



## **D234: Selection, benchmark and design of the individual components forming the core of the global MOSAICO low-latency architecture**

Mid-term project achievements and directions

Bertrand Mathieu<sup>1</sup>, Stéphane Tuffin<sup>1</sup>, Olivier Dugeon<sup>1</sup>, Joël Ky<sup>1</sup>, Guillaume Doyen<sup>5</sup>, Marius Letourneau<sup>2</sup>, Hichem Magnouche<sup>2</sup>, Philippe Graff<sup>3</sup>, Xavier Marchal<sup>3</sup>, Juuso Haavisto<sup>3</sup>, Thibault Cholez<sup>3</sup>, Edgardo Montes de Oca<sup>4</sup>, Huu Nghia Nguyen<sup>4</sup>, and Manh-Dung Nguyen<sup>4</sup>

<sup>1</sup>Orange

<sup>2</sup>UTT

<sup>5</sup>IMT Atlantique

<sup>3</sup>Loria

<sup>4</sup>Montimage

### **Abstract**

The present deliverable stands for the mid-time milestone of the MOSAICO project, started on December 1, 2019. As such it gathers the set of achievements produced over the last two years period. As initially planned, the project has followed a practical approach with the objective to first evaluate some operational situations which can further guide the design and implementation of subsequent components. As such, the project selected cloud gaming as the main use-case for guiding all subsequent works and L4S as the core framework for packet forwarding. In band telemetry, possibly implemented with P4-based network equipment, and closed-loop frameworks for monitoring and troubleshooting solutions are envisioned but still under investigation. In order to reflect at best the works achieved during this first period as well as those currently ongoing, the deliverable presents all the MOSAICO contributions according to: (1) their maturity in the project advancement; and, (2) their fulfilment of an operational aspect we need to address, whatever its belonging to a particular task.

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Mid-Term Project Achievements</b>	<b>4</b>
2.1	Evaluation of the Project Use-Case: Cloud Gaming . . . . .	4
2.1.1	Cloud Gaming testbed . . . . .	4
2.1.2	Initial network constraints . . . . .	5
2.1.3	Sudden network constraints . . . . .	8
2.2	Evaluation and Implementation of the Selected Data-plane Technologies: L4S and P4 . . . . .	9
2.2.1	A P4-based L4S solution . . . . .	9
2.2.2	Local testbed . . . . .	12
2.2.3	P4-based L4S vs Linux-based L4S . . . . .	13
2.3	Addressing the Limitations of the L4S Architecture . . . . .	16
2.3.1	L4S under time varying network conditions . . . . .	16
2.3.2	Threats targeting low-latency traffic with L4S . . . . .	19
2.4	Micro-services for Monitoring . . . . .	24
2.4.1	Analysis of OpenNetVM solution . . . . .	24
2.4.2	Monitoring framework as micro-services . . . . .	25
2.4.3	Evaluation . . . . .	27
2.5	Orchestration of Micro-Services . . . . .	29
2.5.1	Definition and evaluation of a model for orchestrating micro services (Confidential for now) . . . . .	29
2.5.2	Orchestration of micro-services running on GPU . . . . .	38
2.6	Closed-loop solutions . . . . .	43
2.6.1	Closed-loop for QoS . . . . .	44
2.6.2	Evaluation of closed-loop scenarios . . . . .	44
2.6.3	Closed-loop recommendations . . . . .	49
<b>3</b>	<b>Ongoing and Future work</b>	<b>50</b>
3.1	Improvements for Cellular Networks . . . . .	50
3.2	Closed-Loop in MOSAICO Architecture . . . . .	51
3.3	Modular Cloud Gaming Platform . . . . .	52
3.4	Global Testbed . . . . .	54
3.4.1	The 5G-in-a-Box platform . . . . .	54
3.4.2	Evolutions of the testbed . . . . .	55
<b>4</b>	<b>Conclusion and Perspectives</b>	<b>57</b>

# Chapter 1

## Introduction

The present deliverable stands for the mid-time milestone of the MOSAICO project, started on December 1, 2019. As such it gathers the set of achievements produced over the last two years.

The project has been designed around four scientific tasks that we subsequently summarize. Task 1 aims at designing an overall low-latency architecture, respectful of current recommendations, standardization and efforts, for the delivery, security, monitoring and orchestration of the project use-case. Task 2 focuses on the evaluation and possible implementation of micro-services forming the actual components taking part of the architecture. Task 3 concerns orchestration issues of such a set of micro-services to eventually offer the expected performance and security. Finally, Task 4 involves the final integration and evaluation with the deployment of a global testbed integrating the software components implemented and individually evaluated in the other tasks.

As initially planned, the project has followed a practical approach with the objective to first evaluate some operational situations which can further guide the design and implementation of subsequent components. As such, among all the reviewed use-cases in deliverable D1.1, the project selected cloud gaming as the main one for guiding all subsequent works. Following the same idea, it also selected L4S as the core framework for packet forwarding. In band telemetry, possibly implemented with P4-based network equipment, and closed-loop frameworks for monitoring and troubleshooting solutions are envisioned but are still under investigation.

In order to reflect at best the works achieved during this first period, as well as those currently ongoing, we have deliberately chosen in this deliverable to present all the contributions according to: (1) their maturity in the project advancement; and, (2) their fulfilment of an operational aspect we need to address, whatever its belonging to a particular task. Consequently, the deliverable is mainly split into two main chapters. Chapter 2 presents all the work considered as accomplished. This work essentially results from measurement campaigns and for some parts has been either already published in scientific venues or is under submission. Then, Chapter 3 details the set of fields the project is still exploring to eventually conduct to a global architecture encompassing data-plane as well as control and management plane components to offer a fully operational low-latency system dedicated to the targeted Cloud Gaming use-case.

Before entering the detailed presentation of the project achievements, the next section provides a summary of the ongoing work related to the envisioned architecture. This will help the reader to further relate the subsequent contributions to this big picture. However, one should notice that the finalization and presentation of such an architecture is not the target of the present deliverable since this topic will be at the core of the upcoming deliverable D1.2.

## Envisioned architecture

In Task 1 of the project, we study the use-cases, and the architecture as well. One of our basic assumption is that our solution should work for 5G wireless networks and wired networks (e.g., FTTH). This means that our architecture should follow 5G network architecture and principles for smooth integration. However, we should also take into consideration (and eventually propose) possible evolutions of 5G network. For instance, the current Network Functions (such as UPF) are located in regional nodes (i.e., POP : Point-of-Presence), but a possible evolution is to have UPF closer to the end-users in local nodes for specific processing tasks.

Since we are dealing with low-latency services, it means few end-to-end transmission milliseconds

and thus limits the network scope. Our targeting service will then not be a service hosted in a very far location, but rather on a national or European server. We also take as a fact that the core network is very fast, efficient, over-provisioned and without any processing task. Then we assume having processing modules in the access or aggregation part of the network.

With these considerations in mind, we currently imagine a network architecture where the MOSAICO nodes could be deployed in regional or in local part of the network. Nodes in local network will be more numerous and, for cost efficiency, we should reduce their capacity. We imagine to have a simple node in local network, while a more complex node can be deployed in regional networks.

For ensuring low-latency delivery of services, we decided to rely on some data-plane solutions. Amongst the possible candidates, as seen in deliverable D1.1, one very promising approach is the IETF Low Latency, Low Loss, Scalable Throughput (L4S) architecture [9][22]. This is also linked with scalable congestion control algorithms such as TCP-Prague [23]. Another concept which opens many doors for data-plane network programmability and openness is the Programming Protocol-independent Packet Processor language (P4) [21][11].

For control-plane functions, which can be required for orchestrating the services, for security issues, or for other functions, which do not need to be implemented at the data-plane level, we advocate the adoption of an application level programmable environment. This environment can leverage concepts derived from the Network Function Virtualisation (NFV).

We then design a two-level MOSAICO node, where packet processing can be done at one or the other level, depending on the function itself, but also on the complexity of the function. Indeed, P4 modules can apply simple processing to packets transiting via the P4 switch, but it can not handle complex tasks, which then will be done by the application level modules. For instance, we can have L4S running in the P4 node, but complex attack detection modules will be implemented in the application compute node.

Fig. 1.1 depicts our current envisioned architecture.

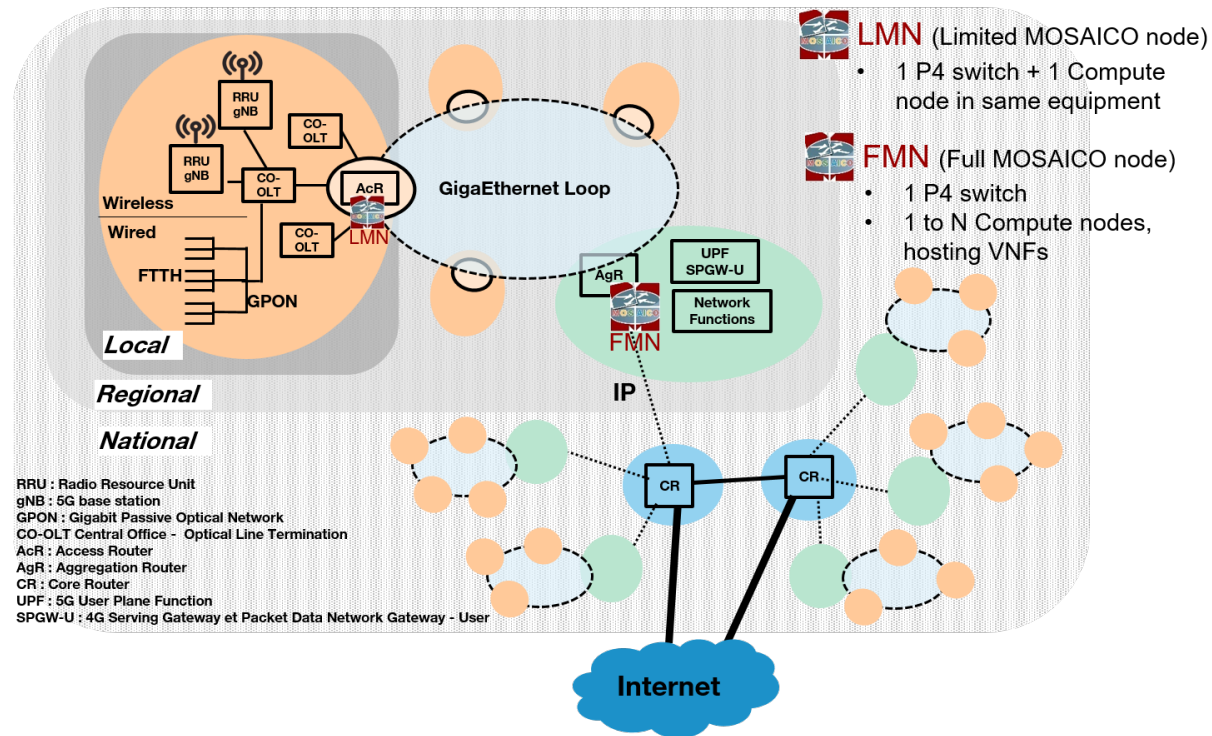


Figure 1.1: Envisioned MOSAICO architecture

## Chapter 2

# Mid-Term Project Achievements

This chapter provides a detailed review of the set of works which has been implemented and validated during the first period of the project. As motivated in the deliverable introduction, it is organized according to the functional aspects the project has to cover to step-by-step reach the overall architecture it aims to design and implement. As such, we first present in Section 2.1 the results of an experimental evaluation of different cloud gaming platforms, standing for the selected project use-case due to both its relevant latency requirements and the availability of solutions making possible measurement and future enhancements. Then, we address the technological components which aims at carrying the low-latency traffic as the core of the future overall architecture. Here, we selected L4S for its maturity and acknowledgement by the research community and normalization bodies, and especially IETF. We propose in Section 2.2 to leverage P4 as a substrate for L4S to ease its adoption by stakeholders such as device manufacturers and telco and we present our original implementation. We subsequently address in section 2.3 situations where L4S exhibits some performance limitations due to technological intrinsic behaviors such as that of time varying conditions (e.g. 4G) or intentional behaviors whose purpose of to deliberately degrade or collapse the expected low-latency of services, thus acting as novel form of denial of services. Since micro-services have been envisioned as a core technology to implement low-latency service function chains, beyond the sole forwarding aspect, in Section 2.4, we present the first results we collected by splitting the monitoring framework of partner Montimage into a basic micro-service architecture hosted in the OpenNetVM reference environment. Then, Section 2.5 presents a comprehensive model for the routing and placement of micro-services with the aim of reaching low latency constraints while preserving the related resource usage as well as the way GPU can be leverage to offload micro-services processing and the way this can be orchestrated. Based on these ground contributions, Section 2.6 addresses closed-loop issues and design guidelines. This encompasses an operational study of existing services (e.g. telco VPN) and leveraged protocols as well as the consideration of current state-of-the-art technologies such as in-band telemetry.

## 2.1 Evaluation of the Project Use-Case: Cloud Gaming

In this subsection, we evaluate the behavior of cloud gaming (CG) platforms under various network conditions. We rely on the testbed described in section 2.1.1 and perform an analysis of the large produced dataset of CG network traces. As first part, section 2.1.2 focuses on initial network constraints, set up before starting a game session. Conversely, section 2.1.3 focuses on constraints that appear during a game session. We can thus observe how CG platforms react to a sudden network constraint and how they recover.

### 2.1.1 Cloud Gaming testbed

The concept of cloud gaming is to run a game on a remote server. User commands pass through the network to reach the server which applies the “*game logic*” and streams back the resulting audio and video. With the recent technological evolution in these fields and the increased deployment of multi-tier clouds, cloud gaming (CG) is gaining renewed interest and a lot of attention. CG is expected to become a core service in the coming years. Its global market is forecasted to rise over three billion gamers by 2023. There is a strong expectation about CG to deliver the quality allowing end-users to fully enjoy their

gaming sessions. Not waiting for network operators to generalize high-throughput low-latency networks, CG providers implement their own application-level mechanisms to cope with varying network conditions. These mechanisms together with congestion control techniques aspire to use available network capacities to jointly optimize the perceived network latency and the delivered video quality, which are tightly linked to players' QoE.

The purpose of this testbed is to evaluate these mechanisms by observing the properties of Cloud Gaming traffic under normal, but especially under deteriorated network conditions. To do this, we have developed the *testbed* that we can see on figure 2.1. We cause network disturbances in the server-to-client

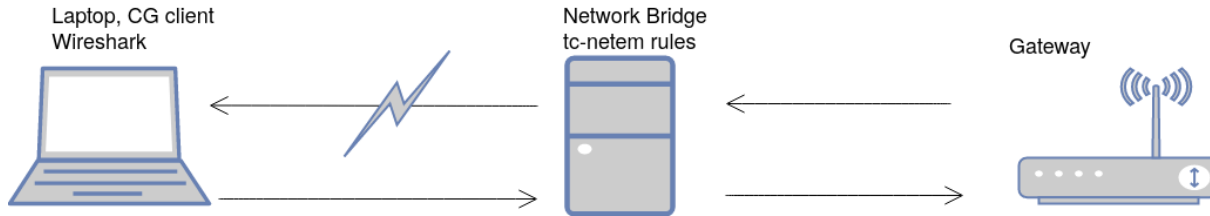


Figure 2.1: Locale Cloud Gaming Architecture

direction using an intermediate network device placed between the gateway and the laptop running the Cloud Gaming client. This intermediate device acts as a network bridge. From `tc-netem` rules, we can add latency, jitter and packet loss. We also limit the available bandwidth, using `Wondershaper`. Table 2.1 lists the different parameters used to impact the network conditions. Each of these parameters are tiered based on their intensity (*Moderate*, *High* and *Extreme*). Their naming represents how much they impact the network quality.

