UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Hop recording and forwarding state logging: Two implementations for path tracking in P4

Knossen, S.; Hill, J.; Grosso, P.

DOI
10.1109/INDIS49552.2019.00010

Publication date
2019

Document Version
Final published version

Published in
Proceedings of 6th Annual International Workshop on Innovating the Network for Data Intensive Science (INDIS) 2019

License
Article 25fa Dutch Copyright Act (https://www.openaccess.nl/en/in-the-netherlands/you-share-we-take-care)

Link to publication

Citation for published version (APA):
Knossen, S., Hill, J., & Grosso, P. (2019). Hop recording and forwarding state logging: Two implementations for path tracking in P4. In *Proceedings of 6th Annual International Workshop on Innovating the Network for Data Intensive Science (INDIS) 2019: Held in conjunction with SC19: The International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, Colorado, November 17-22, 2019* (pp. 36-47). IEEE Computer Society. https://doi.org/10.1109/INDIS49552.2019.00010

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)

Download date:24 Sep 2023

2019 IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)

Hop Recording and Forwarding State Logging: Two Implementations for Path Tracking in P4

1st Silke Knossen
Security and Network Engineering master
University of Amsterdam
Amsterdam, The Netherlands
silkeknossen@gmail.com

2nd Joseph Hill
Systems and Networking Lab (SNE)
University of Amsterdam
Amsterdam, The Netherlands
j.hill@uva.nl

3rd Paola Grosso
Systems and Networking lab (SNE)
University of Amsterdam
Amsterdam, The Netherlands
p.grosso@uva.nl

***Abstract*—Full information on the path travelled by packets is extremely important for network management and network security. We implemented two path tracking methods in hardware with P4. The first approach tracks a packet's path by recording each node along the path of a packet (*hop recording*). The complete path a packet took can be extracted from the packet in the last node of the path. The second approach tracks a packet's path by logging the forwarding state of a network (*forwarding state logging*). The complete path can be reconstructed based on the node where the packet entered a network. We conducted experiments with the two implemented approaches and showed that the paths of the packets are reconstructed correctly. The advantage of using P4 is that the control plane only gets involved when the path of a packet is reconstructed. We finally show how our work provides a working tool in P4 networks that can be used to gain deep insights in traffic patterns.**

***Index Terms*—Programmable networks, path tracking, IPv6, P4.**

I. INTRODUCTION

Knowledge of the paths that data took through the network can be useful to provide more context to solve security incidents. Additionally this information can support more targeted monitoring as well as network and traffic engineering.

The information required to reconstruct the path of data can be gathered by *path tracking*. Traditional methods of tracking network traffic such as NetFlow [1] are done in the control plane (software) of forwarding devices such as routers and switches. Since the control plane executes on the router's processor, these methods can be resource intensive. To limit the resource utilization, NetFlow often is configured to only track the path of every n^{th} packet, which can lead to incomplete information [2]. This prevents to fully unleashing the potentials of full path knowledge.

With the development of the programming language Programming Protocol-independent Packet Processors (P4), it is now possible to program the data plane (hardware) of forwarding devices [3]. This makes it possible to gather network traffic data in-band, without requiring work by the control plane. This has the potential to track the path of all packets and to provide the information more efficiently.

Tracking packet paths in P4 can be done in several ways, as it was envisioned in [4]. In this article we will present our implementation of two such methods: the first method is to add the ID numbers of the forwarding devices which have routed the packet to the IP header of the packet (*hop recording*); the second method relies on recording *forward state logging*.

We will first familiarise the readers with the P4 language (Sec.II) and provide them with an overview of the related work (Sec.III). In Sec.IV we will explain path tracking and introduce our two approaches. In the subsequent sections (Sec.V, Sec.VI and Sec.VII)) we explain the use of the IPv6 extension header and cover extensively the details of the two implementations. Sec.VIII describes our experiments, while Sec.IX and Sec.X present the results we obtained for the two methods. We continue with a discussion of the pros and cons of the two approaches (Sec.XI), show how to use the collected data for visualisation (Sec.XII) and conclude the paper (Sec.XIII) with pointers to future work.

II. P4 LANGUAGE

Our implementation is done in P4, a high-level language for programming protocol-independent packet processors. P4 is designed to program the data plane of packet forwarding devices. P4 wants to offer more flexibility than currently available on forwarding devices. A first advantage is that a device can be reconfigured in real-time. A second advantage is that P4 has no predetermined definition of the format of a packet, which makes it protocol independent. This eliminates constrains on how individual packets can be examined. A third advantage is that P4 allows flexible allocation of device memory. This means that memory that is typically intended for routing tables can instead be used to store path tracking data. Some operations still have to be performed by the control plane of a forwarding device, but P4 has the potential to allow efficient path tracking of data that moves through a network.

P4 is a domain-specific language, designed specifically to describe the behavior of packet forwarding hardware. As a result, it does not have as much functionality as a programming language such as Python or C. For example, P4 generally does not support loops. The use of conditional statements is also very limited. This makes it a challenging language for implementing complex programs.

A. P4 Language Structure

A P4 program describes a forwarding model consisting of three stages: the parser, ingress match+action, and egress

978-1-7281-6666-7/19/$31.00 ©2019 IEEE
DOI 10.1109/INDIS49552.2019.00010

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on March 11,2021 at 16:36:05 UTC from IEEE Xplore. Restrictions apply.

match+action. The parser is the first stage in the model. In this stage the packet is being parsed based on user defined headers. Since P4 is protocol independent, the organisation of these headers can be a format of a protocol that is well known or something that is entirely new. After packets are parsed, a P4 program enters the ingress match+action stage where actions are taken based on the results of table lookups. This stage controls the ingress of packets into the forwarding device. Finally, the third stage - the egress match+action controls the egress of the packets. In the ingress match+action stage, the actions may define the egress port where the packet will exit the device. In the egress match+action stage this is no longer possible, and actions are performed while the P4 program assumes the packet will exit through a specific egress port.

Matches can be performed in P4 based on field values in the headers of a packet. Matches can also be performed based on other meta-data such as ingress ports. This meta-data is data that is generated during execution of the P4 program. There are different types of meta-data fields used in P4. One of them is the user meta-data, which are fields that a user can define in the program. Another type of meta-data is the standard meta-data, which fields are automatically included in the P4 design. The last type, the intrinsic meta-data, is not part of the standard P4 design, but can be used by defining a header *intrinsic_metadata*. The fields in this meta-data contain timestamps which can be used by defining them as header fields. When the intrinsic meta-data is defined, the fields automatically behave as they are defined in the P4 specification. There exists several match options including exact, ternary, and longest-prefix match. Using an exact match, the bits of a value must match exactly in order to be a match. Using a ternary match, a bit-mask is used to match. Using the longest-prefix match, the most specific match according to a bit-mask is selected. When a table entry matches a particular header field value, an action is taken based on a value in the table entry. Whenever a table miss occurs, a default action may be performed. A user can define actions based on a limited set of action primitives which are defined in the P4 specification. These primitive actions allow for the modification of fields in headers, adding headers, removing headers, sending digests, and cloning packets. When a digest is sent, multiple fields that are parsed by the P4 program can be sent in a user's defined structure. The stateful memories P4 has are limited. An example of a stateful memory in P4 is a register, which is an external object where data can be stored. The state that P4 keeps from packet to packet is also very limited due to the limited stateful memory of P4. As a result, when a more significant state change is required, this must be performed by the control plane and P4 has to send it the data required. For example, modifications to the tables in the forwarding device must be done by the control plane. There is no definition of the interface between the data plane and the control plane in the P4 specification.

