university of groningen / faculty of science and engineering / bernoulli institute

Bachelor Thesis

Stateful data analytics over programming models of networks

Author: Gheorghe Pojoga (s3717003)

1st supervisor: Prof. Dr. Boris Koldehofe
2nd supervisor: Bochra Boughzala

August 31, 2021

Abstract

The optimization of network data processing is a scalable approach for meeting the continuously growing Quality of Service requirements. A solution in this regard is offloading some end-host operators into the network devices, also known as in-network computing. This approach is supported by multiple emerging technologies, that aim at providing more flexibility in terms of network programmability, while preserving the line rate. Although, the stateless packet processing has already found multiple applications across the industry, the implementation of stateful operators has proven to be more challenging due to the limitations of the available hardware. In this paper we will focus on building an understanding on the viability and acceleration capabilities of executing stateful data analytics operators inside the network, by analyzing several concrete problems and their solutions across some of the available network processing frameworks.

Contents

1 Introduction **1**

2 Background **3**

2.1 Data plane programming 3

2.2 Data Plane Development Kit 4

2.3 P4 5

2.4 Network emulation 8

3 Design Considerations **9**

3.1 Mininet constraints 9

3.2 DPDK constraints 10

3.3 P4 constraints 10

4 Simple Moving Average **12**

4.1 Operator Specification 12

4.2 Algorithm 14

4.3 P4 implementation 16

4.4 DPDK implementation 18

4.5 Discussion 19

5 Median **21**

5.1 Operator Specification 21

5.2 Algorithm 24

5.3 P4 implementation 29

5.4 DPDK implementation 30

5.5 Discussion 31

6 Analysis **33**

7 Conclusion **40**

8 Future Work **41**

9 Acknowledgements **43**

Bibliography **44**

1

Introduction

The increasing popularity of the Internet, which is expected to reach 6.7 billion users by 2023 [16], as well as the shift towards cloud computing [15], suggest that the development of high performance and reliable networks represents a necessity for accommodating the growing QoS requirements. For instance, in 2020 the input traffic at the Amsterdam Internet Exchange has increased from 4.9 Tbps to 6.7 Tbps [2].

There are generally two approaches for satisfying the growing network traffic [4] [18]. The first one is to upgrade the network devices and increase the network connectivity in order to achieve higher bandwidth. The disadvantage of this approach is that it implies significant costs and does not represent a scalable solution, since it does not decrease the average bandwidth required per connection. Conversely, instead of constantly upgrading the inner network equipment, the network traffic itself can be the target of optimizations. Techniques such as edge computing are meant to perform computations as close as possible to the user in order to prevent additional traffic inside the network. In-network computing is an emerging paradigm in the area of network processing which is meant to directly address this issue, by performing some of the computations in the network devices.

In order to perform software-based network packet processing we can use network switches that are based on general purpose CPUs. This way a switch can inspect a packet and perform complex computations before passing it on. However, due to the increasing QoS requirements the usage of software switches in high-performance environments is limited, where hardware based solutions are preferred. Nonetheless, software switches are still used in data centers at the level of the hypervisors, as well as, in other parts of the network where a flexible computational model has a higher priority than the performance.

The hardware switches in the form of fixed function ASICs are able to satisfy the growing network traffic. However, their usage results into rigid systems, because it is not possible

to modify the built-in algorithms. Such systems are costly and time consuming to upgrade, since it requires the vendor to develop new hardware and firmware in order to accommodate the required updates. Hence, Software Defined Networking (SDN) [11] and Reconfigurable Match Tables (RMT) architecture [7] were developed to address this issue. They make it possible to have configurable networks while preserving the line rate packet processing of the fixed function ASICs. These technologies have formed the foundation of programmable networks and have sparked the research interest in the area of in-network computing. Moreover, multiple technologies have been developed in order to address the shortcomings of the initial models. Even though significant progress has been made, there are still research areas to be addressed. One of them is in-network stateful data analytics.

Stateful data analytics are an important part of in-network computing, since many modern applications require stateful processing of their packets e.g., aggregate queries on distributed database systems, real time processing of IoT data, load balancing, as well as, traffic filtering. Even though these operators can be implemented in the end-hosts or in middle boxes, their in-network implementation can provide significantly better performance since it reduces the number of round trips and can take place closer to the client's data, thus, resulting in better QoS. However, not all end-host operators are suitable for in-network computation, due to the limited resources and high performance requirements of the network devices. For instance, loops are not possible during the processing of a packet since they may take an indefinite amount of time during which all the packets in the pipeline will stall. Another issue is the limited amount of SRAM. This is addressed in the Table Extension Architecture (TEA) [10] by using the RDMA protocol for storing the state in the DRAM of the connected servers. However, this technique also uses packet recirculation, which leads to higher latency. Hence, the operators that are suitable for the implementation in the network devices require only a small amount of data to be stored per flow, such as key-value caching along with data reduction and data aggregation queries, as well as the ones that do not require intensive computations per packet.

There is currently a low understanding on the viability and acceleration capabilities of performing stateful data analytics inside the network. Our goal is to address this shortcoming by performing an analysis of the currently available technologies, that facilitate this kind of computations, and identify the trade-offs and limitations. In order to do this, we will implement two stateful operators *i.e.,* simple moving average, and median over a finite set of possible values, using DPDK [8] as a representative of end-host computing frameworks and P4 [6] in combination with BMv2 [5] as an in-network computing framework.

2

Background

In the recent years, significant progress was made in the area of programmable network switches. Thus, instead of forwarding packets throughout the entire network, in order for them to be processed on the end hosts, it is now possible to perform some computations inside the network, thus, decreasing latency and the network traffic. In order to analyze the performance of stateful operators across several network programming models, we have used the concepts and technologies presented in this chapter.

2.1 Data plane programming

The configuration and operation of programmable switches is based on a network architecture approach called "Software Defined Networking" [11]. SDN dissociates the network interactions into the control plane and the data plane, as well as, the management plane which is outside the scope of this project. The data plane is concerned only with the processing of the network packets, while the control plane is used for the configuration of the network devices.

The separation between the control and the data planes reduces the complexity of the network applications, by splitting them into modules with well-defined purposes. Moreover, the abstractions provided by SDN make the maintenance and further development of such applications more manageable. All the components of the SDN architecture are represented in Figure 2.1.

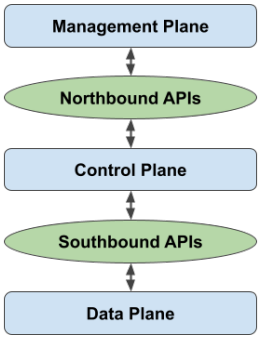


Figure 2.1: SDN Architecture

The management plane represents the highest level of abstraction. This tier includes applications that monitor,

manage and interact with controllers. The interaction takes place through the Northbound APIs. Currently, multiple open and proprietary protocols, that connect the management and control planes, exist.

The control plane is responsible for the administration of the data plane. Each controller is liable for a series of network devices, for which it creates and updates the routing tables, manages the internal state, creates new data flows, as well as, handles the situations which the switches are not able to. Controllers are usually implemented on general purpose CPUs, which allows the usage of a rich computational model. The communication between the control and data planes takes place through the Southbound APIs, where the best known and widely used protocol is OpenFlow [12].

The data plane represents the main focus of this project. It includes all the operations related to the processing of network packets, such as forwarding, manipulations of packet headers, as well as, more complex computations, such as telemetry and QoS related operations. The majority of data plane operations are stateless, because most of the network devices have very limited resources and high performance requirements, which leads to complex solutions for the implementation of stateful operations. Therefore, for a long time it was preferred to keep the application logic away from the network devices and implement it in the end-hosts. However, in the recent years significant progress has been made in the area of programmable networks, which has lead to the development of data plane programming frameworks which facilitate the implementation of complex computations, among which are stateful operators.

Moreover, programmable data planes are diverse in terms of the underlying hardware. They can be based on CPUs, GPUs, NPUs (Network Processing Units), FPGAs (Field Programmable Gate Arrays), as well as, on programmable ASICs (Application-Specific Integrated Circuits). Due to this heterogeneity, the data plane programs are often vendor-specific. Therefore, the code portability is a problem of programmable data-planes. P4, which is presented in section 2.3, is meant to address this issue by providing a common programming language, as well as, an architectural abstraction regardless of the underlying hardware.

2.2 Data Plane Development Kit

Data Plane Development Kit [8] is an open source project hosted by the Linux Foundation [9]. It represents a set of software libraries and drivers, that are meant to accelerate

the processing of the network packets on general purpose CPUs. The main sources of performance acceleration are the kernel bypass and the usage of huge memory pages.

The kernel bypass is aimed at the minimization of context switches triggered by the application. This is achieved by avoiding, or bypassing, the usage of the kernel network stack. The packet processing takes place in the user space, where a custom network stack can be implemented using the libraries provided by DPDK. This way the application requires fewer system calls, which leads to fewer context switches, and ultimately to higher performance.

The usage of huge memory pages is meant to reduce the number of TLB (Translation Lookaside Buffer) misses. A TLB miss occurs when the requested virtual page is not in the TLB. In this case an ordinary page table lookup occurs, which is significantly slower than the TLB mapping. The huge pages solve this problem by minimizing the number of virtual pages required by an application, which, in turn, reduces the number of entries to be saved in the TLB. For instance, if we have a TLB which can store 64 entries and an application that requires 64 MB to operate, with an average Linux page size of 4 kB we would require 16,384 virtual pages, the mapping to which can, obviously, not be stored entirely in the TLB, hence, multiple TLB misses will occur. On the other hand, with huge pages of 2 MB we will only require 32 page table entries, which can be stored altogether in the TLB.

Therefore, huge pages along with kernel bypass are two important techniques that allow DPDK to accelerate the performance of network applications based on general purpose CPUs, and bring the packet throughput close to the line rate of the network interface card used. Hence, we have used DPDK as a representative of the end-host computing frameworks, as due to its high performance potential it serves as a fair comparison to the in-network computing frameworks.

