Executive Summary of the Thesis

CHIMA: a framework for network services deployment and performance assurance

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Elia Battiston

Advisor: Prof. Antonio Capone

Co-advisors: Prof. Giacomo Verticale, Ing. Daniele Moro

Academic year: 2020-2021

1. Introduction

Network Function Virtualization (NFV) has dramatically increased the flexibility in the deployment of network services, but the virtualization of functions on compute nodes can hinder their performance compared to that of the middleboxes they try to replace. This problem has found a solution with the advent of Programmable Data Planes, consisting in the development of forwarding devices with ASIC performance but whose behavior can be defined with the high level and target independent P4 language [1]. Using P4, sections of Virtual Network Functions can be offloaded to programmable network hardware to achieve significantly higher throughput [4]. An application of this approach, based on the use of heterogeneous Service Function Chains, has been studied by Moro et al. [3]. This thesis proposes a framework for the deployment such SFCs, defined as a combination of functions for regular compute hosts as Docker containers and for programmable switches using the P4 language. Programmable data planes are also exploited to perform real time monitoring of the services through In-band Network Telemetry (INT) [2] to guarantee requested levels of performance by redeploying and rerouting sections that are affected by adverse conditions, allowing applications with critical requirements to be deployed as SFCs.

2. System model

The proposed framework is designed to work on a network built with programmable switches that can be targeted by a P4 compiler. The ONOS SDN controller is used to manage the forwarding of packets, and hosts are compute nodes that remotely expose a Docker engine with a configuration that is compatible with the framework.

Such a system can be used for the deployment of network services in the form of heterogeneous Service Function Chains, composed by two types of functions:

General purpose functions: designed to run on compute nodes, provided them in the form of Docker containers.

P4 functions: intended to be built and run on P4 compatible switch, provided as P4 files in which a `control` block with a compatible signature is defined.

In the current prototype of CHIMA, each function of this chain supports one successor at most. The service is only a description of the desired behavior and gives no indication with regard to the physical location of its components. Figure 1 shows an abstract representation of an SFC and how it can be deployed on available devices.

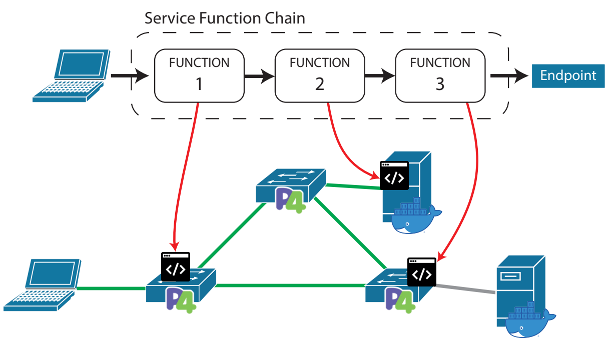


Figure 1: Logical view of a Service Function Chain and how it can be mapped to a physical topology

3. The CHIMA framework

CHIMA is a framework for *CHain Installation, Monitoring and Adjustment*. This section will give an overview of its implementation.

3.1. Framework components

The CHIMA framework consists of multiple components, distributed over a supported network as shown in Figure 2.

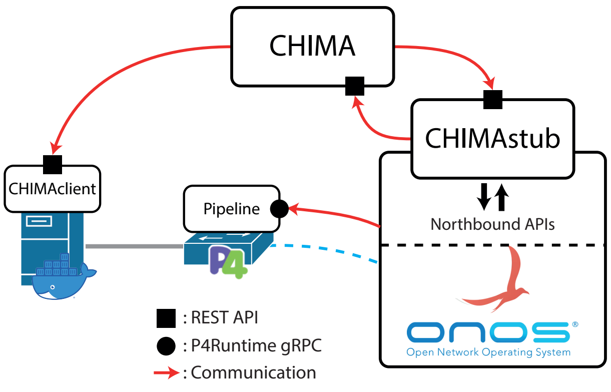


Figure 2: Logical placement of the framework's components in the network and their interactions

CHIMAstub: The Stub is an ONOS application. It exposes topology information and events to CHIMA, and allows interaction with the network's devices through an extension of ONOS REST APIs.
CHIMAclient: This process is placed on hosts to apply header stacks for the correct routing of packets of managed services.
P4 pipeline: Installed on all switches. On top of providing basic forwarding, it supports In-band Network Telemetry according to the INT v1.0 specification. This pipeline is be used as a base for the inclusion of user provided P4 functions at runtime.
CHIMA: The CHIMA process is the core of the system. It manages all other modules through different means of communication. Its tasks are to:

- Construct and maintain an internal representation of the network topology.
- Collect INT data from the reports delivered by switches.
- Compute a deployment strategy based on the available topology information.
- Perform the deployments of functions and manage their routing.

3.2. Template pipeline

The solution adopted to allow the integration of user provided functions is the creation of a pipeline with an extensible section, in which the execution of additional controls can be injected, and only happens if the packet is part of a specific service.

The processing sections of the pipeline are the following.
Forwarding: Since this pipeline is based on the `basic.p4` pipeline included in ONOS, the mechanism it uses forwarding is inherited. If the packet doesn't match any rules in the forwarding table, it is sent to the controller with a *packet out* operation. ONOS will determine its treatment and install additional rules.
Routing: The routing of packets between functions of a service managed by CHIMA is handled separately, and bypasses usual forwarding, as explained in Section 3.3.
INT: The implementation of In-band Network Telemetry is inherited from the ONOS `int.p4` pipeline. The control plane logic for this part of the pipeline is the `inbandtelemetry` ONOS application, which translates INT intents into the rules to be installed.
User functions: At this point, the template pipeline has no instructions. Instead, tokens are placed to signal the spot where user provided

code can be inserted. When CHIMA determines one or more P4 functions of a service have to be deployed on a switch, their code can be injected and managed with conditional blocks.

3.2.1 Setting up the pipeline

The framework compiles the resulting P4 program using a Docker image of the p4c compiler. The installation of a pipeline at runtime can be achieved with the `SetForwardingPipelineConfig` RPC call of P4Runtime. Its implementation in ONOS is exploited by creating a Pipeconf with the pipeline files that is then bound to the device in ONOS's distributed map. In addition to this, the installation process involves the reconciliation of rules installed in the device's tables.
In general, reconfiguration of the pipeline may cause significant downtime, but platform specific features like Tofino Fast Refresh, can greatly accelerate this process.

3.3. Routing

After all the functions have been installed, the correct routing of packets between them has to be configured along the prescribed path.

3.3.1 Segment Routing over MPLS

As defined by RFC8660, in SR-MPLS SIDs are represented as MPLS labels.
CHIMA's implementation of SR-MPLS uses no SR Global Blocks and three SR Local Blocks, within which only single-label SIDs are defined.

- `0x40000 - 0x7FFFF`: used for the execution of user defined P4 functions that have been deployed on the switch.
- `0x80000 - 0xBFFFF`: used for packet forwarding. These segments can be classified as Adjacency SIDs according to RFC8402.
- `0xC0000 - 0xFFFFF`: used to implement the custom extension of INT for the measurement of the delay introduced by general purpose functions, explained in Section 3.4.2.

Keeping in mind that MPLS labels are 20 bits long, their first two bits are used to identify their SRLB, while the remaining 18 can be interpreted as an argument for the action to be performed. The evaluation of segments on a switch continues until the bottom of the stack is reached or an Adjacency SID is found, for which a Penultimate Hop Popping approach is used. Figure 3 shows an example of how MPLS label stacks are used by the framework to perform routing.

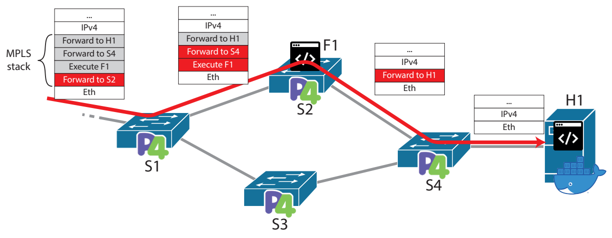


Figure 3: Example of the framework's use of segment routing

3.3.2 Encapsulation and segment distribution

In CHIMA, the MPLS encapsulation of packets belonging to managed services is performed by CHIMAclient. At the heart of CHIMAclient is an eBPF filter that inspects the packets egressing the host. Services are identified using the tuple of source and destination IPv4 addresses. If the tuple is known, it means the filter has a label stack that represent the series of segments used to implement its precomputed path. In this case, the stack of MPLS label headers is inserted between the Ethernet and IPv4 headers. During this process, the EtherType field of the Ethernet header is set to `0x8847` to allow proper parsing. These stacks are computed by the CHIMA process based on the result of an optimization model, and then installed on the CHIMAclient of specific hosts by contacting their REST APIs.

3.4. In-band Network Telemetry

As anticipated, the INT implementation used by the framework is derived from the `int.p4` pipeline included in ONOS, which is designed to be managed by the `inbandtelemetry` application, with which CHIMA interfaces through CHIMAstub. This implementation is based on the INT v1.0 specification, and uses embedded metadata with headers located over TCP or UDP.

3.4.1 Collection of INT data

The CHIMA process includes an INT collector as one of its modules, which is implemented as

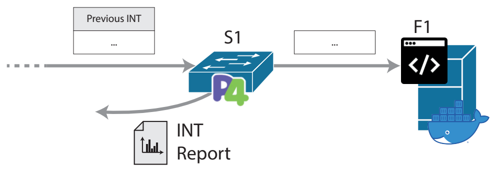
an eBPF filter. The source of the obtained values is identified by the pair of IDs of the switches at the two ends of a link. Every collected value is also exposed to a server of the Prometheus monitoring system.
The values that the collector provides to the framework are an Exponentially Weighted Moving Average (EWMA) of the raw ones, updated at each time step.

$$ewma_0 = value_0 \tag{1}$$
$$ewma_t = (1 - \alpha)\, ewma_{t-1} + \alpha \cdot value_t \tag{2}$$

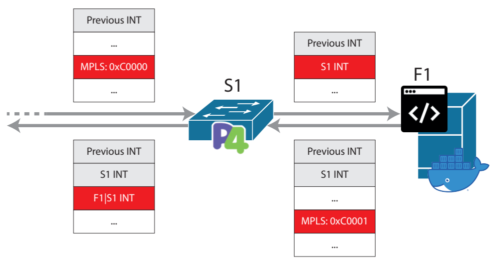
The parameter α acts as a smoothing factor, and can be modified by the user when running CHIMA to tune the framework's response to transient variations of the metrics.

3.4.2 Measurement of general purpose function times

To enforce an upper bound on values such as the time for a packet to reach a particular function, it is essential to consider the execution time of all previous functions in the chain. While P4 functions are guaranteed to run in constant time, general purpose functions cannot assure the same level of stability.
The current state of the project focuses on the measurement of UDP functions in which each packet contains a single execution request. This allows to correlate the time needed by the packet to traverse the function's host and the time of execution of the function itself.
The measurement is performed by altering the forwarding behavior of INT packets to hosts with Segment ID `0xC0000`. INT data will be left in the packet, and new INT transit headers will be added before the packet's egress. The egress timestamp included in these headers will be considered the start of the function's execution. This requires the function to be aware of this data and leave it untouched. The resulting packet will include previous INT data and the function's modified payload. The ingress timestamp embedded by the next switch will mark the end of the function's execution, enabling the computation of its extent. The attribution to the correct function is achieved with a dedicated Segment ID that will be added by CHIMAclient, as shown in Figure 4b.



(a) Regular processing of an INT packet when forwarded to a host



(b) Additional measurement of the function's time using Segment IDs

Figure 4: Comparison of the content of packets and the forwarding behavior with regular INT and with CHIMA's extension

4. Results

4.1. Methodology

To evaluate the detection and redeployment performance, the framework has been instrumented to record the timestamps of relevant events. All measurements have been performed on a bare-metal installation of Ubuntu 20.04 LTS, running on an Intel Core i7-6700 CPU with 64GB of RAM.

4.1.1 Detection delay

The first set of measurements have the objective of determining how much time is needed by the framework to detect the introduction of a perturbation, depending on the values of user-configurable parameters.
The detection delay is computed as the time CHIMA takes to detect an exceeded requirement after the first packet of a perturbed application is sent. Assuming the properties of topology and service to be constant, we can consider the detection delay to be a function of the polling interval and the EWMA coefficient.

Polling interval
This is the rate at which the userspace component of the eBPF INT collector polls new EWMA values. These measurements have been

performed with a value of $\alpha = 2^{-3}$ for the EWMA.
Each data point presented in Figure 5 has been obtained as the mean value of 30 samples. In the same figure, the time taken by a request to traverse the function chain end-to-end is plotted in red for reference.

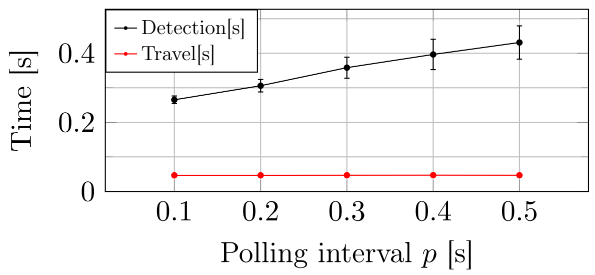


Figure 5: Delay in the detection of an exceeded requirement with different intervals for the polling of new measurements from the INT collector. 95% CI.

As expected, the results present a clear linear trend, directly proportional to p.
A constant contribute is represented by the time needed for the EWMA of the affected measurement to surpass requirements, while the slope of the curve is due to the lower polling frequency. The expected value of the introduced delay will be equal to $\frac{p}{2}$ for a polling interval of p.
These measurements reveal that to achieve minimal detection times, the lowest value of p that doesn't cause excessive system load should be used.

EWMA coefficient
The second parameter is the coefficient for the computation of the Exponentially Weighted Moving Average, explained in Section 3.4.1. While running these tests, the Polling interval has been set to $0.1s$. Since the computation of EWMA is performed in an eBPF filter, it is implemented with bit-shift operators, and only allows users to configure exponent $k \in \mathbb{N}$ where $\alpha = 2^{-k}$.
Greater α values result in more weight given to recent data rather than the old average. This is clearly shown by Figure 6, in which smaller coefficients cause the time needed for convergence to the new measured values to grow exponentially. This data confirms the effectiveness of α to tune the response of the framework in case of short-

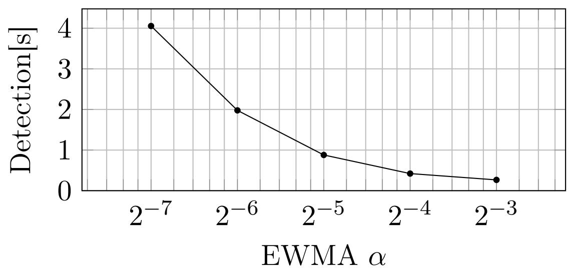


Figure 6: Delay in the detection of an exceeded requirement with different coefficients for the computation of the EWMA on link measurements. 95% CI.

lived congestion events. Therefore, the optimal value for this parameter should be determined based on the intended application.

4.2. Redeployment time

Another crucial measurement to outline the framework's performance is the time needed to complete a redeployment. Table 1 presents a comparison of the relevant characteristics between the test cases for which redeployment times have been measured.

Topology	Switches	Containers	P4 func.
mesh	7	3 (1)	0 (0)
datacenter	6	2 (0)	1 (1)
unbalanced	4	4 (1)	2 (1)
minimal	5	2 (2)	2 (2)
medium	7	3 (3)	3 (3)
large	9	4 (4)	4 (4)

Table 1: Characteristics of the presented test cases. The number of functions that will be moved in each case is stated in parenthesis.

The results presented in Figure 7 show the total redeployment times for these cases, along with the most significant contributing factors. Additionally, all recorded contributes are detailed in Table 2. Since the redeployment of different components is executed in parallel, the recorded times will be equal to the delay caused by the slowest one.
The installation of P4 functions proves to be the dominant factor if present, causing the total time to be in the order of seconds. The two contributions to this delay are the reconfiguration of the switch's pipeline and the reinstallation of the correct set of rules in the pipeline's tables.

Topology	P4[s]	Containers[s]	Paths[ms]	Metadata[ms]
mesh	-	1.13 [1.09,1.17]	54.23 [49.16,59.31]	2.68 [2.55,2.81]
datacenter	5.20 [5.17,5.23]	-	288.60 [283.30,293.90]	2.79 [2.63,2.95]
unbalanced	5.19 [5.16,5.21]	0.97 [0.96,0.98]	60.57 [58.09,63.04]	2.81 [2.64,2.97]
minimal	6.06 [6.02,6.10]	1.50 [1.48,1.51]	48.79 [43.78,53.81]	3.50 [3.31,3.69]
medium	6.93 [6.90,6.97]	2.12 [2.09,2.14]	125.89 [107.94,143.84]	4.81 [3.84,5.78]
large	8.28 [8.19,8.36]	2.54 [2.46,2.63]	477.55 [387.49,567.62]	6.10 [4.84,7.35]

Table 2: Breakout of redeployment times for different topologies. 95% CI.

While the former is due to the use bmv2 switches and could be reduced to tens of milliseconds with vendor specific features, the latter is caused by ONOS's management of programmable data planes, which is not structured for time sensitive pipeline changes. Improvements to target this specific use case could drastically decrease delays.
Times for path distribution and metadata adjustment, which can be entirely attributed to the framework's logic, are much less significant than previous ones, adding minimal overhead.

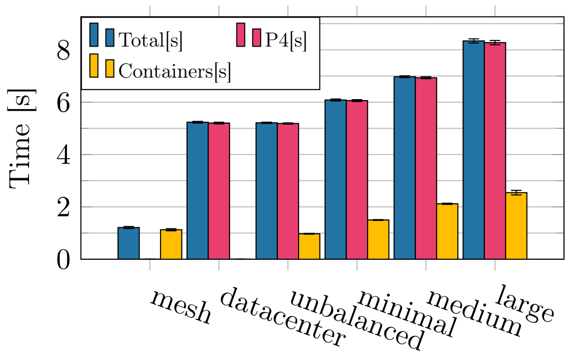


Figure 7: Time for the complete redeployment of a service, along with the contributes of P4 and Container redeployment, in different test cases. 95% CI.

