Emulation of Multipath Transmissions in P4 Networks with Kathará

Marcel Großmann and Tobias Homeyer

Computer Networks Group, University of Bamberg
Bamberg, Germany
Email: marcel.grossmann@uni-bamberg.de | tobias.homeyer@stud.uni-bamberg.de

***Abstract*—Packets sent over a network can either get lost or reach their destination. Protocols like TCP try to solve this problem by resending the lost packets. However, retransmissions consume a lot of time and are cumbersome for the transmission of critical data. Multipath solutions are quite common to address this reliability issue and are available on almost every layer of the ISO/OSI model. We propose a solution based on a P4 network to duplicate packets in order to send them to their destination via multiple routes. The last network hop ensures that only a single copy of the traffic is further forwarded to its destination by adopting a concept similar to Bloom filters. Besides, if fast delivery is requested we provide a P4 prototype, which randomly forwards the packets over different transmission paths. For reproducibility, we implement our approach in a container-based network emulation system called Kathará.**

***Keywords*—P4; Multipath; Emulation; Kathará;**

I. Introduction & Motivation

When data is sent over the network, in many cases a packet can either get lost or reach its destination. Many protocols try to solve this problem like TCP, which resends packets that are lost. But this retransmission consumes additional time and is not the best solution for sending critical data. We try a different approach by sending critical/needed data to its destination by duplicating packets. For example, if a person visits a hospital for a remote surgery, it is important that the data is transmitted quickly and reliably. To achieve this, this paper proposes a solution to duplicate important packets in order to send them to their destination via multiple routes. The receiving switch ensures that only a single copy of the traffic is further forwarded to its destination. To ensure that only one copy is forwarded, this paper proposes a concept where a hash is created over a packet and its value is stored in the switching register in a concept that is similar to Bloom filters [9].

Besides, multipath transmissions can be used to increase data delivery speed by splitting the path. Hereby, data is transmitted over a network by randomly distributing it over different routes to reach its target, which speeds up transmission time.

Motohashi et al. [8] created a multipath P4 implementation for wireless access points, which randomly distributes the traffic. Their prototype is only capable to transmit UDP packets. In our approach, we created two different multipath scenarios for P4 routers by either splitting or duplicating the traffic for both transport layer protocols.

Lindner et al. [7] try to solve this problem with a different approach, where they create a specific header. Our approach offers more flexibility as it can be reused for all protocols and does not necessarily depend on specific protocol parameters (only if the hash is created over such parameters) and it is not needed to create a new header to find duplicate packets.

Our goal is to create a reproducible emulation of multipath transmissions in P4 enabled networks. Therefore, we created different prototypes with a P4 network emulated in Kathará that can send packets over two different paths. First, a random_split setup is created, which forces packets to randomly traverse those paths. Second, a duplicate prototype is implemented, which clones packets to send them over both paths. It ensures that duplicates are removed again on the receiving switch. Our implementation and trial configurations are made publicly available on Github under *uniba-ktr/p4_multipath*[1] to ensure reproducibility of the experiments.

II. Foundation

A. Kathará

Kathará [2] is an open source container-based network emulation system and is the spiritual successor of the notorious Netkit. It can be used to test production networks in a sandbox environment or to develop new network protocols.

In Kathará, each device is emulated by a container, which runs on either Docker or Kubernetes, and the virtual network devices are connected by virtual L2 Local Area Networks (LANs). Each container can run a different Docker image and already provided ones include P4, Quagga, FRRouting, OpenVSwitch, and many more. Besides, it is also possible to use custom container images. We provide a multi-architecture P4 image *unibaktr/ubuntu:p4*[2], which is based on an *ubuntu* image that comes with the necessary networking tools. Kathará uses the concept of network scenarios, where a directory contains a file with the network topology, called `lab.conf`. Files and folders can be created for each device, which represent its configuration.

B. Software-Defined Network

The objective of Software-Defined Networks (SDNs) is the realization of a powerful transport architecture for a next generation packet-switched high speed network and the intelligent orchestration and provisioning system that enables the demand-driven allocation of virtualised resources [5].

[1]https://github.com/uniba-ktr/p4_multipath
[2]https://hub.docker.com/r/unibaktr/ubuntu

This document is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0):
http://creativecommons.org/licenses/by-sa/4.0 This CC license does not apply to third party material (attributed to another source) in this publication.