Table 2.1: Parameters used to degrade network quality

	Moderate	High	Extreme	Reference
Loss	5%	10%	20%	0%
Bandwidth	20 Mbps	10 Mbps	5 Mbps	10Gbps
Latency	20 ms	50 ms	100 ms	8 ms
Jitter	5 ms	10 ms	20 ms	0 ms

Our testbed includes suscriptions to the four main cloud gaming platforms available to date in France, namely Nvidia Geforce Now<sup>1</sup>, Google Stadia<sup>2</sup>, Sony PlayStation Now<sup>3</sup> and Microsoft Xbox Cloud Gaming platform<sup>4</sup> (formerly known as xCloud). We perform a network capture directly from the Cloud Gaming client.

To collect the traces of traffic for further statistical analysis, we use *Wireshark*. A comprehensive dataset of cloud gaming traces produced by this testbed were performed between April and July 2021. For each of the four aforementioned CG platforms, the considered types of network degradation cover packet loss, throughput decrease, latency increase and jitter variation. Two tests campaigns have been performed: 1) sessions started under already degraded network conditions, 2) network degradation introduced and removed during gaming sessions to observe how CG platforms mitigate and recover. The full dataset (160Go) is publicly available in the following repository: <https://cloud-gaming-traces.lhs.loria.fr/index.html>.

### 2.1.2 Initial network constraints

In this section, we evaluated CG platforms with initial network constraints. The idea is to configure network conditions before starting a game session. We therefore observe the behavior of platforms that have had time to probe and to adapt to the network conditions.

On Figure 2.2, we represent the bit rate produced by each platform (from left to right : GFN, STD, PSN, XC) according to bandwidth limitations. Here, “*ratx*” stands for “available bandwidth limited to

<sup>1</sup><https://play.geforcenow.com/>

<sup>2</sup><https://stadia.google.com/>

<sup>3</sup><https://www.playstation.com/en-us/ps-now/>

<sup>4</sup><https://www.xbox.com/en-US/play>



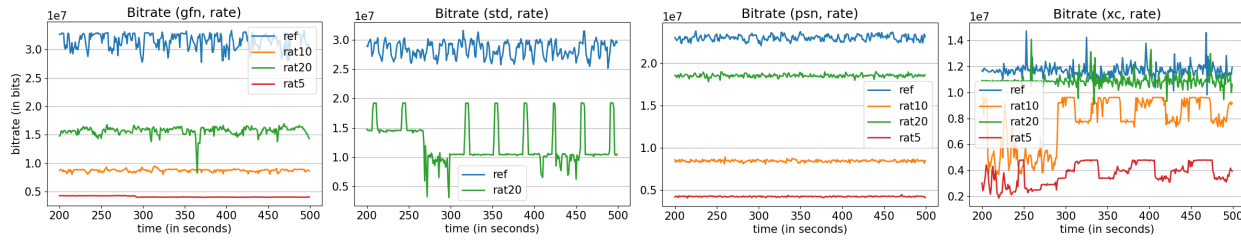


Figure 2.2: GFN, STD, PSN and XC bitrate over time for different rate limitations (service⇒client)

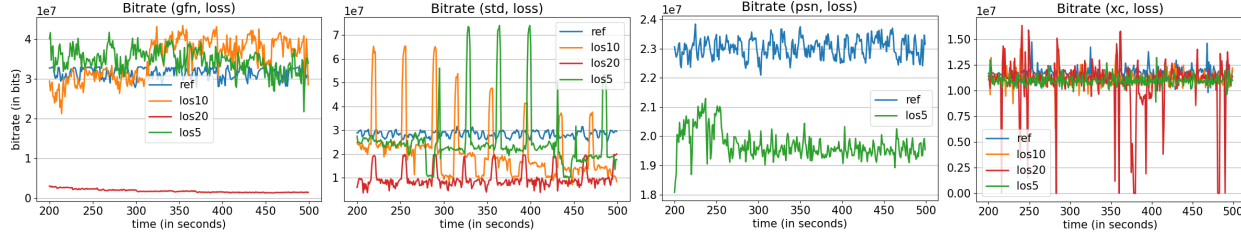


Figure 2.3: GFN, STD, PSN and XC bitrate over time for different loss rates (service⇒client)

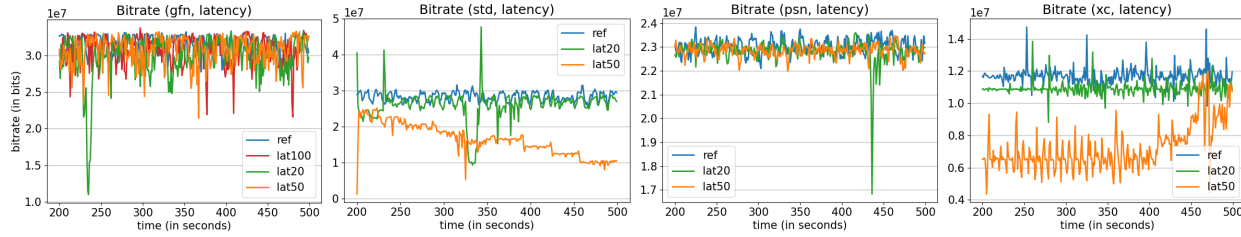


Figure 2.4: GFN, STD, PSN and XC bitrate over time for different added latency values (service⇒client)

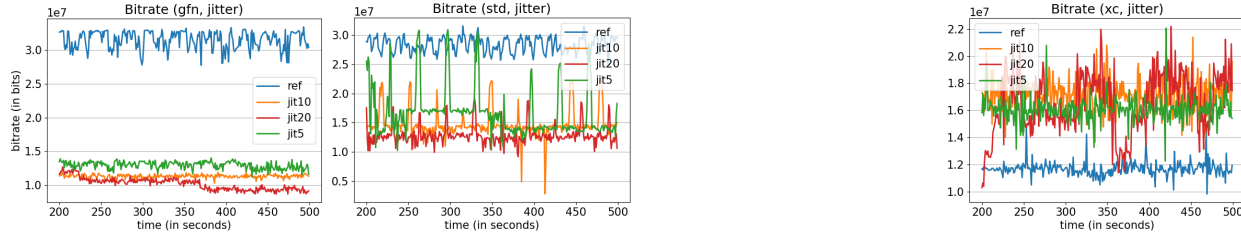


Figure 2.5: GFN, STD, PSN and XC bitrate over time for different jitter values (service⇒client)

‘x’ Mbps”. We see that GFN reduces its bit rate by more than 50% (from 33Mbps to 16Mbps) when we limit the available bandwidth to 20Mbps. It successfully adapts to lower bandwidth capacities. We notice that it uses only around 80% of the maximum link capacity. It seems to indicate that some margin is kept in case of bursts to avoid unneeded jitter and ensure a stable throughput. For STD, the first remarkable result is that it is not possible to complete a game session with a bit rate below 10Mbps. This makes it the most demanding platform regarding the bandwidth requirements. When the constraint is set to 20Mbps, STD does not choose to maximize the utilization of the link capacity. It stabilizes a bit above 10Mbps (around 50% of the link capacity) and exhibits periodical burst spikes to probe the bandwidth. Note that around the 260th second, the platform changes the resolution from 1080p to 720p. This is accompanied by a decrease in bit rate (from around 15 to 10Mbps). PSN shows a very stable bit rate adapted to each constraint, and uses around 90% of the available bandwidth (small margin). XC does not need to adapt to the 20Mbps constraint. Its throughput in normal conditions is indeed in the 12Mbps range. For the 10 and 5Mbps limitation, XC exhibits a specific behavior. It repeatedly increases its bitrate until it reaches the limit, before lowering it abruptly after a few seconds. The bandwidth usage oscillates between 95% and 80%. This behavior suggests that XC does not handle bandwidth constraints well. It has to periodically readjust its throughput, affecting user experience.

Figure 2.3 shows the bit rate of the four platforms under several loss rates. In our convention, “*los<sub>x</sub>*” stands for “‘x’ % of packet losses in addition to the reference ‘*ref*’”. Regarding GFN, the curves are roughly similar for 5 and 10% of packet loss. We see a slight bit rate increase (through the packet rate). We conjecture it is due to the addition of Forward Error Correction (FEC) which would be inhibited in

normal conditions. The behavior is totally different for 20% of packet loss as the bit rate is reduced by more than 90%. The QoE is highly impacted, as the video quality is at its lowest level (540p30fps). STD manages to adapt to loss by reducing the bit rate in proportion to the loss rate. The curves show periodic peaks occurring about every 40 seconds. That is the way STD probes for available bandwidth in case of network issue. With no bandwidth restriction on this experiment, the spikes are taller than in Figure 2.2, with observed burst reaching two or three times the average bitrate with more than 70Mbps. We can see peaks are more pronounced for smaller losses. PSN is far less resilient to losses. For any loss rate exceeding 5%, it refuses to launch a game instance. With a loss rate of 5% the average bit rate slightly drops to 20Mbps versus 23Mbps with no added loss. These bit rates similarities hide a reaction from PSN. Indeed, packet sizes and IAT significantly increase (+39.6% IAT on average) which is certainly due to the addition of error correction within packets. There is not much to say about XC except that it does not seem to (or need to) adapt to losses. The QoE is not impacted despite high loss rates. So we can conjecture XC constantly uses high redundancy to withstand high loss rates.

Figure 2.4 shows the bitrate of the four platforms when we add network delays. “latx” stands for an addition of ‘x’ ms of latency in the server to client direction. GFN does not react much to latency. We observe a slight increase in bit rate variance but the mean is rather constant. The huge spike on the curve associated with 20ms of added latency is due to an artefact of gameplay. GFN is also the only platform to operate despite of 100ms of latency. STD’s behavior varies depending on the amount of added latency. 20 ms of added latency does not trigger any significant reaction. At 50ms, STD gradually reduces its bitrate from 30Mbps to 10Mbps, the same floor as in Figure 2.2. Strangely, STD does not show the spiky pattern like in the other experiments. When adding 100ms latency, STD agrees to launch a game instance, but threatens to shut it down if network conditions do not improve in the next 30 seconds. PSN does not take any action against latency but refuses to launch an instance with 100ms of added latency. XC slightly reduces the bitrate for 20ms of added latency. At 50ms the service shows an unstable bitrate with a heartbeat pattern around 7Mbps, mainly due to a high variance in packets IAT. The bitrate still slowly increases towards the end of the capture until it reaches its regular value. Like STD and PSN, 100ms is beyond its capacity to operate the service.

Finally, Figure 2.5 shows the bitrate of the four platforms with added jitter. The reference curve represents the evolution of the throughput without added jitter. A jitter of ‘x’ ms is represented by the “jitx” curve. GFN’s response to jitter is very important. For the lowest added jitter value, the bit rate is reduced by 20Mbps to reach 13Mbps. The bit rate is further reduced to 9Mbps for 20ms of added jitter thanks to a change of resolution to the most degraded mode at 375 seconds. STD also strongly reacts to jitter with a bit rate reduced by at least 10Mbps for 5ms of added jitter. The bit rate further drops by 5Mbps when we add 20ms jitter. It is then within 13Mbps. There is no data for PSN that failed to finish any capture even with the lowest jitter value. XC totally differs from the others in that it increases its throughput as soon jitter is added. At the first jitter value, the platform increases its throughput by more than 30% by sending more packets per second. We assume that it is due to the retransmission of out of order packets caused by jitter. Same as for latency, a high jitter tends to increase the variance of the stream’s bit rate.

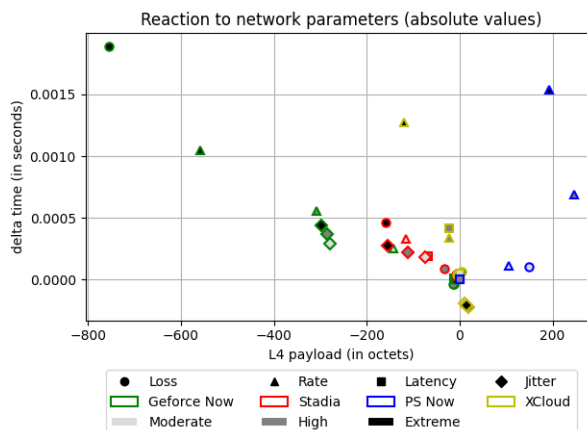


Figure 2.6: Relative 2D space for mean UDP payload and IAT (service⇒client)

Figure 2.6 displays our results in two dimensions with the average UDP payload size on the X-axis and the average IAT on the Y-axis. Each mark is associated with a certain platform, for a certain constraint.



The goal is to highlight notable differences in the adaptation strategies of the different platforms.

Three of the four services: GFN, STD and XC tend to drift to the upper left corner but they have their own way to do it with different trend functions. GFN shows the wider span on the two dimensions which reflects a broader adaptation range. On the other hand, PSN is the sole to be in the upper right corner because of an increase of payload sizes in certain conditions. XC has a few points in the lower right corner because of decreased IAT under jitter.

### 2.1.3 Sudden network constraints

This section focuses on constraints that appear during a game session. Here, each network capture lasts 6 minutes. We deteriorate the network conditions between minutes 2 and 4. We can thus observe how CG platforms react to a sudden network constraint. Once the constraint is removed, we observe their recovery phase.