5. Conclusions

In this thesis, a framework for the deployment, monitoring and realtime readjustment of heterogeneous SFCs has been proposed. The possibility to define performance requirements for functions and services, enabled by the the accurate telemetry powered by programmable data planes, opens the opportunity for application with critical performance demands to use existing networks. The extension of measurements to the execution time of functions allows the constraints to reflect real delays experienced by packets, and not just ones caused by the network.
A prototype has been developed and tested through simulations on the FOP4 platform with bmv2 switches, to show that the detection of exceeded requirements happens in the order of hundreds of milliseconds, and can be tuned by the user to achieve the desired level of responsiveness. Analysis of the redeployment process showed that the overhead introduced by the system is negligible compared to the time needed for the startup of functions, and real time relocation of VNFs to achieve desired levels of performance is feasible.

References

[1] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[2] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, volume 15, 2015.

[3] Daniele Moro, Giacomo Verticale, and Antonio Capone. Network function decomposition and offloading on heterogeneous networks with programmable data planes. *IEEE Open Journal of the Communications Society*, 2:1874–1885, 2021.

[4] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 150–156, 2017.

POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

CHIMA: a framework for network services deployment and performance assurance

Tesi di Laurea Magistrale in
Computer Science and Engineering
Ingegneria Informatica

Author: **Elia Battiston**

Student ID: 939500
Advisor: Prof. Antonio Capone
Co-advisors: Prof. Giacomo Verticale, Ing. Daniele Moro
Academic Year: 2020-21

Abstract

Network Function Virtualization (NFV) has dramatically increased the flexibility in the deployment of network services, but the virtualization of functions on compute nodes can hinder their performance compared to that of the middleboxes they try to replace. The use of programmable network hardware to perform part of the processing at line rate can drastically increase throughput. This thesis proposes a framework for the deployment of heterogeneous Service Function Chains (SFCs), defined as a combination of functions for regular compute hosts as Docker containers and for programmable switches using the P4 language. Programmable data planes are also exploited to perform real time monitoring of the services through In-band Network Telemetry (INT) to guarantee requested levels of performance by redeploying and rerouting sections that are affected by adverse conditions, allowing applications with critical requirements to be deployed as SFCs. The solution has been tested by simulating various topologies and services on the FOP4 platform with bmv2 switches. This analysis showed that the system is capable of detecting faults in the order of hundreds of milliseconds, and the overhead it causes in the process of redeployments is negligible compared to the startup time of functions. Measurements also revealed that the current bottleneck for the runtime relocation of heterogeneous functions is the deployment of P4 programs.

Keywords: Network Functions Virtualization, Service Function Chains, Software-Defined Networking, Programmable Data Planes, In-band Network Telemetry, In-Network Computing

Abstract in lingua italiana

La Network Function Virtualization (NFV) ha notevolmente esteso la flessibilità della messa in campo di servizi di rete, ma la virtualizzazione delle funzioni su server può ridurre le loro prestazioni rispetto a quelle delle middlebox che cercano di rimpiazzare. L'esecuzione di una parte dell'elaborazione a line rate, grazie all'uso di hardware di rete programmabile, può aumentare considerevolmente il throughput. Questa tesi propone un framework per l'installazione di Service Functions Chains (SFCs) eterogenee, definite come combinazione di funzioni sotto forma di container Docker per tradizionali host di calcolo e di funzioni in linguaggio P4 per switch programmabili. I Programmable Data Planes vengono sfruttati anche per eseguire monitoraggio in tempo reale dei servizi grazie alla In-band Network Telemetry (INT), la quale permette di garantire i livelli di prestazione richiesti attraverso la rilocazione ed il reinstradamento delle sezioni influenzate da condizioni avverse. Ciò permette la realizzazione di applicazioni con requisiti critici sotto forma di SFC. La soluzione è stata testata simulando varie topologie e servizi sulla piattaforma FOP4 con switch bmv2. Questa analisi ha dimostrato che il sistema è in grado di rilevare guasti nell'ordine di centinaia di millisecondi e che l'overhead causato durante il processo di rilocazione è trascurabile rispetto al tempo di avvio delle funzioni. Le misurazioni hanno inoltre mostrato che nel processo di trasferimento di funzioni eterogenee durante la loro esecuzione, l'attuale collo di bottiglia è rappresentato dall'installazione di programmi P4.

Parole chiave: Network Functions Virtualization, Service Function Chains, Software-Defined Networking, Programmable Data Planes, In-band Network Telemetry, In-Network Computing

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

Introduction **1**

1 Background **9**

1.1 Service Function Chains 9

1.2 Segment Routing 9

1.3 Software-Defined Networking 10

1.4 P4: Programming Protocol-Independent Packet Processors 11

1.5 In-band Network Telemetry 11

2 Related work **13**

3 System Model **15**

3.1 Network and resources 15

3.2 Service Function Chains 16

3.2.1 Functions 17

3.2.2 Services 17

3.3 Objectives 18

4 The CHIMA framework **21**

4.1 Framework components 21

4.2 Service Definition 24

4.2.1 Requirements . . . 24
4.2.2 Definition format . . . 26
4.3 Topology graph . . . 28
4.3.1 Events detection . . . 29
4.3.2 Initial telemetry . . . 29
4.4 P4 functions deployment . . . 29
4.4.1 Template pipeline . . . 30
4.4.2 Setting up the pipeline . . . 32
4.5 General purpose functions deployment . . . 34
4.5.1 CHIMA network . . . 34
4.5.2 Preloading . . . 35
4.5.3 Running . . . 35
4.6 Routing . . . 35
4.6.1 Addressing functions . . . 35
4.6.2 Routing solutions . . . 36
4.6.3 SR-MPLS . . . 38
4.6.4 Encapsulation and segment distribution . . . 40
4.7 In-band Network Telemetry . . . 41
4.7.1 Telemetry commissioning . . . 41
4.7.2 Collection of INT data . . . 42
4.7.3 Computation of link measurements . . . 42
4.7.4 Switches synchronization . . . 44
4.7.5 MTU concerns . . . 44
4.7.6 Measurement of general purpose function times . . . 45
4.7.7 Computation of path measurements . . . 48
4.8 Redeployment . . . 49
4.8.1 Cleanup . . . 50

5 Evaluation 51
5.1 Methodology . . . 51
5.1.1 Collection of measurements . . . 51
5.1.2 Test setup . . . 52
5.1.3 Hardware . . . 54
5.2 Measurements . . . 54
5.2.1 Detection delay . . . 54
5.2.2 Redeployment time . . . 59

6 Conclusion 63

Bibliography 65

List of Figures 71

List of Tables 73

Acknowledgements 75

Introduction

Nowadays the forwarding of packets is only one of the many features offered by networks. The high volume of traffic operators have to manage every day has to be processed multiple times during its routing. Functions such as firewalls, NATs, DPIs, proxies and caches have to be traversed by packets along their journey. Each of them has different goals, but can be thought as part of an overarching service. The creation of services based on the composition of functions across the network made communication between them a fundamental component of their definition, which led to the standardization of this concept with the abstraction of Service Function Chaining [26].

Originally, network functions have been implemented in specialized and proprietary hardware middleboxes. These solutions caused elevated costs for operators, and left them very small room to flexibly manage the network. To increase throughput, new devices had to be bought and installed where needed, constraining the possible paths of affected traffic. Upgrading their functionality would involve the replacement of already deployed equipment. These processes are not only expensive, but also require time spans that are orders of magnitude larger than the speed at which networks can grow and change.

The trend of Network Function Virtualization (NFV) [30][58] aims to provide a solution to these limitation, removing hardware middleboxes from networks in favor of virtualized services that provide equivalent functionalities. These services, in the form of VMs or containers, can be deployed on common compute resources that may already be available, or can be put in place with limited costs thanks to their widespread use.
Virtual Network Functions (VNFs) dramatically increased the ease of management for computations in the network. For example, they can be spawned at the edge of the network, where they are needed most, minimizing latency. They can be deployed in load balanced clusters to support high loads, which can be scaled up in periods of intense traffic or scaled down when they are not needed, freeing resources for other

functions. These actions can be performed in seconds, enabling a level of elasticity in their administration that optimizes both performance and power consumption. Moreover, the set of functions to be deployed can be changed with no alteration to the underlying hardware, lowering the costs and inertia that previously affected innovation.

This paradigm shift has brought many advantages, but there are also downsides to consider. Spreading computation over multiple locations introduces reliability concerns that are foreign to monolithic deployments [36]. Routes that are shared between services can become congested, loose packets and introduce significant delays. Interfaces can break, causing faults and disrupting whole applications. These issues can limit the possibility of using NFV and SFCs for critical applications if not handled carefully. Furthermore, the implementation of functions on Virtual Machines introduces a new set of challenges. The performance of programs running on general purpose hardware is often lacking compared to that implemented in hardware, requiring the use of a higher number of instances to manage the same throughput. The process of VM scheduling can also cause latency fluctuations that negatively affect some classes of services.
Some of these issues can find their solution in another approach that was embraced by network operators in recent years: Software-Defined Networking.

SDN was born as the concept of separating the control plane from the data plane of network devices, much like the abstraction introduced by NFV. In this scenario, the control plane logic is centralized in an SDN controller, that has visibility on the state of the whole network. The collected information can be used to determine how the traffic should be managed without the need for decentralized routing protocols such as RIP, OSPF or BGP. Switches, that only have the role of "dumb" forwarders, expose their internal structure as a series of match-action tables, that can be controlled with the installation of rules through a standardized interface. The most successful protocol for this task is OpenFlow [40], whose paper introduced the whole concept of SDN. The virtualization of the control plane enables operators to develop tailored solutions for traffic management, but the use of forwarding devices that only support a fixed set of fields and protocols represents the limit of what can be accomplished with this technology by itself.

To change this situation, the natural extension of Software-Defined Networking has been in the direction of Programmable Data Planes. Data plane programmability consists in the creation of forwarding devices that can provide the same level of performance obtained by commonly used switches, but whose behavior can be easily modified by the operator. With programmable network hardware, the support for newer or completely custom protocols can be directly implemented, without the need to wait for vendors to support them. At the same time, features that are not needed can be discarded to reduce complexity and attack surfaces. The mechanism to define such pipelines is the P4 language (Programming Protocol-independent Packet Processors) [12], through which the same high level definition can be compiled and installed onto programmable switches from different vendors and with different features or amounts of resources. Elements of its design, such as the absence of pointers or loops, reflect the rigid structure of the devices it is supported by, but ensures all computations to be executed at line rate.

The ability to freely program devices that can achieve such throughput and can be located as close as possible to sources of information, has led many researchers to try exploiting data planes for more than just forwarding. In-network computation [51] [28] is the recent trend stemming from this concept, and involves the offloading of sections of an application to programmable switches that, even if constrained in the variety of operations they can perform, can do so at ASIC speeds. Examples of this practice include implementations of load balancers [42], consensus protocols [18], processing of neural networks [50] and key-value stores [54]. This approach has also been shown to be a viable way to reduce power consumption compared to host-based solutions at high loads [55].

This realization has been applied to the field of NFV too, where many of the functions to be executed naturally fit in the shape of a packet-processing pipeline. Whole functions or parts of them can be accelerated in the data plane, as shown by Moro et al. [44]. The result is the creation of heterogeneous SFCs, in which P4 and common compute resources work together to provide a high performance service.

Another valuable application of the concept of in-network computation has been the introduction of In-band Network Telemetry [33] [52]. INT proposes the use of programmable switches to bring the monitoring of telecommunications to a greater level of detail compared to what was possible before. The information exposed by P4 pipelines can be used to start treating the network as a white box, collecting hop-by-hop data about every packet. This allows measurements to be fine grained, less noisy and near real time.

The objective of this work is to combine the technologies described above to give an answer to the performance problems of distributed computation with Service Function Chains. First, a framework to ease and automate the deployment of "mixed"

SFCs is proposed. The use of functions of different types gives users the possibility to define portions of logic with the P4 language alongside tasks for general purpose compute resources, enabling substantial performance improvements. Then, In-band Network Telemetry is employed to monitor the communication between functions. The user is given the possibility to define performance requirements for the service, whose satisfaction can be supervised thanks to the accuracy of the collected metrics. Additionally, the flexibility of programmable data planes is employed to route traffic between functions, along paths that allow the service to achieve sufficient performance. The detachment from physical placement, characteristic of Virtual Network Functions, can be leveraged to correct degradation when it is detected, by moving part of the chain's components to different devices and re-routing their communication.

This thesis is organized as follows:

- Chapter 1 provides short descriptions of the key concepts and technologies on which the presented work is based.
- Chapter 2 gives an overview of the related literature for the studied subjects.
- Chapter 3 describes the system and context for which the framework has been designed.
- Chapter 4 explains the design of the proposed solution and the implementation details of the developed prototype, which has been released on GitHub [3].
- Chapter 5 describes how the performance of the framework was evaluated, and gives an overview of the obtained results.
- Finally, Chapter 6 sums up the contributions of this thesis and presents possible future work on the subject.

Use cases

In this section, some examples of applications that could benefit from the features of the CHIMA framework will be presented.

Control of industrial robots

A very broad class of applications in which the ability to request a guaranteed delay in the execution of a function can become crucial is the implementation of safety procedures. As an example, we can consider the remote control of the movements of an industrial robot.

The communication between robots and their controllers is usually performed on specialized and standardized network devices, and with ad-hoc protocols. A configuration in which both the actors of the exchange and the communication channel comply with safety standards such as IEC 61508 is called *White channel communication*. A different approach, that involves the use common networking equipment to construct a communication channel between certified devices, is called *Black channel communication* [17]. With proper care about the reliability of its links, an existing network could be used for this purpose. In such conditions, the framework could be used to deploy the control logic of a robot as a service function chain.

The exchange of information that takes place on the channel would involve the transmission of actuator and sensor data from the robot to the controller service. The latter has to process the received data to construct a representation of the current state of the robot, and send a command across the same channel to instruct the robot on the next action it has to perform. We can assume the presence of one or more sensors dedicated to safety, whose job is to detect the presence of obstacles in the work space of the robot. While the delayed delivery of a regular command could decrease the performance of its operation, a late response to the measurement of these sensors can be critical, as interaction of the machine with such objects could damage the equipment, or even cause harm to operators.

To minimize this risk, the controller service could be structured as follows.

- The first function of the service chain can be implemented in P4, enabling its deployment to be as close as possible to the robot. Since P4 is not suitable for the development of complex logic, it would only check the content of packets coming from the machine to detect the presence of threshold violations, like the measurement from a particular set of sensors. In case this check is positive, a packet to halt operations could be sent immediately.
  In the definition of this function as part of a CHIMA deployment, the requirement for its maximum latency can be set to a predetermined value, depending on the characteristics of the specific equipment.

- The second function can be more complex, and represent the complete logic that the robot has to follow. This function can be implemented with any technology and packaged as a collection of Docker containers. As anticipated, the task of this function would be to collect incoming information and use it to determine successive commands. Even if this procedure is less critical than the previous, requirements to request a limit to the end-to-end delay of its communication can be set in the framework to guarantee adequate performance, also taking into consideration the time needed by the procedure itself.

A similar scenario has been recently proposed and studied by Cesen et al. [13], showing the correlation between the experienced delay and the error in the stopping position of the robot.

With this configuration, empowered by the capabilities of programmable data planes, the framework can assure that safety-related data will be processed in time to take action.

Precision agriculture

Another possible use case for this system can be found in the field of precision agriculture. Precision agriculture [59] consists in the aggregation of measurements from various sources (such as multispectral images acquired with drones or satellites, sensors, weather monitoring, etc. [46]) to determine the optimal amount of resources to administer to sections of a field in order to maximize its yield, while minimizing waste. A very precise prediction of the amount of water, fertilizer and chemicals to apply in a spot can increase efficiency, but also requires precise enough equipment to follow these instructions.

This operation can be automated with remotely controlled tractors. This kind of vehicle would mount a GPS sensor in order to determine its location [19]. This information can be sent to a service deployed with an edge or fog computing approach. By consulting the collected information about the area, it would be able to determine the maneuvers to execute and the precise amount of resources to dispense. The results can be sent as commands to the tractor. Of course the profitability of this process also depends on the speed at which the whole field can be treated, which requires the machinery to sustain certain speeds. In this situation, commands about the rate of distribution must reach the tractor before the spot they refer to has be surpassed.

The deployment of the remote service with the CHIMA framework could be used to guarantee adequate performance. Requirements on the communication can be used to ensure that commands will reach their destination within a predetermined delay based on the granularity of the predictions and the speed of the tractor. In case adverse conditions cause the service to be less responsive than needed, its components can be moved to unaffected sections of the managed network.

1 | Background

1.1. Service Function Chains

The implementation of end-to-end network services usually involves communication to be incrementally processed by various functions, such as Firewalls, NATs, DPIs, etc. With the increasing trend in the virtualization of these kinds of network functions, also called Network Function Virtualization or NFV, their deployment has become more and more flexible and dynamic, introducing the need for a consistent description of the interactions between them. Service Function Chains are the abstraction of these concept, and have been standardized with RFC7665 [26]. The most basic definition of a SFC includes two endpoints, which represent the source and sink of packets, and a set of functions. Each of the functions performs some computation on received packets, and forwards them to its successor. The order of execution of functions may or may not be defined, depending on the application. This high level description of the desired behavior can then be mapped on a physical topology, by determining and enforcing paths that traffic of a particular service has to follow to be correctly processed.

