Flow setup latency in SDN networks

Ramin Khalili, Zoran Despotovic, Artur Hecker

***Abstract*—In Software-Defined Networking (SDN), the typical switch-controller cycle, from generating a network event notification at the controller until the flow rules are installed at the switches, is not an instantaneous activity. Our measurement results show that this has serious implications on the performance of flow setup procedure, specifically for larger networks: we observe that, even with software switches, the flow setup latency for networks of around 500 switches is in order of 50 milliseconds, with 99th percentile exhibiting 10 times higher latencies. To reduce both the latency and the variance of the flow setup, we propose path aggregation strategies, which turn the network into a set of pre-configured pipes that connect any pair of nodes. Our approach radically simplifies the flow setup procedure by minimizing the set of switches to be updated for new user-initiated flows to a constant number. We implement our solution in our testbed and study its performance through measurements. The results show that, in similar settings, it reduces the median and 99-percentile latencies to 5.9ms and 7ms, respectively, significantly improving the performance, especially in the tail.**

***Index Terms*—SDN, Scalability, Latency**

I. Introduction

Conventional network devices embed dedicated, closed software and are characterized by long and costly evolution cycles. Such devices are doomed to complex tradeoffs, such as feature-set richness vs. ease of use and costs. SDN reduces network devices to programmable flow forwarding machines, termed Switches, while the network logic runs in a logically centralized sooftware platform termed Controller [1][2]. The latter collects the network state and exposes it to control applications that make forwarding decisions, while the former encodes and stores these decisions in form of flow rules to apply to incoming flows [3][4].

The SDN research so far mainly addressed design and performance of SDN controllers [5][6][7][8][9] and switches [10][11][12]. A comprehensive evaluation of the entire system, comprising both a controller and a network of switches, is however still missing. This paper fills this void by reporting on a set of experiments performed to understand the end-to-end performance of SDN networks. Although we frame our discussion below in the context of operator networks and specifically use OpenFlow SDN as an example, the conclusions are general and hold for the SDN as a concept.

A. Problems with current SDN deployments

Our tests build on the observation that the typical switch-controller communication cycle, from the controller notification generation (so-called packetIn message in the OpenFlow protocol) to the flow rule installation, is never an instantaneous activity, even if the packetIn processing time at the controller and the controller-switch link latencies are negligible. Instead, it takes a variable and non-negligible time that depends on the changing operating conditions of the switch such as its flow table occupancy, rule complexity and priority, etc.

Even as a minor issue as it may at first sound, this creates serious implications for flow path setup, the most typical task in SDN requiring updates of a set of switches. Normally, the time to install a flow rule along a path of several hops roughly corresponds to the largest of the individual switch times. During this time however, the path is partially established, and there is a non-negligible possibility that a flow packet could be forwarded along this partially established path, triggering a table-miss event on some intermediate switches along the path. To deal with this table-miss event at intermediate switches, the controller needs to know the path assigned to the flow, to be able to enforce packet forwarding on an appropriate outport of the switch. If such information is not available at the controller, it can do no better than to either drop the packet or to flood it over all outports of the switch. Both solutions result in undesirable performance and scalability issues.

Our measurements over a Floodlight/Mininet testbed show that in a small SDN network with around 50 switches, setting up a path takes about 6ms in the median and 11ms in the 99th percentile. However, for a larger network with approximately 500 switches, these times grow to 50ms and 490ms respectively, when flooding is used to deal with packetIn from intermediate switches; and to 16ms and 36ms, when the controller knows the path and forwards the packets over the corresponding outport. Note that the reported values are for one way flow setup latency. In any case, this latency is hardly acceptable in networks that have tight latency requirements, in orders of few milliseconds, such as data centers [13][14] and 5G [15], which provides a strong hint on performance and scalability limits of a straightforward SDN deployment.

B. Our solution

Insights collected from our tests sketch ways, how one can cut down these times. One way is to reduce the individual switch update times and their variations, including across different defined states and optimizing for different hardware platforms. This lies in the focus of the current literature (see §II). Note however that software switches, such as OpenVSwitch (OVS), studied here, have generally very small update times compared to hardware switches [14][16], and that even for such fast operating switches the flow setup latency is very high as shown by our measurements. Hence, reducing the update times of switches is not enough to address

Manuscript received March 10, 2018; revised August 10, 2018; accepted August 20 , 2018. This work was supported in part by the EU H2020 Programme (H2020-ICT-2018-1) under Grant Agreement No. 815279 (5G Verticals INNovation Infrastructure "5G-VINNI"). (Corresponding author: Ramin Khalili).

R. Khalili, Z. Despotovic, and A. Hecker are with Huawei Technologies, Munich 80992, Germany. (E-mail: {ramin.khalili, zoran.despotovic, artur.hecker}@huawei.com).

this problem. Another way is to reduce the number of switches to update, when setting up a path in the network, which we study in this paper. Specifically, we advocate for the edge/core separation idea, where the fine-grained flow management is performed at the edge switches, and the core is configured as a fabric that interconnects these edges.

To configure the fabric, we apply our path aggregation solution proposed in [17], which turns the network into a set of pre-configured pipes that connect any pair of edges. Using this core fabric, flow management becomes simpler and more scalable, as it suffices to maintain only end-points per flow: indeed, the end-to-end flow setup degrades to a simple update of a minimal switch set, thus reducing the flow setup latency and its variance. The measurement results show that our solution provides flow setup latencies below 8ms, even in the largest setting. We further show how more complex tasks, such as user mobility and dynamic topology handling, can be efficiently supported within our solution. Our implementation in Floodlight proves that it is relatively straightforward to realize our solution in a state of the art SDN controller.

C. Main contribution

The main contributions of this article are the demonstration of the performance and scalability issues of the current SDN deployments; the investigation of the underlying causes of such problems; the introduction, implementation, and performance analysis of the edge/core separation solution; and its extension to support network dynamics, together with the implementation and performance analysis of these extensions.

