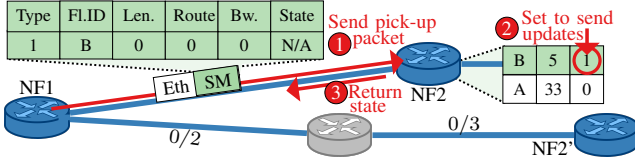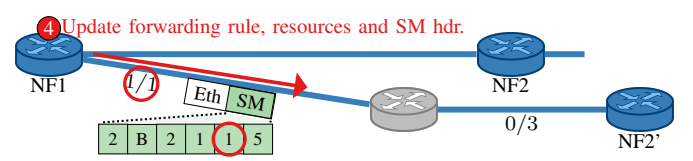
(a) After receiving flow B's packet, NF1 appends the SM header (shown in green) and forwards it to NF2 (step 1). Upon reception, NF2 creates a copy, updates the SM header (e.g., state information, source route to NF1), sets a bit register associated with B to 1, indicating that B's state is migrated (step 2), and returns the packet to NF1 (step 3).

(b) NF1 updates its forwarding rules for flow B, adjusts the allocated bandwidth for the slice to ΔBW (1, circled values, step 4). Next, it updates the SM header (e.g., source route to reach NF2') and forwards it to NF2'. All switches in the path repeat this process, updating their forwarding rule to the value from the source route and allocating the resources.

(c) Upon reception, NF2' updates its state.
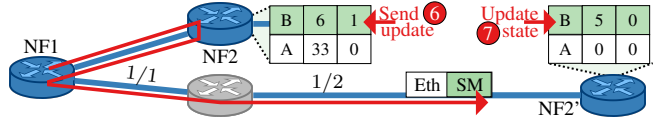
(d) If a packet of flow B is received at NF2, an update is sent to NF2'.

Fig. 4: Virtual link configuration & state management module.

provide a table index mapping to the switch transferring the state, ensuring consistency.

C. Overhead and limitations

Collisions. Indices to P4 register arrays storing the state are calculated by hashing the packet's header fields. Hence, the probability of hash collisions increases with the number of concurrent unique flows in a slice. When flows collide, P4 NFs merge their state. Hence, our framework, which relies on each NF's state management, does not distinguish between colliding flows either and will treat them as the same flow.

Memory overhead. Per state array, DPC uses two bit arrays to store active flows and transferred flows. Moreover, is uses two registers containing source routes (towards the next and previous NF), and eleven counters for load monitoring (e.g., number of red/yellow/green packets in the current and previous intervals, C_d, C_g, C_m)

Latency and packet overhead. Every time the DPC appends the SM header, it increases the packet's transmission delay. Moreover, our solution generates additional packets (e.g., state transfers, state updates). However, due to the small size of the SM header (a few bytes), this overhead is not significant and lower than other data-plane approaches (see Sec. III).

Packet reordering. Like other data-plane scaling approaches (e.g., SwingState) packet reordering can occur if packets are present at the outdated link during rerouting. While we make sure to reroute these packets, we cannot guarantee that all packets will arrive at their correct virtual node in correct order. Consequently, NFs that depend on exact packet order might have their state overwritten by an update packet. To maintain state consistency, a sequence number can be appended to each packet at the first NF, and the state only updated if the sequence number is higher than the last received one.

Hybrid scenario (only some P4-programmable switches). To use our approach, switches acting as NFs must be P4-programmable. For non-virtual nodes, traditional SDN switches can be used. The only difference would be that to adjust bandwidth reservations a packet would be sent to the CC resulting in a potential latency increase. Moreover, to avoid potential inconsistencies during rerouting, the controller would, while deploying a new NF, update the rules on SDN switches connecting this new NF to the rest of the slice.

Recirculations. During the scaling-in process, in some cases we need to read multiple indexes from the same state register array (i.e., a state belonging to the original flow and the state belonging to the flow indicated by the sequential index). For switches with a limited number of memory accesses per register array, we implement these actions using recirculations.