1.2. Segment Routing

Segment Routing (RFC8402 [20]) is the concept of determining the routing of a packet based on a set of instructions called *segments* that are assigned to it. This technique enables extreme flexibility in traffic engineering, allowing the definition of different treatments for each packet by embedding the segments to implement it in the packet itself. Segments can instruct devices that process the packet on any type of operation, both related to the topology (e.g. forwarding) and to other services available in the network. When they are embedded in packets, segments are represented by their Segment IDs or SIDs which, depending on the data plane used for the implementation of SR, can be represented in different ways. For example, SR-MPLS uses the values of MPLS labels, while SRv6 (Segment Routing over IPv6) uses the dedicated IPv6 extension header called Segment Routing Header (SRH). A section of network composed of devices that cooperate for the execution of Segment Routing is called an SR domain. The semantic of SIDs can be defined in SRGBs (SR Global Blocks), or SRLBs (SR Local Blocks). While the interpretation of the former is the same across the whole SR domain, the latter can be interpreted in different ways depending on the device that executes them.

1.3. Software-Defined Networking

Software-Defined Networking is an approach to network management whose goal is to enable experimentation and innovation in the management of traffic. It consists in the decoupling of control plane logic from network devices, which becomes centralized in the form of an SDN controller. ONOS, Open-Daylight and Ryu are examples of successful SDN controllers currently used in the industry. The centralization of the control plane eliminates the need for distributed routing protocols, that were originally executed by switches themselves. The amount of information that each device could obtain by collaborating with peers was limited, while the SDN controller maintains a representation of the state of the network as a whole. This allows the development of more advanced routing techniques, or simpler ones that are optimized for specific topologies. In this configuration, network devices only perform the forwarding of incoming packets based on the instructions provided by the controller. To do so, they expose an abstraction of their capabilities based on match-action tables, which can be populated by the control plane. One of the most successful protocols used for the interaction between the controller and switches is OpenFlow [40], whose original paper introduced the concept of SDN. When switches encounter a packet for which no forwarding rules are defined, it is sent to the controller. According to its programmable logic, the controller will determine how similar packets must be handled. To enforce this behavior, it installs rules in the tables of forwarding devices, that will then execute them for every packet that matches a predefined set of conditions. The greatest limit of SDN is represented by the fixed set of fields and protocols supported by the hardware of OpenFlow switches, solved with the advent of Programmable Data Planes.

1.4. P4: Programming Protocol-Independent Packet Processors

P4 [12] is a high level language for the definition of custom pipelines to be installed on switches with a programmable data plane. Data plane programmability is a natural extension of the customization in network management enabled by Software-Defined Networking. P4 defines an abstract forwarding model in which a packet is parsed, processed with multiple stages of match-action tables, and deparsed. Parsing is defined with a finite state machine, in which each state can *extract* headers with a predefined structure from the received packet. Depending on the values of the extracted fields, such as the EtherType for an Ethernet header, the correct state transitions can be performed until all headers have been correctly parsed. At this point, various operations can be executed on the packet. Depending on metadata, such as the ingress port, or on the content of headers, an egress port can be set. The values of some fields can be modified, like the TTL field of an IPv4 header. Entire headers can be added or removed, for example in the implementation of the encapsulation for tunneling protocols. In the end, the modified headers will be deparsed in the order defined by the developer and joined with the packet's payload to be forwarded to its intended destination.
The syntax to define these operations is close to that of languages like C, but with some important differences that will ensure its execution to terminate in limited time, to achieve line rate performance. For example, no loops or pointers are supported. The combination of these features allows developers to define packet processing functionality independently from the target hardware, on which a compiler with a suitable backend can map the instructions using the available resources. Target specific features can be exposed by hardware manufacturers with *externs* in their architecture model. These are interfaces that can be used by P4 developers to exploit hardware accelerated features like the computation of checksums.

1.5. In-band Network Telemetry

In-band Network Telemetry [33] is a monitoring technique born from the ability of programmable switches to query their internal state during the processing of packets. Information such an identifier of the switch, ingress and egress times, queues and buffers occupancy can be recorded each time a new packet is processed. Different ways to collect this data have been proposed, among which an embedded mode that consists in adding the information to the transiting packet, or postcard mode, in which another packet with such information is generated each time. The aggregation of the data provided by INT can help debug a network at a very low level, changing the assumption of it being a black box. For example, the exact route followed by a packet and the level of congestion and latency experienced by it while travelling can be known with hop-by-hop granularity.

2 | Related work

The problem of placing VNFs of a Service Function Chain in a network while determining the optimal path for their communication is a heavily studied topic. Due to the amount of research in this field, CHIMA does not address this aspect. Instead, it is complementary to it, focusing on the execution and monitoring of the deployment that would follow this optimization step.
On this subject, [8] considers the joint problem of placement and chaining of the VNFs, and proposes a mixed integer linear programming model for its optimization. Bounds on the end-to-end latency are also considered, making the solution suitable for the requirements that can be set with CHIMA. [32] also focuses on the optimization of latency and costs in the placing of SFCs, but does so with heuristic-based algorithms. A similar goal to the one of this thesis is pursued by [41], which studies the use of these kinds of algorithms by an orchestration framework that actually performs the deployment of SFCs, interacting with a variety of SDN controllers. The researchers' work also takes monitoring of the deployed service into consideration but, compared to this thesis, does not propose solutions to guarantee performance with runtime readjustment or the measurement of function execution times. However, all of the above only take into consideration SFCs composed by VNFs that target common compute architectures. CHIMA supports the deployment of heterogeneous SFCs that take advantage of programmable data planes, and can significantly increase the achievable throughput. The concept of decomposed VNFs is first explored by [49], but the use of implementations that take advantage of programmable network hardware is introduced by [45], which takes into consideration the different type of hardware that the network exposes and its compatibility with the requested functions, proposing both an optimization model and a heuristic algorithm for their placement.

The deployment of user functions on programmable data planes as performed by CHIMA is heavily inspired by the approach of [45], which also uses an extensible template pipeline to allow the simultaneous installation of multiple functions on the

same switch while providing basic forwarding functionalities. The execution of such functions on specific packets is then requested with a Segment Routing approach implemented over IPv6 by following the SRv6 RFC. CHIMA does the same while using SR-MPLS, which allows the use of Segment Routing on an IPv4 network layer. This was done because, besides routing, the template pipeline used by this thesis also provides monitoring by expanding the `int.p4` pipeline of the ONOS SDN controller. Other approaches have been proposed for the composition of P4 functions. Some examples are Hyper4 [27], which defines a P4 program that is able to emulate other P4 programs provided at runtime by the control plane, and P4Visor [60] which suggests a technique to merge multiple P4 programs into a single one, while retaining the functionality of all of them.
In a similar fashion, DPPx [48] also enables the installation of data plane programs on P4 switches for the enhancements of NFVs. An alternative take on the acceleration of SFCs with programmable data planes is explored with P4sc [14] and [34], both of which consider the implementation of whole function chains on single switches, lacking the possibility of combining components for different technologies or exploiting the flexibility granted by the distributed nature of VNFs.

This thesis also proposes the relocation of the components of chains at runtime as a solution to performance degradation. The optimization for this kind of readjustment has been explored in [15], proposing an algorithm for the real-time migration of VNFs and demonstrating the possibility of lowering latency with it. The dynamic adjustment of SFCs has also been treated by [37], which however focuses on the amount of available resources when new services are introduced in the network, with integer linear programming and column generation heuristic approaches.
To understand when redeployments must be executed, CHIMA uses In-band Network Telemetry for the monitoring of the deployed SFCs communications. A similar approach is studied with IntOpt [11], that aims to optimize the overhead introduced while achieving optimal measurements of a deployment, but lacks the measurement of the execution time of functions, preventing the calculation of end-to-end metrics as experienced by packets. The same is done by IntSight [39], which has a similar goal to CHIMA in trying to detect requirement compliance. Instead, [16] also proposes the use of INT to heal the performance of services at runtime, but only considering the alteration of traffic flows between fixed endpoints to do so.

The relocation of P4 VNFs at runtime is studied in depth by P4NFV [29], which also enables the migration of stateful functions while preserving their consistency.

3 | System Model

3.1. Network and resources

The proposed system is designed to work on a network, represented in Figure 3.1, with the following characteristics.

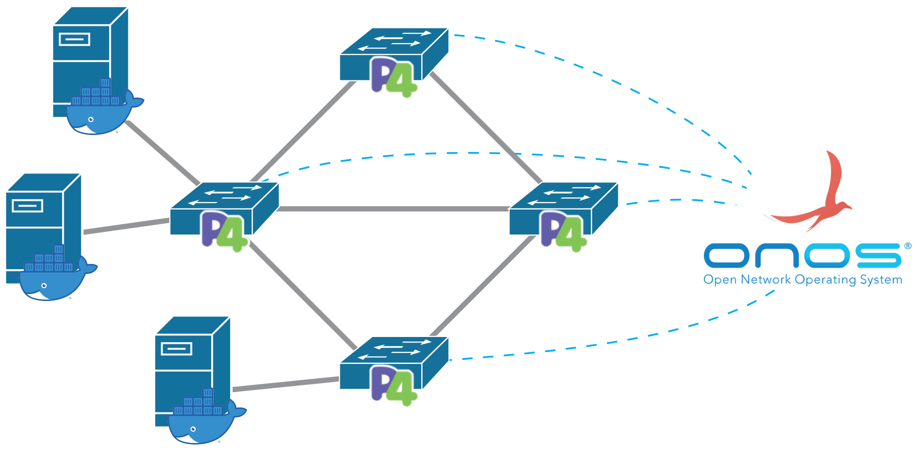


Figure 3.1: Example of a network architecture on which the project could be instantiated

Switches

The use of programmable data planes for both the routing of managed services packets and monitoring of communication through In-band Network Telemetry requires the use of switches that can be targeted by a P4 compiler. This requirement can be satisfied by physical and virtual switches [5] too. The main representative of the second category is the bmv2 [1] software switch, extensively used for P4 prototyping and during the development of the CHIMA framework.
However, execution of P4 programs at line rate and in constant time, which is one of the core features of programmable data planes and drove the recent interest in in-network computing [51], can only be achieved by physical switches that execute

pipelines with dedicated hardware components. Thus, the most effective uses of this system, that involve the deployment of performance dependent P4 functions, require hardware switches to be part of the network.

SDN Controller

The framework makes use of the ONOS SDN controller for many of its main functions as will be explained in detail in the following chapters. For example, ONOS is used to construct an internal representation of the network topology, to detect topology-changing events, to set up In-band Network Telemetry and to install P4 programs on switches at runtime.
In addition, ONOS performs the regular duties of a controller, such as providing switches with rules to allow correct forwarding of packets that are not managed by the framework's routing. These include communication between the CHIMA process and ONOS itself, or the Docker daemon exposed by hosts.

Hosts

The ability of the system to deploy functions for general purpose architectures is achieved through the deployment of Docker images on properly configured hosts. For this reason, servers to be used for this purpose have to run the Docker daemon, and remotely expose it to accept deployment requests from the framework.
The current prototype of the framework only supports communication with remote Docker daemons on the standard TCP port for this service (2375), but could be trivially extended to support connection through SSH. The latter case would prevent the security concern of unauthorized deployment on the hosts by third parties, and would require proper configuration of private keys between each of the hosts and the machine running the framework.

3.2. Service Function Chains

The system described above can be used for the deployment of network services in the form of Service Function Chains. The class of SFCs that have been taken into consideration is defined by their components and structure, as follows.

3.2.1. Functions

Functions are the smallest components of a deployment. They consist of user provided code that performs arbitrary computation on an incoming packet and returns another (likely modified) packet.
The framework supports two types of functions, which are directly dependent on the available components in the networks of Section 3.1.

General purpose functions

This type of function is designed to run on compute nodes powered by processors of common architectures such as x86 or x64. These functions may be implemented with many different technologies, and require particular environments to be run. Since the framework needs to deploy them as atomic units, the user has to provide them in the form of Docker containers, to ensure they can be run without additional configuration.

P4 functions

This type of function is intended to be built and run on P4 compatible switch. The structure of this kind of hardware ensure its execution will happen in constant time, enabling processing of the packets at line rate. The user has to provide them as P4 files in which a `control` block with a compatible signature is defined.

3.2.2. Services

In CHIMA, the concept of *service* can be defined as a chain of connected functions that perform subsequent computations on packets sent by a client. Each function of this chain has at most one successor, to which it forwards packets after its execution completes. The only function without a successor is the last component of the chain, that will send its packets to the endpoint of the service. While the common definitions of SFCs present communication between two endpoints, the current state of CHIMA only allows for the second endpoint to coincide with the client.

The service is only a description of the behavior that should be obtained by the system, and gives no indication with regard to do with the physical location of its components. Figure 3.2 shows an abstract representation of this chain of functions and how it can be deployed on available devices.

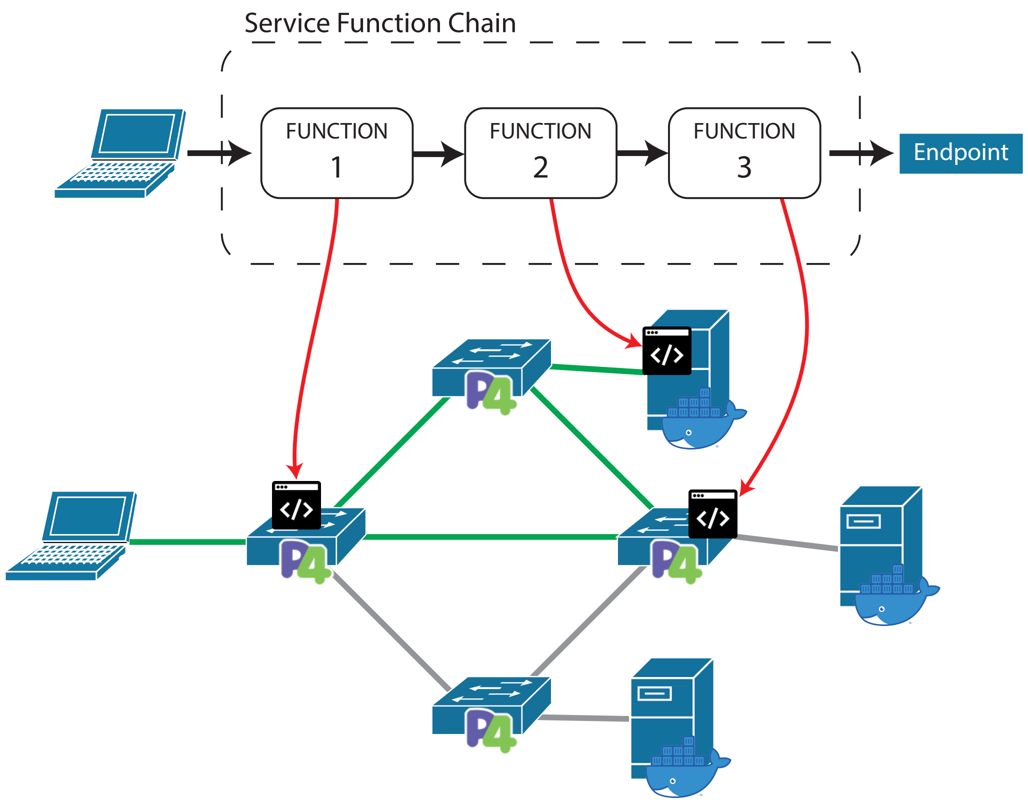


Figure 3.2: Logical view of a Service Function Chain and how it can be mapped to a physical topology

3.3. Objectives

SFCs detach the logic of an application from the concept of its deployment. This allows the orchestrator of such services to act in very flexible ways, determining a placement of functions according to factors such as the condition of the network or the amount of available compute resources. This fact can be exploited in various ways, including the possibility of changing the arrangement of functions after the initial deployment to avoid faulty or congested sections of the network.

The objective of the proposed system is to use the resources of the network (with a particular focus on programmable data planes), combined with the adjustability of SFCs, to design services with resilient communication for which a requested level of performance can be guaranteed. To succeed in this goal, the system should be able to:

- Allow the definition of performance requirements for a service.
- Tightly monitor the communications of services.

- Enable users to inspect the collected telemetry data.
- Detect the perturbations that cause the service to exceed its requirements, with minimal delay.
- If there is a way to satisfy requirements after a perturbation is introduced, move all or some of the components of the perturbed service to bring it back to a compliant state, minimizing overhead on the spin-up time of functions.

4 | The CHIMA framework

CHIMA is a framework for *CHain Installation, Monitoring and Adjustment*.

This chapter will be used to explain how this proposed solution can be used to meet the objectives set in Section 3.3. At first, the components of the system are listed, and a brief explanation of their role is given. Then, different aspects of its implementation are incrementally taken into consideration. The definition of services and functions will be tackled first, moving then to the process of deciding their placement and how their deployment is carried out on the determined devices. Next, the way in which communication is set up between them and how its performance is measured will be illustrated. Finally, the execution of the redeployment process is explained.

4.1. Framework components

The CHIMA framework consists of multiple components, distributed over a supported network as shown in Figure 4.1.

CHIMAstub

The Stub is an ONOS application. Once installed and activated on the controller, it can perform several operations by interacting with ONOS's northbound API and core services. It allows the CHIMA process to access information regarding the network, and interact with its devices.
Its features can be accessed through an extension of ONOS REST APIs.