The article starts with the discussion of related work in §II. §III presents the flow rule installation time for an individual switch. In §IV, we study the flow setup latencies in SDN networks, using OpenFlow SDN as a concrete example. §V introduces our proposal and presents the corresponding results. §VI concludes the paper.

II. Related Work

A number of recent surveys introduced useful SDN taxonomies and defined the relevant SDN terms [1], [2]. Our terms are consistent with the definitions in [1]. Performance evaluation of SDN has focused on understanding performance and scalability limits of individual constituent elements of its architecture, rather than seeing a network as a whole. Further, existing studies typically made distinction between data and control plane performance. To this end, switch design and performance evaluation of the data plane have received a lot of attention [11][18][12]. As for the control plane, we here distinguish controller and switch performance when executing control plane steps such as sending a network event to the controller, determining appropriate flow rules and installing those in a (set of) switch(es). The main focus of the SDN research has been on controller design and performance in this process [5][6][7][13][19].

At the same time, the performance aspects that pertain to the SDN switch, e.g. how fast it can reply to commands from the controller, are just getting in focus of SDN research. A few recent papers report on the performance of the control plane of various SDN hardware switches [14][20][21][22]. In particular, they measure the latencies of the basic control plane operations such as event generation ("inbound latency") and flow rule installation, modification and deletion ("outbound latency"). Specifically, the flow rule installation latency ranges from few to tens of milliseconds depending on the number of the rules installed in the switch, the priority of the rule to insert etc. Besides, there are significant differencies in latency trends across switches with different chipsets and firmware. Our paper builds directly on the insights from these measurements, checking their implications for the network-wide view, i.e. for the path setup. As we use software switches (OpenVSwitch), we first establish equivalent claims for these. We then extend these measurements to a network-wide view.

The literature discusses possible negative effects of the switch diversity. [16] proposes to reduce such effects at the source of the problem, i.e. in the switches, introducing new methods to efficiently manage switch TCAM to provide switch performance guarantees. [22] relies on real-time probing to assess the state of a switch and to make it available to control applications, before they make their control decisions; hence, is closest to our work. However, it evaluates the performance of the control plane only and fails to provide any end-to-end performance values. Finally, [17] proposed a path aggregation method to reduce the flow table sizes at the core switches, which we use in this paper. However, [17] only describes how to create the fabric and provides no end-to-end solution and no flow setup latency evaluation.

III. Single-Switch Flow Setup Measurements

We first consider the simplest scenario, a network with a single switch S. The users are connected to different ports of this switch and communicate with each other. When a new flow arrives, generated by a user and destined to another user, the first packet of the flow triggers a table-miss event at S. The resulting packetIn message is processed at the controller by a control application, and the corresponding flow rule is installed at S. Within our path setup application, we derive a suitable output from the source and destination IP addresses.

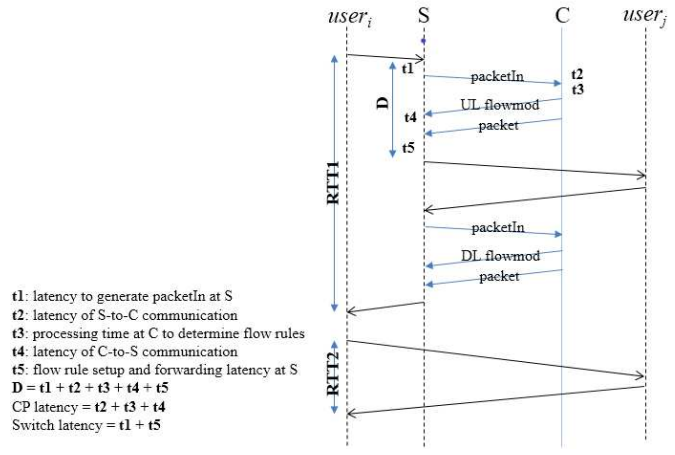

Fig. 1. We measure D indirectly by measuring RTTs observed by the first and the second ping packets, where $D = \frac{RTT1-RTT2}{2}$.

The time difference between the moment of the table-miss event and the moment, when the packet is forwarded over the corresponding outport is referred to as flow setup latency D. This latency is the sum of the control plane latency and the latency at the switch (cf. Fig.1). In our definition, the control plane latency is the time, when a packetIn is sent by the switch to the controller, until the related flow rule is received by the switch. The switch latency is the sum of two terms: the first term is the latency to generate a packetIn message upon receiving the first packet of a new flow; and the second term is the time to set up a flow rule upon receiving the rule from the controller. We test this scenario in our Floodlight [6]/Mininet [23] testbed, running FloodLight v1.1, OpenFlow v1.4 [24] and OpenVSwitch v2.3 [10]. The Floodlight controller runs on a server with 48 Intel CPUs and 24x 32GB of RAM (Huawei RH2288H), hence it is not a bottleneck in our study. We run Mininet on a separate server, with a similar configuration, and interconnect both servers via an otherwise unused Gigabit Ethernet link.

To measure D, users exchange ping flows (i.e. they send ICMP echo request and ICMP echo reply messages, simply called ping request and ping reply). Upon receiving a packetIn at the controller, due to the reception of the first ping-request generated by user i toward user j at S, the controller sets up the uplink flow rule at the switch and forwards the packet over the corresponding outport of the switch to j (cf. Fig.1). Once the ping-request is received by j, a ping-reply message is generated by j to i, which in turn generates a packetIn message to the controller, and the controller therefore installs the downlink flow rule at S and forwards the ping-reply message over the related outport of S to i. We measure the difference between the round trip times (RTT) seen by the first (RTT1) and the second (RTT2) packets of a flow, i.e. the first and second ping requests, which gives us an estimation of the sum of the setup latencies of uplink and downlink of the flow. (By subtracting RTT2 from RTT1, we remove the latency terms due to the transmission and propagation delays over the data plane from our estimation). Dividing the result by two we obtain an estimate of D: $D = \frac{RTT1-RTT2}{2}$.