P4 NF deployment. All currently available programmable hardware requires a firmware reload when a new P4 program is deployed. Since this is never instantaneous, it can lead to some downtime, state loss, and service interruptions for all NFs deployed on the switch and all flows processed by it. Fortunately, various data-plane reconfiguration approaches facilitate uninterrupted reconfigurability of the data-plane [9], [23], [29], and should be used to enable dynamic NF placement. In this paper, for simplicity, the P4-program contains all the NFs, and we just update a register indicating if the NF is active or not.

III. EVALUATION

Experiment setup. To evaluate our solution, we used two topologies, shown in Fig. 5. The first one (Fig. 5a), was emulated using Mininet with the P4 software-switch (behavioural model [1]), while the second one (Fig. 5b) used an Intel Tofino switch [11]. We observed similar results with both our implementations. A notable difference was a more unpredictable latency in Mininet, presumably due to emulation. Due to space constraints, we will focus mostly on the hardware measurements.

Traffic scenarios. We considered two traffic patterns: (1) baseline scenario (Fig. 6a), used to test the scaling processes;
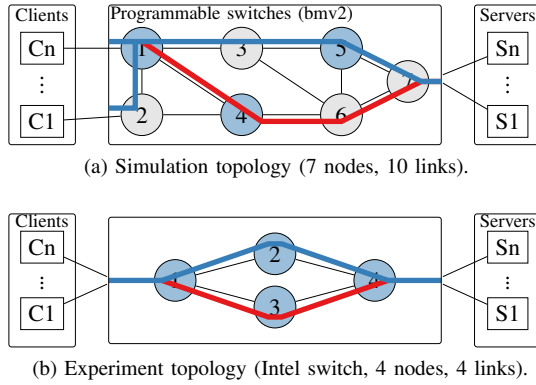
(a) Simulation topology (7 nodes, 10 links).

(b) Experiment topology (Intel switch, 4 nodes, 4 links).

Fig. 5: Topologies. Blue lines represent the initial slice, while red lines represent the path added during horizontal scaling.

(a) Baseline scenario.
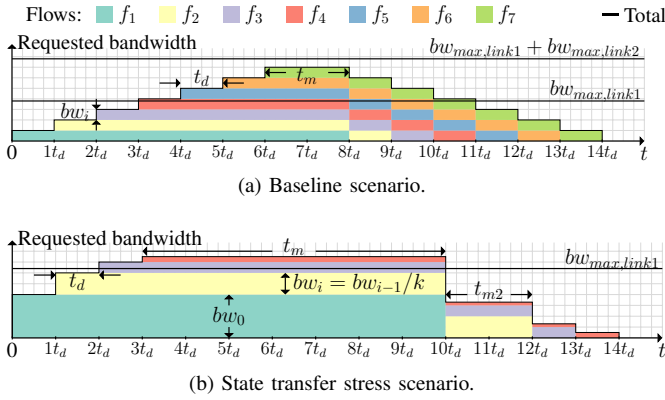

(b) State transfer stress scenario.

Fig. 6: Traffic scenarios.

and (2) state transfer stress scenario (Fig. 6a), to test the state transfer process while scaling-in, especially when some flows send packets rarely. In all scenarios, the number of TCP/UDP flows (generated using iperf3) was varied, as well as t_d, t_m, n, k. Moreover, delay t_c was added to each control-plane request/replay to account for the slower control-plane. Each experiment was run five times. Since we mainly focused on the data-plane's performance, the controller did not track the flows' bandwidths, but always rerouted the last flow that was added to the path.

Comparison baselines. Our monitoring approach has been compared to an approach that uses an SDN controller to re-configure a slice (both route and bandwidth reservations) by periodically monitoring the queuing delay/utilization of the slice (controller-driven approach). We varied the controller's monitoring delay between 1, 3, and 5 seconds, and the number of successive intervals in which an increase/decrease in queuing delay/utilization happens from 1 to 3. For the state transfer, we compared our approach to (1) SwingState [17] and (2) a controller-driven polling approach (i.e., the controller, while rerouting the flow, also polls for state information).

Performance metrics. We evaluated all our schemes on (1) the average and maximum round-trip times, (2) the average

and maximum jitter, (3) overhead caused by the state-transfer process (in bytes and packets), (4) percentage of encountered corrupt states during transfers (both temporary and at the end of the scaling process), and (5) duration of the scaling process.