CHIMAclient

CHIMAclient is a process that has to be placed on hosts to make them available to the framework for the deployment of "general purpose architecture" functions. Its job is to detect if a packet is part of a managed service and, in that case, it applies

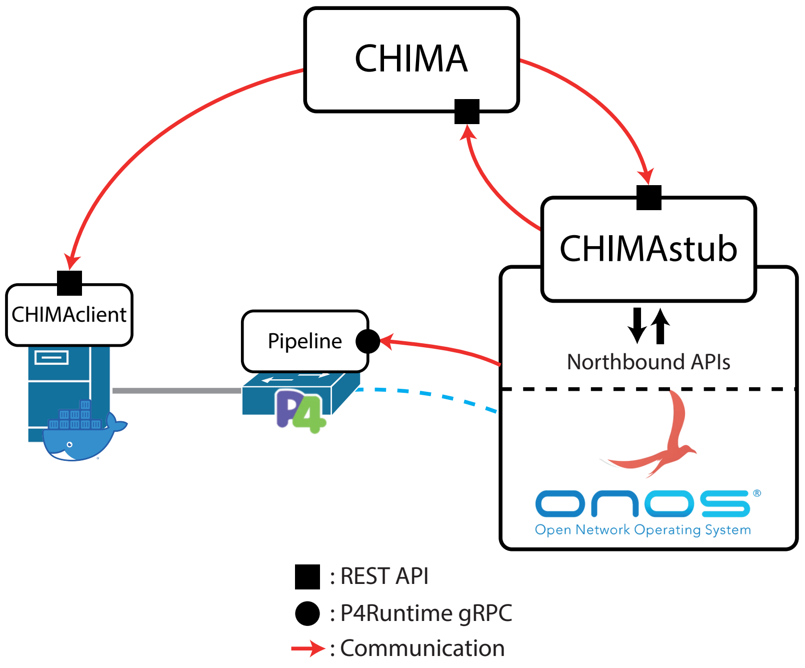


Figure 4.1: Logical placement of the framework's components in the network and their interactions

needed headers to ensure that routing and the execution of functions are correct. The information used to distinguish packets is provided by CHIMA through a REST API when the deployment of a service is created or modified.
In cases where the clients of services deployed by this framework belong to a class of devices that are not suitable for the installation of such a component, the same functionality could be moved to the P4 pipeline, albeit with significant engineering effort.

P4 pipeline

The P4 pipeline used by the framework has to be installed on every switch of the network. On top of providing basic forwarding capabilities, this pipeline supports In-band Network Telemetry as described by the INT v1.0 specification document [24]. The control plane components of both of these functions are part of the ONOS controller, that instructs the pipeline by adding and removing rules to its tables.
In addition, this pipeline can be used as a base for the construction of extended pipelines, that are created and installed at runtime by the CHIMA process. A more detailed overview can be found in Section 4.4.1. The extended pipelines include P4 functions written by the user, which can be executed on a packet when signaled with the addition of a label in the header applied by CHIMAclient. Other kinds of labels

in the same header may also instruct the pipeline on how to forward the packet and how to manage INT data, as will be explained in Section 4.6.3.

CHIMA

The CHIMA process is the core of the system. It manages all other modules through different means of communication, in order to achieve the desired state. The task it performs are:

- Construct and maintain an internal representation of the network topology, including the presence of hosts that are correctly configured for a deployment. This is achieved by registering itself to CHIMAstub, that starts to forward relevant events to the framework's own REST API.
- Collect INT data coming from switches, that forward reports to this process. This data will be used for the computation of function placement and to detect if any user requirement is exceeded.
- Compute a deployment strategy based on the available topology information, the collected telemetry data and the requirements of the user requested service. The current version of the project does not perform this computation and assumes the best deployment to be known, since routing optimization is an extensively treated problem in telecommunication literature. The placement of the components of service function chains in particular, has been studied both for the case of dynamic [37] and latency-aware [53] deployments. For this reason, it was chosen to focus development efforts on other aspects of this system.
- Perform deployments that require:
  - Creation and installation of P4 pipelines that are extended through the embedding of user functions.
  - Management of user functions on general purpose architecture hosts.
  - Delivery of routes to the CHIMAclient component on each of the involved hosts.

4.2. Service Definition

The services managed by CHIMA are those following the description given in Section 3.2.

Services are the units that the user provides to the framework to perform a deployment. To completely describe a service in a way that the framework can handle, the needed information is:

- A list of the functions that make it up
- How the functions communicate between each other
- The location of the client that will use the service

In addition to their basic structure, additional demands can be set during their request.

4.2.1. Requirements

The most prominent feature of CHIMA is to allow its user to define requirements on the performance of the communication between functions. These requirements are used to detect when the communications of a service experience a level of degradation that impair its effectiveness, and some of the functions need to be relocated in the network to keep performance above their threshold. The process of relocating components of the service is called *redeployment* and is described in Section 4.8.

The supported requirements are inspired by the ones specified by the Deterministic Networking (RFC8655 [22]) project, whose objective is similar to the one of the framework in trying to achieve bounded latencies on unicast data flows.
RFC9016 [56] defines various requirements for a data flow, of which the ones supported by CHIMA are a subset. The limits of what is supported strongly depends on the capabilities of the In-band Network Telemetry that the framework uses for measurements.

For example, requirements on packet loss are not supported, since INT works by embedding its telemetry data in the packet itself, which won't be delivered if the packet is lost. Requirements on bandwidth are not supported by the framework since INT only provides information regarding single packets and not the link's bandwidth in its entirety. Requirements on packet misordering are not supported since CHIMA forces all the packets of a service to follow a single predetermined path, as explained in 4.6, preventing the misordering of packets of the same communication.

The supported requirements are:

- The *maximum latency* experienced by a packet of the flow from its departure to its destination. The maximum latency is specified as an integer number of nanoseconds.
  This requirement is supported, since the latency of packets can be directly computed with ingress and egress timestamps provided by INT, with a resolution of nanoseconds. It is checked against the the sum of the latency on the affected links and the time of execution of functions, which are obtained through an extension of INT that will be explained in Section 4.7.6.
- The *jitter*, an instantaneous measurement of the latency variation experienced by different packets on the same path.
  This requirement is checked against the difference between the last and second-last recorded values of latency for the considered path.
- The *maximum latency variation*, which is the difference between the minimum and the maximum latency on a path, and can be used to capture the drift of the latency values experienced by the service since the beginning of its operation.
  This requirement is checked against the difference between the maximum and minimum recorded on a path since the deployment of the service. It is expressed in nanoseconds.

Each of these requirements can be specified for single functions or for the service in its entirety.

- When requirements are set for a function, they refer to the path used by packets to reach the function, starting from the client. This definition enables users to express constraints on the state of the network as seen by the function itself, instead of relying on the incremental definition of limits for the segments of the path..
- When a requirement is set for the whole service, the measurements consider the full end-to-end path. This means the whole route from the client to the last function and back is used in the computation of latency, jitter and latency variation.

Figure 4.2 shows the extent of the paths on which requirements would be computed on a service chain of 3 functions. For example, the total latency considered for requirements set on function 2 would include: the latency on the links between the client and function 1, the execution time of function 1, and the latency on the links between function 1 and function 2.

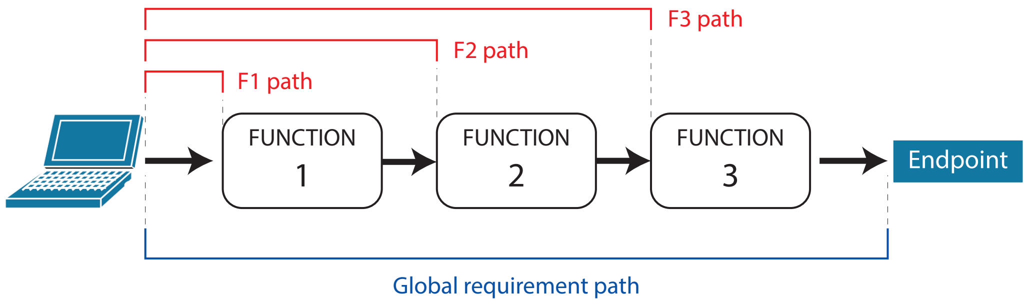


Figure 4.2: Visual representation of the section of path that is considered in the computation of different requirements

Details on the computation of the path-wise metrics can be found in Section 4.7.7.

4.2.2. Definition format

CHIMA accepts service specifications through a YAML file. The following is an example of its structure.

```
---
service:
  client: 10.0.0.200
  port: 12345
  latency: 150000
  jitter: 1000
  variation: 10000
functions:
  changettl:
    type: switch
    file: $HOME/measurements/switches/functions/changettl.p4
    control: changettl_control
    next: toupper
  toupper:
    type: container
    latency: 5000
    jitter: 500
    variation: 1000
```

```
19      file: $HOME/measurements/switches/toupper.yaml
20      next: echo
21    echo:
22      type: container
23      file: $HOME/measurements/switches/echo.yaml
```

Service

The `service` object on line 2 includes information on the communication that the service performs.

- `client` defines the IPv4 address of the device that is supposed to contact the service. The CHIMAclient component has to be installed on this device.
- `port` defines the UDP port used by the application functions to communicate. Only UDP applications are currently supported for the reasons explained in Section 4.7.6. This information is used to target the application's packets when setting up INT rules on the switches.
- `latency`, `jitter` and `variation` allow the definition of requirements on the end-to-end communication of the whole service

Functions

The `functions` object at line 8 defines the functions that compose the service chain. Each function is uniquely identified by its name, and has the following attributes:

- `type`: the type of function, that expresses the kind of device on which it will be installed. Allowed values for this attribute are `switch` and `container`, and it determines the rest of the `functions` object attributes. This attribute is mandatory.
- `latency`, `jitter` and `variation` are optional attributes that can be used to define a requirement on the corresponding telemetry data of the function. They are expressed in nanoseconds, and represent the maximum value the corresponding measure can reach before triggering a redeployment of the service. At the time of deployment, the framework creates a *trigger* that will be checked against INT data every time it is updated.
- `next`: this attribute can be used to define the order of execution of the functions in the service. Its value has to be the name of another function defined

in the same file. Only one function is allowed to lack a `next` attribute, and it will be treated as the end of the chain.

The remaining attributes, that depend on the value of `type`, provide the location of the actual code that implements the function.

- For `container` types, only the `file` attribute is needed. It is a path to the Compose file [4] that defines the set of containers that constitute the function. The deployment of this kind of files will be explained in detail in Section 4.5.
- For `switch` types, both a `file` and `control` attributes are needed. The former is the path to a P4 file that contains one or more P4 controls with a known signature. The latter is the name of the control that represent this specific function.
  The way in which the control is integrated in the template program of a switch and installed on it will be explained in detail in Section 4.4

4.3. Topology graph

After the user has properly defined and submitted a service to be deployed, CHIMA has to map its components on the physical network. It goes without saying that a crucial step towards this objective is to locally construct a precise representation of the network, on which the best placement of components can be computed. The ONOS SDN controller, that the framework already exploits to control programmable switches (Section 4.4), maintains detailed topology information that can be queried through its northbound APIs.

The data needed for the optimization of functions placement is:

- The set of switches in the network, referred to as *devices* by ONOS
- The links between devices' ports
- The set of hosts, the devices they are linked to, and their IP addresses. The addresses will be used to determine whether they are correctly configured for function deployment and to actually perform it.

Of course networks are subject to frequent changes, and a snapshot of their topology is not enough for correct management. The framework has to keep up to date with the current state of the system and react accordingly. For this reason, topology information is obtained through the use of CHIMAstub.

4.3.1. Events detection

Polling ONOS's APIs at regular intervals to detect changes would result in a waste of resources and introduce delays in the response, depending on the polling rate. Instead, the creation of a publish/subscribe system for topology events has been implemented in CHIMAstub.

The stub extends ONOS's REST API with methods to subscribe or unsubscribe from the distribution of events. The framework, instead, exposes another REST API through an internal web server, with methods that can be called by the stub to publish an event. When CHIMA is started, it tries to register to the stub with the information needed to contact its own API as an argument. At the time of a new subscription, the stub collects topology data through ONOS's northbound and sends it to the subscriber as a series of events. At the end of this process, the framework has received all the information needed to construct an internal representation of the current state of the network.

After this setup phase, the stub can use the same APIs to push events to the framework when needed, and allow their processing as soon as they are detected by the controller. This is achieved through the implementation of listeners for the relevant classes of events.

4.3.2. Initial telemetry

Before the service is deployed, CHIMA performs a period of measurements on the communications of the whole network, by specifying its subnet in an INT intent. This allows a baseline evaluation of latency and jitter for each of the links. With this information, an optimization problem can be solved to determine a mapping of functions to devices, and a path that allows them to communicate in a way that satisfies each of their requirements. Assuming such a configuration exists in the observed state of the network, the deployment of functions is executed as instructed by the model.

4.4. P4 functions deployment

The process of deploying arbitrary P4 functions on switches that have to simultaneously perform forwarding and In-band Network Telemetry is not trivial. Since P4 switches only support the execution of a single program, the framework needs to merge the basic functionalities and the custom ones to create a single pipeline.

4.4.1. Template pipeline

The solution adopted to allow the integration of user provided functions is the creation of a pipeline with an extensible section, in which the execution of additional controls can be plugged in. Using a template system, user code can be injected in sections whose execution is controlled by the framework, and only happens if the packet is part of a specific service.

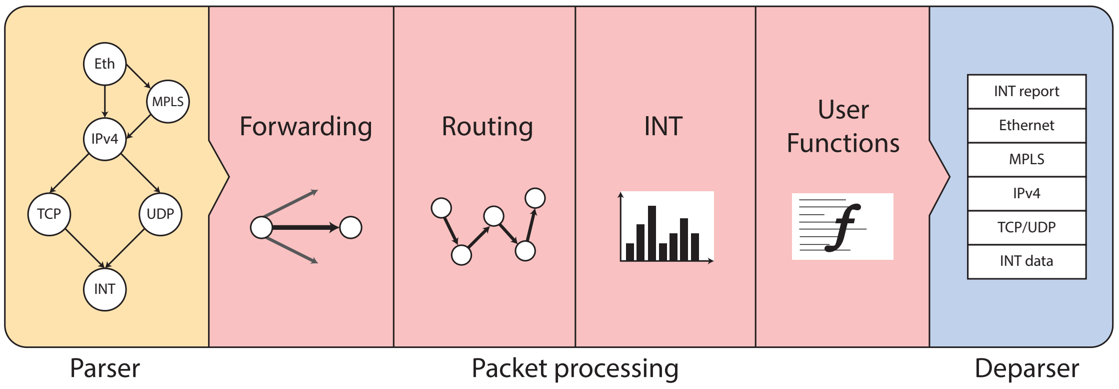


Figure 4.3: Stages of the template pipeline

Forwarding

The most basic function of a data plane program is the forwarding of packets. Since this pipeline is based on the `basic.p4` pipeline included in ONOS, the mechanism it uses forwarding is inherited. When a new packet begins ingress processing, it is checked against a tables of known rules that determine its egress port. If this process results in a table hit, the egress port is set. Otherwise, the packet is sent to ONOS, that determines the correct egress port to reach the destination using the `ReactiveForwarding` application, and installs the corresponding rule on the switch.

Routing

The routing of packets between functions of a service managed by CHIMA is handled separately, and bypasses usual forwarding. The way it works is explained in Section 4.6

INT

The implementation of In-band Network Telemetry is inherited from the ONOS `int.p4` pipeline. This program closely follows the one suggested in the INT v1.0 specification [24]. The control plane entity that maneuvers this part of the pipeline is the `inbandtelemetry` ONOS application, which is able to translate INT intents into the rules to be installed in the tables of switches. More details on the use of INT can be found in Section 1.5.

User functions

At this point, the template pipeline has no instructions. Instead, tokens that are well known to the framework are placed to signal the spot where user provided code can be inserted. The tokens are a series of characters that do not constitute valid P4 code, and thus should not be found anywhere else in a valid program, nullifying the possibility of clashes. The use of a token for the injection of code instead of a hard-coded offset makes it possible to change the basic functionality of switches without altering the framework.

When a new switch is added to the network, the template pipeline with all of its tokens substituted by blanks is used to initialize it. At a later time, when CHIMA determines one or more P4 functions of a service have to be deployed on a switch, a merged pipeline that include all their code is created.

The insertion of user-provided code is divided in three sections:

- Insertion of the `p4` files for each of the functions, using the `#include` preprocessor directive.
- Instantiation of the corresponding controls, which according to the P4 specification [7] is needed to invoke the services from another control (the one that manages the execution of functions, in this case).
  The identifier used for their instantiation is composed by the original name of the control and a number that uniquely identifies it. This way, the same control can be submitted by the user as two different functions without the risk of name clashes.
- Creation of a series of conditional function calls, that will allow the execution of the controls for packets that belong to the correct service.

4.4.2. Setting up the pipeline

Once the merged pipeline is ready, it has to be compiled and installed on the right switch.

Building

The framework compiles the resulting P4 program using a Docker image of the p4c compiler. At the moment, the bmv2 software switch is the only one supported by the framework, but it could easily be extended to any other hardware that provides a p4c backend.

This process generates two files:

- A `json` file, which represents the pipeline itself and can be installed on the switch to alter its behavior.
- A `p4info.txt` file, which describes the tables defined in the pipeline, and is used by the control plane to understand how to interface with the running program.

CHIMA uses the identifier of the destination device as an output name for the files, so that new versions of a pipeline that target the same device will overwrite older ones. This happens when the set of functions it has to execute changes. At this point, the compiled pipeline can be installed on the switch.

Installing

P4Runtime [6] is an API through which control plane entities can change or interact with the components of a programmable data plane defined with a P4 program. This API consists of a set of RPCs (Remote Procedure Calls) defined in the format used by the open-source gRPC framework. Unlike platform specific APIs (such as Thrift in the case of bmv2), P4Runtime is compatible with all P4 programmable devices, ensuring this solution is portable to devices that are not currently supported by the framework.

The installation of a pipeline at runtime can be achieved with the `SetForwarding-PipelineConfig` RPC call. Rather than directly executing the procedure, CHIMA carries it out by exploiting its implementation in ONOS, which is the same used for the initial configuration of switches. Since it already relies on ONOS for the management of tables of forwarding and INT, like it was explained in Section 4.4.1, letting ONOS perform the installation of the new pipeline ensures that the correct handling of those functions is maintained after the substitution.

The process to perform the substitution involves the following steps.

- The framework fires a request to the REST API of the ONOS Stub, providing the ID of the target device, and the paths of the `json` and `p4info.txt` files obtained in the building step.
- ONOS does not work with pipelines directly, but uses data structures that integrate the P4 program with classes, called Behaviours, that instruct the controller on how high level intents can be mapped to the pipeline's tables. These aggregates are called *Pipeconfs*.
  The Stub creates a Pipeconf with the pipeline files and the necessary Behaviours, assigning it an ID that is unique to the device it will be installed on.
- The Pipeconf is bound to the device in ONOS's distributed map, and the `SetForwardingPipelineConfig` call is executed. In addition to this, the installation process involves the reconciliation of the set of rules installed in the device's tables, which is performed asynchronously.
- In the meantime, the framework polls the Stub's REST API at regular intervals to monitor the state of the installation. When the response finally becomes positive, the process is complete and the switch can be considered ready. The user provided functions in its pipeline can now be used as part of a service deployment.