There are currently two versions of P4: $P4_{14}$ and $P4_{16}$. $P4_{16}$ is the newest version of the language, which has some significant differences compared with $P4_{14}$ and is not backwards compatible. An important goal of the newest version of $P4_{16}$ was the revision of the language to provide a stable language definition. This means that the intention of creating $P4_{16}$ is that all programs written in $P4_{16}$ will remain syntactically correct and behave identically when treated as programs for future versions of the language. This is the reason why $P4_{16}$ is used in this research. Each future reference to P4 will refer to the $P4_{16}$ version of P4.

III. Related work

The interest in programmable device and in particular the focus on P4 devices is growing in the last years. The potential of these devices is being recognized, and an increasing body of literature testifies to the many areas where they are being (envisioned to be) applied to: in metro networks for latency-awareness [5]; in IP-over-optical network to visualise network performance in real time [6]; in redefining the way in which Intent-Driven networking can be implemented [7].

Many National Research and Education Networks (NRENs) are also exploring the adoption of P4 devices. The general consensus in the community is that this type of hardware can support in a more flexible way scientific applications, as well as provide enhanced telemetry, security and access to virtual network functions. For example, the GEANT community has started to hold regular meetings on this topic; ESnet is looking at P4 for its upcoming High-Touch Services. We are part of a new program launched recently in the Netherlands that focus on the development and evaluation of mechanisms for increasing the security, stability and transparency of internet communications: 2StiC (https://www.2stic.nl/). Also here P4 telemetry plays a crucial role to support the program's goals.

In this ecosystem there is a strong need for working implementations that can be shared and tested in a testbed first, and more production settings. To the best of our knowledge our work is the first working implementation of path tracking in P4 and as such we expect that it will be of interest to the communities exploring P4 adoption for telemetry.

IV. Tracking Flows

A commonly used protocol to get information about the traffic in a network is NetFlow. This is a protocol developed by Cisco that can collect and analyse IP network traffic as it enters or exits an interface [1] of a network device. Netflow tries to collect path data of a flow, which is considered a sequence of packets having some identical header field. For example, when packets have the same source and destination address, they could belong to the same flow. This method corresponds to *flow tracking*; in contrast with the focus of our research where path information from individual packets will be recorded, hence *path tracking*.

Using Netflow, flow tracking is implemented in the control plane. This can result in a large system resource utilization, which can only be handled by some network devices. This is the reason why a sampling method is often used in order to minimize the resource utilization. This method examines not every packet that a forwarding device forwards, but only every

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on March 11,2021 at 16:36:05 UTC from IEEE Xplore. Restrictions apply.

n^{th} packet. However, previous work found that this approach did not yield enough information to reconstruct the path that data took through the network [2]. It is not guaranteed that every device along the path of a packet will record it when sampling is being used. It is also possible that a packet could be missed by every device along the packet's path. An example of this is provided in figure 1. Here we assume that due to sampling only the blue nodes have recorded the packet. From this data it is not clear what the exact path is the packet took through the network. Between node B and F the packet could have, for example, only gone through node E, or through node C and then E. Even other paths are possible to reconstruct using this information.

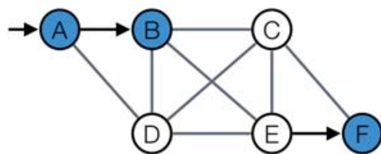

Fig. 1: Incomplete packet capture example

Tracking the path of packets with P4 could solve the difficulties NetFlow has by implementing it in the data plane of a forwarding device. With NetFlow, each device keeps some state where information about the path is stored. In order to reconstruct the path of data, each device has to be queried for information about the path of a flow. Implementing path tracking with P4 allows for a more efficient way of collecting the data required for reconstructing paths, while the impact on system resources could be minimized. In all methods described here, the actual reconstruction of the paths is done in the last node in the path. In order to do so, the information needed must be available in every node in the network. Every solution for path tracking described in this article will therefore add data to the packet which is used for the reconstruction of the path. We focused on two approaches: hop recording and logging forwarding state.

A. Hop Recording

To track the complete path of packets through a network one can record every traversed hop in the packet itself. Each node would have its own node identifier (NID) and adds it to the packet as it moves through the network. In the last node of the packet's path, the path information included in the packet can be extracted to reconstruct the complete path. The complete path would thus only exist in this node. This approach is illustrated in figure 2. The first device adds its NID (A) to the packet, in the second device, the NID (B) is appended to the data, after the last device appends its NID to the packet, the complete path would be A, B, C.

B. Logging Forwarding State

The second method of tracking the path of a packet is based on the global forwarding state of a network, which

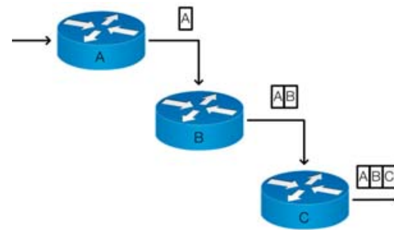

Fig. 2: Illustration of the Hop Recording method

includes all forwarding rules that are used in a network. In this method it is assumed that the current and all previous versions of the global forwarding state of a network are known. The routing of packets may change over time, for instance because of the addition of a node in the network. Knowing the global forwarding state, the complete path of a packet can be determined in the final node. To do so, information about where the packet enters the network and an identification of the version of the global forwarding state that was used to forward the packet is required. Hence, those two factors must be added to the packet at the first node along a packet's route. All the next nodes have to check whether or not they run the same version of the global forwarding state. When the state changes to a new version during the packet's route, some devices may forward the packet based on different versions of the forwarding state. In this case, the complete path the packet took can not be reconstructed, and we have to add another field to the packet (called the *Trackable* field) to indicate that the path cannot be reconstructed. Each forwarding device must examine the packet to check if the version identification in the packet is equal to the version that it uses itself. When the versions are different, the *Trackable* must be set to indicate this. In figure 3 the method is illustrated. Device A and B run the same version, but device C runs another version. In device A, the version used by the device, V1, is added to the packet first. Secondly, the Trackable field is added with default value 0. Finally, the NID of device A is added to indicate the node where the packet has entered the network. Device B checks the version field and, since it runs the same version, changes nothing. When the packet enters device C, the Trackable field is set to 1 indicating the path can not be reconstructed, since device C runs another version of the global forwarding state.

