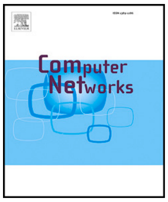
Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet



P4 Telemetry collector

Faris Alhamed [a,*], Davide Scano [b], Piero Castoldi [b], Juan Jose Vegas Olmos [c], Ilya Vershkov [c], Francesco Paolucci [d], Filippo Cugini [d]

[a] *Scuola Superiore Sant'Anna, Pisa, Italy*
[b] *Scuola Superiore Sant'Anna, Via G. Moruzzi 1, 56124 Pisa, Italy*
[c] *NVIDIA Corporation, Yokneam, Israel*
[d] *CNIT, Pisa, Italy*

ARTICLE INFO

Keywords:
SDN
P4
Switch
Telemetry
Postcard
INT

ABSTRACT

As the complexity of computer networks increases to accommodate the demand for massive connectivity and cloud services, so does the probability of fault occurrence and the surface of attacks. Hence the need for constant monitoring of network devices and accurate analysis of traffic patterns to ensure the highest performance and maximum security. This requires collecting and processing telemetry data from many sources in the network which leads to extra bandwidth usage and strains the CPU at the monitoring system resulting in scalability issues as the network grows. In this paper, we propose a two-stage postcard telemetry collector based on data plane programmability using the P4 language to address the scalability issues. We show a decrease in the CPU load of the telemetry server by over 70% while lowering the bandwidth to less than 7% in the most extreme scenario, at the cost of variable delay introduced in the collection of the postcards.

1. Introduction

The complexity of computer networks is always increasing to meet the needs for an ever-growing traffic [1], and to support the rise in demand for ultra-low-latency applications [2], these factors and many others impose the necessity for a well-planned and scalable network design not only to keep up with the current demand but also to provide solid grounds for future growth. Thus, comprehensive monitoring of network performance and traffic patterns is key to detecting misbehavior in the network or parts of it which can potentially result in an outage consequently costing the network operator a loss based on the size and type of organization and can be as high as millions of US dollars [3]. In this regard, it makes economic sense for organizations to invest in network resilience to avoid such losses [4] so it is important to gather telemetry data from devices within the network. Also, the collection of historical data and analysis of traffic patterns and trends is essential for the design of the network and the implementation of traffic engineering measures to provide fault tolerance, and to avoid overloaded links and servers via load-balancing measures [5], and to plan the evolution of the network [6]. There are different methods in the literature to collect telemetry data from the network, according to [7] those different methods can be sorted into the three following categories based on their technologies.

1. Traditional methods include Ping and Traceroute.
2. SDN methods [8].
3. Network telemetry leveraging the programmability of data plane (PDP) [9].

In this research, we will concentrate our focus on taking advantage of the recent developments in the programmability of the data plane for providing advanced collection and processing of telemetry metadata in network devices. Considering that the comprehensive collection of network data made available by PDP opens the door for many possible applications in the network and brings in Artificial Intelligence (AI) to the decision-making. Where the AI can have an important role in steering user traffic to the best serving node for lower latency and better Quality of Service (QoS) as in [10], or even provide filtering of traffic based on signatures that are generated from packet features for enhanced detection and mitigation of DDoS attacks [11].

1.1. Motivation

At the time of writing this paper, we found that Postcard-Based Telemetry (PBT) solutions, even if considered by chipset vendors, have not been widely analyzed in the literature yet, contrary to solutions that leverage embedding the telemetry data in user packets. A comparison between the two aforementioned modes of telemetry is done

[*] Corresponding author.
E-mail address: faris.alhamed@santannapisa.it (F. Alhamed).

https://doi.org/10.1016/j.comnet.2023.109727
Received 29 November 2022; Received in revised form 15 March 2023; Accepted 17 March 2023
Available online 21 March 2023

in [12] which shows that PBT has certain advantages as the user packets remain unchanged and that embedding the telemetry data in user packets has potential vulnerabilities such as eavesdropping and tampering while PBT packets tolerate extra processing for enhanced security. However, PBT suffers from a higher bandwidth overhead compared to embedding the telemetry data in user packets [12]. In both modes of telemetry and especially in PBT, the excessive generation and transport of telemetry data can consume a high percentage of links' capacity and may require significant processing power that a typical data center needs to dedicate thousands of CPU cores just for simple packet I/O operations [13]. This problem has been discussed in the literature and various solutions have been suggested. However, most of these solutions rely on the selectivity of the generation of telemetry data and follow a top-down approach as indicated in [14], while others tried to apply the selectivity within the telemetry server as in [15]. Per comparison, our solution aims to reduce the telemetry bandwidth and the subsequent processing overhead without compromising the granularity of the exported telemetry data. We aim to achieve this reduction by aggregating R payloads of the telemetry packets in a single larger packet taking advantage of the hardware acceleration offered by the programmability at the data plane, as explained in detail in the following sections of this paper.

1.2. Our early work

This paper is based on our previous work in [16] in which we presented the concept of the two-stage telemetry collector where the first stage is a P4 aggregation switch whose purpose is to aggregate the telemetry reports from various packets in one larger packet and forward it to the telemetry server. In our previous work, we performed a fixed-level aggregation at a single P4 aggregation switch and demonstrated a reduction in bandwidth and CPU load on the telemetry server. Also, we mentioned the possibility of aggregating or correlating the telemetry reports at the P4 aggregation switch by either flow-id, switch-id, or both at the same time.

1.3. Paper contribution

We can summarize the contribution of our work on this paper in the following key points:

1. We provide the details of our implementation of the two-stage telemetry collector including the algorithms we used for the different levels of aggregation at the P4 aggregation switch. Also, we explain the implementation of the telemetry server in different cases.
2. We investigate the feasibility of each aggregation level based on the hardware resources needed to perform each level of aggregation. In addition, we introduce the concepts of distributed aggregation and priority-based aggregation and
3. We provide a comprehensive set of experimental results that evaluate the aspects of the different aggregation levels in terms of CPU load at the telemetry server, network bandwidth usage, introduced overall delay, and collector intra-switch latency.

1.4. Organization of this paper

The rest of the paper is divided into sections and is organized in the following way: In Section 2 we briefly go through the main technologies and previous works relevant to the generation and collection of telemetry data in the network. In Section 3 we describe in more detail the concept of the Two-Stage Telemetry Collector. We describe our implementation and the test bed used to conduct the experiments in Section 4, while the results of our experiments are introduced and discussed in Section 5. Finally, the conclusions and a discussion on possible future improvements are presented in Section 6.

2. Background

2.1. Software defined networking (SDN)

As networks grow, they become difficult to configure and manage, and the inefficiency of traditional routing starts to become evident as routing based on network topology alone leads to unbalanced load distributions on links and servers, resulting in congestions to happen which in turn leads to transmission delays or even packet drops. Therefore, to improve path selection in the network, there is a persistent need for an all-out view of the network at a central node in charge of optimizing network operation. This led to the emergence of SDN [17] where the intelligence in the network is moved from each individual device to a centralized controller that has a full view of the network allowing it to manage the packet forwarding policies and communicate these policies to the forwarding devices using a standard Application Programming Interface (API) such as OpenFlow [18] and P4Runtime [19].

2.2. Programmable data plane (PDP)

Traditional forwarding devices had their logic hard-coded in the hardware and they could only recognize and manipulate a limited set of standardized headers, and the processing logic of the forwarding device could not be changed without making changes to the hardware itself, which is in many cases very expensive or not even possible. This characteristic of the traditional devices strictly limited the flexibility of the network and made introducing upgrades to the network a slow and expensive process. With PDP the processing logic can be dynamically enforced to support new non-standard headers and functions. This led to the creation of Programming Protocol-independent Packet Processors (P4) language [20,21] with the goal to allow programmers to change the logic of network devices after they are deployed without being restricted to the use of any specific set of legacy protocols and to make the description of packet processing functionality independent from the underlying hardware.

2.3. In-band network telemetry (INT)

In-band network telemetry is a set of tools and protocols with the goal of gathering information about network state by the data plane without needing intervention from the control plane [9]. INT provides for an extensive collection of data about individual packets as they travel through the network. The analysis of this data can give valuable information about the network state allowing to trace these individual data packets [22] to detect forwarding loops and network black holes, also to enable congestion control [23,24]. In addition, INT allows end hosts to embed instructions within the data packets called Tiny Packet Programs (TPP) [25], able to query the state of the network or even introduce changes in order to meet certain application requirements. The P4.org specifications [9] defines three modes of In-Band Network Telemetry: 1. INT-XD (eXport Data): the telemetry data is exported in a separate packet (postcard) without the modification of the original packet. 2. INT-MX (eMbed instruct(X)ions): an INT source node adds an instruction header to the packet and later nodes in the network follow the instructions to export their telemetry data. 3. INT-MD (eMbed Data): both telemetry instructions and data are inserted into the packet.

The INT-XD and INT-MX modes of telemetry are also known as *postcard-based* telemetry. In this case, the telemetry data are collected at the network nodes and are exported in separate packets toward a telemetry collector. The telemetry data to be collected and exported, referred to as *metadata*, varies according to the application and can be any network- or device-related information of interest. However, the specification in [9] defines a set of useful metadata that can be made available on many devices, including: 1. *Node ID:* to identify the source node at which the telemetry report was generated. 2. *Ingress*

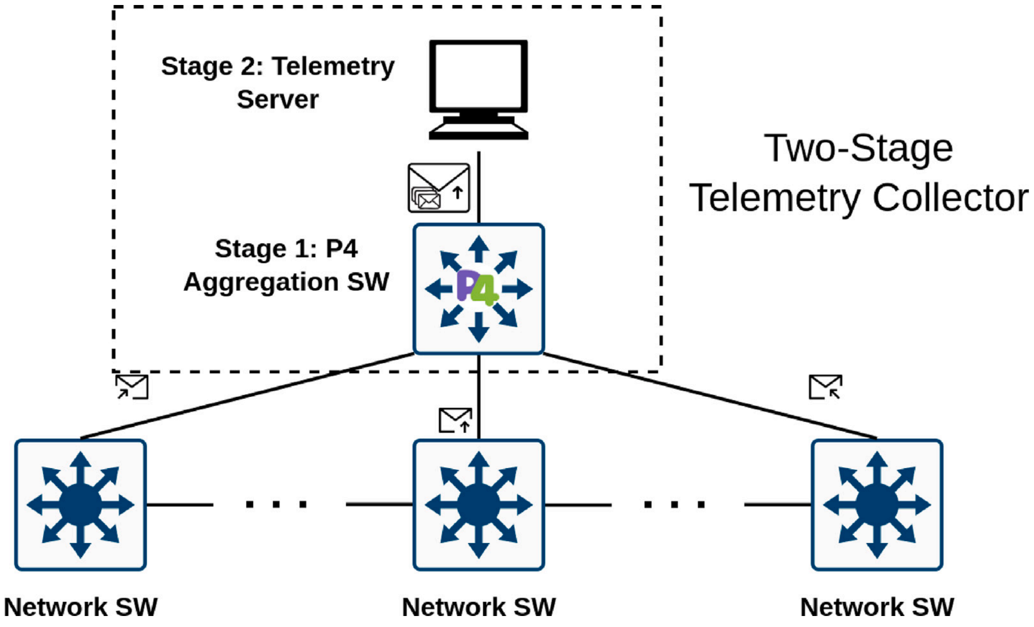

Stage 2: Telemetry Server
Two-Stage Telemetry Collector
Stage 1: P4 Aggregation SW
Network SW
Network SW
Network SW

Fig. 1. A two-stage telemetry collector in the dashed line with the first stage being a P4 aggregation switch and the second stage a telemetry server running on a general purpose computer.