Downtime

In general, the time it takes for devices to reconfigure their pipeline may cause significant downtime. However, platform specific features, especially for hardware switches, can greatly accelerate this process. For example, Barefoot Tofino programmable switches provide a feature called *Tofino Fast Refresh* [9] that grant the substitution of arbitrarily complex P4 programs in less than $50ms$.

4.5. General purpose functions deployment

For the reasons outlined in Section 3.2.1, general purpose functions must be provided by the user in the form of Docker containers. However, a single Docker file may not be enough to completely define how a container should be run. For example, the user may need to define custom bindings for the exposed ports, or run an additional containers that provide a software components used by the main one.
The definition of these settings could have been accomplished with a custom serialization schema, and executed by the framework using the Docker Engine SDK to maximize the control on how the deployment process is carried out. Instead, it was chosen to use Docker Compose, a tool for running multi-container application defined with Compose files [4]. This way to define deployments is already well known to Docker users and widely adopted by the industry. On top of its convenience, it provides every features that the framework would need from a custom solution:

- In the definition of containers, it exposes all the options that would be available to the user through the `docker run` command.
- Supports both images from the Docker Hub repository and ones that have to be built from a Docker file at the time of deployment.
- It can perform deployments on remotely exposed Docker Engines, and automatically manages the transfer of the local files needed to build containers on other hosts.

Therefore, a general purpose function can be considered completely defined by a Compose file.

4.5.1. CHIMA network

To ensure that packets sent and received by general purpose functions are properly routed by CHIMA, Compose files are processed by the framework before their deployment to add network related configuration. As will be specified in Section 4.6, each function of the same service will be assigned an IPv4 address belonging to a common subnet. This is implemented by injecting the definition of a Docker network with such subnet in the Compose file, and the addition of the address to the section that describes containers.

4.5.2. Preloading

Loading the containers that implement a function is a very time consuming operation. If images from Docker Hub are used and they are not locally available on the target host, they have to be downloaded. Of course this process depends on the size of the image and the bandwidth available to the host. Furthermore, the time to build user defined containers has to be taken into consideration.

If this process had to be executed when performance degradation is detected, it would dramatically increase the time to accomplish a redeployment. For these reasons, CHIMA performs a preloading of the functions on all hosts before the service is deployed. This way, at the time of redeployment, the only delay is the one caused by the spin up of already accessible containers. This operation is carried out with the `docker-compose build` command.

4.5.3. Running

After the containers have been preloaded, at the time of first deployment or redeployment of the service, CHIMA can start the function on the predetermined host by running a `docker-compose up` command as a subprocess. To target the correct device, the `DOCKER_HOST` environment variable is set to the address of its Docker Engine.

4.6. Routing

After all the functions have been handled, the correct routing of packets between them has to be configured along the prescribed path.

4.6.1. Addressing functions

For the service to be specified as the destination of a client's requests, it needs to be assigned an address. In the same way, general purpose functions must be able to refer to each other in order to exchange packets. The address of an application, in regular settings, is the one of the device on which it is statically deployed. However, in our case, the components of a service can be moved on different hosts at any time to satisfy requirements.

For this reason, CHIMA assigns an IP address belonging to an overlay subnet to each one of the general purpose functions. The same address is kept by the function independently from its physical location, allowing other components of the same service to communicate with it for its whole lifespan. To reach the service, the client uses the IP address of the first function of the chain, and subsequent requests are forwarded between pairs of functions. This problem does not affect P4 functions, as they are transparently executed on switches along the path of the communication used by general purpose components.

4.6.2. Routing solutions

The objective of the framework's routing is to enforce the communication path selected by the optimization model. Two solutions could be considered to achieve this purpose.

Tunneling

Tunneling protocols are commonly used to allow the transport of packets on sections of networks that may not support their network layer, or to secure the subject of communication with encryption. Examples of tunneling protocols are GRE [35], VXLAN [38] and GENEVE [23]. Their operating principle is based on the encapsulation of the original packet as the payload of the protocol's header, that is then regularly forwarded between two endpoints. As the name implies, the stream of packets among endpoints creates the illusion of a "tunnel" that an application can use to communicate with the other side, and that masks the complexity of the underlying network.

A possible solution to the framework's routing problem would be the encapsulation of packets sent between functions of a service. This could be accomplished with the creation of a tunnel whose overlay network is represented by the subnet of the service's functions, while the underlay is the physical network interconnecting devices. The endpoints of the tunnel, in the underlay network, would be the two devices that currently host the communicating functions.

In an ordinary setting, the control plane would autonomously determine how packets are routed to reach their destination by populating IP tables, even splitting packets between different routes if multipath routing algorithms are used. For this reason, to ensure that the precomputed paths are followed, CHIMA would need to manipulate the forwarding tables of each device.

This would mean working against the strengths of tunneling, that is the possibility to specify endpoints and letting the existing network layer handle routing. On top of that, this approach presents two issues:

- Enforcing the use of a particular path based on the destination device would force regular packets transiting the network to follow it as well. On top of the fact that the route is chosen according to the service's requirements and may not be the best path in general, the inclusion of other traffic on the same links may impact the measurement we are trying to comply with.
- In case two functions that belong to different services are be deployed on the same device, they may be assigned different prescribed paths depending on the expressed requirements. This cannot be enforced by using the destination alone to select a path at the time of forwarding.

Both of these problems could be solved with the addition of parameters to IP routing tables to discriminate between regular traffic and managed service traffic, and between the traffic of different services. However, this makes the solution even less practical.

Segment routing

Segment routing (SR), defined in RFC8402 [20], is a routing technique based on the source routing paradigm. In segment routing, the treatment that a packet receives from a switch is determined by a set of instructions called *segments*, which can be embedded in the packet at its source. Segments are referred to by a Segment Identifier or SID, a value whose semantic is known by both the entity that attaches it to packets and the switches that have to execute them.

Using SR it is possible to directly instruct devices on the forwarding to perform for each packet with a proper sequence of SIDs, allowing those belonging to a service to be routed on the predetermined path. It goes without saying that the granularity of control granted by SR enables the use of different paths for packets of different services even when their destination is the same, unlike the case of tunneling.

Since segments can represent any type of instruction, they can be leveraged to perform other tasks other than routing. For example, three types of segments with their own semantic are defined by the framework and are explained in detail in Section 4.6.3.

SR can be implemented on various data planes, but its specification refers to two possibilities in particular:

- SR over an MPLS data plane, also called SR-MPLS (RFC8660 [10])
- SR over an IPv6 data plane, also called SRv6 (RFC8986 [21])

Since the template P4 pipeline used by CHIMA (Section 4.4.1) is based on existing pipelines included with ONOS, and their implementation assumes the use of an IPv4 network layer, supporting SRv6 would have required to discard significant parts of them. Instead, it was chosen to develop support for SR-MPLS in order to keep the available code that interacts with components such as the `inbandtelementry` ONOS application, that are fundamental to the framework.

4.6.3. SR-MPLS

As defined by RFC8660 [10], in SR-MPLS SIDs are represented as MPLS labels. Apart from MPLS's reserved label values (0-15), SIDs can be arbitrarily mapped to the remaining values by defining Segment Routing Global Blocks (SRGB) and Segment Routing Local Blocks (SRLB). The former are SIDs whose semantic is the same across the whole segment routing domain, and have a scope that can span multiple devices. The latter, instead, are blocks of SIDs whose semantic is specific to the device that executes them.

CHIMA's implementation of SR-MPLS uses no SRGB and three SRLBs, within which only single-label SIDs are defined. Keeping in mind that MPLS labels are 20 bits long, their first two bits are used to identify their SRLB, while the remaining 18 can be interpreted as an argument for the action to be performed.

- The first local block is defined from `0x40000` to `0x7FFFF`, and is used for the execution of user defined P4 functions that have been deployed on the switch. The last 18 bits of these labels are interpreted as the unique identifier of the function to be executed.
- The second local block is defined from `0x80000` to `0xBFFFF`, and is used for packet forwarding instructions. The last 18 bits of these labels are interpreted as an identifier of the interface on which the packet has to be forwarded. These segments can be classified as Adjacency SIDs according to RFC8402 [20].
- The third and last local block is defined from `0xC0000` to `0xFFFFF`, and is used to implement a custom extension of INT for the measurement of the

delay introduced by general purpose functions. This feature will be explained in Section 4.7.6. The last 18 bits of these labels are interpreted as the ID of the function that has just been executed, and to which the measured time is assigned.

The procedure used by the template pipeline to evaluate segments is the following.

- The top-of-stack label is evaluated, and the instruction associated with its SID is determined based on the SRLB it belongs to.
- The segment is executed with the last 18 bits of the label used as an argument.
- The Segment Routing NEXT operation [10] is executed by popping the evaluated label from the MPLS stack. This also happens for Adjacency SIDs, implementing a Penultimate Hop Popping approach.

The evaluation of segments on a switch continues until new labels are available or an Adjacency SID is found. The execution of those segments ends with the forwarding of the packet to another device, which is the intended destination of the remaining labels of the stack.

Example

Figure 4.4 shows how MPLS label stacks are used by the framework to perform routing. In the beginning, the stack includes instructions for all the switches along the path. When the packet reaches switch S1, the label at the top of the stack (located next to the Ethernet header) is evaluated. Its value is a Segment ID belonging to the second local block, and is interpreted as a requests for the forwarding of the packet to an adjacent switch, that in this case is S2. For example, the value of this label could be `0x80002`. After setting the correct egress port, the label is popped from the stack, and evaluation is stopped. The next labels on the stack are intended for the following switches on the path.

At S2, the first evaluated SID belongs to the first local block, and is interpreted as the request for the execution of function 1 on the packet. An example of this label's value could be `0x40001`. Since the execution of the segment does not prescribe the forwarding of the packet, after this label is removed, the next one will be evaluated. The following SID instructs on the forwarding to S4. Analogously to what happened in S1, the egress port is set and the evaluation of further labels stops.

In S4, only a forwarding label to H1 remains. This is the Bottom Of Stack (BOS) label, and after its execution and removal, no MPLS header will be left. At this point, the EtherType field of the Ethernet header is set to IPv4 (`0x0800`), and the packet received by host H1 won't need any special treatment.

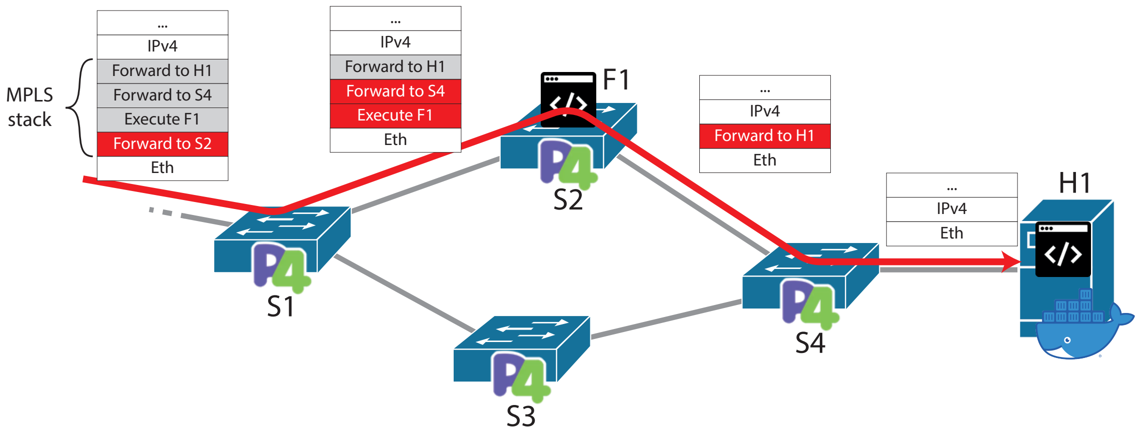


Figure 4.4: Example of the framework's use of segment routing

4.6.4. Encapsulation and segment distribution

In CHIMA, the MPLS encapsulation of packets belonging to managed services is performed by a software component installed on each of the hosts. This component is called CHIMAclient. At the heart of CHIMAclient is an eBPF filter that inspects the packets egressing the host. Packets of a particular service are identified using the tuple of source and destination IPv4 addresses. If the tuple is known, it means the filter has a label stack that represent the series of segments used to implement the precomputed path. In this case, it inserts the stack of MPLS label headers between the Ethernet and IPv4 headers. During this process, the EtherType field of the Ethernet header is set to `0x8847` to allow proper parsing of the following MPLS headers.

These stacks are computed by the CHIMA process based on the result of the optimization model, and then installed on the CHIMAclient of specific hosts by contacting their REST API. The API exposes methods to bind a label stack to a new source/destination tuple, change an existing one (in case of a redeployment) or remove a label stack entirely when the service is decommissioned.

4.7. In-band Network Telemetry

Accurately measuring the latency experienced by packets is fundamental for the objective of the framework. This is made possible by In-band Network Telemetry. This section explains how it is employed by CHIMA.

As anticipated in Section 4.4.1, the INT implementation used by the framework is derived from the `int.p4` pipeline included in ONOS, which is designed to be managed by the `inbandtelemetry` application. This implementation is based on the reference program from the INT v1.0 specification [24], and uses embedded metadata with headers located over TCP and UDP. The layout of INT data used by the framework is shown in Figure 4.5.

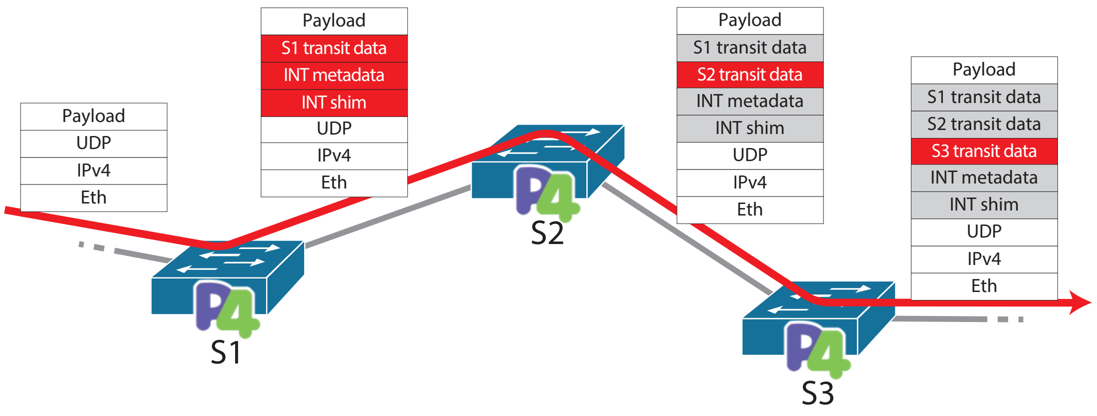


Figure 4.5: Structure of INT data embedded in transiting packets by the base pipeline. Switch S1 acts as the source, initializing shim and metadata headers.

4.7.1. Telemetry commissioning

Measurement of certain flows can be requested by the framework in two situations.

- Before the deployment of a service, to obtain measurements on the whole network. This information will be used by the optimization model to compute paths and function placements.
- After a service is deployed, to keep monitoring its performance and detect degradation.

Once the parameters of the flows are determined, CHIMA can demand the start of their measurement by contacting CHIMAstub through a REST API. The parameters of this request are source and destination IPv4 subnets, level 4 port and protocol. INT will be performed only for packets that match these specifications.

CHIMAstub creates an `IntIntent` with the mentioned information, and installs it through `inbandtelemetry`. Intents created by the stub always request embedded or "Hop-by-Hop" INT to be used, and the following metadata fields to be recorded: `SWITCH_ID`, `INGRESS_TIMESTAMP`, `EGRESS_TIMESTAMP`. These are the only fields needed to compute the metrics used by the framework.

At this point, the application will determine the rules to install on switches throughout the network to obtain the desired result, and starts their distribution. Switches will start adding metadata headers at each hop of the affected packages, and sending INT reports to the collector when the packet reaches an host.

4.7.2. Collection of INT data

The CHIMA process includes an INT collector, implemented as an eBPF filter, as one of its modules. When a report is received by the filter, the packet is parsed and measurements are computed. These values are made available to the userspace component of the collector by adding them to a hash map, that can be accessed with the helper methods of the `bcc` package. The source of the obtained values is identified by the pair of IDs of the switches at the two ends of a link.

All values collected by CHIMA are exposed to a server of the Prometheus monitoring system. This allows users to directly inspect the performance of the network and set up alerts and notifications if needed.

4.7.3. Computation of link measurements

Measurements are computed when an INT report is received by the collector, and the measurements for a particular link is updated when the report includes data regarding that link. For this reason, we can consider the series of measurements of a link to be updated in discrete time steps $t \geq 0$ with $t \in \mathbb{N}$, where $t = 0$ is the first time data for the link was included in an INT report. The values that the collector provides to the framework are not those directly computed from the report, as they can be subject to high variability. Instead, an Exponentially Weighted Moving Average (EWMA) of the raw values is updated at each time step. This is the measurement that is exposed to other modules of the system. The parameter α, which acts as a smoothing factor, can be modified by the user when running CHIMA and will greatly influence the framework's response to variations of the metrics. This point will be discussed in Chapter 5.

Latency

The latency of a link l (that connects switches a and b) at time t, is computed as the difference between the egress timestamp of the source and the ingress timestamp of the destination.

$$latency_t^l = latency_t^{a \to b} = ingress_t^b - egress_t^a \tag{4.1}$$

The obtained latency value has a resolution of nanoseconds, just like the timestamps provided by the P4 pipeline. Of course the obtained value can be considered relevant only if the clocks of the two devices are synchronized. This aspect will be treated in 4.7.4.