A. Overall performance

Control-plane. In all our experiments, we observed that the control plane was limiting the factor in our framework. For example, the switch generates the digest notification (i.e., small notifications to the control-plane indicating the need for, for example, scaling and/or rerouting) much faster than control software can process them and adjust the parameters (e.g., meter rates, deciding which flow to reroute).

Tuning the monitoring module ($N, M, \Delta Bw$)**.** Choosing a lower N or M decreases the detection time since fewer packets need to be processed by the NF to detect overload (Fig. 7a). However, we observed that a very low N (e.g., 64) increases the number of intervals needed for detection (e.g., for $M = 5$, our framework needed 15 intervals on average for $N = 64$ compared to 5 intervals on average for $N = 256$). Moreover, when combined with a very high M, the probability of reset-ting the counter C_g, and missing the scaling event, increases. Similarly, during scaling-in/scaling-down, low values of N and M decrease the detection time (Fig. 7a). However, at the same time, we observed that the probability of scaling-in/scaling-down too quickly (or immediately after the scaling-up/scaling-out process) increases, leading to instability.

Further, we evaluated the influence of different ΔBW. A higher ΔBW decreases the number of generated scaling events, but due to the reduced granularity increases the excess bandwidth (i.e., $Bw_{res} - Bw_{req}$, Fig. 7b). Additionally, since the threshold needed to trigger scaling-down is set at $Bw_{res} - 2\Delta Bw$, the excess bandwidth per-link is usually under $2\Delta Bw$ (Fig. 7b). The only difference to this rule occurs when, due to the adaptive nature in which we assign the bandwidth (e.g., if two subsequent scaling-up events are registered, we double the ΔBw), the bandwidth is scaled higher than $Bw_{req} + 2\Delta Bw$ (Fig. 7b for $bw = 19$ and $\Delta Bw = 10$). However, if no new flow is generated, this increase is only temporary and is always followed by a scaling-down event. Furthermore, if the increased number of scaling events is combined with a very high M and/or N, by very steep bandwidth increases, the time needed to reach the needed bandwidth can be very long (and might never be reached during the duration of the scenario). Hence, we chose $N = 128$ and $M = 5$ and $\Delta Bw = 10$ as the values that provided a good trade-off between fast and stable load detection for the remainder of this paper.

Scaling-down upon competition of all flows on a path. P4 programs are executed upon packet reception. Hence, if a switch does not receive packets, the load monitoring module will not detect the last scaling-down event. Consequently, at least $2\Delta Bw$ resources will remain assigned. The only exception to this is a scaling-in event that releases all the resources assigned on a path. To avoid these situations, the central controller must detect these cases and, subsequently, release the assigned resources.
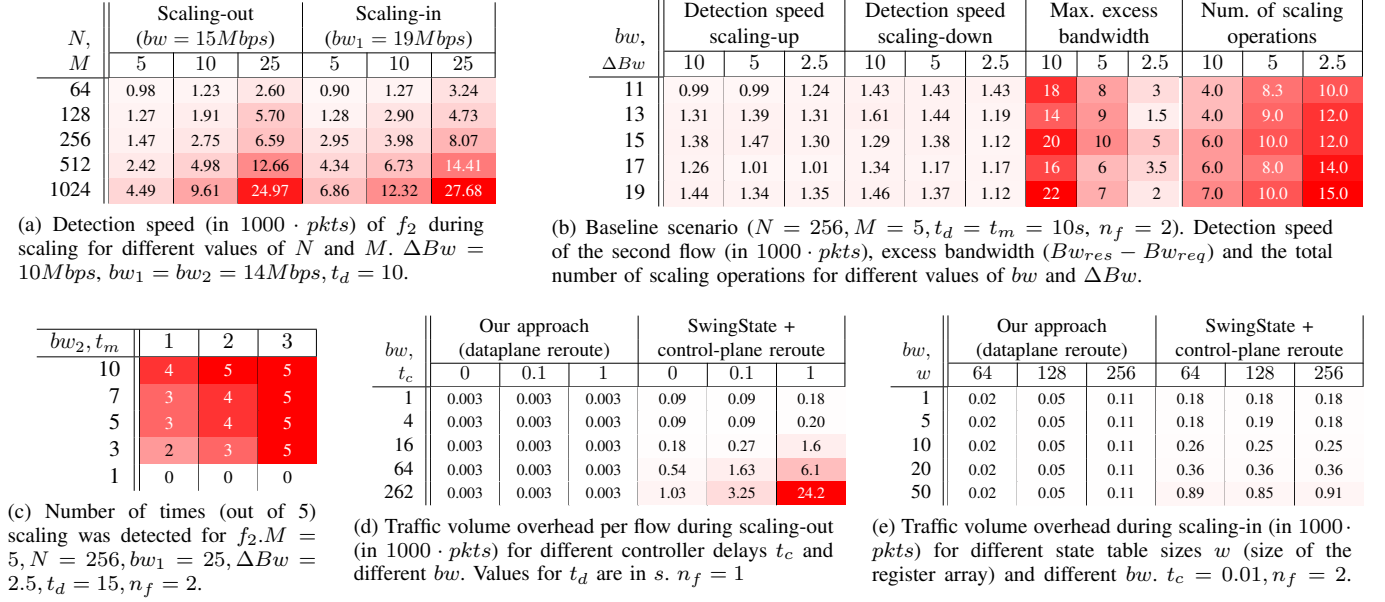
$N,$ M	Scaling-out $(bw = 15Mbps)$			Scaling-in $(bw_1 = 19Mbps)$		
	5	10	25	5	10	25
64	0.98	1.23	2.60	0.90	1.27	3.24
128	1.27	1.91	5.70	1.28	2.90	4.73
256	1.47	2.75	6.59	2.95	3.98	8.07
512	2.42	4.98	12.66	4.34	6.73	14.41
1024	4.49	9.61	24.97	6.86	12.32	27.68

