A state consistency framework leveraging packet cloning and piggybacking for programmable network data planes

Hugo Garcia* and Naercio Magaia*

*LASIGE, Departamento de Informátca, Faculdade de Ciências, Universidade de Lisboa
hgarcia@lasige.di.fc.ul.pt, ndmagaia@ciencias.ulisboa.pt

***Abstract*—The Software-Defined Networking (SDN) technology is a network management method that allows dynamic, programmatically efficient network planning to improve its performance and monitoring. Given that SDN at each passing day becomes more prominent, a framework that can ensure reliable communication and a global state among devices become more important. We propose a state consistency framework that leverages a state machine abstraction using network updates among adjacent switches through packet piggybacking. We also use a moving average that when a condition is met, P4's packet cloning is triggered, hence ensuring that all packets arrive at their destination in the presence of link failures.**

***Index Terms*—SDN, State consistency, Programmable networks, Framework, P4**

I. Introduction

In the primordial phases of the Internet, the foundations for future work had to be done. They were the following: (i) the introduction of programmable functions in the network, (ii) enabling the virtualization of the network, (iii) the ability to demultiplex, and (iv) the idea to unify the architecture for middleboxes.

All of these would eventually lead to the separation of the planes, i.e., the control and data planes, which enabled the idea of Software Defined Networking (SDN) [1] to appear. The control plane relates to all the functions and processes that determine which path to use, and the data plane denotes all the functions and processes that forward packets from one interface to another.

However, there can be inconsistencies between the existing information in the control plane network device, i.e., the controller, and the one on the data plane network devices, i.e., switches, or even among the latter. Let us take a concrete example to give a better understanding of the problem at hand. Suppose an attacker can both inject new packets into the network and take original packets that were sent by trustworthy sources and alter them for their gain. There are various ways to do so, but the simplest one would be to sniff the desired packet, add to it a new "broken link" tag, and send the packet to its origin. The consistency issue would happen when the origin receives the malicious packet. Upon its reception, it will trigger a transition to divert all subsequent packets from the original destination.

This work was sponsored by FCT through the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020.

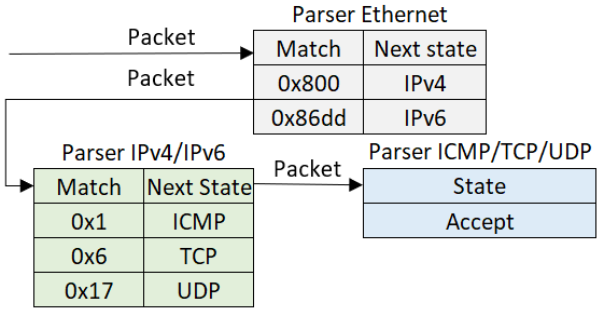

Fig. 1. An implementation example of a state machine filter used in our parser.

Another exacerbating factor of this type of attack is the fact that between adjacent switches there is no form of authentication. Thus, leading back to the original problem, that is, "how to avoid it?", and when trying to prevent it, "which information to send to the controller?", "who should have a consistent view of the network?", so that the controller can make appropriate choices.

In this article, a state consistency framework leveraging packet cloning and piggybacking for programmable network data planes is proposed. It was implemented using the novel Programming Protocol independent Packet Processor (P4) [3] programming language that runs on switches. The proposed framework guarantees that a consistent state is achieved via packet piggybacking, that is, every packet that passes through the switch gets added necessary information for the framework to work. It uses a moving average that triggers packet cloning, hence ensuring that packets arrive at their destination even in the presence of link failures. To the best of our knowledge, this is the first work implementing a network state consistency framework that ensures a high packet delivery rate in the presence of high link error rates using P4.

ISBN 978-3-903176-39-3 © 2021 IFIP

II. The Proposed Solution

We used the V1 model [4] of P4 as the basis for our implementation. But even more important than the model itself is the whole specification of P4 [5]. Please note that the V1 model is used mainly due to its simplicity of implementation compared to the Portable Switch Architecture (PSA) model [6].

A. The state machine abstraction

The state machine's implementation took place using the various match-action tables, sometimes changing their contents depending on the table's purpose. The starting point of our implementation was FAST [2]. However, and considering the difference between Open vSwitch (http://www.openvswitch.org/) and P4, we added tables to ensure that actions related to keeping the information in the packet or switch itself were properly implemented.

The design of the solution is rather straightforward. Given that P4 requires data structures, such as headers and metadata, they were created to enable access to packets.