2.3 P4

A significant breakthrough in the area of programmable network switches, has been made with the introduction of the RMT model. The Reconfigurable Match Tables [7] architecture consists of a pipeline with multiple modules, such as parsers, deparsers, reconfigurable match-action units, as well as, fixed-function units. The Figure 2.2 represents an example of a switch architecture based on the RMT model.

The main benefit of the RMT architecture is the possibility to program most of the pipeline. For instance, we can configure the parser(s) and deparser(s), so that custom

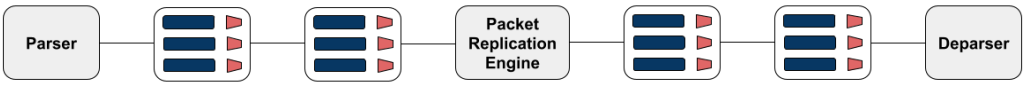


Figure 2.2: RMT pipeline example

headers can be processed without changes to the underlying hardware. Moreover, it is possible to modify the properties of the match tables, such as their size and content, as well as, to create and execute custom actions. Therefore, this architecture facilitates the development of reconfigurable network switches, which support a programmable data plane. An example of such a switch is Intel Tofino 2 [1], which can achieve a remarkable throughput of 12.8 Tb/s. Hence, network switches that are based on the RMT architecture cannot only achieve high performance, comparable with the fixed function chips, but, also, allow partial configuration of its components using software based solutions, both at runtime and compilation time.

An RMT based chip does not represent a complete solution for operating a programmable data plane. An SDN controller which will manage the switch through a Southbound API is required. However, this architecture does not define a standard control protocol, and neither a programming language, that could be used to operate such a switch. The solution to these two problems, has come in the form of a domain-specific language called P4, which is meant to standardize the development of data plane applications targeted towards RMT based devices, as well, as to provide a Southbound API called P4 Runtime API [19], which facilitates the communication between the switch and the controller.

Programming Protocol-independent Packet Processors (P4) only defines the programming language, as well as, the P4 Runtime API. The other components, that are necessary for executing P4 programs are expected to be provided by the chip vendors who want to provide support for P4. These components are : a P4 architecture, a P4 compiler, and the P4 target itself.

A P4 architecture is an abstract representation of an RMT model. The way a P4 program is linked to a specific architecture, is by including the library, that contains all the constructs of that architecture, in the program. In order to implement our stateful operators we have used the P4 architecture called the v1model. It is represented in the Figure 2.3.

This architecture allows the definition of one parser, which is used for extracting the headers from an incoming packet, as well as, one deparser, that can be programmed to

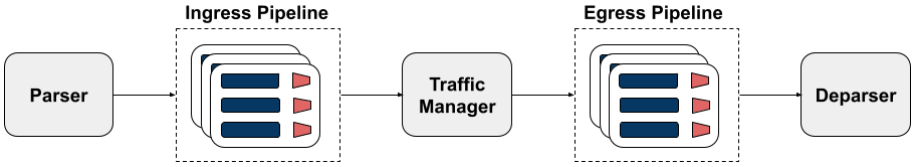


Figure 2.3: v1model architecture

reassemble a parsed packet.

Once the packet is parsed, the headers together with the additional metadata is passed to the Ingress Pipeline. This section consists of a series of Match Action units, which at the end are expected to generate an Egress Specification, that contains a physical port to which the packets will be sent, or, alternatively, a multicast group id, that will cause the packet to be replicated and sent to all the members of the group. Afterwards, the packet is passed to the Traffic Manager which processes the Egress Specification and creates additional packets if it is requested, which are submitted to the Egress Pipeline, where each packet goes again through a series of Match Action units. However, this time it is assumed that the destination of the packet will not change, since it has been determined in the Ingress Pipeline. As the last step the packet is passed to the deparser, from where it is sent to the specified output port.

Besides the control blocks of the pipeline, the v1model also defines a series of externs. A P4 extern is a construct which is not defined in the language itself, but is part of a P4 architecture. The externs provided by the v1model are registers, counters and meters, which are also called stateful memories, since they can maintain state across packets. For the implementation of the simple moving average, and the median over a finite set of possible values, we have only used registers, since they provide all the required functionality. The details of the implementation are presented in the Chapters 4 and 5.

Once the programmer has chosen a P4 architecture and developed the program, it has to be compiled. The P4 compiler is expected to be provided together with the device that is meant to execute the P4 program. Such a device is called a P4 target. The difference between a P4 architecture and a P4 target is that the former is just an abstraction that enables the portability of P4 programs *i.e.,* one P4 architecture might be supported by multiple P4 targets, which implies that the same program can be compiled for multiple targets without significant changes. A P4 target does not, necessarily, have to be a physical device. A virtual switch that supports P4 can also be considered a P4 target. For instance, in order to execute the P4 programs, that were developed as part of this project, we have used the Behavioural Model version 2 (BMv2)[5], which is a virtual switch. We have chosen

BMv2 because it is an open-source project *i.e.,* it is easy to obtain and install, as well as, it is compatible with the network emulation software described in Section 2.4. Adversely, BMv2 is not a production level vSwitch, but it is only meant to be used for checking the correctness of P4 programs that use the supported architectures. However, this fact does not affect our ability to test and debug P4 programs, as well as, to analyze the constraints that arise from the usage of P4 and compare them with the DPDK constraints.

2.4 Network emulation

Network emulation is a technique for testing applications in a environment which resembles as much as possible a production network. Opposed to network simulation, where the analysis is based only on mathematical models, it allows the applications to use the network stack of the operating system, as well as, the network interfaces of the system in order perform the communication. Therefore, by employing this technique it is possible to test, debug, and analyze the behaviour of network applications in a realistic setting.

The network emulator, which has been used for this project, is mininet [14]. This emulation software creates a virtual network of hosts, switches and controllers on a single computer, by leveraging the process-based virtualization provided by Linux since the version 2.2.26. The main benefits of mininet are the ability to create and boot up a virtual network of hundreds of hosts and switches in a matter of seconds, as well as, its open-source nature and good documentation, which makes it a developer friendly choice. Moreover, its extensive Python API facilitates the automation of the virtual network management, as well as, makes it easier to integrate mininet with the other components of the project.

The default mininet switches are instances of Open vSwitch [20], which is a multilayer virtual switch that supports most, if not all, of the standard protocols. Therefore, OvS has been used in the virtual networks where the DPDK implementations were tested, since no additional functionality was required from the network switches. On the other hand, in the virtual networks featuring P4, the OvS switches were replaced with instances of BMv2, since OvS is not a P4 target.

While the features offered by mininet are enough for creating virtual networks featuring P4, in order to test the behaviour of the DPDK applications, we have decided to use containernet [17]. Containernet is an extension of mininet that allows the usage of Docker [13] containers as hosts in the emulated networks. The main benefits of implementing DPDK applications inside a container are a better management of the virtual network, as well as, a more straightforward development and deployment of the application.

3

Design Considerations

The development of network applications is significantly different than the one of the standalone applications. It involves additional considerations, such as protocol design, QoS indicators, and service availability, which are motivated by the high performance requirements of such applications. These requirements are critical in the environments that use DPDK and P4, since the main reason for using these technologies is to accelerate as much as possible packet processing. For instance, such environments are high frequency trading platforms, incident response systems, missile defense systems, and many other time critical applications. For this reason, the computational models are aimed at achieving high performance. However, this leads to a series of constraints, which are discussed in this chapter.

3.1 Mininet constraints

Besides its numerous benefits, mininet, and by inheritance containernet, has several limitations. The first one is that it is only available on Linux platforms, since the mechanisms through which network emulation is achieved are based on Linux specific virtualization techniques. This constraint is easily addressable through the usage of a virtual operating system, in the case when the host OS is not Linux-based. The second limitation is related to the underlying hardware. That is, the virtual network created through mininet is meant to be run on a single device, therefore, the cumulative computational power of the virtual hosts and switches cannot exceed the one of the underlying hardware. The same applies to the throughput and latency of the virtual links. For this reason, it is not possible to compute representative performance metrics using mininet, since in a production setting, such as a data center, there are significantly more computational resources available than

on an average laptop, on which this project was developed. Hence, we have used mininet only to analyze the behaviour, and test the correctness, of our applications.

3.2 DPDK constraints

DPDK can be executed in the user space of an operating system, on a general purpose CPU architecture such as x86. Since it represents a collection of C libraries, the applications that feature it are also, usually, developed in C. For this reason, the computational model does not have any additional limitations. Clearly, the usage of non-constant time operators for the processing of network packets will significantly lower the performance of the application. However, the framework does enforce any constraints in this sense, but makes the developer entirely liable for the memory and time complexity of the application.

3.3 P4 constraints

A network application that features P4 must satisfy a series of constraints. They are the language constraints, architectural constraints and target constraints. The first group of limitations, the language constraints, are universal to all P4 programs. The most notorious one is the lack of loops. It is explained by the pipeline structure of the packet processing model, for which the P4 language was designed. The lack of loops is crucial in such a model, because the processing of a single packet can stall the entire pipeline, which leads to huge performance implications. For this reason, the time complexity of the network applications developed in P4 must be constant, which is enforced at the level of the language. However, it is possible to simulate loops through a technique called packet recirculation. This is when a packet that has passed through the entire pipeline, instead of being sent to a neighbouring network device, is inserted back into the input queue of the pipeline. Even though this way we can mimic the iterations of a loop, by processing the same packet multiple times, it results in significant performance drops. For this reason, the algorithms that have a non-constant time complexity are not suitable for a P4 implementation.

Another P4 language limitation, is the lack of support for decimal numbers. Even though the operations on fixed point numbers, except division, can easily be implemented by using integer operations, floating point numbers are unavailable in the absence of an FPU (floating point unit), which is, usually, not present, or at least not as a part of the pipeline, in the ASIC switches that support P4. Therefore, the language does not

offer native support for such numbers, however, they might be implemented using externs, similarly to how registers and counters are implemented.

