Stateful Distributed Firewall as a Service in SDN

Ali Zeineddine
Computer Science Department
American University of Beirut
Beirut, Lebanon
ahz09@mail.aub.edu

Wassim El-Hajj
Computer Science Department
American University of Beirut
Beirut, Lebanon
we07@aub.edu.lb

***Abstract*—Software-defined networking (SDN) is a newly emerging approach in computer networking which abstracts network control functionalities and enables its direct programmability at the management plane. A new framework of communication between the control-plane and the data-plane is gaining a lot of attraction recently, which combines the advantages of the proactive approach, in pre-installing the flow rules in the data-plane, and the advantages of the reactive approach, in its ability to dynamically react to network events. This hybrid approach utilizes the potential of the SDN switches to recognize and host state machines. While the trending success of SDN is set to continue, this evolving network paradigm requires a new set of tools and strategies to secure the network elements against intrusions and at the same time maintain its efficiency and reliability. In this paper, we take advantage of the hybrid approach of network controllability and management to offload the processing of stateful applications from the control-plane to the data-plane and propose our framework, SDFS, which optimizes a distributed stateful application in the data-plane to transform the SDN network into one big firewall. While maintaining modularity of the framework, SDFS offers an optimized processing burden distribution of the stateful application in the data-plane among the switches in the network with inherent fault-tolerance mechanisms that eliminate the need for immediate controller intervention even in cases of network failure or attacks.**

Keywords—SDN, distributed stateful applications, burden distribution, high-availability

I. Introduction and Background

While Software-defined networking (SDN) turned from a pure idea to a concrete framework [1, 2, 3], it has only gained recognition by vendors when OpenFlow [4] was first adopted as a standard southbound protocol for communication between the control plane and the data plane. Since then, OpenFlow compliant switches were produced such as OpenVswitch. These switches employ OpenFlow Tables to forward traffic. A flow table comprises a list of flow table entries (flows). Each flow table entry is comprised of a Match representing a set of packet fields, in addition to other OpenFlow specific match fields (inport, metadata,..), an Action such as outport, drop, and modify-packet-fields, along with counters that hold statistics about the flows and packets. Currently, four group types exist: Fast-Failover, Select, all, and indirect. The fast-failover group addresses the slow switch-controller communication problem upon link failure. Select group permits the execution of one bucket after hashing on packet fields. All group executes all the buckets in the group and Indirect group executes the one bucket in the group.

There are obvious security advantages that are entitled to the SDN architecture. The centralized intelligence allows for a complete view to analyze all of the feedback from the network. Hence, the controller can virtually act as a global anomaly-detection system. On the other hand, SDN opens the door to a broad variety of new security challenges that require special considerations especially those resulting from the centralized layered architecture [5-7]. In all the employed security strategies, any attempt to augment the SDN network with security enhancements cannot but extensively rely on the controller as it is the only intelligent unit in the architecture. Consequently, a new approach emerged that embraces the idea of discarding the fully centralized architecture of SDN and migrating part of the intelligence to the SDN switches [8-11].

We elaborate on the approach suggested in [11], where the authors proposed SNAP, a programming language and a compiler that abstracts the data-plane into one big switch. This abstraction treats local state variables as global variables in such a way that the programmer does not have to worry about where these variables actually reside in the data-plane while writing the network's stateful program. Such a framework allows the utilization of more switches in the network to host the state variables. SNAP's optimization algorithms divide the global state of the application into several parts and distribute them across various switches provided that packets belonging to the same application instance can still traverse these switches in the required order to maintain a consistent update of the states in the application. Thus, these variables are arrays that are updated consistently as packets traverse the network and together represent the global state of the application. In order to exchange state variables between switches along the path of the packet, SNAP utilizes special header fields. These header fields contain a dictionary of variables and their corresponding values which are used in the switches to match and update the states.