(a) Detection speed (in $1000 \cdot pkts$) of f_2 during scaling for different values of N and M. $\Delta Bw = 10Mbps$, $bw_1 = bw_2 = 14Mbps$, $t_d = 10$.

$bw,$ ΔBw	Detection speed scaling-up			Detection speed scaling-down			Max. excess bandwidth			Num. of scaling operations		
	10	5	2.5	10	5	2.5	10	5	2.5	10	5	2.5
11	0.99	0.99	1.24	1.43	1.43	1.43	18	8	3	4.0	8.3	10.0
13	1.31	1.39	1.31	1.61	1.44	1.19	14	9	1.5	4.0	9.0	12.0
15	1.38	1.47	1.30	1.29	1.38	1.12	20	10	5	6.0	10.0	12.0
17	1.26	1.01	1.01	1.34	1.17	1.17	16	6	3.5	6.0	8.0	14.0
19	1.44	1.34	1.35	1.46	1.37	1.12	22	7	2	7.0	10.0	15.0

(b) Baseline scenario ($N = 256, M = 5, t_d = t_m = 10s$, $n_f = 2$). Detection speed of the second flow (in $1000 \cdot pkts$), excess bandwidth ($Bw_{res} - Bw_{req}$) and the total number of scaling operations for different values of bw and ΔBw.

bw_2, t_m	1	2	3
10	4	5	5
7	3	4	5
5	3	4	5
3	2	3	5
1	0	0	0

(c) Number of times (out of 5) scaling was detected for f_2. $M = 5, N = 256, bw_1 = 25, \Delta Bw = 2.5, t_d = 15, n_f = 2$.

$bw,$ t_c	Our approach (dataplane reroute)			SwingState + control-plane reroute		
	0	0.1	1	0	0.1	1
1	0.003	0.003	0.003	0.09	0.09	0.18
4	0.003	0.003	0.003	0.09	0.09	0.20
16	0.003	0.003	0.003	0.18	0.27	1.6
64	0.003	0.003	0.003	0.54	1.63	6.1
262	0.003	0.003	0.003	1.03	3.25	24.2

(d) Traffic volume overhead per flow during scaling-out (in $1000 \cdot pkts$) for different controller delays t_c and different bw. Values for t_d are in s. $n_f = 1$

$bw,$ w	Our approach (dataplane reroute)			SwingState + control-plane reroute		
	64	128	256	64	128	256
1	0.02	0.05	0.11	0.18	0.18	0.18
5	0.02	0.05	0.11	0.18	0.19	0.18
10	0.02	0.05	0.11	0.26	0.25	0.25
20	0.02	0.05	0.11	0.36	0.36	0.36
50	0.02	0.05	0.11	0.89	0.85	0.91

(e) Traffic volume overhead during scaling-in (in $1000 \cdot pkts$) for different state table sizes w (size of the register array) and different bw. $t_c = 0.01, n_f = 2$.