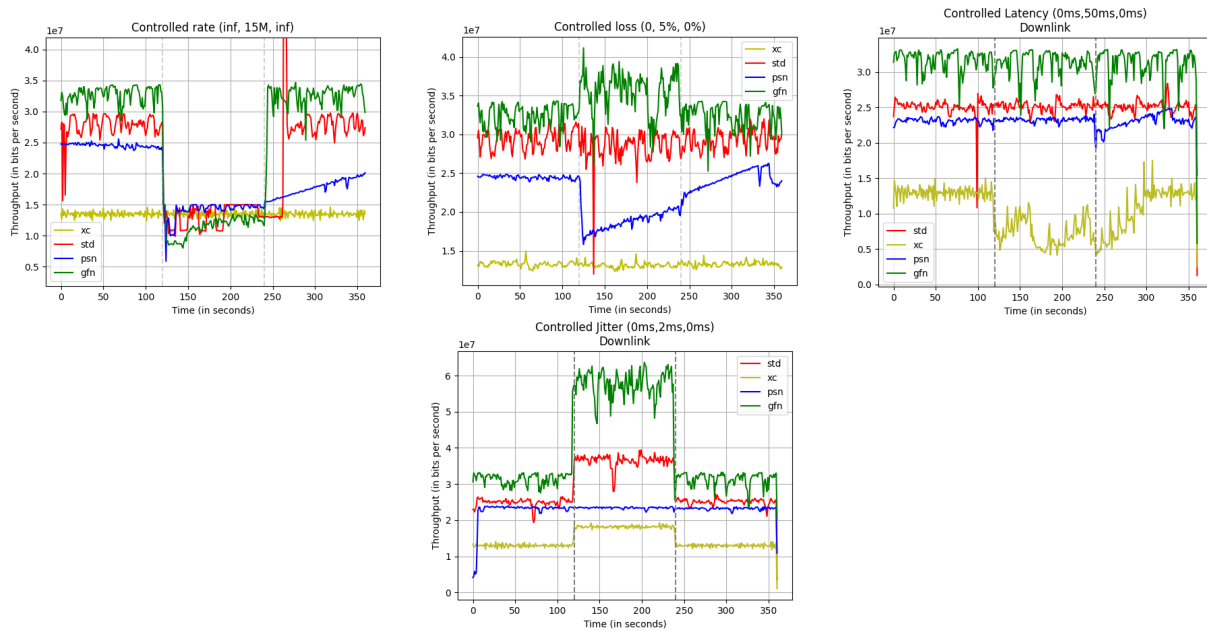


Figure 2.7: Server-to-client throughput of cloud gaming platforms under sudden network constraints

The first subfigure 2.7 shows the bit rate limitation to 15Mbps. XC nominal bit rate being below the constraint it does not need to adapt. We couldn't choose a lower limitation since we previously noticed that Stadia refuses to start with a bit rate of 10Mbps. STD does not decrease much right after the constraint is set but the service has a hard time to stabilize with around 80 second of crenelated bit rate (with a ramping pattern on some of them). Afterward, the bit rate becomes more stable with a good usage of the available bandwidth, but it decreases a bit before the constraint is removed which hints the service was still trying to stabilize. Then, STD needs some time to recover (around 10s) with a huge spike followed by the exact same bit rate we recorded before the constraint was applied. This spike is a pattern regularly seen on STD and probably triggered by the congestion control algorithm to quickly evaluate the available bandwidth. For PSN, we can see a huge decrease of the bit rate the first second the constraint is applied (resulting in a stalled video) certainly due to packet drops in the queue of our computer and inefficiencies in PSN congestion control. Then it recovers quite well with a single notch and a pretty stable bit rate. Afterward, PSN exhibits a slope-like recovery reaching the same level as before the constraint was applied but after the 120s period displayed on the graph. At last, GFN has a reaction strength between the two others, then it keeps this bit rate some time before gradually improving but without fully using the available bandwidth like STD and PSN. After the constraint is removed, GFN exhibits a very fast recovery (between 1 and 2 seconds).

The second subfigure 2.7 shows the bitrate over time when a 5% loss rate is applied. We can see that each service has its own way to handle loss. GFN increases its bitrate probably due to additional FEC or retransmissions, PSN shows a sudden drop followed by a very slow ramping increase that grows in strength when the network conditions are back to normal. XC and STD do not adapt their traffic at all,

probably meaning that their traffic already include enough FEC to withstand a 5% loss rate.

The third subfigure 2.7 shows the reaction to 50ms of added latency. We can see that XC instantly decreases its bit rate up to almost three times at the lowest point, from 14Mbps to 5Mbps, and it never reaches a steady state as long as the perturbation is on going. Even worse, it takes one more minute after the constraint removal to restore the initial throughput. From a QoE perspective, the gaming experience was the most unpleasant on this platform under the above conditions, the game being unresponsive to the controls. Strangely, PSN seems to reduce its sending rate once the disturbance has stopped and it takes about 60 seconds to get back to its normal rate. GFN and STD, on the opposite, do not adapt their bitrate to the added latency.

The last subfigure 2.7 shows the reaction to 2ms of added jitter. Three of the four platforms, STD, GFN and XC significantly increase their bitrate under jitter. GFN reacts the most, doubling the bitrate under jitter up to 60Mbps, STD goes from 25 to 37 Mbps (+66%). The most probable explanation is that jitter creates many out-of-order packets that must be retransmitted if the reception buffer is not large enough to allow packet reordering. Those three platforms also quickly come back to their initial rate as soon as jitter is suppressed. PSN, on the other hand, doesn't seem to react at all to the jitter. Regarding the client-to-server traffic, similar behaviors can be observed for STD and PSN: STD increases its traffic in the same proportions in this direction while PSN does not react. GFN only exhibits a minor increase of its bit rate contrary to the major one in the opposite direction. XC is the most affected platform with client-to-server traffic multiplied by three under jitter, from 420Kbps to 1200Kbps.

In conclusion, all platforms try to adapt their traffic at some point when one or another constraint is applied, reducing their bandwidth usage by adjusting a combination of video resolution, frames per second and compression level, or increasing it to re-transmit packets or include additional FEC. We can notice that all platforms do not react in the same way to the same constraints and all constraints do not have the same impact on the delivered bit rate. Please note that the lack of reaction does not necessarily mean that the quality of experience is not affected. For instance, PSN not reacting to latency or jitter leads to a degraded service. In several experiments the bit rate keeps being adjusted all the time when a constraint is applied, and sometimes even after its release, further altering the QoE.

This study highlights the fact that application-level mechanisms are not enough. They often take too much time to detect, and react to, network issues, regularly leaving the service hardly playable with a lot of stuttering experienced at the user side. Leveraging network-level adaptive mechanisms, like new latency oriented network technologies, could produce an immediate response to preserve latency requirements and let the time for the application to adapt more smoothly and only in case of lasting disturbance. Our future actions on this topic will consist in detecting cloud gaming traffic and apply to it some network processing techniques optimized for low latency services.

## 2.2 Evaluation and Implementation of the Selected Data-plane Technologies: L4S and P4

Given the Cloud Gaming use-case selection we presented above, as a second core activity, the MOSAICO project has drawn a particular attention to the technology which can bring an efficient forwarding of data to actually reach the expected low-latency score. An overview of possible technologies indicate that the L4S proposal could fit with our expectations, owns an operational implementation on Linux and is under standardization at the IETF, thus making it an appropriate candidate as the data-plane technology. However, with the aim to (1) facilitating the adoption of such a novel data-plane routing architecture and (2) enabling an agile management of network components, in this section, we present a work related to the development of L4S in a P4 switch and prove it behaves as expected, when comparing it to the Linux-based reference implementation.

### 2.2.1 A P4-based L4S solution

Latency-aware congestion controls are endpoint based solutions, but they are unable to control the latency and can be starved in presence of latency unaware traffic. To improve the coexistence of different traffic types, the IETF proposes the promising L4S (Low Latency Low Loss Scalable throughput) architecture [9], together with the DualQ Coupled AQM [22], which enables the fair sharing of bandwidth of the low-latency (LL) traffic and other best effort (BE) traffics, via a coupling between the two queuing algorithms.

For ensuring low latency, L4S relies on Accurate ECN [7], an evolution of ECN, which allows the sender to react more finely according to the congestion level, but which requires an adapted transport stack, such as DCTCP, designed for data centers, or TCP-Prague [8], adapted from DCTCP to be deployed over the Internet. Moreover we anticipate that other transport protocols such as QUIC (Quick UDP Internet Connections) and RTP (Real-Time Transport Protocol) and other congestion controls such as RMCAT (RTP Media Congestion Avoidance) will also support L4S in the future.

The L4S system requires upgrades on endpoints but also in network equipment, which is known to be rather static and where evolutions can take a very long time to be deployed. To speed it, the programmable network paradigm was adopted with technologies such as SDN (Software Defined Networking) and NFV (Network Function Virtualisation) and more recently, the P4 (Programming Protocol-Independent Packet Processor) concept [11][21]. This solution makes the network more agile by enabling the quick deployment of data plane networking software that is executed on network equipment hardware, and shortens the update lifecycles.

In MOSAICO, we advocate the use of such concepts, which will be the corner stone of the future networks. We then designed and implemented a L4S solution for a P4-based programmable network equipment, which would allow fast deployment of L4S evolutions, proving the feasibility to make it.

An implementation of L4S has been done in [12] and is available as open source in [16]. This implementation is written in C language in a monolithic approach. However, looking at the L4S design (Fig. 2.8), we might imagine several micro-services to offer the overall L4S service.

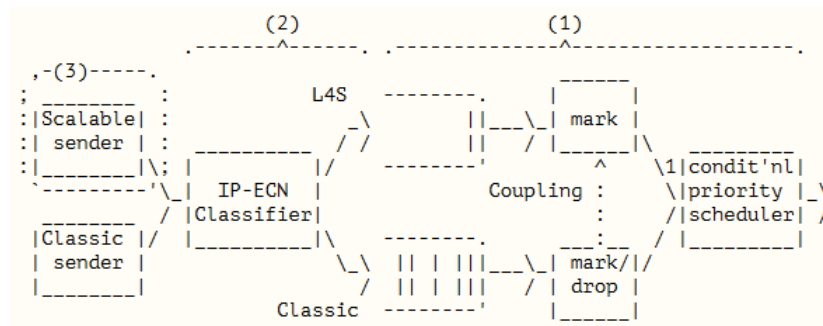


Figure 2.8: L4S architecture defined at IETF [9]

Investigating more finely how to split the L4S architecture, we decided to split it into 4 micro-services : the classifier, the queueing system, the AQM computations and the packet decision. Indeed these 4 functions are independent and each one can be implemented in several ways.

In the MOSAICO project, since we advocate the use of a programmable data plane, we implemented a L4S solution using the P4 approach [11][21]. The use of the P4 environment is motivated by the possibility to have an open network equipment allowing operators to quickly deploy programs in response to the fast evolution of applications, transport protocols or algorithms. Indeed, our system works with IP/TCP and ECN flags, but we might imagine evolutions or other transport protocols with different headers (e.g., QUIC or RTP can also support ECN), easily taken into account with P4 (i.e. by changing the parser of the P4 program). Similarly, we might think that a better solution than L4S can emerge and implementing it in P4 will allow its fast deployment in network equipment. The possible modifications, changes or upgrades of the classifier or AQM, and the genericity of the solution are the reasons why we opt for the P4 framework. The Fig. 2.9 presents how the 4 L4S modules are mapped onto a P4-based node.

The following describes how the P4 program processes packets and explains the role of each of the L4S modules, shown in Fig. 2.9, on top of the P4 pipelines.

- The first component is the P4 parser which extracts the IP and TCP headers of the packet.
- Then the packet classifier, implemented in the ingress part of the P4 program assigns the packet either to the LL or BE queue. In the current version, this module is simple and just parses the ECN bit of the Diffserv field of the IP header to detect if the packet belongs to an “Accurate ECN” flow or not. When ECN-capable applications agreed with network operators, such a classification can

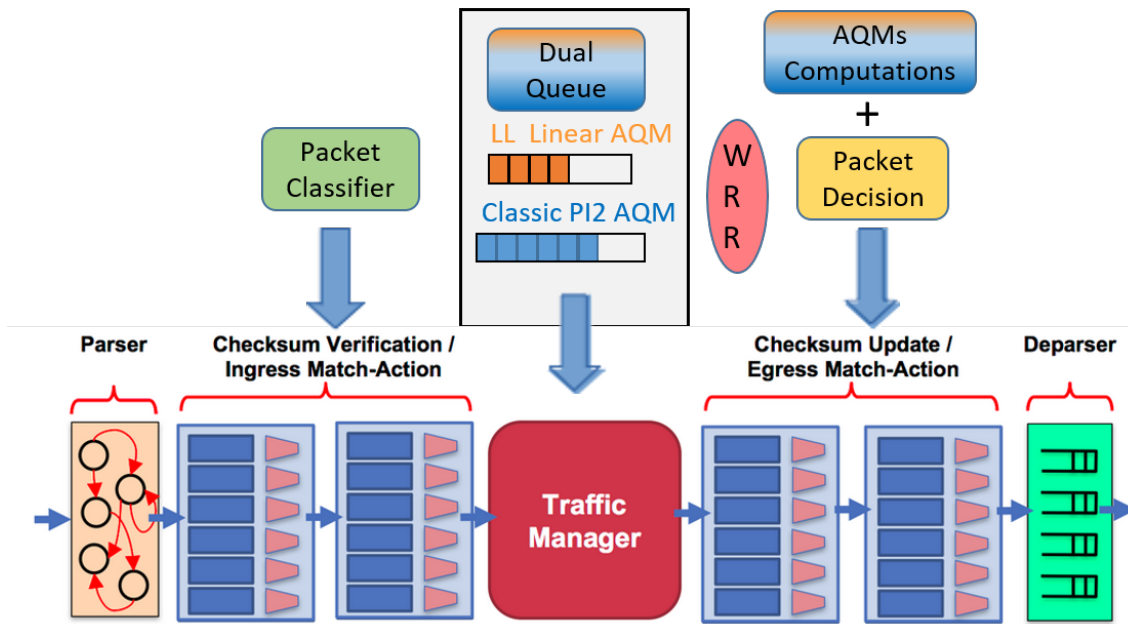


Figure 2.9: P4-based implementation of L4S

provide sufficient security. Furthermore, since the program is aimed at being deployed in a network equipment processing packets at line-rate, we should implement a fast and efficient solution.

- We base our implementation on the P4 Linux BMv2 software. However, it is not fully adapted to our needs, since it currently implements a strict priority queueing mechanism (i.e., a lower priority queue is only served if all higher priority queues are empty) and with such a policy, the LL traffic will starve the BE traffic. It is then necessary to rethink the queue management policy of the BMv2 software to have two distinct queues, together with a weighted round robin (WRR) scheduler, configured with a 1/16 weight as recommended by the IETF [22] to unqueue packets and achieve the desired bandwidth sharing policy between the LL and BE queues. We also have to add throttling at the queue level since BMv2 only throttled the packet rate at the output port level thus creating an unmanaged bottleneck defeating the purpose of the two AQMs. It should be noted that these modifications have currently to be done in the BMv2 switch because the P4 traffic manager is not yet programmable [15] and we expect next P4 releases to change that.
- We implement the AQMs computations of L4S in the egress part of the P4 program. The P4 framework offers an API (Application Programming Interface) to retrieve the time spent by one packet in a queue, which is used as input value to the AQMs. P4 also offers registers to store/retrieve data common to the switch i.e. not specific to the packet being processed. They are used for the last update time, the last mark/drop probability and the last delay in the queue, required by the PI2 algorithm [12]. Finally, the PI2 parameters (Update time,  $\alpha$ ,  $\beta$ ) are stored in a table of the P4 program and configurable by a controller. The latter can then dynamically modify those values to change the behavior of the PI2 AQM. We implement a linear AQM and a PI2 AQM as suggested by the IETF, but if other AQMs were known to be better suited, it would be possible to update network devices with another P4 program.
- After AQM computations, based on their respective probability, a decision for the LL and BE packets is taken: forward as is, mark or drop. The LL probability is the maximum between the probability output of the native AQM (LL traffic) and the probability of the base AQM (BE traffic) multiplied by a factor K (2 by default). This is where the "coupling" feature is, leading not to penalize the BE traffic.
- Finally, the deparser of the P4 program has to reconstruct the final packet before forwarding it, if the packet has not to be dropped.

This current implementation of L4S in P4 is satisfactory, but since it is designed as micro-services, we can imagine to change the modules behaviour. For example, we imagine to have a more intelligent

packet classifier, adapted to less controlled environment, which will no longer only analyse the ECN bits, but which can detect a low-latency flows based on flow patterns analysis (with Machine Learning techniques for instance), which could more securely classify traffic, avoiding malicious applications to play with the ECN flag. If this processing is too complex to be programmed in P4, an option we envision is to deport this network function, which might be deployed as a Virtual Network Function and connect it to a lightweight classifier module developed in P4, running in the network element, remotely configurable by the main module. Another example is to change the AQM computations module if the research community provides a new solution, more efficient than PI2 for instance, or the packet decision if decision should be made upon different considerations than not just the probability. For example, we imagine to drop or mark packets of sessions consuming more the queues than other sessions having a lower bitrate and queue occupancy. This could be done changing the code of this part.