To illustrate this approach, consider the network in figure 1 where hosts h1, h2, h3, and h4 are hosts in two different subnets (orange and yellow) that are allowed to communicate with s2 only if they knock a certain sequence of ports destined to s1 and s1 accepts this communication by a confirmation response. The SNAP compiler would distribute the state of the application into two global variables: *Sequence[client][port_sequence]* and *Allow[client]*. The compiler can store *Sequence* on switch 3 and *Allow* on switch 4. Then, as a client attempts to knock the sequence to s1, when the packet arrives at switch 3 the client's corresponding *Sequence* variable is updated. After a particular client knocks the whole sequence to s1, s1 replies to accept the communication between this client and s2. When s1's reply arrives at switch 4, the client's corresponding *Allow* variable is updated to True to allow its communication with s2. Subsequently, as this client communicates with s2, when the packet arrives at switch 3 the corresponding sequence number is attached to the header of the packet. The packet is then forwarded to Switch 4 which then decides either to forward or

978-1-5386-4633-5/18/$31.00 ©2018 IEEE

drop the packet destined to s2 by asserting the values of both the sequence number in the header of the packet and the *Allow* variable for the corresponding client. In this approach, the SNAP compiler optimizes both the routes in the network along with the state variable placement to ensure that the packets required for consistent state updates converge on the switches in the correct order required by the application. However, single points of failure still exist where the failure of one switch to perform variable updates will result in the whole application behaving incorrectly. Upon such failure in the SNAP framework, controller intervention is required to update the network involving recompilation and re-optimization that forces a major

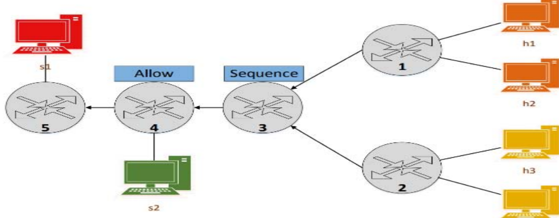

Fig. 2. State-variable placement in SNAP framework

downtime in the network which can be detrimental for time-sensitive applications and gives the attackers enough time to perform malicious jobs. To address all the above issues, we propose SDFS, which has the following properties: (1) all state updates are done atomically, (2) burden is distributed among switches in an optimal fashion based on mathematical formulations, (3) and fault tolerance is achieved.

II. SDFS - A Distributed SDN Stateful Firewall

A. Structure of states at the data-plane

[11]'s state distribution framework attempts to distribute the state variables over the network switches. Accordingly, there arises the burden to maintain atomic consistency of the state that spans multiple switches. However, distributing the state variables over multiple switches introduces constraints in terms of atomic consistency and packet path convergence. It also introduces restrictions on the conditions of state-updates where conditional updates of variables are bound by the placement order of these variables on the path of a packet. In the port-knocking example for instance, the *Sequence* variable cannot be updated based on the state of the *Allow* variable, for packets initiated from the hosts since Allow variable is placed beyond *switch:3* on the route to the servers.

To tackle these issues, SDFS adheres to the state-placement in the data-plane through atomicity of the state-placement at the level of updates. Accordingly, SDFS distributes the global state into *state-instances* where each state-instance comprises all the variables required for a consistent state update. A state instance is the most granular component of the global state that maintains its atomicity and consistency independent of the other state-instances. Referring to the port-knocking example, a state-instance in SDFS would comprise both Sequence and Allow variables that correspond to the same client. Hence, instead of being restricted in convergence to switches 3&4 to host the state-variables, a viable state-placement in SDFS would host state-instance of client-1 on switch:1, that of client-3 on switch:2, and that of clients 2&4 on switches 3&4 respectively.

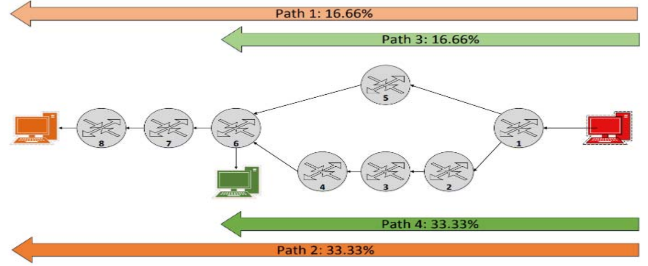

Fig. 1. Connection-tracking example with load-balancing for distributed states in the data-plane

B. Cost & capacity-aware distribution of the state processing burden

Figure 2 represents a topology where connection tracking is applied for *host:red*. That is, if and only if *host:red* initiates a connection with *host:green*, *host:green* can communicate with *host:red* in the same connection session. The same applies for *host:orange* when *host:red* initiates a connection with it. We assume the network employs symmetric load-balancing procedures. The load-balancer at *switch:1* assigns a load-weight *1:2* for the upper and lower outports respectively. Hence, according to our approach, the granularity of the state-instances is confined by the four-tuple combination: *[source-ip, destination-ip,source-port,destination-port]*.

