POLITECNICO
MILANO 1863

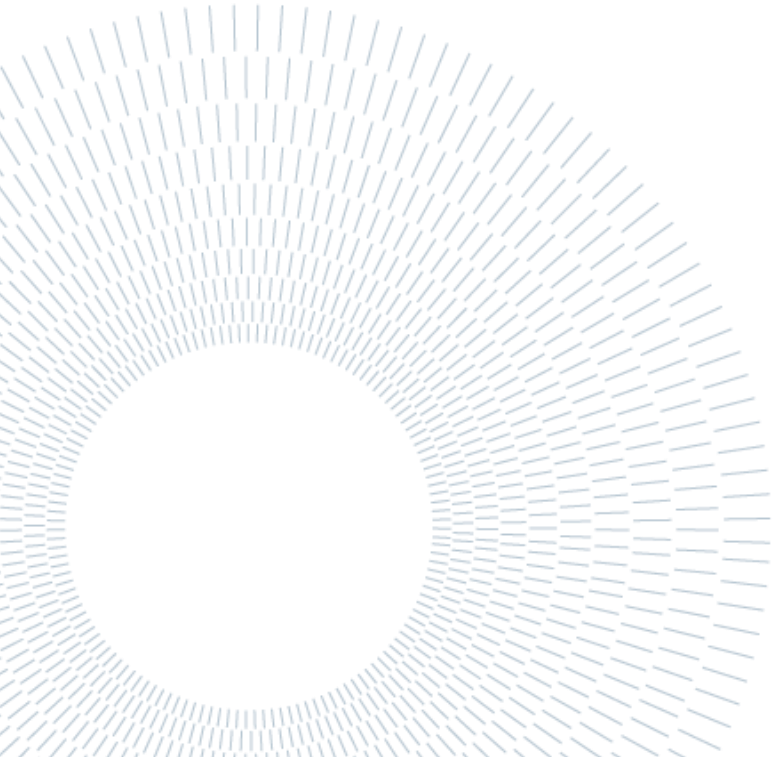
SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

In-band Network Telemetry and CHIMA Framework: A performance Analysis and Optimization Study

TESI DI LAUREA MAGISTRALE IN
TELECOMMUNICATION ENGINEERING
INGEGNERIA DELLE TELECOMUNICAZIONI

Author: **Simone Tonelli**

Student ID: 939480
Advisor: Giacomo Verticale
Academic Year: 2021-22



Abstract

The adoption of Virtual Network Functions has increased the flexibility in the deployment of network services, without the need for specialized hardware, but has lowered the overall performance.

In our study, we focused on a framework called CHIMA, developed for the deployment of heterogeneous Service Function Chains (SFCs). This framework executes SFCs in docker containers and switches running functions written in the P4 language. The goal of this framework was to extend the capabilities of other frameworks in the literature for the deployment of heterogeneous Service Function Chains, exploiting the capabilities of P4 programmable switches to perform real-time monitoring using the In-band Network Telemetry specification and to reroute the traffic in case of violation of the fixed Service Level Agreement (SLA).

In this work, we address the advantages and drawbacks of this framework by introducing some improvements. We replaced the Prometheus web server, an open-source monitoring system, with InfluxDB, an open-source time-series database with a monitoring system, and Telegraf, a specific agent written in the programming language GO to efficiently collect metrics from the CHIMA framework. The advantage of using Telegraf is that this framework is plug-in driven, allowing the user to collect metrics simply and efficiently. We also changed the way we calculate the metrics: latency and jitter, by substituting the EWMA with an arithmetic average. Additionally, we introduced a new metric which is the number of packets received every polling interval.

Key-words: INT, docker, virtualization, VNF.

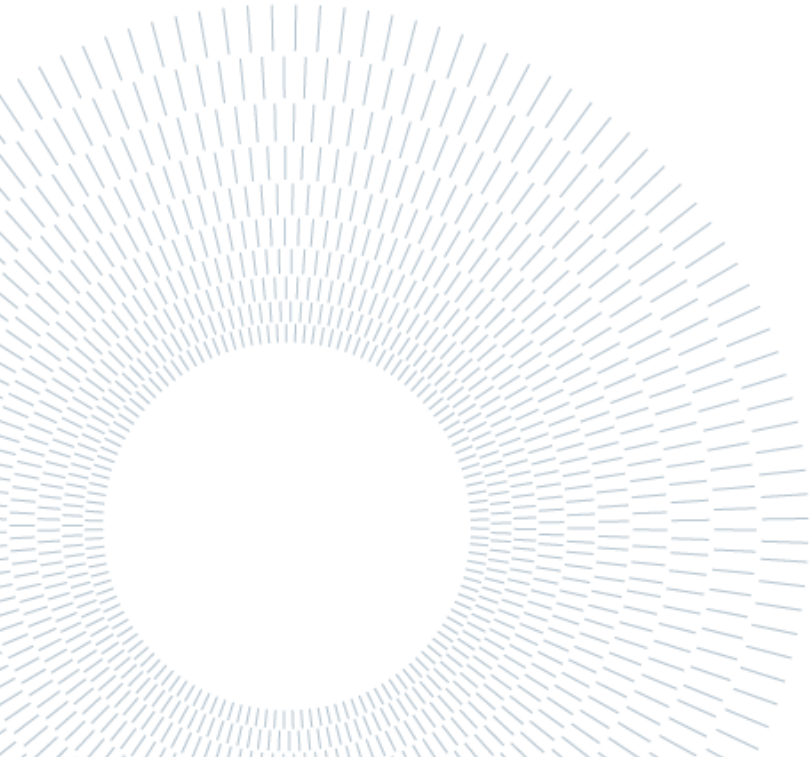
Abstract in italiano

L'adozione di Funzioni di Rete Virtuali ha aumentato la flessibilità nella distribuzione dei servizi di rete, senza la necessità di hardware specializzato, ma ha diminuito le prestazioni complessive.

Nel nostro studio, ci siamo concentrati su un framework chiamato CHIMA [2], sviluppato per la distribuzione di Catene di Funzioni di Servizio eterogenee (SFC). Questo framework esegue le SFC in container Docker e switch che eseguono funzioni scritte nel linguaggio P4. Lo scopo di questo framework era di estendere le capacità di altri framework presenti in letteratura per la distribuzione di Catene di Funzioni di Servizio eterogenee, sfruttando le capacità degli switch programmabili P4 per eseguire il monitoraggio in tempo reale utilizzando la specifica In-band Network Telemetry [1] e per instradare il traffico in caso di violazione del Service Level Agreement (SLA).

In questo lavoro, affrontiamo i vantaggi e gli svantaggi di questo framework introducendo alcune migliorie. In particolare, abbiamo sostituito il web server Prometheus, un sistema di monitoraggio open-source, con InfluxDB, un database di serie temporali open-source con un sistema di monitoraggio, e Telegraf, un agente specifico scritto nel linguaggio di programmazione GO per raccogliere metriche in modo efficiente dal framework CHIMA. Il vantaggio nell'utilizzare Telegraf è che questo framework è guidato da plug-in, permettendo all'utente di raccogliere metriche in modo semplice ed efficiente. Abbiamo anche cambiato il modo in cui calcoliamo le metriche: la latenza e il jitter, sostituendo l'EWMA con una media aritmetica. Inoltre, abbiamo introdotto una nuova metrica che rappresenta il numero di pacchetti ricevuti in ogni intervallo di polling.

Parole chiave: INT, docker, virtualizzazione, VNF.



Contents

1 Introduction

In this work, we will analyse in-depth some aspects of the CHIMA (CHain Installation, Monitoring and Adjustment) framework developed by Battiston, a framework able to deploy services in a network function chain and reroute the traffic in case of a violation of the Service Level Agreement, pointing out the drawbacks of his work and proposing solutions to them. Specifically, we will examine the In-band Network Telemetry collector used in the framework and propose improvements. We noticed that the collector uses an Exponentially Weighted Moving Average (EWMA) to overcome the limitations of eBPF, an efficient kernel-level programming language for networking, which cannot perform division with unsigned integers like jitter values. However, the use of EWMA causes delays in detecting a Service Level Agreement (SLA) violation. Therefore, we propose an efficient way to overcome this problem in this paper.

The second improvement we made is the introduction of InfluxDB, a time-series database, and Telegraf, a plug-in driven agent written in GO-Lang, to collect and process the metrics of the In-band Network Telemetry [1].

The work is structured as follow: First, we are going to give to the reader an introduction to the background technologies we are going to use in this thesis, in particular we're going to analyse the P4 language, able to deploy services in bmv2 P4 switches and enables the In-band Network Telemetry. Then we will talk about the INT and the previous CHIMA framework [2].

In the third chapter we briefly analyse the state of the art regarding the INT from which CHIMA is based.

In the fourth chapter we propose to the reader an overview of the new CHIMA framework, in particular the renewed INT collector and the new ecosystem composed by InfluxDB and Telegraf with the aim of substituting the Prometheus time-series database.

In the fifth chapter we will propose some tests and the associated numerical results to validate the new CHIMA framework and to point out some limitations in the use of docker containers due to lack of performance.

2 background

In this chapter we will briefly introduce the P4 language, the INT concept and the CHIMA framework [2].

2.1. P4 Language

In 2016, the Open Networking Foundation introduced the concept of INT [1], which leverages the P4 language [3] to configure network devices such as switches and routers. P4 is "vendor-agnostic," meaning it can be used with a range of hardware and isn't tied to any specific vendor. Previously, vendors had complete control over network functionality, but P4 enables the extraction of specific data embedded in packets, such as INT data, in the context of INT, which is the focus of this thesis.