Defining the average latency value for link l at time t as $latency\_ewma_t^l$, the computation of the new average value is performed in the following way.

$$latency\_ewma_0^l = latency_0^l \tag{4.2}$$

$$latency\_ewma_t^l = (1 - \alpha)\, latency\_ewma_{t-1}^l + \alpha \cdot latency_t^l \tag{4.3}$$

Jitter

A value of jitter for single links is also computed by the collector. The raw jitter for link l at time t is obtained as the difference between the latency of the link at the current time step and the last available one.

$$jitter_t^l = latency_t^l - latency_{t-1}^l \tag{4.4}$$

Even though jitter is usually considered as the absolute value of this amount, the sign is retained to allow the computation of a path-wise jitter by summing the jitter of single links, as shown in Section 4.7.7.

Defining the average jitter value for link l at time t as $jitter\_ewma_t^l$, its value is computed as follows.

$$jitter\_ewma_0^l = 0 \tag{4.5}$$

$$jitter\_ewma_t^l = (1-\alpha)\, jitter\_ewma_{t-1}^l + \alpha \cdot jitter_t^l \tag{4.6}$$

4.7.4. Switches synchronization

Synchronization of clocks between separate devices is a well known problem, and the computation of latency performed by the collector directly depends on the ability of the switches to overcome it.

As stated in Section 3, the project was tested using bmv2 software switches, for which this concern is mentioned in the provided documentation [2]. The internal clock of these switches is initialized at 0 at their startup, making their timestamps represent the number of elapsed nanoseconds from the moment they started running. Of course this values cannot be used to collect relevant data. Instead, as suggested by the documentation, a patch to use the system clock as initial value for the switch's clock was introduced to the bmv2 switches used to test the project. Naturally, this approach is made possible by the fact that the instances of bmv2 used for testing run on a single machine, whose system clock acts as a global clock. In case the switches were deployed on different machines, or in the much more relevant case of hardware switches, this would not be possible.

The synchronization of real switches, instead, can be achieved with multiple protocols that are commonly used in the industry. The most notable mentions are NTP (Network Time Protocol) and PTP (Precision Time Protocol) [47]. However, while NTP is capable of a synchronization accuracy of $\sim$ 1 to $100ms$, the nanosecond resolution of INT timestamps would require the higher accuracy provided by PTP ($\sim 100ns$ to $1\mu s$) [57]. Synchronization protocols that directly exploit programmable data planes and run on P4 switches such as those required by the framework have also been designed, and could achieve an accuracy in the order of tens of nanoseconds [31].

4.7.5. MTU concerns

The addition of INT headers at each hop (12 bytes in the case of metadata used by the framework) could cause the packet size to exceed the allowed MTU in situations where the number of hops in the path is large or the original packet size is already close to the limit. The only measure taken by the current version of the project to mitigate this problem is to avoid initializing or adding telemetry data in case its length would cause the packet size to become greater than the MTU. This solution is not optimal, since it would cause the framework to lose the measurements associated with packets affected by this conditions.

However, multiple solution could be adopted in a real setting.

- Instead of the INT v1.0 specification, that stores telemetry data in the packet itself until a report is created, the modes defined in newer versions of the specification could be used. An example is the INT-XD or "Postcard" mode defined in the INT v2.1 specification [25]. The behavior of this mode is to directly send new telemetry data to the collector with a report packet, without the need for any modification of the monitored packet.
- In case the interfaces of the devices in the network support them, Ethernet jumbo frames could be enabled, making the transmission of much bigger packets possible.

4.7.6. Measurement of general purpose function times

One of this project's targets is to enable its users to enforce an upper bound on values such as the time it takes for a packet of a request to reach a particular function. The latency of links is not enough to evaluate this amount of time, since a significant part of it is due to the time needed by all previous functions to complete their execution. While P4 functions are guaranteed to run in constant time thanks to the peculiar features of the hardware they are installed on, general purpose functions cannot assure the same level of stability. Therefore, a technique to measure the time of their execution had to be designed.

The time of execution of a general purpose function can be defined as the difference between the time at which the function starts managing a request and the time at which the request is completed. By this definition, applications where a request can span multiple packets would make it necessary to determine which packets belong to the same sequence and to evaluate them as a whole, introducing issues such as their buffering. This is especially true for TCP applications, where packets represent sections of a communication stream. For this reason, the current state of the project focuses on the measurement of UDP functions in which each packet contains a single execution request. This means that we can correlate the time needed by the packet to traverse the function's host and the time of execution of the function itself.

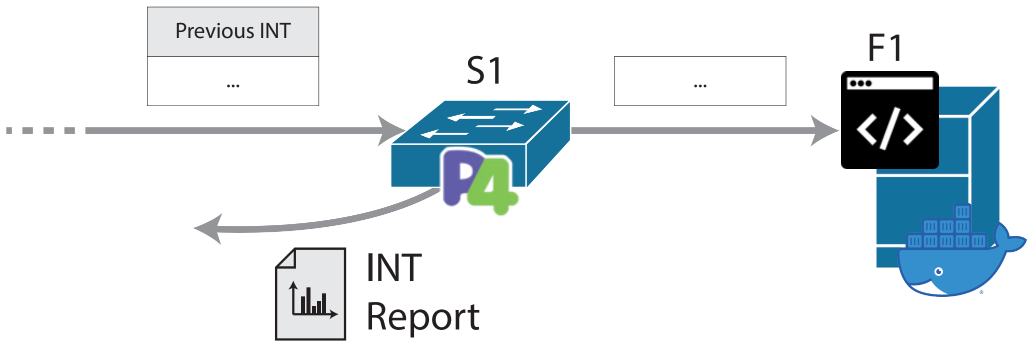
INT extension

CHIMA reaches this goal by extending the INT pipeline. The third Segment Routing Local Block defined in Section 4.6.3, with label values between `0xC0000` and `0xFFFFF`, is used to implement this feature.

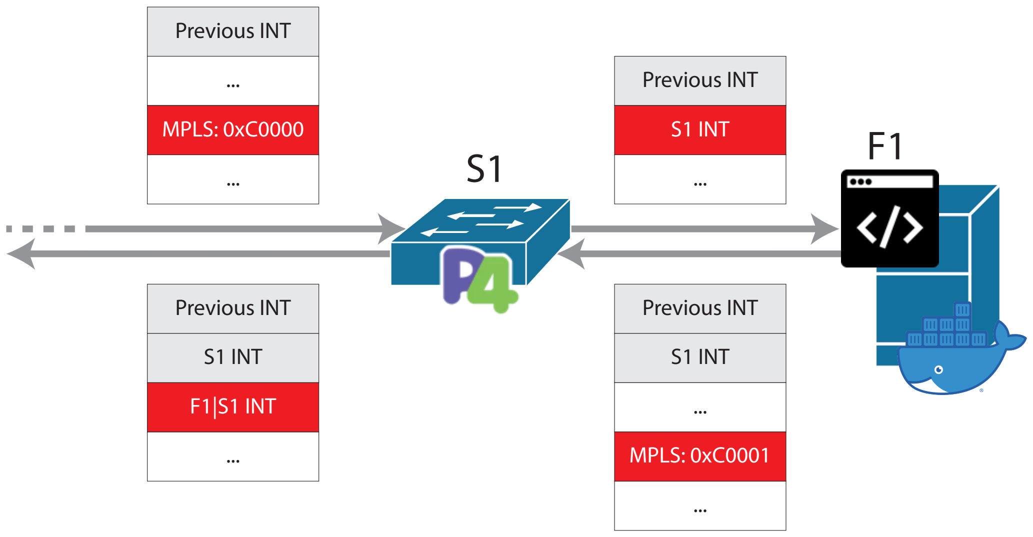
Usually, when the pipeline is about to forward a packet with embedded INT data to a host, the data is removed and an INT report is created and sent to the collector, as shown in Figure 4.6a. This operation makes the packet received by regular devices identical to the one that was sent by its source, allowing the use of In-band Network Telemetry to be transparent from their point of view. This behavior can be altered with SID `0xC0000`. The execution of this SID informs the switch not to act as a sink when forwarding the packet to a host. This means existing INT data will be left in the packet, and new INT transit headers will be added before the packet's egress. The egress timestamp included in these headers will be considered the start of the function's execution.

At this point, since the pipeline implements INT over UDP, the function will receive a packet whose payload includes all INT headers accumulated so far. This fact requires the function to be aware of this data, and to actively cooperate to the measurement. The function must only take the real payload into consideration by reading the size of INT data from the INT shim header [24] and skip ahead by that amount. This step can be accomplished by an explicit declaration of this logic by the developer, or with the use of a library that transparently performs these operations while providing the developer with the same output it would expect from native function calls. The resulting packet will include the INT data collected up to that point and the payload modified by the function. When it will be emitted, CHIMAclient will encapsulate it with the appropriate label stack that will include a segment to record the end of the execution time, as shown in Figure 4.6b.

This segment will be located at the top of the stack, so that the first receiving switch will execute it. The value of the last 18 bits of the label is a unique identifier of the function, that will be used by CHIMA to match the computed execution time to the correct task. The presence of such a segment instructs the pipeline to perform a bitwise OR of its value with the 18 most significant bits of the 32 bits wide `switch_id` transit header [24]. `switch_ids` are assigned to devices by the `inbandtelemetry` ONOS application and, considering that the maximum number of supported switches is 99 (using the least significant 7 bits at most), the two values will never overlap. The same result could be obtained by adding two sets of INT transit headers in the first hop: one for the switch, and one for the function. However, since the included timestamps would have the exact same value, it was chosen to fully exploit the available and unused bits of the `switch_id` field with this solution. The ingress timestamp embedded by this switch will mark the end of the function's execution, enabling the computation of its extent.



(a) Regular processing of an INT packet when forwarded to a host



(b) Additional measurement of the function's time using Segment IDs

Figure 4.6: Comparison of the content of packets and the forwarding behavior with regular INT and with CHIMA's extension

4.7.7. Computation of path measurements

Requirements set by users on the whole service or a function don't refer to single links, but to section of the service's path. To check if the requirements are not exceeded, path-wise measurements of the corresponding quantities must be computed from the ones of single links. During this process, the performance of general purpose functions executed along the path, obtained with the method from Section 4.7.6, is handled as if they were links to traverse. The considered section of path for each of the requirements is determined as explained in Section 4.2.1.

Total latency

The total latency along a path is computed as the sum of the latency measurements of the links that compose the path.

Assuming the set of links belonging to the path is denoted as P, the value at time t is:

$$total\_latency_t^P = \sum_{l \in P} latency\_ewma_t^l \tag{4.7}$$

Total jitter

Keeping in mind that the values of jitter provided by the collector are signed, the jitter of a path can be computed as the sum of the jitter values on the links that compose the path.

Assuming the set of links belonging to the path is denoted as P, the value at time t is:

$$total\_jitter_t^P = \left| \sum_{l \in P} jitter\_ewma_t^l \right| \tag{4.8}$$

Latency variation

The latency variation refers to the difference between the minimum and maximum value of total latency ever recorded for a path. Its computation is achieved by keeping a record of the two extreme values for each of the paths that are subject to this requirement.

Assuming the set of links belonging to the path is denoted as P, the value at time

n is:

$$latency\_variation_n^P = \max_{t\in[0,n]} \{total\_latency_t^P\} - \min_{t\in[0,n]} \{total\_latency_t^P\} \quad (4.9)$$

4.8. Redeployment

Each time path-wise metrics are computed, CHIMA compares the obtained values with the corresponding requirements. As soon as one of these limits are exceeded, the redeployment process is started in an attempt to mitigate it. The course of action that leads to the detection of such states is represented in Figure 4.7. The same would happen in response to a topology update that disrupts the path used by a service. The focus of this procedure is the speed at which the service can be brought back to a requirement compliant state.

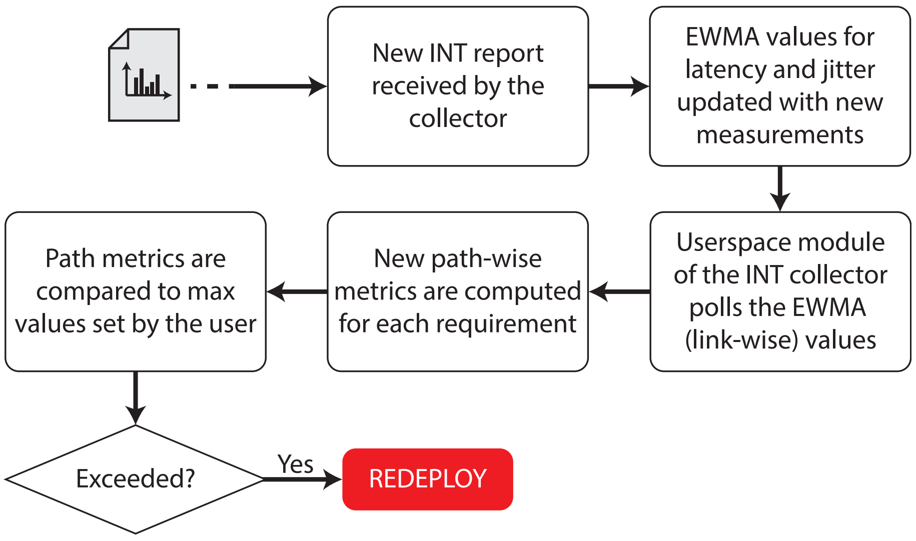

New INT report received by the collector
EWMA values for latency and jitter updated with new measurements
Userspace module of the INT collector polls the EWMA (link-wise) values
New path-wise metrics are computed for each requirement
Path metrics are compared to max values set by the user
Exceeded?
Yes
REDEPLOY

Figure 4.7: Series of events that lead to the detection of an unmet requirement

First of all, the computation of a new placement for the affected service's functions and their routing has to be carried out. Besides ensuring that the new deployment meets every requirement, the model used in this process should minimize the number of components to move, since the deployment time of functions of both types is the most significant contribute to the time for a redeployment to take place. This will be outlined in Section 5.

When a new optimal placement becomes available, the framework must determine which components already reside on the correct device and which have to undergo a new deployment. This is done by comparing the new direction with the internal data structures that describe the current state of the service. Such portions of metadata are moved from the old deployment instance to the new one, leaving behind only the data of components to be removed after the redeployment is complete.

At this point, the actual deployment of moving functions must be carried out, as described in Sections 4.4 and 4.5. The necessary procedures for each of the target devices are performed simultaneously. In the meantime, if the redeployment wasn't caused by the complete unavailability of a link, the original deployment of the service is not affected and is still able to serve the client, even though its performance is degraded.

After the completion of all parallel installations, the communication can be safely steered towards the newly deployed functions by updating the label stacks used by hosts. This is achieved with requests to the API exposed by relevant instances of CHIMAclient. With this step the performance of the service is finally restored.

4.8.1. Cleanup

After all time-sensitive tasks are completed, unused components of the old deployment must be removed to reclaim resources. In practice, this only involves tearing down the Docker containers of general purpose functions. P4 functions installed on switches don't impact the device's performance if their execution is not requested through a Segment ID, and will be removed with the installation of a new program if needed at a later time.

5 | Evaluation

Measurements for the evaluation of the framework's performance and an assessment of its success in satisfying objectives will be presented in this chapter.

5.1. Methodology

Among the objectives set in Section 3.3, only the last two can be objectively measured. Both refer to the amount of time needed by the framework to complete operations that are crucial for the success of a redeployment. For this reason, a method to measure the time at which different events happen across components of the simulated system had to be put in place.

5.1.1. Collection of measurements

The code of the framework's components have been instrumented so that when an event that is relevant to the measurement of redeployment times happens, a timestamp with nanosecond resolution will be taken. These timestamps, with the corresponding description of the events they refer to, are written to a single file that can be parsed at a later time to extract precise time spans between episodes. During the automated execution of test cases, that will be explained shortly, a record of these amounts is saved with information about the setup that generated them, including the topology used and the values of variable input parameters.

The complete list of measurements of a test run is shown in Figure 5.1.

Each of the data points presented in the following sections has been obtained as the mean value of 30 samples.

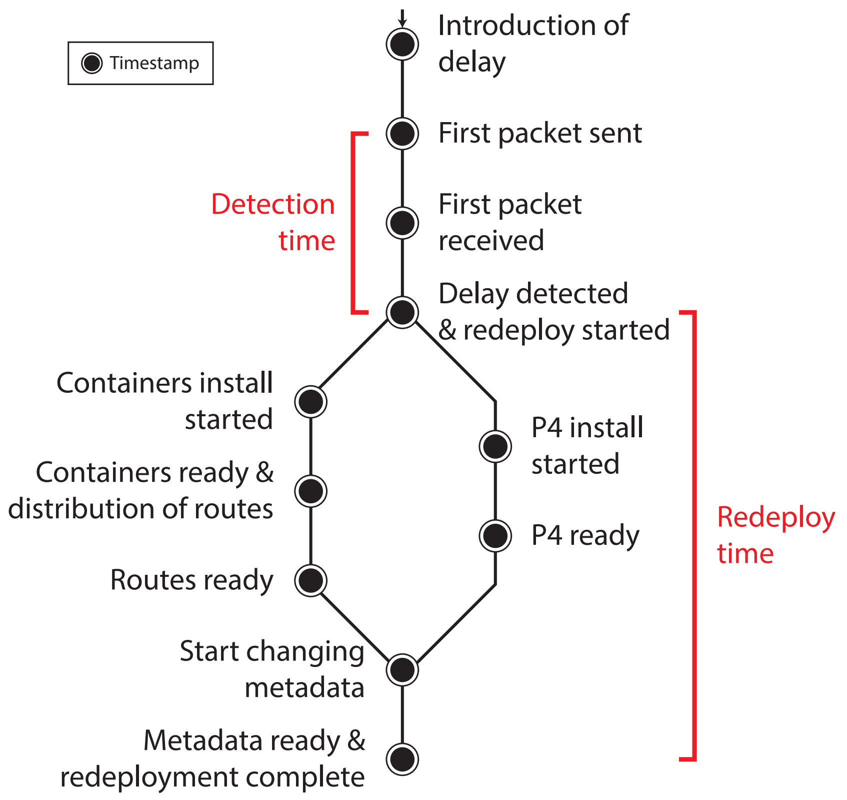


Figure 5.1: Representation of the timestamps collected during one test run, and their relationship in time. The time spans considered for the computation of detection and redeployment performance are highlighted in red.