### 2.2.2 Local testbed

A local testbed has been set up in order to evaluate that the P4-based L4S switch (see section 2.2.2) behaves as expected, i.e. that the LL packets are processed within the requested time and that the LL traffic does not starve the BE traffic. We also wanted to compare it with the linux-based L4S solution [12][6]. Furthermore, since not already done by any research team, we wanted to evaluate the L4S behavior with time varying network conditions. Indeed, in current networks, the main bottleneck is the access network (e.g., the radio link between a user equipment and a cellular base station) where congestion can often happens because of the limited and varying capacity of the radio link. Cellular networks, where latency can greatly change, is then a key issue for L4S and it is therefore crucial to evaluate it in such conditions, what we performed with our L4S system.

The local testbed (Fig. 2.10) consists of :

- 1 client and 1 server for a low-latency (LL) connection. For this LL connection, both endpoints implements the TCP-Prague protocol and the Accurate ECN bit allowing fine-grained estimation of the possible network congestion.
- 1 client and 1 server for a best-effort (BE) classic connection. For this BE connection, both endpoints implements the classic TCP protocol with the widely used Cubic congestion control algorithm.
- A network switch, being either the P4-based L4S system or the Linux-based L4S program, depending whether we evaluate the P4-based L4S solution or the linux L4S solution.

The 5 entities are hosted on the same laptop (a HP Elitebook 840, i5 bi-core and 8 GB Ram), the clients and servers being launched in a dedicated KVM virtual machine, whereas the network node runs in the bare linux.

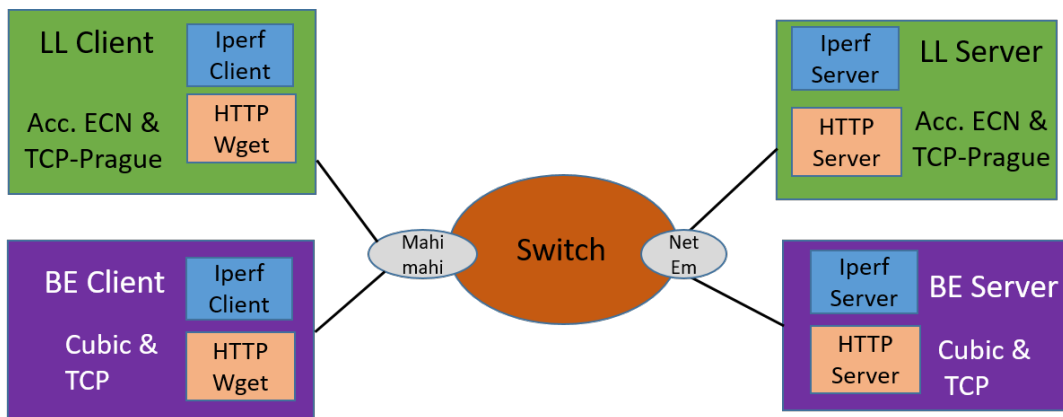


Figure 2.10: L4S Testbed

To precisely evaluate the performances and behaviour of the L4S system, we instrumented the switch to measure the time spent by packets in each queue of the L4S solution, the number of packets being in the queues after the one just sent, and the number of unused transmission opportunities. We configured the P4-based switch to start notifying possible congestion for LL traffic with a low threshold at 3ms.



For the BE traffic, this value is configured at 15ms. For bandwidth sharing, we measure the throughput received by each client and the senders' congestion window.

We performed the tests with Iperf3 (during 300sec) and with a HTTP download (using lighttpd/Wget and content files of 1Mb, 10Mb and 100Mb). The sender tries to send as much packets as it can and based on the received ECN marks (indicating congestion) for the LL traffic or on the detection of lost packets for the BE one, it reduces its bitrate.

We configured delay (5ms, 10ms, 20ms, 50ms and 100ms as done in [6]) with NetEm to emulate RTT between the servers and the switch.

On the client side, we perform tests with constant bitrate configurations at 4 Mbps, 12 Mbps, 24 Mbps, 40 Mbps to have the same tests configuration than for the evaluation of linux-based L4S solution [6] when we wanted to evaluate our P4-based L4S solution.

However, since in real cellular networks, the bitrate is not constant and can greatly vary, we should have another configuration to evaluate L4S under such time varying conditions. Mahimahi [20] offers a tool named Link Shell which allows to schedule the transmission of packets at a given interface, based on the transmission opportunities (txops) specified in a file. Several such files, generated from measurements on real cellular networks, are available as open data in Mahimahi's github [2]. To use the same time varying behaviour as in Mahimahi with our P4-based L4S solution, we modified the scheduling process of the P4 BMv2 switch to reproduce the Link Shell behavior and send packets according to the transmission time provided by the files capturing real txops measurements.

We performed tests with two Mahimahi txops files (Verizon and TMobile dataset) but the L4S system reduced the low-latency traffic to a very low bitrate, sometimes leading to connections resets. Analyzing the datasets showed the mean bitrate is low since the duration between two txops (inter-txops time) is sometimes long, much more than the configured L4S threshold (3ms) and that very few packets could be sent during txops. We think these txops files, extracted from network captures dated from 2016, are no more representative of current cellular bitrates and decided to make our own captures, using the saturator tool [25][1]. This new dataset shows an average bitrate of almost 19 Mbps, shorter inter-txops times and more packets sent during txops. Fig. 2.11 shows the CDF of inter-txops time and the CDF of the number of packets transmitted per txop for the Verizon and Orange datasets. We can see that with the Orange traces, about 80% of the inter-txops times are less than 3ms, whereas it is below 70% for Verizon, and that 3 or more packets can be sent per txop about 50% of the times, whereas Verizon offers it for only 5% of the time.

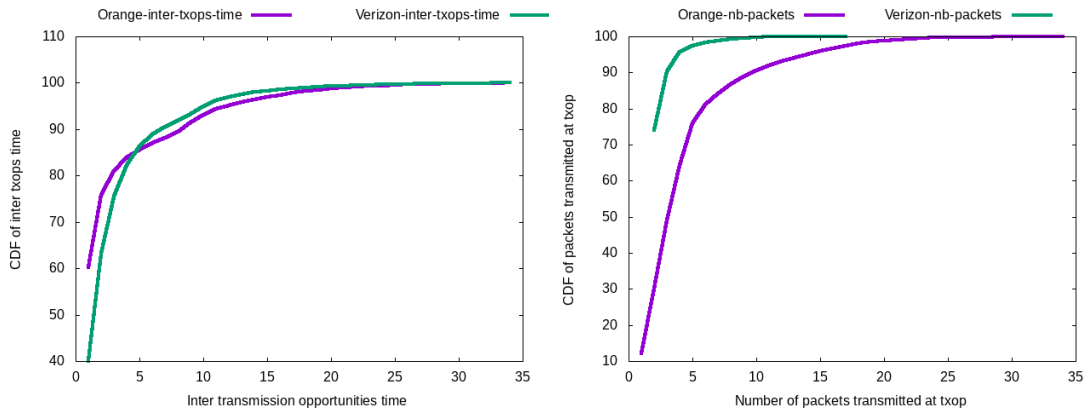


Figure 2.11: Verizon and Orange Mahimahi traces file, left) inter txops time; right) packets transmitted per txop

### 2.2.3 P4-based L4S vs Linux-based L4S

In this section, our goal is to validate the P4-based implementation of L4S by comparing its results with the ones obtained by the Linux-based reference implementation [12], also validated by [6]. As we can see on Fig. 2.12 and Fig. 2.13, bandwidth sharing between the LL traffic and the BE traffic is very fair (almost the same bitrate, ratio close to 1) for our P4 implementation. Our solution is very stable and

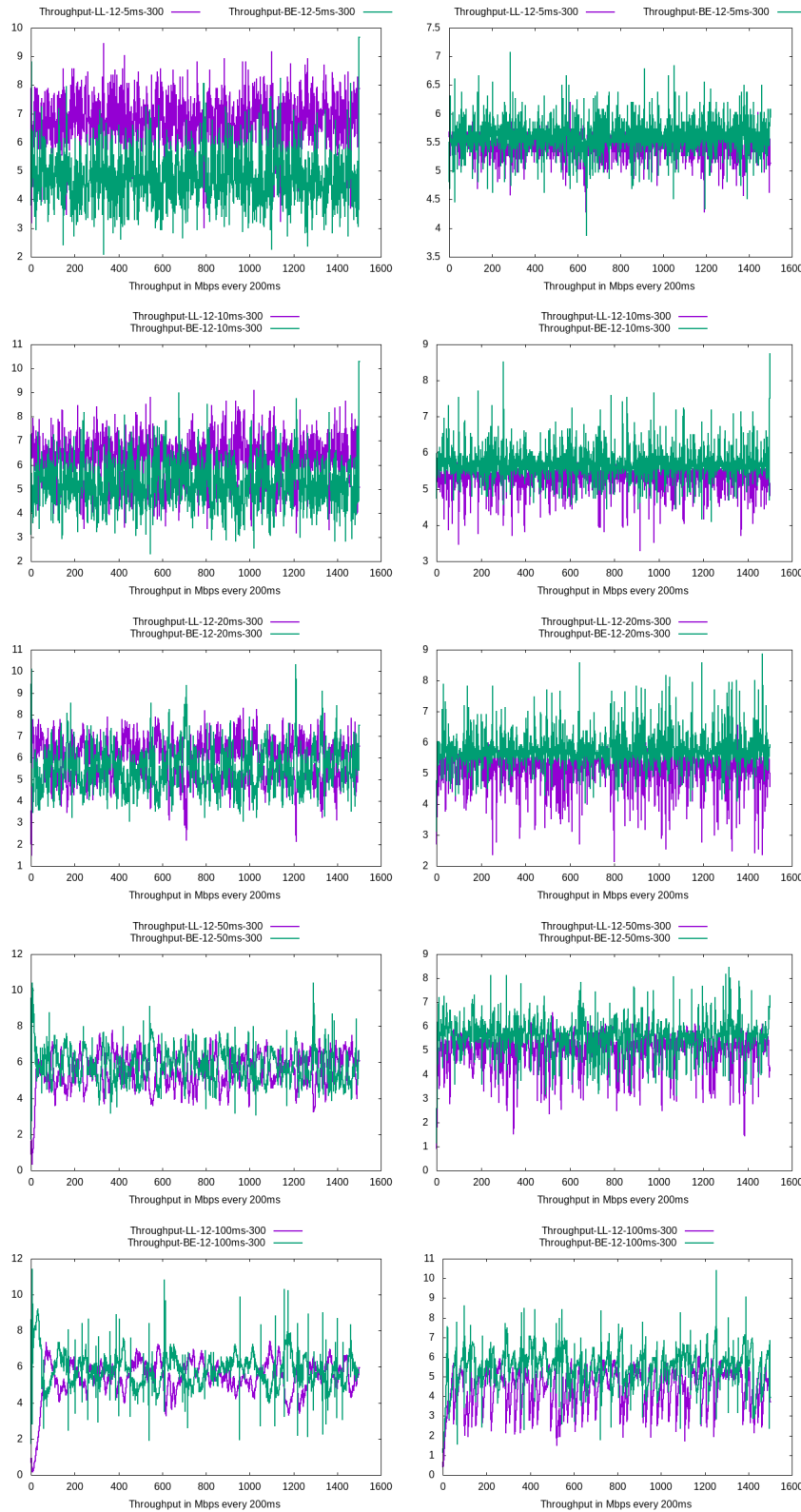


Figure 2.12: Bandwidth sharing with delay=5, 10, 20, 50 and 100ms for left) Linux implementation; right) P4 implementation

even better than the Linux-based one, where the ratio decreases as the delay increases. One reason can be that the scheduler is different. For Linux, the authors implemented a Time-Shifted FIFO scheduler, whereas we implement a Weighted Round-Robin one, assumed more stable, as recommended by the IETF [22].

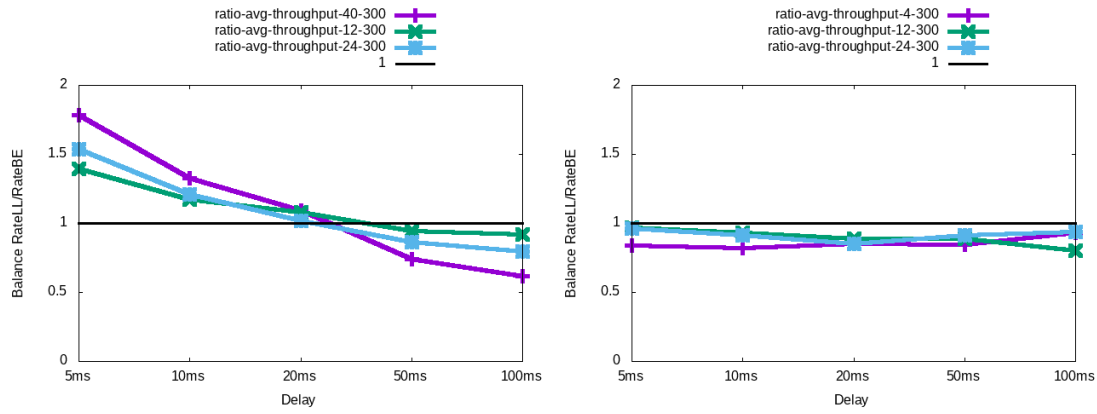


Figure 2.13: Bandwidth sharing ratio with delay=5, 10, 20, 50 and 100ms for left) Linux implementation; right) P4 implementation

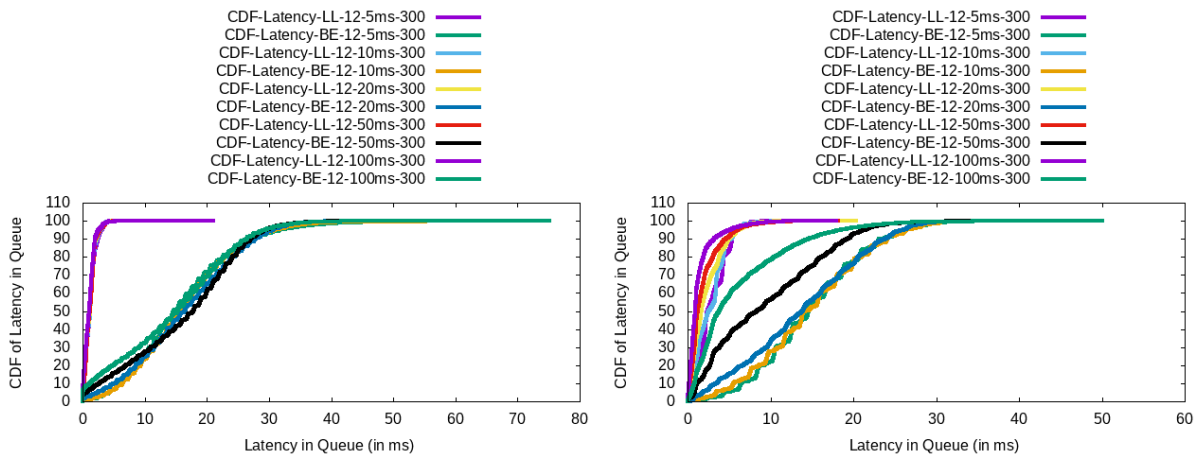


Figure 2.14: CDF of latency in queue (LL & BE) with Bandwidth=12Mbps and delay=5, 10, 20, 50 and 100ms for left) Linux; right) P4

For the time spent by the packets in the two queues (Fig. 2.14), we can see that our solution ensures that LL packets are delivered in a very short time as L4S requires, most of the times in less than 3ms, and less than 5ms for 100% of the packets. Our implementation is a bit less stable than the Linux-based software, but this is not crucial. This is due to the linux L4S software running in the kernel space, while our P4 program runs in the user space, having thus longer and more variable processing times. Indeed, we roughly measured a processing time of a few micro-seconds for the linux L4S module and a time of hundreds of micro-seconds for our P4 implementation. Since the final goal is to deploy such a P4 program onto hardware P4 switch, this issue would be avoided as processing times would be much shorter.