Fig. 7: Evaluation of the separate modules of the framework (4-node topology, TCP). All bandwidth values are in $Mbps$.

State transfer. SwingState transfers the state in the data-plane, but relies on an external entity (in this case, the central controller) to reroute the traffic. Hence, until the controller reroutes the traffic, SwingState continues sending updates to the newly deployed NF. Consequently, as Fig 7d illustrates, the overhead of transferring one flow during scaling-in depends on the controller delay t_c and the flow's bandwidth (how many packets are sent before the controller can react). In contrast, our approach incorporates a data-plane rerouting procedure. Hence, it depends solely on how fast the network can transport the SM header to the previous NF, which will, upon reception, update its forwarding rule (i.e., stateful register storing the output port). Furthermore, during scaling-in, our approach depends on the width of the state array and the number of flows processed by the NF. For example, if we consider the scenario in Fig. 7e, for a table size of 64, our approach sends 64 packets towards the NF to pick up the states stored in each register. However, the two flows we were transferring in this example had hashes 41 and 56. Hence, the first 40 packets were processed as usual, and no updates were created (the bit array index indicated that flows with indexes lower than 40 were not active). Next, the state for the first flow (with the index 41) was transferred. Next, for each index between 41 and 56, packets belonging to the first flow triggered an update for the first flow and were sent back to be processssed by the other NF. Finally, all packets between 56-64 triggered an update as well (since they can only belong to the flows that were already transferred). In contrast, SwingState does not have this dependency. However, it depends on the traffic pattern of the incoming packets. Hence, if an infrequent flow would need to be transferred during scaling-in, it would delay the whole process (and the controller would not be able to remove the NF).

B. Dataplane vs. controller-driver approach

Control-plane vs data-plane load monitoring. In our experiments, we observed that the controller-driven approach is unreliable. When we increased the monitoring interval (from 1 second to 3 or 5 seconds), but kept the number of successive intervals in which an increase in queuing/utilization should be observed constant (2 or 3), the TCP's congestion control mechanisms at the end-hosts kicked-in, reducing the rate and, consequently, the observed queueing delay/utilization. Thus, the controller immediately detected a decrease and concluded that the congestion was merely a consequence of a short-term fluctuation and that scaling is not needed. In contrast, if we reduced the number of successive intervals to 1, we observed that the controller detected the need for scaling too early and, consequently, oscillated between a scale-in/down and scale-out/up phase. The data-plane solution, due to the possibility of using smaller monitoring intervals (number of packets N) and the possibility of aggregating the statistics on the switch, detected the overload faster, and, consequently, maintained a lower average and maximum delay for all the flows (Fig. 8).

Very low t_d. As mentioned above, our framework was limited by the latency between the switches and the central controller. Consequently, in scenarios with a lower t_d our framework had less improvement than with higher t_d (Fig 8). Moreover, when we set $t_d < t_c$, our framework could not scale in time.

Monitoring overhead and limitations. The CC must periodically query the switches' registers. This overhead depends on the configured monitoring interval t_{mon} and is equal to $n_{vnf} \cdot t_{mon}$. Moreover, during scaling procedures, the CC could not process all the tasks within the given monitoring interval,

	Max. delay [ms]			Avg. delay [ms]			Reserved resources [Gb]			Min throughput [%]		
Slicing approach	$t_d = 1$	$t_d = 5$	$t_d = 10$	$t_d = 1$	$t_d = 5$	$t_d = 10$	$t_d = 1$	$t_d = 5$	$t_d = 10$	$t_d = 1$	$t_d = 5$	$t_d = 10$
Our approach	171.0	127.4	58.06	69.8	31.6	38.0	6.1	12.2	20.6	81.29	99.78	99.89
SwingState + Polling	185.5	186.8	186.6	105.7	63.6	54.9	6.8	14.1	23.4	70.25	90.75	99.85
Controller-Driven + Polling	226.3	189.7	186.8	107.1	62.1	53.3	6.3	13.6	23.6	67.7	90.75	99.83
No Slicing	186.8	186.8	186.8	135.9	127.2	125.5	4.8	10.4	17.4	35.66	59.10	57.99

(a) Baseline scenario. Observed QoS at the end-hosts and resource utilization.