The second group of constraints, the architectural ones, result from the P4 architecture for which the program is developed. As mentioned in section 2.3, we have chosen the v1model, since the features that are provided by this model are sufficient for the implementation of our custom operators. An alternative architecture supported by BMv2 is the Portable Switch Architecture [3]. PSA is a more elaborate architecture, that contains more control blocks, as well as, provides additional packet paths. Even though PSA is superior to the v1model, since it provides all the functionality of the v1model and more, we did not require most of its features. For this reason, we have decided to choose a less complicated model.

The v1model offers three externs : Registers, Counters, and Meters. In order to implement the simple moving average and the median operators we have only used registers. A constraint that has emerged from the usage of registers is that their number, as well as, the data types of the values to be stored, must be known at compile time. This is a significant limitation, since this implies that we cannot dynamically acquire and release resources at runtime depending on the network traffic. In order to address this shortcoming, we have decided to preallocate big arrays of registers at compile time, and manage them through the control plane at the runtime. The disadvantage of this approach is that it sets an upper bound on the amount of registers available to the application, which can only by altered by recompiling the program.

Ultimately, the target constraints are imposed by the software or hardware device that executes the P4 program. As described in section 2.3, BMv2 has been chosen as our P4 target. BMv2 does not allow division at runtime. This is a crucial constraint that has influenced the implementation of the simple moving average. It is possible to overcome this limitation, but only in the case when the divisor is of the form $2^k, k \in \mathbb{N}$, by using bit shifting operators. Another constraint is that BMv2 limits these operators to shifts of at most 2^8 bits. A solution is to use precomputed values, or, in the case when these values are only known at runtime, we can use the controller to store them in registers or table entries. The last constraint that we have identified, is that the bit strings are limited to 2048 bits. This limitation did not significantly affect our implementations, however, when the usage of long bit masks is required, this might cause design changes of the algorithms.

The P4 constraints enumerated above do not represent an exhaustive list. These are only the limitations that have directly affected the design decisions taken in this project.

4

Simple Moving Average

In this chapter we describe the Simple Moving Average operator, provide details regarding its implementation and argue the design decisions taken, as well as, perform an analysis of the P4 and DPDK implementations. Since, we did not have access to the hardware specifications of the state of the art P4 targets, such as, Intel Tofino, we have focused our analysis on the computational models, as well as, on determining the trade-offs between these two frameworks, in terms of stateful processing. Moreover, we have identified the situations and flow types that will benefit the most from each of these frameworks.

4.1 Operator Specification

This operator is based on the following requirements. We assume that we have an infinite data stream and we want to compute the average of the last n values received. These values constitute a window, and, since the window is updated every time a new packet *i.e.,* a new value, is received, it is a moving average. Moreover, the values are assumed to be integers.

The Algorithm 1 represents a high level overview of the operator. As it can be observed, the window sum and the index are updated every time a new value is received. The computation of the average is straight forward *i.e.,* $avg = winsum \ / \ n$. Even though this pseudo code features the idea behind the operator, it

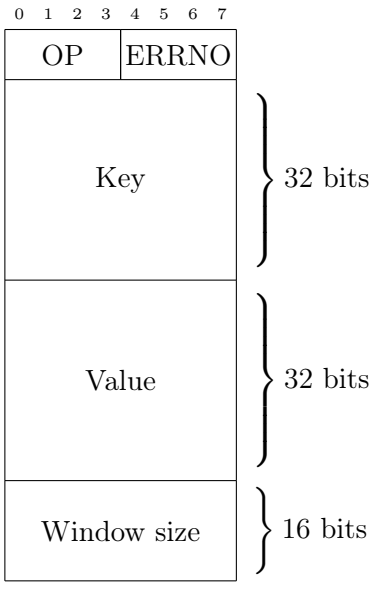


Figure 4.1: SMA header

Algorithm 1 Simple moving average

```
1: procedure SMA(n, flow)          ▷ n - window size, flow - infinite data stream
2:     window ← array of size n    ▷ It holds the last n values of the flow
3:     winsum ← 0                  ▷ The sum of the window values
4:     index ← 0                   ▷ The index of the oldest value in the window
5:     for val in flow do
6:         winsum ← winsum − window[index] + val
7:         window[index] = val
8:         index ← (index + 1) mod n
9:     end for
10: end procedure
```

does not explain the communication between the network entities, and neither the management of the resources on the network devices, which, together, have represented the biggest challenges of this project, and are discussed in detail in sections 4.3 and 4.4.

In order to perform the communication across a network, we have decided to implement a custom protocol. Since we did not have any restrictions in terms of the network architecture, we tried to make it as simple as possible, with the intention to emphasize the implementation of the stateful operator itself. Therefore, we have built a custom header on top of the IPv4 header, which is represented in Figure 4.1.

The operation field (OP) indicates which of the supported operations is requested. The protocol supports 4 operations :

- CREATE (0x0) - create a new data flow.
- ADD (0x1) - add a new value to the flow.
- GET (0x2) - get the current result.
- REMOVE (0x3) - remove a data flow

The error number field (ERRNO) specifies the status of the request. There are 4 possible values of this field :

- SUCCESS (0xF) - the operation has been successfully executed
- UNKN_OP (0x1) - unknown operation
- UNKN_KEY (0x2) - the given key is unknown

- NO_RESRC (0x3) - the server does not have enough resources for satisfying the request

The other 3 fields are self explanatory, and their use cases are discussed in detail in section 4.2.

4.2 Algorithm

In order to implement the procedure presented in Algorithm 1, as a network function, we need to consider some additional aspects, such as, flow identification, resource management and communication handling. Therefore, we have decided to implement the operator using the request-response pattern, which supports 4 operations, as presented in section 4.1. For identifying the data flows we can generate a random key per flow, which can be used in combination with the data structure presented in Listing 4.1 to store all the required flow data in a hash table.

Listing 4.1: SMA flow table entry

```
{
    window,        // an array holding the window values
    window_size,   // the size of the window
    index,         // index of the oldest value in the window
    window_sum     // the sum of the window elements
} SMA_entry
```

Each SMA flow must start with a CREATE request. The processing of such a request is presented in Algorithm 2. Through this operation, a host requests the allocation of resources for processing a potentially infinite stream of data using the SMA Operator. Moreover, the client must specify the size of the window.

Algorithm 2 SMA CREATE

1: **procedure** CREATE(sma_hdr, $flow\_table$)
2: $key \leftarrow$ GENERATE_RANDOM_KEY
3: $window \leftarrow$ ALLOCATE_ARRAY($size = sma\_hdr.winsize$)
4: $entry \leftarrow$ CREATE_SMA_ENTRY($window, sma\_hdr.winsize, 0, 0$)
5: $flow\_table$.ADD($key, entry$)
6: $sma\_hdr.key \leftarrow key$
7: **end procedure**

If the server has enough resources, it will return the error number `0xF` (success) together with the key of the flow. The key can be used by the client in its subsequent requests. It can also be shared with the other hosts on the network, which can contribute to the same flow.

Once the flow is established and the client is in possession of a key, it can start adding values to the flow. In order to do this, the client must specify the flow key, as well as, the value to be added. The processing of this request is presented in Algorithm 3. If the key is correct *i.e.,* there exists a flow with such a key on the server, the client will receive a copy of the SMA header sent, but with the error number `0xF`.

Algorithm 3 SMA ADD

1: **procedure** ADD(sma_hdr, $flow\_table$)
2: $\{win, win\_size, index, win\_sum\} \leftarrow flow\_table.\text{GET}(sma\_hdr.key)$
3: $win\_sum \leftarrow win\_sum - win[index] + sma\_hdr.value$
4: $win[index] \leftarrow sma\_hdr.value$
5: $index \leftarrow (index + 1)$ `mod` win_size
6: **end procedure**

After a series of values are added to a certain flow, the client, or some other entity in possession of the key, might want to retrieve the current result. In order to do this, the client must send a get request, which is processed by the server according to Algorithm 4. As it can be observed, if the given key is valid, the server returns the sum of the values in the window together with the size of the window. Even though, these values do not represent the result, the computation of the final value is straight forward *i.e.,* $result = window\_sum \ / \ window\_size$. The main reason, for returning the result in this form, is the inability of the P4 targets to perform division at runtime, with the exception of the case presented in section 3.3, when the divisor is of the form $2^k,\ k \in \mathbb{N}$. Additionally, returning the dividend and the divisor does not bound the final result to a certain precision, but allows the client to decide upon the number of decimal points itself.

Algorithm 4 SMA GET

1: **procedure** GET(sma_hdr, $flow\_table$)
2: $\{\_, window\_size, \_, window\_sum\} \leftarrow flow\_table.\text{GET}(sma\_hdr.key)$
3: $sma\_hdr.value \leftarrow window\_sum$
4: $sma\_hdr.window\_size \leftarrow window\_size$
5: **end procedure**

The last SMA request is used for removing data flows. This is a very important operation, especially in the case of the in-network computing frameworks, which have very limited memory for storing the state of their applications. The processing of this request is presented in Algorithm 5. If the client specifies a valid key, the server will release all the allocated resources for that flow, and return the `0xF` error number. After this operation, any other request that involves that key will result in the Unknown Key error.

Algorithm 5 SMA REMOVE

```
1: procedure REMOVE(sma_hdr, flow_table)
2:     entry ← flow_table.GET(sma_hdr.key)
3:     DEALLOCATE(entry)                          ▷ Release resources
4:     flow_table.REMOVE(sma_hdr.key)
5: end procedure
```

4.3 P4 implementation

In spite of the constrained computational model of the P4 framework, the language offers built-in support for hash tables, which in P4 are called exact match tables. Therefore, the implementation of the flow table is straight forward. On the other hand, the flow table entry, presented in Listing 4.1, is significantly harder to implement, because the only P4 stateful constructs that are appropriate for this situation are registers, which can only be allocated at compile time. In order to solve this problem we have decided to preallocate arrays of registers, which are managed at runtime by the controller. Since the window size remains constant per flow, we can store it, together with the internal index of the flow, in the flow table. Hence, we require three register arrays for storing the other SMA_entry data.