Interface ID: identifies the network interface on which the packet was received. 3. *Ingress Timestamp:* the device's local time at which the packet was received on the ingress interface. 4. *Egress Interface ID:* identifies the network interface through which the packet was sent. 5. *Egress Timestamp:* the device's local time at which the packet was processed by the egress interface. 6. *Hop Latency:* the time taken by the packet to be switched by the network node. 7. *Queue Depth:* queue depth information when the packet entered and/or left the queue. The *Telemetry Report* is defined in the P4.org specifications [26] as a message generated by a network device that supports In-Band Network Telemetry for a certain packet and is sent to the telemetry collector. The report aims to set a standard for interoperability between different network devices. Moreover, it carries the metadata collected from the network device for a certain data packet. In case the INT-MD mode of operation is used, the report can additionally carry telemetry metadata from the upstream nodes when exported by the sink node.

2.4. Telemetry collection

After the telemetry metadata is generated by the network devices, it is packaged in a telemetry report and forwarded to a telemetry collector, in charge of extracting the telemetry data, correlating them, and possibly storing them in a database. In addition, such data may be consumed by the SDN controller to perform routing decisions based on the current network status. When the number of telemetry reports generated by network devices is very high (this is especially the case with postcard-based telemetry), the collector will require more resources to process data from the reports and might incur high CPU usage.

2.5. Related work on overhead reduction

To detect and diagnose problems within the network in a near real-time manner, a huge amount of telemetry data need to be generated at any given moment. This huge amount of data is mostly redundant and carries no useful information for the management system, for this purpose there exist many works in the literature to address this issue [13,15,27–29]. For example, the work in [27] employs a Postcard-Based Telemetry Marking (PBT-M) to propose a Traffic-Aware Network Telemetry (TANT) framework that manages to reduce the telemetry overhead by over 75% in exchange for reduced granularity of telemetry data. The TANT framework uses a Machine-Learning based classifier in the telemetry controller to configure the granularity of the exported telemetry data at the network nodes. In the TANT approach network nodes can assume three different roles. First, a telemetry source node marks the data packets with a special mark to indicate to the downstream nodes the granularity of the telemetry export for that type of traffic, the source node also exports its own data. Second, a telemetry transit node exports its telemetry data based on the granularity indicated by the sink. Finally, a sink node removes the mark from the data packet and forwards the packet to its next hop, in addition to exporting its own telemetry data.

Meanwhile, the work in [29] proposes the Probabilistic In-band Network Telemetry (PINT) framework employs a probabilistic sampling approach to spread the telemetry data over multiple packets to reduce the per-packet overhead.

Another telemetry framework that aims to reduce the telemetry overhead is DeltaINT [28] in which telemetry data are embedded in the data packets (INT-MD mode of INT). In DeltaINT each node compares its last exported state with the current state and only exports parameters whose change (Delta) exceeds a predefined threshold. The authors of DeltaINT compare their solution to the PINT and show that DeltaINT offers multiple improvements in various use cases such as congestion control, path tracing, and latency measurements.

Other works focused on solving the CPU overhead issues at the telemetry collectors. One example is the work in [13] that proposes a solution called Distributed Aggregation of Rich Telemetry (DART). DART works by leveraging Remote Direct Memory Access (RDMA) to write telemetry data directly in the collector's memory bypassing the CPU, it uses a hash function to generate a stateless mapping between telemetry data and memory addresses. The authors show that their implementation uses 30.0 GB for storing telemetry data coming from up to 100 million flows assuming a data length of 160-bit in addition to a 32-bit checksum for detecting overwritten data.

Another work was done in [15] where the authors proposed an INT collector made of two data processing paths, a fast path, and a normal path. The fast path runs at the kernel level to achieve a high rate of packet processing, it parses every packet to extract the telemetry report and runs the report through an event detector. When the fast path detects a networking event, the event is then sent to the normal path for processing and exporting the data to a database.

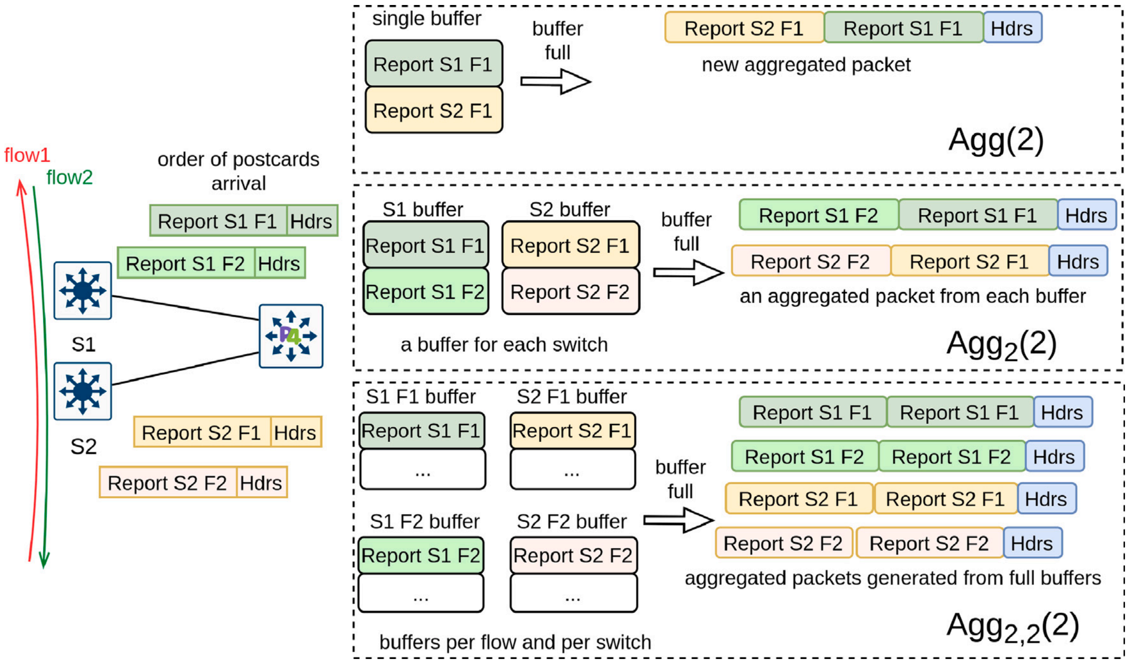


Fig. 2. An illustration of different scenarios of aggregation with $Agg(2)$ in the top box, $Agg_2(2)$ in the middle box, and $Agg_{2,2}(2)$ in the bottom box.

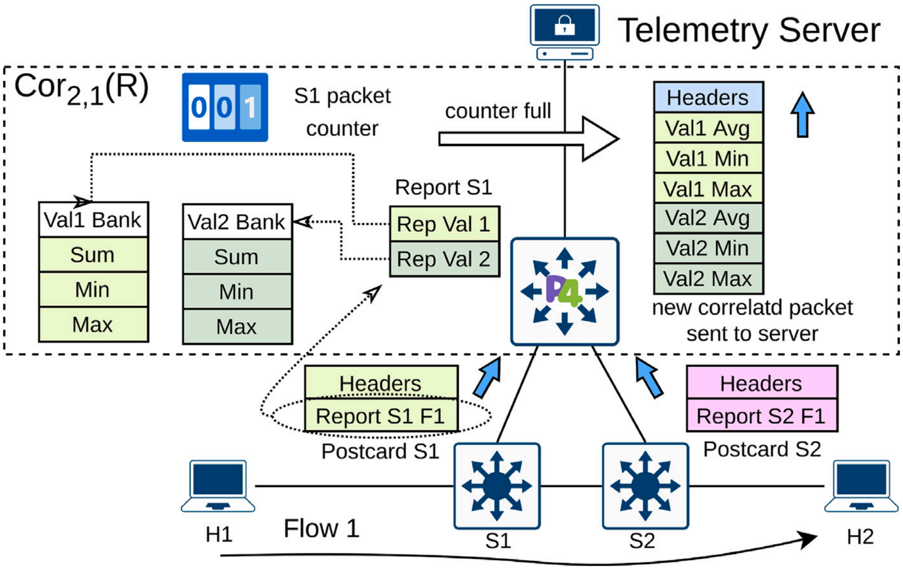


Fig. 3. A representation of the $Cor_{2,1}(R)$ case.

3. A two-stage telemetry collector

The exhaustive monitoring of flows in the network and the generation of a telemetry report for each packet at every switch can produce plenty of telemetry reports which in turn can consume a great amount of bandwidth at the network links and strain the CPU at the telemetry collector. We propose a solution to mitigate this problem by implementing a two-stage telemetry collector, as shown in Fig. 1, leveraging the data plane programmability using the P4 language. The first stage of the collector is a P4 capable switch based on the v1model architecture per P4 standards [30], whose purpose is to perform aggregation on telemetry packets coming from various sources in the network. Specifically, the proposed idea is to extract the telemetry reports from these packets and aggregate the reports in one larger packet before forwarding it to the telemetry server.