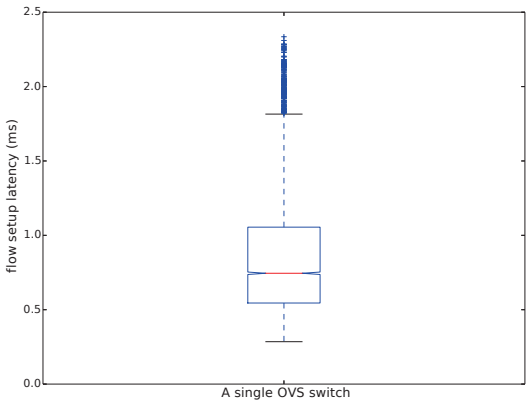

Fig. 2. The flow setup latency on a single switch.

We measure D, where new ping flows arrive according to a Poisson process with rate 1000 requests/sec. No forwarding rule is initially installed at the switch, hence, to set up a flow, the controller only needs to add new rules. No rule deletion is performed by the controller, unless stated otherwise. All the rules have similar priority and match on the source and destination IP addresses of the flows. Source and destination of the generated flows are selected randomly. Each experiment has a duration of 10 seconds. We run 20 independent experiments and collect all the measurements together. We report the measurement results for D in Fig.2. We observe that the flow setup latency is not negligible and has a random distribution with median around 0.75 ms and with a large dispersion (e.g. its 99-percentile is around 2ms).

IV. Flow Setup Latency in SDN Networks

In this section, we study flow setup latencies in SDN networks, starting with a simple linear topology.

A. Linear topology

We first consider a linear topology composed of N switches, $S_1, S_2, \cdots, S_N$, with the users attached to S_1 and S_N. Flows are generated by the users at S_1 towards the users at S_N. As in the previous case, we generate ping traffic among the users with Poisson arrivals with rate 1000 requests/sec. To set up a flow, the controller needs to configure all the N switches over the path as depicted in Fig.3. When S_1 receives a ping request, if there is no flow rule that matches the packet, it generates and sends a packetIn message to the controller. The controller determines the necessary rule for each switch along the path and pushes these rules to the corresponding switches. The ping request will therefore be forwarded to the destination. A similar process will be performed for the ping-reply. The configurations of the switches are performed in parallel by the controller; hence, one may expect similar flow setup latency as in the single switch case. Our measurement results, however, show that the flow setup latency in this setting is much higher than for the single switch case, especially in the tail of the distribution. We report the results in Fig.4 for $N = 4$ and $N = 8$. According to our results, median latencies are 1.22ms and 1.8ms for $N = 4$ and $N = 8$, respectively, and the corresponding 99-percentiles are 3.7ms and 12ms.

The observed increase in network-wide flow setup latencies can be attributed to the following reasons:

(R1) - increase of the control plane latency: First, the controller application needs to perform a more complex computation to calculate a path than in a single switch case. Second, the number of flow setup commands (in OpenFlow: OFP_FLOW_MOD, in the following simply "flowmod") sent by the controller increases to N, which increases the control plane latency. Note that the exact mechanism is quite complex: the control plane latency depends on how Floodlight processes the incoming packetIns, how TCP schedules the transmissions of small flowmod messages, and how the device manager and other modules of Floodlight interact with our application. We refer to [25] for a deeper analysis of the processing latency specifically in the Floodlight controller.

(R2) - maximization effect: As the controller sets up the whole path on the reception of the initial packetIn, the overall process is not sequential. In other words, a path is established,

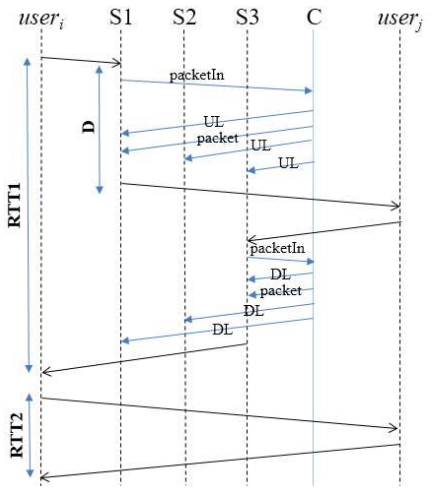

Fig. 3. Flow setup process for a path composed of three switches. This example depicts an ideal situation, where the flow rule at S2 (resp. S3) is installed before receiving UL packet from S1 (resp. S2), and where the flow rule at S2 (resp. S1) is installed before receiving DL packet from S3 (resp. S2). As we discuss later in this section, this is not always the case in practice.

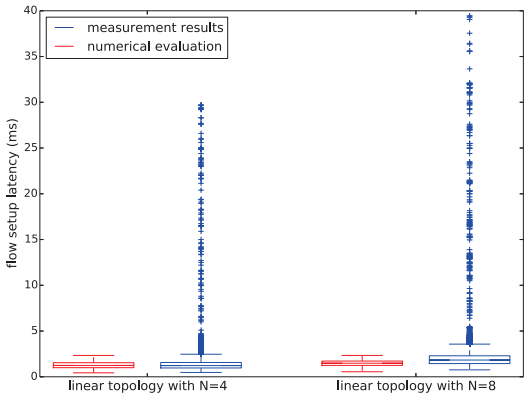

Fig. 4. The flow setup latency on a linear topology of 4 and 8 hops.

when all the switches over the path are configured. We are not aware of any dedicated mechanisms in place in barebone Floodlight controller to serialize these actions. Rather, the new flow rules are calculated and distributed to all relevant switches on the calculated path over the available point-to-point TCP control channels. Indeed, in our measurements we observe an increase of the median of the flow setup latencies. This effect could be modeled by representing the individual switch latencies as a set of i.i.d. random variables with distributions similar to what is depicted in Fig.2. Let X_i be the latency of configuring switch i over the path, then the flow setup latency of a path of length N is

$$D(N) = \max\{X_1, X_2, \cdots, X_N\}. \quad (1)$$

In other terms, we have $P(D(N) \leq l) = \prod_i P(X_i \leq l)$. This would imply that the distribution of $D(N)$ is stretched towards larger values of Xs, corresponding to the observation. Using (1), we can numerically evaluate the distribution of $D(N)$.