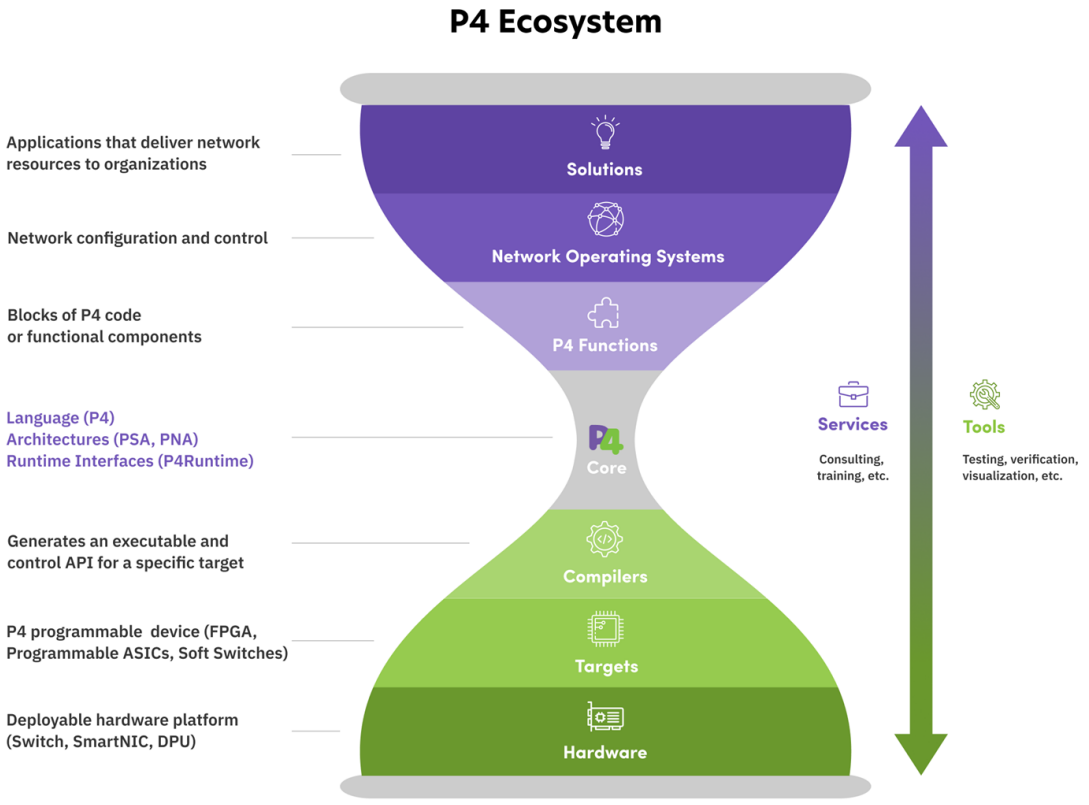


Figure 1 The P4 ecosystem [3]

2.1.1. Why P4

The use of P4 programming language gives a lot of advantages compared to the previous vendor-specific implementations.

1) Protocol independence: P4 allows the user to define his own network protocols and packet processing logic, rather than being bounded to some pre-defined protocols. This translates into a flexible and adaptable framework.
2) Hardware independence: as said earlier in the previous paragraph, P4 is completely vendor-agnostic, meaning that its code can be compiled to run on various network hardware platforms. For example, in this work P4 runs on bmv2 switches.
3) Customizable packet processing: With P4, you can define the exact behaviour of how packets are processed through a network device. The advantage is that this allows for fine-grained control over network traffic and can improve network performance.
4) Faster development and testing: P4 code can be easily tested and simulated using software-based environments, which allows for faster development and testing cycles.
5) Standardization: P4 code can be written and run across various vendors' network hardware. This promotes interoperability and reduces vendor lock-in.

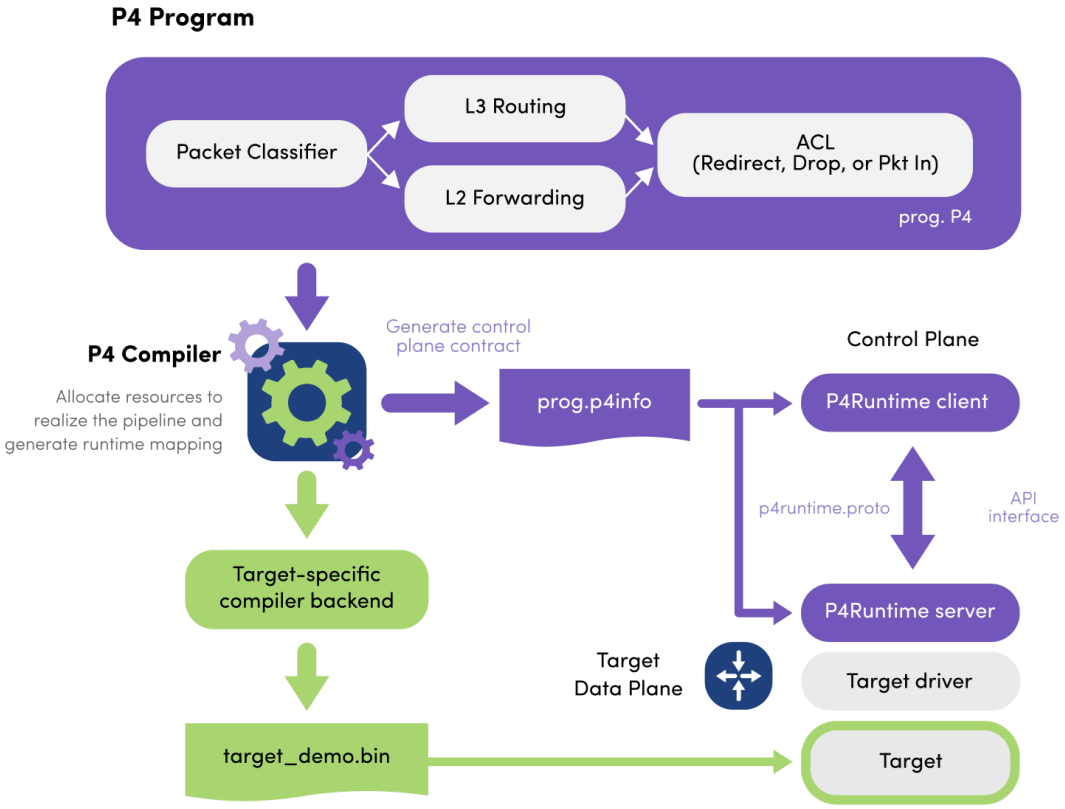


Figure 2 P4 workflow [3]

2.2. INT

In-band Network Telemetry (INT) [1] is a rapidly evolving technology that enables network operators to extract detailed, real-time network performance data, without the need of specialized monitoring tools or probes. It is a related specification of the P4 language. With INT, telemetry data is carried inside the network traffic, embedded into packets, providing an accurate and comprehensive view of network performance. This technology has become increasingly important as networks become more complex and dynamic, with the rise of cloud computing, network virtualization, and software-defined networking.

INT is exhaustively described in literature, and it is outside the scope of this paper. However here we are going to give the reader an overview based on the specification [1].

2.2.1. INT Application Modes

Original data packets are monitored and may be modified to carry INT instructions and metadata.

There are three variations based on the level of packet modifications.

- **INT-XD** (eXport Data): In this mode the INT nodes directly export metadata from their dataplane to the monitoring system without further packet modifications.
- **INT-MX** (eMbed instruct(X)ions): The INT Source node embeds INT instructions in the packet header, then the INT Source, each INT Transit, and the INT sink directly send the metadata to the monitoring system by following the instructions embedded in the packets. The INT Sink node strips the instruction header before forwarding the packet to the receiver. Packet modification is limited to the instruction header, the packet size doesn't grow as the packet traverses more Transit nodes.
- **INT-MD** (eMbed Data): In this mode both INT instructions and metadata are written into the packets. This is the classic hop-by-hop INT where 1) INT Source embeds instructions, 2) INT Source and Transit embed metadata, and 3) INT Sink strips the instructions and aggregated metadata out of the packet and (selectively) sends the data to the monitoring system. The packet is modified the most in this mode while it minimizes the overhead at the monitoring system to collate reports from multiple INT nodes.

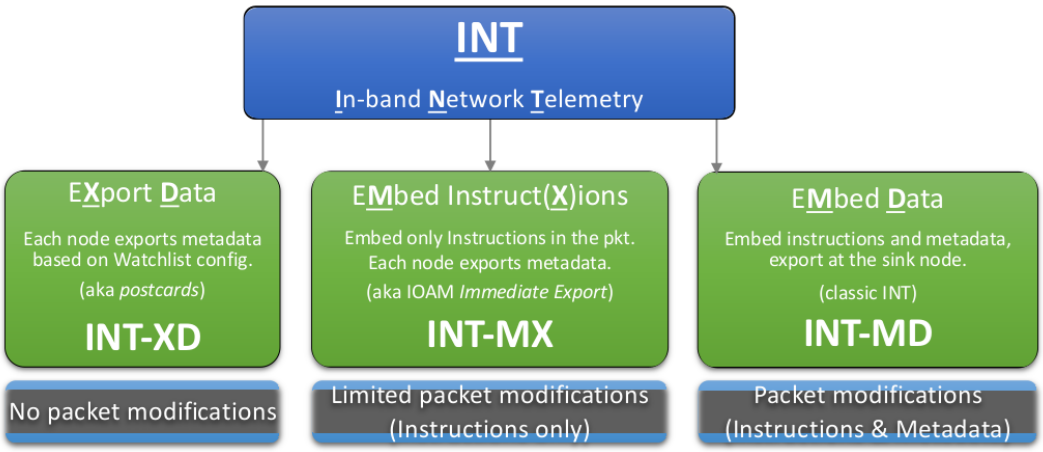


Figure 3 INT application modes

2.2.2. What is monitored

Even if it is technically possible to monitor any device information using the INT, it is a best practice to define a small set of metadata that can be made available on a wide variety of devices.

- Node id: This is the unique ID of an INT node to identify a node in the network.
- Ingress interface identifier: The interface on which the INT packet was received.
- Ingress timestamp: the local time in nanoseconds when the INT packet was received on the ingress port.
- Egress interface identifier: The interface on which the INT packet was sent out.
- Egress timestamp: The local time in nanoseconds when the INT packet was processed by the egress port.
- Hop latency: Time taken for the INT packet to be switched within the device.

There are other metadata defined in the INT specification; however, these are the ones that CHIMA utilizes.

2.2.3. Structure of an INT packet

The structure of INT is embedded within a TCP/UDP packet (this work focuses on UDP packets), and it can be divided into two sections: the INT Header and INT Data. The protocol structure is exhaustively described in the specification paper [1]. Here, we provide the reader with an overview of the protocol.

2.2.3.1. INT Header

The INT Header is added at the beginning of the INT packet and is responsible for carrying information about the transmission. Like other headers such as TCP/UDP, it is also responsible for ensuring the successful processing of the data. The INT Header consists of several fields, including:

- Type: the type of INT header, which can be either “switch metadata” or “hop-by-hop metadata”.
- DSCP: Differentiated Services Code Point (DSCP) value of the packet.
- Length: The length of the INT header.
- Maximum hop count: The maximum number of hops the packet can travel before the INT header is removed.
- Instructions: A set of instructions that specify the types of telemetry data that should be collected by network devices as the packet travels through the network.
- reserved bits for eventually further use.

2.2.3.2. INT Data

Here we have the actual telemetry data collected by our network devices. It consists in several fields, most of them covered in the section 2.2.2.