For the BE traffic, the delivery is a bit longer but in a reasonable time, in line with the PI2 configuration of L4S. In our solution, the BE delay in the queue is a bit shorter than the Linux one. This is due to our WRR scheduling implementation, modified to optimize the transmission opportunities. Indeed, if there is no packet in the LL queue while it is scheduled to send one, we send one (if any) of the BE queue so as not to lose this transmission time.

Performing tests with a HTTP download, using the WGET command, we wanted to evaluate if the size of the contents to download can have an impact of the queuing mechanisms. Fig. 2.15a shows that the size of the downloaded data file has no impact on the time spent by each packet in each queue and that the time is very stable. There is one exception for the BE traffic, the curve for 1MB is not exactly aligned, but the reason is that 1MB size is small and do not allow to reach the stability phase.

Fig. 2.15b is another representation of the time spent in queues by the LL and BE packets. In this figure, each point represents the time for one packet. We can see that the forwarding of LL packets are quite stable while it can greatly vary for BE packets. Indeed, since the sender tries to increase its bitrate, the number of packets in the queue can increase and the time to forward it naturally also increases,

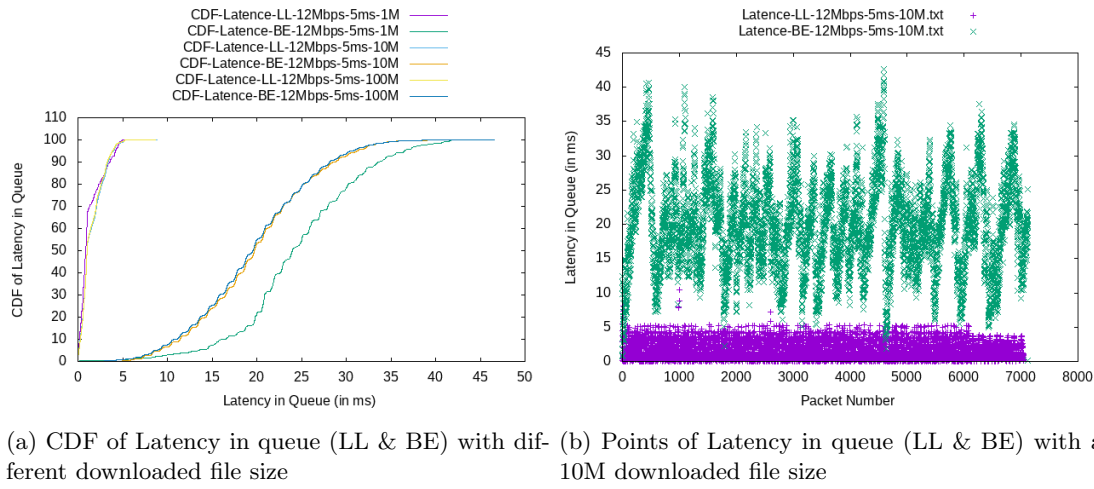


Figure 2.15: Latency for a bandwidth=12Mbps, delay=5ms, for the P4 implementation

because the output throughput of the switch is still limited to 12 Mbps. Those variations highlight the classic Cubic TCP behavior.

Finally, we measured the number of packet retransmissions of the TCP flows for each configuration. Unsurprisingly, Fig. 2.16 shows that BE traffic, trying to fill the bottleneck buffer and only reacting to packet loss, has a lot of retransmissions, while for the LL traffic, reacting at the congestion onset and adjusting to the congestion level, we have no retransmission (only 1 remitted packet in the beginning of the test for 2 configurations, probably due to the slow-start phase overshooting the AQM threshold).

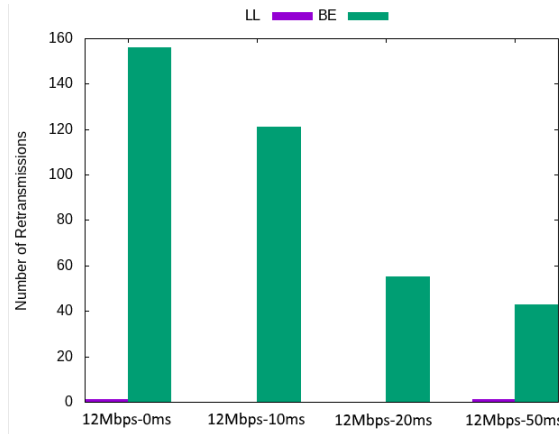


Figure 2.16: Number of retransmitted packets for LL and BE traffic for 12 Mbps bandwidth with varying RTT for the P4 implementation

To conclude this comparison, we can say that our P4-based implementation of L4S works very well and offers the expected behavior for both types of traffic.

## 2.3 Addressing the Limitations of the L4S Architecture

The MOSAICO project addressing low-latency services delivery in 4G/5G network and security, it is crucial to evaluate the L4S architecture in such a context. In this section, we present our evaluation of L4S, highlighting its current limitations when used with cellular network and for managing traffic threats.

### 2.3.1 L4S under time varying network conditions

L4S has been evaluated only under constant network conditions [12][6]. However, in real cellular networks, the bitrate is not constant and can greatly vary. L4S behavior then needs to be evaluated under

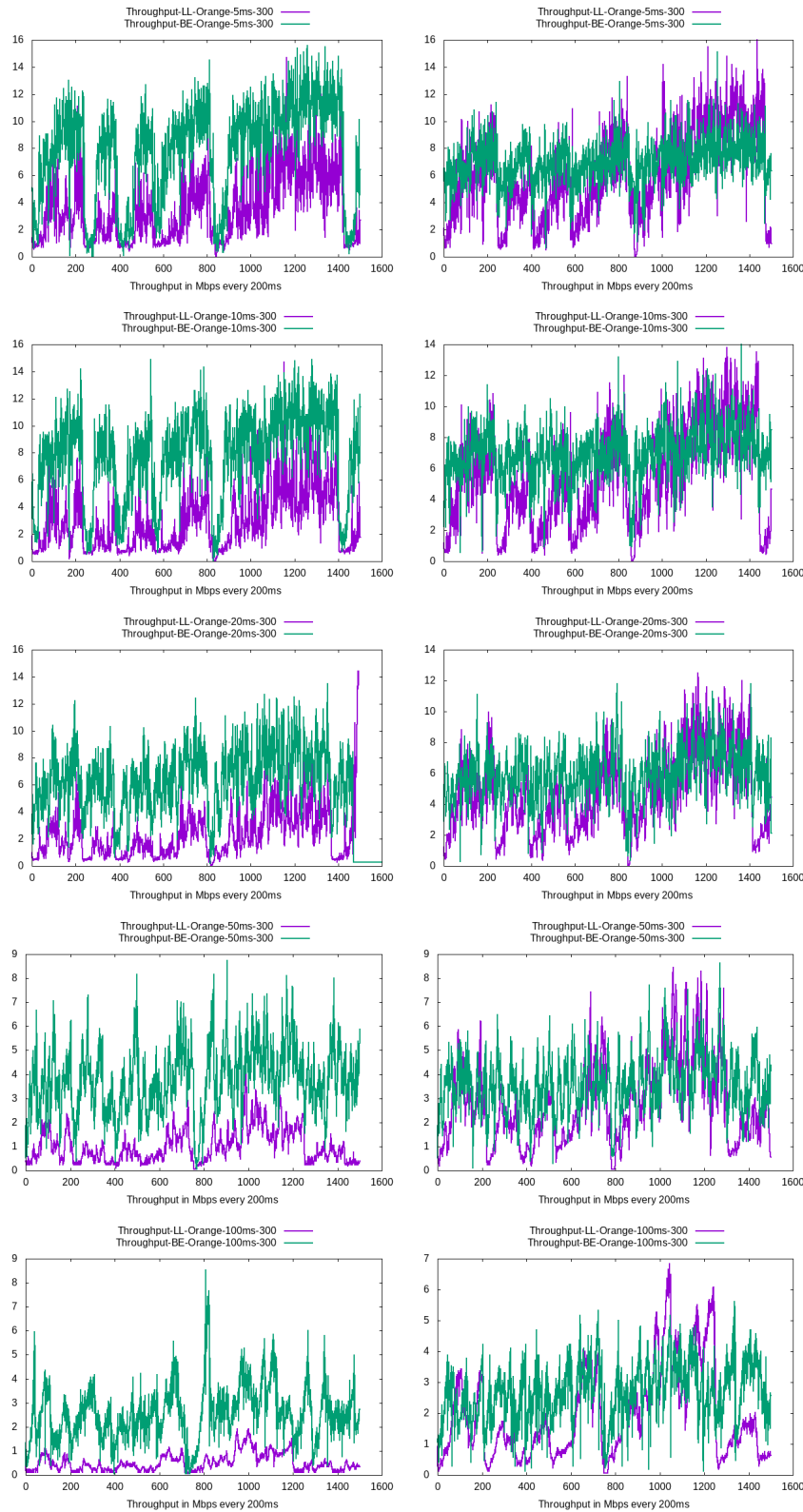


Figure 2.17: Bandwidth sharing (in Mbps) with Orange traces, delay=5, 10, 20, 50 and 100ms, for left) L4S with Threshold=3ms; right) L4S with Threshold=9ms

such time varying conditions. Using the tested presented in Fig. 2.10, we performed such evaluations using a functionality similar to the Link Shell tool offered by Mahimahi, as described in 2.2.2.

With the Orange file, we were able to perform tests but even if the file corresponds to a higher



capacity cellular network, the results of the L4S system is far from being satisfactory, compared to the tests performed with a constant bitrate.

Indeed, although the Orange file leads to a mean bitrate of about 19 Mbps, the L4S system behaves less efficiently (Fig. 2.17-left) than the tests performed with a constant bitrate at 12 Mbps (Fig. 2.12-right), the LL traffic throughput being far from the BE traffic throughput. The reason is that L4S very often marks LL packets because the time spent by the packet in the LL queue is frequently more than the configured 3ms threshold. This marking leads to sender's reduction of the bitrate. This is a normal behavior looking at the CDF of the Orange dataset (Fig. 2.11-left) where we can see that about 20% of the dataset have an inter-txops time above 3ms. Furthermore, the LL queue can only hold a few packets so as to ensure low latency. Indeed, more than 5 packets in the queue leads to marked packets (with a rough estimation of a constant 19 Mbps bitrate, one packet is sent every 0.65 ms, thus 5 packets in the queue are enough to reach the 3ms threshold before marking LL packets). This is confirmed by Fig. 2.18-left, where the LL queue size contains more than 5 packets less than 5% of the time, while the Orange network can simultaneously send more than 5 packets 20% of the time (Fig. 2.11-right). Thus, the txops are wasted because there is no packet in the LL queue.

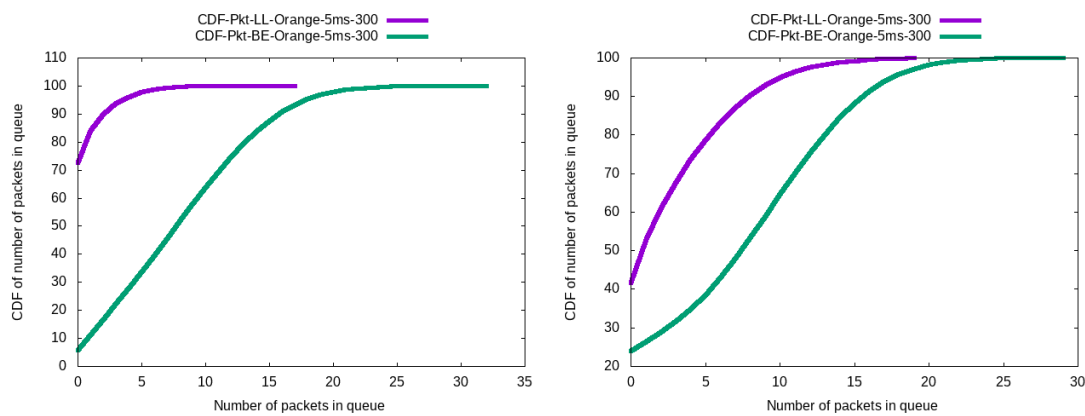


Figure 2.18: CDF of the number of packets in queues for P4-based L4S with Orange traces and delay=5ms, for left) L4S Threshold=3ms; right) L4S Threshold=9ms

To confirm this, we measure on the P4 node the number of unused transmission opportunities. We can see that this value is huge for the LL traffic, about 140000 for a 5ms delay, which is 28 % of all the txops (Fig. 2.19-left), while it is about 10000 (3% of txops) for a constant bitrate of 12 Mbps.

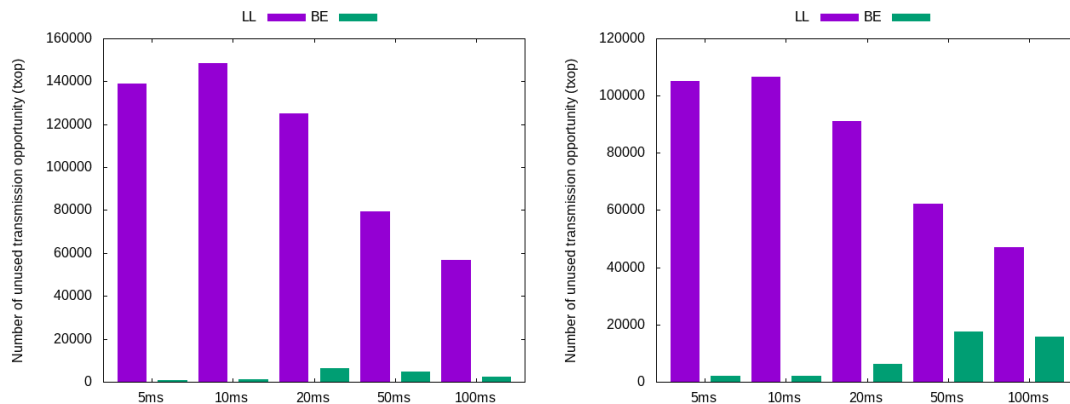


Figure 2.19: Number of unused txop for P4-based L4S with Orange traces, for left) L4S Threshold=3ms; right) L4S Threshold=9ms

We also measure on the server's side the congestion window (with the linux ss tool), and Fig. 2.20-left highlights that the congestion window is always quite low for the LL traffic, and when it tries to increase, the sender quickly reduces it, because of the marked packets. Contrarily, the congestion window for the BE traffic is much higher.