Note that the i.i.d. assumption might not hold in general. Moreover, the distribution of X_i, for $i > 1$, might not exactly follow the same distribution as X_1, as the controller does not need to send the original packet to all the switches along the path but to $S1$ only. Hence, this numerical evaluation only provides an estimation of $D(N)$. We observe that despite all these reservations, (1) can provide a good approximation of the median, as shown in Fig.4 for $N = 4$ and 8. We can see that the medians are 1.25ms and 1.55ms for $N = 4$ and 8, respectively, and hence are close to the median observed through measurements (1.22ms and 1.81ms, respectively). It however fails to predict the tail, as the tail is mainly a result of the effect of partially established paths, as explained below.

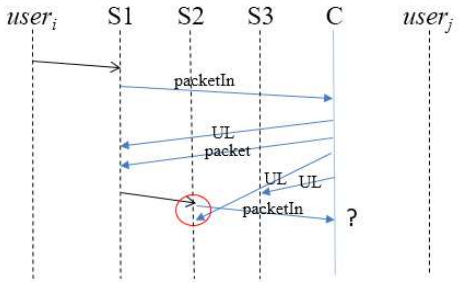

Fig. 5. An illustration of the effect of partially established paths for a line with three switches and for the uplink flow setup procedure. As the uplink forwarding rule is not yet installed at S2, the switch needs to generate a packetIn to the controller, requesting for appropriate action.

(R3) - effect of partially established paths: The flow setup process is not instantaneous, as there is a gap between pushing forwarding rules to a switch set and the time that the entire path is established. During this gap of random duration, the path is partially established, and there is a non-negligible possibility that a flow could be forwarded along it triggering a table-miss event on some of the switches along the path.

Fig.5 illustrates this situation for three switches in a line and the uplink flow setup. Assume that the flow rule installation at S_2 is delayed, either due to the control plane delay or due to some switch delay, and, hence, the user flow arrives at S_2 before the corresponding rule is installed. S_2 then generates a repetitive packetIn to be processed by the controller. Our results show that such events are not rare, even in such a small setting. Specifically, for the scenario with $N = 4$, 200 out of 2800 flows have experienced this problem. The ratio is 600 out of 2800 flows for $N = 8$. The setup latencies for the flows that are affected by this are high, explaining the long tail we observe in our measurements.

Receiving such packetIn, the controller needs to determine an appropriate action. The default "flooding" solution used by Floodlight is our first baseline: Floodlight divides the set of switches in the network into two disjoint sets, the set of edge switches and the set of non-edge switches. The set of edge switches is defined by Floodlight as the set of switches that are directly connected to the users (i.e. the first and the last switches in this example). The rest of switches in the network belong to the set of non-edge switches (i.e. switches 2 to $N-1$). When a packetIn is received from a non-edge switch, the controller simply floods the packet over all outports of the switch, exempting the inport. A more advanced solution, which we use as our second baseline, is to assume that the controller

stores flow information, i.e. the sequence of switch-port pairs of the path that the flow ought to take. Receiving a packetIn from a non-edge switch, the controller uses this information to determine the outport at this switch. This solution has the advantage that it can prevent flooding. Its drawback is the requirement to store path information for each individual flow in the controller, which might be impractical in large settings.

For this linear setting, both baseline solutions eventually result in the same action: send the packet over port 2, if the packet is received over port 1, and vice versa. The measurement results shown in Fig.4 hold for the second baseline.

B. Hierarchical topology

We now study the flow setup latency in larger and more complex settings. Specifically, we use network topologies generated from [26] [27], representing backhauls of carrier networks. They resemble k-array fat-trees, which are also used in data centers [28]. They are hierarchical and consist of three layers: access, aggregation, and core. They contain K switches at the core, K pods of size K switches, e.g a total of K^2 switches, at the aggregation, and K^2 switches at the access. $K/2$ switches of each aggregation pod are connected to the access, with five access switches per each of these switches, and $K/2$ of them are connected to the core, each of which is connected to two core switches. The degree of connectivity in the core and within aggregation pods is set to 3.

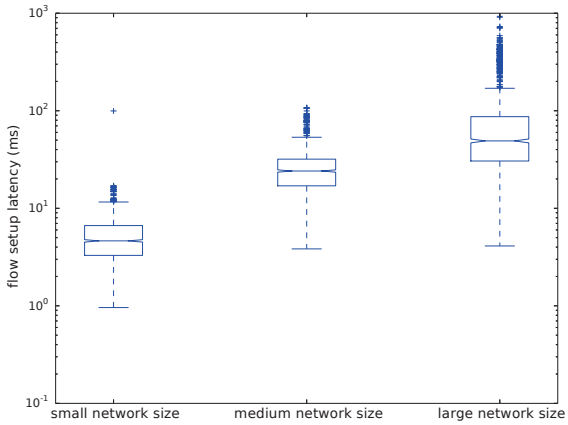

Fig. 6. The flow setup latency using the first baseline solution, which uses flooding to deal with packetIn received from non-edge switches.

We consider three different topologies: a small topology ($K = 4$) with 40/16/4 acc/agg/core switches; a medium size topology ($k = 8$) with 160/64/8 acc/agg/core switches; and a large topology ($k = 12$) with 360/144/12 acc/agg/core switches. There are 800 users attached to the access switches. Flows arrive according to a Poisson process with parameter 1000 requests/sec. The source-destination pairs are selected randomly, under the constraint that their access switches are not connected to the same aggregation pod. Hence, their paths should cross at least one core switch and four aggregation switches: the flow path lengths are minimum 6 hops and maximum as many hops as the network diameter, which is 7, 8, and 9 for the small, medium and large topologies, respectively.