The first array is the one holding the values received within the active window. Since this array must hold the values of multiple flows, we have to keep track of the starting index of each window, as well as, its size. This information is stored in the flow table.

The second array stores the index of the oldest window value per flow. Even though we only need to store one such index per flow, due to the fact that the size of the windows is not fixed, this array must be of the same size as the first one. This is because the worst

	Flow 1			Flow 2					Flow 3	
Values	1	7	8	25	4	12	79		1	3
Indexes	1	X	X	4	X	X	X	X	0	X
Sums	16	X	X	120	X	X	X	X	4	X

Figure 4.2: SMA registers example

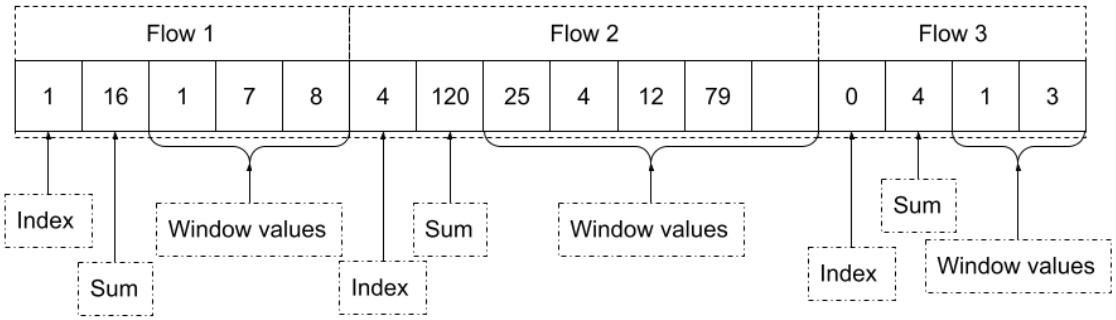
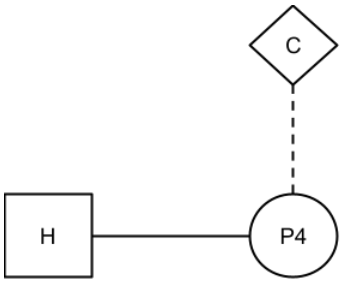


Figure 4.3: SMA registers example 2

case scenario is when we have multiple flows with the window size equal to one. The third array holds window sums, and is implemented similarly to this one. An example of the occupation of these arrays is presented in Figure 4.2. As it can be observed, significant amount of memory is lost in the arrays two and three, however, this is the price we have to pay for having variable sized windows in an environment where dynamic memory allocation is not available. In order to optimize the memory usage, we could impose constraints on the window sizes. This will allow us to allocate smaller arrays for the indices and the sums. Alternatively, we can combine the three register arrays and store the index and the sum as a header in front of the flow window, as presented in Figure 4.3. This approach will prevent the redundant register cells of the `Indexes` and `Sums` arrays, however, it is necessary that the registers of all these arrays to have the same bit size. This constraint might also result in memory loss, in the case when one of these type of values has a different bit size. For instance, if the `Sums` value requires 64 bits, while the other ones only 32, we will have to set the size of all the registers in the array to 64, which results in a loss of 32 bits per value. Therefore, different implementations of the Listing 4.1, result in different use cases, where a choice has to be made based on the parameters of the data flows.

In order to test and debug the P4 implementation of the operations described in section 4.2, we have used the topology presented in Figure 4.4. Clearly, this model is oversimplified, since in a real scenario we have multiple hosts that contribute to the same flow, which is the main reason for implementing this operator as a network function. However, a simple topology allows us to focus the attention on the main components of the application, which are the implementation of the request-reply pattern, as well as, the stateful processing of the SMA packets inside the switch. These components can, later, be used on more intricate topologies and use cases.

The SDN controller is an important part of the operation of the switch. Even though the P4 targets offer constructs that can be used for implementing stateful operations, these constructs cannot be managed directly from the data plane. Therefore, a control plane is necessary.



H - network host that uses the SMA operator
C - SDN controller
P4 - P4 switch

Figure 4.4: SMA P4 topology

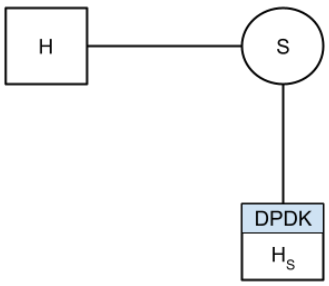
The creation and removal of a data flow are the operations that have to be handled by a controller, because they require manipulations of the flow table, which can only be done by the control plane. Therefore, when such a request reaches a P4 switch, that supports the SMA operator, it will be forwarded to the controller, responsible for that switch, for further processing. In order to keep track of the occupation of the register arrays, mentioned before, the controller stores a bitmap of the array holding the window values. This way, when a creation request is received it will parse the bitmap in order to find a segment able to store a window of the requested size. If it cannot find any available space, it will report that it does not have enough resources for establishing the flow. Otherwise, all the required resources will be reserved on the P4 switch and a new entry will be added to the flow table, which is also stored in the switch, using a randomly generated key. The key will be sent in response to the client. The removal of a flow is implemented in a similar fashion, however, instead of acquiring resources, the controller deallocates all the resources in possession of the flow, as well as, invalidates the flow key.

The implementation of the ADD and GET operations is implemented similarly to the pseudo code presented in the Algorithms 3 and 4, and are processed entirely on the P4 target.

4.4 DPDK implementation

The computational model supported by DPDK is significantly less constrained than the one offered by P4. Nonetheless, in order to achieve high performance, the developer should preserve the constant time complexity of the operator. Even though, the C programming language, for which DPDK has been developed, does not provide built-in hash tables, they are offered by DPDK as part of the `rte_hash.h` library. Therefore, the implementation of the flow table is straight forward.

An advantage of DPDK over P4 is the support for dynamically allocated memory. Hence, we can implement the SMA_entry, defined in Listing 4.1, using a dynamically allocated C `struct`. The array holding the window values, can also be dynamically allocated. The advantage of this method is that we do not have to allocate redundant memory, as well as, we can easier manage the application's state since no additional mapping between the flow key and the flow data is needed, as all the flow data is stored in a single structure.



H - network host that uses the SMA operator
S - network switch
H_s - network host that executes the DPDK program *i.e.,* SMA server

Figure 4.5: SMA DPDK topology

The applications developed with DPDK do not require a control plane for the management of their stateful constructs, since they are executed on general purpose CPUs and have direct access to the memory. Therefore, the network topology that has been used for testing and debugging the DPDK implementation does not feature a controller. It is presented in Figure 4.5.

The operations described in section 4.2 are entirely executed on the DPDK server. Their implementation is very similar to the pseudo code. The presence of pointers and the ability to dynamically create and group data structures make the development easier and increase the maintainability of the program. However, a particularity that must be considered when developing applications with DPDK, and which can potentially be the source of incorrect computations, is endiannes. The preferred network ordering is Big-endian, while most of the processor architectures, such as x86, prefer Little-endian. Therefore, a conversion of the packet data might be needed, depending on the CPU architecture used.

4.5 Discussion

As it can be observed in section 4.2, the time complexity of all the operations that constitute the simple moving average is constant. This applies to the DPDK implementation, however, in the case of P4, the creation and removal of a flow is linear. This is due to the lack of pointers and the inability to use dynamically allocated memory. The controller must store a bitmap of the occupation of the register array that contains the window values. This bitmap has to be linearly checked whenever a new flow is created, in order to identify a segment of the array that is large enough for storing a window of the requested size. Therefore, the creation and removal of a flow, besides requiring a controller for the

allocation and deallocation of system resources, which generates additional network traffic, is significantly slower in the case of P4 when compared with DPDK. On the other hand, the processing of the ADD and GET requests is faster with P4, since the computation takes place closer to the client, which implies lower latency, as well as, the ASIC based switches that support P4, such as Intel Tofino, have a significantly higher throughput than the packet processing frameworks based on general purpose CPUs, such as DPDK. Hence, the DPDK implementation is preferred in the case of small, in terms of bandwidth, data streams, where the computation needed for the creation and removal of a flow is considerable in comparison with the amount of ADD and GET requests. Adversely, large, in terms of bandwidth, data streams will benefit more from the P4 implementation, since the effort required for establishing and removing a flow is insignificant, when compared with the processing of the numerous ADD/GET requests.

The memory complexity is linear in both cases. It is linear in terms of the window size, since we need to store the values of the window, as represented in Listing 4.1. The additional data required by the `SMA_entry`, together with the entry of the flow table, constitute a constant amount of memory per flow, and therefore do not influence the memory complexity of the algorithm. Nonetheless, the P4 implementation allocates redundant memory, due to the lack of dynamic memory allocation. Therefore, the DPDK implementation is more memory efficient. The available memory on the underlying device represents the only limitation of the proposed solution. It affects the maximal window size, the number of data flows that can be processed simultaneously, and, consequently, the scalability of the application.

P4 offers a significantly more constrained computational model than DPDK, which results in additional design considerations of the applications, but due to its support for stateful constructs such as register arrays, it allows the development of stateful operators. Hence, we can observe that both P4 and DPDK are suitable for the implementation of the SMA operator. However, the acceleration obtained by each of these frameworks depends on the size of the data streams and the available system resources.

5

Median

In this chapter we describe the implementation of the median over a predefined set values, which is computed over an infinite stream of values, hereafter referred to as the median operator. Similarly to the chapter 4, we will provide the P4 and DPDK implementations of the operator, as well as, will argue the design decisions taken. Finally, a comparison of the implementations will be performed, together with an analysis of the use cases and the acceleration capabilities.