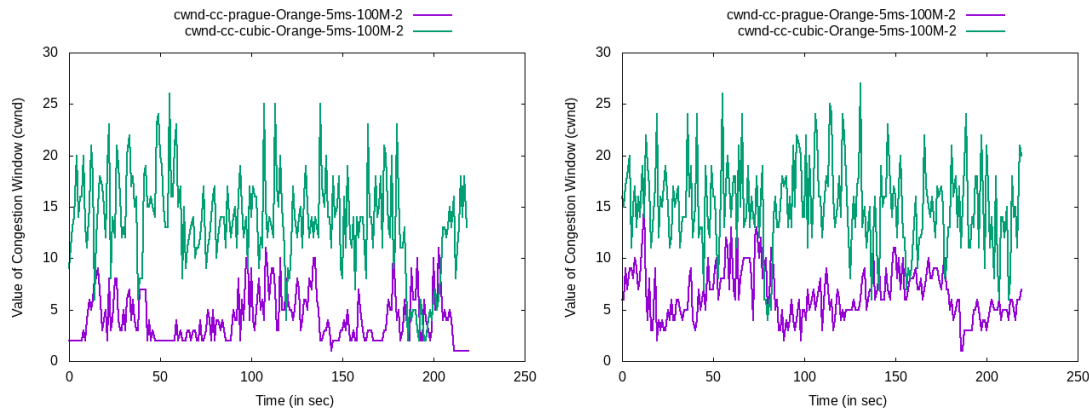


Figure 2.20: Congestion Window for P4-based L4S with Orange traces, delay of 5ms for left) L4S Threshold=3ms; right) L4S Threshold=9ms

To double-check that the current L4S architecture, with a low threshold value for LL traffic, is not adapted to current time varying network conditions, we perform the same tests but with a LL threshold configured at 9ms (instead of 3ms). This value is far from being satisfactory for a low-latency application, but it just aims to confirm the effect of the variability of the inter-txops time on the L4S system.

With such a threshold and with the same Orange dataset, the results are much better. We can see that the LL traffic can be transmitted at a higher bitrate, much closer to the BE traffic (Fig. 2.17-right), as with a constant bitrate. The number of unused txops (Fig. 2.19-right), while still high, is lower than the one with a threshold of 3ms (about 30% lower) and that the number of packets in the LL queue size is more than 5 packets about 20% of the time (Fig. 2.18-right). Looking at the servers' congestion window (Fig. 2.20-right), we can see that the LL server can increase much more its window, confirming higher delivery rate because of less frequently marked packets.

This evaluation allows us to conclude that the L4S architecture is sensitive to the inter-txops time of the network and requires frequent txops. The number of packets that can be transmitted during a transmission opportunity is less critical since the small number of packets in the LL queue limits its effect. In current cellular networks, L4S is then not really effective and would require shorter time transmission intervals, such as allowed by 5G New Radio numerology.

### 2.3.2 Threats targeting low-latency traffic with L4S

In this section, we wanted to characterize the impact of an undesirable flow to a legitimate flow. To that aim, we used the following topology, depicted in Figure 2.21. A baremetal server act as a router and is equipped with an Intel(R) Xeon(R) CPU E5-2430 v2 @ 2.50GHz, and two Intel Corporation Ethernet 10G 2P X520 Adapter runs DualPI<sup>2</sup>. One Virtual Machine for each endpoint is hosted in an OpenStack cloud platform. L4S nodes are using the Linux image from the L4STeam<sup>5</sup> and are configured to use ECN with the convenient codepoint to be classified in the low latency queue. Receivers are connected to the same router interface (*eno1* in the figure), classic senders and low latency senders are connected to different interfaces. On the egress direction of *eno1* (receivers side), DualPI<sup>2</sup> is configured with a 10 Mbps rate, all other parameters left to default, i.e. 10000 packets limit, the coupling factor  $k$  is set to 2, *drop\_on\_overload* is the strategy to adopt when high congestion occurs, the target queue delay is 15ms for the classic queue, aggregated packets are split with the *split\_gso* option and the step threshold, in other words, the sojourn time threshold from which DualPI<sup>2</sup> will always mark exceeding packets within the low latency queue, is set to 1ms.

The legitimate flows are generated with *iperf3* and controlled by TCP Prague while the malicious node is using QUIC, a protocol based on UDP which is enhanced in the userspace with congestion control algorithms and other features that are present in TCP. This lets a malicious user to easily modify the sending behavior without having to deal with the Linux kernel networking stack. We especially chose to use *picoquic*<sup>6</sup>, a minimalist implementation of QUIC which has been forked by the L4S creators to

<sup>5</sup><https://github.com/L4STeam/linux>

<sup>6</sup><https://github.com/private-octopus/picoquic>

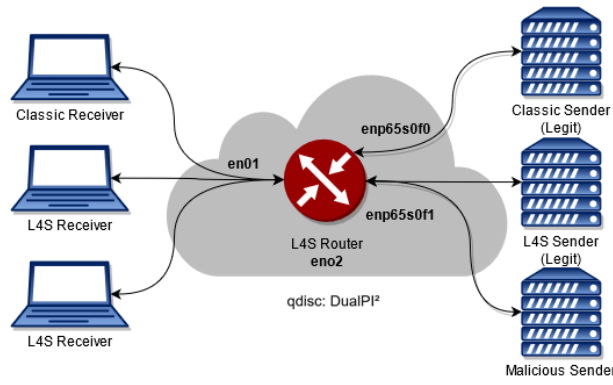


Figure 2.21: Network topology for our experiments

support the Prague congestion control. We can adjust *maxrate* and enable or disable pacing with the system program traffic control (*tc*).

The base RTT is set to 15ms with *wondershaper* in server's interface. Metrics are collected from the endpoints with calls to socket statistics (*ss*) 4 times per base RTT. This provides the *tcp\_info* data structure of the kernel, which contains reported RTT and its mean deviation, the congestion window (*cwnd*), maximum potential sending rate based on the formula:  $\frac{cwnd \times MSS}{RTT}$  and last bytes acked. For the router, metrics are collected at the same frequency using *tc*, which provides metrics from DualPI<sup>2</sup> such as classic and low latency queue occupation and queuing delay, marking probability from the PI controller (base probability), the amount of ECN marks and the amount of dropped packets. Finally, each experiment lasts 60 seconds.

## Reference Traffic

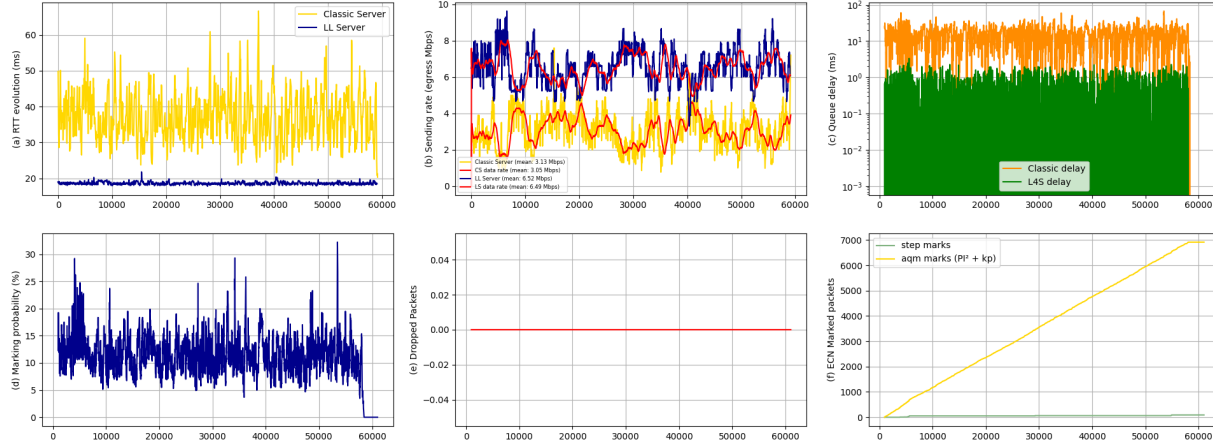


Figure 2.22: Standard behavior of the L4S architecture with one LL and one classic flows. (a) RTT evolution (ms). (b) Sending rate (egress Mbps). (c) Queue delay (ms). (d) Marking probability (%). (e) Dropped packets. (f) ECN Marked packets. Horizontal axis is the time in ms.

In order to highlight and quantify the impact of undesirable flows, we need to have a control sample to compare with our different scenarii. To that aim, we consider that the measurements depicted in Figure 2.22 act as the baseline of our experiments. The reader can refer to it to identify the traffic alterations due to the different undesirable flows we consider subsequently. Figure 2.22 shows regular network conditions when there is one classic flow and one low latency flow. In that situation, the router sends 9.54 Mbps, shared, in accordance with Figure 2.22.b, which represents the sending rate. The maximum potential sending rates are depicted in blue and yellow and the red line indicates the actual data rate, based on received acked bytes. Given the behavior of TCP Prague, it tries to takes all the available bandwidth. For the classic flow, the data rate is limited to 5 Mbps to make sure that the observed effects are not due to natural saturation of the router. Figure 2.22.c shows the differentiated queue delay in the LL and the classic one and the corresponding RTT is represented in Figure 2.22.a. We can observe that the RTT of

the classic flow is much more variable than for the LL flow which is close to the base RTT set to 15ms in our case. As we can see, flow congestion is well handled by ECN marking and even though the LL traffic wants to take all the possible bandwidth, Figure 2.22.e shows that no packet drop is necessary to guarantee the flows respective requirements. The number of marked packets is measured in Figure 2.22.f. For Low-Latency packets, the decision is based on the maximum between the probability of LL AQM (which corresponds eventually to the number of step marks also showed in Figure 2.22.f) and the base probability of the classic AQM, showed in Figure 2.22.d, multiplied by the coupling factor  $k$ .

## Results Analysis

This section explores to what extent L4S is affected by an unexpected behavior in the presence of undesirable flows.

**Misbehaving flow: Unresponsive ECN** As a misbehaving flow, we first propose to implement an ECN abuse attack targeting the increasing of a legitimate flow latency. The protocol manipulation we implemented consists in being unresponsive to congestion notification while respecting ECN signaling. This was implemented by removing the congestion windows reduction when updating the coefficient of reduction in Prague. As such, it expects to saturate the LL queue to increase the marking probability (leading eventually to some packet drops) and thus generate an extra delay, sufficient enough, to make the LL application unusable. A side effect of this attack relies in bandwidth stealing: such a behavior will indeed take advantage of the reduction of other participants when a congestion event occurs, resulting in the robbery of all the available bandwidth from legitimate users.

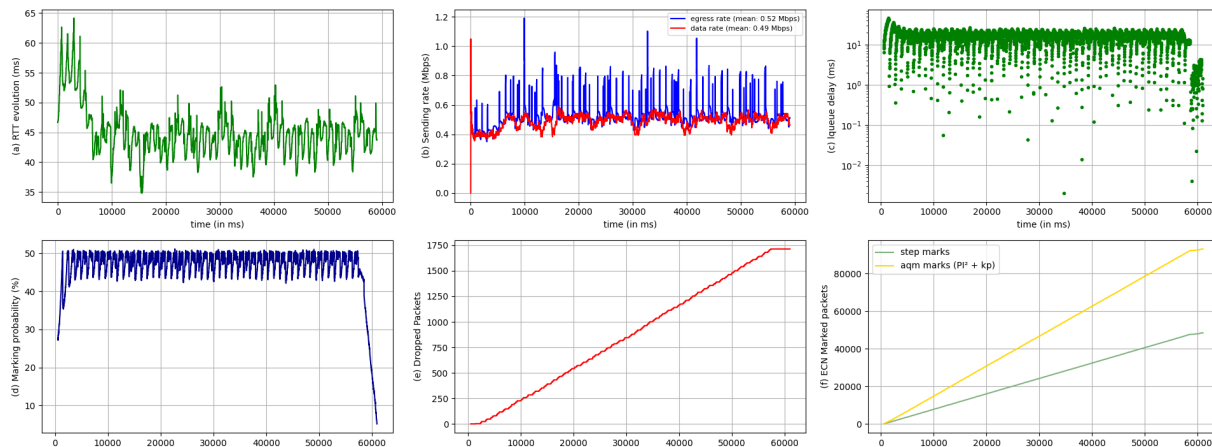


Figure 2.23: Unresponsive ECN impacts on a legitimate low latency flow. The vertical axes of the subfigures are the measured metrics (two first ones show metrics from the legitimate flow, the others show metrics from the router) and the horizontal axis is the time in ms.

Figure 2.23 shows time series for the different metrics we consider in the context where the malicious user is generating such an unresponsive ECN-capable flow. We can see that it brings a higher queue delay which in return increases the average RTT. The RTT is twice bigger than that of Figure 2.22. Unresponsive ECN puts DualPI<sup>2</sup> to saturation and triggers the *drop\_on\_overload* reaction. The amount of ECN marks is twice bigger than step marks (i.e. amount of ECN marks due to exceeding the threshold). We can also notice that the marking probability is around 50% yet rather stable. The sending rate is also stable yet very low. This is due to the fact that the legitimate flow responds correctly to congestion notification while the attacker steals all the available bandwidth pretending reducing the sending rate.

**Unresponsive flow: Bursts** For the unresponsive behavior, we generate traffic bursts from a classic sender to disturb a legitimate flow in the low latency queue. To that aim, we configure the ECN flags of the attack traffic to be classified in the classic queue. We saturate the low latency queue with traffic from the classic one. The attack consists here in playing with the coupling mechanism. The bursty behavior will trigger a high queue variation which is more likely to increase the marking probability due to the PI proportional gain  $\beta$  in comparison with a constant saturation.