	Corrupt [%]			Overhead [#kB]			Overhead [#pkts]			Max. Duration [s]		
Slicing approach	$t_d = 1$	$t_d = 5$	$t_d = 10$	$t_d = 1$	$t_d = 5$	$t_d = 10$	$t_d = 1$	$t_d = 5$	$t_d = 10$	$t_d = 1$	$t_d = 5$	$t_d = 10$
Our approach	0.00	0.00	0.00	0.28	0.28	0.29	12.2	12	12.4	0.001	0.001	0.001
SwingState + Polling	0.00	0.00	0.00	50.36	46.94	47.93	1398.8	1304.0	1331.5	0.1	0.1	0.1
Controller-Driven + Polling	43.0	48.1	55.2	-	-	-	-	-	-	0.1	0.1	0.1

(b) State transfer accuracy.

Fig. 8: Baseline scenario (4-node topology, TCP, $t_m = 30$, $n_f = 8$, $bw = 20Mbps$, $bw_{link1} = 100Mbps$, $t_c = 0.1$, $w = 64$.)

resulting in delays. In contrast, by offloading monitoring to the data-plane, our solution only contacted the controller in case overload was detected. This resulted in a significant reduction of this overhead to a few digest notifications per switch for each scaling-event.

State corruption. During every reroute (scaling-in or scaling-out), the controller-driven approach could not transfer the state in time, causing all rerouted flows to have an incorrect count. Moreover, while deploying the state, the controller overwrote the present state in the switches, deleting all the state information and, hence, in contrast to data-plane approaches (our framework, SwingState), it could never recover. Consequently, all packets that followed had an incorrect count ($\approx 50\%$ of the packets, since 4, flows got rerouted to the second path in Fig. 8). In contrast, our approach and SwingState maintained state consistency, with our scheme being faster (due to the offload of the rerouting procedure to the data-plane).

C. State transfer stress scenario

To test the scaling-in functionality, we configured the controller to reroute 8 of the 11 flows to the red path. After the first flow was completed, the controller initiated the scaling-in. We observed that t_c (the delay between controller and switch) limited the speed of the controller-driven approach (Fig. 9). Moreover, as previously, during each rerouting, the state was corrupted. In contrast, SwingState and our framework avoided inconsistencies, recovering from them using update packets. Moreover, low-speed flows determined the duration of this process for SwingState (Fig. 9). In our experiments, we noticed that iperf3 (which we used to generate traffic) sent bursty traffic, especially at low speeds, and would remain idle the rest of the time. Consequently, packets originating from flows with the lowest bandwidth occurred even less frequently than initially expected, and only after the last flow (with the lowest bandwidth) terminated SwingState was able to complete the scaling-in. Furthermore, since our framework was transferring only eight flows but directed 64 packets (table width size) to the NF, many packets needed to be sent back as an update (since their state was already transferred), increasing the overhead in bytes.

IV. RELATED WORK

Over the years, several network-slicing frameworks have been proposed. However, most of them focus on providing isolation between different slices and do not provide QoS guarantees inside a slice, cannot handle the problems associated with P4 NFs, and/or do not adapt to the time-varying requirements that slices may have. The slicing framework presented in [20] does dynamically scale the resources assigned to a flow. However, it relies on the network edge to detect when scaling needs to occur and only supports vertical scaling.

NFV frameworks. Depending on how the state is organized, stored, and accessed, the different NFV scaling solutions can be divided into local (e.g. [4], [6], [19], [21], [24]), remote (e.g. [12]), and distributed approaches (e.g. [8], [22], [28]). Remote approaches store all the state remotely at some centralized storage and can thus not be used with P4 NFs, since they would impose significant performance penalties per processed packet (to retrieve the state). Local approaches never migrate the state and can, therefore, not deal with scenarios in which an increase of the flow's volume causes NF overload. Additionally, they must wait for all flows present on an NF to finish before shutting it down, resulting in inefficient resource utilization. Furthermore, most NFV frameworks were not designed with P4 NFs in mind. The ones that were, such as P4NFV [10], use a controller-driven approach and will, consequently, suffer from all aforementioned issues associated with that approach.