- Switch ID
- Ingress port
- Egress port
- Queue ID
- Timestamps
- Packet Length

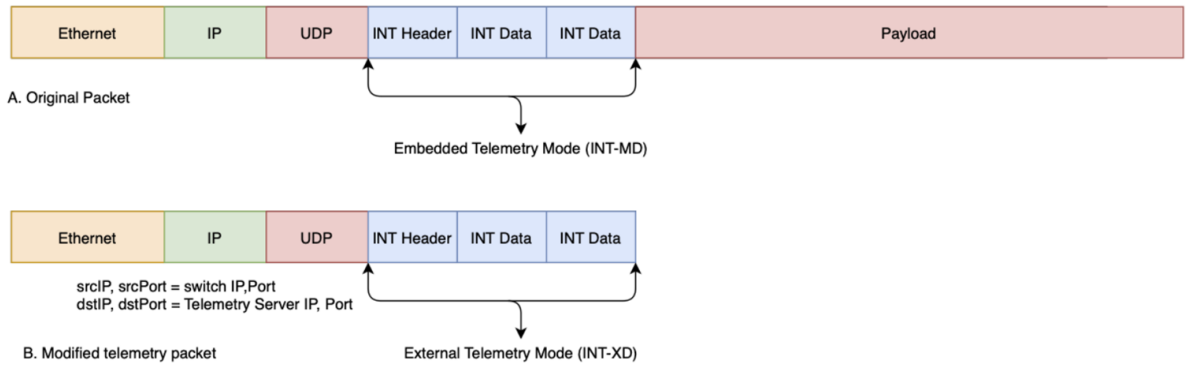


Figure 4 Packet carrying INT [4]

2.3. CHIMA

The use of Virtual Network Functions (VNFs) in modern networks has made it easier and faster to manage the provisioning of network services. However, this introduces a trade-off between flexibility and performance. Packet processing logic executed on regular CPUs may be less efficient than specialized hardware middleboxes in terms of throughput and power consumption. Recent advances in Programmable Data Planes, and In-Network Computing in particular, have shown that offloading sections of these services to programmable switches can eliminate this trade off, bringing back processing performance to the level offered by specialized hardware middleboxes.

One such technique is the use of In-band Network Telemetry (INT), which enables the real-time monitoring of flows and can be exploited for more than just error diagnosis or logging. It can provide real-time feedback on the performance of a service, allowing an orchestrator to take immediate action in response to congestion or faults.

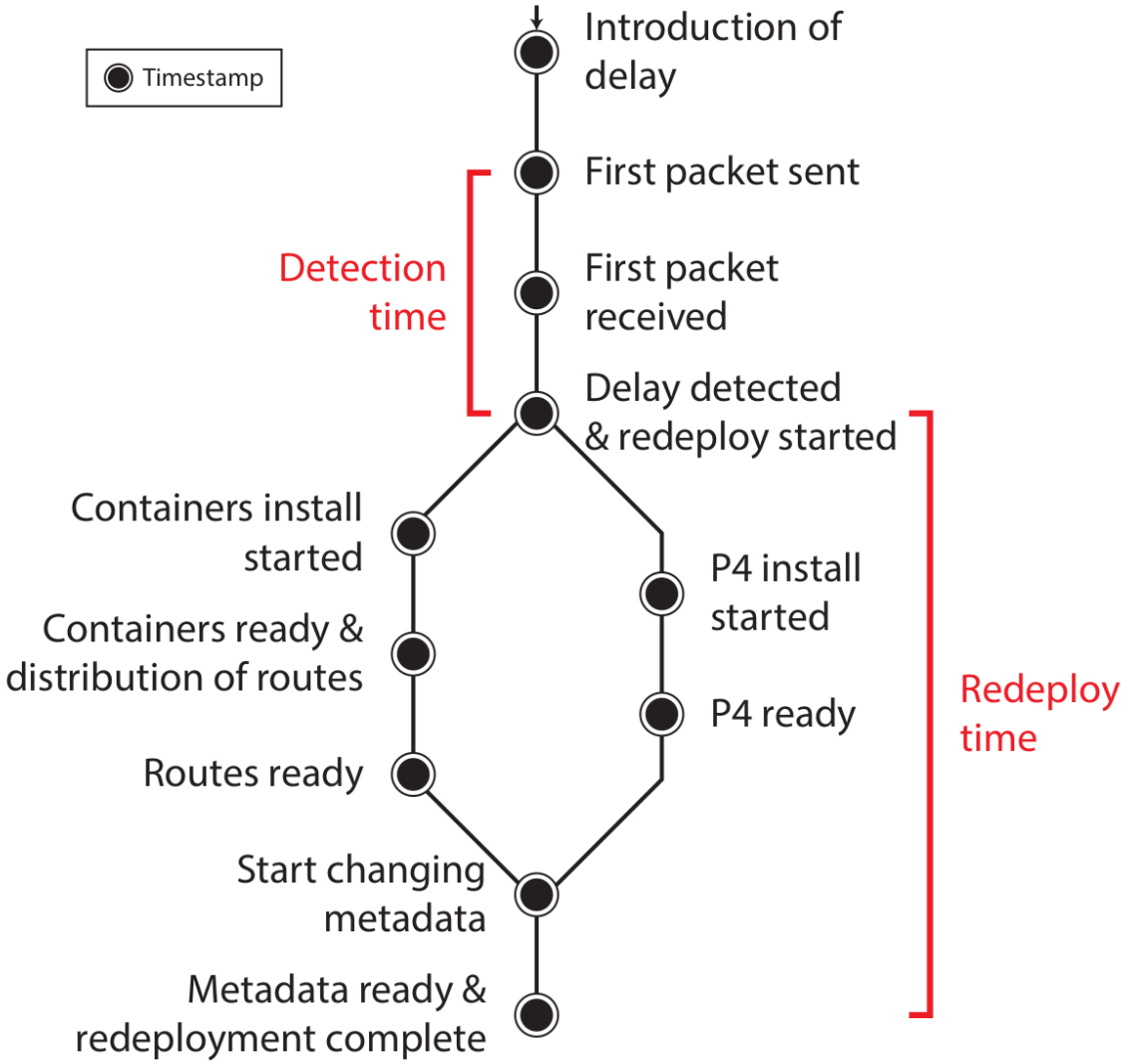


Figure 5: Representation of the steps needed to perform a redeployment of the service through the new path after a SLA violation

The system proposed by Battiston [2] consists of a chain of functions that can run on P4 switches and hosts capable of running Docker containers, each of them are associated with an SLA specifying the maximum end-to-end delay and jitter. The packet routing is managed by the ONOS SDN controller [5]. In the next page is depicted, as example, a service function chain mapped into a physical topology.

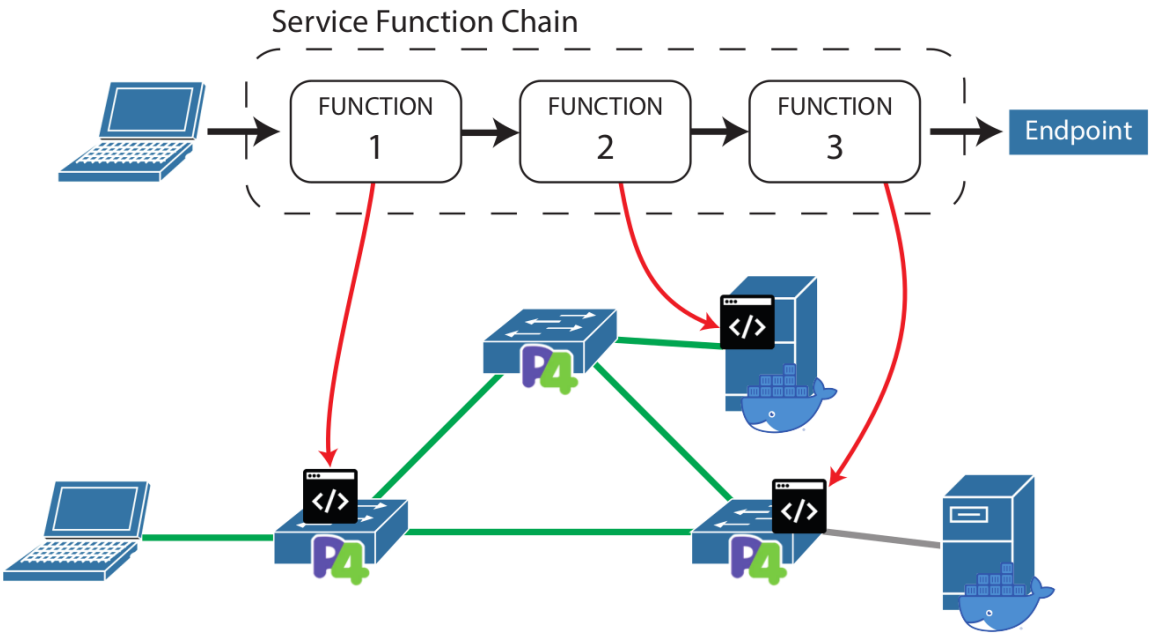


Figure 6: Example of a service function chain mapped into a physical topology [2]

2.3.1. The CHIMA framework

In this chapter, we will provide an overview of the CHIMA framework, as described in [2]. CHIMA consists of several components, which are illustrated in the next page:

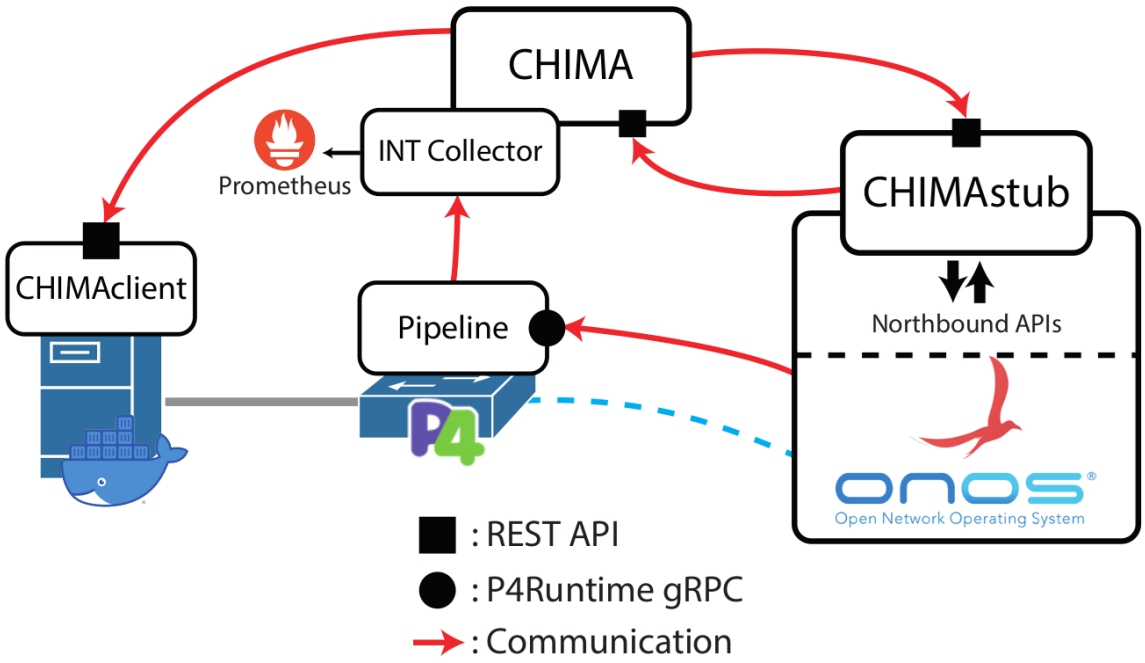


Figure 7: The previous CHIMA "ecosystem" [2]

1) CHIMAstub: It is a component able to expose the topology to the ONOS application and allows interactions with the network devices by means of an extension built in the ONOS REST APIs.