5.1 Operator Specification

We assume that we have a stream of integer values and we want to compute the median, from the beginning of the stream, for each new value received. A naive solution is to store all the values of the stream in an array, and, whenever we get a new value, we add it to the array, sort it ($O(n\ log\ n)$) and return the "middle value" ($O(1)$). The time complexity of this approach is ($O(n\ log\ n)$) per value received, which is prohibitively expensive for high performance packet processing. Alternatively, we can also store the values of the stream, but in an ordered array. This way, whenever we obtain a new value we can use binary search ($O(log\ n)$), to determine the correct position in the array, and insert the value ($O(n)$). Then we can return the "middle value" ($O(1)$). Even though, this approach improves the time complexity from $O(n\ log\ n)$ to $O(n)$ per value, it is still too high for the use cases that require high performance. Moreover, both these approaches require the storage of the entire data stream, which in the case of a stream with billions of values is not feasible, especially for the memory-constrained in-network computing frameworks. Therefore, we have constrained the data stream *i.e.,* all the values of the stream must be part of a finite set of possible values, which is known in advance, that is when the flow is

created. Under this assumption, we have developed an algorithm that is able to compute the median in constant time, per value received. It is presented in Algorithm 6.

Algorithm 6 Median

Require: vals - the array of possible values; flow - the data stream

```
1:  procedure MEDIAN(vals, flow)
2:      median ← NULL
3:      left_dist, right_dist ← 0
4:      SORT(vals)
5:      for val in flow do
6:          if median is NULL then                ▷ The first value is received
7:              median ← val
8:          end if
9:          if val == median then
10:             left_dist = left_dist + 1
11:             right_dist = right_dist + 1
12:         else if val < median then
13:             left_dist = left_dist − 1
14:             right_dist = right_dist + 1
15:         else if val > median then
16:             left_dist = left_dist + 1
17:             right_dist = right_dist − 1
18:         end if
19:         if left_dist == 0 then
20:             median ← MOVE_LEFT(vals)
21:             left_dist, right_dist ← UPDATE_DISTS(vals)
22:         else if right_dist < 0 then
23:             median ← MOVE_RIGHT(vals)
24:             left_dist, right_dist ← UPDATE_DISTS(vals)
25:         end if
26:     end for
27: end procedure
```

In order to avoid the floating point computations, when the size of the array is even, we do not consider the median as the average between the middle values, but as the lower middle value.

The intuition for this algorithm is based on the idea, that an array of numbers represents a sequence of cells and the median is a pointer. Therefore, at any given time the median

can either point to the middle of a cell *i.e.,* when the size of the array is odd, or the space in-between the cells *i.e.,* when the size of the array is even. Hence, if the array is sorted, whenever a new value is received the median will only move to the left or to the right. If the set of possible values is known in advance, then instead of creating a new cell for each new value received, we can represent each possible value as a cell, which can be thought of as a value type. Each value type has internal distances to the left and to the right neighbours. These distances are then updated whenever a value instance is obtained, and based on this we can identify the current position of the median. This approach allows the computation of the median in constant time, per value received, as well as, the memory required for processing a flow is identified during its creation, and remains constant afterwards.

Similarly to the development of the SWA operator, we have created a custom protocol on top of IPv4 that is based on the request-response pattern. The header is represented in Figure 5.1.

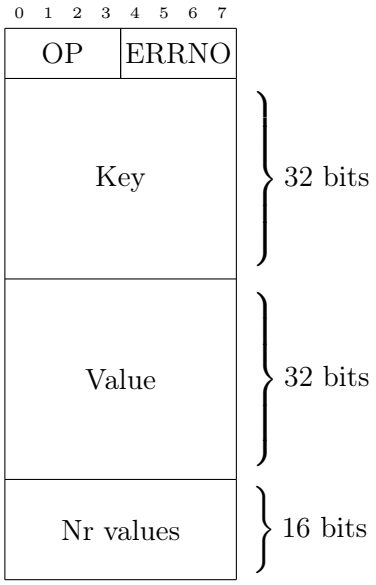


Figure 5.1: Median header

The operation field (OP) indicates the operation that is requested. The protocol supports four operations :

- CREATE (0x0) - create a new data flow
- ADD (0x1) - add a new value to the flow
- GET (0x2) - request the value of the current median
- REMOVE (0x3) - remove the data flow

The error number field (ERRNO) specifies the status of the request. The protocol supports 5 error codes :

- SUCCESS (0xF) - the operation has been successfully executed
- UNKN_OP (0x1) - the requested operation is unknown
- UNKN_KEY (0x2) - the given key is unknown
- NO_RESRC (0x3) - the system does not have enough resources for executing the operation
- UNKN_VAL (0x4) - the given value is not part of the set of possible values

The use cases of the Key and Value fields is presented in section 5.2. The last field - Nr values, is used during the creation of a flow. It specifies the size of the set of possible values. These values must be sent as part of the packet that requests the flow creation.

5.2 Algorithm

The most important parts of the implementation of the procedure presented in Algorithm 6, are the data structures used for storing the flow data, as well as, the procedures used for updating the median when a change is required. These two parts are the key for achieving the constant time complexity of the algorithm. The data structures are presented in Listings 5.1 and 5.2.

Listing 5.1: Median frame cell

```
{
    value;      // the value of the cell
    left_dst;   // distance to the left neighbour
    right_dst;  // distance to the right neighbour
    nr_vals;    // 2 * the number of packets received with this value
} frame_cell
```

A frame cell is created for each element in the set of possible values, that is specified during the creation of a flow. It holds all the necessary information for keeping track of the median, when it points to this cell.

Listing 5.2: Median flow table entry

```
{
    frame;        // an array of frame cells
    bitmap;       // a bit map of frame occupancy
    idx;          // current index of the median
    frame_size;   // the number of possible values in the flow
    val_idx;      // a mapping of the flow values to their indices
} median_entry
```

Similarly to the implementation of the SWA operator, we use a hash table for storing the data of multiple flows. An entry of the flow table contains a flow key, which is used to uniquely identify the flow, and a median_entry, presented in Listing 5.2. The median_entry contains all the data required for the implementation of the Algorithm 6.

Each median flow must start with a CREATE request. This way the client requests the allocation of system resources for the processing of a data stream using the median

operator. The handling of this request is presented in Algorithm 7. It is necessary that the request contains the list of possible values.

Algorithm 7 Median CREATE

Require: *median_hdr* - median header; *vals* - the sorted list of possible values *flow_table* - the flow table

```
1: procedure CREATE(median_hdr, vals, flow_table)
2:     key ← GENERATE_RANDOM_KEY
3:     frame ← INITIALIZE_FRAME(vals)
4:     val_idx ← CREATE_HASH_TABLE
5:     for i in range(vals.size) do
6:         val_idx.ADD(key = vals[i], data = i)
7:     end for
8:     entry ← CREATE_MEDIAN_ENTRY(frame, 0, −1, vals.size, val_idx)
9:     flow_table.ADD(key, entry)
10:    median_hdr.key ← key
11: end procedure
```

If the allocation of resources is successful the server will return a copy of the packet, but with the error code `0xF` (success). The packet will also contain the flow key, which can be used by the client, or some other network entities, in the subsequent requests.

Once the resources are allocated the client can start adding values to the flow. In order to do this, it must send an ADD request, which contains the key of the flow, as well as, the value to be added. If the value is not part of the set of possible values, specified during the creation of the flow, UNKN_VAL error will be returned. Otherwise, the request is processed according to Algorithm 8.

This request is the most complex part of the operator, since we have had to use multiple techniques in order to ensure the constant time complexity. In order to understand it better we can think of it as consisting of three parts. The first one is presented in Algorithm 8. Its purpose is to compare the new value with the current median, update the positional registers, and determine if the value of the median must change.

The second part is described in Algorithm 9. It is meant to increment the counter of the new value's cell. This counter is important, because it is later used for determining the position of the median. Secondly, this procedure updates the bitmap, which keeps track of the frame occupancy *i.e.,* when at least one value of a given type is received, the bit that corresponds to its cell is set to 1.

Algorithm 8 Median ADD

```
1: procedure ADD(median_hdr, flow_table)
2:     {frame, bitmap, idx, frame_size, val_idx} ← flow_table.GET(median_hdr.key)
3:     value ← median_hdr.value
4:     cell_idx ← val_idx.GET(value)                    ▷ frame index of the new value
5:     UPDATE_CELL(frame, cell_idx, bitmap)             ▷ see Algorithm 9
6:     if idx == −1 then                                ▷ the first value in the flow is received
7:         idx ← cell_idx
8:     end if
9:     median ← frame[idx].value
10:    if value == median then
11:        frame[idx].left_dst ← frame[idx].left_dst + 1
12:        frame[idx].right_dst ← frame[idx].right_dst + 1
13:    else if value < median then
14:        frame[idx].left_dst ← frame[idx].left_dst − 1
15:        frame[idx].right_dst ← frame[idx].right_dst + 1
16:    else if value > median then
17:        frame[idx].left_dst ← frame[idx].left_dst + 1
18:        frame[idx].right_dst ← frame[idx].right_dst − 1
19:    end if
20:    if frame[idx].left_dst == 0 then
21:        idx ← MOVE_LEFT(frame_size, idx, bitmap)     ▷ see Algorithm 11
22:        frame[idx].left_dst ← frame[idx].nr_vals
23:        frame[ix].right_dst ← 0
24:    else if frame[idx].right_dst < 0 then
25:        idx ← MOVE_RIGHT(idx, bitmap)                ▷ see Algorithm 10
26:        frame[idx].left_dst ← 1
27:        frame[idx].right_dst ← frame[idx].nr_vals − 1
28:    end if
29: end procedure
```

Algorithm 9 Median UPDATE_CELL

```
1: procedure UPDATE_CELL(frame, idx, bitmap)
2:     frame[idx].nr_vals ← frame[idx].nr_vals + 2
3:     bitmap ← bitmap | (1 << idx)                     ▷ mark the cell as occupied
4: end procedure
```

The third part, is concerned with updating the index of the median, and consists of two subparts *i.e.,* the movement to left and to the right, described in Algorithms 11 and 10. Even though, they are opposed to one another, they are based on the same principles. Let us analyze the movement to the right on a concrete example.