Figure 2.24 shows the results we collected with such a malicious user generating a bursty flow within the classic queue. The marking probability is erratic but is around 20% in average. Both ECN marks

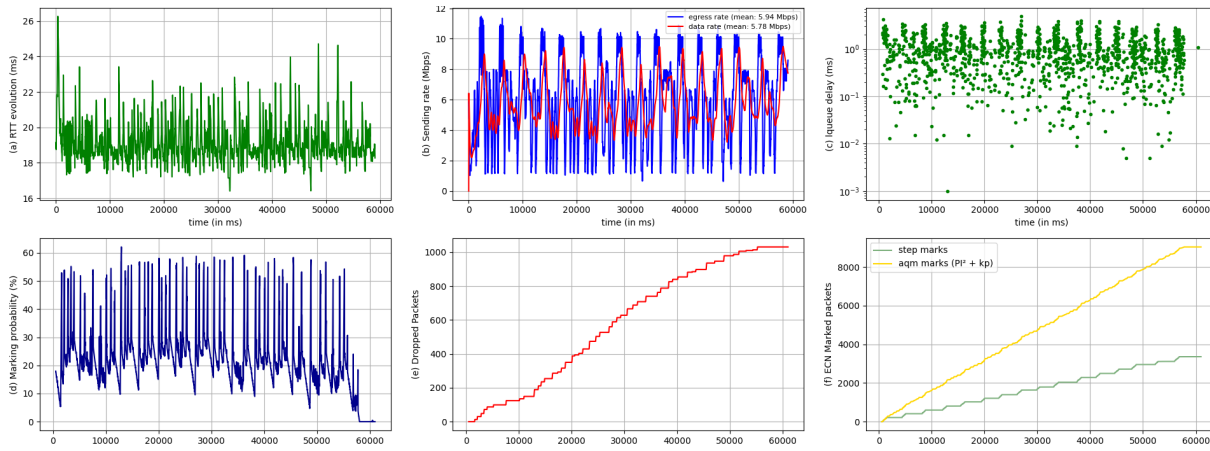


Figure 2.24: Bursts within the classic queue impacts on a legitimate low latency flow. The vertical axes of the subfigures are the measured metrics (two first ones show metrics from the legitimate flow, the others show metrics from the router) and the horizontal axis is the time in ms.

and step marks have a step pattern and occur ten times less often than in case of Unresponsive ECN. As anticipated previously, the marking probability is very sensitive to high variation due to the weighting factor  $\beta$  of the PI controller. Thus, when a burst occurs, DualPI<sup>2</sup> reacts very quickly to punctual events, resulting in a sending rate reduction from the legitimate user which in the end induces heavy fluctuation and wide dispersion of sending rate possible values. RTT and low latency queue delay are also affected by 4ms when a burst occurs, even though the burst takes place in the classic queue. This is due to the coupling mechanism in DualPI<sup>2</sup> which is designed not to hurt classic flows since it is rather preferred to sacrifice L4S delay on saturation to ensure a fair flow coexistence. More precisely, when the LL queue delay is under 1ms, the marking probability of the LL queue is governed by the base probability multiplied by the coupling factor. This is why, in the end, the sending rate of the legitimate low latency flow is affected.

**Malformed flow: No pacing** A solution to avoid bursty emitting pattern to be sent from an endpoint is to pace packet emission over a RTT instead of sending a bulk. This operation is accomplished by the TCP stack of the Linux kernel and also as a queuing discipline with Fair Queuing (FQ). Consequently, in this experiment, we used FQ as a queuing discipline at the sender endpoint, which lets us control the pacing on the egress traffic as well as the maximum rate limitation in order to control the sending rate of the application layer. When pacing is disabled, it generates undesirable flows. This kind of undesirable flow is meant to generate micro-saturation with sub-RTT bursts (also termed micro-bursts). We aim at seeing the impact of pacing and at generating micro-bursts when disabled.

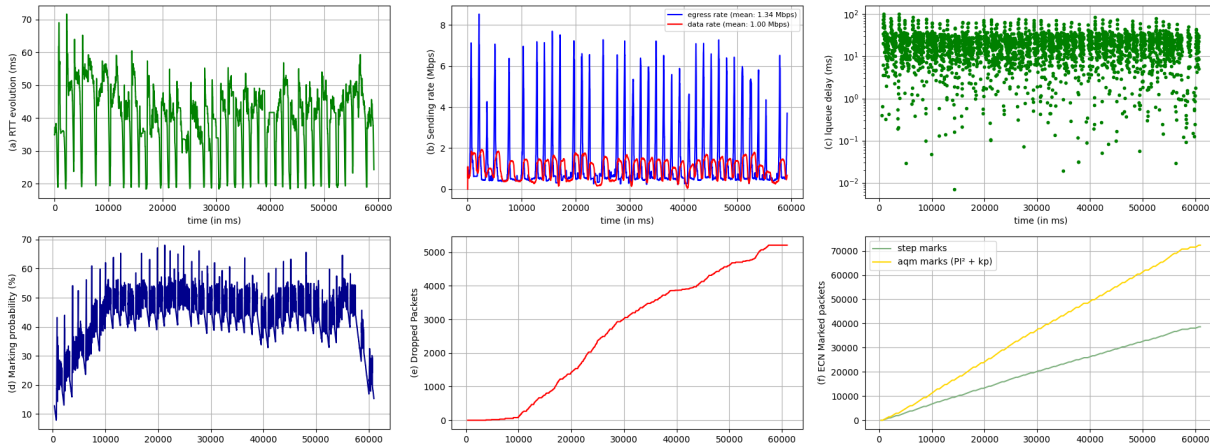


Figure 2.25: Malformed flow impacts on a legitimate low latency flow. The vertical axes of the subfigures are the measured metrics (two first ones show metrics from the legitimate flow, the others show metrics from the router) and the horizontal axis is the time in ms.



Figure 2.25 shows the results we collected in the context where the malicious user is generating such micro-bursts. The marking probability is less erratic compared to the previous experiment, but still not stable and is in average around 50% and sometime even more, which is similar to the Unresponsive ECN case. This leads to a large number of ECN marks and by far to the highest number of dropped packets among the three experiments. RTT is very erratic, averages around 40ms, and has the widest dispersion among other measured undesirable flows of our study. We can see that it is mostly due to LL queue delay which has the widest dispersion. The sending rate is higher than for the Unresponsive ECN scenario because the legitimate flow is able to increase his congestion window between two micro-bursts.

## Discussion

order to clearly assess to what extent undesirable flows induce changes on the different metrics we observe as compared to our reference experiment, Figure 2.26 summarizes the results in boxplots.

Generally speaking, we can see that bursty traffic has a limited impact on the RTT when it lodge itself within the classic queue but it can bring a lot of variation in the sending rate when it passes through either queues (Figure 2.26.a and 2.26.b). Micro-bursts from malformed flows are on the other hand clearly responsible for packet drops, as we can observe in Figure 2.26.e. We can conclude that pacing is essential within endpoints to support the data plane performance. Unresponsive ECN impacts the marking probability, the number of marked packets and the low latency queuing delay (Figures 2.26.d, 2.26.f and 2.26.c). Contrasting unresponsive ECN with malformed flows, we can see that pacing results in less packet drop, thus a higher number of marked packets, a stable sending rate and RTT and limits effects on LL queue delay from the Unresponsive ECN attack. However, regardless the undesirable flow we consider, they all clearly exhibit an impact that defeats the expected property of the L4S architecture and eventually the LL applications running in endpoints.

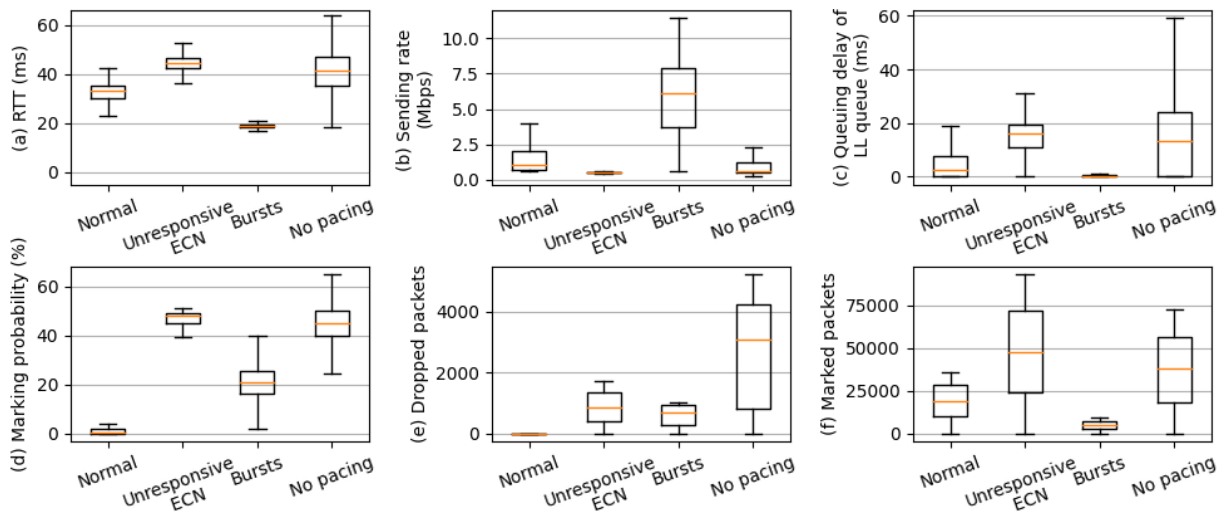


Figure 2.26: Multi-criteria comparison of the impact of undesirable flows on a legitimate LL flow. Vertical axes are measured metrics represented as a boxplot with the median value and first and third quartile and the first and ninth decile. Horizontal axis is the type of undesirable flow.

## Conclusion

The L4S architecture is a promising approach to deliver low-latency contents under a few milliseconds. But, to be deployed in operational networks, such a solution should be robust to attacks and to legitimate undesirable flows. In this section, we have studied to what extent the L4S architecture can be threatened by several types of undesirable flows. By implementing and evaluating three type of abnormal flow behaviors, we have quantified their impact and proved that the current L4S architecture cannot efficiently deal with them since eventually the low-latency requirement is defeated as well as the jitter and the sending rate.

The three main threats have been evaluated independently to isolate their effect on L4S, but further studies need to be made to see what happen when these undesirable flows are combined. Consequently, our ongoing work concerns undesirable flows combined altogether and preliminary results seems to show

that pacing at senders can drastically reduce most of the impacts. Besides, we also plan to study the impacts of each presented attacks when we vary their parameters to determine where each attack is the most powerful and by contrast how an attack can dissimulate its behavior to be undetected.

In our mid-term future work, we will investigate detection modules, which can detect such attacks, via for example traffic pattern analysis (e.g. inter-arrival time or packet size) or via machine learning techniques. Finally, our ultimate objective consists in defining countermeasures, which can help to mitigate such attacks, and consequently enable a safe and stable operation of low latency forwarding in the Future Internet.

## 2.4 Micro-services for Monitoring

Following the initial view of the global MOSAICO architecture, the project has considered the possible split of management plane components into micro-services. Indeed, bringing such features to monitoring services could enable the agile deployment of the sole required components at some strategic location and allow them to be dynamically spawned or moved according to the evolution of the service performance. In this effort, this section depicts the early performance evaluation work which has been conducted on the monolithic monitoring solution of partner Montimage on OpenNetVM, the bottom-line micro-service solution.

### 2.4.1 Analysis of OpenNetVM solution

Networking Functions (NFs) enable a wide variety of applications to be implemented as in-network software functions that range from lightweight software firewalls, high performance proxy to complex Deep Packet Inspection (DPI) applications. To improve the flexibility and adaptability of the NFs, one technique is to introduce the composition of network functions based on micro-services at the control and data-planes. This is referred to as Service Function Chaining (SFC) to build more flexible and complex security services that involve the monitoring and enforcement, as depicted in Figure 2.27. For example, network slicing in the complex 5G network converging both mobile and fixed networks, could be achieved thanks to SFC when data processing will become a sequence of services managed by different stakeholders.

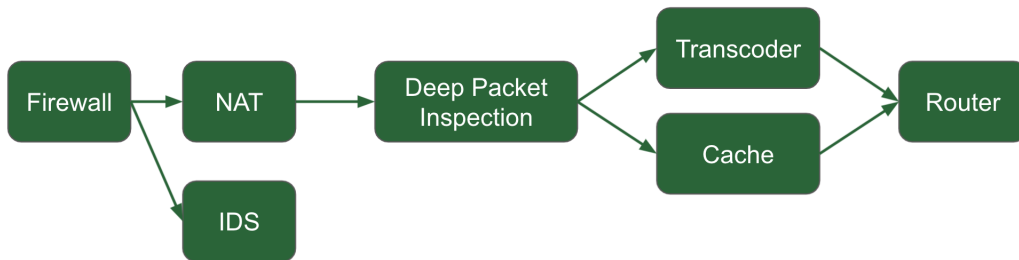


Figure 2.27: An example of architecture of micro-service chains

Furthermore, deploying a monitoring service that monitors the network traffic is crucial for detecting anomalies in NFV-based architectures, as the monitoring service allows to provide detailed metadata and security reports concerning the network traffic in real-time. One of the main challenges of deploying a chain of services is to find a ways to move efficiently transmit packets through a processing chain efficiently. In this perspective, the objective is to identify the security NFs that can be composed from micro-services, define multi-layer orchestration, and optimize micro-services' interactions.

### Overview of OpenNetVM Platform

We have investigated a potential solution – the OpenNetVM platform [27] for flexible and high performance micro-service chains. Overall this tool provides a virtualization-based high-speed packet delivery platform that obtains low resource overhead and low latency. The key idea is to eliminate the overhead of Operation System kernel packet processing, allowing zero-copy packet delivery between NFs, requiring no interrupts using DPDK poll-mode driver, and allowing dynamically managing and reconfiguring the NFs

using its NF Manager. It relies on a shared memory across NFs and NF Manager that eliminates overhead introduced when copying network packets or traversing network interfaces. It also allows defining security domains, i.e., security boundaries between trusted and untrusted NFs.

The NFs are run inside Docker containers and use the NF Lib library that allows each NF to interface with the NF Manager and other NFs. The shared memory handles huge pages for making the packets processing efficient. The RX threads directly receive the packets from Network Interface Cards (NICs) using DPDK, and the TX threads make the packet pointers available between NFs. Each NF is identified by its **ServiceID** (SID) and its **InstanceID** (IID), several NFs can share the same SID but the IIDs are unique. For a given SID, the manager will decide which instantiated NF will process the incoming packet, improving flexibility and fault tolerance. In case of a sudden freeze of an NF, the manager can simply route the packets to another one using the same SID.

### 2.4.2 Monitoring framework as micro-services

The goals of our development and analysis of the monitoring solution on the platform OpenNetVM g are twofold: 1) We develop our monitoring framework that consists of several main components as micro-services. We then evaluate the performance of our micro-services by considering the complexity-accuracy tradeoff of our techniques in processing high-throughput network traffic, 2) We develop some principal NFs, such as NF Firewall and NF Load Balancer, on the platform OpenNetVM and see how packets flow through a chain of those principal micro-services in a use case.

This section presents the results of our analysis, development and evaluation.

#### Overview of Montimage Monitoring Framework

Montimage provides a solution that monitors network or system traffic to detect behaviors, security, performance incidents based on a set of properties. The monitoring framework consists of 2 principal components: MMT-Probe and MMT-Operator, as shown in Figure 2.28. MMT-Probe captures packets in real time using advanced DPI techniques to extract metadata, verify security rules and send periodical reports to MMT-Operator, which is a Web application that presents metadata and reports via graphs.