Given an estimation of the relative traffic weights of the paths in the network, for example ⅓ of the traffic takes the upper path and ⅔ of the traffic takes the lower path when initiated from *host:red*, SDFS optimizes the distribution of prospective state-instances to be initiated in the network as to equally distribute the burden of processing upon all the switches.

B.1 The Tracking Metric

The tracking metric is a measurement of the state processing weight carried out by a switch per path. The processing weight in this example is the amount of connection tracking instances that a switch is responsible for as a proportion of the total processing weight (TTW), where the latter can be an arbitrary value assigned in the network. If the network assumes one big firewall to track connections initiated from *host:red*, then traffic

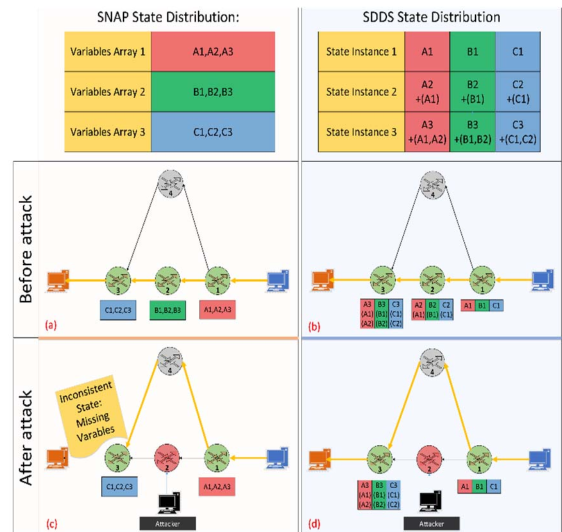

Fig. 3. SNAP vs SDFS: Distribution of states and availability of the stateful application under healthy network conditions and after attack

from *host:red* can take multiple paths to reach *host:orange* and *host:green* as shown in figure 2 where each path has its corresponding path weight (P_i) as a percentage of all the processing weight employed in the network. We denote by $tp_{n,i}$ the tracking weight per path assigned to *switch:n* pertaining to *path:i*. Since *path:2* does not pass through *switch:5*, $tp_{5,2}$ is zero. The total tracking weight carried by a switch, denoted by T_n is then the sum of the tracking weights per path for all paths on this particular switch. For example $T_1 = tp_{1,1} + tp_{1,2} + tp_{1,3} + tp_{1,4}$. Put another way, the tracking burden on *switch:1* is the sum of the burden assigned to *switch:1* for each of the paths 1,2,3,4. Hence the definition:

- $T_n = \sum_{i=1}^{I} tp_{n,i}$ where n is the switch number, i is the path number, and I is the total number of paths.

B.2 Optimization of the Tracking Values

In order to achieve a distributed burden, we employ the least squares method subject to certain constraints. Our objective function in this optimization is as follows:

- $obj.function = min \sum_{n=1}^{N}(T_n)^2$

where **n** is the switch number, **N** is the total number of switches, and T_n is the tracking burden for switch n. $\sum_{n=1}^{N}(T_n)$ = TTW, where the sum of the tracked connections for all switches is equal to the total traffic tracking weight required in the network.

A Change in Variables: It does not suffice, however, to find the overall tracking weight per switch as represented by T_n. For the optimization to produce optimal results, it must calculate the required tracking weight resulting from each path on each switch. The decision whether to track a connection is tightly dependent on which path the packet is taking through the network. Thus, substituting the definition for T_n, the objective function can be written as:

- $obj.function = min \sum_{n=1}^{N}(\sum_{i=1}^{I} tp_{n,i})^2$ where n is the switch number, N is the total number of switches, i is the path number, and I is the total number of paths.