Algorithm 10 Median MOVE_RIGHT

```
1: function MOVE_RIGHT(idx, bitmap)
2:     bitmap = (bitmap >> (idx + 1)) << (idx + 1)
3:     neighbour ← bitmap & ∼(bitmap − 1)
       return LOG2(neighbour)
4: end function
```

We assume that a flow has been created with the set of possible values being equal to $\{2, 4, 5, 7, 9\}$, and until this moment the flow has received the following sequence of numbers : 9, 2, 5, 2. According to our definition of the operator, the median is currently pointing to the value 2, and the bitmap is ***1***0101, where the highlighted bit corresponds to the median. Now we suppose that the value 9 is received again. This implies that the median must move to the right, since the new value is higher than the current median. However, simply moving to the right would result in the value 4 being chosen as the new median, which is, obviously, incorrect. Therefore, we should choose the smallest value, which is greater than the current median, and has been received before *i.e.,* the bit that corresponds to its cell is set to 1 in the bitmap. Hence, if we can identify the position of this bit, we will, automatically, obtain the index of the new median. In order to do this, we can, first, set to 0 all the bits at the indices smaller or equal to the index of the current median *i.e.,* 10101 → 00101. This can be done using bit shifting *i.e.,* $(bitmap >> 1) << 1$. Afterwards, we want to isolate the least significant bit that is set to 1. In order to achieve this, we can subtract from the bitmap the value 1 *i.e.,* 00101 → 11001, and negate the result *i.e.,* 11001 → 00110. The result is then used as a mask on the bitmap which leads to the value 00100. This value can be rewritten as 2^2, where the exponent represents the index of the only bit that is set. Hence, if we apply $log2$ on the result, we obtain the index of the new median.

The movement to the left, described in Algorithm 11, is very similar to the movement to the right. The only difference is that instead of using the bitmap, a reversed bitmap together with a reversed index of the median are used. In the pseudo code we use a function called $REVERSE$, which we assume to reverse the bitmap in constant time. However, neither DPDK nor P4 support the implementation of such function in constant time. In

order to address this limitation, we have added the reversed bitmap to the median_entry, described in Listing 5.2, and update it in Algorithm 9 along with the bitmap.

Algorithm 11 Median MOVE_LEFT

```
1: function MOVE_LEFT(frame_size, idx, bitmap)
2:     rev_bitmap ← REVERSE(bitmap)                          ▷ 0111 → 1110
3:     rev_idx ← frame_size − idx − 1
4:     rev_bitmap ← (rev_bitmap >> (rev_idx + 1)) << (rev_idx + 1)
5:     neighbour ← rev_bitmap & ∼(rev_bitmap − 1)
       return frame_size − LOG2(neighbour) − 1
6: end function
```

After a series of values is added to the flow, the entities that are in possession of the flow key might want to retrieve the current value of the median. In order to do this, they must send a GET request, which is processed according to Algorithm 12.

Algorithm 12 Median GET

```
1: procedure GET(median_hdr, flow_table)
2:     {frame, _, idx, _, _} ← flow_table.GET(median_hdr.key)
3:     median_hdr.value ← frame[idx].value
4: end procedure
```

Finally, in order to release the resources allocated for a specific flow, a REMOVE request must be sent. Clearly, the flow key has to be specified, in order to identify the flow. Once the entity liable for the management of the operator receives the packet it will proceed as specified in the Algorithm 13. If the key is valid, the flow will be removed, the key invalidated and the error code `0xF` (success) will be sent to the client. Otherwise, the error UNKN_KEY will be signaled.

Algorithm 13 Median REMOVE

```
1: procedure REMOVE(median_hdr, flow_table)
2:     entry ← flow_table.GET(median_hdr.key)
3:     DEALLOCATE(entry.frame)
4:     DEALLOCATE(entry.val_idx)
5:     DEALLOCATE(entry)
6:     flow_table.REMOVE(median_hdr.key)
7: end procedure
```

5.3 P4 implementation

The most difficult part in the P4 implementation of the operations presented in section 5.2, was the design of the underlying data structures, due to the inability to dynamically allocate memory, as well as, to manage this memory directly from the data plane. Similarly to the implementation of the SWA operator, the flow table is created using the built-in exact match tables. This table contains a mapping between the flow key and the size of the flow, together with the internal index of the median_entry. Due to the limitations mentioned above, the median_entry, described in Listing 5.2, cannot be implemented directly, using the P4 constructs. Therefore, we have used an additional layer of abstraction which provides a mapping between the fields of the median_entry and the structures available in P4. It is implemented through a series of register arrays, which are preallocated at compilation time and managed at runtime using the control plane. An example that features these arrays is presented in Figure 5.2.

The array registers Left_Dist and Right_Dist only contain meaningful data in the cells that correspond to the current median. Whenever the median changes, the registers at the new index are initialized according to the lines 20-28 of Algorithm 8, using the values of the array Counters. The array Values is self explanatory *i.e.,* it holds the sets of possible values. The array Indices stores the absolute index of the current median, and it only uses the register that is at the same position as the first value in the frame. The last two constructs - Bitmap and Rev_Bitmap (reversed bitmap), are not arrays of registers, but single binary values. As it can be observed, the bits of the bitmap are only set when the corresponding value of the Counters array is greater than 0.

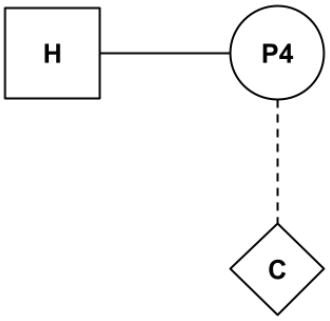
	Flow 1		Flow 2					Flow 3		
Values	1	7	2	3	6	24	42	8	56	91
Counters	0	2	2	8	0	32	0	24	0	64
Left_Dists	-	1	-	-	-	11	-	-	-	20
Right_Dists	-	1	-	-	-	21	-	-	-	44
Indices	1	X	5	X	X	X	X	9	X	X
Bitmap	0	1	1	1	0	1	0	1	0	1
Rev_Bitmap	1	0	1	0	1	0	1	1	1	0

Figure 5.2: Median registers example

The lack of dynamically allocated memory has lead to the modification of the data structure design defined in Listings 5.1 and 5.2. Especially, we can note that the val_idx structure that is meant to perform a mapping between a value and its corresponding index in the frame, has been implemented as a hash table. Since this table is not created for each flow, in order to return the correct index it requires both the value and the flow key, because multiple flows might contain the same value.

Another particularity of the P4 implementation is due to the inability to perform arbitrary bit shifting at runtime *i.e.,* bit shifts of at most 2^8 are allowed by BMv2. Since bit shifting is required by the Algorithms 9, 11 and 10, this constraint would limit the maximum frame size to 256.

In order to overcome this limitation, we can precompute the values needed by these algorithms and store them in hash tables. Similarly, the log2 function can also be implemented using a precomputed hash table, since P4 and the v1model do not support it.



H - network host that uses the Median operator
C - SDN controller
P4 - P4 switch

Figure 5.3: Median P4 topology

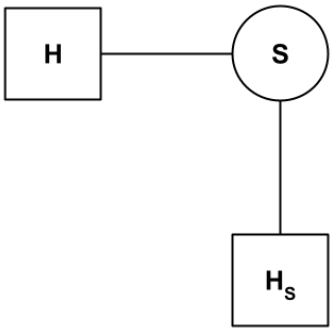
The implementation of the request-response pattern is similar to the one used for the SWA operator. The topology is presented in Figure 5.3. The CREATE and REMOVE requests are redirected to the SDN controller, since the data plane cannot allocate and deallocate system resources. The controller must implement data structures for keeping track of the register arrays' occupation. These data structures are used for determining whether the P4 target has enough resources for processing an additional flow, as well as, their location. The controller also has to communicate with the P4 switch in order to update the flow table, as well as, to allocate the additional resources. The ADD and GET requests are entirely processed on the P4 target, using the Algorithms 8 and 12.

5.4 DPDK implementation

The computational model offered by DPDK is significantly less constrained than the one used by P4. This allows the implementation of the stateful constructs, required by the median operator, to preserve the structures described in Listings 5.1 and 5.2. This way, it is significantly easier to manage the state of each flow, since no additional abstractions are necessary. The field val_idx of the median_entry has been implemented using a hash table. Even though, with P4 we have also used this data structure, in DPDK we can allocate a hash table per flow, instead of a single table for all the flows. This is because in P4 we can only allocate these tables at compile time, while in DPDK we can do this at runtime. The benefit is that we do not need to use the flow key anymore in order to perform the mapping, since only the value is sufficient. Another advantage of DPDK is that, due to the fact that it is implemented using the C programming language, we can use

the *log*2 operator provided in the library `math.h`. Therefore, no additional data structures are necessary for the implementation of the Algorithms 10 and 11.

Similarly to the P4 implementation, we have created a virtual network, in order to test and debug our application. The topology is presented in Figure 5.4. Since the DPDK implementation is executed on a general purpose CPU and has direct access to the memory, a control plane is not necessary for the allocation/deallocation of resources, since the server can do it itself. Hence, the creation and removal of a flow does not generate additional network traffic, as well as, it is faster, since all the operations are executed on the same device. On the other hand, the processing of the ADD and GET requests is slower, when compared to the P4 implementation, since they have to travel through the entire network in order to reach the server. However, the fact that these requests are processed on a general purpose CPU offers more flexibility in terms of the computational model. Nonetheless, the constant time complexity must be preserved, in order to achieve high performance. At the same time, most general purpose CPUs use Little-endian, while the preffered network byte ordering is Big-endian. This particularity should be considered, when processing network packets using DPDK, as it can, potentially, be the source of incorrect computations.



H - network host that uses the Median operator
S - network switch
H_s - network host that executes the DPDK program *i.e.,* Median server

Figure 5.4: Median DPDK topology

5.5 Discussion

The algorithms provided in section 5.2 suggest that the time complexity for processing the ADD, GET, and REMOVE requests is constant. This time complexity has been achieved with the DPDK implementation. However, in the case of P4 the time complexity for removing a flow is linear, because the controller must reset the internal bitmaps that are used for the management of the register arrays, and the time complexity of this procedure is in `O(n)`.