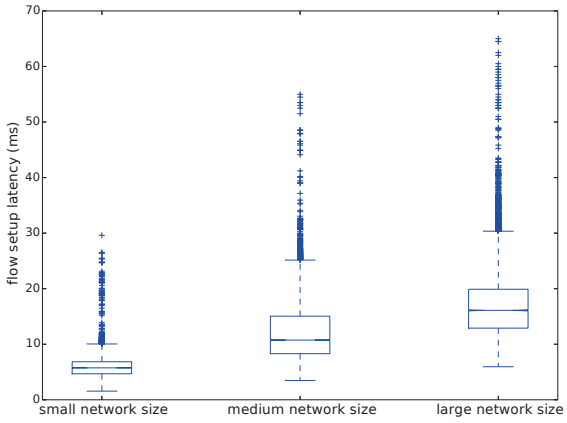

Fig. 7. The flow setup latency when the second baseline solution is applied.

In Fig.6, we present the measurement results for different network sizes, when the first baseline solution is applied. Clearly, the performance is very bad (y-axis in this figure is logarithmic). This is due to the packet flooding in the network, i.e. packetIns from non-edge switches (refer to **R3**). Fig.7 presents the performance of the second baseline solution. We observe that the flow setup latencies are significantly smaller that those observed in Fig.6. In particular, the median and 99-percentile latencies are 5.8ms and 11ms for the small topology, 10.8ms and 26.8ms for the median size topology, and 16.1ms and 36.5ms for the large topology. Even so, these results show that none of these solutions can efficiently mitigate problems arising from **R1**, **R2**, **R3**, and none of them is scalable.

In §V, we proposes a solution that mitigates all these problems by minimizing the number of switches to be updated for a new flow. Before that, we provide a short discussion about other possible solutions to mitigate performance issues arising from **R3** using the current SDN model.

C. Discussion

To mitigate the effect of partially established paths, one may delay the transmission of the first packet of the flow, until the path is fully established. For instance, the controller could estimate the time that it takes for the full path setup and delay the transmission of the first packet by this time. A real-time estimation of this delay time is, however, difficult to obtain. Another possibility is to send the path commands in the reverse way using e.g. OpenFlow bundles [24], which result in explicit confirmations by switches. The controller could send the flow setup to all but the original requestor switch and wait for the confirmations before setting up the first switch and sending out the original packet from it. Yet, both solutions penalize all flows, which is especially bad for short or single-packet flows, and, hence, not of interest for us. We should also note that our baseline solutions in this section also send the path setup in the reverse order, but do not use OF bundles.

An alternative is to ignore packetIn from intermediate switches at the controller. Doing so results in packet loss, because switches do not keep the original user packet after they send packetIn to the controller [24]. Buffering packets

comes with a considerable complexity increase at the switch and does not always work: e.g. buffer could be full [29], resulting again in packet loss. The end users can recover from such packet losses using retransmission schemes such as TCP. This, however, does not solve the effect of partially established paths but delegates it to the end users. Besides, these solutions, if applicable, can only mitigate the effect of the performance problems caused by **R3** but do not provide any solutions for the performance problems due to **R1** and **R2**.

V. Core as a Fabric

We argue that in many scenarios, the fine-grained flow management should be left over to the edge (or access) switches, while the core should perform as a fabric that interconnects these edge switches. For example, backhauls of mobile networks and the transport networks of data centers have hierarchical topologies. Here, the traffic load in the core switches is by several magnitudes higher than in the access switches: e.g. a typical number of users attached to an access switch in a carrier network would be around 1000, which, with each user generating 10 flows on average, results in a total of 10000 flows per access switch. In contrast, the number of flows crossing a core switch in a carrier network could easily surpass millions of flows [30].

We therefore advocate for the edge/core separation. In our view, the edge is a set of *logical switches* that are directly connected to either the users, or middle-boxes, where the network functions reside [31]. We see the transport/core switches as interconnecting such edge switches. In our solution, the core switches are proactively configured to provide connectivity among all edge switches. The individual flows are processed at the ingress edge, from where they are forwarded through pre-configured fabric pipes to the egress edge switches.

We configure the fabric using the Access Switch Classification (ASC) algorithm from [17]. It assigns to each edge switch i a *multi-layer classification vector*, ID, of the form

$$ecv^i = (v^i(1), v^i(2), \cdots, v^i(L))$$

with $v^i(j)$, $1 \leq j \leq L$, being 1, if there is a "routing" path crossing directed arc j that ends in edge node i, and 0 otherwise. L is the size of the vector, as determined by the algorithm. The set of paths in the network is determined by the controller. ASC does not impose any constraints on the routing decision of the controller, but uses the routing information from the controller to assign IDs to the edges. It performs a multi-layer classification of edge switches, where any edge can belong to a number of groups. Let N_e be the number of paths crossing a directed arc $e \in E$, the set of all directed arcs in the network. ASC groups all edge switches that are destinations of the paths crossing $e \in E$ with $N_e > 1$ and repeats this for all such edges. The classification vector of each access switch is determined from this grouping. This information is also used to encode flow rules in the core switches.

With ASC edge addressing, the core network becomes a static fabric that delivers any packet arriving at one of its inports to an appropriate outport without any further action. ASC achieves the minimum internal state at core switches

Algorithm 1: AASC algorithm

Input: routing and graph information, limit L;
Extract grouping information and construct the set of all these groups, named $\mathcal{G}$;
Calculate number of rules per switch, set $k = 1$;
while $\mathcal{G} \neq \emptyset$ *or* $k \leq L$ **do**
  Find c, switch with largest number of rules;
  Find G, largest group in $\mathcal{G}$ with source index c;
  Set $v(k) = 0$ for $a \in G$ and $v(k) = 1$ for $s \in G^C$;
  Remove G and all other elements equal to it from $\mathcal{G}$;
  Recalculate number of rules in c;
  Set $k = k + 1$;
Add extra bits to v where necessary;
End.

[17]. As large networks may require a large ID space, [17] proposes an approximate method that trades off the state of transport switches for ID space and shows that a small increase of this state brings a large reduction of the ID space. This method, Approximate Access Switch Classification (AASC), is what we use in this paper (cf. Algo.1).