3.1. Different aggregation levels

We propose and discuss different levels of aggregation to be performed by the P4 aggregation switch in Fig. 1, where an aggregation level refers to the number of telemetry reports aggregated in one packet and if further processing is applied to these reports. The proposed aggregation levels are the following:

1. $Agg(R)$: This is the simplest level of aggregation and requires only one generic buffer. In this solution, the P4 aggregation switch extracts and aggregates R telemetry reports in one larger packet and forwards it to the telemetry server regardless of the flow that triggered the generation of the telemetry report and regardless of the switch that originated the telemetry report.
2. $Agg_N(R)$: This is a slightly more complicated solution that extracts and aggregates telemetry reports based on the network

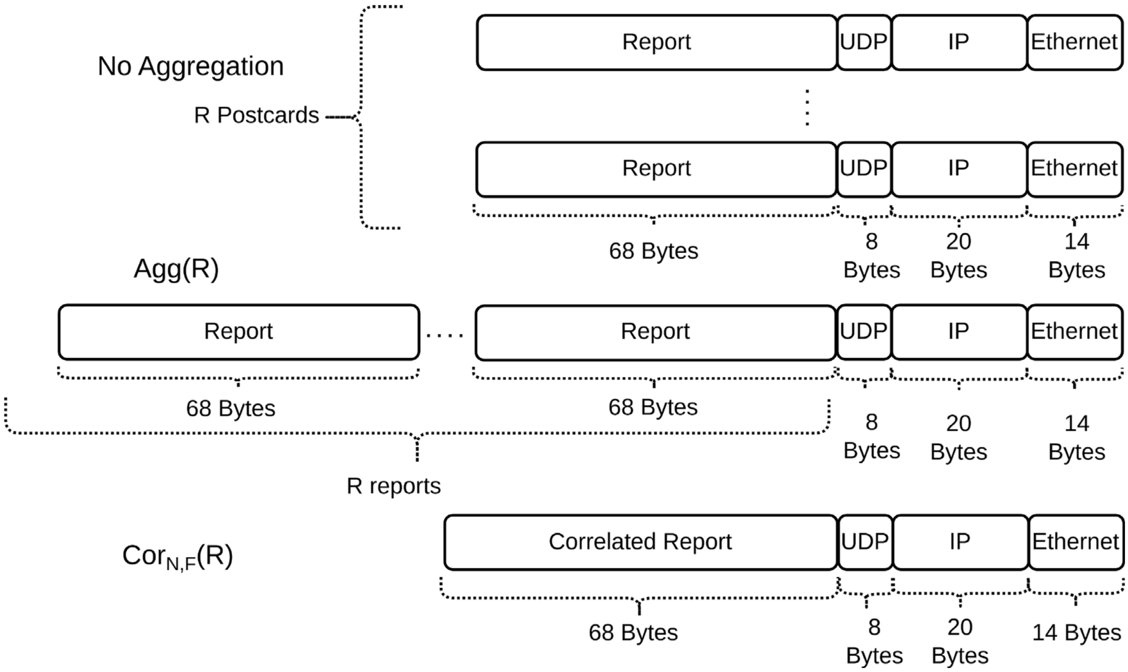


Fig. 4. Comparison between different aggregated packets.

node where they originated. This solution requires N buffers at the aggregation switch where each buffer stores the telemetry reports that originated from a distinct node.

3. $Agg_{N,F}(R)$: This is the most complicated solution in which the P4 switch has N buffer banks where N corresponds to the number of network nodes that are generating telemetry reports. Each buffer bank has F buffers where F is the number of flows that flow through that network node. This solution can greatly reduce the subsequent correlations that need to be performed by the telemetry server.

We can illustrate the different aggregation concepts using Fig. 2, in which we have two switches in the network and two monitored flows with arbitrary directions. One postcard is generated by each switch for every packet that belongs to one of the monitored flows. In the dashed rectangle on top of Fig. 2 the $Agg(2)$ scenario is depicted, where the P4 switch has only one buffer and is aggregating reports from both switches and both flows in the same buffer. Once that buffer is full then a new packet is generated with reports from both switches. The middle dashed box in Fig. 2 represents the $Agg_2(2)$ scenario, where the P4 switch has two buffers to sort reports by their originating switch. Once a buffer is full its content is emptied into a new aggregated packet and forwarded to the telemetry server. In this scenario, the aggregated packet contains only reports generated by the same switch. The bottom dashed box represents the $Agg_{2,2}(2)$ scenario, where the P4 switch has four buffers to sort the reports by flow and by originating switch. Once a buffer is full its contents are emptied into a new aggregated packet. In this case, the aggregated packet only contains reports that belong to the same flow and are generated by the same switch.

3.2. *Correlation of telemetry reports*

In some scenarios, where it is not critical to collect comprehensive information about the network or about certain low-priority flows at every moment, the P4 aggregation switch can perform extra functionality to provide only peak, minimum, and average values. This aims to further decrease the bandwidth used by the postcards and reduces the CPU cycles needed to process the collected data. For this reason, based on the three aggregation levels described in the previous section, it is possible to derive three new solutions. That is, by performing correlations on the telemetry data at the P4 aggregation switch. The purpose is to calculate maximum, minimum, and average values of the network and switch metadata, such as queue lengths, latency experienced by packets traversing a certain switch, traffic at network interfaces, and any other relevant parameter. These three solutions can be explained as follows:

1. $Cor(R)$: This aggregation level is based on $Agg(R)$ mentioned in Section 3.1. The P4 switch performs the correlations on telemetry reports from various switches and various flows. The information generated by this solution has a large granularity and offers insights about the entire network or at least the aggregation segment (aggregation segments are discussed in Section 3.4).
2. $Cor_N(R)$: This aggregation level is based on $Agg_N(R)$ mentioned in Section 3.1. The P4 aggregation switch performs the correlations on telemetry reports that are organized by the originating network node. This solution offers a better granularity than $Cor(R)$ as the correlation process offers per switch details.
3. $Cor_{N,F}(R)$: It is based on $Agg_{N,F}(R)$ and performs correlation on reports organized per-flow and per switch. This solution offers the best granularity and greatly reduces the work required by the telemetry server.

Fig. 3 shows the $Cor_{2,1}(R)$ solution with two switches and one flow. Each generated postcard contains a report with two values $Val1$ and $Val2$. When the P4 aggregation switch receives a postcard it increases the relevant counter (Fig. 3 only shows the counter and registers for switch S1 for simplicity reasons but a similar set exists for the switch S2) and then the switch extracts the report and adds the values $Val1$ and $Val2$ to their respective sum registers and then compare each value with its max and min and updates the relevant register accordingly. When the counter is full the values are extracted into a new correlated packet where the average values are calculated by dividing the sum values by the number of postcards correlated (the counter value).

We can illustrate the difference between different aggregation levels in terms of the number of packets and bandwidth usage in Fig. 4. With no aggregation, R telemetry reports are sent in R packets. When performing aggregation it is possible to remove $R-1$ set of headers

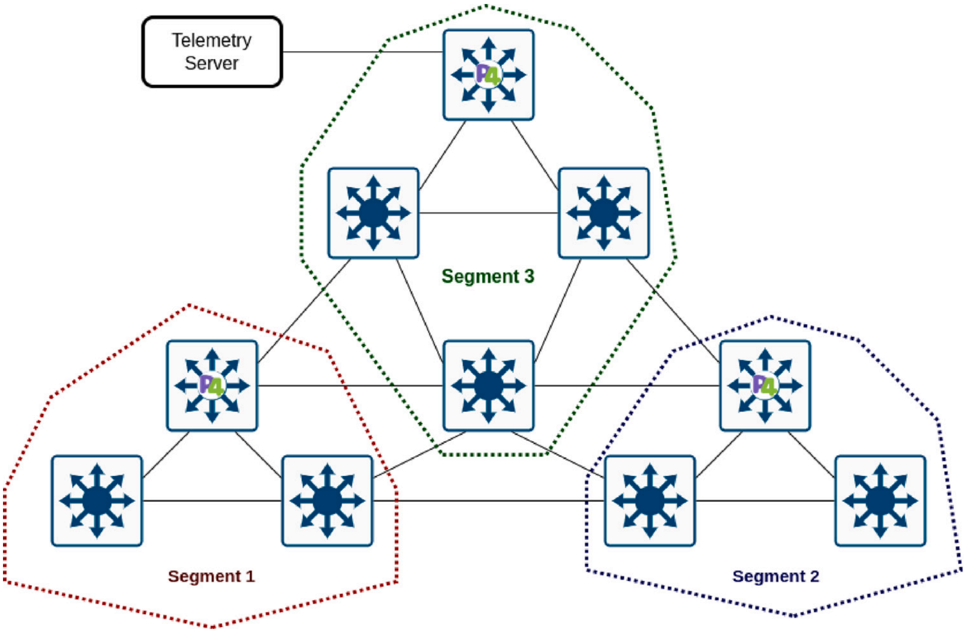


Fig. 5. The telemetry aggregation domain is segmented and each P4 switch handles the aggregation of telemetry reports from its segment.

consisting of Ethernet, IP, and UDP. This removal saves redundant bytes from being sent and reduces the number of sent packets by a factor of R.

3.3. Processing of telemetry reports

The second stage of the telemetry collector is the telemetry server which can be either software running on a general-purpose server or specialized hardware. The exact functionality of the telemetry server is open to different implementations. A generic telemetry server receives the telemetry reports, checks for non-ordinary values, stores the received data in a database and make it available to the SDN controller, and shows the data in a human-readable way.

In the case of Agg level of aggregation, the telemetry server can perform the additional functionality of calculating minimum, maximum, and average values from the data contained in the aggregated reports. This can reduce the amount of data that needs to be displayed and stored in the database while slightly affecting the view granularity of the network state.

In the case of Cor level of aggregation, the telemetry server receives the data already correlated by the P4 switch and does not need to perform any extra calculations on the received data. Thus, it can store directly in the database, significantly saving processing efforts.

3.4. Distributed aggregation of telemetry reports

When small networks with a limited amount of traffic are considered, a single P4 aggregation switch may be able to handle the process of extracting and aggregating telemetry reports from numerous packets. As the number of switches and monitored flows in the network grow, so do the memory requirements at the P4 aggregation switch for the aggregation levels that require having per-switch and per-flow dedicated registers, as discussed in Section 5.4. Additionally, not all switches in the network may offer an advanced level of programmability. For these reasons, the two-stage telemetry collector shown in Fig. 1 can be extended to a distributed model where multiple P4 switches capable of aggregating the telemetry reports are placed at specific points in the network in such a way the aggregation domain is segmented as in Fig. 5. This segmentation reduces the memory requirement for the higher aggregation levels, as the number of memory registers required at each aggregation switch is proportional to the number of network switches in its aggregation segment as discussed in Section 5.4. As an example, consider that the aggregation level $Agg_N(R)$ is applied in Fig. 5 in which the top P4 switch is dedicated for aggregation, and the two P4 switches in the middle take part in the forwarding (thus generate their own postcards). Now we can compare two scenarios, in the first scenario, the two P4 switches in the middle do not perform an aggregation, hence the top P4 switch requires 9 registers to store the telemetry reports in $Agg_N(R)$. In the second scenario, the two P4 switches perform aggregation in addition to forwarding, hence each P4 switch requires 3 registers to perform the same $Agg_N(R)$. This segmentation does not affect the CPU results obtained in Section 5.1 as the total number and size of aggregated packets that arrive at the Telemetry Server remain unchanged, while the bandwidth savings can be calculated as discussed in Section 5.2.

The aggregation segment is defined within the P4 aggregation switch using a match-action table where the aggregation switch after detecting a postcard, matches the source of the postcard (e.g. IP address) to see if it originated from a network node within its aggregation segment.