As it can be observed in Algorithm 7, which defines the processing of a CREATE request, the time complexity for creating a flow is linear. This is due to the initialization of the value-index mapping. This mapping has been implemented through hash tables, both in

the case of P4 and DPDK. This structure is crucial in the processing of ADD requests, and it remains constant throughout the entire lifetime of a flow.

Therefore, the operation with the highest time complexity is the creation of a flow. Additionally, in the case of P4 the removal of a flow, also, has linear time complexity. Hence, DPDK offers better performance for the processing of these two requests, considering that all the operations are executed on the same device. However, P4 is more suitable for the processing of the ADD and GET requests, since it can perform the computations as close as possible to the client's data. Moreover, the ASIC switches that support P4, offer significantly higher throughput than the packet processing platforms based on general purpose CPUs. For this reason, we can conclude that DPDK offers better performance in the case of small data streams, while P4 is preffered for the processing of data streams with numerous values.

In terms of the memory complexity, both implementations are linear with respect to the number of possible values. However, the P4 implementation requires slightly more memory, than the DPDK alternative. This is due to the additional hash tables that are needed for the implementation of the Algorithms 9, 10, and 11. These, hash tables are meant to address the lack of support for arbitrary bit shifting of more than 2^8 bits, as well as, to implement the $log2$ operator. They are only initialized during the compilation time, and remain constant for the entire runtime of the application.

The number of values that can be processed by this operators depends on the amount of memory allocated for the fields of Listing 5.1, since the values stored in `left_dst`, `right_dst` and `nr_vals` will overflow in the case of infinite data streams.

The support for dynamically allocated memory makes the DPDK implementation more structured in terms of the state management. It also allows the removal of a flow to take place in constant time. Nevertheless, the native P4 constructs together with the P4 Architecture externs, facilitate the implementation of most of the basic data structures, which, even though, require more resource management, can achieve the same outcome as their alternatives, based on a general purpose programming language.

6

Analysis

The purpose of this project was to assess the acceleration capabilities of performing stateful data analytics inside the network, to determine the constraints of the in-network computing frameworks, as well as, to study the viability of moving stateful operators from the end-hosts to the network devices. In order to address these research questions we have developed two stateful data analytics operators, which are the simple moving average, and the median over a finite set of possible values, and have implemented them using P4, as a representative of the in-network computing model, and DPDK, as an end-host computing framework.

The computational model offered by P4 is significantly more constrained than in the case of DPDK, because the P4 targets are developed, specifically, for high performance network applications. The limitations that have been identified during the implementation of the SMA and Median operators, are :

- **Lack of loops** - Due to the pipelined architectures of the P4 targets, loops are not supported by the language. In order to simulate them, a programmer can use packet recirculation. However, this technique leads to higher latency, and higher utilisation of the system resources.

- **Lack of dynamically allocated memory** - All the memory used by a P4 application must be allocated at the compilation time. This is a critical constraint, especially, for stateful operators, since most of the time it is only possible to determine the amount memory required at runtime. This limitation can be addressed by allocating significant chunks of stateful constructs, such as Registers, at the compilation time. Therefore, at runtime the data plane can use these resources without the need to allocate or deallocate memory, as long as, the required resources do not exceed the ones that are made available at the compilation time. The disadvantage of

this approach is that additional memory management is needed, in order to partition and group these memory blocks.

- **Lack of floating-point numbers** - Considering that most data plane applications only require trivial operations on the header fields of the network packets, P4 targets do not have to perform floating point computations. Therefore, the physical devices that support P4 do not have FPUs (floating-point units), which are required for this kind of computations, as these components occupy additional on-chip space. On the other hand, fixed-point operations can easily be implemented in terms of integer operations. A solution for this limitation is to store and process decimal numbers in fraction form. The disadvantage is that additional application logic is required in order to support this kind of computations, as well as, more memory is needed to store both the divisor and dividend.

- **Limited support for arbitrary bit shifting** - This constraint is imposed by BMv2, which has been used as a P4 target in this project. It limits the number of bits, which can be shifted at once, to 2^8. This limitation makes it harder to create bitmasks at runtime. If the set of these masks is finite for a specific application, it can be implemented through hash maps, by precomputing all the masks at the compilation time. The disadvantage is that additional memory is required, which could be prohibitively expensive when the set of such masks is large.

- **Lack of division at runtime** - BMv2 also does not support division at runtime. This limitation can be addressed in the case when the divisor is 2, by using bit shifting. Otherwise, we could either use precomputed values, or the solution proposed for the lack of floating-point numbers.

- **Lack of complex operations** - P4 does not offer support for complex operators, such as *log*, *sqrt*, *exp* etc. These operators can be implemented using hash maps, if the set of function arguments is relatively small. This approach results in higher memory requirements of the application.

During the implementation of the SMA and Median operators, we have identified three types of data storages, that are necessary for the stateful processing of network packets, which are :

- **Permanent** - the storage of data that is constant throughout the runtime of an application. In order to chance its content the application has to be recompiled.

- **Semi-Permanent** - the storage of data that is constant for a specific data flow. Its content is initialized during the creation of a flow and does not change throughout the flow's life-time.
- **Volatile** - the storage of data that has to be frequently modified throughout the lifetime of a data flow.

The first type of data storage can be implemented using constant values that are hard-coded into a P4 program, or by using constant hash tables. These tables are initialized at the compilation time and have a fixed set of entries, which cannot be modified at runtime. This type of storage is useful for overcoming some of the limitations of the P4 computational model, described above, such as the lack of complex operations and the limited support for arbitrary bit shifting.

The second type of data storage is implementable, mainly, through modifiable hash tables. These tables are, usually, updated when a new flow is created. This can only be done through the Southbound API, which causes additional computations and network traffic, and results in performance penalties. Therefore, this method should only be used for the data that rarely changes throughout the lifetime of a flow, and preferably for the one that remains constant after the initialization of a flow.

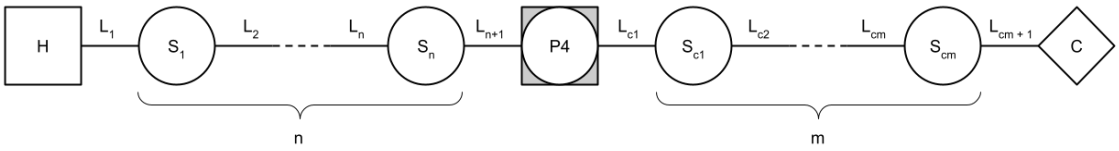
The last type of data storage, is crucial for the implementation of stateful network applications. It can only be, efficiently, implemented using P4 externs. The v1model offers three externs that facilitate this type of storage *i.e.,* Registers, Counters and Meters. Since they cannot be allocated at runtime, we have to preallocate big arrays of these constructs during the compilation time. The partitioning is later done by the controller, during the creation of flows, such that each flow receives a part of these resources, as presented in Figures 4.2, 4.3 and 5.2. This method requires additional abstractions for the management of the register arrays, however, it facilitates the development of stateful data analytics as in-network operators.

DPDK offers a more flexible computational model than P4, since it is executed on a general purpose CPU. All of the P4 constraint mentioned above are absent in DPDK. Since this framework uses C as the programming language and has the ability to dynamically allocate memory, it makes the management of system resources significantly easier, as no additional abstractions are necessary in order to partition and group memory blocks.

Even though, DPDK offers a superior computational model, the main advantages of the applications developed with P4 are the high performance of the P4 targets *e.g.,* Intel Tofino 2 [1] achieves an impressive throughput of 12.8 Tb/s, as well as, the ability to perform the

computations as close as possible to the client's data. Therefore, when we choose P4 over DPDK, we obtain higher performance for the price of a more constrained computational model.

The performance acceleration obtained by moving stateful operators from the end-hosts to the network devices depends on several parameters, which are : the topology of the network, the performance of the P4 target and the corresponding SDN controller, as well as, the performance of the DPDK server. Due to the lack of technical specifications of the state of the art P4 targets, we have only performed a qualitative assessment of the acceleration capabilities.



H - Source of requests	L_i - network link
P4 - P4 switch	S_i - network switch between H and P4
C - SDN controller	L_{ci} - network link
	S_{ci} - network switch between P4 and H

Figure 6.1: P4 topology

The Figures 6.1 and 6.2 describe a generalization of the network topologies used for the implementation of the request-response pattern in the case of P4 and DPDK. In both of these representations we assume that the P4 switch can be installed on the path between the source of the requests and the server which would process them in the end-host computing model. This server is not depicted in Figure 6.1, since it is no longer needed, as the P4 switch in combination with the controller have taken over its attributions.

Based on these representations, let us identify the formulas for computing the expected latency for the creation and removal of a flow, for both of these topologies. The host H will send the request to the server where it thinks that the processing takes place. In the case of P4, once the packet reaches the P4 switch, the switch determines that it can process this protocol and proceeds with the creation/removal of the flow. Since it cannot process this request locally, it redirects it to the controller. The controller allocates all the internal structures and using the Southbound API sends a request to the operating system of the P4 switch, to set the required registers and update the flow table. Once this is done, the controller returns packet, that now contains a flow key, to the switch, which in turn

forwards it back to the host H. This communication results in the following cumulative latency

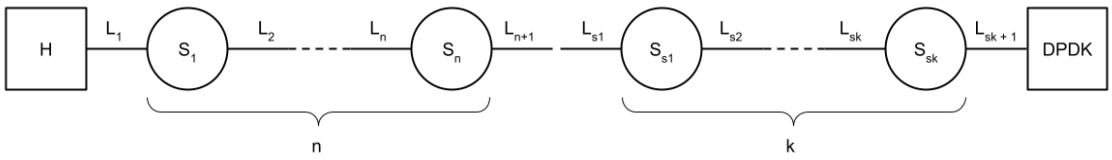
$$2(\sum_{i=1}^{n} S_i + \sum_{i=1}^{n+1} L_i + P4) + 3(\sum_{i=1}^{m} S_{ci} + \sum_{i=1}^{m+1} L_{ci}) + C + T_{P4OS}$$

, where T_{P4OS} is the time required by the operating system of the P4 switch to process the controller's request, and all the other terms represent the time spent by the packet in the corresponding network component.