Data-plane state migration. SwingState [17] depends on the arrival pattern of the incoming packets and can therefore have a long transfer time. LODGE [25] targets distributed network applications by creating a shared network state. SNAP [2] and U-HAUL [16] move only the state of long-lived flows. Thus, these approaches are not well suited for this paper's objective.

V. CONCLUSION

This paper has presented an elastic network-slicing framework for P4-programmable network devices. The framework has a hierarchical design, focusing on both the control and data-planes. With its global overview of the network, the control-plane guides the data-plane behavior, but offloads all

	Corrupt [%]			Overhead [#kB]			Overhead [#pkts]			Max. duration [s]		
Slicing approach	$t_d = 1$	$t_d = 5$	$t_d = 10$	$t_d = 1$	$t_d = 5$	$t_d = 10$	$t_d = 1$	$t_d = 5$	$t_d = 10$	$t_d = 1$	$t_d = 5$	$t_d = 10$
Our approach	0.00	0.00	0.00	55.68	55.68	55.68	45	45	45	0.07	0.07	0.07
SwingState	0.00	0.00	0.00	10.48	11.45	12.5	291.2	318.3	347.4	7.3	42.2	86.5
Polling	5.62	5.95	6.14	-	-	-	-	-	-	0.1	0.1	0.1

Fig. 9: State transfer scenario for the 4-node topology, Overhead during scaling-in. $t_m = 120$, $n_f = 11$, $bw = 1kbps$, $k = 4$, $bw_{link1} = 1.32Gbps, t_c = 0, w = 64$.

time-sensitive tasks, such as overload (under-load) detection, rerouting, and state transfer to the fast data-plane. Consequently, at time-sensitive moments, the data-plane component can perform tasks autonomously and with limited input. Finally, this paper has demonstrated that offloading time-sensitive tasks to the data-plane can increase slice resource efficiency, while minimizing scaling time and maintaining state consistency, especially when compared to state-of-the-art controller-driven approaches.

REFERENCES

[1] P4 behavioral model. https://github.com/p4lang/behavioral-model. Accessed: 19-03-2018.

[2] ARASHLOO, M. T., KORAL, Y., GREENBERG, M., REXFORD, J., AND WALKER, D. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), pp. 29–43.

[3] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (July 2014), 87–95.

[4] FAYAZBAKHSH, S. K., SEKAR, V., YU, M., AND MOGUL, J. C. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking* (2013).

[5] FOUKAS, X., PATOUNAS, G., ELMOKASHFI, A., AND MARINA, M. K. Network slicing in 5g: Survey and challenges. *IEEE Communications Magazine 55*, 5 (2017), 94–100.

[6] GEMBER, A., ANAND KRISHNAMURTHY, S. S. J., GRANDL, R., GAO, X., ANAND, A., BENSON, T., AKELLA, A., AND SEKAR, V. Stratos: A network-aware orchestration layer for middleboxes in the cloud. corr (2013), 2013.

[7] GEMBER, A., PRABHU, P., GHADIYALI, Z., AND AKELLA, A. Toward software-defined middlebox networking. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks* (2012), pp. 7–12.

[8] GEMBER-JACOBSON, A., VISWANATHAN, R., PRAKASH, C., GRANDL, R., KHALID, J., DAS, S., AND AKELLA, A. Opennf: Enabling innovation in network function control. *ACM SIGCOMM Computer Communication Review 44*, 4 (2014), 163–174.

[9] HANCOCK, D., AND VAN DER MERWE, J. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies* (2016), ACM, pp. 35–49.

[10] HE, M., BASTA, A., BLENK, A., DERIC, N., AND KELLERER, W. P4nfv: An nfv architecture with flexible data plane reconfiguration. In *2018 14th International Conference on Network and Service Management (CNSM)* (2018), IEEE, pp. 90–98.

[11] Intel® Tofino™. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html. [Online; accessed 03-November-2020].

