# Lambda functions for network control and monitoring

**Auteur :** Leduc, François
**Promoteur(s) :** Mathy, Laurent
**Faculté :** Faculté des Sciences appliquées
**Diplôme :** Master en sciences informatiques, à finalité spécialisée en "computer systems security"
**Année académique :** 2020-2021
**URI/URL :** http://hdl.handle.net/2268.2/13159

# Lambda functions for network control and monitoring


**Author**:

François Leduc

**Supervisor**:

Prof. L. Mathy


Thesis submitted for the degree of
Master of Science in Computer Science with a professional focus on "Computer systems security"


School of Engineering and Computer Science

University of Liège



Academic year 2020-2021

# Lambda functions for network control and monitoring

François Leduc

Lambda functions for network control and monitoring

https://matheo.uliege.be/

# Abstract

Monitoring a network in a precise manner is becoming more interesting in light of the volume of traffic that new infrastructures can accommodate. With the advent of programmable switches and routers, monitoring systems are turning to solutions that benefit from this new capability. There is also the establishment of a new back-end approach known as serverless computing, which consists in uploading lambda functions to the cloud. These functions offer backend services on an as-needed basis.

The goal of this work is to develop a monitoring system capable of detecting network attacks and specific events of interest to a network operator. To accomplish this, the two previously introduced notions are used, namely a backend architecture based on serverless computing and the assumption that the network is made up of programmable devices.

In terms of packet processing technology, we used XDP, which allows us to create a hook at the switch's network interface and execute a program. The program's goal is to save the headers of IP packets locally. These data are then formatted as custom events and transferred to an intermediate server. In order to do this, we have designed a protocol on top of UDP. The server will then trigger the execution of the lambda functions associated to the events. For their execution we decided to choose Kubeless, a Kubernetes-native serverless framework. A Proof of Concept was created to see if our solution was scalable and possible. We then evaluate the amount of network traffic generated by our approach and discuss protocol limitations.

We conclude by suggesting several sorts of prospective improvements ranging from security to better benchmarking and other architectural options.

# Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Prof. Laurent Mathy as well as his team composed of Cyril Soldani and Gaulthier Gain for their continued support and needed feedback throughout the year.

I would also like to express my heartfelt gratitude to my friends and family for their unwavering support throughout the year, and in particular to Loïc Champagne, Philippe Schommers, and Egon Scheer for their sharp critiques and invaluable assistance in providing me with different perspectives on this highly specialized subject.

Finally, I want to use this occasion to express my gratitude to all the professors at the Faculty of Applied Sciences for their expertise and contributions to my academic success.

Liège, 23rd August 2021

François Leduc

# Contents

# List of Figures

# List of Tables

# Acronyms

**ALF** Application Layer Framing. 40

**API** Application Programming Interface. 3, 5, 6, 13, 17, 30, 33

**BCC** BPF Compiler Collection. 30, 33, 35

**BPF** Berkeley Packet Filter. 12, 13, 22–24, 29–33, 46, 56, 57

**CLI** Command-Line Interface. 6, 13

**CPU** Central Processing Unit. 12, 16, 24, 56, 57

**DHCP** Dynamic Host Protocol. 9

**DHT** Distributed Hash Table. 28

**DNS** Domain Name System. 9, 44

**DoS** Denial of Service. 8

**eBPF** Extended Berkeley Packet Filter. 11–13, 21, 22, 29–33, 35, 49

**FaaS** Function as a Service. 1, 15, 16

**FD** File Descriptor. 34

**HPA** Horizontal Pod Autoscaler. 27, 28

**HTTP** Hypertext Transfer Protocol. 15, 26, 42, 43

**IaaS** Infrastructure-as-a-Service. 15

**ICMP** Internet Control Message Protocol. 9

**INT** In-Band Network Telemetry. 17, 18

**IoT** Internet of Things. 27

**IP** Internet Protocol. 7, 9, 22, 33, 39, 40, 57

**JIT** Just-in-Time. 12, 33

**JSON** JavaScript Object Notation. 40

**LAN** Local Area Network. 24

**LLC** Logical Link Control. 40

**MTU** Maximum Transit Unit. 39, 40

**NIC** Network Interface Card. 20, 33, 56

**OS** Operating System. 2, 4, 5, 11, 16

**P4** Programming Protocol-independent Packet Processors. 13, 17, 20

**PoC** Proof of Concept. 29

**QoS** Quality Of Service. 48

**RAM** Random Access Memory. 4, 51

**SDN** Software-Defined Networking. 1, 3, 13, 17, 18, 23, 57

**SNMP** Simple Network Management Protocol. 1, 9

**SOTA** State of the Art. 2

**TCP** Transmission Control Protocol. vi, 8–10, 24, 39, 42, 46

**UaaF** Unikernel-as-a-Function. 16

**UDP** User Datagram Protocol. i, 9, 10, 24, 36, 39–42, 59

**UI** User Interface. 6

**VM** Virtual Machine. 1, 16, 25

**XDP** eXpress Data Path. i, 11, 12, 20–22, 29, 46, 59

# Chapter 1

# Introduction

Network operators are constantly looking for solutions to continuously monitor events ranging from simple congestion data to potential attacks on the network. This monitoring requires the ability to handle huge amounts of data in real time. Previously some used the Simple Network Management Protocol (SNMP) as their primary data collection method. However, using SNMP requires deploying SNMP agents on every device in the network and collecting them one after the other, allowing very little flexibility. Furthermore, the amount of data to be collected from these devices, or the compression of the data or even the data itself to be collected is quite limited. It does not account for fine-grained statistics such as flows sizes counters for example, an almost pre-requisite when you think about today's network infrastructures such as data centers and Software-Defined Networking (SDN) networks.

Programmable switches allow nowadays to parse packets at line rate and if accompanied by stream processors to parse these data in real time, to be able to interpret the network behavior and track performance impairments or attacks. This allows the programmer to program a switch according to her needs. The switch can be programmed to be a firewall or even a load-balancer or simply an advanced switch with the ability to record data and parse the headers of higher layers such as L2-L3 switches.

Furthermore, we see an increase in interest in cloud technologies, particularly in the design of cloud-hosted applications. A new event-driven paradigm known as Serverless computing, also known as "Function as a Service (FaaS)", emerged a few years ago, allowing programmers to create functions in response to events that together form an application. These functions can interact with one another and use databases, and they are managed and executed by the cloud service provider. These functions, triggered by client-defined events, would be dynamically provisioned as containers and Virtual Machine (VM), and their number would be scaled in accordance with the real-time load.

The goal of this thesis is to develop a draft of a complete monitoring system that makes use of the ability of switches and routers to be programmable, allowing active monitoring of a medium to large scale network, as well as to use a serverless architecture to respond to network events and determine so-called network behavior.

In chapter 2 and 3 the State of the Art (SOTA) technologies will be reviewed as well as similar works on active monitoring of Internet networks and important concepts that might appear in this thesis. In chapter 4, the architecture of the developed solution will be presented. The chapter 5 describes the implementation details of the solution. The simulation environment and the results are discussed in chapter 6. Chapter 7 will conclude this thesis by describing the possible improvements of the different modules of the architecture as well as the improvements regarding the implementation.

**Defining goals**

The project consists in monitoring a computer network in order to detect in real time possible attacks or unusual events. To do this we assume that our network is composed of programmable devices (switches and routers) which implies that each packet received on their respective interfaces can be recorded, processed, modified and forwarded at almost the same speed as state of the art hardware processing technologies.

Each device being programmable means that they have an Operating System (OS) evolved enough to give them the capability to run programs and thus to communicate and send statistics with other network entities such as network controllers or servers.

To each network event would be associated a piece of code (lambda function) defining an automatic decision to be taken. These lambda functions are the core of the serverless architecture. They would analyse the event and based on past events determine whether or not something abnormal is detected and signal the network administrator. Exploring a serverless solution is not trivial and comes with several challenges to ensure proper scaling if the size of the network at hand is important.

# Chapter 2

# Background

## 2.1 Data plane - Control plane

Device control plane and device data plane are conceptually separated in conventional network equipment independently of implementation (e.g., pure software or specific hardware) or purpose (e.g., a switch or an edge router).

Network routing's control plane includes all functions and processes that define how to process packets or how to determine which path a packet or frame should take to get to its destination. For example, routing tables and forwarding tables are filled by the control plane, enabling the data plane's capabilities. In a nutshell, it is responsible for how packets should be routed.

The data plane, on the other hand, refers to all the functions and processes that use control plane logic to move packets/frames from one interface to another. It is responsible for executing what has been decided in the control plane often with high performance. Forwarding plane is another word for it.

Software-Defined Networking (SDN) is a new approach that separates physically the control plane and the data plane in order to be able to flexibly modify the rules contained inside the switches and routers tables. The control plane in a traditional network is distributed between all the routers of the network and communicate with each other to ensure proper routing. However, in SDN Network this would be calculated by an sole entity called a SDN controller that possesses the full network knowledge such as the full topology, the cost per link, the bandwidth available and can run many advanced algorithms to ensure flexible routing thanks to real time feedback from the network devices. This type of architecture is shown on Figure 2.1.

There are several popular APIs and protocols used for communication

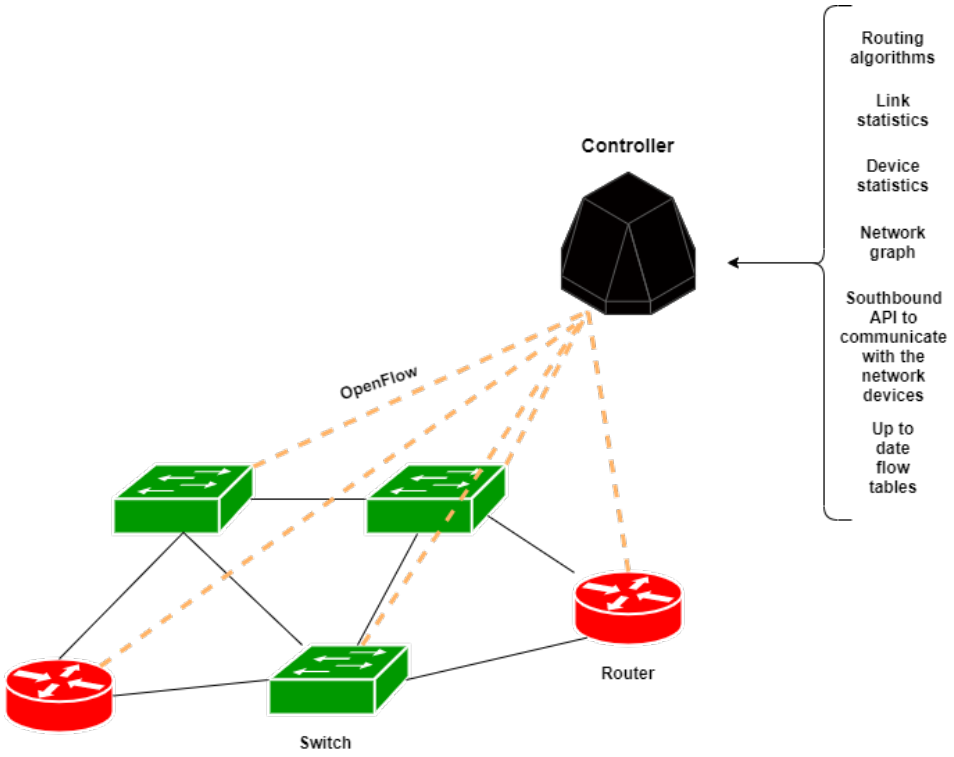



Figure 2.1: Typical SDN architecture

between the controller and routers including OpenFlow [21], P4RunTime [27] , ForCES [38] and many more.

## 2.2 Kernel Space vs User space

Random Access Memory (RAM) is separated into two parts: kernel space and user space respectively. The kernel runs in kernel space, which is inaccessible to other applications. It has full access to all memory address spaces as well as all underlying hardware. It is only used for the most trustworthy functions within a system. Kernel mode is often designated for the operating system's lowest-level, most trusted functions.

User programs must execute in user space. User space is a type of sandboxing in which user programs can only access memory that has been assigned to them by the OS so that they do not interfere with other programs nor the kernel.

This in mind we must be able to discern between the execution of operating-system code and user-defined code in order to ensure that the operating system runs properly [31]. On most computers, this is accomplished by supplying hardware capability for distinguishing between execution modes. A minimum of two separate modes of operation

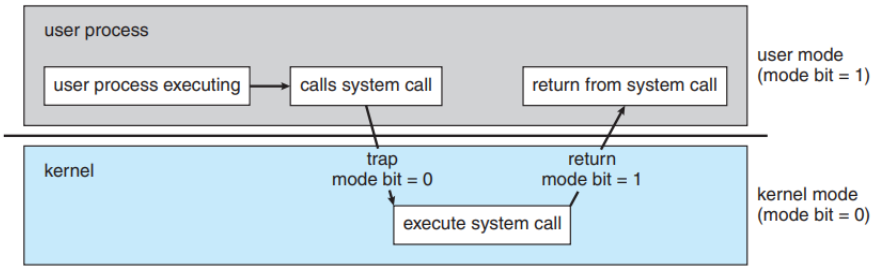



Figure 2.2: User mode to kernel mode via system call. Source [31]

is required, namely, user mode and kernel mode. The mode bit is a bit added to the computer's hardware that indicates the current mode: kernel (0) or user (1).

It is possible to distinguish between tasks performed for the operating system and those conducted on behalf of the user using the mode bit. When a user program is running on the computer, the system is in user mode. However, when a user program requests a service from the OS, the system must go from user to kernel mode to meet the request thanks to a system call.

System calls give a user program the means to ask the operating system to undertake user program tasks reserved for the operating system. Depending on the capability of the underlying processor, a system call is called in various ways.

The whole process is depicted on 2.2 to better illustrate the bit change and the clear separation that exists between user space and kernel space and the security that the system call API provides.

## 2.3 Kubernetes

In the recent years, the move towards the microservice based architecture has attracted numerous framework.

Kubernetes [15] is one, if not the vendor-agnostic cluster and container management tool that allows containerized applications to be deployed, scaled, and managed automatically with a large community and support.

Kubernetes relieves application developers of the burden of building resilience in their applications. As a result, it has grown in popularity as a platform for delivering microservice-based applications.