2) CHIMAclient: this is a process running on hosts enabling routing schemes exhaustively described in [2] outside the scope of this paper.

3) P4 pipeline: It is installed on all switches, and it is what enables the support to the In-band Network Telemetry v1.0 [1]

4) INT Collector: this is the component on which this paper leverage. It is responsible to extract the telemetry data generated by the P4 switches. In the CHIMA framework proposed by Battiston [2] it maintained a moving average of the measures (EWMA). In this paper we are proposing a new model based on an arithmetic average computed every "polling interval" defined by the user. In the previous version of CHIMA the INT collector sent data do the Prometheus client. We substituted it with InfluxDB and telegraf [6]. This new system which is the main topic of this work will be discussed in 2.3.4.

5) CHIMA: The CHIMA module is the manager of the system. It can:

"• Build and maintain an internal representation of the

network topology.

• Compute a deployment strategy based on the available topology information.

• Inject user provided P4 functions into the template pipeline.

• Carry out the installation of P4 pipelines through CHIMA Stub.

• Deploy Docker containers on hosts using Docker Compose.

• Compute performance metrics for communication paths, using data provided by the INT Collector, and compare them with user set requirements.

• Alter the placement of functions and reroute traffic accordingly in case of a requirement violation." [2]

2.3.2. The P4 Pipeline

In the following section, we will provide an overview of how the P4 switches are programmed with a specific pipeline to enable the deployment of Virtual Network Functions (VNFs).

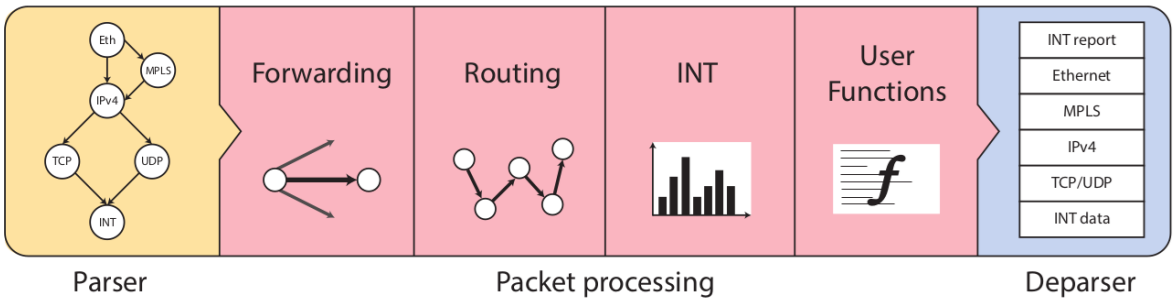


Figure 8: Stages of the template pipeline [2]

1) Forwarding: This pipeline is based on a built-in function "basic.p4" in ONOS and it is responsible of the forwarding mechanism.
2) Routing: This section of pipeline is managed by CHIMA and bypasses the usual forwarding mechanism by means of MPLS, as will be described later in the section 2.3.3.
3) INT: The implementation of the INT is based on the ONOS P4 pipeline int.p4. This is the part on which we made most of the improvements on the framework.
4) User's functions: The template pipeline includes a section in which user code can be inserted. Users provide functions to execute on packets in the form of control blocks with a predefined signature, such as the one presented in Listing 1. Control blocks allow the definition of arbitrary computation on the packet by defining tables, instantiating other controls, using registers, etc. The access to parsed headers and metadata is provided through the parameters of the control, enabling the user's function to change the content of the packet and even altering the predetermined egress port. When CHIMA determines one or more P4 functions of a service must be deployed on a switch, their code is injected, and will only be executed for packets of the correct service.

Setting up the pipeline: here the decision of what user functions must be deployed in each switch is made by means of an orchestrator.

```
control ingress(inout headers_t hdr,
                inout local_metadata_t local_metadata,
                inout standard_metadata_t standard_metadata) {

  apply {
    port_counters_ingress.apply(hdr, standard_metadata);
    port_meters_ingress.apply(hdr, standard_metadata);
    packetio_ingress.apply(hdr, standard_metadata);
    table0_control.apply(hdr, local_metadata, standard_metadata);
    host_meter_control.apply(hdr, local_metadata, standard_metadata);
    wcmp_control.apply(hdr, local_metadata, standard_metadata);
  }
}
```

Figure 9 Snippet of basic.p4 [2]

2.3.3. Routing

The correct routing of packets along the intended path and ensuring compliance with the SLA after deploying network functions is absolutely fundamental and can be considered one of the core aspects of the project developed by Battiston [2].

To achieve this goal MPLS was used in the deployment of the routing path, in particular the Segmentation Routing over MPLS was used. This MPLS implementation uses values called Segment Ids to instruct switches on the operations to execute on a packet by embedding them in the packet itself. The implementation used in CHIMA is the one described in the RFC8660.

In CHIMA the encapsulation of the MPLS packets is performed by the CHIMAclient module. The services are identified by a tuple of source and destination and if it is known, it means that the eBPF filter has a label stack that represent the series of segments used to implement its pre-computed path.

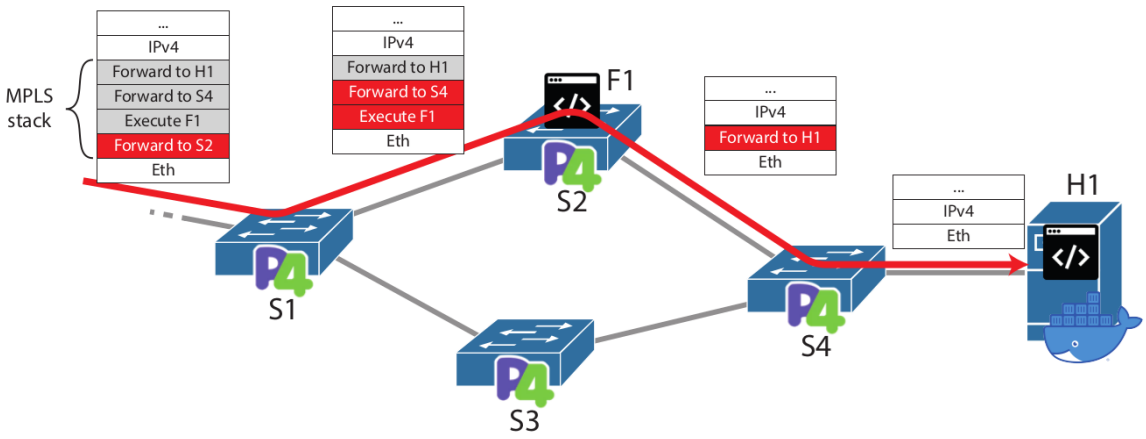


Figure 10: Example of routing by means of MPLS in the CHIMA framework [2]

The eBPF filter is the core of the "collector" in the CHIMA framework, and it is worth going into more detail about it in the next section since it is one of the parts in which we introduced improvements.

2.3.4. The INT Collector

Here we will describe the functions of the pre-built INT Collector. In Section 4.1, we will detail the original contributions and improvements introduced.

The INT Collector in CHIMA is composed of an eBPF filter and a Python application that interacts with it using the BCC (BPF Compiler Collection) library [7].

"The CHIMA process includes an INT collector as one of its modules, which has been adapted form an existing eBPF based implementation [...]. The source of the obtained values is identified by the pair of IDs of the switches at the two ends of a link. For each incoming INT packet, the collector computes the packet delay *dp*. The collector also maintains a running Exponentially Weighted Moving Average (EWMA) of the delay *d*, which is updated for each incoming packet as follows:

$$d \leftarrow (1 - \alpha)\, d + \alpha dp$$

The average value of jitter is also measured following the same procedure. Periodically, at each polling interval, the running values of delay and jitter are also conveyed to a server of the Prometheus monitoring system. The α parameter acts as a smoothing factor and can be modified by the user when running CHIMA to tune the response to transient variations of the metrics." [2]

As we are going to describe in section 4.1, we modified this feature by eliminating the need for the use of EWMA and instead using arithmetic mean to enhance performance.

2.3.4.1. eBPF Filter

eBPF (extended Berkeley Packet Filter) is a technology that allows for dynamic and efficient tracing and monitoring of system and application behaviour in Linux-based operating systems. eBPF achieves its flexibility and efficiency by providing a way to inject and execute small snippets of code directly into the kernel, known as eBPF programs. These programs can be dynamically loaded and unloaded, enabling the creation of custom tracing and monitoring functionality without the need to recompile or modify the kernel itself. The fact that these programs run directly in the kernel space is an incredible advantage in our environment because it allows the processing of INT packets arriving at very high frequencies without interruptions.

eBPF programs are written in a restricted subset of the C programming language and are executed in a secure virtual machine within the kernel. This allows eBPF programs to safely interact with the kernel and other system resources, without compromising system stability or security.

```
static inline
uint32_t compute_ewma_unsigned(uint32_t measure, uint32_t avg_measures) {
   return (measure >> SHR_EWMA) + avg_measures - (avg_measures >> SHR_EWMA);
}

static inline
int32_t compute_ewma_signed(int32_t measure, int32_t avg_measures) {
   return (measure >> SHR_EWMA) + avg_measures - (avg_measures >> SHR_EWMA);
}
```

Figure 11: eBPF code snippet, EWMA calculation of latency and jitter

2.3.4.2. Userspace Python script

In this section, we will provide an overview of the Python script that interacts with the eBPF.