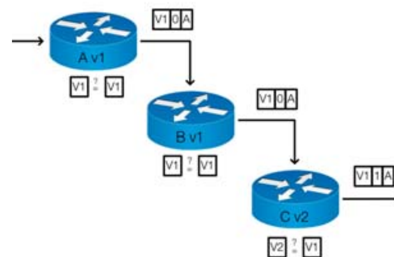

Fig. 3: Illustration of the Logging Forwarding State method

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on March 11,2021 at 16:36:05 UTC from IEEE Xplore. Restrictions apply.

V. IPv6 Extension Header

In order to add additional data containing information about the path to a packet, an extra header must be added to the packet. This header could be implemented using an entirely new protocol. However, we decided to use the *IPv6 extension header* [8], since network devices which do not recognize an IPv6 extension header will ignore it. This means the implementations will also work in a network where not all nodes are enabled with P4. When using a completely unknown header, nodes that are not enabled with P4 may discard the packet, since they are not able to parse the header correctly. An IPv6 packet may include zero, one, or multiple extension headers, which are located between the IPv6 header and the upper-layer headers in a packet.

There are different extension header types, identified by a value in the Next Header field of the IPv6 header. Only the *Hop-by-Hop Options header* is examined by every node along a packet's path through the network and for this reason it is perfectly suited for our usecase. This header is identified by a Next Header field value of 0 in the IPv6 header and has the format shown in figure 4.

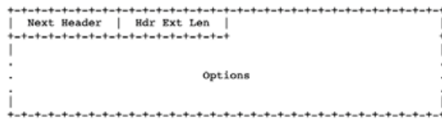

Fig. 4: The Hop-by-Hop Options header format [8]

The Next Header field is an 8-bit identifier of the type of header that is immediately following the Hop-by-Hop Options header. The Header Extension Length field is an 8-bit unsigned integer, which indicates the length of the Hop-by-Hop Options header in 8-bytes, not including the first 8 bytes. The Options field is a variable-length field, of a length such that the total Hop-by-Hop Options header is an integer multiple of 8 bytes long. A variable padding length is used to fulfil this requirement. This field also contains a variable number of *options* of the format shown in figure 5. Options added to this Options field offer us a way to add data in order to track the path the packet took through the network.

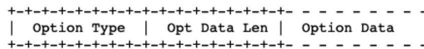

Fig. 5: The Hop-by-Hop Options header options format [8]

The Option Type field is an 8-bit identifier of the type of option. The Option Data Length field is an 8-bit unsigned integer that is the length of the Option Data Field of this option in bytes. The last field is called Option Data and it is a variable-length field that includes option-type-specific data. This is the field that can be used to add path information to the packet. Since this data is not yet covered by an existing option type, the implementation requires a non-existing option type. The Option Type field is encoded such that the three high-order bits have a special meaning. The highest-order 2 bits specify the action that must be taken in case a node using IPv6 does not recognize the Option Type. Since this research focuses on extracting information from the packet about its path at the last node, discarding the packet would lead to information loss. This is the reason why the Option will be skipped over when it is not recognized. This action is indicated by the value 00. The third-highest-order bit of the Option Type specifies whether or not the Option Data of that specific option can change during the route of a packet to its destination. Since the options are examined by the nodes along the route and may change depending on the action that has to be applied, the value of this bit is set to 1 indicating the option data may change en route. The remaining low-order 5 bits are chosen such that the full 8-bit value is a non-existing option type. For instance, all bits with value 1 makes the Option Type field value a non-existing option. This results in a option type value of 0x3F.

In the following section we will describe in detail our two implementations.

VI. Hop Recording

A. The use of the Hop-by-Hop Options header

Since the Option Data field in the Hop-by-Hop Options extension header allows data to be added to the packet, this field is used to add the NIDs to the packet. As a packet moves through the network, the first node would add the Hop-by-Hop Options extension header with one option containing the first NID. Each next node along the packet's path to its final destination would add one option to the extension header. The length of the Option Data field is chosen such that the padding will be a constant value. Since the total Hop-by-Hop Options extension header must have a size which is a multiple of 8 bytes, the smallest option size to add - while containing a constant padding size - is 8 bytes. It results in a Data Option field length of 6 bytes. Every time a node adds its NID the extension header will thus grow with a size of 8 bytes. If the Option Data field length would be smaller, the padding must be recalculated each time a node adds an option. The extension header format with field values used for hop recording is shown in figure 6. In this figure the extension header contains two options, each with its own NID. This figure illustrates how the extension header would be organised when the packet has traveled across two P4 nodes in the network.

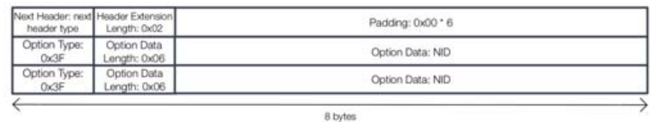

Fig. 6: Hop-by-Hop Options extension header format used for implementation of Hop Recording

B. The implementation in P4

The implementation of hop recording in P4 can be split into four different stages: 'packet header parsing', 'initiating

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on March 11,2021 at 16:36:05 UTC from IEEE Xplore. Restrictions apply.

the Hop-by-Hop Options extension header', 'adding an option to the Hop-by-Hop Options extension header', and 'sending the path information to control plane'.

1) Packet header parsing: The first step P4 will take is trying to parse a packet based on different headers in order to allow examining the data. The headers are parsed in similar order as their appearance in the packet. To enable this, P4 requires a structure describing the organisation of a packet header. This includes defining each field and its size in bits. Since a packet sent into the network will be an Ethernet packet, the first header to be parsed is an Ethernet header. For this implementation this header will not be examined, but it has to be parsed in order to parse the next headers. The IPv6 header will be parsed next, containing eight fields which are shown in listing 1.

Listing 1: IPv6 header definition

```
header ipv6 {
    bit<4>    version;
    bit<8>    typ;
    bit<20>   fl;
    bit<16>   plen;
    bit<8>    nh;
    bit<8>    hlim;
    bit<128>  src;
    bit<128>  dst;
}
```

The Hop-by-Hop Options extension header is the next header to parse. To enable adding multiple options to the Hop-by-Hop Options extension header a header stack structure can be used in P4. This is a way to enable stacking multiple headers with the same format, represented as an array of headers [9]. With this data structure it is possible to parse multiple options in the extension header. The organisation of the extension header is shown in listing 2.

Listing 2: Hop-by-Hop Options extension header definition

```
header extension {
    bit<8>   nh;
    bit<8>   hlen;
    bit<48>  pad;
}

header extension_options_stack {
    bit<8>   typ;
    bit<8>   len;
    bit<48>  nid;
}
```

The listing shows that the extension header is split into two parts, the first two fields of the extension header and the padding, and the option fields of the extension header. This is done to enable the header stack data structure to stack a variable number of options. The initiation of the header stack can be found in listing 3.