In the case of distributed aggregation of telemetry reports, each P4 switch handles the aggregation of the telemetry reports generated at its specific segment. In other words, the P4 switch can forward the traffic packets normally in the network like any other switch, and using a match-action table the P4 switch can detect that a telemetry packet is sent to the telemetry server and that the packet is generated at the switch's segment. When such a packet is detected the switch will then extract and aggregate the telemetry report contained in the aforementioned packet as in Fig. 5. This distributed aggregation of telemetry reports eases the hardware requirements of the P4 switches and saves part of the bandwidth otherwise consumed by headers of the telemetry packets headed toward the telemetry server. One important point to note is that even though the aggregation can be added as an extra functionality to any P4-capable switch, the current and future load on the P4 switch must be taken into account. For example, the additional processing at the switch pipeline introduced due to aggregation might add some latency [31]. As our implementation is done only in software, we left out this point to be investigated on a hardware implementation in the future.

The P4 language allows the programmer to write custom and flexible match-action tables to be implemented in each P4 switch, and

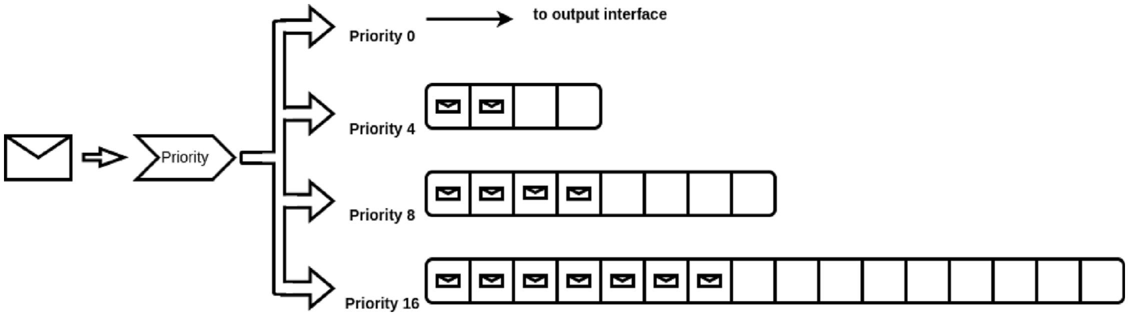


Fig. 6. Using different buffers to store telemetry reports based on their priority.

the matching of the packets can be done based on the destination IP address and the port number. This way, the P4 switch can understand that the matched packet contains a telemetry report, and by matching the source IP address the P4 switch can determine whether or not this packet was generated inside its aggregation domain. A central controller with an extensive view of the network (e.g., traffic engineering database) can program each P4 switch with the relevant match-action rules through the P4Runtime API [19].

3.5. *Priority-based aggregation*

It makes practical sense to assign different priorities to telemetry packets that belong to different flows and that are generated by different switches in the network [32]. A higher priority in this case corresponds to a lower aggregation level with the highest priority flows being forwarded without any type of aggregation. The reason is that storing the telemetry reports at the P4 switch will introduce a delay in the forwarding of the reports to the telemetry server. Depending on the application, the priority can be inferred at the P4 aggregating switch in different ways. One way to infer the priority is using a flow ID field included in the headers of the telemetry report for a per-flow priority assignment. Alternatively, the priority can be inferred using the ID of the switch that originally exported the telemetry report for a per-switch priority assignment. Another way for introducing a priority-based aggregation is to assign priorities by the switch from which the telemetry report originated. The latter case will require a dedicated field in the headers of the telemetry report for priority assignment.

3.6. *Multiple variable-size buffers*

The P4 aggregation switch has multiple buffers in its stateful stages which are used to temporarily store the telemetry reports extracted from the incoming packets. When a buffer is full its content gets encapsulated in the payload of a new packet and is forwarded to the telemetry server. The P4 aggregation switch uses one of the methods mentioned in Section 3.5 to determine the priority of the telemetry report and whether to store it in a buffer to perform the aggregation or just forward the packet as is without any further manipulation as in Fig. 6.

4. The implementation

In this section, we refer to Fig. 1 to explain our implementation of the P4 aggregation switch along with a description of the implementation of the network switches that are used to generate the postcards and the telemetry server. The workflows described in Section 3 have been mapped into P4-suitable algorithms. The implemented algorithms are described as Algorithm 1, showing the process of $Agg_N(R)$, and Algorithm 2 showing the process of $Cor_N(R)$.

4.1. *The P4 aggregation switch*

The program of the P4 aggregation switch is written using $P4_{16}$ and is compiled with the P4 compiler [33], the output of the P4 compiler is a JSON file that can be fed to the P4 switch. The architectural model of the switch is shown in Fig. 7 and it consists of a pipeline with a few processing blocks starting with a parser for dissecting and extracting the packet headers, an ingress processing block, an egress processing block, and finally a deparser that puts the headers back in the packet. The pipeline blocks can communicate a set of predefined standard metadata to pass the information on the state of the current packet, in addition to user-defined metadata.

4.1.1. *The parser*

The processing pipeline of the switch architecture starts with a parser that dissects the packet headers and treats the payload that carries the telemetry data as another header. The parser detects the existence of a telemetry report in the packet based on the destination IP address and the port number of the transport layer protocol. When a telemetry report is detected the parser of the P4 program extracts the telemetry report from the packet and adds it to the headers structure object which can be accessed in the P4 program to manipulate the headers.

Algorithm 1 $Agg_N(R)$ Algorithm

```
Define: $R, N$                                   ▷ aggregation level, number of switches
Reserve: aggregation_register( $size = [N, R \times report.length]$);
Reserve: sequence_number($size = bit < 22 >$)
for each packet do                                         ▷ for all received packets
    headers = packet.extractHeaders()
    if report in headers then                               ▷ packet is a postcard
        nodeIndex = mapNodeIdToIndex( headers.report.srcNode )
        aggregation_register.append( headers.report, position = nodeIndex )
        if aggregation_register[ position = nodeIndex ].isFull then
            headers.update( Eth, IP, UDP, Report_Group )
            headers.append( aggregation_register.extractAllReports )
            sequence_number++
        else
            dropPacket()
        end if
    else
        headers.update( Eth, IP )
    end if
    packet.attachHeaders( headers )
    forwardPacket()
end for
```

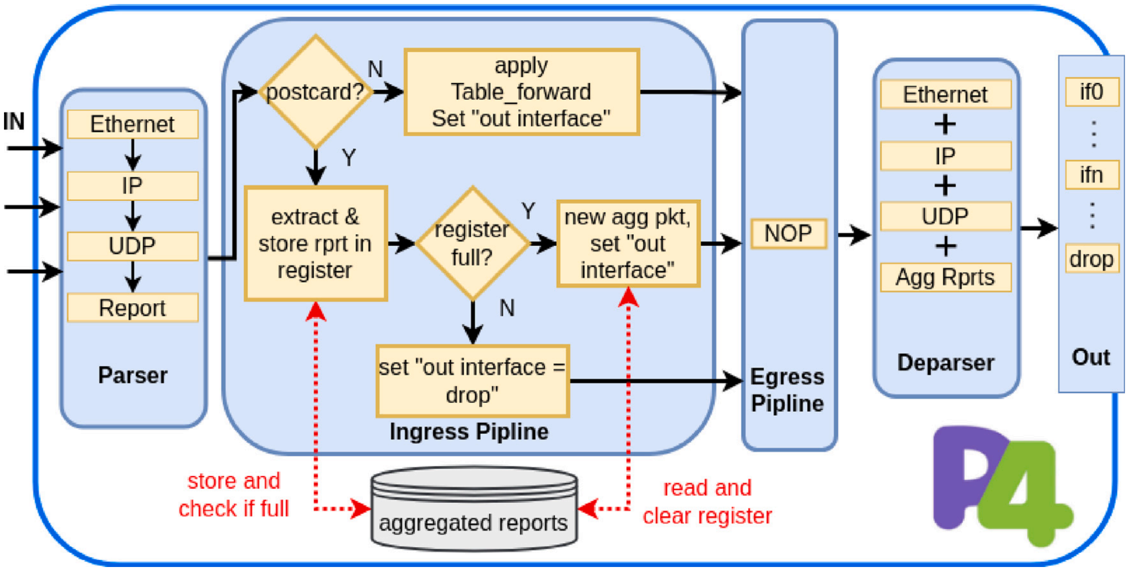


Fig. 7. The P4 architecture of the aggregation switch.

Algorithm 2 $Cor_N(R)$ Algorithm

```
Define: R, N                          ▷ aggregation level, number of switches
Reserve: N × 3 registers (min[N], max[N], sum[N]) for each correlated value
Reserve: sequence_number, postcardCount[N]
for each packet do                              ▷ for all received packets
    headers = packet.extractHeaders()
    if report in headers then                       ▷ packet is a postcard
        nodeIndex = mapSwitchIdToIndex( headers.report.srcNode )
        for correlatedValue in report do
            updateRegistersOfValue( min, max, sum,
                                    value, position = nodeIndex )
        end for
        postcardCount[ nodeIndex ]++
        if postcardCount[ nodeIndex ] == R then
            correlatedHeader = makeCorrelatedHeadersFromRegisters()
            headers.update( Eth, IP, UDP, Report_Group, correlatedHeader )
            postcardCount[ nodeIndex ] = 0
            sequence_number++
        else
            dropPacket()
        end if
    else
        headers.update(Eth, IP)
    end if
    packet.attachHeaders(headers)
    forwardPacket()
end for
```

4.1.2. The ingress

The ingress checks if the current headers include the telemetry report header: if so, and in the case of $Agg_N(R)$ aggregation, then the P4 program stores the extracted report header in a reserved register bank (the aggregation buffers) and checks if the end of the buffer is reached. If the end of the buffer is not reached then the current telemetry packet is assigned to the default drop interface to be dropped at the switch. The drop of the packet does not lead to a loss of its content as the telemetry report has already been extracted and stored in the buffer, and the related metadata will later be forwarded to the telemetry server in an aggregated packet. When a buffer is full then a new header is created from the telemetry reports included in that buffer. This header will then replace the payload of the last received telemetry packet (i.e., the telemetry packet that contains the telemetry report which filled the last position in the aggregation buffer). In addition, the headers of that packet will be modified to update the relevant fields, for example, the Time to Live (TTL), the Source IP, and the Length fields in the IP header. Finally, the egress interface will be assigned to the packet to be forwarded to the telemetry server.

The register bank can be visualized as a table with R columns and N rows (buffers), the row is determined by a match action table based on the source IP address of the node that generated the postcard. While the P4 program uses a “cursor” variable to keep track of the last column stored in the register bank.