The Kubernetes basic architecture depicted on Figure 2.3 is made up of at least one master node (often several for redundancy) and a couple of

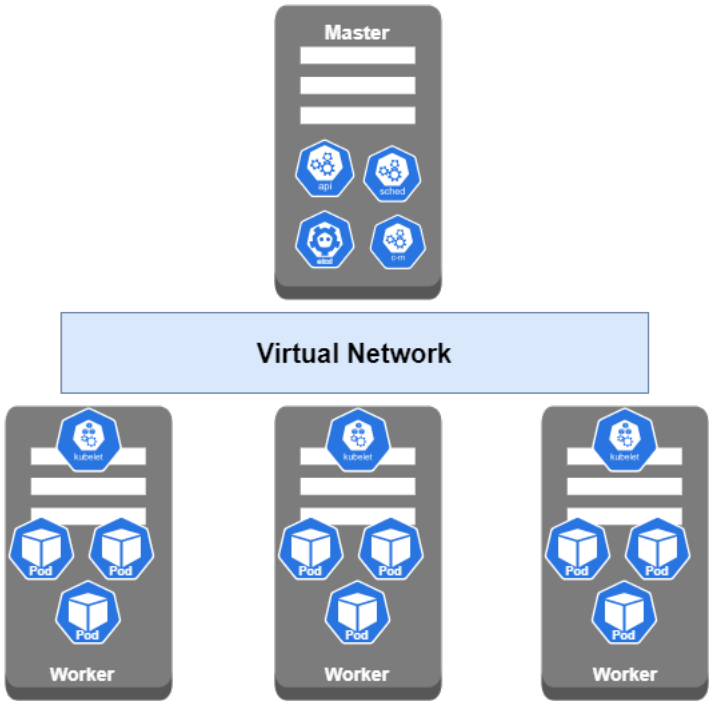



Figure 2.3: Kubernetes basic architecture

worker nodes, each of which has a kubelet process running on it. Kubelet is a Kubernetes process that allows the elements of the cluster to talk to each other, communicate with each other, and actually execute some tasks on those nodes. Each worker node has docker containers for various apps installed on it, hence the number of docker containers operating on worker nodes will vary depending on how the workload is spread.

It is important to note that several Kubernetes processes run on the master node and they are absolutely necessary for the cluster to run correctly.

The first process is an API server, which is also a container. That server is the Kubernetes cluster's entry point, therefore this is the process that different Kubernetes clients will communicate with, such as a User Interface (UI) if you're using the Kubernetes dashboard, an API if you're using scripts, and a command-line tool if you're using a Command-Line Interface (CLI).

A controller manager is another process that runs on the master node and keeps track of what is going on in the cluster, such as if anything needs to be fixed or whether a container has died and has to be restarted. A third process is the scheduler, which is in charge of scheduling containers on different nodes based on the workload and available server resources on each node. It is a smart process that determines which worker node the next container should be scheduled on based on the available resources on

those worker nodes and the load that container must meet.

The most essential component of the entire cluster is a `etcd` key/value store, which effectively keeps the current state of the Kubernetes cluster at any given moment, including all of the configuration data as well as the status data of each node and container within that node. Last but not least, a very essential component of Kubernetes that enables those worker nodes and master nodes to communicate with each other is the virtual network that spans all of the cluster's nodes. In basic terms, a virtual network combines all of the nodes in a cluster into a single powerful system with all of the resources of individual nodes.

There might also be different terms related to Kubernetes that are used in this work. We will try to demystify some of them in order to better understand once they appear.

In Kubernetes, a pod is the smallest unit configurable a user can interact with. A pod is just a container wrapper. There will be many pods on each worker node, and each pod can hold several containers. A database, for example, would be one pod, a server would be another, a java application would be yet another, and so on.

A virtual network assigns each pod its own Internet Protocol (IP) address, thereby making each pod a self-contained server with its own IP address. Internally, pods interact with each other using this IP address. During the configuration phases, the client configures or creates containers within the Kubernetes cluster, but the user only interacts with pods, which are an abstraction layer over containers and a Kubernetes component that controls the containers operating within it.

However, pods can come and go, crash, and restart, which implies they will be restarted frequently. This is where the concept of services comes into play.

A new pod is generated and given a new IP address whenever it is restarted. If the application relied on other pods, it would be annoying to have to change the IP address all the time. As a result, we will make use of another Kubernetes component known as service. It is utilized as a stand-in for those IP addresses in order to make things as consistent as feasible.

The services in front of each pod communicate with one another, and if a pod behind the service dies, it is replaced by the service that remains in place because their life cycles are not linked. In a sense, the service has two basic functions: the first is a permanent IP address and the second one is a load balancer functionality.

## 2.4 Network Attacks

This work consists partly in identifying attacks on the network. It is important to recall how these attacks work in order to better understand what information will be exchanged and stored for their detection.

### 2.4.1 TCP SYN Flooding

The Transmission Control Protocol (TCP) SYN flood attack is a Denial of Service (DoS) attack. It consists in exploiting the way the TCP three-way handshake works in order to consume a lot of resources on the targeted server and to make it overloaded to the point that it cannot handle the legitimate requests of other users.

As a reminder, the TCP three-way handshake depicted on Figure 2.4 works as follows:

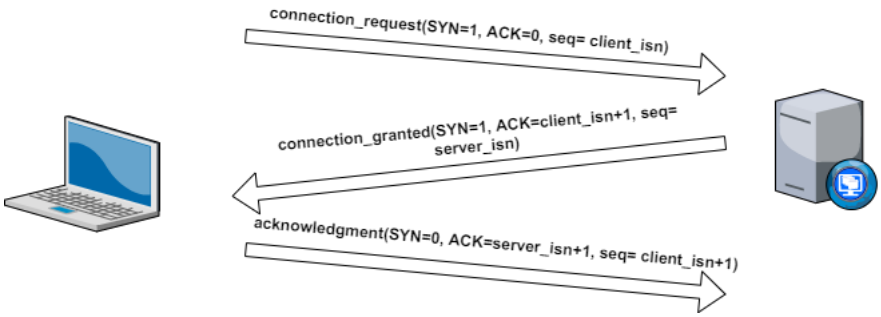



Figure 2.4: TCP three-way handshake

1. The client will choose an initial sequence number and send it in the first TCP segment with the SYN flag set to 1.

2. Once the server receives this segment it will allocate a TCP buffer and useful variables to maintain the future state of the connection. The server will respond with a TCP segment with the ACK flag set to the value of the client's sequence number incremented by 1. It also sets the SYN flag of the TCP segment to 1. It then chooses its own sequence number randomly and indicates it in the header field provided for this purpose.

3. The client that receives the SYN+ACK from the server will also allocate a TCP buffer and the necessary variables to keep the connection state. It will finish the three-way handshake by sending a last TCP segment with the ACK flag set according to the sequence number received by the server incremented by 1.

Once these steps are completed the server and the client have established a connection and can now exchange the desired data on this channel.

The TCP SYN flood consists of the attacker sending step one of the three-way handshake many times using fake IP addresses in order to force the server to allocate memory and block potential sockets if they reach a predefined maximum size, which is commonly done in practice to avoid server crashes. The server often struggles to detect this kind of attack because these requests appear to be legitimate requests and therefore will respond to each of them.

### 2.4.2 TCP SYN Scanning

Another attack that we will try to detect in this work is TCP SYN scan which also exploits the three-way handshake seen previously. Indeed, knowing which port is open on a remote server can allow an attacker to exploit the vulnerabilities of a particular service. It does not constitute an attack directly but generally indicates a more than suspicious behavior and the detection of it would allow to trigger alerts before the real attack occurs.

SYN scanning or half-open scanning consists of sending TCP segments with only the SYN flag set and observing the server's response which can be either an RST, ACK, SYN or ACK. Based on these responses the attacker can determine whether or not the port is open.

If she observes a RST or ACK or no response after a certain time then the port is probably closed or filtered and the attacker will close the connection by sending an RST to abort the process. If on the other hand a SYN+ACK is sent from the server then the attacker knows that a service is known to be running on that port and will quickly close the connection by sending an RST to avoid handling a real connection with the server.

### 2.4.3 UDP Port Scanning

The most popular User Datagram Protocol (UDP) scans are used to detect Domain Name System (DNS), SNMP, and Dynamic Host Protocol (DHCP) services. UDP scans operate by transmitting a packet that is typically empty. For each port, this can be adjusted or even set to a random payload.

The port is deemed closed if the target answers with an Internet Control Message Protocol (ICMP) unreachable error (type 3, code 3) packet. The packet is considered filtered if it responds with an ICMP unreachable error message with other codes. The port is considered open or filtered if no answer is received at all.

The difficulty with utilizing UDP for any kind of communication is that it is unreliable since it lacks the ability to create a connection or synchronize packets like TCP does. As a result, UDP scans are usually slow. Because you are waiting for a packet that may or may not arrive, and you have no means of knowing if the packet arrived at all, you may need to transmit multiple packets and then wait to ensure that a port is considered open or filtered.

# Chapter 3

# State of the Art

This chapter looks at a few projects and technologies that have some similarities to the solution we intend to bring. Some of these were used to inspire the architecture while some are proposed as improvements because not fully developed at the time of conceptualization.

## 3.1 Packet processing

### 3.1.1 XDP

High performance packet processing done in software is a real challenge as it requires a maximum time per packet to guarantee a maximum bandwidth close to what can be achieved thanks to hardware packet processing techniques.

It is important to note that the network stack currently used on our operating systems is optimised for other purposes and therefore performs too many operations that could not guarantee high throughput to be used directly.

This is the reason why before eXpress Data Path (XDP) [12] was invented there were already different techniques such as "kernel bypass" to bypass the operating system completely and pass the packet directly to the application. However, this approach, while effective, has several drawbacks, such as the difficulty of integrating this bypass into any type of system or without having to re-implement different OS functionalities in the application.

XDP aims at the same goal as "kernel bypass" but has been conceptualized differently. It works by defining a limited execution environment in the form of a virtual machine running Extended Berkeley Packet Filter

(eBPF) code. Before the kernel touches the packet data, this environment runs custom programs in kernel context. This allows for custom processing as soon as a packet is received from the hardware.

It is then the role of the kernel to ensure the safety of the custom programs by statically verifying them at load time thanks to an eBPF verifier to avoid any type of crash or corruption of the running kernel. The verifier checks for any loops not specifically bounded which could lead to possible infinite loops. It does also check for any unsafe memory accesses and will reject any program that does not meet these measures. These programs are then dynamically compiled into native machine instructions to ensure high performance.

More specifically an XDP program is run by a hook in the network device driver each time a packet arrives. The infrastructure to execute the program is contained in the kernel which means that the program is executed in the device driver.

### 3.1.2 eBPF

eBPF is an evolution of the original Berkeley Packet Filter (BPF) [20] which has been used extensively in various packet filtering applications over the last decades.

The original BPF virtual machine consisted of 2 32-bit registers and a small set of eleven instructions. This obviously led to some key restrictions.

The number of registers in eBPF is increased to eleven, and the register widths are increased to 64 bits. To improve program execution speed, the Just-in-Time (JIT) compilation stage converts the program's generic bytecode into a machine-specific instruction set. This allows eBPF programs to run as efficiently as natively compiled kernel code.

The program can write any section of the packet data, including adding or removing headers by extending or reducing the packet buffer. This enables it to handle encapsulation and decapsulation as well as modify address fields for forwarding, for example. Various kernel helper functions are available to aid with tasks such as calculating the checksum of a modified packet. The XDP program issues a final verdict for the packet at the end of processing which is accomplished by selecting one of the four return codes available.

In order to be able to store relevant data BPF has defined what is called BPF maps. These are key/value stores that are defined when loading the eBPF program and can be accessed from within the program thanks to a set of instructions. These maps have several variants, and several data structures: they can be global or unique per Central Processing Unit (CPU),

they can be simple hash tables, arrays or even radix trees. They serve several purposes: they act as a form of persistent storage as well as a means of communication between the eBPF program that modifies the state of the BPF maps and a user space program that can poll the data stored in the BPF maps.

### 3.1.3 P4

Data and control plane algorithms are used by traditional networking equipment like routers and switches to process packets. Users can customize control plane features and protocols using all sorts of tools such as CLIs, management APIs but only the manufacturer can change the underlying algorithms. SDN and, to a lesser extent, data plane programming have broken down this barrier.

Programming Protocol-independent Packet Processors (P4) [5] [26] attempts to provide programmability to the data plane. It enables the programmer to entirely define how to deal with data packets that cross the programmable data plane block.

The runtime mapping metadata is generated by a P4 compiler, allowing the control and data planes to communicate.

A P4 compiler additionally creates a target data plane executable (target prog.bin), which specifies the header formats and actions for the target device.

## 3.2 Network measurements

### 3.2.1 Sketches

According to Wikipedia [32]:

> "In computer science, streaming algorithms are algorithms for processing data streams in which the input is presented as a sequence of items and can be examined in only a few passes (typically just one). In most models, these algorithms have access to limited memory (generally logarithmic in the size of and/or the maximum value in the stream). They may also have limited processing time per item.
>
> These constraints may mean that an algorithm produces an approximate answer based on a summary or "sketch" of the data stream".

For network measurements, the data represents packets received on the network interfaces of the devices and will create a sketch about the state of the packet.

In OpenSketch [39] they present common properties of sketches such as the low memory usage. For example, one can use an array as a sketch that maintains counters incremented each time a packet destination port hash matches. It would allow then to gather knowledge about what port is commonly targeted, how often, and in what proportion. Moreover, it has proven to offer a provable trade-off between accuracy and memory which makes them appropriate for network measurements considering great hashing functions, and reasonable storage size.

Sketches may be used for a variety of reasons in network measures, including spotting heavy hitters [4] as well as making flow size distribution estimations [16], or traffic change detection [30] and many more.

Another key contributor to this field is SCREAM [25]. They suggest in their work to utilize a dynamic resource allocator that gives each job just enough resources for the traffic it observes, and dynamically adapts the resource allocation as traffic varies over time.

An important contribution is the use of Sketch-based task implementation across many switches. SCREAM use cutting-edge technology to combine sketches of varying sizes from several switches. Each task type implementation must collect sketch counts from multiple switches and present the user with measurements.

Because switches see different types of traffic, each sketch may require different sizes in order to be efficient and accurate. They carried out three measurements tasks such as heavy hitter, hierarchical heavy hitter and super source/destination which appear to be the most common and benchmarked sketches among the field.

## 3.3 Serverless Computing

### 3.3.1 Serverless

Serverless computing is an architecture in which code execution is completely controlled by a cloud provider, as opposed to the typical way of building applications and deploying them on servers. [36] [35]

It means that when deploying code, developers don't have to worry about managing, or maintaining servers. Previously, a developer had to estimate how much storage and database capacity would be required prior to deployment, which slowed down the entire process.

Users prepurchase units of capacity in an Infrastructure-as-a-Service (IaaS) cloud computing paradigm, which means you pay a public cloud provider for always-on server components to operate your apps.

It is the responsibility of the user to scale up server capacity during periods of high demand and scale down when that capacity is no longer required. Even when an app isn't being used, the cloud infrastructure is still required to support its operation.

Serverless architecture, on the other hand, allows apps to be launched only when they are needed. The public cloud provider dynamically provides resources for app code that is triggered by an event. When the code has completed its execution, the user is no longer charged. Serverless frees developers from repetitive and laborious activities associated with app scalability and server provisioning, in addition to the cost and efficiency benefits.