From Fig. 1, it is possible to see that when a packet arrives at a switch, it will first try to match the contents of the Ethernet header to know where to proceed next. Afterward, it will match the IPv4/IPv6 header's contents to know again where to go. It will only accept and finish when it is done with the ICMP/TCP/UDP header contents.

B. Packet cloning and piggybacking

Packet piggybacking is used to allow switches to exchange state information, i.e., a timestamp and two ports, among themselves. It avoids creating new packets, and, as such, to get better overall network performance. The TCP options [7] was used, in particular the SACK and Timestamp options.

When a packet arrives, it is parsed and run through the ingress pipeline, where the state's maintenance occurs via the Timestamp option. We decided to use this option because it provides a straightforward way to send timestamps across switches. In addition, in case we do not need to send them, this option provides space for two 4 byte sized timestamp fields that could be filled with information [7].

```
bit<32> indexCounter = 0;
indexForPackets.read(indexCounter, 0);
networkPortsEgress
    .write((bit<32>)indexCounter,
    meta.ingressMetadata.egress_port);
networkPortsIngress
    .write((bit<32>)indexCounter,
    meta.ingressMetadata.ingress_port);
```

The code above shows how state maintenance occurs in our solution. First, a variable that will be used to store the index to access the registers is initialized. Afterward, we read from the register where the indexes are stored and place the `indexCounter` variable's value. Then, we write in the registers the values of the egress and ingress ports to create an artificial knowledge of the network state.

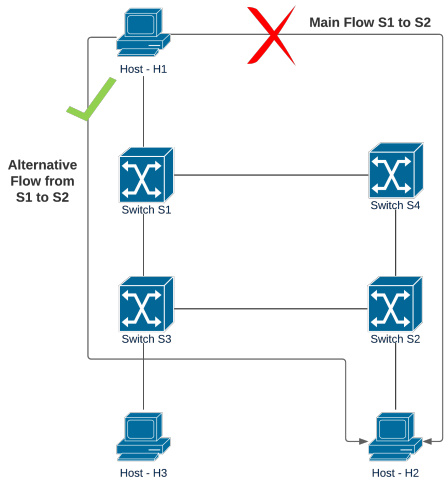

Fig. 2. An alternative path for packets to go from H1 to H2

Please note that the SACK option [8] is not used in its intended way. Instead of containing the planned blocks of data, i.e., two 32-bit unsigned integers [8], we used this space to put the egress and ingress ports in the packet. We use these specific ports so that when switches have a few pairs, they can be aware that traffic flows from these ports. Consequently, they can perceive as being safe to send packets through them.

Conversely, the Timestamp option is used normally. In its TS Value, we put the timestamp of the packet so that every other switch can make the necessary interactions with it after the first one. Other switches can compare their timestamp values to the one present in the packet, and depending on the value of the difference, it will then take the actions needed when certain conditions are met. These conditions and actions are rather simple. Suppose the difference between the timestamp that is stored on the TCP Timestamp option and V1 model metadata is greater than an average. In that case, it is assumed that the link can be having connectivity issues, and as such, the switch starts making clones hence ensuring that packets reach their intended destination.

A clone is merely a complete copy of the packet, and it is sent through a different egress port. Fig. 2 further illustrates this.

Afterward, we do not do anything in the egress pipeline, and every change is made during the ingress phase of the implementation. Finally, we remake the hash of the packet as we insert new data to it because if not and another switch verifies, it would not match, and the packet would be discarded. We also deparse the headers in the correct order. This is done to ensure that the packet is correctly formed hence avoiding it being discarded by the switches later on.

III. Performance Evaluation

The Mininet (http://www.mininet.org) simulator was used to evaluate the performance of the proposed framework. It was

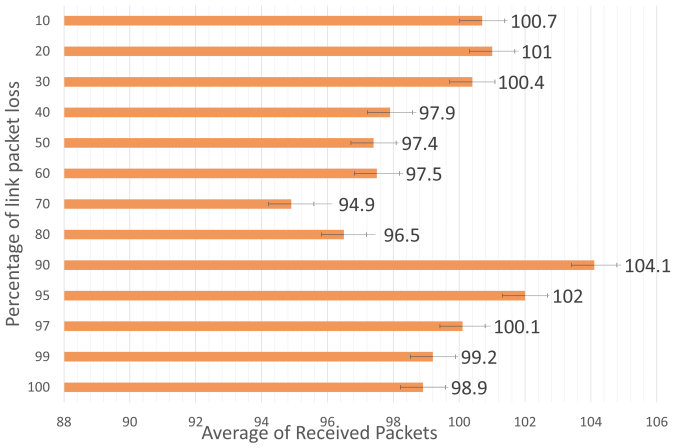

Fig. 3. Average number of received packets to H2.

used to simulate a network setup. We used a simple set of commands in Python to send a specific number of packets that the user would insert in a command line or through a Python script.