5.1.2. Test setup

Running a test case to measure the performance of a redeployment involves the execution of multiple steps over different processes. Since the order of these operations is important for the collection of meaningful data, a parametric `expect` script has been developed to easily and consistently run them. During these operations we assume that ONOS is running and all needed applications (including CHIMAstub) have been activated on it.

Topology

First of all, a fresh instance of the target test topology must be started. Even if successive tests are performed for the same topology, the older instance is torn down and started again to avoid possible influences on performance.
Topologies are simulated in FOP4 [43], an extension of mininet that allows the use

of Docker containers as hosts and bmv2 [1] instances as switches. The hosts used in these tests are custom Docker-in-Docker (or *dind* for short) images, modified to allow the installation of the eBPF filter used by CHIMAclient with the `bcc` python package. These containers are configured to expose their Docker engine as specified in Section 3, and to run CHIMAclient at their startup.
After the topology has started up completely, a `pingall` command is used to make hosts exchange packets in order to allow ONOS to detect them.
One of the interfaces added to the switches of the simulated network will be a virtual interface that links the simulation namespace and the root network namespace of the machine. In this way, it will be possible for regularly run programs to act as clients for the deployed application.

CHIMAclient and CHIMA

As stated above, thanks to the virtual interface, programs that should be run by a client device can be started without additional configuration. The first one will be an instance of CHIMAclient, that will bind to the virtual interface to install its eBPF filter on it. CHIMA is started as well. It will registers to CHIMAstub to collect topology information and perform initial telemetry on the communications of the network. In order to receive INT reports, it will bind the eBPF component of its collector to the same interface.

Test service

At this point, everything is in place for the deployment of a test application. This dummy service has been designed with the goal of using the same client logic across all tests, but with the possibility of using a variable number of functions depending on the desired complexity for the particular case. The client application sends a packet with a string as payload, and expects a modified version of the string (on which each function applies a different transformation) to be returned.
A command to request the deployment of the service is fired to CHIMA, and as soon as its deployment is completed, communication can start.

Introduction of the perturbation

After the completion of the deployment, the client application can be started. Packets are sent at regular intervals, and CHIMA starts collecting measurements of their performance.

To simulate the introduction of a perturbation that will cause requirements to be unsatisfied by the current state of the service, an extension of FOP4's CLI has been written to allow the introduction of delays on particular links. When it is executed, the first of the timestamps described in Section 5.1.1 will be recorded. Since telemetry data is carried by packets, the time at which the first packet is sent after the introduction of delay is relevant. This event, together with the time at which it is received, are timestamped as well.

Redeployment

When the metrics computed by CHIMA finally exceed one of the requirements, a timestamp for the moment of the detection is recorded. At the same time, a redeployment is triggered. As explained in Section 4.1 the new target placement for the service is already known, and only has to be deployed. Various steps during this process are timestamped, some of which happen in parallel, as described in 4.8 and shown in Figure 5.1. When the new state of the service becomes ready, one last timestamp is taken.

Cleanup

After a successful redeployment, measurements for the test have been completed. All the processes that have been started for the test can be shut down. This includes the topology simulation, CHIMAclient, CHIMA and the dummy service's client.

5.1.3. Hardware

All measurements have been performed on a bare-metal installation of Ubuntu 20.04 LTS, running on an Intel Core i7-6700 CPU with 64GB of RAM.

5.2. Measurements

5.2.1. Detection delay

The first set of measurements have the objective of determining how much time is needed by the framework to detect the introduction of a perturbation, depending on the values of user-configurable parameters.

As highlighted in Figure 5.1, the detection delay is computed as the elapsed time between the first packet sent after the beginning of the adverse event, and the detection of an exceeded requirement by CHIMA. There are multiple reasons for this choice.

- Since the framework uses embedded INT for telemetry, information on the performance of a path cannot be known until a packet is sent on that path. At the same time, the service is not affected by the change until some communication is performed.
- The time span between the artificial introduction of a delay and the first chance of receiving telemetry data is highly dependent on the frequency at which the service performs transmission, which is not a characteristic of the framework itself.

The time spans that contribute to this amount are laid out in Figure 4.7. Assuming the properties of topology and service to be constant, we can consider the detection delay as defined above to be a function of the polling interval and the EWMA coefficient.

The topology and service used for these test is shown in Figure 5.2. The service is initially deployed on the devices of one branch of the topology. Green functions belong to this original layout, and green links are those carrying packets for their communication. A lightning bolt marks link on which an artificial delay that exceeds the maximum latency requirement on the first function is introduced. The redeployment procedure will transition the service to the state highlighted with orange functions and links. In all tests, the application is configured to generate a packet every $100ms$.

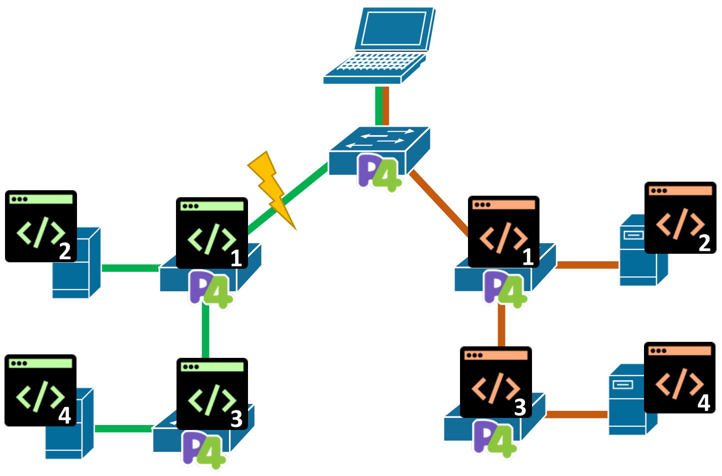


Figure 5.2: *minimal* test case with 4 functions, 2 of which are general purpose and 2 P4 ones. All the functions are relocated after the introduction of a delay

Polling interval

The first parameter that influences detection times is the rate at which the userspace component of the eBPF INT collector in CHIMA polls new EWMA values of latency and jitter. Apart from the varying value for the polling interval, these measurements have been performed with a value of $\alpha = 2^{-3}$ for the computation of the EWMA.

In the same figure, the time taken by a request to traverse the function chain end-to-end is plotted in red. This duration is only a characteristic of the studied service and topology, but can have important effects on the detection time, delaying the moment in which the first affected INT report is received.

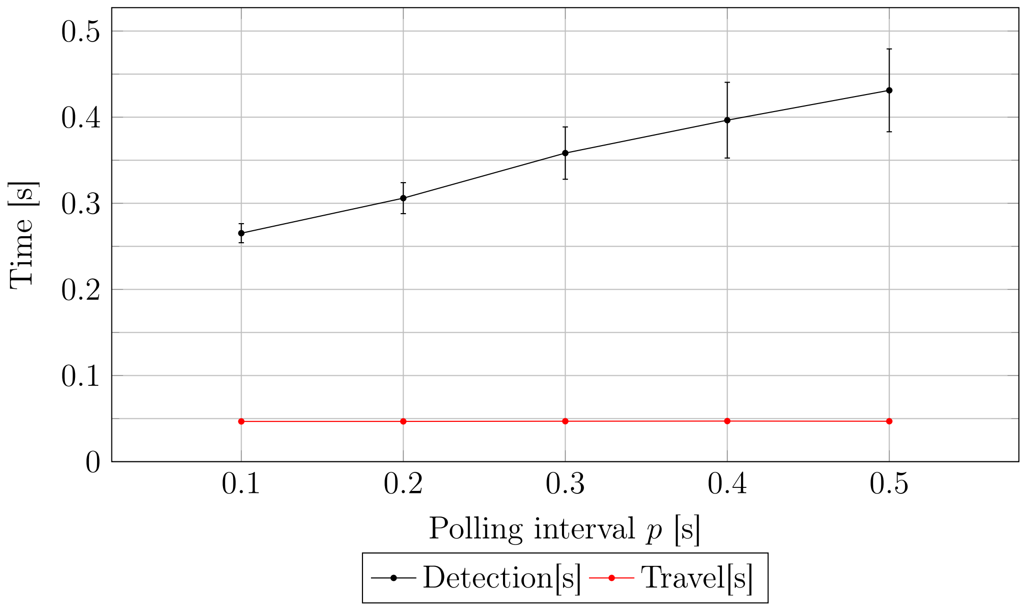


Figure 5.3: Delay in the detection of an exceeded requirement with different intervals for the polling of new measurements from the INT collector. 95% CI.

As expected, the results of Figure 5.3 present a clear linear trend, directly proportional to p.
The recorded values can be thought as the sum of two contributes. The first one is represented by the time needed for the EWMA of the affected measurement to converge to a value that surpasses requirements. This only depends on the used value of α, which is constant for the whole graph. The second one is the delay caused by the polling interval. Since the moment in which the EWMA reaches the threshold in kernel space is independent from the one in which it will be polled by the userspace, its expected value will be equal to $\frac{p}{2}$ for a polling interval of p. This fact is reflected by the slope of the curve, for which $|d_1 - d_2| \approx \left|\frac{p_1}{2} - \frac{p_2}{2}\right|$, where d_n is the detection delay measured with a polling interval of p_n.

These measurements reveal that to achieve minimal detection times, the lowest value of p that doesn't cause excessive system load should be used.

EWMA coefficient

The second examined parameter is the coefficient used in the computation of the Exponentially Weighted Moving Average, explained in Section 4.7.3. While running these tests, the Polling interval has been set to $0.1s$.

Since the computation of EWMA is performed in an eBPF filter, and it has to be computed for each of the measurements of a received INT report, its performance is crucial. For this reason, it is implemented with bit-shift operators, and only allows users to configure exponent $k \in \mathbb{N}$ where $\alpha = 2^{-k}$.

As for the previous case, the travel time of packets is plotted in red.

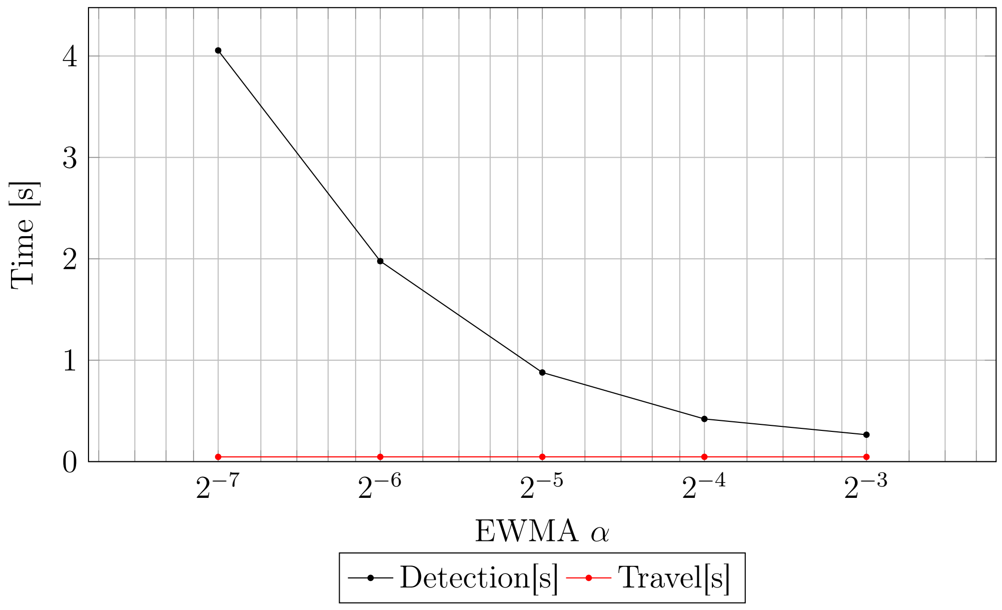


Figure 5.4: Delay in the detection of an exceeded requirement with different coefficients for the computation of the EWMA on link measurements. 95% CI.

In the computation of new values, detailed in Section 4.7.7, greater α results in more weight given to recent data rather than the old average. This is clearly shown by Figure 5.4, in which smaller coefficients cause the time needed for convergence to the new value of latency to grow exponentially.

This data confirms the effectiveness of the use of α as a smoothing parameter, that can be used to tune the response of the framework in case of short-lived congestion events. A high value will correspond to a very fast response, but could be too aggressive and cause frequent and unneeded redeployments. Therefore, the optimal value for this parameter should be determined based on the intended application.

5.2.2. Redeployment time

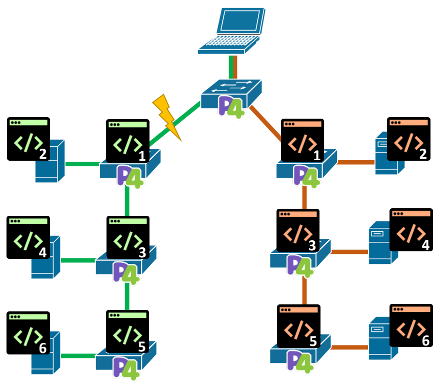
Another crucial measurement to outline the framework's performance is the time needed to complete a redeployment. Of course this time is heavily dependent on the considered topology and service. The test cases used to obtain the following values are, besides the *minimal* topology of Figure 5.2, the ones shown in Figure 5.5. The *medium* (Figure 5.5a) and *large* (Figure 5.5b) cases are extensions of *minimal*, causing the redeployment of 6 and 8 functions respectively. Instead, the *mesh* case (Figure 5.5c) is much simpler and only needs the redeployment of a general purpose function in the middle of the chain. The *unbalanced* case (Figure 5.5d) presents a situation in which many functions are deployed together, but the redeployment involves only one function of each kind. Finally, the *datacenter* (Figure 5.5e) case only needs the redeployment of a P4 function, thanks to the highly redundant spine leaf topology.
Table 5.1 presents a comparison of the relevant characteristics between test cases.

Topology	Switches	Containers	P4 functions
mesh	7	3 (1)	0 (0)
datacenter	6	2 (0)	1 (1)
unbalanced	4	4 (1)	2 (1)
minimal	5	2 (2)	2 (2)
medium	7	3 (3)	3 (3)
large	9	4 (4)	4 (4)

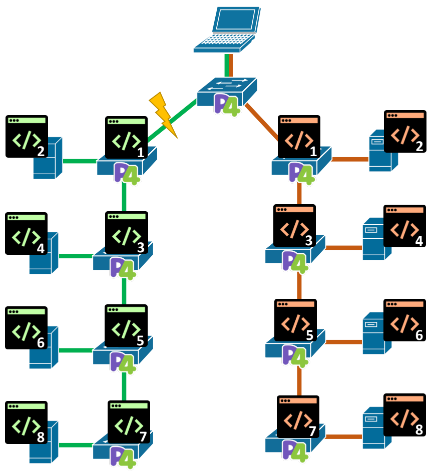
Table 5.1: Characteristics of the presented test cases. The number of functions that will be moved in each case is stated in parenthesis.

The results presented in Figure 5.6 show the variation of the total redeployment time for these services and topologies, along with the most significant contributing factors. Additionally, all recorded contributes are detailed in Table 5.2.
The temporal relationship between these sums is highlighted in Figure 5.1: while the adjustment of internal metadata is executed after all other operations have been completed, the installation of P4 functions happens in parallel with the startup of containers and the distribution of new paths. For a correct evaluation, it's also important to keep in mind that the installation of multiple functions of the same type is executed simultaneously rather than sequentially. This means the recorded times will be equal to the delay caused by the slowest one.

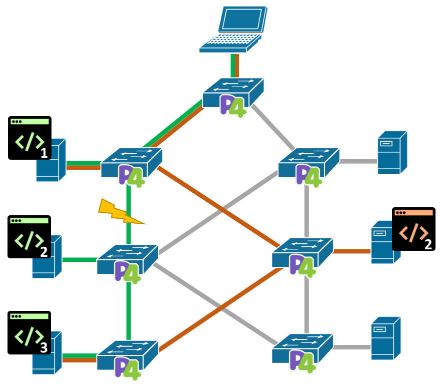
Apart from the *mesh* case, with only a general purpose function to be moved, chang-



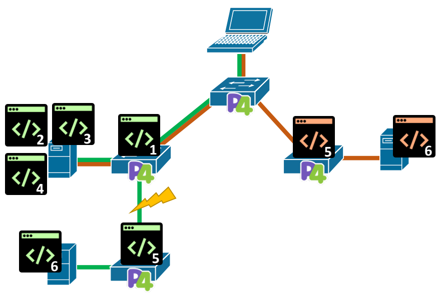
(a) *medium* test case



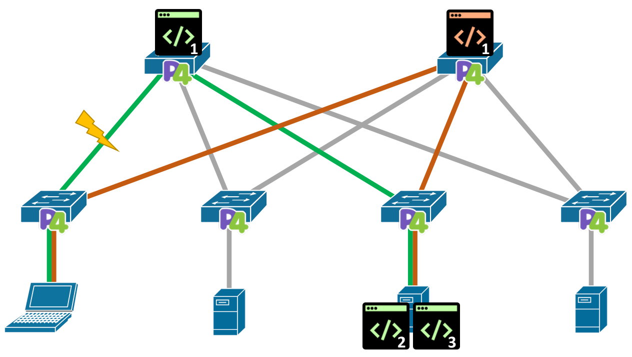
(b) *large* test case



(c) *mesh* test case



(d) *unbalanced* test case



(e) *datacenter* test case

Figure 5.5: Representations of the additional test cases used to study redeployments times

ing the P4 program of affected switches proves to be the dominant factor across the redeploy procedure, causing the total time to be in the order of seconds. As explained in Section 4.4, this operation is executed by ONOS. The two contributing factors to this delay are the time needed to reconfigure the switch's pipeline, and that required to reinstall the correct set of rules in the pipeline's tables.
While the former is caused by the use of the bmv2 software switch, and could be reduced to tens of milliseconds with specific hardware features (Section 4.4.2), the latter is mainly due to the fact that ONOS's management of programmable data planes is not structured for frequent and time sensitive pipeline changes. The introduction of improvements to target this specific use case could drastically decrease delays.