FaaS is available from all of the major public cloud providers. Amazon Web Services (AWS Lambda) [3], Microsoft Azure (Azure Functions) [22], Google Cloud [11], and IBM Cloud (IBM Cloud Functions) [13] are just a few of them.

### 3.3.2 Kubeless

Kubernetes was initially introduced because Kubeless is a Kubernetes-native approach of deploying and managing serverless functions using the serverless framework. The functions are self-contained and can be thought of as Kubernetes-managed microservices. They are pieces of code that the developer writes and subsequently uploads to the Kubernetes cluster.

Kubeless has set up a system for triggering functions for them to run. These triggers are defined by the framework as events such as Hypertext Transfer Protocol (HTTP) endpoint, Kafka queue message, timers.

### 3.3.3 Unikernel

In a serverless architecture, code snippets executed in response to an event are allocated an execution environment. This environment must be able to be created quickly, be very light, have the basic functionalities and libraries to execute the code in question and be completely freed once the execution is complete. It is also important to take into consideration the isolation, as the cloud provider wants to make sure that the execution of a function from one client cannot interfere with the execution of a function from another client on the same host or corrupt the host machine itself.

It is therefore very popular to use lightweight virtual machines or

containers or even unikernels that are even lighter and have a very low boot time.

Unikernels [19] are built using library operating systems, in which the OS kernel is tightly linked with user applications as separate software appliances.

They may be used in conjunction with standard hypervisors as well as lighter virtualization methods. Unikernels are lightweight and have great isolation which makes them a very good match for the FaaS paradigm. They are also as quick as processes and as safe as virtual machines.

### 3.3.4 Unikernel FaaS

In contrast to the current FaaS model, in which large, stateless functions are run in heavy sandboxes, Unikernel-as-a-Function (UaaF) [33] breaks down the main functionality into a sequence of low-level function invocations, each of which runs in its own unikernel (lightweight sandbox). UaaF is made up of two kinds of functions: session and library functions. A session is a proxy function that specifies the application's skeleton. When the application starts to run, it specifies which library functions to call. Library functions are pre-defined routines that application developers upload to the cloud and may be called and used by different apps.

When a user requests that a serverless application be invoked, the cloud provider constructs a session function for the application and ties it to the library functions given in the user's configuration file.

UaaF most important takeaway is that it allows to create more sophisticated serverless applications with inter-function communication, a feature that would make possible one function in one unikernel to call and wait for the return value of another function in another unikernel with little performance and startup time overhead.

In order to do that they have to minimize the performance penalty of function invocation across unikernels. UaaF makes use of VMFUNC, a new virtualization capability on Intel CPUs. A unikernel running as a guest OS in a virtual machine can use VMFUNC to call a function in another VM without exiting the virtual system. UaaF uses VMFUNC to switch the Extend Page Table Pointer between two unikernels, allowing one unikernel to access virtual memory and execute functions in another unikernel making this technology a real contender in the field.

## 3.4 In-Band Network Telemetry

Data plane telemetry and switch internal state information can be collected at line rate using In-Band Network Telemetry (INT), an unique architecture. Custom metadata is used to read the telemetry headers and identify which telemetry items to add utilizing programmable data planes and P4 programming language.

In [34] by using an SDN controller, the control plane can set per-flow event detection mechanisms, such as a switch's role (such as source or sink) and detection algorithm (see Figure 3.1). P4 exposes a controller API that communicates this configuration to the switch. Event detection parameters can be specified by the SDN controller through this configuration API, allowing for the detection of many different sorts of events at the sink switches (e.g. queue information, burst and so on).

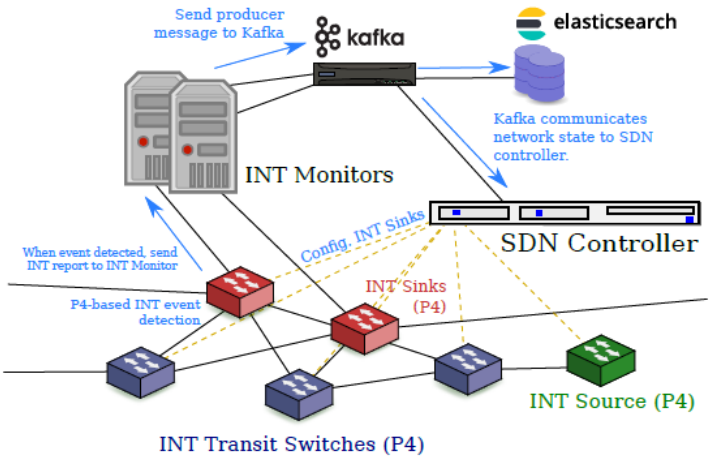



Figure 3.1: Overview of the INT monitoring. Source [34]

Each transit switch along the path parses the instruction and puts on the INT metadata in the telemetry header (such as queue occupancy) specified by the instruction.

Upon receiving a packet containing a telemetry header, an INT sink (collector) strips all telemetry headers and uses the event detection mechanism provided in the control plane.

Telemetry reports from the sink switches are then sent to INT monitors if the event criteria are met. The sink switches delete telemetry information if no event is detected, lowering the strain on the stream processor. INT monitors use AF_XDP, which streams remaining telemetry items to a distributed Kafka cluster.

This collected data retained in Kafka topics may be used by the SDN

controller to modify and fine-tune the threshold and algorithm settings on a per-flow basis, boosting report resolution for relevant traffic while decreasing it for background traffic, for example. Kafka also sends INT event reports to an Elastic Search stack for further analytic processing and visualization.

They utilize numerous traces from real-world data center workloads to assess the impact of various event detection algorithms and threshold settings on the event detection ratio and stream processor load. The article demonstrates that their system can handle hundreds of millions of telemetry headers every second.

In this study, they use a mechanism that adds additional headers to network packets that carry important information, which is then read and decapsulated by switches at certain network locations. The inclusion of more data is indeed interesting if we are interested in delays, queue sizes, and congestion rather than network attacks or events that can be inferred directly from the original packets without the addition of new data. They presume an SDN network and chose not to overburden this controller with INT Monitoring functions by establishing entities dedicated just to this task, which is promising and will be used as inspiration.

# Chapter 4

# Architecture

This chapter will display the overall architecture that can be seen on Figure 4.1. We introduce mainly three big components called modules that interact with each other two by two vertically which are : packet processing module, lambda server module, lambda functions module.

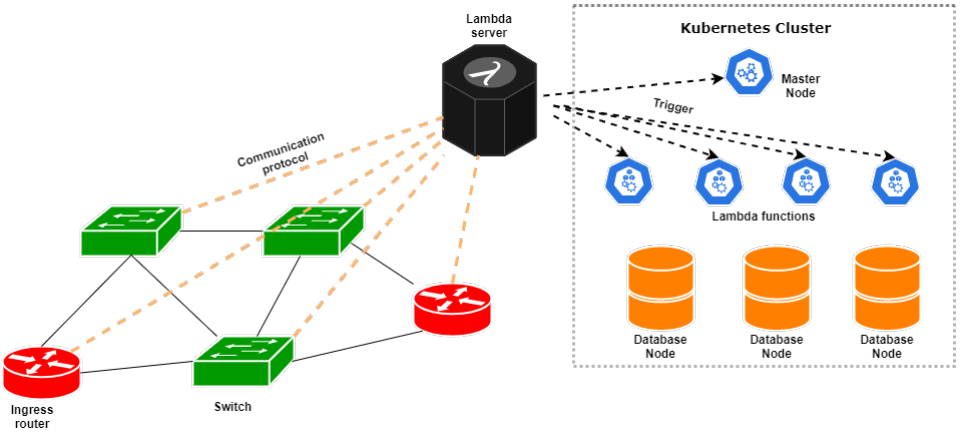



Figure 4.1: Overview of the architecture

## 4.1 Approach

When you are tasked to design a complete solution it is important to understand what needs to be designed and more specifically divide the architecture into structural layers from bottom to top where the top represents the application using the data collected and gathered in the bottom layers.

We will therefore introduce in a first module the very low level technologies linked to the programming of switches and routers which allow us to efficiently retrieve information of each packet directly arriving

at the Network Interface Card (NIC).

We will discuss a second module which consists of exchanging this information with either other switches and routers or a dedicated server present in the network topology. Protocols might be included to ensure the exchanges and will also be described as well as the challenges they bring.

The third module will cover the connection that exists between the controller namely the lambda server, and the serverless architecture that we will use in order to provide a scalable solution to execute our lambda functions in response to network events that can vary enormously depending on the assumptions about the size of the network at hand. This part is also dedicated to the execution environment of these lambda functions with their advantages and disadvantages as well as the persistent storage technology used.

## 4.2 Packet Processing

During the state of the art review several packet processing concepts were introduced. Indeed the goal is to recover information on packets and store them in an efficient way while ensuring a sufficient bandwidth not to impact the whole network in which we will implement our solution.

P4 is a serious choice when considering fast packet processing. However, it still has the disadvantage of requiring some hardware compatibility. "P4 compliant" switches and routers are not the vast majority of devices available and even less so at the price level if you want to consider a network like the one of the University of Liege or a large company for example.

That's why the choice has been made for a solution that is easy to develop, to test and that can be portable on almost any type of device available on the market.

Rather than transferring networking device management out of the kernel, the XDP technique [12] chooses that performance-sensitive packet processing tasks are relocated inside the kernel and executed before the operating system networking stack starts processing. This maintains the benefit of removing the kernel-user space boundary between networking hardware and packet processing code while keeping the kernel in control of the hardware, preserving the management interface and operating system security assurances. The use of a virtual execution environment, which guarantees that loaded applications will neither harm or crash the kernel, is the essential invention that enables this.

As it can be noted in Figure 4.2 the XDP program is executed before any allocation in the kernel (sk_buff allocation) and even before being

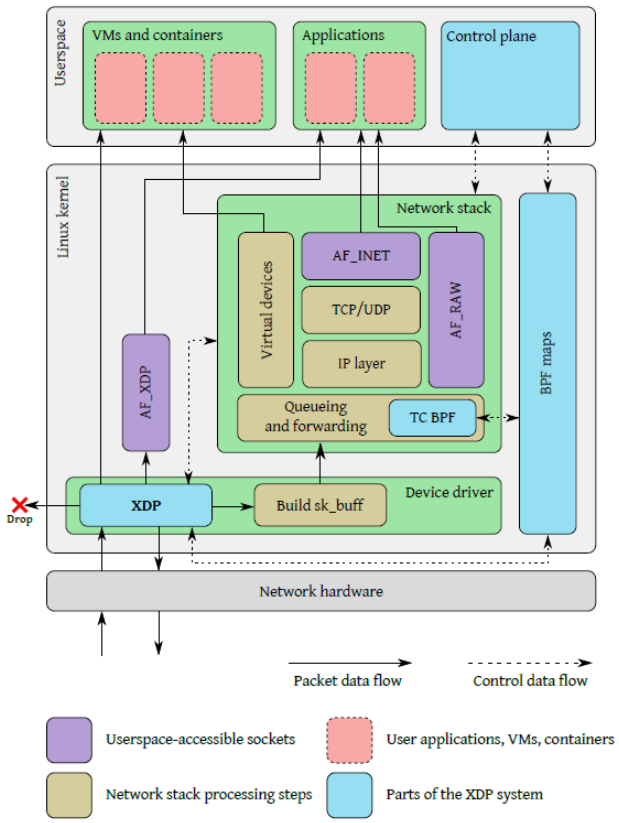



Figure 4.2: XDP's integration with the Linux network stack. On packet arrival, before touching the packet data, the device driver executes an eBPF program in the main XDP hook. Source [12]

passed to the network stack. This XDP hook at the device driver level allows to execute the eBPF program which is the responsibility of the programmer and have been created for this work. This eBPF program can store, drop, forward on the same interface or another one, the packet received on the interface it came from. It can also decide to pass the packet to the network stack to be processed in user space by an application that needs it and follow a more classical path that we find if XDP was not in the picture.

## 4.3 Storage

It is clear that to be able to interpret the behavior of the network in real time we need real time data. We must therefore consider a means of storage and especially a selective choice of data packets to store and where to store them.

We are going to introduce the different components that intervene at

the packet processing level see Figure 4.3. We will have three components: The first one will be the XDP hook which will allow the execution of the BPF program. The second one will be a user space process that will run permanently in order to poll the hash tables that are the third component, the BPF maps. This third component although introduced previously, will allow us to store for a key an associated value which will not be a simple value but information such as flows composed of source IP address, destination IP address, source and destination ports, sizes of the flow, the number of packet it is made of and so on. We will therefore map keys to structures representing this information.

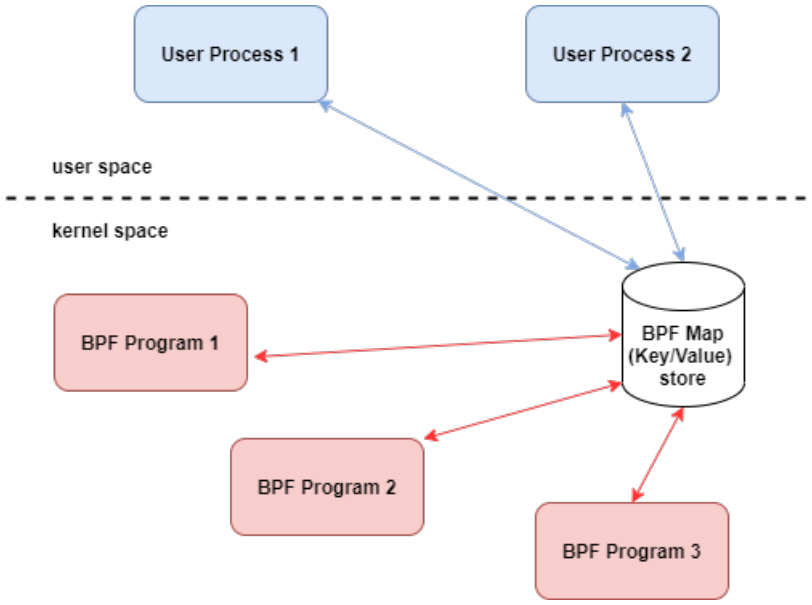



Figure 4.3: Overview of the components for packet processing and temporary storage

The kernel allocates these BPF maps. As a result, the user space program can only access them through system calls. However, because the eBPF program runs in kernel space, it does not require the usage of a system call.

In this work we take inspiration from sketches in a much more simplistic manner. Indeed, sketches are very often compact ways of representing information in a data stream. They generally use very compact tables and arrays combined with hashing functions.

Our idea is to hash fields of the header of the packet thanks to hashing functions. Then, to map this hashed value to an index of an array and to increment different kinds of counters such as the number of bytes, the number of packets, the number of SYN flags or any type of statistics the user may want to use to understand at best what is going on in the network.

## 4.4 Controller

The next critical step is to collect or aggregate this data. There are various ways; in fact, just one of multiple options was chosen.