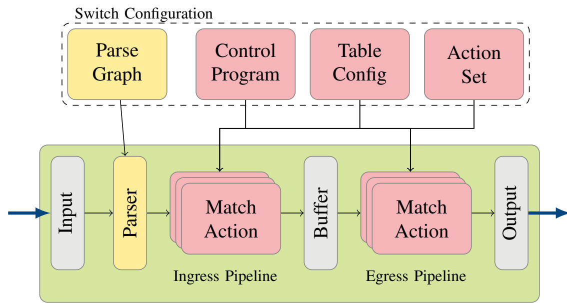

Figure 1. P4's abstract forwarding model [cf. 3]

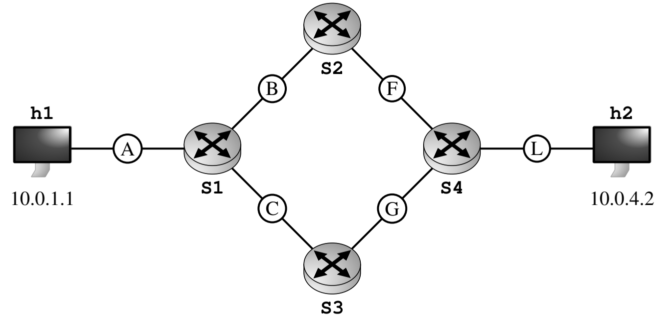

Figure 2. Network topology

SDN is vendor neutral as the data plane switches, SDN controller, and network control applications are separate entities and can be provided by different vendors [6, p. 459-460].

C. OpenFlow

OpenFlow provides a simple way to enable the control plane to program hardware and software from different vendors. The OpenFlow interface is used to match packets with rule tables based on header fields (e.g. IP addresses, MAC addresses, etc.). Each iteration of OpenFlow has shown that more header fields and rule tables are needed for switches to expose their capabilities to the controller [3].

D. Programming Protocol-Independent Packet Processor

Programming Protocol-Independent Packet Processor (P4) [3] is a high-level language. P4 is capable of working with an SDN control protocol such as OpenFlow. The P4 programming language is used to increase flexibility and allows programmers to change how a packet is processed at deployed switches. In addition, P4 is protocol-independent and the switches are not bound to a specific network protocol. Also the programmer is able to describe the packet processing functionality independent of the hardware.

Instead of extending the OpenFlow specification, *"future switches should support flexible mechanisms for parsing packets and matching header fields, allowing controller applications to leverage these capabilities through a common, open interface (i.e., a new "OpenFlow 2.0" API)"* [3]

If a packet arrives at the P4 parser, as depicted in Figure 1, it extracts the header fields from the packets. The header fields are used to define the protocols supported by the switch. After that, the header fields are used in the input and output port "match+action" tables, which determine into which output port and queue the packet is placed. At the input port it is decided whether a packet should be discarded, forwarded, flow control triggered, or replaced. The output port "match+action" is used to perform pre-instance modification to the packet header, e.g., multicast copies [3].

The P4 language is used to express how packets are processed by the data plane of a programmable target. It is only designed to specify data plane functionality and does not consider the control plane. In conventional switches the data plane functionality is defined by the manufacturer. The control plane is used for controlling the data plane by managing entries of the routing tables, configuring specialized objects, and by processing control packets.

The latest version of P4 is $P4_{16}$ (version 1.2.2)[3] and has made many non-backward compatible changes compared to $P4_{14}$ (version 1.0) to achieve a more stable language definition. Many functions were outsourced into libraries, e.g., counters, checksum units, etc., to transform P4 from a complex language with 70+ keywords into a small core one with around 40 keywords. Due to these changes, the new keyword "external" was introduced to describe library elements.

Each manufacturer must provide both a P4 compiler and an accompanying architecture definition for their target. The input controls provide information to the P4 program and the output control can be written by the P4 program.

The Very Simple Switch (VSS) is an example architecture provided by the P4 specification[3]. It can receive packets through input ports where the arbiter performs a checksum check and drops failed packets. The VSS has one parser, a match-action pipeline, and a deparser. When a packet is deparsed, it is emitted through an Ethernet output port. There are three special Ethernet output ports:

- drop port, which discards a packet,
- recirculate port, which sends a packet back to an Ethernet input port,
- and CPU port, which sends a packet to the control plane.