A. Edge Configuration

We now discuss, how the flow setup can be done with our fabric. To set up a flow, the controller needs to configure only two switches: the source and the destination edge switch. Consider a user (a mobile node) j attached to edge node i. To locate the user in the network, we assign it the following locator ID: $(ecv^i.uid^i_j)$, where ecv^i uniquely identifies edge node i, and uid^i_j identifies the user in the context of that edge. This address changes as the user moves though the network. This is to contrast with the IP address that the user receives as part of the network attachment, which is essentially serving as an identifier and remains constant throughout the session of the user. Assuming IPV4 transport in the switches, we assign 12 bits to identify a user within an edge node and use the remaining transport bits for the edges, i.e. $L = 20$.

The controller maintains a mapping between the IP and the locator address for each user. On flow arrival, the attachment edge switch (referred to as source edge) sends a packetIn to the controller. The controller looks up the locator IP of the destination of the flow (we assume that the destination is in the same network). It then pushes a flow rule to the source edge to encapsulate flow packets with the locator ID of the destination edge. Similarly, it pushes a rule to the destination edge to de-capsulate the packet. The forwarding in the core switches is performed on (parts of) the locator ID in the outer header. The destination edge uses the original IP address in the packet to forward it to the receiver.

Our solution mitigates **R1** by reducing the computational load at the controller and the number of flowmods per flow to exactly two, independently of the network size; it mitigates **R2** by reducing the number of terms under maximization in (1) to two. Finally, it solves **R3** by pre-installing paths among edge switches. Note that our solution maintains the control logic and the fine-grained management capability of SDN: indeed,

as it relies on usual OpenFlow flow rules, each of these can be changed as required. Hence, the controller can reconfigure core switches or the entire fabric as necessary. Furthermore, the controller can deploy more complex fabrics catering for different traffic classes or provide multiple paths among two edges for load-balancing purposes. Besides, even within our solution realm, the fine-grained management is preserved, as the controller can still manage each individual flow at the edge.

The encapsulation and de-capsulation at the edge switches may increase the processing load. However, in many settings we expect much lower traffic loads at the edges than in the core. Besides, the load of an edge switch can be well addressed through network planning, as an edge switch only serves local users: an edge switch can be assumed to be dimensioned to process the load of the attached users (cf. [32] for an example). We therefore do not expect that this extra processing overhead at edges affects the performance in any significant way. Our evaluation shows that this is indeed the case.

We implemented this solution in our testbed and studied its performance through measurements. We show how the flow setup procedure and more complex tasks, notably the support of user mobility and the support for network topology dynamics, perform over the fabric created using AASC. Our implementation uses wildcard matching to forward packets based on bits of the locator IDs embedded in the destination IP address. We developed our flow setup and mobility solutions on the Floodlight controller.

B. Flow Setup Latency

Fig.8 depicts the flow setup latency of our mechanism for different topologies described in §IV-B. The results are obtained through measurements over our testbed. We observe that the latency is small in all cases: the median and 99-percentile latencies are 1.15ms and 2.6ms for the small, 2.9ms and 3.8ms for the medium, and 5.9ms and 7.7ms for the large topology. With this, we conclude that our solution clearly outperforms those studied in §IV-B and effectively reduces the flow setup latencies, both in the median and the tail, by factors 3 to 5 compared to the results shown in Fig.7. Specifically, it provides a good performance in the tail, which indicates its scalability: while we increase the network size from 60 switches for small networks to 516 switches for large networks, the 99-percentile of flow setup latency increases by only a factor of 3 and remains below 8ms in all the cases.

Although our solution mitigates **R1**, it cannot completely solve it, as the interplay of the multi-threaded Floodlight controller, the used Java networking packages and the underlying Linux operating system is quite complex, as discussed in more detail in [25]. Our analysis shows that this explains the observed increase of the flow setup latencies with the growing network size. We insist, however, that these increases are negligible, compared to values in Fig. 6 and Fig. 7.

In short, these results indicate that the proposed solution efficiently mitigates problems **R1**, **R2**, and **R3** and also provide significantly better performance than the state of the art solutions, as discussed in §IV-B. Furthermore, the proposed solution is scalable, both for the fabric setup and, even more so, for the end-to-end flow setup, which is paramount to be able to use SDN e.g. as an enabler for 5G mobile networks.

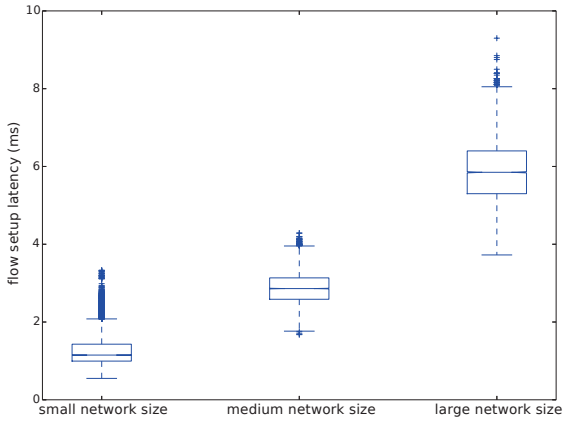

Fig. 8. The flow setup latency measured over our testbed using our solution.

C. Network Dynamics and User Mobility

We now show how our solution can easily handle network dynamics, such as topological changes and user mobility.

Support for topological changes and diverse failures in the network, even if infrequent, is important, as the controller may need to modify a large set of paths in the network when they happen. The central insight is that we do not need an instantaneous reconstruction of the classification vectors in the case of topological changes, which would be time consuming. To avoid it, we apply a "fast reaction" strategy: following a topology change event in the fabric, we specify correcting forwarding rules at the core switches for the flows affected by these changes. This is a tradeoff: it comes at the cost of additional flow table space, yet efficiently mitigates the time criticality of the potentially required re-routing. This is also to contrast to [33], in which interventions in all source access switches of the flows affected by the changes are necessary.