When using sketches SCREAM [25] also designed an algorithm that allows the different switches to communicate their sketches with each other so that at any time each entity can have a global view of what is happening in the network. We decided not to go along this path mainly because first it consumes more bandwidth even if you take into consideration that they are using a compression algorithm. Secondly, because we wanted to be able to transpose our solution into an SDN network with a controller. This controller would have a double role since it would be used on the one hand to control the control plane and the routing and on the other hand as a drop point of the various metrics collected by the switches and routers of the network that they would communicate to it thanks to a protocol to be defined.

The role of the controller will be reduced to trigger the lambda functions according to the reception of events sent by the switches themselves. It is the role of the user space program on each switch that polls the data from the BPF maps to create events with a specific format and to send them to the controller.

It is clear that the controller could have a more active role in the process. Indeed, we could assign it the task of collecting all this data, storing it, filtering it and then only triggering the lambdas functions. However, because an SDN network may have many SDN controllers, a synchronization system should have been implemented to ensure that no data was lost if one of them stopped working. Protocols exist, but the idea was not to overcomplicate each layer, which would make the entire process more complicated. As a result, they will only act as a triggering engine for lambda functions. Another point is that their principal purpose is to make routing decisions, and calculations. These can be very demanding in terms of computing power and we did not want to overload it with too much additional work.

As our solution is also applicable for any network that is not SDN, we will use the word "Lambda server" to designate this relay process in direct communication with the switches and routers of the network topology.

## 4.5 Protocol

In order to communicate the events from the switch to the lambda server, it was necessary to establish a protocol. Indeed, the user space program of the switch has to send the data of the BPF maps formatted in a precise way

so that the server can read them, understand them, reject them if necessary or accept them and trigger the lambda functions in a third time. It works as a carousel, i.e. it will browse the modified BPF maps entries, format them as events, gather them and send them to the server every two seconds, which guarantees a certain refresh rate that aims to channel the number of exchanges between the switch and the server. A refresh rate that is too high will lead to a lot of packets being sent containing very few events and a refresh rate that is too low will lead to a mismatch between what is actually happening in the network and what the server will perceive and it might not be real time information.

The choice of the transport protocol was turned to UDP. The first reason being that we had considered the fact that some events to be sent to the lambda server would not require any acknowledgement from it, such as for example some counters that would be modified frequently and therefore to be resent according to the defined period no matter what. On the other hand, events for which the switch expects an acknowledgement but has yet to receive one will be resent during the next period. This causes network overload the exact reason we wanted to avoid TCP in the first place. However, it is important to remember the assumption we made about the network in which we would deploy our solution. It is a Local Area Network (LAN) with very short links, very few hops between the ingress and egress points which will naturally decrease the drop rate of packets exchanged between the switches and the lambda server.

This protocol is quite simple and inspired by a stop-and-wait type of protocol for the acknowledgment algorithm. The reason for not choosing a very complex protocol is also a strategic choice that was made with full knowledge of the facts. Indeed, the reasoning throughout the development was to not overload the switches and routers with too much logic at the level of the protocol and exchanges to ensure their main task was not compromised. The goal was to avoid consuming high percentages of their CPU.

The protocol works as follows: the switch will create two types of UDP payload either composed of events that all require an acknowledgment, or events that do not require an acknowledgment. This payload will be called a transaction and will therefore be composed of several events grouped together. If an acknowledgement is expected from the lambda server, then the client (switch) will wait a predefined time after which it will re-send the transaction if it was waiting for an acknowledgement.

Upon receipt of an acknowledgement the client will directly resend a new transaction unless no transaction is possible, for example if there were no events in the queue.

Since some transactions do not require an acknowledgement from the lambda server, it was decided to create another policy. According to the

same timer if events are available they are sent and the timer is reset which means that transactions are only sent to the lambda server at regular interval but does not expect an acknowledgment and will never retransmit transactions even if they are lost.

On the server side the logic is relatively simple: either it responds by sending an ACK, or it does not respond because the transaction received does not require an explicit response.

The protocol will be further explained in more detail when we discuss its implementation in section 5.2.

## 4.6 Lambda functions execution environment

The execution environment of the lambda functions was part of the requirements of this work. It is often a container, a VM or a unikernel according to the latest technological advances. The design of a unikernel for this kind of task is promising but very laborious and would deserve a complete focus on the adaptation of these to this solution in order to compare it to other technologies. It is also important to note that a framework is required to easily deploy and manage the lambda functions and their execution on the hosts, and all the issues that this entails.

Thus the choice was very targeted on an opensource framework that we could test and that would be among the most promising out of the ones available to us.

#### Choice & Motivations

There are several studies that try to compare frameworks in different scenarios. We have selected two of them to make our choice on the technology to use.

In [23] they look at opensource serveless frameworks like Kubeless, OpenFaas, OpenWhisk, and Fission. To ensure isolation, all of the frameworks studied run each serverless function in its own Docker container.

A container orchestrator is used by OpenFaaS, Kubeless, and Fission to handle the networking and lifecycle of the containers.

They chose Kubernetes as the orchestrator in their experiment since it is the only one supported by all of the frameworks mentioned above.

Under various levels of concurrent queries, they first examined the

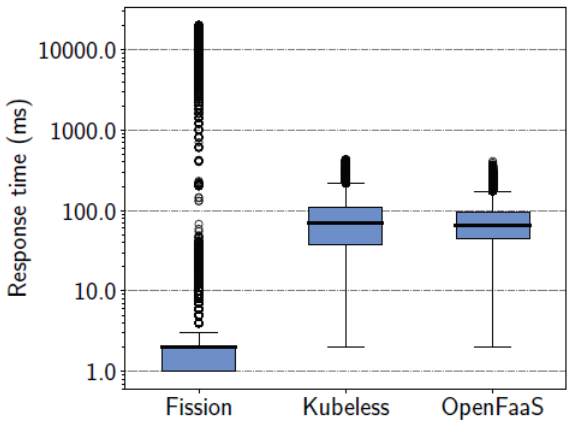



Figure 4.4: Response time with one function replica and 100 concurrent requests. Source [23]

| Framework | Repl. | Number of concurrent users | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 5 | 10 | 20 | 50 | 100 |
| **Kubeless** | **1** | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| | **25** | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| | **50** | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| **Fission** | **1** | 100.00 | 99.90 | 99.84 | 99.78 | 99.54 | 99.32 |
| | **25** | 100.00 | 99.89 | 99.85 | 99.77 | 99.48 | 99.19 |
| | **50** | 100.00 | 99.88 | 99.81 | 99.79 | 99.61 | 99.31 |
| **OpenFaaS** | **1** | 99.95 | 99.99 | 99.91 | 99.58 | 98.73 | 98.27 |
| | **25** | 100.00 | 100.00 | 99.92 | 99.67 | 97.76 | 96.04 |
| | **50** | 100.00 | 100.00 | 99.93 | 99.61 | 97.48 | 96.52 |

Figure 4.5: Success ratio (in %) of all requests for different serverless frameworks. Source [23]

average response time and the ratio of correctly received responses. As it can be seen on Figure 4.4 Fission has many outliers, and it suffers from high workloads regardless of the number of function replicas. This is attributed to Fission's router component, which routes all incoming HTTP requests to the correct function. As the workload grows, this component becomes a bottleneck. Kubeless, on the other hand, relies as much as possible on native Kubernetes components: it uses the Kubernetes Ingress controller to route requests and balance the load.

They then go a step further and look at the same parameters but for a number of other function replicas variation. The results are depicted on Figure 4.5.

They discovered that Kubeless had the best performance, with a success rate of 100 percent across all experiments, whereas Fission maintains a success rate of above 99 percent even at larger levels of concurrency. When there are 50 or more concurrent requests, the success rate of OpenFaaS drops to 98 percent or below, according to the researchers.

The influence of auto-scaling on response time and the ratio of successfully received responses is also highly important in serverless applications, which is why they looked at how the impact of auto-scaling on the response time and the ratio of successfully received responses varied. The Kubernetes orchestrator handles scaling for all frameworks. They did note, however, that the ratio of successful responses and the response time distribution differed from one framework to the next. In comparison to Fission, which had a deceptive 98.11 percent success rate across all experiments, Kubeless and OpenFaaS had a high 100 percent success rate.

In the second study [28] they also compare the same Opensource serverless framework but adding a new one namely, Knative.

They run a similar testing environment except that this time the lambda functions will be triggered by Internet of Things (IoT) devices since they are analysing this in the deployment of IoT devices on the edge.

Despite the fact that all frameworks delegate scaling decisions to the Kubernetes Horizontal Pod Autoscaler (HPA) feature, they discovered that the values obtained for the evaluated metrics differ significantly. For example, Apache OpenWhisk has the worst overall performance across all metrics of any framework analyzed. One service/device has a similar success rate to the other three frameworks, however it lowers dramatically as the load increases.

There is no discernible difference in success rate or throughput between Kubeless, OpenFaas, and Knative. However, when the demand grows, they noticed that Kubeless scales better in terms of reaction time. As the number of services per IoT devices grows, Kubeless maintains an average response time of 12.57 to 13.79 milliseconds, whereas OpenFaaS response time increases from 96.92 to 106.82 milliseconds and Knative response time climbs from 86.27 to 253.66 milliseconds on average.

In conclusion based on those two studies and given an incredible close gap in terms of features, Kubeless seems to outperform the other frameworks across the different scenarios presented in those papers. This directed our choice towards Kubeless using Kubernetes as the orchestrator and thus leaving Kubernetes HPA automatically scaling the number of pods in a replication controller, deployment, replica set or stateful set based on observed CPU utilization in real time.

## 4.7 Long-term storage for lambda functions

Since the beginning of the conception of the architecture, there was a need for storing in a more permanent way the data initially collected by the

switches and routers. Indeed, each time a lambda function is triggered, its code can be made up of an access to a database, distributed or not, and to store the values of these counters. It can increment them and check that they do not exceed certain threshold each time they are called. It is therefore obvious that to exploit the real-time information to the maximum we need a more persistent storage, whether it is useful for very reactive lambdas functions or for offline analyses of these data.

A Distributed Hash Table (DHT) is a decentralized data storage that uses key-value pairs to look up data. In a distributed hash table, each node is in charge of a set of keys and their values. The key is a unique identifier for the data value it refers to, which is generated by passing the value through a hashing process. Any type of data can be used as data values. Because distributed hash tables are decentralized, all nodes in the system work together without the need for centralized coordination. Generally, the data is replicated over numerous nodes which makes the whole system fault-tolerant. This form of storage was initially considered for resiliency purposes and also allowed to be very flexible in term of its implementation which would have allowed to have a distributed database designed specifically for the kind of operations and interactions with the lambda functions that were required.

However, having already a cluster under Kubernetes at our disposal and our serverless infrastructure with Kubeless it was natural to find a workable solution within Kubernetes to have something consistent. It is not as easy to deploy a distributed database in time and quality with Kubernetes, especially since there are other options available to us. Indeed, the strength of Kubernetes lies in its ability to scale any resource according to demand in real time thanks to the Kubernetes HPA. So we chose the option of deploying a MongoDB database within the cluster with a master on which the lambdas functions would come to read and write and two replicas synchronized on the master for questions of redundancy if this one had to give way. It is clear that having only 3 replicas is a starting point and that Kubernetes would allow us to deploy more replicas if needed.

# Chapter 5

# Proof of Concept

In order to prove the feasibility of our framework, we decided to develop a Proof of Concept (PoC). The code is available on a Github repository at https://github.com/francoisleduc/TFE with additional details and installation steps.

## 5.1 Packet processing

The technology we will use for the packet processing phase is the XDP hook system mentioned in section 3 alongside BPF and eBPF programs, terms that are used interchangeably in this work.

When learning a very specific technology with its own concepts to grasp, it is common to have to learn by yourself and to exchange more formally with experienced people. A first pointer to a Github directory [2] with tutorials, explanations, official papers, learning content was given to me by my advisor. They offer a multitude of examples from which I was inspired for the implementation further explained in the following sections. Three of them must be mentioned in particular such as the "XDP Hands-On Tutorial" [37] github repository, the "BPF and XDP Reference Guide" [6] and "Andrii Nakryiko's Blog" [1], a blog written by a Facebook developer that has also developed libbpf and is currently working on BPF.

However, even if these pointers were very useful and full of concrete examples, this was not a direct solution to my problem and I had to integrate my own ideas. Some libraries and dependencies were reused but is not the core of my work.

### 5.1.1 Dependencies

Long time before now, when it came to developing BPF applications that needed looking into the kernel, BPF Compiler Collection (BCC) [7] was the framework of choice. BCC had a built-in Clang [8] compiler that could build BPF code in runtime and adapt it to a certain host kernel. This was the only method to write a maintainable BPF program that dealt with kernel internals that were constantly changing.

Nowadays libbpf [18] is the reference as it understands the structure of BPF code and properly load everything into the kernel. The fact that it is not runtime compiled avoids BCC compilation-time errors that were sometimes detected only during the runtime. BCC also required dependencies on some kernel headers per target host and sincerely reduced the lack of portability. libbpf also uses the Clang compiler to compile once the BPF code and ensure portability. It is also capable to load the BPF program thanks to its loading functionalities as well as a means of interaction between the BPF program and user space from user space programs using libbpf APIs.

### 5.1.2 Kernel program implementation

As the code was designed in an incremental way, it seems natural to explain it in the same manner in order to be able to explain the built-in functionalities as well as the associated explanations without being lost in a mass of terms.

The following eBPF program will be executed when a packet arrives on the interface to which an XDP-hook has been attached.

This first program (see Listing 1) is fairly basic, and it only serves as a primer for learning and conceptualizing the many eBPF programs implemented throughout this project.

The headers contained in the package are the first things that must be looked at. Indeed, the first `linux/bpf.h` contains several fundamental BPF-related types and constants required for using the kernel-side BPF APIs and is needed for the next header included.

The second `bpf/bpf_helpers.h` file will allow us to use a whole panel of functions to interact with bpf objects, such as BPF maps but also access and set other structures as well as purely utilitarian functions to handle clocks, timers, and debugging functions like `bpf_trace_printk()`, which will allow you to write in a specific debug file.