III. Multipath Prototypes

Our prototypes use the same topology depicted in Figure 2 with four switches, namely *s1* to *s4*, which are running our *unibaktr/ubuntu:p4* image and two computers, *h1* and *h2*, which are equipped with our *unibaktr/ubuntu* base image[2]. Currently, the switches are configured manually and only show the capabilities of a P4 program, where its main functionality to determine the multipath behavior resides on *s1* and *s4*, which connect the network to the end users respectively. A further hierarchical execution of the distribution algorithms is not covered by now.

A. Multipath Random Split

This prototype uses two tables for the IPv4 addresses, the first one is instantiated on *s2* and *s3*, as only normal forwarding is needed here. The second table is integrated on *s1* and *s4* to forward the packet either to *s2* or *s3* with the function random_split_group_to_nhop of Listing 1. When a packet is forwarded to the IP 10.0.4.2 then the random_split_group method is called, which needs three parameter, first the

[3]https://p4.org/p4-spec/docs/P4-16-v1.2.2.html

random_split_group_id, second the threshold and last the maxNum parameter; maxNum is used to create a random number. If it is greater than the threshold *egress* port 0 is used, such that the packet is forwarded to *s2*. And if this is not the case, the packet is forwarded to *s3*. This is analogous for the other direction, when a packet is sent from *h2* to *h1*.

Listing 1. Ingress of random_split prototoype

```
table random_split_group_to_nhop {
    key = {
        meta.random_split_group_id: exact;
        meta.random_split_port: exact; }
    actions = {
        drop;
        set_nhop; }
    size = 1024; }
```

B. Multipath Duplication

The duplication prototype exists to ensure a save, fast, and reliable route for all packets from *h1* to *h2*. In order to achieve this, all packets are duplicated and sent over L3. All packets from *h1* use the standard route via *s3* to reach *h2*. However, to ensure a fast and above all reliable route, the packets are cloned at *s1* and the duplicate is sent via *s2* to *h2*. The packet that reaches switch *s4* first is forwarded to *h2*, the second one is dropped.

For example, when *h1* sends a packet p to *h2*, the p is transmitted from *h1* to *h2* via *s3* and *s4*. At *s1*, p is duplicated and $\tilde{p}$ is sent to *h2* via *s2* and *s4*. The packet that reaches *s4* last will be dropped and the other one is forwarded to its destination.

At *s1* a mirroring_add function is executed, which allows to clone every packet at the ingress port and route it over the secondary egress port of *s1*. Listing 2 shows the action clone_packet, which specifies that all packets are cloned with ingress to egress (I2E) when the REPORT_MIRROR_SESSION_ID is equal to 500.

Listing 2. Ingress of duplicate prototype

```
action clone_packet() {
    const bit<32> REPORT_MIRROR_ID = 500;
    clone(CloneType.I2E, REPORT_MIRROR_ID); }
```

Switch *s4* runs with its own P4 file and uses a concept similar to Bloom filters [9] for packet deduplication. For each packet arriving at *s4*, a hash value is created. This hash value is stored in a register and is compared to other hash values that are already stored to find duplicate packets. The hash value is created with the help of the IP source address, IP destination address, source and destination port of the transport layer protocol with the help of the CRC hash algorithm from P4[3]. The created hash value of a packet is compared to the value of the previous packets and when the same value is recognized, the packet is dropped and the saved hash value is removed. Our prototype can hold up to five hash values and save them in the register. Registers work similar to an array in comparison to other programming languages. The hash values must be stored in a register and not in a table, because it is necessary to update the value regularly, which is not possible with a P4 table. To get a few packets between duplicates, it is necessary to store more than one value, and this can be increased if necessary. Each time a new packet arrives, the hash value is stored in register 0 and the other values are moved to the next register (e.g. the value stored in 0 is then stored in 1 and the value of 1 in 2, etc., only the value from register 4 is removed).

IV. Evaluation

In order to evaluate both prototypes, we used Wireshark to sniff the traffic off all Collision Domains (CDs). Therefore, we created Python clients that can send UDP or TCP packets to a corresponding server across the topology.