In the case of DPDK, the request from the host H travels through all the intermediary links and switches, until it reaches the device hosting the DPDK server. On the server, all the required data structures are initialized. Once this is done, the server sends a confirmation to the client, which again has to travel through the intermediary network components. This results in the following formula,

$$2(\sum_{i=1}^{n} S_i + \sum_{i=1}^{n+1} L_i) + 2(\sum_{i=1}^{k} S_{si} + \sum_{i=1}^{k+1} L_{si}) + DPDK$$

As it can be observed, the creation/removal of a flow generates more network traffic and higher latency in the case of P4. This is due to the additional communication between the SDN controller and the P4 target. Therefore, a DPDK implementation shows better performance for handling these two types of requests.



H - Source of requests
DPDK - DPDK server
S_{ci} - network switch between P4 and H
L_i - network link
S_i - network switch between H and P4
L_{si} - network link

Figure 6.2: DPDK topology

Now let us analyze the processing of the requests, using P4 and DPDK, on an, already, initialized flow *e.g.,* the ADD and GET requests. In both cases, the host H sends the requests to the network device where it thinks that the processing takes place. In the case of P4, once the packet reaches the P4 switch, it identifies the packet as part of a supported protocol, so it does not forward the packet to its end destination, but performs the required computations locally. Once this is done a response is sent to the source of the request. Therefore, the cumulative latency of this communication is equal to

$$2(\sum_{i=1}^{n} S_i + \sum_{i=1}^{n+1} L_i) + P4$$

In the case of DPDK, the request has to travel through all the intermediary network components in order to reach the server, where the processing takes place. Afterwards a response is generated, and sent back to the source of the request. This communication results in the following formula,

$$2(\sum_{i=1}^{n} S_i + \sum_{i=1}^{n+1} L_i) + 2(\sum_{i=1}^{k} S_{si} + \sum_{i=1}^{k+1} L_{si}) + DPDK$$

As it can be observed, the processing of requests on an established flow is significantly faster with P4 than with DPDK, as well as, less network traffic is generated. It can also be noticed that in the case of DPDK there is no difference, in terms of the cumulative latency, between the creation/removal of a flow, and the operations of the data stream.

Therefore, the viability of moving stateful operators from end-hosts to the network devices depends on two main aspects *i.e.,* the topology of the network and the characteristics of the data streams. The topology is very important, because the main benefit of the in-network computing model is that it can perform computations closer to the client's data, also known as edge-computing. The most favourable situation is when the programmable switch is on the path between the client and the server, as presented in the Figures 6.1 and 6.2. However, if the requests have to use an alternative path, which is longer than the initial one to the server, the in-network implementation of the operator might result in a performance loss. For this reason, network topology and the positioning of the P4 switches have a crucial role in the acceleration of stateful operators.

The second aspect that influences the viability of in-network stateful processing is the characteristics of the data streams. As we have observed previously, DPDK offers better performance for the creation and removal of data flows, while P4 accelerates the processing of requests on already established flows *i.e.,* operations that do not involve the management of system resources. Therefore, for short-lived data flows, also known as mice flows, an end-host implementation is more suitable, since the amount of computations required for establishing a flow is significant in comparison with the computations performed on the data stream. Adversely, in the case of long-lived flow, also called elephant flows, an in-network implementation might be preffered, due to the acceleration obtained from the processing of the data stream, which outweighs the performance penalties for the creation and removal of the flow.

Hence, the choice between P4 and DPDK, for the implementation of a stateful operator, is influenced by multiple factors, such as the constraints of the computational model, the topology of the network, and the performance of the underlying hardware. For this reason, neither of these approaches represents a universal solution, and should rather be seen as high performance tools that are only preferred for specific use cases.

7

Conclusion

The goal of this project was to analyze the acceleration capabilities of executing stateful data analytics operators inside the network, to assess the viability of such implementations, as well as, to identify the trade-offs, in terms of stateful processing, between different programming models of networks. These objectives have been achieved, by developing concrete stateful operators, using P4 and DPDK, on which a qualitative analysis has been performed. The analysis has resulted in a better understanding of the limitations, use cases, and acceleration potential of the inspected network programming models.

We have also identified the required characteristics of stateful operators which qualify for in-network computing. The most important ones are the constant time complexity and the ability to conform to the constraints of the computational model. Additionally, we have determined the storage types required for stateful processing of data flows, as well as, the constructs and techniques that facilitate the implementation of such storages.

In conclusion, all of the inspected network programming models have their advantages and disadvantages, and neither of them should be considered superior to the other, but rather a part of the ultimate solution, which is a combination between these models. Therefore, a decision should be taken based on the topology of the network, the performance of the underlying hardware, and the nature of the data flows, in order to achieve an optimal performance.

8

Future Work

The main purpose of this project was to assess the acceleration capabilities of different network programming models for the implementation of stateful data analytics. In order to do this, we have designed two operators which have been implemented with P4 and DPDK. Since we have, mainly, focused on understanding the constraints and trade-offs between the computational models offered by these frameworks, we have tried to keep the data structures, that maintain the application's state, as simple as possible. As it can be observed in the case of both the SWA and median operators, register arrays, presented in Figures 4.2 and 5.2, contain redundant memory cells. It is possible to reorganize the data structures and combine several register arrays in order to minimize the amount of wasted memory, however, due to the nature of the register array allocation, that requires all the registers to have the same bit size, it might result in additional limitations in terms of the management of register arrays. Therefore, a further analysis of management techniques of register arrays, supported by in-network computing frameworks, would be beneficial for reducing the amount of memory loss.

Additionally, due to the inability to dynamically allocate memory at runtime, we have had to preallocate big register arrays at the compilation time, which are managed by the control plane at runtime. The memory management has been a complex part in the implementation of the median operator, since there are multiple aspects that have to be considered, such as the data structures that have to be implemented in the controller, in order to keep track of the occupancy of the register arrays, the trade-offs between redundant storage of flow data on the controller and Southbound API queries of the data plane, as well as, the algorithm used for determining the location of a new frame in the shared register array. Therefore, the memory management is an important component of stateful applications, which has proven to be more challenging in an SDN setting. Hence,

additional research in this area will benefit the development process of applications based on the in-network computing model.

Another aspect that is, especially, relevant in the case of elaborate operators, are non-trivial arithmetic operators, such as logarithm, exponentiation and square root. These functions are not supported by P4, however, are important in the development of complex operators. In their absence, the in-network computing model might not be a viable option. The implementation of the median, described in section 5.2, requires the usage of the logarithm, which was possible in P4, because we have only needed the base to be equal to 2 and the arguments were always of the form $2^n,\ n \in \mathbb{N}$. Hence, we have precomputed a series of results which were stored in a hash table. Nonetheless, this approach might not always be an option, and, therefore, additional research is needed in order to understand the limitations of the in-network computing model in terms of the nontrivial arithmetic operations.

As mentioned in chapter 6, the network topology has a high influence on the acceleration capabilities of in-network applications. This aspect of network development, has not been considered in-depth as part of this project. Nonetheless, the research in this area is crucial for identifying the networks that will benefit the most from moving to the in-network computational model.

In the case of both the SWA and median operators, we have implemented custom protocols in order to assure the communication between the network entities. These protocols are unreliable, since we have only used them in virtual networks running on a single device, which have a relatively low packet loss. However, in a real setting the packet loss is expected to be higher. Hence, the further research of the processing of reliable protocols using the in-network computing model will advance the understanding of the limitations of this model, as well as, its use cases.

All in all, the programmable networks are a relatively new research area, which is meant to revolutionize the network architectures, as well as, the development of network applications and their capabilities. Therefore, additional effort will help to discover the full potential of these technologies, and make the next big step towards high performance networks.

9

Acknowledgements

I would like to thank prof. dr. Boris Koldehofe and Bochra Boughzala for their valuable reviews and suggestions, as well as, for our productive meetings that have greatly benefited the development of this project.

Bibliography

[1] Intel Tofino 2, 2021. `www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html`. 6, 35

[2] AMS-IX, 2021. `www.ams-ix.net/ams/documentation/total-stats`. 1

[3] Portable Switch Architecture, 2021. `p4.org/p4-spec/docs/PSA-v1.1.0.pdf`. 11

[4] Saurabh Bagchi, Muhammad-Bilal Siddiqui, Paul Wood, and Heng Zhang. Dependability in edge computing. *Commun. ACM*, 63(1):58–66, December 2019. 1

[5] P4 BMv2, 2021. `github.com/p4lang/behavioral-model`. 2, 7

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014. 2

[7] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, August 2013. 2, 5

[8] Intel DPDK, 2021. Data Plane Development Kit. 2, 4

[9] Linux Foundation, 2021. `www.linuxfoundation.org`. 4

[10] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. SIGCOMM '20, page 90–106, New York, NY, USA, 2020. Association for Computing Machinery. 2

[11] Keith Kirkpatrick. Software-defined networking. *Commun. ACM*, 56(9):16–19, September 2013. 2, 3

[12] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008. 4

[13] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014. 8

[14] Mininet, 2021. `mininet.org/`. 8

[15] Amit Mitra, Nicholas O'Regan, and David Sarpong. Cloud resource adaptation: A resource based perspective on value creation for corporate growth. *Technological Forecasting and Social Change*, 130:28–38, 2018. 1

[16] Cisco Annual Internet Report (2018–2023) White Paper, 2021. `www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html`. 1

[17] Manuel Peuster, Holger Karl, and Steven van Rossem. Medicine: Rapid prototyping of production-ready network services in multi-pop environments. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 148–153, 2016. 8

[18] Dan R. K. Ports and Jacob Nelson. When should the network be the computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 209–215, New York, NY, USA, 2019. Association for Computing Machinery. 1

[19] P4 Runtime, 2021. `p4.org/p4-spec/p4runtime/v1.3.0/P4Runtime-Spec.pdf`. 6

[20] Open vSwitch, 2021. `www.openvswitch.org`. 8