The aim of this script is defined in the function *send metrics*. This function is responsible for starting an HTTP server on TCP port 8000 that is used by Prometheus to collect data, including the latency and jitter data extracted by the eBPF filter previously discussed. The function also calls another function defined in a separate script to check if the triggers, which are SLA parameters such as the maximum latency and jitter, are exceeded. If the triggers are exceeded, the function triggers a redeployment of the VNFs.

```
49      def send_metrics(self, network, int_funcs):
50          #Prometheus exporter
51          start_http_server(8000)
52          gauges = self.gauges
53
54          #logging.info("POLLING_INTERVAL is %d" % (self.POLLING_INTERVAL))
55          #logging.info("Printing stats, hit CTRL+C to stop")
56          while True:
57              try:
58                  current_time = time.time()
59                  wait_time = self.POLLING_INTERVAL - current_time % self.POLLING_INTERVAL
60                  time.sleep(wait_time)
61
62                  for link_key, link_metrics in self.link_metrics_map.items():
63                      label = str(link_key.switch_id_1)+"_"+str(link_key.switch_id_2)
64                      latency_label = "l_" + label
65                      jitter_label = "j_" + label
66
67                      if latency_label not in gauges:
68                          gauges[latency_label] = Gauge(latency_label, 'Latency of the link')
69                          # logging.debug("New gauge: LINK_KEY = (%u,%u) LATENCY = %u" \
70                          # % (link_key.switch_id_1, link_key.switch_id_2, link_metrics.latency))
71
72                      if jitter_label not in gauges:
73                          gauges[jitter_label] = Gauge(jitter_label, 'Jitter of the link')
74                          # logging.debug("New gauge: LINK_KEY = (%u,%u) JITTER = %u" \
75                          # % (link_key.switch_id_1, link_key.switch_id_2, link_metrics.jitter))
76
77                      gauges[latency_label].set(link_metrics.latency)
78                      gauges[jitter_label].set(link_metrics.jitter)
```

Figure 12: Snippet of the userspace python script

2.4. InfluxDB and Telegraf

In this chapter, we introduce two software: InfluxDB, an open-source time-series database, and Telegraf, a GO-Lang agent to collect metrics from a variety of sources.

2.4.1. InfluxDB

InfluxDB is an open-source time-series database that is designed to handle high volumes of time-series data, such as metrics, events, and logs. It is built with performance, scalability, and ease of use in mind, and is widely used in a variety of industries for real-time monitoring, analytics, and IoT applications. InfluxDB uses a schema-less data model, which means that it doesn't use the traditional relational model of tables and columns. Instead, data is organized into measurements, tags, and fields, providing a flexible and efficient way to store and query time-series data.

We can list the key features of InfluxDB as follows:

1. High performance: InfluxDB can handle a large amount of time-series data while ensuring low latency and high throughput.
2. Query language: InfluxDB comes with a query language called InfluxQL, which is specifically designed for time-series data. It allows filtering, aggregation, and downsampling of data.
3. Built-in visualization: InfluxDB provides a visualization tool called "Chronograf" that is integrated into the database, allowing for real-time dashboards for monitoring and data analysis.
4. Integration: InfluxDB can be easily integrated with other tools such as Kubernetes, Grafana (a visualization tool), and Telegraf.

2.4.2. Telegraf

Telegraf instead is an agent written in the programming language GO to collect and report data from a variety of sources. Telegraf is plug-in based meaning that it works by means of plugins. These plugins can be divided into four groups:

- Output plugins: they are responsible to send the collected metrics to a specific service, like a SQL database, a text file or, and this is the case, InfluxDB.
- Input plugins: they are responsible to collect the metrics before sending them to the output plugins. For this work we used the "Socket Listener Input Plugin".
- Processor plugins: they are responsible to manipulate, filter and decorate the metrics before sending them to the output plugins.
- Aggregator plugins: they are responsible to aggregate metrics, creating, for example: minimum, maximum, average of the metrics.

2.4.2.1. Why Telegraf

The use of Telegraf offers a lot of advantages in our system. For example:

1. Easy to use: We shouldn't neglect the user-friendly design of Telegraf, which allows users to easily change plugins and configure them using TOML file format. It's also worth considering that in most cases, there will be a pre-built plugin available to meet the user's needs.
2. Flexible and modular: Telegraf's support for over 200 plugins makes it a highly flexible and modular tool. It allows users to collect, process, aggregate, and output data from a variety of sources in a customizable way. The availability of numerous plugins enables users to choose the ones that best fit their needs and to create custom data pipelines. This flexibility and modularity make Telegraf a powerful tool for data processing and management in various use cases.
3. Low Resource Usage: Telegraf is designed to be extremely fast and efficient, with a low resource usage. This makes it feasible to run on edge-devices, such as Docker containers running on hosts.
4. Integration: As previously mentioned, Telegraf can integrate with other tools such as InfluxDB, Prometheus, and Grafana, as well as with the Linux kernel to extract various information such as CPU usage and disk I/O. This enables Telegraf to be a versatile agent that can be used in a variety of contexts and workflows, making it a valuable tool for monitoring and collecting data.
5. Real-time monitoring: Thanks to its high efficiency and low resource usage, Telegraf enables real-time monitoring of data, allowing users to quickly identify and respond to issues as they occur.

Overall, Telegraf is a powerful and versatile tool that can help you collect and process data from a wide range of sources and send it to a variety of destinations. Its ease of use, flexibility, and low resource usage make it a popular choice for monitoring and data collection in a variety of industries. This is why we consider this product the most feasible solution to be implemented in our system.

3 State of the Art

In the past, telemetry data was collected using out-of-band methods such as SNMP (Simple Network Management Protocol), NetFlow, and packet capture systems like Wireshark or TCPdump. However, as networks became more complex and dynamic, traditional monitoring techniques were unable to keep pace with the demands of modern applications. This led to the emergence of the idea of embedding telemetry data in the network itself, which has become more popular in recent years.

3.1. Related work

CHIMA was not the first work aimed at determining the optimal path for communication in a network. It was developed based on existing literature, which we will discuss below. Addis et al. [8] used ILP (Integer Linear Programming) to optimize the deployment of VNFs, and related work was also done by Khoshkholghi et al. [9] using heuristic-based algorithms.

> "Focusing on service chaining, we formulate a Mixed-Integer Linear Programming problem with the objectives of load balancing as well as reducing drop rate." [9]

and Mechtri et al. [10] proposed an SFC orchestration framework that takes the monitoring of the deployed service into consideration but does not propose solutions to guarantee its performance.

> "This article presents the SFC orchestration framework, an implementation, and a qualitative and quantitative evaluation of its components in an experimental environment." [10]

However, these previous works only considered SFCs composed of Virtual Network Functions (VNFs) that target homogeneous compute architectures. In contrast, CHIMA [2]

"Supports the deployment of heterogeneous SFCs that take advantage of programmable data planes and significantly increase the achievable throughput" [2].

Also, other approaches have been studied, such as the use of Hyper4 [11] and P4Visor [12]. However, CHIMA instead leverages FOP4 [13], a mininet fork that introduces support for P4 switches and SmartNICs.

In the following chapter, we will discuss the improvements implemented into the CHIMA framework.

4 System Model

The system model we are going to use for the tests is the same as that proposed by Battiston [2], but with some improvements in certain components which will be described in more depth in this chapter.

4.1. The INT Collector

In this paragraph, we will analyse the INT collector of CHIMA and the improvements made to it in more depth.

Inband Network Telemetry (INT) has the potential to revolutionize the way we monitor and manage networks by providing a more holistic and proactive approach to network performance management. The INT collector proposed by Battiston [2] was derived from a previous eBPF implementation [14]. In this implementation, each value is identified by a couple of IDs associated with the two switches at the ends of the single link. For each incoming packet, the collector computes the packet delay.

In the original implementation proposed by Battiston, the collector maintains an Exponentially Weighted Moving Average (EWMA) of the delay updated for each incoming packet. However, in this work, we modified this mechanism and switched to an arithmetic average.

The choice to use EWMA in the original solution was induced by the fact that eBPF, due to its extremely high efficiency in executing code at the kernel level, supports signed division of integers but not unsigned division of integers. The EWMA avoids the use of divisions entirely, but it has infinite memory, so the measurements obtained in a previous polling interval may perturb the further measurements in other polling intervals.

To address this issue, a new implementation of the way measurements are averaged was proposed. Instead of sending every measure to CHIMA, every time a measure is received, a counter is updated, and a variable is updated to store the sum of the measurements received, allowing an arithmetic average to be performed every polling interval. Additionally, the new implementation calculates the new minimum and maximum latency, as well as the minimum and maximum jitter, according to RFC3393 [15], for every packet arrival. The average is calculated with a workaround in user space, at every polling interval, by a modified version of a Python script developed in [2], which interacts with the eBPF using APIs developed by BCC [7]. This allows for the avoidance of the limitation of eBPF regarding the division of unsigned integers. The modified Python script is also responsible for sending the metrics to InfluxDB, by means of Telegraf [6].

4.1.1. The new eBPF implementation

The previous eBPF collector proposed by Battiston has this interesting code snippet for the calculation of latency and jitter.

```
if (link_metrics_ptr == NULL || last_latency_ptr == NULL) {
   struct link_metrics_t link_metrics = {.latency = latency, .jitter = 0};
   uint32_t new_latency = latency;
   link_metrics_map.update(&link_key, &link_metrics);
   last_link_latency_map.update(&link_key, &new_latency);
   } else {
      jitter = latency - *last_latency_ptr;
      link_metrics_ptr->latency = compute_ewma_unsigned(latency,
                                 link_metrics_ptr→latency);

      link_metrics_ptr->jitter = compute_ewma_signed(jitter,
                                                     link_metrics_ptr→jitter);

      *last_latency_ptr = latency;
        }


     }
     return XDP_DROP;
   }

   return XDP_PASS;
}
```

Figure 13: The previous CHIMA's collector implementation on metric calculation [2]

This is the code snippet that we modified. The functions *compute_ewma_unsigned* and *compute_ewma_signed* are portrayed in figure #11. The code initializes the pointers if necessary and calculates the jitter as the difference between the new latency and the last latency. Then, it saves the EWMA latency value and the jitter inside the map *link_metrics* using a pointer to the eBPF map. The value of the last latency is also updated.