In order to use Wireshark with Kathará it is necessary to connect each CD with a Wireshark container. For random split we observed with Wireshark running on the CDs A, B, C, and L that not always the same route is taken to reach *h2* from *h1*.

To show that the duplication behavior works as intended, it is necessary to see exactly one UDP or TCP message on all six CD when a UDP or TCP test packet (with the same data) is sent over the network. This means that the UDP or TCP packet sent over the network is duplicated at *s1* and thus also sent over *s2*, and at *s4* one of the two packets is discarded and only one message reaches its destination.

We verified that the creation of a hash over the packet is working as expected and that hash collision rarely happen. It is very easy to replace the hash function, or its parameter, to reduce the risk of hash collisions if necessary. With the help of the log message, it could be verified that a hash is created for the UDP or TCP packet and compared with the values stored in the register.

V. Conclusion and Future Work

The development of the prototypes was successful and they work with UDP and TCP. P4 with the Kathará emulator is able to duplicate and deduplicate incoming packets. Packet deduplication with the creation of a hash over a packet works well and is easily adaptable for other protocols. The disadvantage, however, is that the switch must have a register and, depending on the hash algorithm used, there is a risk of hash collisions.

The duplicate prototype is only a small step to ensure a reliable and fast transmission of packets over a network. It is necessary to add multiple features to it, such as load balancing, congestion control, and the automatic re-routing of packets when a switch fails. The reliability of the random_split prototype can be enhanced by integrating Raptor Codes [4] in the corresponding P4 implementations. Currently, everything is set up manually and if a switch has crashed, it will still try to send the packet via the crashed route.

A next step of our resilience concept is given by the integration of network control over SDN functionality by integrating a controller like ONOS [1] into our prototypes. It would allow a better control of the control layer. P4 is designed to define the interface by which the control and data plane can communicate, but P4 cannot be used to describe the control plane functionality of the target [3]. With the help of SDN we can develop ways to overcome these deficiencies.

References

1. Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W., and Parulkar, G.: ONOS: Towards an Open, Distributed SDN OS. In: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking. HotSDN '14, 1–6. Association for Computing Machinery, Chicago, Illinois, USA (2014). DOI: 10.1145/2620728.2620744
2. Bonofiglio, G., Iovinella, V., Lospoto, G., and Di Battista, G.: Kathará: A container-based framework for implementing network function virtualization and software defined networks. In: NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium, pp. 1–9 (2018). DOI: 10.1109/NOMS.2018.8406267
3. Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D.: P4: Programming Protocol-Independent Packet Processors. SIGCOMM Comput. Commun. Rev. 44(3), 87–95 (2014). DOI: 10.1145/2656877.2656890
4. Eittenberger, P.M., and Krieger, U.R.: Performance Evaluation of Forward Error Correction Mechanisms for Android Devices Based on Raptor Codes. In: Fischbach, K., and Krieger, U.R. (eds.) Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance, pp. 103–119. Springer International Publishing, Cham (2014)
5. Kreutz, D., Ramos, F.M.V., Veríssimo, P.E., Rothenberg, C.E., Azodolmolky, S., and Uhlig, S.: Software-Defined Networking: A Comprehensive Survey. Proceedings of the IEEE 103(1), 14–76 (2015). DOI: 10.1109/JPROC.2014.2371999
6. Kurose, J.F., and Ross, K.W.: Computer Networking: A Top-Down Approach (6th Edition). Pearson (2012)
7. Lindner, S., Merling, D., Häberle, M., and Menth, M.: P4-Protect: 1+1 Path Protection for P4. In: Proceedings of the 3rd P4 Workshop in Europe. EuroP4'20, 21–27. Association for Computing Machinery, Barcelona, Spain (2020). DOI: 10.1145/3426744.3431327
8. Motohashi, H., Nguyen, K., and Sekiya, H.: Enabling P4-based Multipath Communication in Wireless Networks. In: 2020 IEEE Globecom Workshops (GC Wkshps, pp. 1–5 (2020). DOI: 10.1109/GCWkshps50303.2020.9367468
9. Tarkoma, S., Rothenberg, C.E., and Lagerspetz, E.: Theory and Practice of Bloom Filters for Distributed Systems. IEEE Communications Surveys & Tutorials 14(1), 131–155 (2012). DOI: 10.1109/SURV.2011.031611.00024