The second biggest time is caused by the startup of Docker containers, which while being in the same order of magnitude, always result to be significantly faster than P4 functions.
This time is due to the use of Docker Compose for their management, explained in Section 4.5. The implementation of a custom solution with a focus on spin-up performance could lower this amount.

Finally, the times for path distribution and metadata adjustment can be entirely attributed to the framework's logic. New paths for each changed route must be sent to CHIMAclient processes through API calls, while the changes applied to internal data structures are needed to keep track of the new state of the service.
These values are much less significant than previous ones, and thus can be considered adequate for the objective of minimizing overhead.

Topology	**P4[s]**	**Containers[s]**	**Paths[ms]**	**Metadata[ms]**
mesh	-	1.13 [1.09,1.17]	54.23 [49.16,59.31]	2.68 [2.55,2.81]
datacenter	5.20 [5.17,5.23]	-	288.60 [283.30,293.90]	2.79 [2.63,2.95]
unbalanced	5.19 [5.16,5.21]	0.97 [0.96,0.98]	60.57 [58.09,63.04]	2.81 [2.64,2.97]
minimal	6.06 [6.02,6.10]	1.50 [1.48,1.51]	48.79 [43.78,53.81]	3.50 [3.31,3.69]
medium	6.93 [6.90,6.97]	2.12 [2.09,2.14]	125.89 [107.94,143.84]	4.81 [3.84,5.78]
large	8.28 [8.19,8.36]	2.54 [2.46,2.63]	477.55 [387.49,567.62]	6.10 [4.84,7.35]

Table 5.2: Breakout of redeployment times for different topologies. 95% CI.

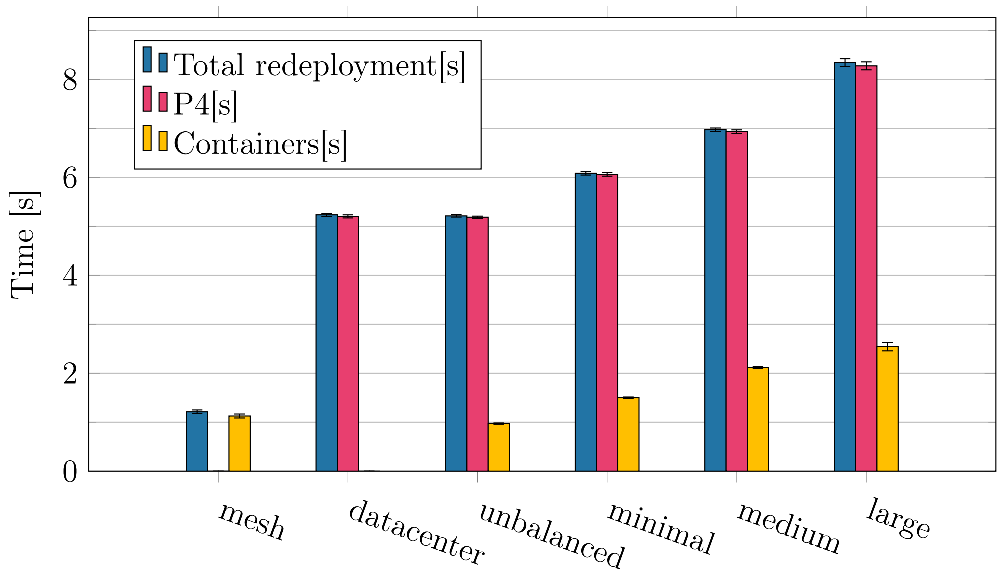


Figure 5.6: Time for the complete redeployment of a service, along with the contributes of P4 and Container redeployment, in different test cases. 95% CI.

6 | Conclusion

In this thesis, a framework for the deployment, monitoring and realtime readjustment of heterogeneous SFCs has been proposed.
The ability to define functions for programmable switches, with the use of an extensible template pipeline, grants the possibility to accelerate services by offloading sections of their computation to the network, while locating them as close as possible to their client to reduce latency. The use of Segment Routing, implemented on top of MPLS, ensures that the communication between functions will follow predefined paths, computed according to the requirements set by users. The same technique is used to request the execution of P4 functions on transiting packets. Programmable data planes are also exploited to perform In-band Network Telemetry, which ensures accurate monitoring with hop-by-hop resolution for every packet of a targeted communication. At first, it is used to collect information on the latency and jitter of links, which can be used to determine the best placement and routing for the functions of a performance-constrained SFC. After its deployment, while the service is running, INT collects metrics on its packets specifically. An extension of this process has been proposed to also measure the time required for the execution of general purpose functions. This allows the constraints to reflect real delays experienced by packets, and not just ones caused by the network. All sources of data are combined to to guarantee that the thresholds set for latency, jitter and latency variation will be corrected as soon as a violation is detected, by triggering a process of redeployment. In those cases, functions will be moved to a new location that will restore the desired performance with minimal overhead.

A prototype, released on GitHub [3], has been developed and tested through simulations on the FOP4 platform with bmv2 software switches, to show that the detection of exceeded requirements happens in the order of hundreds of milliseconds, and can be tuned by the user to achieve the desired level of responsiveness. Analysis of the redeployment process showed that the overhead introduced by the system is negligible compared to the time needed for the startup of functions, and real time relocation of VNFs to achieve desired levels of performance is feasible.

Further work on the proposed solution could follow various directions:

- Testing the framework on networks with hardware programmable switches, instead of software ones, would provide closer results to real world performance. Taking advantage of platform specific features redeployment times of P4 functions could be lowered significantly, reducing delays in the restoration of a requirement compliant state.
- VNF placement algorithms proposed in literature could be introduced in the phases of initial deployment and runtime redeployment, evaluating which would provide the best results, taking into consideration user-set requirements and the limited time available when reacting to adverse events.
- The solution used for the measurement of execution time of general purpose functions could be extended to support more complex scenarios, for instance services whose requests span multiple packets.

Bibliography

[1] Behavioral model (bmv2), . URL https://github.com/p4lang/behavioral-model.

[2] Bmv2 timestamp implementation notes, . URL https://github.com/p4lang/behavioral-model/blob/27c235944492ef55ba061fcf658b4d8102d53bd8/docs/simple_switch.md#bmv2-timestamp-implementation-notes.

[3] Chima: Chain installation, monitoring and adjustment. URL https://github.com/ANTLab-polimi/CHIMA.

[4] Compose file. URL https://docs.docker.com/compose/compose-file/.

[5] P4 ecosystem. URL https://p4.org/ecosystem/.

[6] P4runtime specification. URL https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html.

[7] P4 16 language specification. URL https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.

[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)*, pages 171–177. IEEE, 2015.

[9] A. Bas. Leveraging stratum and tofino fast refresh for software upgrades. *Accessed: Jul*, 4:2021, 2018.

[10] A. Bashandy, C. Filsfils, S. Previdi, B. Decraene, S. Litkowski, and R. Shakir. Segment Routing with the MPLS Data Plane. RFC 8660, Dec. 2019. URL https://rfc-editor.org/rfc/rfc8660.html.

[11] D. Bhamare, A. Kassler, J. Vestin, M. A. Khoshkholghi, and J. Taheri. Intopt: In-band network telemetry optimization for nfv service chain monitoring. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2019.

[12] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[13] F. E. R. Cesen, L. Csikor, C. Recalde, C. E. Rothenberg, and G. Pongrácz. Towards low latency industrial robot control in programmable data planes. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pages 165–169. IEEE, 2020.

[14] X. Chen, D. Zhang, X. Wang, K. Zhu, and H. Zhou. P4sc: Towards high-performance service function chain implementation on the p4-capable device. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 1–9. IEEE, 2019.

[15] D. Cho, J. Taheri, A. Y. Zomaya, and P. Bouvry. Real-time virtual network function (vnf) migration toward low network latency in cloud environments. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 798–801. IEEE, 2017.

[16] N. Choi, L. Jagadeesan, Y. Jin, N. N. Mohanasamy, M. R. Rahman, K. Sabnani, and M. Thottan. Run-time performance monitoring, verification, and healing of end-to-end services. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 30–35. IEEE, 2019.

[17] G. Creech. Black channel communication: What is it and how does it work? *Measurement and Control*, 40(10):304–309, 2007.

[18] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 28(4):1726–1738, 2020.

[19] R. Eaton, J. Katupitiya, K. Siew, and K. Dang. Precision guidance of agricultural tractors for autonomous farming. In *2008 2nd annual IEEE systems conference*, pages 1–8. IEEE, 2008.

[20] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir. Segment Routing Architecture. RFC 8402, July 2018. URL `https://rfc-editor.org/rfc/rfc8402.html`.

[21] C. Filsfils, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima, and Z. Li. Segment

Routing over IPv6 (SRv6) Network Programming. RFC 8986, Feb. 2021. URL `https://rfc-editor.org/rfc/rfc8986.html`.

[22] N. Finn, P. Thubert, B. Varga, and J. Farkas. Deterministic Networking Architecture. RFC 8655, Oct. 2019. URL `https://rfc-editor.org/rfc/rfc8655.html`.

[23] J. Gross, I. Ganga, and T. Sridhar. Geneve: Generic Network Virtualization Encapsulation. RFC 8926, Nov. 2020. URL `https://rfc-editor.org/rfc/rfc8926.txt`.

[24] T. P. A. W. Group et al. In-band network telemetry (int) dataplane specification - version 1.0, 2018. URL `https://github.com/p4lang/p4-applications/raw/master/docs/INT_v1_0.pdf`.

[25] T. P. A. W. Group et al. In-band network telemetry (int) dataplane specification - version 2.1, 2020. URL `https://github.com/p4lang/p4-applications/raw/master/docs/INT_v2_1.pdf`.

[26] J. M. Halpern and C. Pignataro. Service Function Chaining (SFC) Architecture. RFC 7665, Oct. 2015. URL `https://rfc-editor.org/rfc/rfc7665.txt`.

[27] 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*, pages 35–49, 2016.

[28] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth. A survey on data plane programming with p4: Fundamentals, advances, and applied research. *arXiv preprint arXiv:2101.10632*, 2021.

[29] M. He, A. Basta, A. Blenk, N. Deric, and W. Kellerer. P4nfv: An nfv architecture with flexible data plane reconfiguration. In *2018 14th International Conference on Network and Service Management (CNSM)*, pages 90–98. IEEE, 2018.

[30] K. Joshi and T. Benson. Network function virtualization. *IEEE Internet Computing*, 20(6):7–9, 2016.

[31] P. G. Kannan, R. Joshi, and M. C. Chan. Precise time-synchronization in the data-plane using programmable switching asics. In *Proceedings of the 2019 ACM Symposium on SDN Research*, pages 8–20, 2019.

[32] 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*, 25 (6):2191–2205, 2020.

[33] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, volume 15, 2015.

[34] J. Lee, H. Ko, H. Lee, and S. Pack. Flow-aware service function embedding algorithm in programmable data plane. *IEEE Access*, 9:6113–6121, 2020.

[35] T. Li, D. Farinacci, S. P. Hanks, D. Meyer, and P. S. Traina. Generic Routing Encapsulation (GRE). RFC 2784, Mar. 2000. URL `https://rfc-editor.org/rfc/rfc2784.txt`.

[36] J. Liu, Z. Jiang, N. Kato, O. Akashi, and A. Takahara. Reliability evaluation for nfv deployment of future mobile broadband networks. *IEEE Wireless Communications*, 23(3):90–96, 2016.

[37] J. Liu, W. Lu, F. Zhou, P. Lu, and Z. Zhu. On dynamic service function chain deployment and readjustment. *IEEE Transactions on Network and Service Management*, 14(3):543–553, 2017.

[38] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348, Aug. 2014. URL `https://rfc-editor.org/rfc/rfc7348.txt`.

[39] J. Marques, K. Levchenko, and L. Gaspary. Intsight: diagnosing slo violations with in-band network telemetry. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 421–434, 2020.

[40] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.

[41] M. Mechtri, C. Ghribi, O. Soualah, and D. Zeghlache. Nfv orchestration framework addressing sfc challenges. *IEEE Communications Magazine*, 55(6):16–23, 2017.

[42] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-

4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[43] D. Moro, M. Peuster, H. Karl, and A. Capone. Fop4: Function offloading prototyping in heterogeneous and programmable network scenarios. In *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–6. IEEE, 2019.

[44] D. Moro, G. Verticale, and A. Capone. A framework for network function decomposition and deployment. In *2020 16th International Conference on the Design of Reliable Communication Networks DRCN 2020*, pages 1–6. IEEE, 2020.

[45] D. Moro, G. Verticale, and A. Capone. Network function decomposition and offloading on heterogeneous networks with programmable data planes. *IEEE Open Journal of the Communications Society*, 2:1874–1885, 2021.

[46] F. Y. Narvaez, G. Reina, M. Torres-Torriti, G. Kantor, and F. A. Cheein. A survey of ranging and imaging techniques for precision agriculture phenotyping. *IEEE/ASME Transactions on Mechatronics*, 22(6):2428–2439, 2017.

[47] T. Neagoe, V. Cristea, and L. Banica. Ntp versus ptp in computer networks clock synchronization. In *2006 IEEE International Symposium on Industrial Electronics*, volume 1, pages 317–362. IEEE, 2006.

[48] T. Osiński, H. Tarasiuk, L. Rajewski, and E. Kowalczyk. Dppx: A p4-based data plane programmability and exposure framework to enhance nfv services. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 296–300. IEEE, 2019.

[49] S. Sahhaf, W. Tavernier, M. Rost, S. Schmid, D. Colle, M. Pickavet, and P. Demeester. Network service chaining with optimized network function embedding supporting service decompositions. *Computer Networks*, 93:492–505, 2015.

[50] D. Sanvito, G. Siracusano, and R. Bifulco. Can the network be the ai accelerator? In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, pages 20–25, 2018.

[51] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 150–156, 2017.

[52] L. Tan, W. Su, W. Zhang, J. Lv, Z. Zhang, J. Miao, X. Liu, and N. Li. In-band network telemetry: a survey. *Computer Networks*, 186:107763, 2021.

[53] P. K. Thiruvasagam, A. Chakraborty, A. Mathew, and C. S. R. Murthy. Reliable placement of service function chains and virtual monitoring functions with minimal cost in softwarized 5g networks. *IEEE Transactions on Network and Service Management*, 2021.

[54] Y. Tokusashi, H. Matsutani, and N. Zilberman. Lake: the power of in-network computing. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2018.

[55] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman. The case for in-network computing on demand. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[56] B. Varga, J. Farkas, R. Cummings, Y. Jiang, and D. Fedyk. Flow and Service Information Model for Deterministic Networking (DetNet). RFC 9016, Mar. 2021. URL `https://rfc-editor.org/rfc/rfc9016.html`.

[57] S. T. Watt, S. Achanta, H. Abubakari, E. Sagen, Z. Korkmaz, and H. Ahmed. Understanding and applying precision time protocol. In *2015 Saudi Arabia Smart Grid (SASG)*, pages 1–7. IEEE, 2015.

[58] B. Yi, X. Wang, K. Li, M. Huang, et al. A comprehensive survey of network function virtualization. *Computer Networks*, 133:212–262, 2018.

[59] N. Zhang, M. Wang, and N. Wang. Precision agriculture—a worldwide overview. *Computers and electronics in agriculture*, 36(2-3):113–132, 2002.

[60] 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*, pages 98–111, 2018.

List of Figures

3.1 Example of a network architecture on which the project could be instantiated . . . 15
3.2 Logical view of a Service Function Chain and how it can be mapped to a physical topology . . . 18

4.1 Logical placement of the framework's components in the network and their interactions . . . 22
4.2 Visual representation of the section of path that is considered in the computation of different requirements . . . 26
4.3 Stages of the template pipeline . . . 30
4.4 Example of the framework's use of segment routing . . . 40
4.5 Structure of INT data embedded in transiting packets by the base pipeline. Switch S1 acts as the source, initializing shim and metadata headers. . . . 41
4.6 Comparison of the content of packets and the forwarding behavior with regular INT and with CHIMA's extension . . . 47
4.7 Series of events that lead to the detection of an unmet requirement . 49

5.1 Representation of the timestamps collected during one test run, and their relationship in time. The time spans considered for the computation of detection and redeployment performance are highlighted in red. . . . 52
5.2 *minimal* test case with 4 functions, 2 of which are general purpose and 2 P4 ones. All the functions are relocated after the introduction of a delay . . . 56
5.3 Delay in the detection of an exceeded requirement with different intervals for the polling of new measurements from the INT collector. 95% CI. . . . 57

5.4 Delay in the detection of an exceeded requirement with different coefficients for the computation of the EWMA on link measurements. 95% CI. 58
5.5 Representations of the additional test cases used to study redeployments times 60
5.6 Time for the complete redeployment of a service, along with the contributes of P4 and Container redeployment, in different test cases. 95% CI. 62

List of Tables

5.1 Characteristics of the presented test cases. The number of functions that will be moved in each case is stated in parenthesis. 59
5.2 Breakout of redeployment times for different topologies. 95% CI. . . . 61

Acknowledgements

Desidero ringraziare il Prof. Antonio Capone, il Prof. Giacomo Verticale e l'Ing. Daniele Moro per l'inestimabile sostegno fornitomi durante lo sviluppo di questa tesi.

Grazie ai miei genitori, che hanno sempre creduto in me e mi hanno dato la possibilità di seguire questo percorso.

Grazie alla mia famiglia, che ha gioito con me di ogni piccolo successo. Un pensiero particolare va ai miei nonni Adriana e Giulio, che avrebbero voluto assistere a questo traguardo.

Grazie a Silvia, che mi è stata accanto dal primo giorno di questo viaggio, soprattutto nei momenti più difficili.

Infine, ringrazio i miei compagni di corso, con i quali ho condiviso le sfide degli ultimi cinque anni.