As previously mentioned, the choice to use EWMA in the original CHIMA architecture was induced by the limitation of eBPF, which is not capable of performing divisions with unsigned integers (which the jitter requires). We proposed to substitute this snippet (Figure 13) with the following code snippet (Figure 14), which is more complex but more efficient in terms of detecting SLA violations, as we will see in the next chapter.

```
if (link_metrics_ptr == NULL || last_latency_ptr == NULL)
        {
          struct link_metrics_t link_metrics = {.latency = latency, .jitter = 0};
          uint32_t new_latency = latency;
          link_metrics_map.update(&link_key, &link_metrics);
          last_link_latency_map.update(&link_key, &new_latency);
        }
        else
        {
          jitter = abs(latency - *last_latency_ptr);
          link_metrics_ptr→jitter = jitter;

          if ((link_metrics_ptr→jitter_min == 0) && (link_metrics_ptr→jitter_max                == 0)
&& (link_metrics_ptr→latency_min == 0) &&
(link_metrics_ptr→latency_max == 0))
          {
            link_metrics_ptr→jitter_min = jitter;
            link_metrics_ptr→jitter_max = jitter;
            link_metrics_ptr→latency_min = latency;
            link_metrics_ptr→latency_max = latency;
          }
          else
          {
             if (jitter < link_metrics_ptr→jitter_min)
               link_metrics_ptr→jitter_min = jitter;
             else if (jitter > link_metrics_ptr→jitter_max)
                  link_metrics_ptr→jitter_max = jitter;

             if (latency < link_metrics_ptr→latency_min)
               link_metrics_ptr→latency_min = latency;
             else if (latency > link_metrics_ptr→latency_max)
                  link_metrics_ptr→latency_max = latency;
          }

          link_metrics_ptr→counter += 1;
          link_metrics_ptr→tot_latency += latency;
          link_metrics_ptr→tot_jitter += jitter;
          *last_latency_ptr = latency;
```

Figure 14: Snippet of the new eBPF collector [2]

As we can see, we first initialize the pointers if needed. If not, we jump to the else block of code. First, we calculate the jitter and save this value into the eBPF map. Then, we check if the minimum and maximum jitter and latency are initialized. If not, meaning them equal to zero in eBPF, we update them with the current values of jitter and latency. If they are initialized, we jump to check if the new values of jitter and latency are the new minimum or maximum. At the end of the snippet, we increase a counter variable, which is also the number of packets in the polling interval. We update the

total latency and the total jitter. The value referenced by the last latency pointer is also updated.

4.1.2. The new Python implementation

As we stated before, the eBPF collector alone is not sufficient to perform all the required tasks due to the limitation with unsigned integer division. Therefore, we are proposing a new implementation of the Python script responsible for extracting latency and jitter values from eBPF by means of the *bcc* Python library.

```
for link_key, link_metrics in self.link_metrics_map.items_lookup_and_delete_batch():
            label = str(link_key.switch_id_1)+"_"+str(link_key.switch_id_2)
            latency_label = "l_" + label
            jitter_label = "j_" + label
            if latency_label not in gauges:
               gauges[latency_label] = Gauge(latency_label, 'Latency of the \                link')

            if jitter_label not in gauges:
               gauges[jitter_label] = Gauge(jitter_label, 'Jitter of the link')

            ##Send metrics to telegraf socket_listener input plugin
            if link_metrics.counter > 1:
            sock.sendto(json.dumps({'metric_name': 'latency %s_%s' % \
(str(link_key.switch_id_1), str(link_key.switch_id_2)), 'latency_mean':\
link_metrics.tot_latency / link_metrics.counter, 'latency_max': \
link_metrics.latency_max, 'latency_min': \
link_metrics.latency_min}).encode(),('localhost', 8094))
            sock.sendto(json.dumps({'metric_name': 'jitter %s_%s' % \
(str(link_key.switch_id_1), str(link_key.switch_id_2)), 'jitter_mean': \
link_metrics.tot_jitter / (link_metrics.counter - 1), 'jitter_max': \
link_metrics.jitter_max, 'jitter_min': \
link_metrics.jitter_min}).encode(),('localhost', 8094))
            sock.sendto(json.dumps({'metric_name': 'contatore', 'value1': \
link_metrics.counter}).encode(),('localhost', 8094))
```

Figure 15: Snippet of the new python script userspace

In the updated script, we have introduced the *items_lookup_and_delete_batch* function from the bcc library, which allows us to iterate through the eBPF map for each link_key pair (consisting of the IDs of two switches or Docker containers) and their corresponding *link_metric* values (jitter, latency, and associated counter). As the function iterates through the data, it calculates the arithmetic mean of the jitter on-the-fly, along with the minimum and maximum latency and jitter values. We have also implemented a specific plugin in the system that sends this data to the Telegraf agent using a UDP socket. Once the iteration process is complete, the eBPF map is cleared, making it ready for the next iteration. A check on the SLA is performed, and if violated,

the redeployment Python function is called, which is implemented in other components of CHIMA.

4.2. InfluxDB and Telegraf

As introduced in the previous chapters, InfluxDB and Telegraf are the two softwares implemented to substitute the Prometheus webservice to collect the metrics. These two softwares were installed on the same virtual machine used to perform the tests, but they don't run inside Docker containers like CHIMAclient.

4.2.1. Telegraf

The Telegraf agent is started before the tests by loading a particular configuration file that we implemented for our system:

```
[global_tags]

[agent]

  interval = "10ms"
  round_interval = true
  metric_batch_size = 1000
  metric_buffer_limit = 10000
  collection_jitter = "0s"
  flush_interval = "10s"
  flush_jitter = "0s"
  hostname = ""
  omit_hostname = false
  precision = "1ns"

[[outputs.influxdb_v2]]
  urls = ["http://localhost:8086"]
  token= "$INFLUXDB_TOKEN"
  organization="PoliMi"
  bucket="COLLECTOR"

[[outputs.file]]
  ## Files to write to, "stdout" is a specially handled file.
  files = ["stdout", "../../output"]
  data_format = "carbon2"

[[inputs.socket_listener]]
  service_address = "udp://:8094"
  data_format = "json"
  json_name_key = "metric_name"
```

Figure 16: the used Telegraf configuration file

the [agent] section defines the basic configurations of the Telegraf agent.

- Interval: the frequency at which the metrics are collected. 10ms allows the use of polling intervals down to 10ms, but it can be regulated lower or higher depending on the necessity of the system.
- round_interval: Whether to round the collection interval to the nearest whole number of seconds.
- metrics_batch_size: The maximum number of metrics to buffer before sending them to the output plugins.
- metric_buffer_limit: The maximum number of metrics to buffer in memory before dropping them. This is to avoid an overload of the system.
- collection_jitter: The maximum amount of jitter to add to the collection interval to spread out metrics collection.
- flush_interval: The maximum amount of time to wait before flushing the metric buffer.
- flush_jitter: The maximum amount of jitter to add to the flush interval to spread out metric flushes.
- Hostname: The hostname to include in the metric data. If left blank, the system hostname will be used.
- omit_hostname: Whether to exclude the hostname from the metric data.
- Precision: The precision to use for timestamps in the metric data.

[[outputs.influxdb_v2]] is the plugin responsible to send the metrics to InfluxDB.

- Urls: The URL of the InfluxDB webservice.
- Token: The API access token. It is written as local variable in compliance with the cybersecurity policy.
- Organization: The name of the organization we choose.
- Bucket: This is the InfluxDB "bucket" in which all the data will be written

[[outputs.file]] is a simple plugin writing all the data into a text file. We choose to introduce this plugin because InfluxDB allow the visualization of data with at least 1 second of precision. Considering we have data collected by Telegraf every hundreds of millisecond will be worth to have a reference for all of them.

[[inputs.socket_listener]] this is the input plugin we decided to use. It collects the metrics acting as an UDP socket listening for datagrams.

- service_address: The address to listen on for incoming metrics data.
- data_format: The format of the incoming metric data. In this case, it is JSON.
- json_name_key: The name key in the JSON data that contains the metric name.

4.3. InfluxDB

After installing InfluxDB, we started the service and the web server to access the web-based graphical interface. Through the interface, we created a bucket, which serves as a container for our data, an organization name, and an API key, which is necessary to interact with InfluxDB through Telegraf. For security reasons, we stored the API key in the *etc/environment* file instead of directly in the Telegraf configuration file. With the provided Telegraf configuration file (shown in Figure #16), the system was then able to collect the metrics and send them to InfluxDB.

5 Validation of results

In this chapter, we are going to describe the test we performed, following the methodology applied by Battiston [2].

5.1. Methodology

The topologies described in [2] are depicted in the next page:

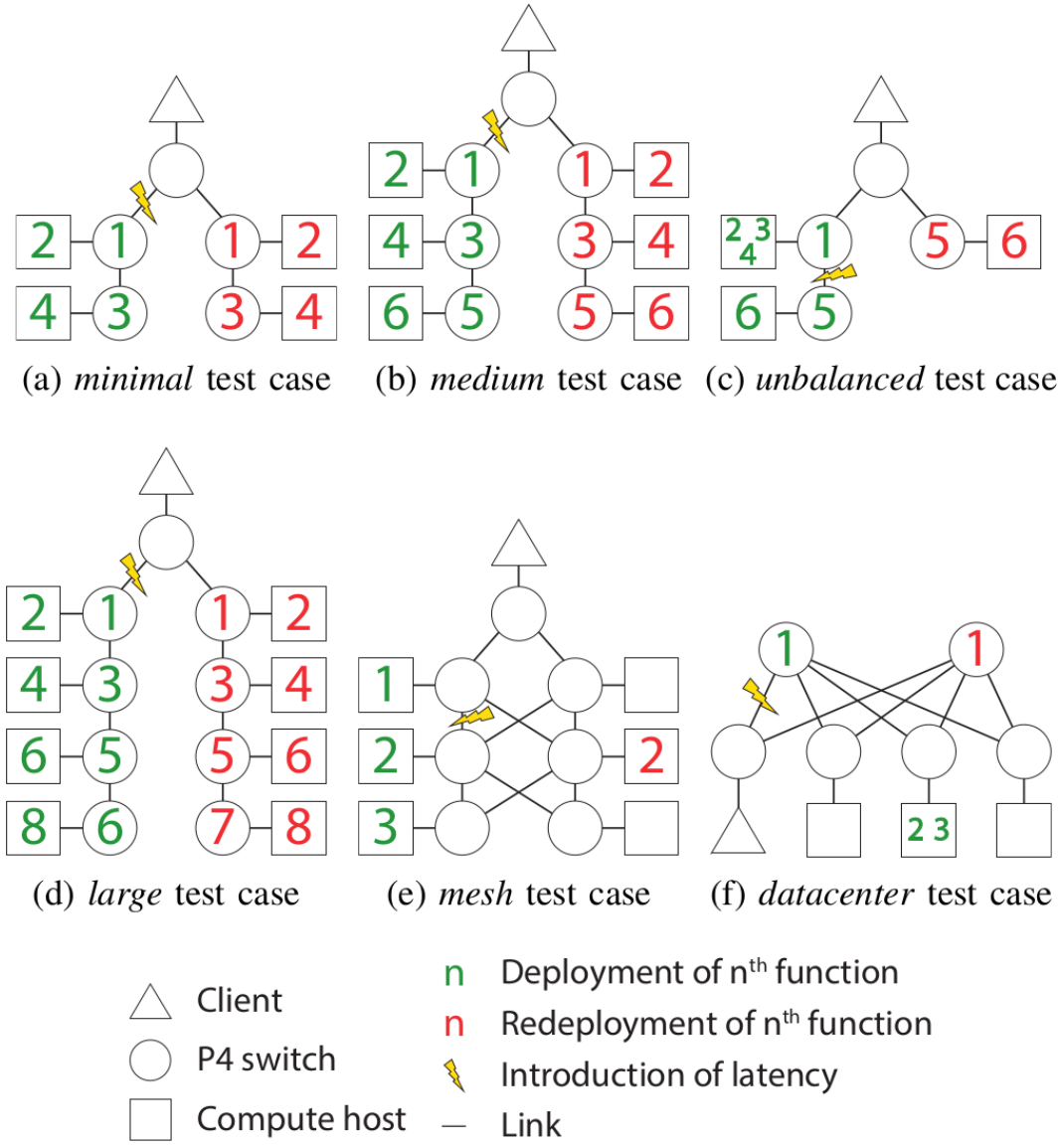


Figure 17: Topologies available for tests in the CHIMA framework [2]

We evaluated the new proposed CHIMA framework using the minimal test case described in [2].