To analyze the behavior of our solution under topological changes, we perform additional experiments. To emulate such network dynamics, we randomly choose core arcs in the network and drop them, such that the edge-to-edge paths affected by this change need to be re-routed. According to the fast reaction strategy, this requires both addition of new flow rules in some switches in the network and the removal of rules in some others to avoid routing over old, incorrect paths. We measure these additions and removals and report them in Table I, varying the fraction of path changes from 10% to 30%. For comparison, we report the performance of a solution, which performs per edge switch aggregation [30], referred to as SoftCell. The results are shown for the large network with $K = 12$. These results show that the proposed fabric solution not only reduces the number of rules in individual switches, but also does it in a way that enables efficient handling of changes in the network: the number of added and removed rules are in the same order, or even smaller, than when a per edge aggregation mechanism such as SoftCell is used, a clear indication that our solution does not affect the fine-

grained management capabilities of SDN and imposes no extra overhead on the controller.

Path Changes	#added rules		#removed rules	
	SoftCell	Our solution	SoftCell	Our solution
10%	1390	1390	1065	895
20%	2520	2520	1975	1450
30%	3530	3530	2615	2105

TABLE I
TOTAL NUMBER OF ADDED AND REMOVED RULES FOR SOFTCELL AND OUR SOLUTION UNDER TOPOLOGICAL CHANGES.

In essence, our fabric produces logical pipes, which carry traffic between any two edge switches in the network. When users move through the network, their flows get migrated from one pipe to another under the coordination of the controller. This is done in the edges, using the old and new locator assigned to the user. Thus, only three switches need to be updated on user mobility: source, old destination, and new destination edge switches (in the case the destination moves; similar applies to source mobility). The motivation is to reduce the involvement of the controller to the minimum possible and to reduce the user mobility latency in SDN (by reducing the effects of **R1**, **R2**, and **R3**). While being minimal, this still allows the implementation of different edge-based mobility support strategies. In this paper, we use the following strategy.

When a user moves to a new edge switch, a new locator ID is assigned to the user by the controller. This new locator ID is used to forward the new packets generated towards this user. The controller therefore needs to update the forwarding rule at the source edge switch, using this new locator ID. It also installs a forwarding rule at the new destination edge switch, in order to remove the encapsulation header and to forward the packet to the user. The inflight packets however still carry the old locator ID of the user in the header. The controller thus installs a forwarding rule at the old destination edge switch, to replace the old locator ID with the new locator ID in the header of these packets, and to forward them to the new destination edge switch. (It may set a timeout for this re-forwarding rule, to be able to reuse the old locator ID for other users attached to this edge switch). Besides, the old forwarding rules at the previous destination switch needs to be removed. Clearly, this is a more complex task than a classical flow setup procedure and requires updating three access switches, notably adding new and removing old rules. Our results show that all these tasks can be performed very efficiently.

We implemented this mobility solution within Floodlight and evaluated its performance on our testbed. We define the handover latency as the difference between the RTT of a ping packet sent immediately after a mobile user changes the point of attachment and the RTT of a ping packet sent without mobility. In other terms, we measure the additional latency observed by the user, when the user changes the point of attachment. Consequently, in contrast to the results in Fig.8, which reports one-way latency, here we report the round-trip time (RTT), which includes the time to reconfigure uplink and downlink paths of a flow. Note that these two paths could be disjoint. We show the results for the three topologies used in this paper and for a user handover rate of 1000 handovers per second. Fig. 9 depicts the results. We observe that our solution can efficiently handle user mobility providing handover latencies below 20 ms in the largest topology. Specifically, the median and the 99-percentile handover latencies are 8ms and 17ms, respectively. Note that these latencies, as RTT values, are almost twice as big as the one-way flow setup latencies depicted in Fig. 8. We can therefore claim that the one-way handover latency remains in the same order as the one-way flow setup latency, and hence, while dealing with a more complex task, our solution provides the same performance.

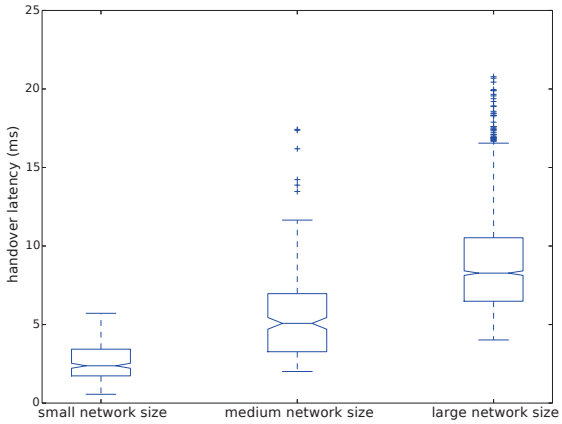

Fig. 9. The handover latency measured over our testbed using our solution, for different network sizes and for a handover rate of 1000 users per seconds.

In summary, this section shows that the fabric-based solution not only can mitigate the scalability problem we observed in §IV, but can also be easily extended to perform more complex tasks such as handling user mobility and topological changes. Using these insights, [34] describes, how this solution can be used to provide an efficient service function chaining in the network, and hence, support complex settings such as core networks of mobile carriers.

VI. CONCLUSION

This article studies the performance of SDN from the end-user perspective. It shows that a strictly separate handling of topology and flows is essential for low latency guarantees. These are in turn critical for many novel use cases such as machine-type or V2X communications. The article presents an alternative approach to flow setup in SDN, which separates the control of edge and core switches and scales well to large settings. We showed that the proposed solution not only supports fine-grained end-user flow management, but can also support more complex scenarios related to network dynamics. In particular, we demonstrated and evaluated the support for user mobility and network topology changes.

Multiple directions of future work are open. First, we are interested in extending our study to the case of hardware switches. As the flow rule installation latencies in hardware switches exhibit higher variances than in software switches, we expect the performance problem to be aggravated, especially when the network is composed of a diverse set of hardware switches. This is because switches from different vendors

switches might present different performances, increasing the effect of **R2** and **R3**. Second, we plan to study how to adapt the hierarchical control [35] to our solution, to make it even more scalable. Our initial results show the feasibility of that. We are interested to study how more complex procedures, such as dynamic SFC chaining, perform using our solution. As indicated in §V, these procedures can be easily integrated in our solution. The question is how they perform and scale.