Listing 3: Hop-by-Hop Options extension header Options header stack definition

```
extension_options_stack[MAX_SIZE] options;
```

Since loops don't exist in P4, parsing a variable amount of options in the extension header is done with recursion. The algorithm that makes this possible is shown in listing 4. In this listing two parse states are shown. The first one parses the first three fields of the extension header. In this parse state a counter is set to the number of options in the extension header. The number of options is the Extension Header Length field in the Hop-by-Hop Options extension header. This operation can be found in line 3 of listing 4. The second parse state parses the options in the extension header. This counter is decreased each time one of the options in the extension header is parsed. When the counter reaches the value 0, parsing all options is completed.

Listing 4: Parsing extension header with a variable number of options

```
state parse_ext {
        pkt.extract(hdr.ext);
        umd.cnt = hdr.ext.hlen;
        transition select(umd.cnt) {
            0:        accept;
            default:  parse_ids;
        }
    }

state parse_options {
        pkt.extract(hdr.options.next);
        umd.count = umd.count - 1;
        transition select(umd.cnt) {
            0:        accept;
            default:  parse_options;
        }
    }
```

Due to this limitation of the Netronome SmartNICs we used for our implementation, our header stack size has a maximum of 16. Hence, the *MAX_SIZE* value in listing 3 can be set to a maximum value of 16 in order to compile the P4 program. This means a maximum of 16 NIDs could be added to a packet. In some scenarios this may not be sufficient. For this reason a different method of parsing the packet is created. The organisation of the extension header remains the same for this method and thus can be seen in listing 2, but this method does not make any use of a header stack structure. It does not parse the full extension header, but only the three fields shown in the first header structure of listing 2. These fields are the last fields that the parser will parse. This way the P4 program can add an option using the second header structure of listing 2 to the extension header. Note that since the P4 program does not parse the option fields, it does not know the values in the options of the Hop-by-Hop Options extension header.

The first time a P4 enabled node examines a packet, the packet will not contain an extension header yet. To avoid the P4 program from trying to parse the extension header when it is not there, the Next Header field of the IPv6 header is

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on March 11,2021 at 16:36:05 UTC from IEEE Xplore. Restrictions apply.

checked on its value. If this value indicates another header then the Hop-by-Hop Options extension header, parsing the headers will be completed after parsing the IPv6 header.

2) Initiating the Hop-by-Hop Options extension header: When the headers are parsed, the P4 program will add a Hop-by-Hop Options extension header when it does not exist in the packet yet. This action is shown in listing 5. To be able to add a header to a packet, it must be set as a valid header. This operation can be found in line 2 of the listing. The next step in the action is to set all the field values of the Hop-by-Hop Options extension header. Since it is added immediately after the IPv6 header, the Next Header field of the extension header becomes the Next Header field of the IPv6 header. This one changes to the value '00' indicating the Hop-by-Hop Options extension header. These operations can respectively be found in line 3 and 5 of listing 5. To ensure the packet can be parsed correctly at its final node in the network, the length fields in the headers must contain the correct values according to the description in section VI-A.

This action is only applied when the node that will forward the packet is the first P4 node along the route of the packet. To determine whether or not this action must be applied, a check for the presence of the extension header is made.

Listing 5: Action to initiate an extension header for the Hop Recording implementation

```
action init_exthdr() {
        hdr.ext.setValid();
        hdr.ext.nh = hdr.ipv6.nh;
        hdr.ext.hlen = 0;
        hdr.ipv6.nh = 0x00;
        hdr.ipv6.plen = hdr.ipv6.plen + 8;
    }
```

3) Adding an option to the Hop-by-Hop Options extension header: Each time a packet is forwarded by a P4 enabled device, the P4 implementation must always add an option to the extension header. Since no option is parsed in the parser of the implementation, each option added to the packet, is added as first option in the extension header. This results in creating a packet with options in the reversed order of addition by the nodes along the packet's path. The action for adding an option is shown in listing 7. The option header from listing 2 is set as a valid header. All the fields of the option are set with their corresponding values. The Option Data field is set to the NID of the operating P4 node. The NID is read from a register where it has been stored. A register in P4 is an object that is used to store values and has a state that can be read and written by both the control plane and data plane. To use registers they have to be defined in the P4 program with a size (indicating the number of register fields), field size and name. Listing 6 shows this register definition with one field with a size of 48 bits, because the Option Data field contains a value of 48 bits. This register is called 'nid' and can be read by the function *nid.read()* when it is defined. This operation can be found in line 6 of listing 7. The first argument in this function call specifies the field to read into. The second argument specifies the index of the register field.

Since this action adds an option to the extension header, the fields with information about the lengths of headers in the IPv6 header and extension header are updated according to the description in section VI-A.

Listing 6: Register definition for NID

```
register<bit<48>>(1) nid;
```

Listing 7: Action to add an option to extension header

```
action add_option() {
        hdr.option.setValid();
        hdr.ext.hlen = hdr.ext.hlen + 1;
        hdr.option.typ = 0x3F;
        hdr.option.len = 0x06;
        nid.read(hdr.option.nid, 0);
        hdr.ipv6.plen = hdr.ipv6.plen + 8;
    }
```

4) Sending the path information to the control plane: The specific data fields required to reconstruct the path could have been sent to the control plane using a digest. However, this implementation only parses the first three fields of the extension header to allow for adding an unlimited number of options. This method thus can not send all individual options to the control plane through a digest, because then all options must be parsed. Another way of sending data to the control plane is to clone the packet and sending the clone to the control plane. Whenever a packet is cloned, this means the entire packet is duplicated. P4 can still make a difference between the original packet and the clone based on the the *instance_type* value in the *standard_metadata* struct. The instance type indicates a clone with value 0x8. Sending the clone to the control plane can be done by sending every packet that has instance type value 0x8 to the control plane.

To be able to calculate throughput, a timestamp is added to the clone. In the *intrinsic_metadata* in P4, a timestamp is automatically assigned to the *ingress_global_time-stamp* field when starting the parsing of the packet in the P4 program. To be able to use this field, the *intrinsic_metadata* header must be defined in the implementation with the field that will be used as shown in listing 8. This header is added to the cloned packet at its very beginning in order to extract it from the packet in the control plane.