The process for performing the correlation of the telemetry reports (i.e., a Cor level of aggregation) at the P4 aggregation switch is obtained by reserving three registers for each measurement parameter: a first register for holding the sum of the readings, a second register for holding the minimum value, and a third register for holding the maximum value. When the telemetry report is detected and extracted by the P4 aggregation switch each reading in the report is added to the first register storing the sum of the values, and then the reading is compared against the current minimum and the maximum values to evaluate if they need to be updated. When the selected aggregation level is reached, the values within those registers are extracted and a new payload is constructed that contains a correlated telemetry report. This payload then replaces the payload of the last received telemetry report packet, and the headers of that packet are updated accordingly.

4.1.3. The egress

In the P4 program related to the aggregation and correlation of the telemetry reports the egress block remains empty as no further processing is needed at this stage.

4.1.4. The deparser

The deparser block checks for valid headers. A header is valid if it has been extracted earlier at the parser or if it has been constructed during the processing of the packet in the pipeline either at the ingress or at the egress blocks. If the headers are found to be valid, then they are re-attached to the packet in the correct order and the packet is sent to the proper egress interface.

4.2. Correlation on the P4 aggregation switch

The P4 switch can perform correlations on the contents of the telemetry report to calculate and send maximum, minimum, and average values to offload this process from the telemetry server's CPU and to reduce the traffic on the links toward the telemetry server. For calculating the average value $\overline{x}$ we use the equation $\overline{x} = \frac{\sum_{p,v}}{R}$ where $\sum_{p,v}$ is the sum of the value readings v that belong the parameter p that we want to average and R is the number of value readings which equals the number of aggregated reports. In the previous equation, it is worth selecting $R = 2^y$ where y is a positive integer. When the number of value readings is a power of two, then the division process is just a matter of shifting the value to the right by a number of bits which avoids time-demanding calculations.

4.3. The telemetry server

The telemetry server is a basic implementation written in Python3 using the Scapy library. Other tools might have been used for the telemetry server, for example, a server written in C that leverages a DPDK implementation for quick extraction and parsing of the aggregated telemetry reports. However, as our experiment mainly focuses on the implementation of the P4 aggregation switch and uses bmv2 software switches [34], the performance offered by Scapy is sufficient to support our application and demonstrate the benefits of aggregation.

The code of the telemetry server uses multi-threading for handling different packets. The server listens to the network interface for incoming telemetry packets and when a telemetry packet is detected its content is then dissected and the values are written to a local InfluxDB database.

4.4. The network switch

The network switch has the functionality of forwarding the data packets to their destination based on IP addresses. In addition, they are programmed to generate postcards (telemetry packets) and send them to the server. Flow rules are installed at each switch to properly forward the packets and trigger the generation of telemetry reports based on matching the source IP address and port number of each packet passing through the switch.

5. Results

Using the testbed in Fig. 1 we evaluated the performance of our proposed telemetry server. The testbed comprises three network switches each of which supports data plane programmability through P4. The purpose of the network switches is to generate the telemetry reports and send them via telemetry packets (Postcards) to the two-stage telemetry collector. The testbed is also composed of the two-stage telemetry collector with the first stage being the P4 aggregation switch and the second stage being the telemetry server. Each of the switches we used in our testbed is a bmv2 software switch based on the v1model architecture and is running on its own general-purpose Ubuntu server (CPU AMD EPYC 7262 8-core 3.4 GHz, 16 GB RAM).

A traffic generator is used to generate traffic that traverses the network in Fig. 1 at a rate of 10,000 pps (packets per second). This traffic triggers the generation of 10,000 postcards at each network switch, thus a total of 30,000 postcards are sent to the telemetry server.

The number of generated postcards may not be representative of a real network scenario, as our experiment uses software switches running on general-purpose hardware and have performance limitations. The number has been selected to be high enough to overload the CPU of the telemetry server and allow the comparison between the various proposed solutions.

The WireShark capture of Fig. 8, recorded at the P4 collector, shows the $Agg_N(16)$ packet along with the last two postcards originated from the same switch (i.e., with IP address 1.1.1.1) subject to aggregation. Each postcard has a payload of 68 bytes where 8 bytes belong to the telemetry group header and the other 60 bytes belong to the report. Another WireShark capture is shown in Fig. 9, referred to as the $Cor_N(2)$ packet conveying the minimum, maximum, and average values of the collected intra-switch latency.

5.1. CPU load of the telemetry server

We evaluate the capability of the proposed schemes to reduce the number of packets that need to be decapsulated and sent to higher layers at the telemetry server, as the number of such packets greatly affects the telemetry server CPU load. Fig. 10(a) shows the results obtained in the testbed as a comparison between three aggregation levels. The $Agg(0)$ case is considered as a baseline, as the P4 aggregation switch just forwards postcards to the telemetry server without any modification. From $Agg(0)$ we can clearly see a CPU overload event (i.e., 100% CPU) which can be explained by the need to dissect a high number of packets received at the network interface.

The second case considers the P4 aggregation switch performing $Agg(15)$. Results show a reduction of the CPU load down to around 50% due to the reduced number of received packets, accounting for a reduction factor of around 15.

The third case is $Agg(25)$ when the number of received packets is reduced by a factor of around 25 and we observe even further reduction in the CPU load down to less than 30%.

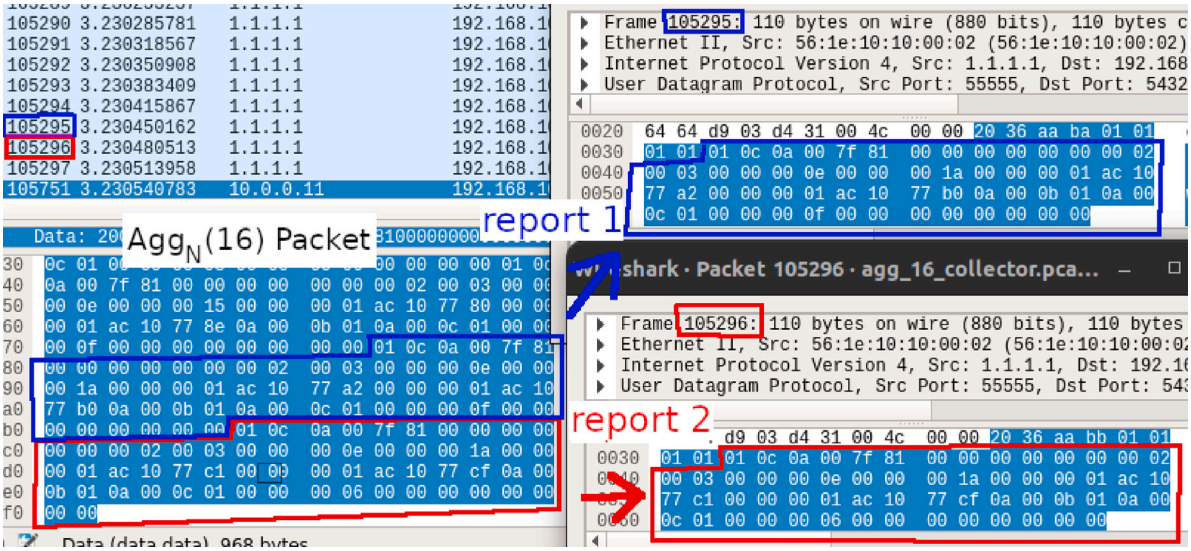
For reference, we ran a tcpdump process [35] which is a high-performance tool for reading packets from a network interface. The numbers we obtained when measuring the CPU usage of the tcpdump process were 13% when no aggregation was applied, and 8% for the $Agg(25)$ aggregation level that shows a 5% reduction in CPU usage or a 38% improvement. However, in the case of tcpdump no further dissecting or writing to the database was performed.

We can conclude that the decapsulation process of the received packets is a very expensive task in terms of CPU usage and reducing the rate of received packets by performing aggregation at the P4 switch frees a significant amount of CPU cycles that can be allocated to other useful tasks. It is worth noting that in each case there is no actual loss of any information and the benefit is obtained merely by aggregating the reports from multiple packets in one.

In the next step, we compare the impact of Agg and Cor solutions at the telemetry server. In Agg, the telemetry server is receiving an aggregated packet, extracts all the different values, and computes the statistics (i.e., average values in addition to maximum and minimum values). In Cor, the calculation of minimum, maximum and average values is done by the P4 aggregation switch before sending the correlated packet to the telemetry server.

We did not measure a significant difference in the CPU load on the telemetry server between the two cases of correlation when the packets are received by the network interface of the telemetry server due to the fact that most of the load on the CPU was caused by the decapsulation process of the received packets, and since the number of the packets is the same in both cases the CPU load was almost identical.

For this reason, we conducted a similar experiment; however, instead of reading packets from the network interface at the telemetry server, we read the packets from two packet-capture files (a .pcap file). Both files include a total of 1000 packets. The first .pcap file contains 1000 telemetry packets, each aggregating 16 telemetry reports originating from a single node. The reports are not modified and are kept as they were exported by their original node with no correlation inside the reports. In this case, the telemetry server reads the sixteen reports from each packet and calculates the minimum, maximum, and average values. At the end of each packet, the calculated values are written to a local database. The second .pcap file contains 1000 telemetry packets that originated from a single node in the network. However, each packet contains a correlated telemetry report in which