In this representation, the solver will try to find the value of each tracking weight for every switch and for every path. This optimization is subject to the following set of constraints:

(1) $\sum_{n=1}^{N} tp_{n,i} = P_i$ The sum of the tracking weights on all switches resulting from path i is equivalent to the probability of this path.
(2) $tp_{n,i} \geq 0$ for all n,i
(3) $tp_{n,i} = 0$ for path i that does not pass through *switch:n*
(4) $\sum_{i=1}^{I} tp_{n,i} \leq C_n$ The sum of tracking weights at each switch n for all paths is less or equal to the capacity of switch n.

Upon the termination of the optimization computation, the system will yield values for each $tp_{n,i}$. As they represent the state processing weight for each switch per path, these values can then be used by the application to distribute the state-instances among the network switches.

B.3 Distribution Results for Connection-Tracking Example:

Referring to the network in figure 2 where the load balancer at switch:1 outputs ⅓ of the traffic from *host:red* in the upper path and ⅔ of the traffic from *host:red* in the lower path, we will assume that the total traffic tracking weight in the network is 600. This number refers to the total amount of connections being initiated from the protected host in the network (*host:red*) per an arbitrary unit time. We will also assume that the connections initiated from *host:red* are divided equally to *host:orange* and *host:green* for simplicity. Accordingly, the load balancer will establish the path weights as: Path:1 = 100, path:2 = 200, path:3 = 100, path:4 = 200. Applying the burden distribution optimization framework on the load-balanced network provided previously, without capacity constraints, will produce results in Table 1. Looking at the results, it should not be surprising that the total processing burdens for all of the switches are equal (75 per switch) since this is the outcome of an optimal distribution of burden among the switches.

TABLE I. OPTIMAL BURDEN DISTRIBUTION RESULTS ACCORDING TO PATH EXPLICIT OPTIMIZATION METHOD

Processing Burden	path:1	path:2	path:3	path:4	Total Processing/ switch
Switch:1	5.2	18	23.2	28.5	75
Switch:2	-	27.4	-	47.6	75
Switch:3	-	27.4	-	47.6	75
Switch:4	-	27.4	-	47.6	75
Switch:5	21.5	-	53.5	-	75
Switch:6	5.2	18	23.2	28.5	75
Switch:7	34	41	-	-	75
Switch:8	34	41	-	-	75
Total path weight	100	200	100	200	600

C. High availability of the application in the data-plane

As the controller intervenes to restore the network into a stable state upon network failures. it is constrained by the communication time requirements. This might work well for applications that do not require immediate rectification. For applications that do, however, such approach is useless and can consequently cripple the availability of the whole service in the network. In SNAP, if one of the switches is attacked, the state variables that are hosted on this switch will be lost, which will have a detrimental effect on the correctness of all state instances in the network, even those at the switches that were not attacked, leaving the whole application dysfunctional. For this reason, the best option for data-plane distributed firewall applications is to inherently employ high-availability techniques. In order to demonstrate this behavior in comparison to our proposal, we will consider the scenarios in figure 3. Scenarios (a) and (b) represent the healthy network scenario for SNAP and our proposed framework (SDFS) respectively while (c) and (d) represent the network directly after an attack on switch:2 for SNAP and SDFS respectively. The first difference to be highlighted, between SNAP and SDFS, is the structure of the distributed state itself. As mentioned in Section B, SNAP state distribution relies on distributing the state variables among the switches in contrast to distributing state instances in SDFS. Hence, as shown in figure 3, while SNAP hosts the red, green, and blue variables on switches 1, 2, &3 respectively, SDFS hosts state-instances 1, 2, and 3 on these switches respectively. The second difference to be highlighted here is the inherent cumulative structure of the state instances in SDFS, where state-instance:2 subsumes state-instance:1, and similarly state-instance:3 subsumes both state-instance:1 and state-instance:2. Upon the failure of a switch to handle state updates, resulting from a successful attack, for example, this story changes drastically between the two approaches. In the SNAP scenario after the attack, scenario (c), the switch hosting the green variables is down. A fast-failover

mechanism on switch:1 will reroute the traffic through the redundant switch- switch:4- destined to host:orange. When the traffic arrives at switch:3 from switch:4, it falls inconsistently with the state updates since the green variables are missing. Accordingly, all traffic from host:blue to host:orange will not be consistent with the application's intended behavior.