Listing 8: Intrinsic metadata header with timestamp field

```
header intrinsic_metadata_t {
    bit<64> ingress_global_timestamp;
}
```

C. Extracting path data from packets in the control plane

To extract the data that indicates the complete path a packet took through the network, the Option Data fields - each with

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on March 11,2021 at 16:36:05 UTC from IEEE Xplore. Restrictions apply.

an NID - in the packet must be examined. This is done with the Scapy library of Python which includes functions to unpack a packet based on layers and fields [10], [11]. With the function *getlayer(HopbyHopOptExtHdr)* the extension header layer is extracted from the packet. From this object the value in each of the Option Data fields is obtained and appended to a list, which forms the path of the packet. The paths must be reversed to get the chronological order of the NIDs along the packet's path. The length of the payload in the packet is also obtained by extracting the value from the Payload Length field of the IPv6 header in the packet. This Payload Length field is used, because it shows the size of the packet with extension header in bytes that is being sent through the network. This way, a larger extension header is indicated by a packet of more bytes. Also the timestamp is extracted from the packet. This is done by extracting the first 8 bytes of the packet and converting it to an integer. The first 4 bytes indicates the time in seconds, and the last 4 bytes the time in nanoseconds. This data is used to calculate throughput of the path in bits per second.

VII. Logging Forwarding State

A. The use of the Hop-by-Hop Options header

As in the first method, the Option Data field in the Hop-by-Hop Options extension header can be used to add the version of the current global forwarding state, and the NID of the node where the packet enters the network. As a packet moves through the network, the first node would add the Hop-by-Hop Options extension header with options containing the three factors as illustrated in figure 3. Since each next node only has to examine the data in this Option Data field, the extension header will have a fixed size. To minimize the size, the three factors can be combined in one option in the extension header.

This option would thus be encoded in a way that the highest-order x bits would specify the global forwarding state version used by the first node of a packet's path at the time of sending the packet into the network. The next y bits specify wheter or not the path of the packet can be reconstructed. The remaining low-order $48-(x+y)$ bits would specify the NID of the node where the packet enters the network. The values chosen for x and y can differ according to the size needed in a certain network. In this implementation the following sizes are chosen. For x the value of 8 bits is chosen, because this will suffice for the purpose of the experiments. For the value of y 4 bits is chosen, although this field can only indicate two different states for the trackability of a path. However, for debugging reasons it is easy to work with values of a size which is a multiple of 4 bits, because the hexadecimal notation is used to show a packet's data. This results in a size of 36 bits for the NID of the first node. The format of the Hop-by-Hop Options extension header with field values used for this method are shown in figure 7.

B. The implementation in P4

The implementation of hop recording in P4 can be split into four different sections according to the design of this approach: 'packet header parsing', 'creating the Hop-by-Hop Options extension header', 'version comparison', and 'sending the path information to the control plane'.

1) Packet header parsing: As in the Hop Recording method, the headers must be parsed by the P4 program in order to examine data and add data to the packet. This method can only be used in IP networks, because it depends on the routing protocol used by the Network layer. The packets thus will use an Ethernet header and an IPv6 header that will be parsed respectively in P4 in corresponding to the Hop Recording approach. The organisation of the Hop-by-Hop Options extension header as described in section VII-A is shown as a header definition in listing 9.

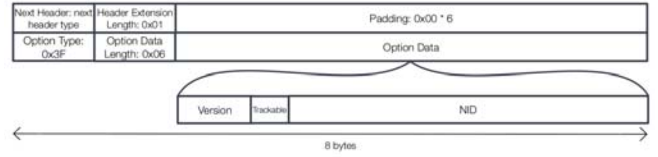

Fig. 7: Hop-by-Hop Options extension header format used for the logging forwarding state implementation

Listing 9: Definition of the Hop-by-Hop Options extension header used in the implementation of Logging Forwarding State

```
header ipv6_extension {
    bit<8>  nh;
    bit<8>  hlen;
    bit<48> pad;
    bit<8>  otyp;
    bit<8>  olen;
    bit<8>  version;
    bit<4>  track;
    bit<36> entry;
}
```

This header will only be parsed when it is already present in a packet. This can be checked by examining the Next Header field in the IPv6 header. If this value is '00' it indicates that the Hop-by-Hop Options extension header is present in the packet.

2) Creating the Hop-by-Hop Options extension header: When the headers are parsed, the P4 implementation will add the Hop-by-Hop Options extension header when it does not exist in the packet yet. This action is shown in listing 11. As with the Hop Recording method, the extension header is set as a valid header to add it to the packet first. Subsequently, the field values of the extension header must be set to the correct values. To get the NID and the version of the current forwarding state used by the first node of a packet's path, registers are used. The register definitions are shown in listing 10. The values stored in these registers are read into the version and entry fields of the extension header in the implementation. These operations can be found in line 7 and 8 of listing 11. The Trackable field is initially set to value 0, because the first node is defining the global forwarding state that must be used in order to reconstruct the path of the packet. The value 0

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on March 11,2021 at 16:36:05 UTC from IEEE Xplore. Restrictions apply.

indicates the packet's path can be tracked. When the value changes to 1, this indicates the path can not be reconstructed. Finally, the fields in the IPv6 header must be updated to the correct values. The field that will change are the Next Header and Payload Length field due to the addition of the extension header.

Listing 10: Register definitions for the Logging Forwarding State implementation

```
register<bit<8>>(1) version;
register<bit<36>>(1) nid;
```

Listing 11: Action to initiate an extension header in the Logging Forwarding State implementation

```
action init_extension() {
        hdr.ext.setValid();
        hdr.ext.nh = hdr.ipv6.nh;
        hdr.ext.hlen = 1;
        hdr.ext.otyp = 0x3F;
        hdr.ext.olen = 0x06;
        hdr.ext.track = 0x0;
        id.read(hdr.ext.entry, 0);
        version.read(hdr.ext.version, 0);
        hdr.ipv6.nh = 0x00;
        hdr.ipv6.plen = hdr.ipv6.plen + 16;
    }
```

3) Version comparison: The forwarding state version comparison is done in the egress match+action stage in the P4 model. This stage is used because of load balancing considerations, since the ingress match+action stage already applies the forwarding rules to the packets. All the actions taken in this stage are shown in listing 12. If there already exists a Hop-by-Hop Options extension header in the packet, the packet is being examined by a node along the route of a packet other than the first node. This node will compare the version of the global forwarding state that it is running itself to the version that is in the packet. Since each node has the version value stored in a register, the node will compare this register field and the version value in the extension header. This comparison is shown in line 5 of listing 12. Since a value in a register field can only be used for an operation when it is read into a variable, a variable in the *user_metadata* struct is used for storing this value. This is shown in line 2 of 12. When the versions are not equal and therefore the path of the packet through the network can not be reconstructed, the Trackable field is set to value 1. This is shown in line 5 and 6 of listing 12.