The BPF program and the kernel both require and enforce the license variable, which is located at the end of the file. It's also worth noting that

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

#define bpf_debug(fmt, ...)                             \
    ({                                                  \
        char ____fmt[] = fmt;                           \
        bpf_trace_printk(____fmt, sizeof(____fmt),      \
            ##__VA_ARGS__);                             \
    })

int count = 0;

SEC("xdp_flow")
int process_packet(struct xdp_md *ctx)
{
        count++;
        bpf_debug("BPF program triggered successfully\n");
        return XDP_PASS;
}

char _license[] SEC("license") = "GPL";
```

Listing 1: XDP Program basic

BPF programs without the compatible GPL license variable will be missing out on several capabilities.

Then we have a global variable called `count` and set to 0 in this example. This variable is global and will act as such in concept of the C language. These global variables can be seen as state variables of the eBPF program. They also have an interesting effect because they can be modified by the user space program as well as other eBPF programs defined in the same file. They are therefore also used as a means of communication but especially as a means of configuration between the two layers. These variables are often used to store settings, low overhead stats, and so on.

The core line of the BPF program listed on Listing 1 is the call to `bpf_debug()` which is a macro that actually calls the `bpf_trace_printk()` function. It is the BPF equivalent of a `fprintf('')` call if we were using standard C. This output however, is located in a specific file at `/sys/kernel/debug/tracing/trace_pipe` that you can only access with root privileges. This kind of printing is obviously not for production but gives such an insight of what is going right or wrong in the BPF program and is one of the very few ways to debug your code and log events.

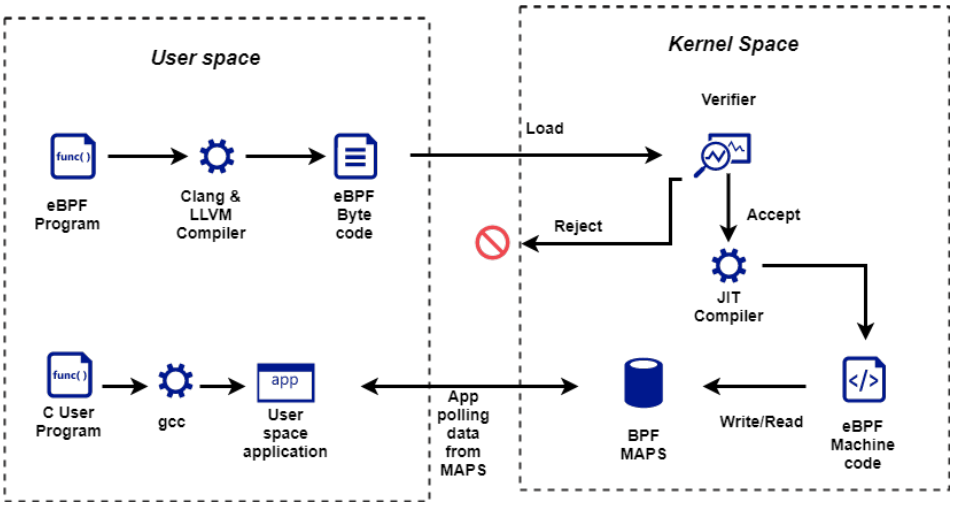



Figure 5.1: Compilation process

### Data Structure

```
struct bpf_map_def SEC("maps") flow_map = {
        .type = BPF_MAP_TYPE_ARRAY,
        .key_size = sizeof(__u32),
        .value_size = sizeof(struct flow_meta),
        .max_entries = MAX_MAP_SIZE,
};
```

Listing 2: BPF_MAP definition code example

BPF map is a BPF data abstraction concept. In the BPF universe, many different generic structures such as arrays, hash tables, and ring buffers are abstracted and represented as BPF maps. It is the most common way for storing data and allowing access to read, write, and delete elements via a key/value system. Because both the user space program and the eBPF program can access this data structure, these maps will be a second means for them to communicate and exchange data between themselves.

To create a BPF map, we must define a global `struct bpf_map_def` in the BPF program with a special section `SEC("maps")` as shown in the BPF map definition example in Listing 2.

### Compilation

Several compilations occur in the eBPF pipeline. The Figure 5.1 was included to bring clarity to the different pieces and to also bring an overview of what is in user space and in kernel space.

BPF applications are typically written in C, LLVM-compiled [17] into object/ELF files, processed by user space BPF and ELF loaders (such as iproute2 or BCC or libbpf), and pushed into the kernel through the BPF system call API. The kernel then checks thanks to the verifier and JITs the BPF instructions, producing a new file descriptor for the program to attach to a subsystem. If supported, the BPF software might be further offloaded to hardware (e.g. NIC).

The eBPF JIT compiler is included in the kernel for the most prevalent 64-bit architectures such `x86_64`, `arm64`, and `ppc64`, as well as the 32-bit versions `arm`, and `x86_32`. Because it reduces the time per instruction compared to when BPF code is parsed using an interpreter, this JIT compiler provides significantly faster BPF code execution. This is made feasible by a mapping of the program's instructions to the native instructions of the underlying architecture.

**Examples**

For this master thesis I have developed several BPF programs with quite similar functionalities where only the stored data in the BPF maps changes.

First, the programs proceed to a progressive de-encapsulation of all the different headers and a rigorous parsing by protocol by checking at each step if the accesses are well within the limits of the packet, and that each access is authorized in order not to be rejected by the eBPF verifier which will prohibit the loading of the program on the interface once JIT compiled.

Each header de-encapsulation has its own importance since the idea is to store information about the flows consisting of a source IP address, a destination IP address, the source and destination ports and the transport protocol used.

These values obtained through meticulous parsing will be hashed in order to obtain an index in the BPF map used for the program that we use to store a value represented by a group of data such as the size of the flow, the number of packet it is made of and the 5-tuple flow itself. The index obtained being in a range starting from 0 to the maximum size of the BPF map introduces a risk of collision. This collision rate is studied in section 6.1 as it will impact the accuracy of the network measurements because some flows might get deleted or ignored in such a case.

The hashing functions used are `jhash_3words()` and `jhash2()`. Few concatenations of different variables representing the flow had to be made in order to respect the definitions of these two functions and to include all the parameters defining a flow.

In order to access this element we will use one of the many functions discussed earlier available from the `bpf_headers.h` to access an element of the BPF map which is `bpf_map_lookup_elem()`. It's a straightforward system call wrapper that works with the map File Descriptor (FD). The system call retrieves the value from the key and puts it in the memory location provided by the value pointer.

Then, after checking that the access was done without error, several possibilities are open to us depending on what is already present in the BPF map.

```
hash = hash_tuples(&f, is_ip6);
key = hash % MAX_MAP_SIZE;

v = bpf_map_lookup_elem(&flow_map, &key);
if(!v)
{
      return XDP_PASS;
}

__sync_fetch_and_add(&v->count, 1);
__sync_fetch_and_add(&v->bytes, (data_end - data))
```

Listing 3: BPF_MAP access and basic value modifications

The implemented logic is as follows: if the entry is brand new it means that no flow had been mapped on this key yet and therefore the value is instantiated with the values collected from the packet. If the input is already in use and the flow corresponds to the received packet information then the statistics are incremented by their respective values. If the input is already in use and the flow does not match the received packet then we are in a situation where we would like to use this key to initiate a new flow while there is already one stored for that same key. However, we will first check if the flow in the table has been recently modified, in which case we would keep the current entry and just dump the new flow defined by the received packet. If not recently modified, the current flow is overwritten by the new one defined by the packet parsed.

This will obviously not guarantee 100 percent accuracy because some flows will be forgotten after a while, others will never be taken into account or only partially. It is a compromise that was made in the implementation of this collection of statistics by flow that we considered good enough considering the exploitation that we were going to make out of it.

On the listing 3 you can see an example of instruction sequence and especially the use of atomic operations on BPF map values. Indeed, the call to `__sync_fetch_and_add()` is often used. It provides a sequential access

between potentially different processors for a same data and increment it by the specified value.

### 5.1.3 User space program implementation

Once the kernel program is implemented we have to think about how to extract this data quite frequently given the implementation seen in the previous section.

The user space program is a program written in C but it could have been written in another language as there were several ways to access the maps as we saw with either BCC or libbpf. Since BCC is also supported on python it was just as valid to switch to another programming language to produce a similar result.

The user program is separated into two threads. Indeed, the first one will have for role to poll the data of the BPF map every two seconds approximately. Depending on the eBPF program at the kernel level, and especially on the map settings, this thread will have to create a series of events with a format that will be discussed in a few moments. These events, made of the statistics collected in the maps, will be assigned to two queues and these queues will then be emptied as they are transmitted to the lambda server by the second thread. The second thread is named "communication thread" to have an explicit notion of its unique role which is to send all collected events to the lambda server.

#### Polling thread

The data-structure utilized for the value record is unknown to the BPF map; it just knows the size. As a result, it is up to the two sides (user space and eBPF kernel side program ) to keep the content and structure of value in sync.

This verification can be seen on Listing 4. Once this is done, we are sure that the map setting matches and the values can be accessed by their corresponding key.

In this example, each value in the data structure is of type `struct flow_meta` and has a field that contains the time of last update. It allows the polling thread to determine whether or not an event should be created from this entry based on the update time and the time it is retrieved from the map. We chose an acceptance window that was smaller or equal to the polling frequency, such that only values that had changed since the last evaluation were the only considered. This prevents retransmission of irrelevant messages and avoids utilizing more bandwidth than truly necessary.

```c
// Look for map file, file descriptor
stats_map_fd = find_map_fd(bpf_obj, "flow_map");
if (stats_map_fd < 0)
{ // could not find map fd
        xdp_link_detach(cfg.ifindex, cfg.xdp_flags, 0);
        return EXIT_FAIL_BPF;
}

/* check map info, e.g. value is expected size */
map_expect.key_size    = sizeof(__u32);
map_expect.value_size  = sizeof(struct flow_meta);
map_expect.max_entries = MAX_MAP_SIZE;
err = __check_map_fd_info(stats_map_fd, &info, &map_expect);
if (err)
{
        fprintf(stderr, "ERR: map via FD not compatible\n");
        return err;
}
```

Listing 4: BPF_MAP compatibility check from userspace program

#### Communication thread

The role of the communication thread has already been introduced. It consists mainly in acting as a sender in the designed event exchange protocol. It is therefore equivalent to explaining the protocol implementation to understand how it works. This is done in the following section (see 5.2.2).

## 5.2 Protocol specifications

### 5.2.1 Event format

The payload of the UDP datagram, also called a transaction in the context of this work, has a specified format that must be followed in order to be correctly read by the receiving server.


Figure 5.2: Transaction Header

This one is made up of a fixed-size header represented in red on Figure 5.2 that the protocol requires in order to distinguish between defective and good datagrams. This means that regardless of how many events are put in it, the header will always be present and will not change. This header is 10 bytes in length and is composed of 4 elements.

On the first byte, a version of the protocol is encoded, which may enable for backward compatibility if the protocol is drastically updated or modified. This is followed by a 4-byte identifier that is unique to each sending switch or router, allowing us to have statistics per machine and thus better visualize which network element is sending the most data, or to have a unique configuration per machine that would be mapped onto the identifier if we wanted to implement it. The header's third field is a sequence number. Remember that some transactions will only be made if the situation allows them, and only for events that require a server acknowledgement. This means that the server will utilize the transmitted sequence numbers to detect if a packet has been ignored or lost along the way, and will compel a retransmission of those packets. The number of events to expect to parse in the transaction is the last element in the header and is encoded on a single byte.

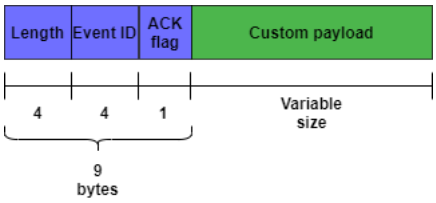



Figure 5.3: Event Header

We can see in Figure 5.3 that each event has a fixed size header and a variable size payload. The length of the event header is the first field and is 4-byte long. Indeed, because a transaction can be made up of numerous events, each of which is identified by the general header field, several of them can be encoded one after the other and are not always of the same length. Instead of using the length field, a delimiter might have been used between each event to indicate when one concluded and another began. The second header field unique to an event is its identifier, which is also encoded on 4 bytes just like the length field. As a reminder, we decided to divide the actions to be conducted into categories based on the data collected in the form of network events. This may be a simple update of a flow or an alert of new routing or a link that is operating better, and each would have a distinct payload format because it would be made up of different data. If the client wants to ensure that the server receives the event, there is an acknowledgement flag encoded on a single byte that must be set to 1, otherwise it is set to 0. This event header is 9 bytes long, and it is followed by the payload. The payload is formatted to match the event identifier. As a result, it is critical that both the client and the server understand how to parse the payload of each event for a certain

event identifier.

### 5.2.2 Exchange protocol

The client, which in this work is either a switch or a router, will handle two queues: one for events that will lead to transaction requiring an ACK from the server and another for events that will not require a response from the server.

The decision to proceed with two queues was made for the following reasons: to avoid superfluous ACK exchanges on the network and to always strive to minimize the impact of packets in transit that are not allocated to monitoring. As a result, the client will have two timers for the two event queues. The first is to send transactions that are only made up of events that do not require an ACK to be received. The second will use a stop-and-wait approach, sending the transaction first and then waiting for an acknowledgement from the remote server. If no response is received within the time limit set by the programmer, the timer will cause the last transaction from the client to be resent.

It seems natural that rather than merely retransmitting the previous transaction we could include fresh events if the capacity still allows it. It would then not be purely a retransmission per se but seems like an improvement to consider right off the bat. However, such techniques complicate the client's logic, whether at the sequence number algorithm, for example. To allow the insertion of new events that were not in the transaction in the first place we would need some kind of per event acknowledgment over a transaction-based acknowledgment and would result in a per event sequence number type of algorithm.

The more we wish to optimize the amount of events per transaction, the more complicated the software running in user space on the switches and routers can be. It is a compromise we made from the start since we wanted a working draft of a protocol without being too greedy and killing the network performances with too much additional telemetry packets once it was released.

The choice of using two queues for two types of events introduces a risk of race condition if no synchronization mechanism is used.

In fact, these queues are implemented as linked lists and used on one end by the polling thread, which adds events (elements) at a regular interval. On the other end the same queue is used by the communication thread that removes these events to transmit them to the server. It is therefore possible that these data structures are accessed at the same time, that some pointers are erroneously assigned resulting in lost events, memory leaks, or worst-case scenario a program shutdown.

To solve this problem, the header `<pthread.h>` is included, which allows us to call the function `pthread_mutex_lock(pthread_mutex t *mutex)` which locks a mutex object that identifies a mutex. If another thread has previously locked the mutex, the thread waits for it to become accessible. When a thread locks a mutex, it becomes the current owner and remains so until another thread unlocks it. These functions were used to perform add and delete operations on the two queues.

Another technical detail can be found in the `recvfrom()` method used to read on the socket. This call is blocking by default. It means that when a process issues a `recvfrom()` that cannot be completed immediately (due to the lack of a packet arrival), the process is put to sleep until a packet arrives on the socket. As a result, if a series of events are available and the thread is blocked on a `recvfrom()` call waiting for an ACK response, there is a missed opportunity to send a transaction that does not require an ACK.

There are numerous ways to make this call non-blocking, one of which is to use the `O_NONBLOCK` flag in the `fcntl()` function to set the socket to non-blocking mode, used in the implementation.

### 5.2.3 Framing

When creating a transaction it is imperative to define its maximum size so that it does not exceed the maximum size of a UDP datagram which is $\approx 65Kb$. However, rather than deciding to create huge transactions approaching the maximum payload size of a UDP datagram, it is probably just as interesting to reduce it to avoid IP fragmentation.

Indeed, when a source sends an IP packet on the network, it cannot be transmitted in its entirety if its size is greater than the smallest Maximum Transit Unit (MTU) of the network through which it is transmitted. In other words, this means that if each link has an identical MTU an IP packet larger than the MTU would be broken into several fragments all smaller than the MTU, which on Ethernet networks are 1500 bytes. This phenomenon is called IP fragmentation. The fragments are then reassembled at the destination to reconstruct the original IP packet sent by the source.

However, these fragments are routed as standard IP packets and therefore can be lost, dropped, delayed by switches or routers on the path from the source to the destination.

Fragmentation has several disadvantages but there is one that is of great concern in our case.

If at the receiver end a fragment is missing at the time of reassembly then all the fragments initially associated with the data to be sent are resent by the source in the case of a reliable transport layer protocol like TCP

or are simply ignored and dropped in the case of an unreliable transport layer protocol like UDP which is used in our implementation. This would mean that a lost fragment of 1500 bytes out of $65Kb$ (1 fragment out of 43) would cause our protocol to retransmit the 65Kb in the case of a transaction requiring an acknowledgement or a loss of $65Kb$ of events if the transaction did not require an acknowledgement. An alternative would be to ensure that each transaction contained in a UDP datagram encapsulated in an IP packet does not exceed 1500 bytes of MTU. That way, only the lost transaction (composed of fewer events) will be retransmitted, as if only the missing fragment were retransmitted rather than all the fragments.

A design approach that takes advantage of the application-specific semantics details of network environments in order to be efficient and thus give the programmer control over the data transmission is a concept called Application Layer Framing (ALF) introduced by D. D. Clark and David L. Tennenhouse [9] and is applied.

Since we considered an Ethernet network in our implementation, the maximum payload size is 1500 bytes. From this we have to subtract the 20-byte IP header and the 8-byte UDP header that together would give a maximum transaction size of 1472 bytes. However, if we want to go deeper we must also take into account the 9 bytes that could be introduced within the Logical Link Control (LLC) layer which is the upper sublayer of the data link layer (layer 2) of the seven-layer OSI model. We also decided to introduce a safety margin of 28 bytes [1] to ensure that no fragmentation would occur resulting in a maximum transaction size of 1435 bytes.

## 5.3 Lambda server

The lambda server will load a JavaScript Object Notation (JSON) config file that contains a mapping between the event identifier and the identification of the lambda function associated with it when it first starts up. This mapping can obviously contain many event identifiers for a single lambda or multiple lambdas. This file must be manually configured and is written in the form of a dictionary for readability and simplicity.

Because the server was developed in C++, it was great to be able to leverage external libraries to read and write JSON files, as well as manipulate them in memory. To do so, I utilized Niels Lohmann JSON C++ library [14]. It's currently available from a Github repository and it's one of the most popular C++ references with over 25k stars.

If the server loads this mapping in memory it is because it is the role of the lambda server or lambda servers that are in the network to

[1] Related to other protocols that might be used (e.g MPLS) or if IPv6 is used instead of IPv4

trigger the lambda functions associated with the events received from the programmable devices.

In this work we gave the lambda server three main functionalities with the first one being the set up of a UDP socket to receive event transactions from all the routers and switches of the network.

The second is to parse these transactions and respond to the initial sender with an acknowledgement or not if no response is required. To this end, we are using a sequence number system implemented as an array where each index corresponds to a switch identifier number and whose state is kept at any time.

The third entails triggering the execution of one or more lambda functions that are linked to each event identifier in the transaction. We are re talking about triggering the execution here and not direct execution, because the piece of code associated with the event identifier will be executed by the various host machines in the Kubernetes cluster and not on the lambda server itself as previously described in the architecture.

## 5.4 Lambda functions and lambda triggers

Once Kubernetes is up and running and that Kubeless is installed the next step is to register the function with Kubeless. Registering the function with Kubeless involves telling Kubeless few bits of information such as :

1. Name of the lambda function
2. Trigger type
3. File location
4. Function name to call within the file
5. The runtime to be executed to run the code
6. Dependencies (external libraries)

The command used looks like this