Conversely, upon a successful attack on switch:2 in the SDFS framework, scenario (d), traffic that was initially being processed through state-instance:1 and state-instance:3 is not affected since these state-instances are still intact and consistent. Not only the already established (through state-instances 1&3) connections remain intact, but also future traffic that falls in these state-instances remains intact too. Regarding traffic that was being tracked at switch:2, these connections will be lost. However, future connections that fall within state-instance:2 will now be taken by switch:3. Since state-instance:3 subsumes state-instance:2, traffic "intended" for state-instance:2 that arrives at switch:3 unprocessed, will be inherently processed by state-instance:3. In this scenario, switch:3 will be processing ⅔ of the traffic in the network. Although the burden is skewed, the point is to maintain the correctness of the application for all future connections at least, and give time for the controller to respond and reconfigure the network by distributing the burden through switches 1,4, and 3. To establish this behavior, we employ the cumulative accept/forfeit probabilities, as discussed below, in a group:select action on the switches. For every path, we assign two buckets for its group, the **Accept** bucket and the **Forfeit** bucket, each with its corresponding weight. Then, when a packet is matched on its path, it is submitted to its corresponding group, where the select action uses a hash function that takes predetermined packet fields as input and chooses the bucket with the larger output value:

*Accept_output = hash(packet_fields + accept_bucket_id)*Accept_weight*
*Forfeit_output = hash(packet_fields + forfeit_bucket_id)*forfeit_weight*

The decision is then made according to the action in each bucket, where the accept bucket will initiate the state tracking at the switch, and the forfeit bucket will leave it for the next switch. The power of this rises in that every switch's tracking decision is computed as to inherently assume a cumulative behavior. In the healthy network scenario, each switch carries the responsibility of its assigned burden. However, in case of a faulty behavior, say a defective switch on the path that did not perform its tracking responsibility, the subsequent switches will automatically carry the tracking responsibility. In the next section, we discuss how to compute the Accept/Forfeit bucket weights in order to arrive at the proposed high availability framework.

C.1 Cumulative Probabilities in Accept/Pass Group:Select Buckets for High Availability

Each $tp_{n,i}$ value, produced from the optimization procedure, as described in Section B, represents switch n's tracking participation in path i. In other words, it represents the probability that a certain connection will be tracked by switch n and not any other switch on the path of this connection. For the switches in figure 4 for example, each switch has a 1/3 processing burden. To arrive at the cumulative subsumption behavior, we can simply calculate the cumulative accept weights according to the below formula:

- $Ap_{u,i} = \sum_{k=1}^{u} ntp_{k,i}$, where u is the position index of the current switch along path i. A_p is the probability of processing packets belonging to path:i at switch:u (accept probability).
- $Fp_{u,i} = 1 - Ap_{u,i}$, where F_p is the probability of not processing packets belonging to path:i at switch:u (forfeit probability)

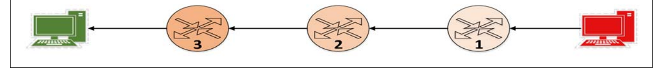

Fig. 4. Network example to demonstrate cumulative chunking approach

Thus, for the network of figure 4, switch:1 will have 1/3 cumulative burden, switch:2 will have 2/3, and switch:3 will have 3/3. Although switch:2, for example, has a 2/3 cumulative probability of processing, it will only process 1/3 of the traffic simply because the first 1/3 of the traffic will already be processed by switch:1 and tagged as **already processed** using a preassigned reserved header. However, if switch:1 fails to process a packet it was responsible for, this packet will arrive at switch:2 unprocessed. And accordingly, it will be processed by switch:2 instead, by virtue of it having a higher cumulative probability which completely overlaps that of switch:1. **Note:** Here, it is enough to calculate the cumulative accept probability at any switch simply by having a 1/3 probability increase over that of the previous switch because we force the decision events to be probabilistically dependent events by choosing a consistent *accept_bucket_id* and *forfeit_bucket_id* in all switches in the network. To arrive at the *accept_weight* and *forfeit_weight*, the probabilities must undergo the following transformation:

If accept_probability < forfeit_probaility:
*accept_weight = 2 * accept_probability*
forfeit_weight = accept_probability + forfeit_probaility
Else
accept_weight = accept_probability + forfeit_probaility
*forfeit_weight = 2 * forfeit_probaility*

TABLE II. VARIABLE DEPENDENT ACCEPT CUMULATIVE PROBABILITY RESULTS

Accept cumulative weight per path	Path:1 /100	Path:2 /200	Path:3 /100	Path:4 /200	Total Processing/ switch
Switch:1	5.2	18	23.2	28.5	75
Switch:2	NA	45.4	NA	76.1	75
Switch:3	NA	72.7	NA	123.8	75
Switch:4	NA	100.1	NA	171.4	75
Switch:5	26.6	NA	76.7	NA	75
Switch:6	31.8	118	100	200	75
Switch:7	66	159	-	-	75
Switch:8	100	200	-	-	75
Total path weight	100	200	100	200	600

C.2 Cumulative Probabilities Results for Connection-Tracking Example

Applying the cumulative approach on the optimization results in Table 1 will produce the results in Table 2. As can be seen, the probabilities converge to a "full" probability at the last switch to the destination (100/100 for path:1, 200/200 for path:2, 100/100 for path:3, and 200/200 for path:4).

III. FRAMEWORK EVALUATION

On an Ubuntu Linux machine (kernel v. 4.4.0), the network environment is simulated using mininet (version 2.2.1) which takes as input a topology file and creates the corresponding network comprised of openVswitch bridges (version 2.8.90),

links, and hosts (namespaces). The proposed SDFS engine is implemented in python as a proof of concept and uploaded to github [12]. Figure 5 (a) depicts the primary topology structure used in testing which corresponds to the campus networks used in SNAP. Same structure is also used for tiered data-centers: access layer, aggregation layer, core layer, and edge routers. In this topology, we employ a distributed connection tracking application where the colored hosts are protected hosts in the network that cannot be contacted unless they initiate the connection with the black hosts. The routing in the network incorporates four routes colored in correspondence with the protected hosts. Accordingly, we give each one of the four paths a weight of 1000. Thus in total, we have 4000 connections being established in the network (1000 for each host) per unit time. We take a snapshot of the amount of connections being processed at each switch after each 100 fired connections from each host (increments of 400 new connections).