[12] KABLAN, M., ALSUDAIS, A., KELLER, E., AND LE, F. Stateless network functions: Breaking the tight coupling of state and processing. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)* (2017), pp. 97–112.

[13] KHALID, J., GEMBER-JACOBSON, A., MICHAEL, R., ABHASHKU-MAR, A., AND AKELLA, A. Paving the way for {NFV}: Simplifying middlebox modifications using statealyzr. In *13th {USENIX} Sympo-sium on Networked Systems Design and Implementation ({NSDI} 16)* (2016), pp. 239–253.

[14] KIM, C., BHIDE, P., DOE, E., HOLBROOK, H., GHANWANI, A., DALY, D., AND HIRA, MUKESH AMD DAVIE, B. In-band network telemetry (int), 2016. https://p4.org/assets/INT-current-spec.pdf, Last accessed on 10-06-2020.

[15] KREUTZ, D., RAMOS, F. M., VERISSIMO, P., ROTHENBERG, C. E., AZODOLMOLKY, S., AND UHLIG, S. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE 103*, 1 (2015), 14–76.

[16] LIU, L., XU, H., NIU, Z., WANG, P., AND HAN, D. U-haul: Efficient state migration in nfv. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems* (2016), pp. 1–8.

[17] LUO, S., YU, H., AND VANBEVER, L. Swing state: Consistent updates for stateful and programmable data planes, 2017.

[18] ORDONEZ-LUCENA, J., AMEIGEIRAS, P., LOPEZ, D., RAMOS-MUNOZ, J. J., LORCA, J., AND FOLGUEIRA, J. Network slicing for 5g with sdn/nfv: Concepts, architectures, and challenges. *IEEE Communications Magazine 55*, 5 (2017), 80–87.

[19] PALKAR, S., LAN, C., HAN, S., JANG, K., PANDA, A., RATNASAMY, S., RIZZO, L., AND SHENKER, S. E2: a framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 121–136.

[20] POLACHAN, K., TURKOVIC, B., PRABHAKAR, T., SINGH, C., AND KUIPERS, F. A. Dynamic network slicing for the tactile internet. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)* (2020), IEEE, pp. 129–140.

[21] QAZI, Z. A., TU, C.-C., CHIANG, L., MIAO, R., SEKAR, V., AND YU, M. Simple-fying middlebox policy enforcement using sdn. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM* (2013), pp. 27–38.

[22] RAJAGOPALAN, S., WILLIAMS, D., JAMJOOM, H., AND WARFIELD, A. Split/merge: System support for elastic execution in virtual mid-dleboxes. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)* (2013), pp. 227–240.

[23] SAQUETTI, M., BUENO, G., CORDEIRO, W., AND AZAMBUJA, J. R. Virtp4: An architecture for p4 virtualization. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2019), IEEE, pp. 75–78.

[24] SEKAR, V., EGI, N., RATNASAMY, S., REITER, M. K., AND SHI, G. Design and implementation of a consolidated middlebox architecture. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)* (2012), pp. 323–336.

[25] SVIRIDOV, G., BONOLA, M., TULUMELLO, A., GIACCONE, P., BIANCO, A., AND BIANCHI, G. Lodge: Local decisions on global states in prograramaable data planes. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)* (2018), IEEE, pp. 257–261.

[26] TURKOVIC, B., KUIPERS, F., VAN ADRICHEM, N., AND LANGEN-DOEN, K. Fast network congestion detection and avoidance using p4. In *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies* (2018), pp. 45–51.

[27] VASSILARAS, S., GKATZIKIS, L., LIAKOPOULOS, N., STIAKOGIAN-NAKIS, I. N., QI, M., SHI, L., LIU, L., DEBBAH, M., AND PASCHOS, G. S. The algorithmic aspects of network slicing. *IEEE Communications Magazine 55*, 8 (2017), 112–119.

[28] WOO, S., SHERRY, J., HAN, S., MOON, S., RATNASAMY, S., AND SHENKER, S. Elastic scaling of stateful network functions. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)* (2018), pp. 299–312.

[29] ZHANG, C., BI, J., ZHOU, Y., AND WU, J. Hypervdp: High-performance virtualization of the programmable data plane. *IEEE Journal on Selected Areas in Communications 37*, 3 (2019), 556–569.