```
$ kubeless function deploy lambda1 --trigger-http --runtime \
nodejs8 --handler lambda1.main --from-file \
/road/to/file/lambda1.js --dependencies package.json
```

Kubeless will create a new `lambda1` function. It is important to note that this does not have to be the same as the function name used within the

code. It will be possible to trigger the execution of the function via HTTP. The runtime instructs Kubeless to run the code using NodeJS v8, and the handler specifies which function within our code should be executed, as the file may contain several other functions. The dependencies settings specify the runtime libraries that might get accessed and must be included to be able to run the code.

Deploying a function has never been so easy and really sums up in a few words why a serverless infrastructure is advantageous and how it allows the developer to not spend hours managing code deployment. Once the lambda function is deployed it is fully supported by Kubernetes and Kubeless whether it is for scaling or allocating the execution environment, executing it, logging it, freeing the space once executed.

The two lambda functions developed for this master thesis are very close in terms of implementation.

The first one, called `heavy-hitter-detection`, aims at detecting heavy hitters. The heavy hitters are in fact streams that exceed a certain accumulative size over a time window. This size is defined by the network administrator as a realistic threshold for what he expects to receive or observe in general. A heavy hitter could be a stream exceeding Terabytes at Netflix while on the university campus one might associate this behavior with several tens of gigabytes for example.

The lambda function will have the simple goal of determining whether or not this flow passed to the function is considered as a heavy hitter or not by exploiting information passed in the HTTP request as well as information stored in the database hosted in the Kubernetes cluster. If it was not a threat, it would simply update the corresponding entry in the database with the new information received. If it was, the lambda function would log the detection of a heavy hitter in a log file for further analysis.

The second function, called `port-scan`, aims to detect different behaviors related to port scanning. In the same way the lambda function will receive as argument of the HTTP request emitted by the lambda server the data of the flow and its statistics associated. The statistics collected are no longer the size and number of packets described for the previous lambda function but rather the number of TCP segments with SYN flag, the number of empty UDP packets, and the number of TCP segments with RST flag. The thresholds will be checked for each of them and if one of them is exceeded the lambda function will alert in a log file that someone is potentially scanning the ports of different services present in the network or might be doing something fishy by bombarding SYN segments that might be associated with a SYN flooding attack.

To enable function routing, Kubeless uses Kubernetes Ingress. A deployed function is matched to a Kubernetes service by default, with ClusterIP as the service name. That is, the feature isn't available to the

whole public and stays private to the Kubernetes network.

These services are therefore reachable via HTTP as mentioned previously. From there, the lambda server written in C++ must be able to communicate via this protocol and this is only possible if an external library is used. For that `cpp-httplib` [10] of Yuji Hirose under MIT license has been used. Again, this is very well supported and up to date as well as very easy to use for the present purpose.

## 5.5 Kubernetes

### 5.5.1 Minikube

Minikube is a command-line tool for running Kubernetes (k8s) on your own PC. It produces a virtual machine-based single-node cluster. This cluster allows you to demonstrate Kubernetes operations without having to install full-fledged K8s, which takes time and resources.

You can quickly try out Kubernetes installations, conduct development activities, and test settings thanks to this flexibility.

The `Minikube 1.22.0` was used for this work alongside the `kubectl client 1.16.0` and `Kubernetes 1.21.2` running on `Docker 20.10.7`

### 5.5.2 MongoDB

The set up of MongoDB on Kubernetes was highly inspired through the article [24] and the official documentation provided directly by Kubernetes official website [29]

It takes a lot of effort to run and manage stateful applications or databases like MongoDB and MySQL. When a container is shut down or moved to a different node, stateful applications must keep their data (for example, if during a failover or scaling operation, the container was shut down and re-created on a new host).

So we're aiming for a MongoDB replica set, which consists of multiple instances of the database distributed across multiple nodes for redundancy. A master with read and write permissions is necessary for synchronization, while secondaries can only do reads and must be synced with the master to stay up with changes such as Figure 5.4.

In order to implement this, we can find the Kubernetes ReplicaSets and

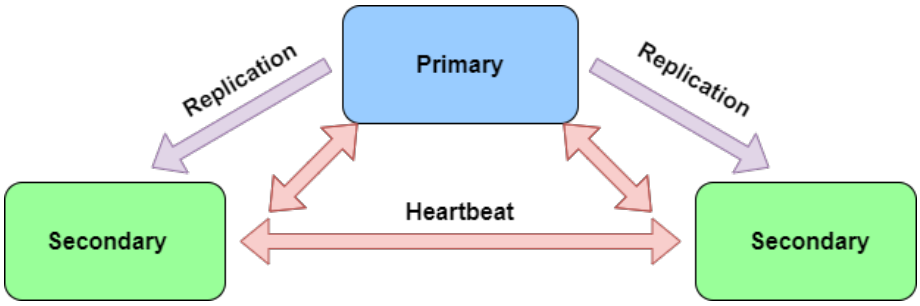



Figure 5.4: Typical MongoDB structure with replicas

StatefulSets. The goal of a Kubernetes ReplicaSet [2] is to keep a consistent set of replica pods running at all times. As a result, it's frequently used to ensure the availability of a certain quantity of identical Pods. A ReplicaSet ensures that a specified number of pod replicas are running at any given time and can share the same persistent volume. However, this would not work for Database nodes because no synchronization would allow for pod coordination, and the ReplicaSet does not include any stateful features required for such applications.

A StatefulSet [3], on the other hand, can correctly detect pods in stateful applications with persistent storage in order to preserve the state. This feature is provided by StatefulSet, which creates pods with a persistent identity that will remain relevant throughout rescheduling. This way, even if a pod is recreated, the storage volumes will be correctly mapped, and the application state will be retained. Pods in a StatefulSet are launched in a prescribed order. The next pod will not be launched until the preceding one has been successfully initialized. This manner, you can plan your pod deployment with confidence, knowing that "name-0" will be the first to launch. The master node in MongoDB will create a replica set. The pods with the names "name-1," "name-2," will identify that a replica set has already been formed and will connect to the existing nodes.

To obtain a MongoDB replica set (not to be confused with the notion of Kubernetes ReplicaSet seen before) we will need three things:

1. The StorageClass is there to define the type of database that we will use for the nodes that will be in charge.

2. Then, we create a dedicated service that points to each of its member pods. This service must be "headless", i.e. it does not create a ClusterIP for load balancing, but is used for the static DNS naming of the pods that will be launched. This service will cause the creation of DNS records listed in this format: "name-0", "name-1", "name-2" as explained above.

[2]https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/
[3]https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

3. The StatefulSet is the most important component of the three because it is the one that will glue everything together and will allow MongoDB to run properly by specifying how to use the storage, what volumes to mount, and the specifications of MongoDB replica set such as the number of members in the replica set and ports to use.

# Chapter 6

# Evaluation

The purpose of this evaluation is to determine the suitability of this framework for the task at hand and to ensure scalability.

## 6.1 Architecture analysis

The analysis of the architecture consists in analyzing in more detail the numbers and giving an order of magnitude of the load introduced in the network as well as the various parameters which influence it in order to be able to adjust these at best. This analysis can also be used to transpose this solution to other problems that require the same needs but might operate in a different environment with more or less constraints on the network parameters. This section also tackles the limitations of the implemented protocol and the way the counters are stored and polled from the switches.

This section is going to analyse the first steps of the whole pipeline described in this work and for that they are briefly summarized below.

Counters are incremented internally in a table in the internal memory of the switch each time a packet is received on one of its interfaces. It is made possible thanks to an XDP hook attached to it, allowing a small BPF program to be executed. Its role is to parse the packet and store information related to it in a BPF map. This data structure called BPF map is a hash table where the key is a hash of the fields defining the flow and where the associated value is composed of the flow itself as well as counters that are called statistics of the flow such as its size, its number of packets, TCP flag counter and a last important data that the time of last update of this entry in the table.

At the same time a part of the program in user space has for only role to read this BPF map every two seconds and from the entries modified

between the last poll and the current one, to create events to be inserted in queues intended to be sent to the lambda server.

In the same way, another part of the program aims to send the events of this queue to the lambda server. It is important to remember that two queues are used rather than one: one dedicated to events that must be acknowledged and the other not. The sending period of this program is also 2 seconds but its starting point is slightly delayed by a few milliseconds in order to be out of sync with the polling thread. If they were to start their execution at the same time, new events might need to wait a whole sending period whereas in our case it is bounded by the delay introduced of a few milliseconds only. It is clear that an event is not necessarily created for each packet and that it is up to the network administrator to choose what the event is made of and if it is imperative to send it as soon as it is created.

### 6.1.1 Bandwidth requirements

From that it is interesting to see the bandwidth requirements introduced by the protocol by looking at the worst case scenario.

The first parameter that will influence this additional load is the size of the BPF map. Indeed, let's take an example where this hash table is composed of 1000 entries that by definition can store up to 1000 flows. Each of these entries must be able to store twelve integers which is equal to

$$12 \times 4bytes = 48bytes \tag{6.1}$$

of information on average for the programs provided for this project. This includes the flow fields, counters, time of update. This would correspond to a capacity of the BPF map of

$$1000 \times 48bytes = 48Kbytes \tag{6.2}$$

Additionally, in the worst case scenario all the entries would be modified between two polling probe of the user program. Furthermore, to each entry would be associated an event to be sent to the lambda server.

Let's now calculate what will actually be sent to the lambda server in terms of number of bytes. The transaction header is made of 10 bytes. It is followed by several events composed of a fixed size header of 9 bytes and their variable size payload. If we consider that each event payload is the equivalent of what is stored in the table for each entry (that is a reasonable assumption because that is exactly what it is doing) we would end up with an average of

$$48bytes + 9bytes = 57bytes \tag{6.3}$$

per event.

With an application payload of 1435 bytes, we can transport 25 events ($25 \times 57bytes$) plus a transaction header of 10 bytes in each packet. For 1000 events, we need 40 packets, and this every 2 sec leading to a rate of

$$\begin{aligned} 20\ packets/sec \times 1500bytes &= 30Kbytes/sec &(6.4)\\ &= 240Kbps &(6.5)\end{aligned}$$

in a scenario with 1000 active flows in the switch. This rate would vary linearly with the number of active flows.