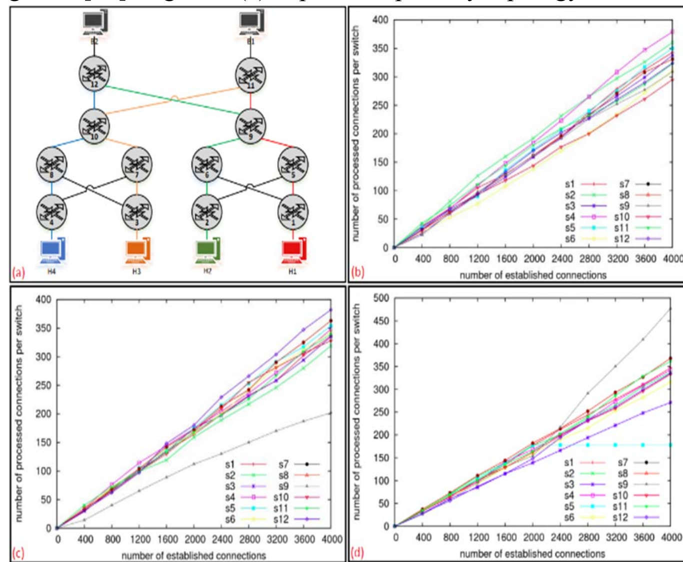

Fig. 5. SDFS in healthy network experiment

Figure 5 (b) shows the amount of processed connections per switch at each snapshot. Since we have 12 switches, optimally, each switch should be sharing 1/12 of the processing burden in the network at any particular snapshot. As can be seen in the results, each switch roughly processes 33 connections when the number of fired connections in the network is 400. Since the packet L3-L4 fields are chosen at random, where each combination resembles a connection, the processing decision taken at each switch is probabilistic. This is why we see a slight divergence in the distribution as we arrive at 4000 established connections. While the optimal processing burden at 4000 established connections is 333 for each switch, switch:10 processes 295 connections while switch:4 processes 379 connections. Hence, a fair distribution is achieved even with slight differences attributed to the heuristic assignment of ports in the generated TCP packets upon which the switches decide whether to process the connection distributively without communicating with the controller. Figure 5 (c) depicts the results of the experiment where we introduce a capacity constraint of 200 on switch:9. As can be seen, at each snapshot taken, switch:9's relative processing burden was 200/4000*number of established connections. When the amount of established connections was 4000, switch 9's participation was limited to 200. However, the remaining 3800 connections in the network are distributed equally among the other switches that collaborate to cover switch 9's shortage. To investigate the high-availability of the application as discussed in section C, we simulate the same provisioned topology, however, switch 5 now fails midway. Accordingly, a fast-failover mechanism takes place at switch 1 where instead of forwarding traffic to switch 5, switch 1 forwards the traffic to switch 6. While switch 6 can take part in the burden processing, it does not have to, and can effortlessly leave the burden onto switch 9, since the packets are not yet tagged as **already processed**. Here we show how the processing burden can be handed-over to switch 9 when switch 5 fails. As can be seen in figure 5 (d), when switch 5 fails, it stops accepting (recording) new connections, while this burden is carried over to switch 9. Here, when switch 5 fails, all the connections that were already being processed by it are lost. However, subsequent connections that are expected to be processed by switch 5 are handed over to switch 9.

IV. Conclusion and Future Work

In this paper, we introduced SDFS, a distributed stateful SDN approach which offers an optimized processing burden distribution and high availability features. We are exploring approaches to (i) aggregate path requirements, (ii) make the application completely pluggable, and (iii) maintain a fair distribution of processing burden even after cases of failure.

V. References

[1] Casado, Martin, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. "Ethane: taking control of the enterprise." In *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 1-12. ACM, 2007.

[2] Casado, Martin, Tal Garfinkel, Aditya Akella, Michael J. Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. "SANE: A Protection Architecture for Enterprise Networks." In *Usenix Security*. 2006.

[3] Greenberg, Albert, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. "A clean slate 4D approach to network control and management." *ACM SIGCOMM Computer Communication Review* 35, no. 5 (2005): 41-54.

[4] McKeown, Nick, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: enabling innovation in campus networks." *ACM SIGCOMM Computer Communication Review* 38, no. 2 (2008): 69-74.

[5] Zhang, Ying. "An adaptive flow counting method for anomaly detection in SDN." In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pp. 25-30. ACM, 2013.

[6] Hinrichs, Timothy, Natasha Gude, Martın Casado, John Mitchell, and Scott Shenker. "Expressing and enforcing flow-based network security policies."*University of Chicago, Tech. Rep* 9 (2008).

[7] Ahmed, Mohamed Fekih, Chamssedine Talhi, Makan Pourzandi, and Mohamed Cheriet. "A Software-Defined Scalable and Autonomous Architecture for Multi-tenancy." In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pp. 568-573. IEEE, 2014.

[8] Moshref, Masoud, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. "Flow-level state transition as a new switch primitive for SDN." In*Proceedings of the third workshop on Hot topics in software defined networking*, pp. 61-66. ACM, 2014.

[9] Yuan, Yifei, Rajeev Alur, and Boon Thau Loo. "NetEgg: Programming network policies by examples." In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, p. 20. ACM, 2014.

[10] Bosshart, Pat, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger et al. "P4: Programming protocol-independent packet processors." *ACM SIGCOMM Computer Communication Review* 44, no. 3 (2014): 87-95

[11] Arashloo, Mina Tahmasbi, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. "SNAP: Stateful network-wide abstractions for packet processing." In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pp. 29-43. ACM, 2016.

[12] "SDFS." Stateful Distributed Firewall as a Service. https://github.com/ali-hz/SD