The test cases were simulated with FOP4 [13] using bmv2 instances as switches. To evaluate detection and redeployment time, we used the built-in instruments developed by Battiston in the previous framework [2]. Additionally, we examined how the system responds to a traffic load with a step increase and sinusoidal variation with a period equal to 10 times the sampling rate.

The tests were performed on a clean installation of Ubuntu 20.04 LTS in a virtual machine with 8 dedicated virtual cores and 16GB of virtual RAM. The virtual machine ran on a host machine with a Ryzen 5 3600 and 32GB of RAM.

We use the same methodology used in [2]. At the beginning of each experiment FOP4 sets up the simulated topology and the deploys the CHIMA service (MPLS routing, INT Collector, CHIMAclients). The service runs for several seconds, then the latency of the target link is artificially increased in order to simulate a failure to match the target SLA causing a redeployment of the service in the backup path. A dedicated script is responsible to save timestamps of the events to calculate the time needed to redeploy the service and the time to detect the failure.

5.2. Numerical results

In this chapter we are going to presents the results we obtained in various simulated behaviours.

It is worth remembering that in the previous CHIMA implementation, we were assuming that the need for an EWMA to create an average of the results was impacting the overall performance in terms of detecting an SLA violation. The problem of the delay introduced by the EWMA was studied by Battiston and can be summarized in the following picture:

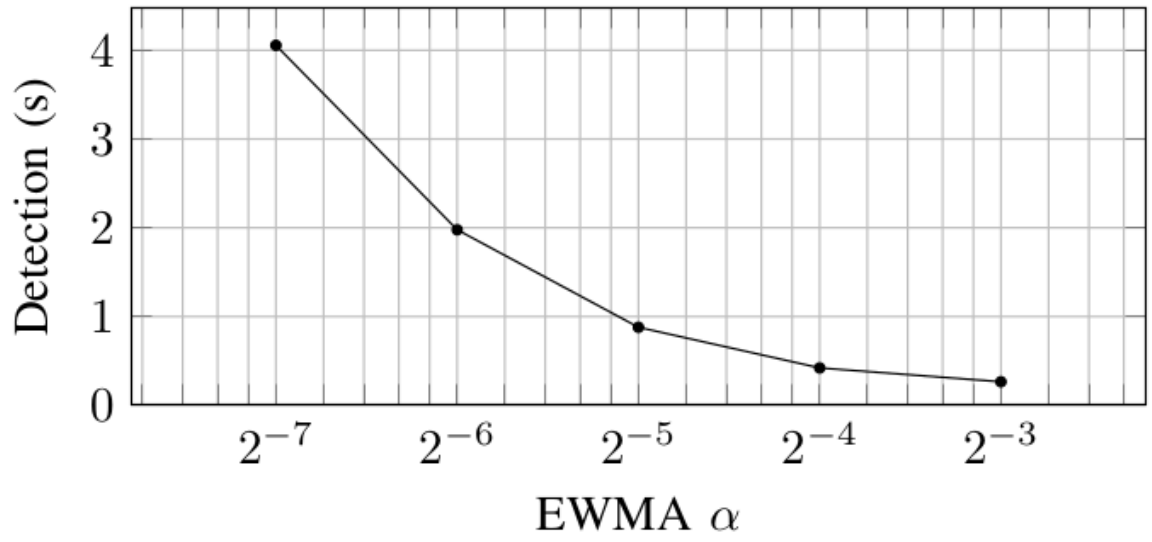


Figure 18: Detection time with minimal topology and polling interval 0.1s in function of the EWMA coefficient [2]

As we can see, the lower the EWMA coefficient, the shorter the detection time.

5.2.1. Detection time

In this section, we are going to compare the mean detection time in the minimal topology of the tests performed by Battiston with the test performed using the new

CHIMA implementation. The new tests were performed by conducting 15 tests for every polling interval, and then the results were arithmetically averaged.

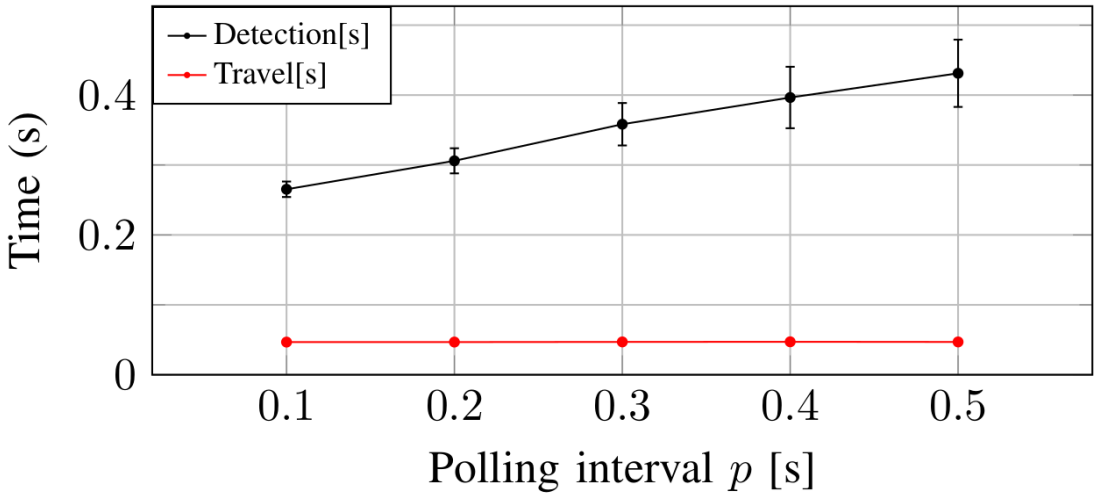


Figure 19: mean detection delay with the previous CHIMA framework [2]

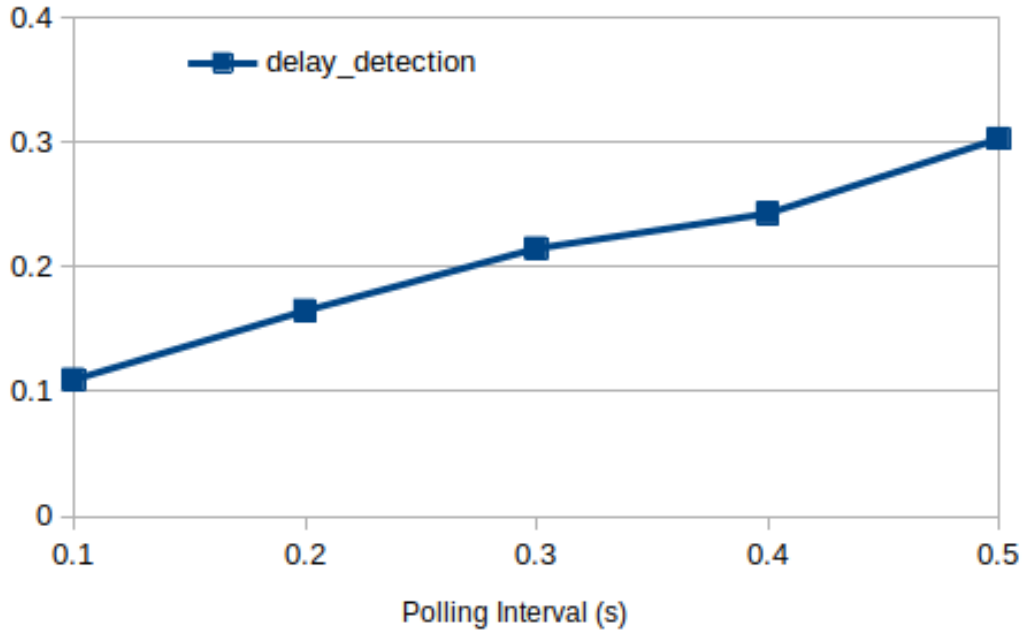


Figure 20: mean detection delay with the new CHIMA framework

As we can see, for every polling interval the mean detection time is 150ms lower with respect to the previous CHIMA implementation with EWMA.

5.2.2. Containers saturation

In this paragraph, we will conduct new tests on the newly proposed CHIMA framework. The aim of these tests is to evaluate how the system performs under different traffic loads. To accomplish this, we developed several Python scripts to

generate various types of traffic. The tests were performed with a fixed polling interval of 1s to provide better visualization of the time series.

5.2.2.1. Step traffic according to Poisson arrivals

Here we are going to propose the results of how the system responds to a stepwise increase in traffic generated according to a Poisson process in the minimal topology.

For this test I used a python script I developed for the scope:

```
import socket
import time
import termios
import sys
import select
import random
from math import exp

IP = "10.0.0.2"
PORT = 12345
MESSAGE = b"A" * 1000 #1000 bytes

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
print ("WELCOME!!! PRESS THE r KEY TO REDUCE THE WAITING TIME BY A FACTOR 0.75")
reduction_factor=0.75
time.sleep(5)
print ("Current frequency is 10Hz")
def message_sending():
 lambd=0.1
 while True:
  for i in range(1000):
   sock.sendto(MESSAGE, (IP, PORT))
   if sys.stdin in select.select([sys.stdin], [], [], 0)[0]:
       line = sys.stdin.readline()
       if line[0] == 'r':
          lambd = lambd * reduction_factor
          print ("New frequency is ", 1/lambd, "Hz")
   time.sleep(random.expovariate(1/lambd)) #sleep for a mean of lambd seconds before sending
the next packet (Poisson distribution)

termios.tcflush(sys.stdin, termios.TCIFLUSH)
termios.tcsetattr(sys.stdin, termios.TCSADRAIN, termios.tcgetattr(sys.stdin))
message_sending()
```

Figure 21: Python script to send UDP traffic according to a Poisson statistic.