In conclusion, the main parameter for evaluating the throughput introduced is the size of the hash table used. A smaller size and the throughput will be reduced but the risk of collision will increase which will negatively impact the accuracy of the network as such. A larger size and the number of flows that can be stored increases but at the expense of a larger amount of network traffic.

A collision in the implementation's state results in the new flow colliding being ignored in order to retain the flow currently stored for that same key. We describe this inaccuracy as global network inaccuracy, not the loss or incorrect change of an already existent flow. As a result, we lose information on a flow that could be stored later if the key becomes available but several packets would have been forgotten in the meantime, making the statistics obtained inaccurate and resulting in another type of inaccuracy which is a per flow inaccuracy.

The overall throughput created seems minimal but depending on the number of switches for which these programs would have been deployed and depending on the maximum bandwidth of the links of the topology and the capacity of the lambda server to receive such a load, one may wonder if this is not already too much network traffic introduced into the network. This could negatively impact the Quality Of Service (QoS) for a rather minimalist BPF map size of 1000 entries.

## 6.2 Functional tests

In order to ensure the reliability and functionality of the provided code, several tests have been carried out. Indeed each module was tested individually before being integrated together and proceeding to the final

tests which consists in linking all the pieces of the puzzle and checking that the described attacks are detected. In the same way as verifying the good functionality of the developed product it was also important to test some additional functionalities and the behavior of the programs under stress in order not to bias the measures taken and presented in this thesis.

### 6.2.1 Mininet Testing

Regarding the tests of the different eBPF programs it was necessary to create virtual network interfaces in order to avoid working directly on the real interfaces of the machine. For this we chose to use `Mininet 2.3` which allows in a few commands to create a very realistic virtual internet network with the possibility to customize the topology like the technologies used, the protocols, the controller type as well as to take measurements and simulate traffic between several entities.

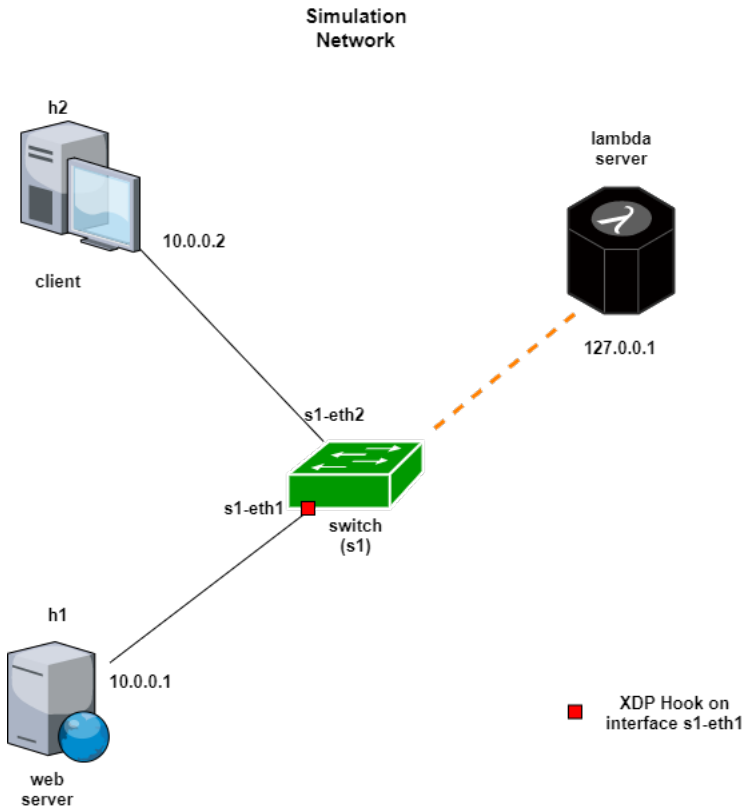



Figure 6.1: Mininet testing topology

You will find on Figure 6.1 the Mininet topology composed of a Web server, a client making requests to the web server and a switch linking both entities. The lambda server is in some regards directly connected to the switch in this configuration. This topology was used to test the different eBPF programs but also to test the protocol between the switch and the lambda server.

### 6.2.2 Packet loss simulation

In order to test if the packet retransmission part of the protocol worked properly for transactions that needed it we had to simulate packet loss in one way or another. A first way of doing it would have been to simulate this using the Mininet features but it was just as simple to simulate this drop at the lambda server level. Indeed, when launching the server at the command line, a parameter specifying the drop rate is requested.

### 6.2.3 Protocol assessment

With the help of `Wireshark` [1] `2.6` to capture and then analyze the order of the packets as well as their content for a more detailed analysis we were able to test the protocol. Additionally the number of events at the sender and the receiver end were compared as well as the buffer and events detailed that were sent. This was mentioned briefly, but the packet loss simulation functionality was also used to verify the operation of all parts of the protocol as well as the correct triggering of the timers used.

### 6.2.4 Valgrind

Knowing that the implementation was going to be done in C and C++ languages which do not benefit from any garbage collection systems, and that the application was intended to run on switches or routers with limited memory capacities, it was very important to use Valgrind [2] throughout the development. Indeed, Valgrind is a tool that allows to check for memory leaks, out of band memory access, and many others in order to optimize the allocation of memory space and avoid any type of error.

## 6.3 Measurements

In this section, a brief performance evaluation of the protocol is discussed. It was chosen to show only the performance of the protocol and not the lambda functions execution since it's already been discussed partially in section 4.6 although it still might have been interesting to draw our own conclusion on that matter.

The Proof of Concept was developed and tested on a Ubuntu 18.04.5 Bionic Beaver 64bit distribution on virtual machine using `VirtualBox` [3]

[1] https://www.wireshark.org
[2] https://valgrind.org
[3] https://www.virtualbox.org

6.1.18. The hardware available is an Intel i5-8300H (8) CPU with a base frequency of 2.3GHz and a max turbo frequency of 4.0Ghz with 16GB of RAM. It is important to clarify that only 4 hyper-threads out of 8 were assigned to the virtual environment as well as 8GB of RAM. It is the same configuration that was used for the measurements presented in this section.

Only two processes were active during the measurements: the dummy switch polling and delivering events to the server, and the lambda server receiving the events. It's important to realize, however, that these measures will not be worth as much as if we had been able to test on equipment that was physically separate from each other. As a result, the virtual machine simulation operates the client and server on the same machine, with specifications that are a priori different from what is available on the market as server or switch configurations.

### 6.3.1 Results

There is a new network traffic drawback that, as we've seen, is proportional to the size of the BPF map. We are entitled to wonder if the protocol would still scale if larger BPF map sizes were used in order to limit the collision rate and thus limit inaccuracies of what could be measured.

We begin by looking at how the size of the BPF map affects the transmission of event transactions that do not require a response from the lambda server. To accomplish so, we'll simulate an addition of events to the queue every two seconds and watch how the queue's capacity changes. We'll see if the sending thread, which has a also a two-second sending time period, is able to send the events or not.

Figure 6.2 shows a filling rate equivalent to a polling rate of 200 000 events every two seconds. We can see that the blue curve representing the state of the queue at any time of the simulation is emptied fast enough before the next filling cycle.

On Figure 6.3 2 000 000 events every two seconds are added to the queue. The blue curve allows us to see that the communication thread is no longer able to send events as fast as they are added to the queue. Indeed, this number keeps growing as well as the number of events sent. The protocol is still able to process them but its ability to send large amount to the lambda server reaches a limit. It does not hold anymore and the queue might fill up the whole memory of the device if the simulation kept going for several minutes.

What is not shown on the graph is the tipping point at which the polling thread is filling the queue faster than the sending thread can send the events to the lambda server. In order to obtain that number many simulations between 200 000 events/2s and 2 000 000/2sec were conducted

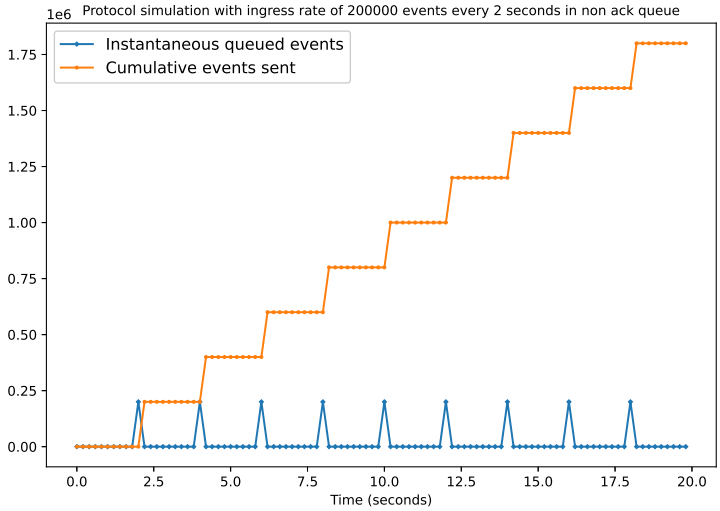



Figure 6.2: Protocol measurement for non-ack queue with a BPF map polling rate of 200 000 events/2sec

to conclude that 950 000 events every 2 seconds is the maximum threshold regarding the non-ack queue alone.

In a second step we decided to analyze the performance of the stop-and-wait type of logic used to process the events that required an ACK from the lambda server. To this end, we simulate three different network drop rates of 1, 3 and 5 percent for different BPF map sizes. The drop rates are chosen in order to force retransmissions and thus be able to observe the limit beyond which the protocol is no longer capable of working as expected.

Figure 6.4 shows a queue filling rate of 15 000 events per two seconds. When we look at the three graphs we can see very little variations in the period of time necessary to empty the queue as retransmissions occur. However, on the third graph of the same figure we can observe these irregularities at a bigger scale because more retransmissions occur. That being said, the queue is still emptied before the start of the next filling cycle which ensures that the protocol can hold this rate.

In Figure 6.5, we propose 80 000 events every 2 seconds for the three drop rates already presented.

The first graph shows no sign of weakness, but the two following graphs allow us to observe the limit of the stop-and-wait protocol implemented with regard to the events requiring an acknowledgment. Even if it seems that the second graph is able to fully empty the queue it is not the case. The size is increasing at a very slow pace.

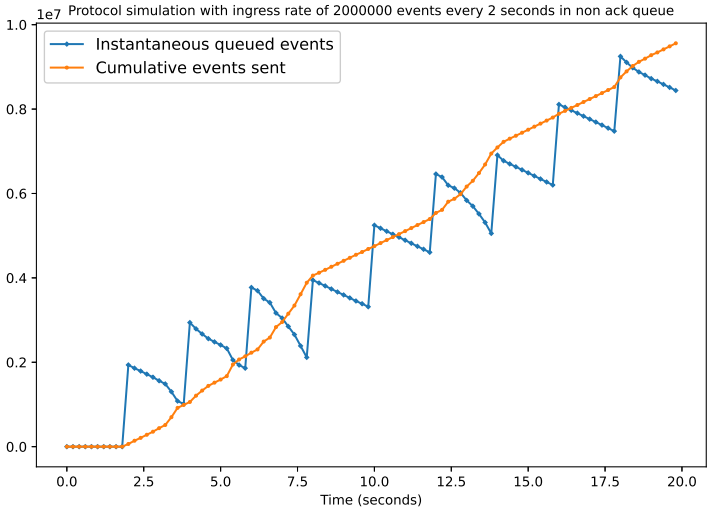


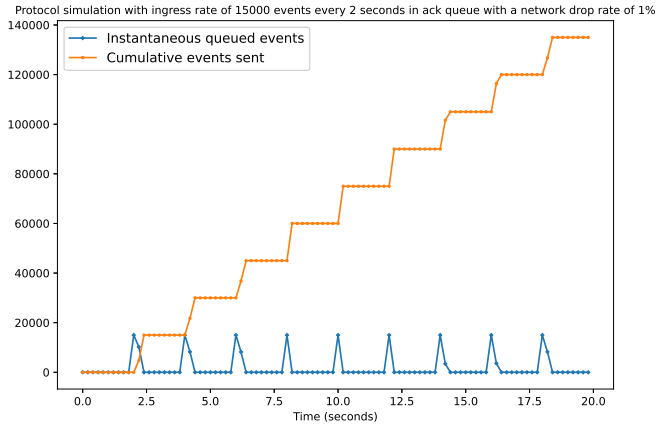

Figure 6.3: Protocol measurement for non-ack queue with a BPF map polling rate of 2 000 000 events/2sec

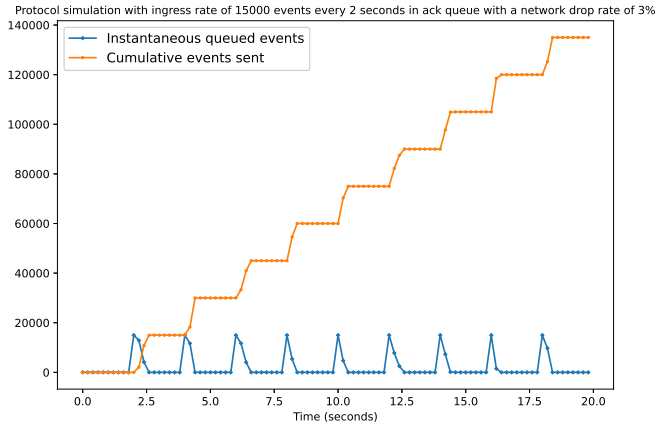
As the size of the queue only increases, it may eventually occupy all the memory available, ending the execution of the entire program. In fact both maximum thresholds were also obtained by doing multiple simulations at different filling rates. We found that the maximum rate that the process could hold is about 70 000 events/2s for a network drop rate of 3 percent and 35 000 events/2s for a drop rate of 5 percent. Regarding the network drop rate of 1 percent, the sending thread went higher with a maximum of 170 000 events/2s before breaking.

These results are summarized in Table 6.1.

| Max Threshold (number of events every 2 seconds) | Simulation Type |
|---|---|
| 35 000 | Stop-and-wait (ACK) with 5% network drop rate |
| 70 000 | Stop-and-wait (ACK) with 3% network drop rate |
| 170 000 | Stop-and-wait (ACK) with 1% network drop rate |
| 900 000 | No acknowledgment policy |

Table 6.1: Summary of the maximum number of events that can be added in the different type of queue when considered independently of the others



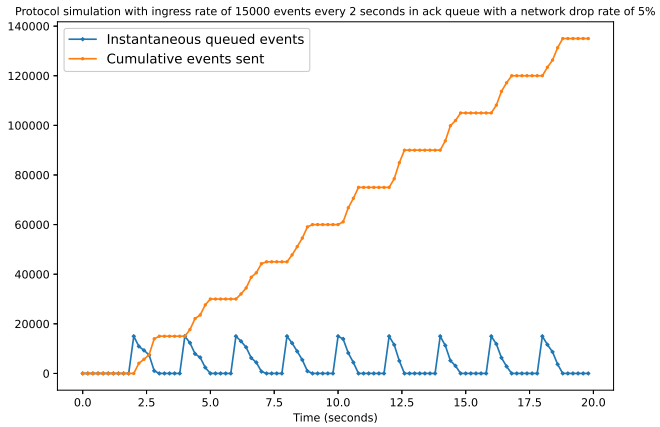
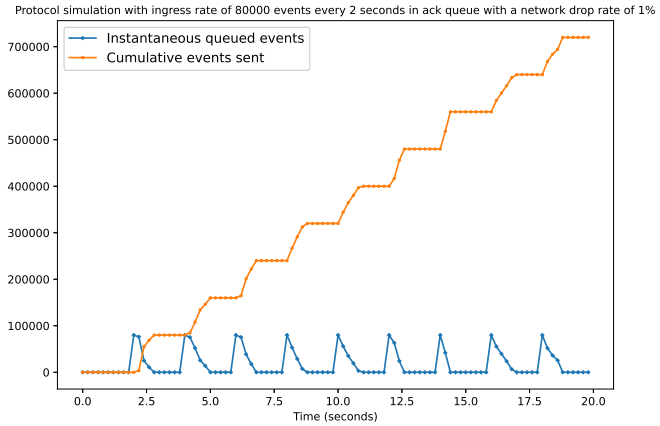
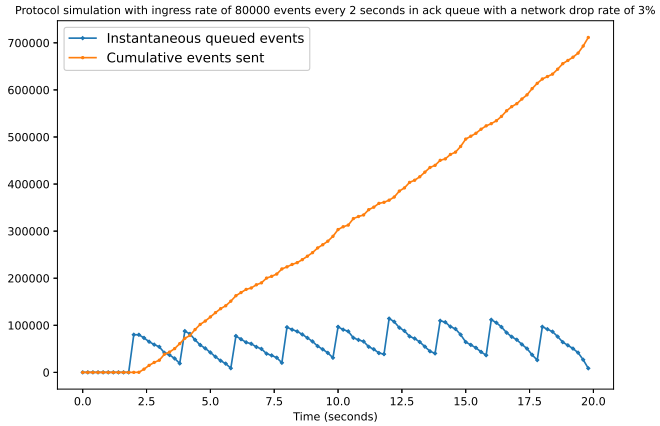




Figure 6.4: Protocol measurements for the acknowledgment queue with a BPF map polling rate of 15 000events/2s and a network drop rate of respectively 1%, 3%, and 5 % (top to bottom)



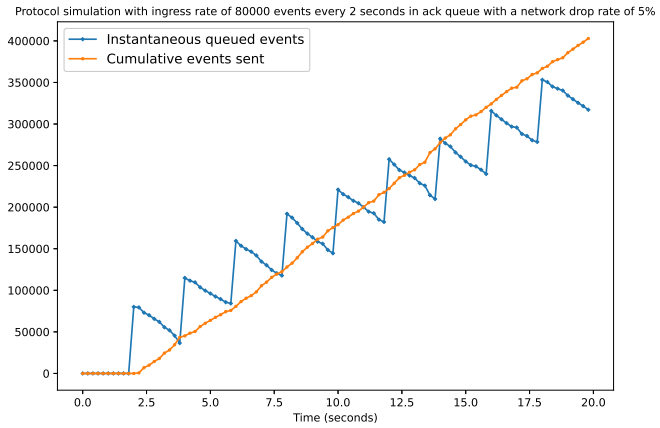




Figure 6.5: Protocol measurements for the acknowledgment queue with a BPF map polling rate of 80 000events/2s and a network drop rate of respectively 1%, 3%, and 5 % (top to bottom)

# Chapter 7

# Future Work

Significant improvements and additional work are possible. These are detailed further below.

## 7.1 Overall architecture

A clear limitation of the solution proposed in this work is the fact that the BPF program which collects the data of the packets received on the NIC of the switch could be duplicated if a frame had to pass by several switches having the same program. Indeed, these switches would increment their local counters each on their side and then communicate them to the lambda server which will make the erroneous aggregation of the values that will not correspond to the reality of things. This means that we have to discuss on which interface of which switch we should attach these BPF programs to avoid this kind of scenario. This limits the flexibility of the proposed solution which, for example, will impose that only the ingress points of the network will be equipped with these programs, thus preventing certain monitoring metrics from being available everywhere in the network.

Secondly, there is also the weakness of the polling system in place. Should we choose a regular polling at the expense of the intensive use of the switch CPU ? Should we accumulate in memory longer at the expense of the precision of the tables and the memory of the switch? This trade-off needs to be explored in much greater depth than it has been and requires asking the right questions about what we want to achieve and at the expense of what service.