Listing 12: Comparison of version values in the Logging Forwarding State implementation

```
apply {
    version.read(umd.version_value, 0);
    if (! hdr.ext.isValid()) {
        init_extension();
    } else if (hdr.ext.version != umd.version_value) {
        hdr.ext.track = 0x1;
    }
    if (smd.egress_spec == 0x0301 &&
hdr.ext.track == 0x0) {
        digest_umd.entry = hdr.ext.entry;
        digest_umd.dst = hdr.ipv6.dst;
        digest_umd.plen = hdr.ipv6.plen;
        digest_umd.timestamp =
         hdr.intrinsic_metadata.ingress_global_timestamp;
        digest<digest_umd_s>(1, digest_umd);
    }
}
```

4) Sending the path information to control plane: Since the size of the Hop-by-Hop Options extension header is fixed, a digest can be sent by the P4 program. This allows specific fields required to recreate the path of the packet being sent directly to the control plane. Additional data that can be used to collect and analyse metrics of the network can also be sent. The same information extracted by the control plane in the Hop Recording approach, is sent in this approach. To send a digest with multiple values, a struct must be defined containing those values. This struct is shown in listing 13. The destination address from the IPv6 header is also sent in the digest, because the rule that matches this address is used to forward the packet and therefore it is required to reconstruct the path. From an efficiency perspective, this address is used to recreate the path. It can also be done by using the NID of the node that sends the digest. But in that case, an externally stored value must be examined in stead of a value that already exists in the program.

Listing 13: Digest definition in the Logging Forwarding State implementation

```
struct digest_umd_s {
    bit<36>  entry;
    bit<128> dst;
    bit<16>  plen;
    bit<64>  timestamp;
}
```

The digest object is created by assigning the values from the fields in the packet, to the members of a *digest_umd_s* struct, shown in line 9-12 of listing 12. This is done in the egress match+action stage, since the digest must be sent after the last forwarding state comparison is made. Whenever the last node is using an earlier version of the global forwarding state, it can not reconstruct a packet's path. To check if a digest must be sent to the control plane, the egress port and the Trackable field in the packet are examined. When the egress port indicates a process of an upper layer in the network, the node is leaving the network and the digest must be sent. Such a port is simulated in the research by virtual port v0.1, corresponding with hexadecimal value 0x0301 in the devices. Furthermore, when the Trackable field is set to 1, no digest is being sent to avoid the collection of incorrect data. In listing 12 the check for sending the digest and is shown in the lines 8 through 13. This approach also sends a timestamp to the control plane - created in a similar way as by the Hop Recording approach - to calculate throughput.

C. Reconstruction of the path in the control plane

The reconstruction of the path by the control plane uses a version of the forwarding state that is archived in the last

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on March 11,2021 at 16:36:05 UTC from IEEE Xplore. Restrictions apply.

node of a packet's path. In this forwarding state all rules for forwarding a packet are listed. With use of the information from the digest each forwarding rule can be determined in chronological order. To determine the applied forwarding rule, all the rules for the current node are filtered by matching the NID. In case of the first node, this would be the NID that is in the entry field of the extension header of the packet. Since the packets are being forwarded based on an IPv6 destination address, the applied rules can then be determined by matching the destination address from the digest with the filtered rules. The NID of the next node is appended to the path. This algorithm is recursively called until the final node is reached in the reconstructed path.

VIII. Experiment

For our research we used a small network with two servers. Each server has two Netronome SmartNICs (Agilio® CX 2x25GbE SmartNIC) [12] identified by the numbers 20206 and 20207. Each SmartNIC has two physical interfaces identified by the values 0 and 1. Figure 8 shows our experiments' topology.

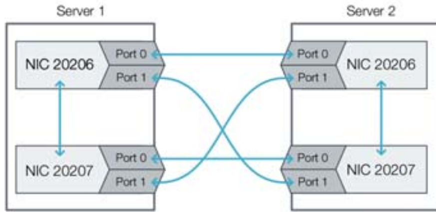

Fig. 8: Network Topology used in our experiments

Using the four Netronome SmartNICs, we created a network with four nodes and connected them as a full mesh. The connection between nodes in different servers is made with physical Ethernet links each of 25 Gbps. To connect nodes within one server, we established a virtual connection. The full mesh enables the highest number of paths between four nodes and allowed us to as many paths as possible. Our goal was to verify that our implementations allows us to track correctly all possible paths. The NID used in the experiments for each the node are the server number value prepended to the NIC number, e.g. the top node in server one has an NID of 120206.

A. *Configuration of the forwarding rules*

After compiling the P4 program, we loaded it onto each SmartNIC in the network. To be able to forward the packets to any next node, rules for the forwarding table must be created and loaded on each node together with the values of the registers that are used for each P4 implementation. The rules and registers are generated based on entries in a comma separated values (CSV) file. For each experiment we created a CSV file including entries for each node in the network. The forwarding rules are all the same for each CSV file. The rules consists of arbitrary paths, with various path lengths and network links used. This way, all path lengths and links are tested. As illustration, Table I shows two example entries in one of the CSV fileswe generated, each one corresponding to a two different node as it can be seen from two different values in the NID column.

index	NID	version	dst IPv6	egress port	next NID
1	120206	2	2000::	p0	220206
2	120207	1	2001::	p1	220206

TABLE I: Entries in a CSV file for the generation of rules for Logging Forwarding State

From this table it can, for instance, be derived that a packet with destination address 2000:: received on the node with NID 120206 will be outputted on egress port p0 which links to the node with NID 220206. Using a Python script, these values are stored in the forwarding table which the P4 program will examine. This script also sets the registers required by the relevant method. The Logging Forwarding State method requires the NID and the version field; when the Hop Recording method is used, the version field will not be included in the CSV file. Table II shows which NID corresponds with the destination IPv6 addresses used for forwarding the data.

dst IPv6 address	NID
2001::	120206
2000::	120207
2002::	220206
2003::	220207

TABLE II: The destination IPv6 addresses with corresponding NIDs

B. *Creating packets to obtain path data*

To create the packets used to experiment with the implementations, we used the Scapy library of Python. This allows for the creation of packets with headers of the user's choice. The format of the packets created to obtain data must contain an Ethernet header and an IPv6 header in order to be parsed correctly by our implementations. The transport-layer header is not restricted to be an UDP header or a TCP header, since this header will not be parsed by the implementations. We chose to use the UDP header for speed of testing since no connection is required. Hence, all the sent packets will have the same format with three headers and some payload. We chose a payload of the same length for each packet. This is necessary because the packets will get extension headers added of different lengths according to the path they take and the method used. When the payload length is the same for each packet, different path lengths and methods can be distinguished according to a packet's total length. If we would have used a different payload length for each packet, it would not be clear from its length if a packet is taking a path of four or three hops. This would make our validation unnecessarily more complex.

To test the correctness of all recorded path we sent a flow of packets into the network for each possible path. We randomised the amount of packets in one flow as a randomized

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on March 11,2021 at 16:36:05 UTC from IEEE Xplore. Restrictions apply.

integer between 5 and 100. We chose a minimum of 5 to avoid uncertainties about a path that is reconstructed only once. We chose a maximum of 100 to avoid the experiments taking a lot of time. We used this approach to create random numbers of packets for the three following experiments:

- Experiment 1: Hop Recording;
- Experiment 2: Logging Forwarding State when all nodes run the same forwarding state version; and
- Experiment 3: Logging Forwarding State when one node is running a different forwarding state version than the others.