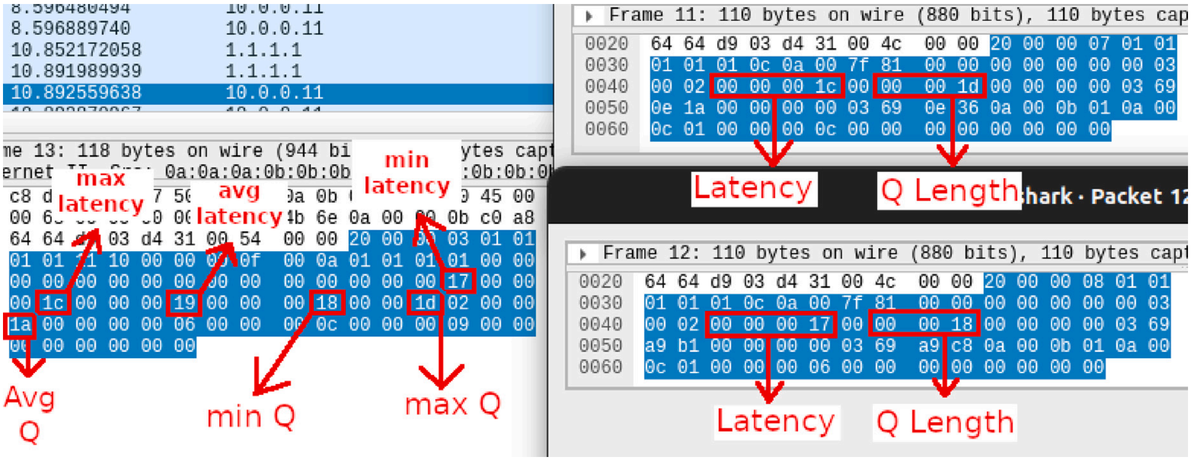


```
105290 3.230285781   1.1.1.1             192.168.1
105291 3.230318567   1.1.1.1             192.168.1
105292 3.230350908   1.1.1.1             192.168.1
105293 3.230383409   1.1.1.1             192.168.1
105294 3.230415867   1.1.1.1             192.168.1
105295 3.230450162   1.1.1.1             192.168.1
105296 3.230480513   1.1.1.1             192.168.1
105297 3.230513958   1.1.1.1             192.168.1
105751 3.230540783   10.0.0.11           192.168.1

 Data: 20                       8100000000
30  0c 01 00                        00 00 00 00 01 0c
40  0a 00 7f 81 00 00 00 00  00 00 00 02 00 03 00 00
50  00 0e 00 00 00 15 00 00  00 01 ac 10 77 80 00 00
60  00 01 ac 10 77 8e 0a 00  0b 01 0a 00 0c 01 00 00
70  00 0f 00 00 00 00 00 00  00 00 01 0c 0a 00 7f 81
80  00 00 00 00 00 00 00 02  00 03 00 00 00 0e 00 00
90  00 1a 00 00 00 01 ac 10  77 a2 00 00 00 01 ac 10
a0  77 b0 0a 00 0b 01 0a 00  0c 01 00 00 00 0f 00 00
b0  00 00 00 00 00 00 01 0c  0a 00 7f 81 00 00 00 00
c0  00 00 00 02 00 03 00 00  00 0e 00 00 00 1a 00 00
d0  00 01 ac 10 77 c1 00 00  00 01 ac 10 77 cf 0a 00
e0  0b 01 0a 00 0c 01 00 00  00 06 00 00 00 00 00 00
f0  00 00
 Data (data.data), 968 bytes
```

```
▸ Frame 105295: 110 bytes on wire (880 bits), 110 bytes c
▸ Ethernet II, Src: 56:1e:10:10:00:02 (56:1e:10:10:00:02)
▸ Internet Protocol Version 4, Src: 1.1.1.1, Dst: 192.168
▸ User Datagram Protocol, Src Port: 55555, Dst Port: 5432

0020  64 64 d9 03 d4 31 00 4c  00 00 20 36 aa ba 01 01
0030  01 01 01 0c 0a 00 7f 81  00 00 00 00 00 00 00 02
0040  00 03 00 00 00 0e 00 00  00 1a 00 00 00 01 ac 10
0050  77 a2 00 00 00 01 ac 10  77 b0 0a 00 0b 01 0a 00
0060  0c 01 00 00 00 0f 00 00  00 00 00 00 00 00
```

```
shark · Packet 105296 · agg_16_collector.pca...

▸ Frame 105296: 110 bytes on wire (880 bits), 110 bytes
▸ Ethernet II, Src: 56:1e:10:10:00:02 (56:1e:10:10:00:0
▸ Internet Protocol Version 4, Src: 1.1.1.1, Dst: 192.16
▸ User Datagram Protocol, Src Port: 55555, Dst Port: 54

       d9 03 d4 31 00 4c  00 00 20 36 aa bb 01 01
0030  01 01 01 0c 0a 00 7f 81  00 00 00 00 00 00 00 02
0040  00 03 00 00 00 0e 00 00  00 1a 00 00 00 01 ac 10
0050  77 c1 00 00 00 01 ac 10  77 cf 0a 00 0b 01 0a 00
0060  0c 01 00 00 00 06 00 00  00 00 00 00 00 00
```

Fig. 8. WireShark captures showing the payload of an $Agg_N(16)$ along with the payload of the last two received postcards.

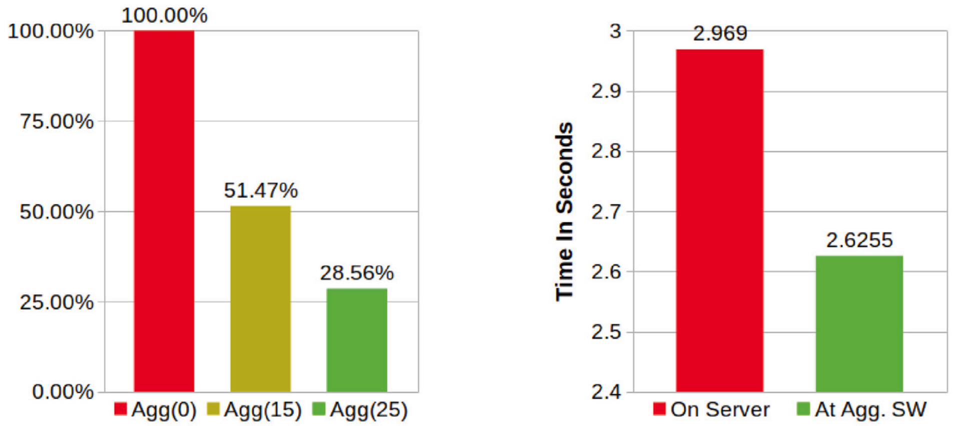


```
8.596889740        10.0.0.11
10.852172058       1.1.1.1
10.891989939       1.1.1.1
10.892559638       10.0.0.11

me 13: 118 bytes on wire (944 bi             ytes capt
ernet                0a:0a:0a:0b:0b:0b      :0b:0b:0
c8 d            7 56              0a 0b         0 45 00
00 6               0 00           4b 6e 0a 00 00 0b c0 a8
64 64 d9 03 d4 31 00 54  00 00 20 00 00 03 01 01
01 01 11 10 00 00 00 0f  00 0a 01 01 01 01 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 17 00 00
00 1c 00 00 00 19 00 00  00 18 00 00 1d 02 00 00
1a 00 00 00 00 06 00 00  00 0c 00 00 00 09 00 00
00 00 00 00 00 00
```

```
▸ Frame 11: 110 bytes on wire (880 bits), 110 bytes cap
0020  64 64 d9 03 d4 31 00 4c  00 00 20 00 00 07 01 01
0030  01 01 01 0c 0a 00 7f 81  00 00 00 00 00 00 00 03
0040  00 02 00 00 00 1c 00 00  00 1d 00 00 00 00 03 69
0050  0e 1a 00 00 00 00 03 69  0e 36 0a 00 0b 01 0a 00
0060  0c 01 00 00 00 0c 00 00  00 00 00 00 00 00
```

```
shark · Packet 12

▸ Frame 12: 110 bytes on wire (880 bits), 110 bytes capt
0020  64 64 d9 03 d4 31 00 4c  00 00 20 00 00 08 01 01
0030  01 01 01 0c 0a 00 7f 81  00 00 00 00 00 00 00 03
0040  00 02 00 00 00 17 00 00  00 18 00 00 00 00 03 69
0050  a9 b1 00 00 00 00 03 69  a9 c8 0a 00 0b 01 0a 00
0060  0c 01 00 00 00 06 00 00  00 00 00 00 00 00
```

Fig. 9. WireShark captures showing the payload of a $Cor_N(2)$ along with the payload of the two correlated postcards.



(a)

(b)

Fig. 10. (a) Comparing the load on the CPU for different aggregation levels, and (b) time in seconds taken by the telemetry server to process 1000 packets from a .pcap file.

only the minimum, maximum, and average values for the 16 telemetry reports that otherwise should be carried in the packet's payload. We show the impact carried by *Cor* levels of aggregation by feeding .pcap files to the telemetry server and thus skipping the reading of the packets from the network interface. The result is a reduction of the processing time of 1000 telemetry packets taken by the telemetry server from an

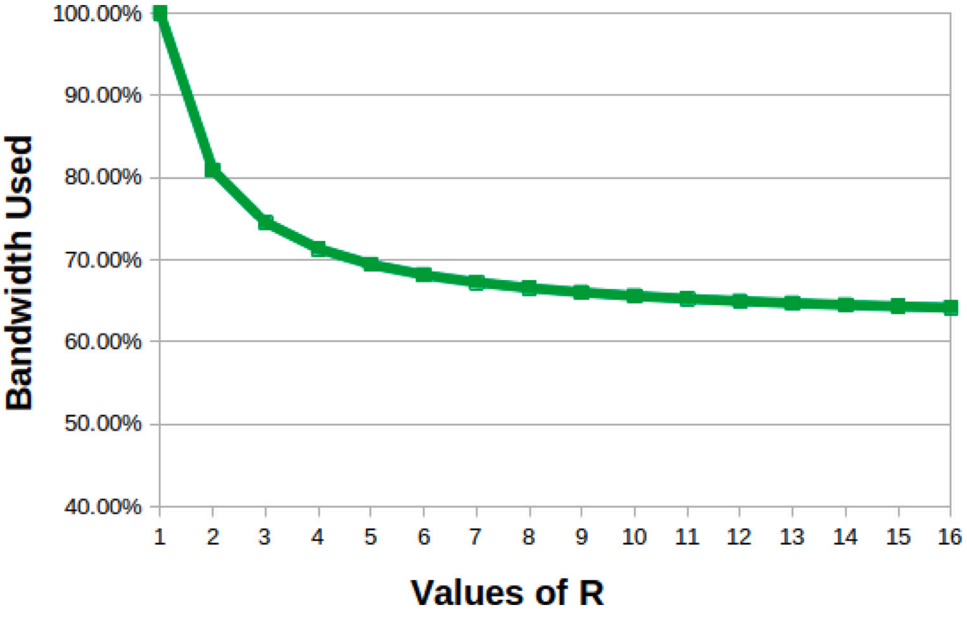


Fig. 11. Agg(R): normalized bandwidth consumption for different values of R.

average time of 2.969 s down to an average time of 2.6255 s, reaching about 11.57% reduction of the processing time as shown in Fig. 10(b).

5.2. Bandwidth usage

The aggregation of report packets leads to the removal of packet headers encapsulating individual reports, which in turn leads to a reduction of the bandwidth. We can derive a simple formula for calculating such reduction. Let H_O be the length of the headers encapsulating the payload which contains the telemetry report, this can be assumed as a stack of an Ethernet header followed by an IP and UDP headers. Assuming minimum lengths, $H_O = 14 + 20 + 8 = 42$ bytes. A single telemetry report, according to [26], has a group header with length $H_G = 8$ bytes (used to identify the source node and the hardware that generated the report) and an individual report header with length $H_R = 60$ bytes for a report carrying the entire set of P4 metadata [26]. Thus, the total postcard length is 110 bytes. In the case of $Agg(R)$, the total length of an aggregated packet will be: $H_O + R \times (H_G + H_R)$ where R is the level of aggregation, instead of $R \times (H_O + H_G + H_R)$ in case of no aggregation. Using the above numbers, the normalized bandwidth can be derived as a function of the aggregation level: $BW_n = (42 + R \times 68)/(R \times 110)$. The plot of the normalized bandwidth consumption for different values of R for an $Agg(R)$ level of aggregation is shown in Fig. 11. Fig. 12(a) shows the normalized usage of bandwidth for different $Agg(R)$ levels of aggregation, and we can see the total bandwidth used by the telemetry packets is reduced to 66% for $Agg(8)$ while the bandwidth usage is down to only 63% of its original amount when using $Agg(25)$ level of aggregation.