REFERENCES

[1] Y. Jarraya, T. Madi, and M. Debbabi, "A survey and a layered taxonomy of software-defined networking," *IEEE Communications Surveys Tutorials*, vol. 16, no. 4, pp. 1955–1980, 2014.
[2] B. A. A. Nunes, M. Mendonca, X. N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Communications Surveys Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.
[3] N. McKeown and et al., "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, April 2008.
[4] ONF, "Software-defined networking: The new norm for networks," in *Open Networking Foundation*, April 2012.
[5] P. Berde and et al., "Onos: Towards an open, distributed sdn os," in *ACM Workshop HotSDN*, 2014.
[6] Floodlight, "Floodlight OpenFlow Controller – Project Floodlight." [Online]. Available: http://www.projectfloodlight.org/floodlight/
[7] T. Koponen and et al., "Onix: A distributed control platform for large-scale production networks," in *USENIX OSDI*, 2010.
[8] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *USENIX Workshop Hot-ICE*, 2012.
[9] M. Jarschel, F. Lehrieder, Z. Magyari, and R. Pries, "A flexible openflow-controller benchmark," in *European Workshop EWSDN*, 2012.
[10] OVS, "Open vswitch," http://openvswitch.org/.
[11] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "Oflops: An open framework for openflow switch evaluation," in *ACM PAM'12*.
[12] A. Bianco, R. Birke, L. Giraudo, and M. Palacin, "Openflow switching: Data plane performance," in *IEEE ICC*, 2010.
[13] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," in *ACM SIGCOMM*, 2010.
[14] K. He and et al., "Measuring control plane latency in sdn-enabled switches," in *ACM SOSR*, 2015.
[15] 3GPP, "Service requirements for the 5G system; Stage 1," 3GPP, Technical Specification (TS) 22.261, 09 2017, version 15.2.0.
[16] H. Chen and T. Benson, "The case for making tight control plane latency guarantees in sdn switches," in *ACM SOSR*, 2017.
[17] R. Khalili, W.-Y. Poe, Z. Despotovic, and A. Hecker, "Reducing state of sdn switches in mobile core networks by flow rule aggregation," in *IEEE ICCCN*, 2016.
[18] R. Bifulco and M. Dusi, "Position paper: Reactive logic in software-defined networking: Accounting for the limitations of the switches," in *European Workshop EWSDN*, 2014.
[19] A. R. Curtis and et al., "Devoflow: Scaling flow management for high-performance networks," in *ACM SIGCOMM*, 2011.
[20] K. He and et al., "Latency in software defined networks: Measurements and mitigation techniques," in *ACM SIGMETRICS*, 2015.
[21] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about sdn flow tables," in *ACM PAM*, 2015.
[22] A. Lazaris and et al., "Tango: Simplifying sdn control with automatic switch property inference, abstraction, and optimization," in *ACM CoNEXT*, 2014.
[23] "Mininet," http://mininet.org/.
[24] ONFv14, "Openflow switch specification v1.4.0," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.
[25] C. C. Marquezan, Z. Despotovic, R. Khalili, D. Perez-Caparros, and A. Hecker, "Understanding processing latency of sdn based mobility management in mobile core networks," in *IEEE PIMRC*, 2016.
[26] R. Nadiv and T. Naveh, "Wireless Backhaul Topologies: Analyzing Backhaul Topology Strategies," White Paper, Ceragon, August 2010.
[27] M. Howard, "Using carrier Ethernet to backhaul LTE," White Paper, Infonetics Research, February 2011.
[28] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM*, 2008.
[29] T. A. Pascoal, Y. G. Dantas, I. E. Fonseca, and V. Nigam, "Slow tcam exhaustion ddos attack," in *IFIP SEC*, 2017.
[30] X. Jin, L. E. Li, L. Vanbever, and J. Rexford, "SoftCell: Scalable and Flexible Cellular Core Network Architecture," in *ACM CoNEXT*, 2013.
[31] J. Halpern and et al., "Service function chaining (sfc) architecture," in *RFC 7665 (INFORMATIONAL)*, October 2015.
[32] K. He and et al., "Presto: Edge-based load balancing for fast datacenter networks," in *ACM SIGCOMM*, 2015.
[33] A. Hari, T. Lakshman, and G. Wilfong, "Path Switching: Reduced-State Flow Handling in SDN Using Path Information," in *ACM CoNEXT'15*.
[34] R. Khalili and et al., "Optimized service function chaining," in *IETF draft*. draft-khalili-sfc-optimized-chaining-00, March 2018.
[35] M. Moradi, L. E. Li, and Z. M. Mao, "SoftMoW: A Dynamic and Scalable Software Defined Architecture for Cellular WANs," in *ACM Workshop HotSDN*, 2014.

Ramin Khalili received his B.Sc. from Shiraz University, his M.Sc. from the Sharif University of Technology, both in Iran, and his Ph.D. in computer networks and distributed systems from UPMC, France. He was with the University of Massachusetts at Amherst, EPFL, and the Telekom Innovation Laboratories in Berlin, before joining the Huawei Research Center in Munich, Germany. Ramin published over thirty scientific papers and received multiple best paper awards during these years.

Zoran Despotovic received his M.Sc from University of Belgrade, Serbia and his PhD in Computer and Communication Systems from École Polytechnique Fédérale de Lausanne (EPFL), Switzerland. After his graduation he worked for NTT DOCOMO until September 2011, first as a senior researcher and then as a manager. In 2011 he joined Huawei in Munich, where he still works as a principal engineer. Zoran participated in many EU funded projects and published around forty scientific papers.

Artur Hecker (Dipl. inform. from Universität Karlsruhe, Germany; PhD from ENST, Paris, France) is Director of Future Network Technologies at the Munich Research Center of Huawei Technologies. From 2006 to 2013, Artur was Associate Professor at Télécom ParisTech, acting as Head of Security and Networking research. Overall, Artur looks back at more than 15 years of entrepreneurial, academic and industry experience in networks, systems and system security.