IX. Hop Recording Results

We verified the correctness of our implementation by examining the packets that arrives at the last hop, inspecting them with Wireshark. After that we compared the number of times we recorded certain paths to the number of times we had generate packets travelling it.

Figure 9 shows a Wireshark capture of a packet as it appears in the last node of its path. With the Hop Recording approach, Wireshark will not parse all fields of cloned packets correctly, because in this method we append the timestamp to the packets at the very beginning. This required us to perform manual inspection of the values seen to verify their correctness.

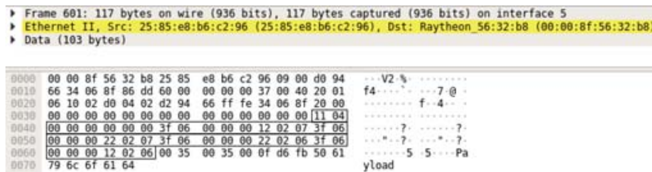

Fig. 9: Packet captured by Wireshark from experiment 1

The information Wireshark parses out of the packet is shown in the first three lines of figure 9: the first 8 bytes of the byte data is the timestamp added to the cloned packet; while the Hop-by-Hop Options extension header can be found in the outlined bytes in the figure and can be checked for its correctness. The first byte (0x11) has value 17 which indicates the UDP header type as next header. The next byte with value 4 indicates that there are four options in the extension header. The options in the extension header start after the next 6 bytes which are all zeros indicating padding. The first two bytes of the options (0x3f and 0x06) specify the Option Type and Extension Header Length as described in section V. As the figure shows all options start with the same values for these fields. The low-order 6 bytes of each option in the extension header indicate the NID of the node that forwarded the packet. The destination of this packet can be found in the 16 bytes in front of the contoured bytes in figure 9. This shows the destination address 2000:: which is node 120207 according to table II. This NID is also seen in the first option of the extension header in the packet. When all options in the extension header are examined, it can be derived that the NIDs are in reverse order of passing by the packet. However, this is the result of adding the options to the extension header without having the options parsed in the implementation.

We confirmed that the number of packets that are sent during the experiment is exactly the same as number of times the path is seen. We could confirm that all packets sent are properly captured and their path is reconstructed. W

X. Logging Forwarding State results

We followed the same verification procedure for Experiment 2 and 3, as we had done in Experiment 1.

Figure 10 shows an example packet as it appears in the last node of its path.

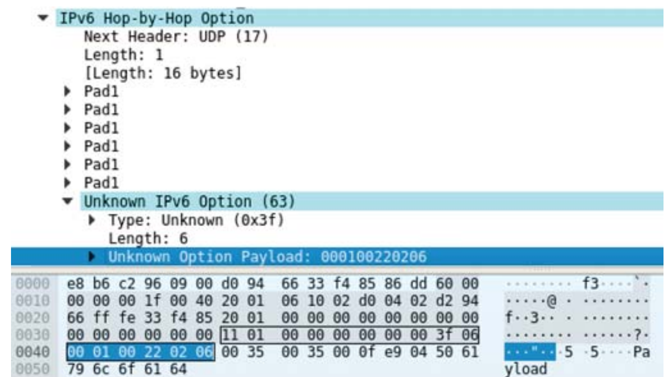

Fig. 10: Packet captured by Wireshark from experiment 2

In this case Wireshark can correctly parse these packet, and we can see that Wireshark parses the Hop-by-Hop Options extension header correctly. This can also be checked manually by examining the contoured bytes in the figure. This part of the packet is the Hop-by-Hop Options extension header. The first byte (0x11) has value 17 which indicates the UDP header type as next header. The next byte with value 1 indicates that there is one option in the extension header. The option in the extension header starts after the next 6 bytes which are all zeros indicating padding. The first two bytes of the option (0x3f and 0x06) specify the Option Type and Extension Header Length as described in section V. The selected bytes of the contoured bytes indicate the Option Data field of the Hop-by-Hop Options extension header. The first byte has value 0, which indicates that the packet's path can be tracked. This packet thus only traversed nodes using the same version of the forwarding state. The second byte has value 1, which specifies the version of the forwarding state that is used in the node with NID 220206, which is the value of the lowest-order 4 bytes in the selected bytes of the data.

Comparison of the number of reconstructed paths and the number of generates paths showed also in these two experiments that we can reconstruct all paths correctly, hence once more validating the solidity of our implementation.

XI. Discussion

A. Hop recording

As said before our Hop Recording implementation allows us to track all packets and correctly reconstruct their paths.

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on March 11,2021 at 16:36:05 UTC from IEEE Xplore. Restrictions apply.

An advantage of this method is that the control plane is only required to perform tasks in the last node, since all the information of the path is stored in the packet. Reconstructing the paths based on a clone of the packet, however, is less efficient than reconstructing them based on a digest that already includes all NIDs of the path separately. This way the fields would only have to be put in the right order to reconstruct the path.

This implementation has also downsides. First of all, a variable amount of data is added to the packet. This results in an increase in the size of the packet at each hop. In general, this makes it difficult for a sending device to know what maximum transmission unit (MTU) should be used. When the MTU is chosen too small or too big, this could result in the packet being dropped. With our use of the header stack structure a maximum of 16 NIDs can be added to the packet, and consequently the maximum size of a packet is known. However, depending on the network in which this approach is used, this could not satisfy the needs of the user. Additionally, the documentation of the Netronome SmartNICs is not clear about when 16 headers is the maximum for a header stack and in what cases it is less. The other approach implemented, parses only the part of the Hop-by-Hop Options extension header without the Options. The fact that the option values are not known when they are not parsed, and the fact that it is difficult to use variable size fields in P4, results in the complexity to send digests with the information required to reconstruct the path. To workaround this difficulty, the packet has to be cloned and this clone is sent to the control plane in the last node of a packet's path. However, sending a clone to the control plane is less efficient than sending a digest, since a digest contains less data to be processed.

A second downside of our approach is that it requires additional information about the hops taken when there exist multiple links between two nodes. This would require more data added to the packet. This would also result in more work for the control plane, since this information needs to be extracted from where it is stored in a device. Additionally, more data to be extracted from the cloned packets leads to more work for the control plane to reconstruct the paths.

The implementation can be optimized by determining the size of the NID based on the number of devices in the network. For instance, when this implementation is used in a network with 8 nodes, the values 0 trough 7 can be used to represent each NID. This would require onhly 4 bits. However, using the smaller size of the NID would require calculations in each hop to determine the size of the padding in the Hop-by-Hop Options extension header.

B. Logging Forwarding State

Also this implementation to track the path of data through a network has certain consequences. A first downside is that this method can only be used when a global forwarding state is available and when this is the only aspect regulating the routing of packets. However, even when a global forwarding state is available, some packets may not be tracked, e.g. when there are different versions of forwarding states in the nodes along the path of the packets for example at times of routing reconvergence.