To perform the test, we reduced the waiting time between two consecutive packets by a factor of 0.75 every 10 seconds using the built-in function in the script. The results are shown in the following figure.

Figure 22: Latency of the first docker container (values in microseconds)

Figure 23: step increase of the number of packets sent per second

As we can see, after a certain threshold (in our system, more than 350 packets/second), the value of latency on link 101_101 tends to increase significantly while the other links remain within a certain range. This can be easily explained by looking at the minimal topology (Figure 13.a) and considering the design of the CHIMA framework. The traffic entering the network through a specified port (in this case, UDP 12345) is sent through the following path: 1-2-1-3-4. Nodes 2 and 4 are represented in Figure 13 as the nodes 101_101 and 102_102, respectively, and they are the Docker images running the CHIMA client. After a certain threshold of traffic, the Docker containers cannot handle it and start to send the traffic to the next CHIMA client (in this case, 102_102) at a rate lower than the incoming traffic ratio. This causes an increase in latency in node 2 (101_101 in InfluxDB) but normal behaviour in node 4 (102_102 in InfluxDB) because it receives packets from the first CHIMA client at a rate that is slightly below the maximum it can handle, considering every Docker container has the same resources assigned. We have seen that this behaviour is repeated with the medium and large topologies, saturating just the first container but not the following ones.

In the following graph, we can observe the behaviour of the second Docker function. The cyan line represents the number of packets per second, while the purple line represents the second Docker function. In the first stage, the mean packets per second were set to 10 packets per second. Then, it was increased to 420 packets per second, which caused saturation in the first Docker container but not in the second one.

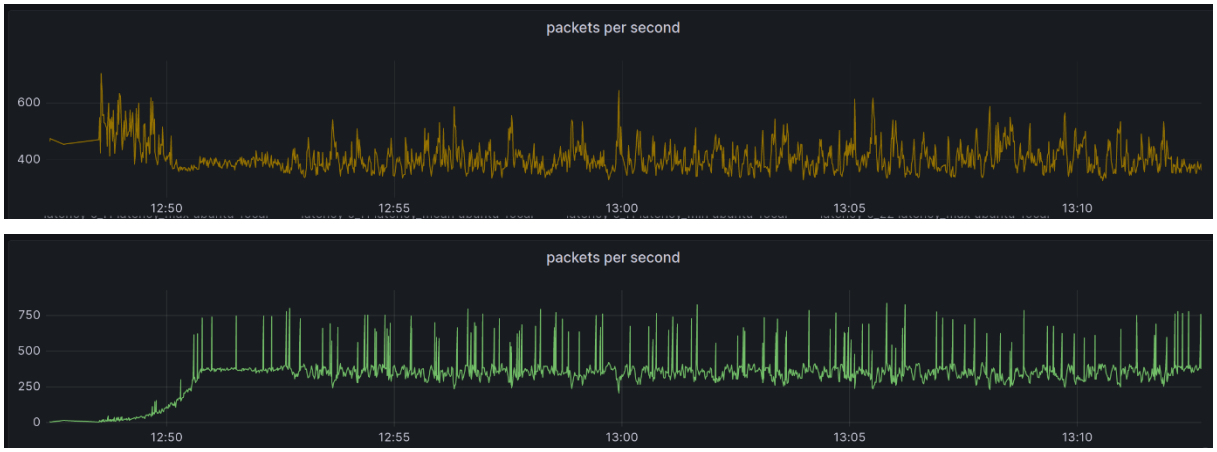


Figure 24: Latency of the second docker container (yellow line, values in microseconds) in function of a step increase in traffic (green line, values in packets per second)

In this graph, as soon as the traffic overcome a certain threshold, even if the first docker contain saturates, the other functions (e.g. the second CHIMAclient docker container) remain quite constant in values. This confirms what we observed in the previous graphs, namely that the first docker container is saturated and cannot handle the incoming traffic, while the other containers can still operate normally. Also, the latency of the second docker container tends to be more stable after the saturation of the link 101_101.

5.2.2.2. Sinusoidal traffic

Here we are going to present the results of a test where we examined how the CHIMA system responds to a sinusoidal variation in traffic with a peak value near the saturation threshold of our specific system in the minimal topology. The test was conducted with a polling interval of 1s and a sinusoidal period of 10s.

We used a Python script to generate a sinusoidal traffic pattern where the traffic intensity varied between 1 packet per second to 360 packets per second following a sinusoidal wave. The results of this test are presented in the following pages.

```
import socket
import time
import math
from math import exp
import random

IP = "10.0.0.2"
PORT = 12345
MESSAGE = b"A" * 1000 #1000 bytes

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
[...]

def message_sending():
 t = 0
 while True:
   number_of_packets = int(180 + 179 * math.sin(2 * math.pi * t / 10))
   t += 1
   for i in range(number_of_packets):
      sock.sendto(MESSAGE, (IP, PORT))
      time.sleep(random.expovariate(number_of_packets))

message_sending()
```

Figure 25: Python script that generates a sinusoidal traffic

Here is portrayed the result obtained by running the script on the first docker container.

Figure 26: Latency of the first docker container (purple line, values in microseconds) with sinusoidal traffic in input (green line, values in packets per second)

As we can see, there is a clear correlation between the latency and the number of packets sent per second, where low packet interarrival rates are associated with low saturation of the first docker function, and high packet interarrival rates are associated with a higher saturation of the first docker function.

As tested in the previous chapter, we are now going to present the results of how the other links respond to the sinusoidal traffic. The cyan line represents the evolution

over time of the packets per second, ranging from 1 to about 420 packets/s. It is important to note that the latency of the first docker image with the sinusoidal traffic spikes up to 20 ms while the latency of the other links remains in the range between 0.3 ms to 0.6 ms, with variations not correlated to the sinusoidal wave, as we can see in the following graph:

Figure 27: Latency of the other links with a sinusoidal traffic in input (green sinusoidal line)

It will be interesting to investigate how the jitter variation is. As we can see, even with the sinusoidal traffic, the jitter is not affected, remaining quite constant and near zero during the stress-test, meaning that we have low variation in the latency of the links and quite constant variations.

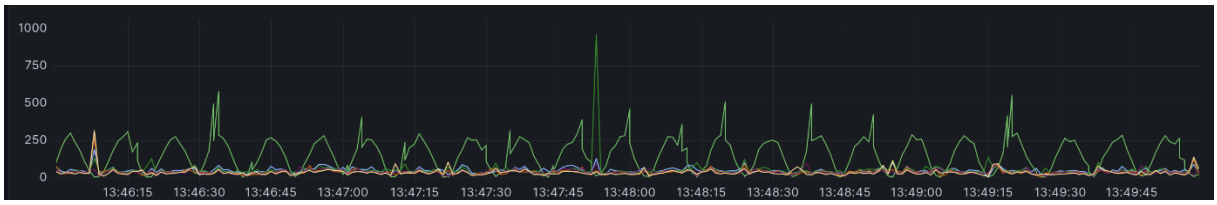


Figure 28: Jitter of the other links with a sinusoidal traffic in input (green sinusoidal line)

These results show clearly a limitation in the capacity of the CHIMAclient running inside docker containers to handle a certain number of packets carrying INT data. This suggests implementing soon a mechanism to avoid the insertion of INT data in every packet to avoid the saturation of the resources.

6 Conclusions

In this work, we explored the CHIMA framework, its advantages, drawbacks, and how we improved it. We introduced the Telegraf agent to collect the metrics generated through INT and used InfluxDB to visualize and manage them. We enhanced the performance of SLA violation detection by reducing the detection time and avoiding the use of EWMA. We performed tests to expose the limitations of CHIMAclient running inside Docker containers.

The proposed new CHIMA framework is faster in detecting SLA violations, more reliable using arithmetic average instead of EWMA, and has a new time-series database to visualize data.

Further work will involve porting this infrastructure into a testbed to analyze its behavior in a non-virtualized environment.

Bibliography

[1] T. P. A. W. Group et al., "In-band Network Telemetry (INT) Dataplane Specification - Version 2.1," 2018. [Online] https://p4.org/p4-spec/docs/INT_v2_1.pdf.

[2] Battiston Elia: "CHIMA: a framework for network services deployment and performance assurance" Tesi di Laurea [Online] https://www.politesi.polimi.it/handle/10589/181816?mode=simple

[3] Open Network Foundations: https://opennetworking.org/p4/

[4] Ismail Butun, Yusuf Kursat Tuncel, Kasim Oztoprak, "data Application Layer Packet Processing Using PISA Switches". In MDPI, Sensors, 2021, 21, 8010.

[5] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow et al., "ONOS: towards an open, distributed sdn os," in Proceedings of the third workshop on Hot topics in software defined networking, 2014, pp. 1–6.

[6] InfluxDB and Telegraf. Available: https://www.influxdata.com/

[7] BPF Compiler Collection (BCC) Available: https://github.com/iovisor/bcc

[8] B. Addis, D. Belabed, M. Bouet, and S. Secci, "Virtual network functions placement and routing optimization," in 2015 IEEE 4th International Conference on Cloud Networking (CloudNet). IEEE, 2015, pp. 171–177.

[9] M. A. Khoshkholghi, M. G. Khan, K. A. Noghani, J. Taheri, D. Bhamare, A. Kassler, Z. Xiang, S. Deng, and X. Yang, "Service function chain placement for joint cost and latency optimization," Mobile Networks and Applications, vol. 25, no. 6, pp. 2191–2205, 2020.

[10] M. Mechtri, C. Ghribi, O. Soualah, and D. Zeghlache, "NFV orchestration framework addressing SFC challenges," IEEE Communications Magazine, vol. 55, no. 6, pp. 16–23, 2017.

[11] D. Hancock and J. Van der Merwe, "Hyper4: Using P4 to virtualize the programmable data plane," in Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies, 2016, pp. 35–49.

[12] P. Zheng, T. Benson, and C. Hu, "P4visor: Lightweight virtualization and composition primitives for building and testing modular programs," in Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies, 2018, pp. 98–111.

[13] FOP4: Function Offloading Prototyping with P4 [online] https://github.com/ANTLab-polimi/FOP4

[14] "INT Collector - Atmosphere project." [Online]. Available: https://github.com/eubr-atmosphere/distributed-network-federation-probe/tree/master/int collector

[15] IP Packet Delay Variation Metric for IP Performance Metrics (IPPM) Available: https://www.rfc-editor.org/rfc/rfc3393

List of Figures

Acknowledgments

I would like to thank my parents, my grandparents, and my aunt Antonella who have always supported me in my studies, both financially and emotionally. I would also like to thank my supervisor Giacomo Verticale for his support and guidance during the thesis writing process.