Fig. 2 shows a scenario consisting of four switches (Switch 1 - S1, S2, S3, and S4) and three hosts (Host 1 - H1, H2, and H3). All switches are adjacent to each other. In this scenario, H1 is connected to S1, H2 to S2, and H3 to S3. From H1 to H2, there are two possible flows: H1 - S1 - S4 - S2 - H2 and H1 - S1 - S3 - S2 - H2.

We obtained information on how many packets were received at the end host (i.e., H2) to compare it with the number of packets retransmitted using our method. This enabled us to know how many packets arrived through the regular and alternative (i.e., retransmissions) paths. With that in mind, we also needed to collect information on the number of retransmitted packets. These two go hand in hand with each other.

1) The number of received packets: First, we start analyzing the number of packets received at H2. Over the 130 runs, 75% correspond to 97 runs with near 100 received packets (99 to be exact); 13% correspond to 17 runs of received packets with great deviation (i.e., < 95 packets and > 105 packets); and, 12% correspond 16 runs of received packets with small deviation (i.e., between ≥ 95 and < 97 packets and between > 103 and ≤ 105 packets).

Fig. 3 shows the average number of received packets per percentage of link packet loss with 95% confidence intervals. One can conclude that most of the time the same number of packets that are sent from the source were received, that is, 100 packets. It is possible to see that when the link has a 100% packet loss rate, the number of packets received is expected to be equal to the number of packets retransmitted, since based on our implementation, the first packet is never retransmitted. Also, due to the moving average, it unclear how to handle this packet, i.e., whether it arrives or not at its destination.

For link losses between 30% and 90%, one may see that the proposed framework had a rough time dealing with them. Nonetheless, our framework still provides high packet delivery rates.

Throughout the different runs, the lost packet is not always the same. This can deter the moving average, hence making the switch not activate packet cloning, leading to some packet loss.

Finally, from 30% to 10%, the link has a very low packet loss rate. Since the packets are not lost, and as such, their time difference will be rather small, and with that, the switch will not trigger packet cloning. However, there are always some exceptions to the rule, happening most of the time a (one) packet is retransmitted. Why not two? Because if almost no packets are lost, probabilistically speaking, almost every time 100 ± 1 packets will arrive at the end host, thus justifying our results.

2) The number of retransmitted packets: In this scenario and regarding the types of retransmitted packets, we identified the following types: (i) retransmissions with spurious retransmissions in them, (ii) retransmissions that were affected by the problem of packet congestion or busy CPU cycles, and (iii) the normal retransmissions. As the name suggests, the latter retransmissions are those where nothing out of the ordinary happens, i.e., cloned packets are sent via the alternative route to their original destination. We noticed that retransmissions with spurious packet retransmissions are truly random and either repeat 1 or 2 packets throughout all those sent.

Lastly, there are the retransmissions that have happen when the problem might have occurred. They were simply marked to distinguish them from others. In most cases (i.e., 75% of the time) 98 packets were retransmitted. It is also possible to see a higher deviation from the expected value.

IV. Conclusions And Future Work

This article proposes a state consistency framework leveraging packet cloning and piggybacking for programmable network data planes using P4. Performance evaluation results have shown that despite link failures higher than 70%, our framework still managed to deliver more than 95% of packets successfully.

References

[1] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, A Survey on Software-Defined Networking, IEEE Commun. Surv. Tutorials, 2015.
[2] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, Flow-level state transition as a new switch primitive for SDN, Comput. Commun. Rev., vol. 44, no. 4, pp. 377378, 2015.
[3] P. Bosshart et al., P4: Programming protocol-independent packet processors, Comput. Commun. Rev., vol. 44, no. 3, pp. 8795, 2014.
[4] p4c/v1model.p4 at master p4lang/p4c GitHub. [Online]. Available: https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4. [Accessed: 29-Sep-2020].
[5] The P4 Language Consortium, P4 16 Language Specification v1.2.1, p. 129, 2018.
[6] p4c/psa.p4 at master p4lang/p4c GitHub. [Online]. Available: https://github.com/p4lang/p4c/blob/master/p4include/psa.p4. [Accessed: 30-Sep-2020].
[7] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger, RFC 7323 TCP Extensions for High Performance, 2014.
[8] M. Mathis and J. Mahdavi, RFC 2018 TCP Selective Ack Options. pp. 113, 1996.