However, in contrast to Hop Recording, a constant size of data is added to a packet. This eliminates some difficulties with determining the MTU and implementation design choices as discussed in section XI-A. Additionally this method allows to send only a digest with specific header fields to the control plane. The control plane does not need to extract the fields from the entire packet to reconstruct the path of packets. The only information added to the packets is the entry node, the forwarding state version, and a field indicating the trackability of the paths. Though in contrast to Hop Recording, this method requires more resources along the path of a packet, where the forwarding state database must be searched in order to find the applied forwarding rule. Additionally, the corresponding NID to the output link must be found in order to append it to the complete reconstructed path. Note that we could extend this method to support the case of a change in the forwarding state. Though this would result in packets of variable size, which as in the case of Hop Recording bring in MTU mismatches.

Finally, we must point that we chose IPv6 for our implementation as this supports both bound and unbound extentions to the header. While it is possible to use Options in IPv4 headers they have limited and fixed size, and could not be used in any case in methods such as the Hop Recording that do not limit the growth of the additional information in the packets.

XII. Visualisation

Knowledge of network paths allows for insightful visualisation at different level of granularity. We expect that network using P4 devices and adopting our implementation will initially focus on this. We therefore used the path information we had collected to explore this usage of the data.

Global view of the paths can show all the paths takes by the traffic in a certain topology, and provide handles to make routing in the network more efficient. Hot spots would be visible in such maps and finding alternative routes for certain paths would re-balance utilisation. Figure 11 is a global view of the paths we created in Experiment 2.

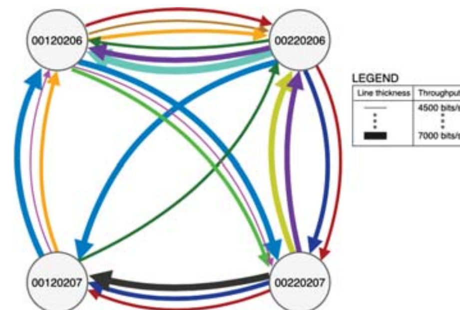

Fig. 11: Global view visualisation of data from experiment 2

Each path data takes through the network has a different color and its throughput is roughly indicated by the thickness of the lines as shown in the legend of the figure.

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on March 11,2021 at 16:36:05 UTC from IEEE Xplore. Restrictions apply.

A second topological level of visualisation focus on how traffic reaches a specific end-node. This *end-node view* can be done in two ways: one could select two nodes (x and y) to see the path between them (see 12a); or one could selecting one node to see all the paths that have it as final destination (figure 12b).

Fig. 12: End-node view visualisation in experiment 2: path from 120206 to 120207 (a) and all paths to 220206 (b)

In both figures each path has its own color and its throughput is roughly indicated by line thickness. These types of visualisations could be useful to network users who wants to know how traffic is reaching them, particular when interested in the sources of malicious traffic.

Finally, a *link view* could use our implementation to provide information on the paths that traverse a specific link, information that is useful for traffic load investigation. Figure 13 shows this for the data collected in Experiment 2.

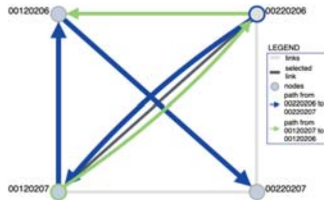

Fig. 13: Link-view visualisation of all paths that share the link between 120207 and 220206 in experiment 2

XIII. Conclusions and Future Work

In this article we presented the implementation of path recording methods in P4; our effort will benefit networks as it provided handles for increased security and for novel insights toward delivering newer services.

The first approach is Hop Recording, which each node along the path of a packet adds its own NID to the packet. The data is extracted from a clone of the entire packet in the last node in the path of a packet by the control plane. This is the only time the control plane gets involved. The second approach, Logging Forwarding State, adds the version of the forwarding state that is used by the first node along a packet's path. In each next node, this version is compared to the version used by the current device in order to check if the path can be reconstructed. In the last node of the path, the complete path is reconstructed based on the information in the packet itself and the version of the forwarding state that is used when the packet was sent. The results of the experiments with these implementations showed that all paths of all packets sent into the network that are trackable, were reconstructed correctly by the last node of the path.

Both implementations can be used to track the paths of packets through a network in hardware with P4. We in fact show that easy to implement visualisations, at different granularity level, provide the network operators and engineers with extensive insights.

There are some possible optimisation we plan to explore. Of particular interest would be to limit the amount of packets that have to be sent to the control plane in order to reconstruct their paths. This could done by storing some identifier of the data indicating a certain path when the device sees this path for the first time. Whenever a packet with the same path data is examined by this device, it can determine the path based on this identifier instead of reconstructing it in the control plane. This would make the implementations more efficient, since it decreases the amount of tasks the control plane must perform.

Acknowledgment

Part of this work was funded by the RoN - Research on Networks- from SURFnet. We are particularly thankful to Ronald van der Pol and Marijke Kaat. This research is also part of the 2StiC program - Security, Stability and Transparency in inter-network Communication - (https://www.2stic.nl/).

References

[1] Cisco Systems Inc, "Netflow performance analysis," 2005.

[2] R. Koning, N. Buraglio, C. de Laat, and P. Grosso, "Coreflow: Enriching bro security events using network traffic monitoring data," *Future Generation Computer Systems*, vol. 79, pp. 235–242, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17305952

[3] P. Bosshart, , D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[4] J. Hill, M. Aloserij, and P. Grosso, "Tracking network flows with p4," in *2018 IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)*. IEEE, 2018, pp. 23–32.

[5] F. Cugini, P. Gunning, F. Paolucci, P. Castoldi, and A. Lord, "P4 in-band telemetry (int) for latency-aware vnf in metro networks," in *Optical Fiber Communication Conference*. Optical Society of America, 2019, pp. M3Z–6.

[6] B. Niu, J. Kong, S. Tang, Y. Li, and Z. Zhu, "Visualize your ip-over-optical network in realtime: a p4-based flexible multilayer in-band network telemetry (ml-int) system," *J. Lightw. Technol., submitted*, pp. 1–9, 2019.

[7] M. Riftadi and F. Kuipers, "P4i/o: Intent-based networking with p4," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 438–443.

[8] S. Deering and R. Hinden, "Rfc 8200: Internet protocol, version 6 (ipv6) specification," 2017. [Online]. Available: https://tools.ietf.org/html/rfc8200

[9] The P4 Language Consortium, "P4-16 language specification," 2018. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.pdf

[10] P. Biondi, "Scapy documentation," 2019.

[11] O. Eggert, "Ipv6 packet creation with scapy documentation," 2012.

[12] Netronome Systems Inc., "Agilio® cx 2x25gbe smartnic," 2017. [Online]. Available: https://www.netronome.com/m/documents/PB$_{A}gilio_{C}X_{2}x25GbE.pdf$

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on March 11,2021 at 16:36:05 UTC from IEEE Xplore. Restrictions apply.