When performing the correlation at the P4 aggregation switch, it is possible to achieve higher bandwidth savings due to the fact that most of the data inside the telemetry reports are discarded. For example, neglecting timestamps and interfaces and sending only minimum, maximum, and average values, only $H_R + H_G = 68$ bytes are necessary for a total telemetry packet length of 110 bytes. We can see the benefits of correlation in terms of bandwidth savings in Fig. 12(b), which shows the usage of the normalized bandwidth in cases of $Cor(8)$ with a reduction to only $110/(8 \times 110) = 12.5\%$ of the original bandwidth consumed by telemetry packets, and $Cor(16)$ which can achieve a further reduction to less than 7% of the original bandwidth used.

Table 1
Different delays in milliseconds at the P4 aggregation switch for different arrival rates.

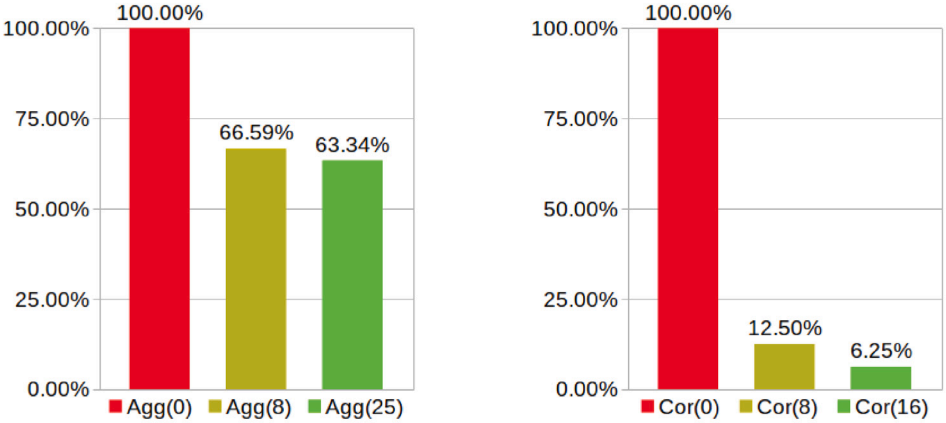
	Agg. level			
Arr. rate (pps)	Agg(4)	Agg(8)	Agg(16)	Agg(25)
2000	2	4	8	12.5
5000	0.8	1.6	3.2	5
10000	0.4	0.8	1.6	2.5

5.3. Delay

Besides CPU load and bandwidth occupancy, each proposed solution has a different impact in terms of the delay of the whole telemetry process. The delay includes the time needed to wait for the reception and the aggregation of subsequent telemetry packets, and the time needed at the aggregation switch to process the single postcards reports and generate the aggregated version.

The maximum queuing delay (i.e., the time elapsing between the arrival of the first and last postcard in the queue) can be estimated using the formula $D = R/A$ where A is the arrival rate of the telemetry packets measured in packets per unit time and R is the aggregation level, i.e. the number of reports that need to be stored in memory before building the aggregated packet to the telemetry server. R can be set in terms of priority where lower values of R correspond to higher priorities, and $R = 0$ corresponding to $D = 0$ is the highest possible priority with no extra delay introduced in the arrival of telemetry reports. We calculate in Table 1 the queuing delay in milliseconds introduced by the aggregation process at the P4 aggregation switch for different aggregation levels and different arrival rates.

Another source of delay referred to as *aggregation delay*, is introduced in the P4 aggregation switch due to the processing of the aggregated packet, including reading the aggregated packet from memory and the deparsing of the headers. The aggregation delay is measured as the time elapsing between the arrival of the last postcard in the queue and the transmission of the aggregated packet out of the output interface. We measured the aggregation delay inside the P4 aggregation switch (i.e., a BMv2 software switch) and we observed a similar performance between different aggregation levels for a certain postcard arrival rate, around 60 μs. This result is interesting and confirms that the P4 aggregation switch implementation introduces a fixed delay regardless of the selected aggregation level. The maximum delay in the



(a)

(b)

Fig. 12. (a) Normalized bandwidth usage for different $Agg(R)$ levels of aggregation. (b) Normalized bandwidth usage for different $Cor(R)$ levels of aggregation.

arrival of a telemetry report will be equal to the sum of queuing delay plus the aggregation delay.

5.4. Scalability

We evaluate the resources needed by the P4 aggregation switch in order to perform the various aggregation levels. For all levels of aggregation, the P4 aggregation switch must detect the existence of a telemetry report via the parser and by matching the flow through the destination IP address and UDP port number. Thus, a new state is needed at the parser state machine, and the amount of resources needed for every aggregation level is quantified as follows:

1. $Agg(R)$: for the aggregation of the telemetry reports the P4 aggregation switch needs a register bank of capacity $R{\times}L_r$ where L_r is the length of the telemetry report and R is the number of aggregated reports. An addition register of size L_p is required for storing the variable that points to the last location of the register bank in which a telemetry report was written. In this case, the resources needed at the P4 aggregation switch are independent of the number of switches and flows in the network.
2. $Agg_N(R)$: for this level of aggregation the switch requires a register bank of the capacity of $N{\times}R{\times}L_r$ where N is the number of monitored switches as well as a match-action table with N entries for setting a switch ID to identify the register to which the telemetry report must be written. In addition, a register bank of size $N \times L_p$ is required to store the different variables needed to point to the locations in the register banks of each switch to which it was written last. In this aggregation level the hardware requirements scale with the number of monitored switches in the network.
3. $Agg_{N,F}(R)$: in this aggregation level a register bank of the capacity of $F \times N \times R \times L_r$ is needed where F is the number of monitored flows assuming the same number of monitored flows at every switch. In addition to the previous register bank, another bank of registers of size $F \times N \times L_p$ is needed to keep track of various pointers that belong to different switches and different flows. Also, two match-action tables are needed for the setting of switch ID and flow ID for the identification within the P4 program of the proper registers within the P4 aggregation switch. In this solution the hardware requirements scale with the product between the number monitored of flows and the number of monitored switches.
4. $Cor(R)$: for this aggregation level in our implementation three registers of adequate size (variable per the monitored parameter) were needed to store the maximum, minimum, and the sum of values of every monitored parameter the average values are then calculated by dividing the sum by the number of aggregated reports. In addition, a single register is needed to hold the value of a counter variable for detecting when the correlation level is reached. The hardware requirements are constant regardless of the network size and the number of flows.
5. $Cor_N(R)$: for this solution the number of registers needed is the same number as $Cor(R)$ multiplied by the number of monitored switches in the network N. In addition to requiring a match-action table with N entries for setting identifying the correct registers in the P4 program. The hardware requirements for this solution scale with the number of monitored switches in the network.
6. $Cor_{N,F}(R)$: for this solution the number of the required register is the same as the number needed by $Cor_N(R)$ multiplied by the number of flows F, assuming the same number of flows at every switch. In addition, a second match-action table is required for adding the flow ID and correctly identifying the register to which the values must be written. The hardware requirements of this solution scale with the product between the number of switches times and the number of flows. This solution may not realistic in the majority of cases. However, a network manager might still choose to implement this solution for a selected configurable number of flows to collect only a summary of the telemetry data.

Among the different solutions that we have shown in this paper, we conclude that the $Agg(R)$ solution is the easiest to implement in the P4 aggregation switch as the needed hardware resources do not change with the number of the switches and the number of flows in the network and requires a static number of registers; in addition, the $Agg(R)$ solution does not compromise any of the information included in the telemetry reports. Meanwhile, the $Cor(R)$ solution blurs all the details about the network state and only offers a general insight which may not be very useful in most cases.

The $Agg_N(R)$ and solution could also be feasible for a low and known number of switches in the network. The same thing can be said about the $Cor_N(R)$ solution in case the network operator can tolerate the loss of some info due to the correlation process.

On the other hand, the $Agg_{N,F}(R)$ along with $Cor_{N,F}(R)$ solutions may not be practical as the costs of such implementations may overwhelmingly outweigh any benefits.

6. Conclusions

In this paper, we proposed pre-processing postcard telemetry messages at a dedicated P4 capable switch forming the first stage of a two-stage telemetry collector. We demonstrated the potential CPU load reduction at the telemetry server and bandwidth savings in the network at the cost of an extra, limited, introduced delay due to the postcards queuing behavior. The different levels of aggregation, along with the concept of distributed telemetry aggregation allows to adapt the application to the hardware available at the P4 switch and makes the application suitable for various hardware capabilities. We proposed a complete P4 design for the aggregation switch and evaluated a proof of concept in an SDN network testbed using the P4 reference software switch. We introduced the $Agg(R)$ solution which offers the greatest scalability independently from the number of switches and flows in the network and we demonstrated that it is possible — using the $Agg(25)$ solution to reduce the load of the CPU by around 70% and the bandwidth consumed by telemetry packets by almost 35% at the cost of a maximum added delay of around 72.5 μs for an arrival rate of 2000 pps. By performing correlations in the P4 aggregation switch we showed that it is possible to reduce processing time in the CPU by around 11% and reduce the bandwidth consumption by up to over 93% at the cost of losing some of the information contained in the telemetry reports. The next research steps will investigate implementations using commercially available P4 switches in the market and explore the true feasibility and limitations of our application.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request

Acknowledgments

This work has been funded by the European Commission Horizon Europe SNS JU DESIRE6G project, under grant agreement No. 101096466, and by the KDT JU ID2PPAC Project, under grant agreement No. 101007254.

References

[1] R. Tkach, Network traffic and system capacity: Scaling for the future, in: 36th European Conference and Exhibition on Optical Communication, 2010, pp. 1–22.

[2] A. Nasrallah, A. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, H. ElBakoury, Ultra-low latency (ULL) networks: The IEEE TSN and IETF DetNet standards and related 5G ULL research, IEEE Commun. Surv. Tutor. 21 (2019) 88–145.

[3] S. Wang, U. Franke, Enterprise IT service downtime cost and risk transfer in a supply chain, Oper. Manag. Res. 13 (6) (2020) 94–108, http://dx.doi.org/10.1007/s12063-020-00148-x.

[4] P. Smith, D. Hutchison, J. Sterbenz, M. Schöller, A. Fessi, M. Karaliopoulos, C. Lac, B. Plattner, Network resilience: A systematic approach, IEEE Commun. Mag. 49 (2011) 88–97.