A third point that must be emphasized is the good collection of data. Indeed, the lambda functions (application) will only be as useful as the quality of the data they have. For example, the detection of SYN flooding requires more information such as whether the 3-way handshake was

able to complete or not in order not to confuse a client that opens many legitimate connections with a server and an attacker that would only make the first exchange under different IP addresses making it impossible for him to correctly process the entire 3-way handshake. This data again requires more storage space, more logic and therefore more CPU from the BPF program but would drastically increase the accuracy of detection of such an attack.

## 7.2 Security

Security was not a topic to consider in the development of this project. It is therefore always wise to recall that the deliverable is not a deployment version for this security reason and for many others described throughout the work. The implemented protocol does not require any authentication between the two entities (switch and lambda server) and the collected metrics (transactions) are exchanged in clear in the network. This network being private and the data not sensitive, we assume that this does not represent a main threat at the time of the development of this kind of solution and that it seems a great idea to bring security its own focus in a follow up of this thesis only.

## 7.3 Implementation

There are several implementation enhancements that might be made. One of them is about optimizing the transmission of events generated by BPF map entries. To avoid too many collisions, it is tempting to utilize huge hash table sizes. However, we have seen that if a big enough number of entries are modified between two polling probes, the sending thread might encounter serious difficulties (depending on the type of queue used discussed during the evaluation). It would thus be interesting to cope this limitation with a mechanism that allows some sort of filtering mode when the number of modified BPF map entries exceeds the set threshold. This mode would include delaying or disregarding some table entries that convey very little new information in order to never surpass the defined threshold and prevent the system from entering a limit condition.

## 7.4 Controller

The emergence and popularity of SDN networks with a controller responsible for routing and decision making to ensure quality of service is significant. It is therefore also interesting to explore a way that consists in not only

collecting data allowing to detect attacks in the network but also metrics of queue sizes, congestion, delays in order to have lambda functions able to give feedback to the controller so that it can with a certain precision modify routing and thus optimize the network load.

## 7.5 Protocol

The protocol developed for the exchange of transactions between the switches and the lambda server is simple and could seem very slow or blocking since it is based on a stop and wait protocol. It does not allow a huge throughput but it does manage a relatively low volume of additional monitoring data introduced on the network. This trade-off should be studied more fundamentally in practice in order to obtain a fair balance for the type of hypothesis made at the beginning.

## 7.6 Benchmarking

More benchmarking and performance testing are required. This is due to the fact that the goal of this study is to create a platform that functions on a rather large private network involving a wide range of hardware and networking devices. Local testing is difficult since replicating an authentic environment is strongly dependent on usage patterns, available resources, and the accuracy that these tests aim for.

# Chapter 8

# Conclusion

This thesis begins by defining the objectives that are related to the detection of attacks and network events in real time. Several assumptions are made such as the fact that the switches and routers of the network under consideration are programmable and that the size of the network is large. A serverless infrastructure is suggested because it would allow to associate lambda functions in response to events perceived in the network and would be entirely managed by a cloud provider.

Then a study of technology and what was being done in this field had to be researched in order to perhaps draw inspiration from previous projects or identify helpful building blocks for the architecture's design.

XDP was used for the packet processing part of the solution allowing to parse packets at line rate and to be able to store the necessary information for the detection of attacks and network events. There is also the idea of introducing a controller called lambda server, which works as a proxy for the lambda functions since it is in charge of triggering their execution. A protocol to exchange data from the switches to the controller had to be designed and implemented. It was decided to use UDP with an acknowledgement system if it was necessary for some types of messages or events. The part of the protocol that handles acknowledgements is inspired by a stop-and-wait protocol and uses sequence numbers. As for the serverless infrastructure, we finally decided to go for Kubeless, an existing open source framework that relies on Kubernetes. We also wanted to be able to give a maximum of data to the lambda functions other than just the data related to the trigger of these functions. To this end, we introduced persistent storage within the Kubernetes cluster using MongoDB deployed with several replicas.

In the fifth chapter we discussed the details of the implementation of a Proof of Concept and is followed by the sixth chapter which explored the scalability of the proposed solution by focusing on the established protocol and the limits it proposes in terms of the number of events it can route to

the lambda server and therefore the number of lambda functions that can be triggered per second.

Finally we discuss potential improvements that can be made to the project, both in terms of the technologies used and the architectural choices that can be further explored.

# Bibliography


[1] *Andrii Nakryiko's Blog*. 2021. URL: http://nakryiko.com (visited on 27/07/2021).

[2] *Awesome eBPF*. 2021. URL: https://github.com/zoidbergwill/awesome-ebpf (visited on 27/07/2021).

[3] *AWS Lambda*. 2021. URL: https://aws.amazon.com/ (visited on 10/07/2021).

[4] Nagender Bandi et al. 'Fast Data Stream Algorithms Using Associative Memories'. In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. Beijing, China: Association for Computing Machinery, 2007, pp. 247–256. ISBN: 9781595936868. DOI: 10.1145/1247480.1247510. URL: https://doi.org/10.1145/1247480.1247510.

[5] Pat Bosshart et al. 'P4: Programming Protocol-Independent Packet Processors'. In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. URL: https://doi.org/10.1145/2656877.2656890.

[6] *BPF and XDP Reference Guide*. 2021. URL: https://docs.cilium.io/en/stable/bpf/ (visited on 14/07/2021).

[7] *BPF Compiler Collection*. 2021. URL: https://github.com/iovisor/bcc (visited on 10/08/2021).

[8] *Clang*. 2021. URL: clang.llvm.org (visited on 12/07/2021).

[9] D. D. Clark and D. L. Tennenhouse. 'Architectural Considerations for a New Generation of Protocols'. In: *Proceedings of the ACM Symposium on Communications Architectures & Protocols*. SIGCOMM '90. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1990, pp. 200–208. ISBN: 0897914058. DOI: 10.1145/99508.99553. URL: https://doi.org/10.1145/99508.99553.

[10] *cpp-httplib*. 2021. URL: https://github.com/yhirose/cpp-httplib (visited on 04/08/2021).

[11] *Google Cloud Serverless*. 2021. URL: https://cloud.google.com/serverless/ (visited on 10/07/2021).

[12] Toke Høiland-Jørgensen et al. 'The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel'. In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '18. Heraklion, Greece: Association for Computing Machinery, 2018, pp. 54–66. ISBN: 9781450360807. DOI: 10.1145/3281411.3281443. URL: https://doi.org/10.1145/3281411.3281443.

[13] *IBM Functions*. 2021. URL: https://cloud.ibm.com/functions/ (visited on 10/07/2021).

[14] *JSON for Modern C++*. 2021. URL: https://github.com/nlohmann/json (visited on 04/08/2021).

[15] *Kubernetes*. 2021. URL: https://kubernetes.io/ (visited on 10/07/2021).

[16] Abhishek Kumar et al. 'Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution'. In: *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '04/Performance '04. New York, NY, USA: Association for Computing Machinery, 2004, pp. 177–188. ISBN: 1581138733. DOI: 10.1145/1005686.1005709. URL: https://doi.org/10.1145/1005686.1005709.

[17] Chris Lattner and Vikram Adve. 'LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation'. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, p. 75. ISBN: 0769521029.

[18] *Libbpf*. 2021. URL: https://github.com/libbpf/libbpf (visited on 10/08/2021).

[19] Anil Madhavapeddy et al. 'Unikernels: Library Operating Systems for the Cloud'. In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 461–472. ISBN: 9781450318709. DOI: 10.1145/2451116.2451167. URL: https://doi.org/10.1145/2451116.2451167.

[20] Steven McCanne and Van Jacobson. 'The BSD Packet Filter: A New Architecture for User-Level Packet Capture'. In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX'93. San Diego, California: USENIX Association, 1993, p. 2.

[21] Nick McKeown et al. 'OpenFlow: Enabling Innovation in Campus Networks'. In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. URL: https://doi.org/10.1145/1355734.1355746.

[22] *Microsoft Azure Functions*. 2021. URL: https://azure.microsoft.com/en-us/solutions/serverless/ (visited on 10/07/2021).

[23] S. K. Mohanty, G. Premsankar and M. di Francesco. 'An Evaluation of Open Source Serverless Computing Frameworks'. In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2018, pp. 115–120. DOI: 10.1109/CloudCom2018.2018.00033.

[24] *MongoDB with statefulset in Kubernetes*. 2021. URL: https://deeptiman.medium.com/mongodb-statefulset-in-kubernetes-87c2f5974821 (visited on 25/07/2021).

[25] Masoud Moshref et al. 'SCREAM: sketch resource allocation for software-defined measurement'. In: Dec. 2015, pp. 1–13. DOI: 10.1145/2716281.2836099.

[26] *P4*. 2021. URL: https://p4.org/ (visited on 10/07/2021).

[27] *P4 Runtime Specifications. The P4.org API Working Group*. 2021. URL: https://p4.org/p4-spec/p4runtime/v1.2.0/P4Runtime-Spec.html (visited on 10/08/2021).

[28] Andrei Palade, Aqeel Kazmi and Siobhán Clarke. 'An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge'. In: May 2019. DOI: 10.1109/SERVICES.2019.00057.

[29] *Running MongoDB on kubernetes with StatefulSets*. 2021. URL: https://kubernetes.io/blog/2017/01/running-mongodb-on-kubernetes-with-statefulsets/ (visited on 10/07/2021).

[30] Robert Schweller et al. 'Reversible Sketches for Efficient and Accurate Change Detection over Network Data Streams'. In: *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*. IMC '04. Taormina, Sicily, Italy: Association for Computing Machinery, 2004, pp. 207–212. ISBN: 1581138210. DOI: 10.1145/1028788.1028814. URL: https://doi.org/10.1145/1028788.1028814.

[31] Abraham Silberschatz, Peter B. Galvin and Greg Gagne. *Operating System Concepts*. 9th. Wiley Publishing, 2012. ISBN: 1118063333.

[32] *Streaming Algorithms*. 2021. URL: https://en.wikipedia.org/wiki/Streaming_algorithm (visited on 16/07/2021).

[33] B. Tan et al. 'Towards Lightweight Serverless Computing via Unikernel as a Function'. In: *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. 2020, pp. 1–10. DOI: 10.1109/IWQoS49365.2020.9213020.

[34] Jonathan Vestin et al. 'Programmable Event Detection for In-Band Network Telemetry'. In: *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*. 2019, pp. 1–6. DOI: 10.1109/CloudNet47604.2019.9064137.

[35] *What is serverless ?* 2021. URL: https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless (visited on 10/07/2021).

[36] *What is serverless and which entreprises are adopting it ?* 2021. URL: https://www.computerworld.com/article/3427298/what-is-serverless-computing-and-which-enterprises-are-adopting-it.html (visited on 10/07/2021).

[37] *XDP Hands-on Tutorial*. 2021. URL: https://github.com/xdp-project/xdp-tutorial (visited on 27/07/2021).

[38] L. Yang et al. *RFC3746: Forwarding and Control Element Separation (ForCES) Framework*. USA, 2004.

[39] Minlan Yu, Lavanya Jose and Rui Miao. 'Software Defined Traffic Measurement with OpenSketch'. In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 29–42. ISBN: 978-1-931971-00-3. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/yu.