[5] E. Jafarnejad Ghomi, A. Masoud Rahmani, N. Nasih Qader, Load-balancing algorithms in cloud computing: A survey, J. Netw. Comput. Appl. 88 (2017) 50–71, https://www.sciencedirect.com/science/article/pii/S1084804517301480.

[6] A. D'Alconzo, I. Drago, A. Morichetta, M. Mellia, P. Casas, A survey on big data for network traffic monitoring and analysis, IEEE Trans. Netw. Serv. Manag. 16 (2019) 800–813.

[7] L. Tan, W. Su, W. Zhang, J. Lv, Z. Zhang, J. Miao, X. Liu, N. Li, In-band network telemetry: A survey, Comput. Netw. 186 (2021) 107763, https://www.sciencedirect.com/science/article/pii/S1389128620313396.

[8] H. Zhang, Z. Cai, Q. Liu, Q. Xiao, Y. Li, C. Cheang, A survey on security-aware measurement in SDN, Secur. Commun. Netw. 2018 (4) (2018) 2459154, http://dx.doi.org/10.1155/2018/2459154.

[9] P4.org P4 in-band telemetry specification v2.1, 2020, https://p4.org/p4-spec/docs/INT_v2_1.pdf. (Accessed 16 August 2022).

[10] D. Scano, F. Paolucci, K. Kondepu, A. Sgambelluri, L. Valcarenghi, F. Cugini, Extending P4 in-band telemetry to user equipment for latency- and localization-aware autonomous networking with AI forecasting, J. Opt. Commun. Netw. 13 (2021) D103–D114.

[11] M. Dimolianis, A. Pavlidis, V. Maglaris, Signature-based traffic classification and mitigation for DDoS attacks using programmable network data planes, IEEE Access.

[12] E.F. Kfoury, J. Crichigno, E. Bou-Harb, An exhaustive survey on P4 programmable data plane switches: Taxonomy, applications, challenges, and future trends, IEEE Access 9 (2021) 87094–87155, http://dx.doi.org/10.1109/ACCESS.2021.3086704.

[13] Jonatan Langlet, Ran Ben-Basat, Sivaramakrishnan Ramanathan, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, Gianni Antichi, Zero-CPU collection with direct telemetry access, in: Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks, HotNets '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 108–115, http://dx.doi.org/10.1145/3484266.3487366.

[14] Minlan Yu, Network telemetry: Towards a top-down approach, SIGCOMM Comput. Commun. Rev. 49 (1) (2019) 11–17, http://dx.doi.org/10.1145/3314212.3314215.

[15] J. Hyun, N.Van Tu, J. Yoo, J. Hong, Real-time and fine-grained network monitoring using in-band network telemetry, Int. J. Netw. Manage. 29 (2019) e2080, https://onlinelibrary.wiley.com/doi/abs/10.1002/nem.2080.

[16] F. Alhamed, D. Scano, P. Castoldi, F. Paolucci, F. Cugini, I. Verschkov, J. Vegas Olmos, P4 postcard telemetry collector in packet-optical networks, in: 2022 International Conference on Optical Network Design and Modeling, ONDM, 2022, pp. 1–3.

[17] N. Feamster, J. Rexford, E. Zegura, The road to SDN: An intellectual history of programmable networks, Queue 11 (12) (2013) 20–40, http://dx.doi.org/10.1145/2559899.2560327.

[18] N. Feamster, J. Rexford, E. Zegura, The road to SDN: An intellectual history of programmable networks, Queue 11 (12) (2013) 20–40, http://dx.doi.org/10.1145/2559899.2560327.

[19] P4.org P4$_{16}$ P4runtime specifications v.1.3.0, 2020, https://p4.org/p4-spec/p4runtime/v1.3.0/P4Runtime-Spec.pdf. (Accessed 20 August 2022).

[20] P4.org P4$_{16}$ language specification v1.2.3, 2022, https://p4.org/wp-content/uploads/2022/07/P4-16-spec.pdf. (Accessed 10 August 2022).

[21] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker, P4: Programming protocol-independent packet processors, SIGCOMM Comput. Commun. Rev. 44 (7) (2014) 87–95, http://dx.doi.org/10.1145/2656877.2656890.

[22] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, N. McKeown, I know what your packet did last hop: Using packet histories to troubleshoot networks, in: 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 14, 2014, pp. 71–85, https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/handigol.

[23] N. Katta, M. Hira, C. Kim, A. Sivaraman, J. Rexford, HULA: Scalable load balancing using programmable data planes, in: Proceedings of the Symposium on SDN Research, 2016, pp. 3–14, http://dx.doi.org/10.1145/2890955.2890968.

[24] N. Katta, A. Ghag, M. Hira, I. Keslassy, A. Bergman, C. Kim, J. Rexford, Clove: Congestion-aware load balancing at the virtual edge, in: Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies, 2017, pp. 323–335, http://dx.doi.org/10.1145/3143361.3143401.

[25] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, D. Mazières, Millions of little minions: Using packets for low latency network programming and visibility, in: Proceedings of the 2014 ACM Conference on SIGCOMM, 2014, pp. 3–14, http://dx.doi.org/10.1145/2619239.2626292.

[26] P4.org P4 telemetry report format v2.0, 2020, https://p4.org/p4-spec/docs/telemetry_report_v2_0.pdf. (Accessed 20 August 2022).

[27] Cesar Gomez, Abdallah Shami, Xianbing Wang, Efficient network telemetry based on traffic awareness, 2021, http://dx.doi.org/10.36227/techrxiv.15057981.v1, TechRxiv. Preprint.

[28] S. Sheng, Q. Huang, P.P.C. Lee, DeltaINT: Toward general in-band network telemetry with extremely low bandwidth overhead, in: 2021 IEEE 29th International Conference on Network Protocols, ICNP, Dallas, TX, USA, 2021, pp. 1–11, http://dx.doi.org/10.1109/ICNP52444.2021.9651963.

[29] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minlan Yu, Michael Mitzenmacher, PINT: Probabilistic in-band network telemetry, in: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 662–680, http://dx.doi.org/10.1145/3387514.3405894.

[30] P4.org P4 behavioral model targets, 2022, https://github.com/p4lang/behavioral-model/tree/main/targets. (Accessed 18 August 2022).

[31] H. Harkous, M. Jarschel, M. He, R. Pries, W. Kellerer, P8: P4 with predictable packet processing performance, IEEE Trans. Netw. Serv. Manag. 18 (2021) 2846–2859.

[32] P. Gupta, N. McKeown, Algorithms for packet classification, IEEE Netw. 15 (2001) 24–32.

[33] P4.org P4 compiler, 2022, https://github.com/p4lang/p4c. (Accessed 03 August 2022).

[34] P4.org behavioral model v2, https://github.com/p4lang/behavioral-model.

[35] 9 (2021) 113061–113076, https://www.tcpdump.org/.

Faris Alhamed received his B.S. in Telecommunication and Electronics Engineering in 2017 from Tishreen University, Latakia, Syria. And in 2021 he received a joint M.S. degree form both Aston University, Birmingham, UK, and Scuola Superiore Sant'Anna, Pisa, Italy, in Photonic Integrated Circuits, Sensors and NETworks — PIXNET. He is a currently a Ph.D. student at Scuola Superiora Sant'Anna expected to graduate by the end of 2024. His research focuses on data-plane programmability and its applications in switches and SmartNICs.

Davide Scano received his B.S. in telecommunication engineering from the University of Pisa (2017) and his M.S in computer science and networking from the University of Pisa and Scuola Superiore Sant'Anna (2019), discussing a research thesis on software defined networking for guarantee QoS in network slicing. In 2020 he got a Research Scholarship at Scuola Superiore Sant'Anna, Pisa. Currently, he is Ph.D. student at Scuola Superiore Sant'Anna. His research interests are software defined networking, network slicing, optical networks.

Piero Castoldi is a Full Professor and leader of the "Networks and Services" research area at the TeCIP Institute of Scuola Superiore Sant'Anna, Pisa, Italy. He has been involved with various responsibilities in several national and EU FP7 and H2020 projects and he has managed several corporate-sponsored projects with the italian Railway Infrastructure Company, Ericsson, Telecom Italia. He is the Director of the Erasmus Mundus Master on Photonic Integrated Circuits, Sensors and Networks (PIXNET). His most recent research interests lie in the areas of optical network architectures, interconnection networks for Data Centers, networks for industrial applications, 5G networking. He is author of more than 400 technical papers published in international journals and international conference proceedings. He has also filed more than 20 patents and he has authored a book on multiuser detection for CDMA mobile terminals published by Artech House.

Juan Jose Vegas Olmos received the B.Sc. and the M.Sc. in Telecommunications and Electronic Engineering, respectively, in 2001 and 2003. He obtained the Ph.D. degree from the Eindhoven University of Technology, The Netherlands, in 2006. He also holds a MA in East Asian Studies, a BA in Business Administration and an MBA. He was a Research Fellow at Osaka University, Japan, from 2006 to 2008, and a Research Associate at the Central Research Laboratory, Hitachi Ltd, until 2011. He is now the Research Program Coordinator for NVIDIA Corporation in the area of Software Architecture.

Ilya Vershkov graduated from the Technion Israel Institute of Technology.

He has held diverse leadership positions at Seabridge and Nokia, until joining Mellanox Technologies, currently NVIDIA Corporation, in 2021.

Since then, he has been driving R&D efforts in the area of switches and advanced telemetry systems. He is Senior Architect in the area of Software Architecture.

Francesco Paolucci Received the Laurea degree in telecommunications engineering from the University of Pisa in 2002, and the Ph.D. degree from Scuola Superiore Sant'Anna, Pisa, in 2009. In 2008 he was granted a research Merit Scholarship at the Istitut National de le Recherche Scientifique (INRS), Montreal, Quebec, Canada. Currently, he is Senior Researcher at CNIT, Pisa Italy. His main research interests are in the field of network control plane, orchestration for edge/cloud platforms, traffic engineering, network disaggregation, advanced network telemetry, SDN/P4 data plane programmability. He has been involved in many European research projects on next generation control networking (E-Photon/ONe+, BONE, NOBEL, STRONGEST, IDEALIST, PACE, 5GEx, 5GTRANSFORMER, METROHAUL, 5Growth, BRAINE). He is co-author of 2 IETF Internet Drafts, more than 180 publications in international journals, conference proceedings and book chapters, and filed 4 international patents. He is Executive Editor of the Transactions on Emerging Telecommunications Technologies (ETT) and Associate Editor of the IEEE/OSA Journal of Optical Communications and Networking (JOCN).

Filippo Cugini is Head of Research Sector at CNIT, Pisa, Italy. His main research interests include theoretical and experimental studies in the field of communications and networking.

He serves as Coordinator of the ECSEL BRAINE Project, an EU-funded project aiming at boosting Artificial Intelligence at the network Edge.

He is co-author of 14 patents and more than 250 international publications.