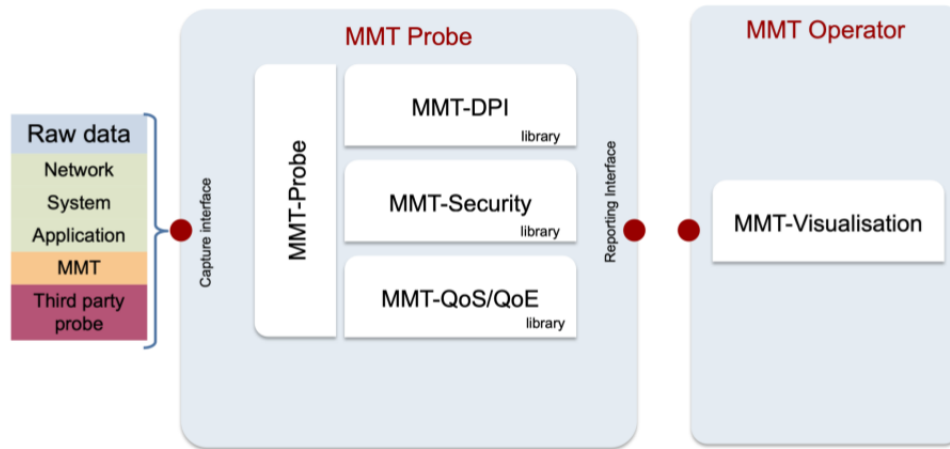
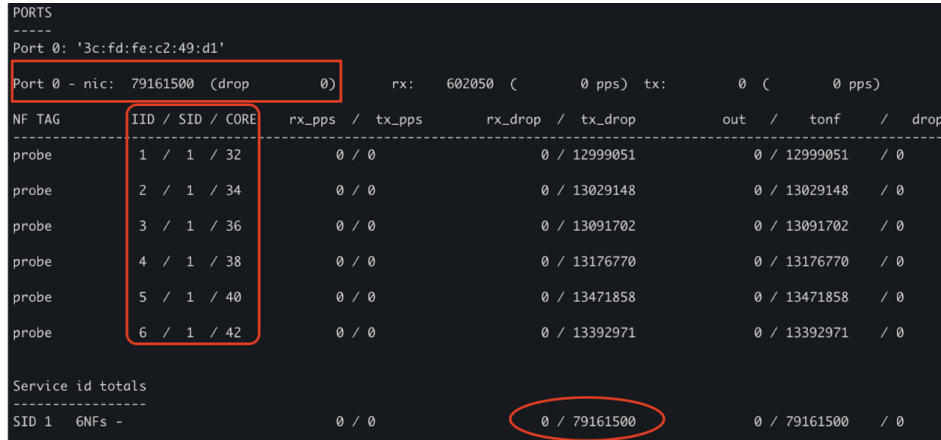


Figure 2.28: General architecture of Montimage Monitoring Framework

We could split the monitoring framework into 2 micro-services: the monitoring part and the visualization part. Obviously, those 2 micro-services are independent with different functionalities and implementations. Furthermore, the monitoring micro-service is designed in a modular way by including several libraries: (1) MMT-DPI for examining network traffic by employing complex DPI techniques; (2) MMT-Security for verifying Boolean security rules; and (3) MMT-QoS/QoE for measuring quality of service/experience. Therefore, we can easily plug or unplug a specific library depending on the monitoring context. We can also make the monitoring micro-service more lightweight by specifying a list of specific protocols to be analyzed according to the context of analysis goal.

## Running Parallel

To increase the throughput speed and avoid the packet loss problem, another possible solution is to develop a scaled version of complex services, such as the monitoring NF Probe, running on multiple cores on the platform OpenNetVM. Technically, the scaling NF will spawn multiple children NFs, and each child NF which has an unique IID runs on a specific core. For instance, as shown in Figure 2.29, we run 6 instances of our NF Probe with SID 1 on 6 different cores. OpenNetVM uses the Receive Side Scaling (RSS) hash value to provide flow consistency and support load balancing among children NFs with the same SID. All spawned NFs manage their own RX/TX ring buffers to receive and send packets to the destination NF in the service chain. Clearly, the total number of processed/dropped packets by our NF Probe is the sum of each instance's processed/dropped packets.



```

PORTS
-----
Port 0: '3c:fd:fe:c2:49:d1'
Port 0 - nic: 79161500 (drop 0) rx: 602050 ( 0 pps) tx: 0 ( 0 pps)
NF TAG IID / SID / CORE rx_pps / tx_pps rx_drop / tx_drop out / tonf / drop
-----
probe 1 / 1 / 32 0 / 0 0 / 12999051 0 / 12999051 / 0
probe 2 / 1 / 34 0 / 0 0 / 13029148 0 / 13029148 / 0
probe 3 / 1 / 36 0 / 0 0 / 13091702 0 / 13091702 / 0
probe 4 / 1 / 38 0 / 0 0 / 13176770 0 / 13176770 / 0
probe 5 / 1 / 40 0 / 0 0 / 13471858 0 / 13471858 / 0
probe 6 / 1 / 42 0 / 0 0 / 13392971 0 / 13392971 / 0

Service id totals
-----
SID 1 6NFs - 0 / 0 0 / 79161500 0 / 79161500 / 0

```

Figure 2.29: Scaled version of NF Probe

## Example of a Micro-service Chain

In order to create a service chain, we implemented on the platform OpenNetVM for our further evaluation:

- **NF Firewall** simply drops or forwards packets based on Long Prefix Match (LPM) rules specified in a json file. For instance, according the following IP-based rules, the NF Firewall will drop packets sent from source IP addresses “192.168.1.3” and “68.232.34.200”. Otherwise, the packets will be forwarded to the next NF in the chain.

```

{
  "rule1": {
    "ip": "192.168.1.3",
    "depth": 32,
    "action": 0
  },
  "rule2": {
    "ip": "68.232.34.200",
    "depth": 32,
    "action": 0
  }
  ...
}

```

- **NF Load Balancer** acts as a round-robin load balancer. When a packet arrives, this NF checks whether it is from an already existing flow. If not, it creates a new flow entry and assigns it to the destination server. Specifically, the NF Load Balancer decides what to do with the packet, such as forward it to the specified NF or NIC port, based on the predefined criteria as follows: (1) source IP address; (2) TCP/UDP packets; (3) HTTP/non-HTTP packets; etc.
- **NF Probe** and **NF Operator** are discussed in details above.

By composing different kinds of micro-services discussed above, we generate sequential chains on the platform OpenNetVM, as depicted in Figure 2.30. More specifically, there is a NF acting as a firewall to filter the packets that do not need to be processed by the security solution, and the disaggregation of the packets to optimize the processing by separating it into specialized processing and/or parallelising it. Concretely, if the traffic is HTTP, the NF Load Balancer will forward the packets to a server that runs NF Probe 1 on OpenNetVM. Otherwise, the NF Load Balancer will forward non-HTTP packets to another server.

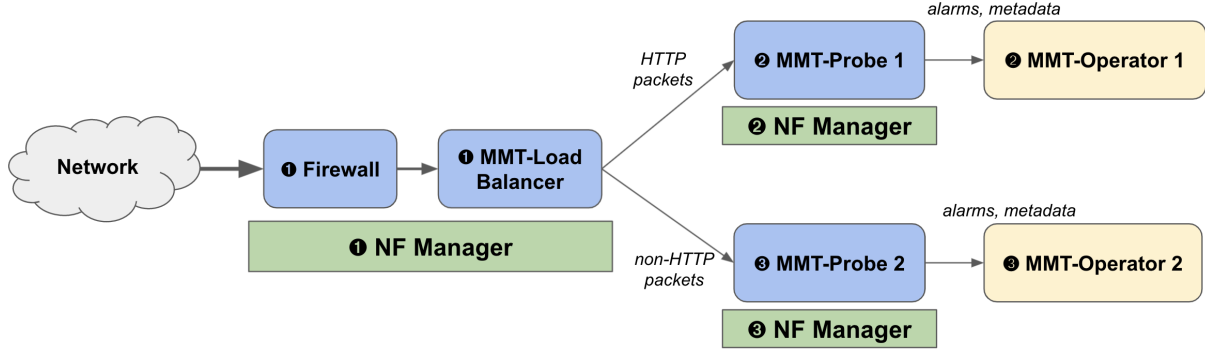


Figure 2.30: SFC example composing of different micro-services

### 2.4.3 Evaluation

In this section, we present our evaluation of the monitoring platform developed in micro-services running on OpenNetVM.

#### Evaluation Setup

As depicted in Figure 2.31, to evaluate the performance of the monitoring NF, our experiment setup consists of two servers that are connected to each other using a 40Gb Ethernet cable on two DPDK ports as follows:

- For traffic generation, we use an Intel server that runs our packet generator tool `mmt-dpdk-replay` using DPDK version 17.11.4. Basically, our tool replays pcap files in Tcpreplay Suite [4], as shown in Table 2.2, to create and send network traffic flow to another server. Another option is to use the tool Pktgen-DPDK, but it is easier to control the bandwidth of network traffic by using our tool.
- We use another Intel server to run our monitoring NF on openNetVM.

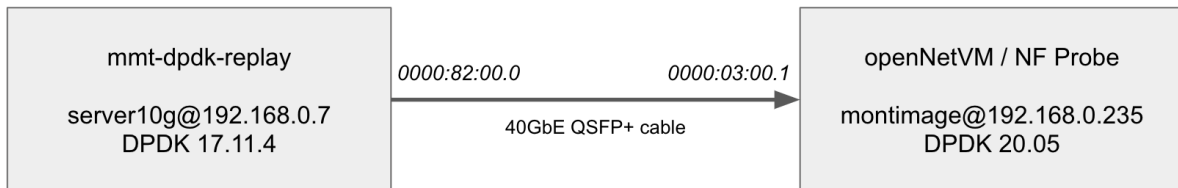


Figure 2.31: Architecture of evaluation setup

#### Evaluation Metrics

As discussed in [27] and confirmed by our preliminary evaluation, the performance difference for OpenNetVM when using processes or Docker containers is negligible. Therefore, in our experiments, we run NFs as processes instead of Docker containers. For simplicity, our NF Probe with SID 1 will forward packets to the next NF with SID 2 in the chain that does not exist. Thus, we count the number of packets processed by our NF Probe as the value `tx_drop`, because those packets can't be sent out. For instance,



Table 2.2: Pcap files in the Tcpreplay Suite

	SmallFlows.pcap	BigFlows.pcap
Size	9.4MB	368MB
Number of packets	14261	791651
Number of flows	1209	40686
Number of applications	28	132
Average of packet size	646 bytes	449 bytes

as shown in Figure 2.29, our NF Probe processes successfully all packets received at NIC (i.e., `tx_drop` = 79161500) without any dropped packets (i.e., `rx_drop` = 0). As OpenNetVM only provided information about number of received/sent packets of the NF Manager, we modified its source code [3] to print out those values at NIC. Indeed, it is important to identify where the packet loss happens, either at NIC or at the NF Manager of OpenNetVM. In case of packet loss, we calculate the packet loss ratio using the formula below:

$$Packet\_loss\_ratio = \frac{Number\ of\ dropped\ packets\ (at\ NIC) * 100}{Total\ number\ of\ packets\ sent} \quad (2.1)$$

To double-check the number of processed packets of our NF Probe, we verify the number of packets displayed on NF Operator interface. We also disable the flow table lookup that the NF Manager performs on every packet to make the NF Probe run faster.

## Discussions

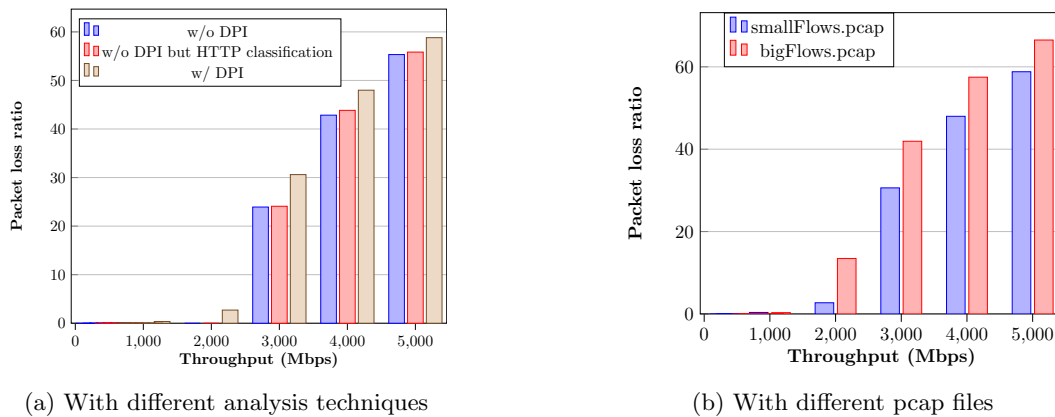


Figure 2.32: NF Probe performance on 1 core

We can split the workloads of our NF Probe into different levels. Figure 2.32a shows the performance of our NF Probe with different workloads on 1 core:

- *without DPI*: NF Probe only extracts metadata of each packet without applying advanced DPI techniques
- *without DPI but with HTTP classification*: NF Probe does not use DPI techniques, but rather classify HTTP traffic
- *with DPI*: NF Probe applies DPI techniques in order to extract more useful metadata and statistics of the network traffic

We can see the loss packet problem occurs at 2Gbps. This is because these packets are being dropped before the NF Manager of OpenNetVM receives them from the NIC as OpenNetVM's RX threads couldn't read the packets fast enough.

Figure 2.32b shows the performance of NF Probe when the packet generator tool continuously replays two different pcap files that are smallFlows.pcap and bigFlows.pcap with size 9.4 MB and 368 MB, respectively. Not surprisingly, the packet loss ratio of NF Probe is proportional to the complexity of pcap

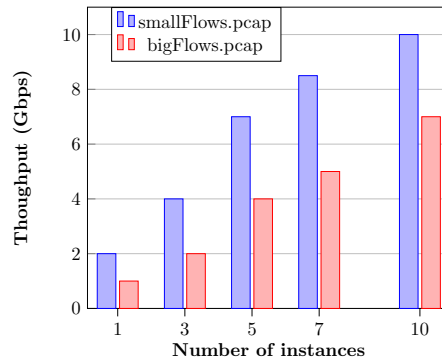


Figure 2.33: NF Probe achieves high throughput when running on multiple cores

files to be analyzed, as bigFlows.pcap is much bigger than smallFlows.pcap in terms of number of flows and applications.

As shown in Figure 2.33, NF Probe achieves high throughput when running multiple instances. The 7Gbps and 10Gbps rate can easily be met when running NF Probe on 10 cores, respectively. In the use case, the NF Probe could also achieve 16Gbps without packet loss problem, respectively.

## 2.5 Orchestration of Micro-Services

As a closely-coupled component on the management plane, the orchestration of micro-services, more precisely designed by routing and placement algorithms, completes the monitoring part addressed in the previous section. Following, we present two main contributions in that regards. The first one is an optimization model grounded by the features required by the envisioned MOSAICO architecture. The model has been implemented in the CPLEX solver and its exact resolution process has been benchmarked under several operational conditions. Then, we explore to what extent, GPU can be leveraged to offer a substantial computing resource for micro-services and especially, under these circumstances, how such a setup can be orchestrated with state-of-the-art technologies.

### 2.5.1 Definition and evaluation of a model for orchestrating micro services (Confidential for now)

The evolution of the Internet tends toward ever requiring lower latency services. Cloud robotics or drone piloting are service use-cases in which the latency of traffic cannot exceed a few milliseconds. Reducing the latency can be achieved through several means, and micro-services deployed over virtual infrastructures appears as a promising way by enabling service chain reductions, micro-function mutualization and parallelism. However, the placement and routing of such components appears as an harder task to achieve as compared to monolithic approaches. Consequently, we propose in this section a comprehensive optimization model in charge of placing micro-services in a virtualized network infrastructure, under ultra-low latency constraints while preserving resource consumption. By challenging our model with several realistic scenarios in terms of topology and service function chains (SFC), we demonstrate to what extent it improves the overall performance of SFC by especially minimizing the gap between the expected latency and the actual one, as compared to several competitors, thus making it a well-fitted approach for ultra-low latency services.

#### A micro-service orchestration model

In this section, we present the micro-services placement and chaining orchestration problem and then, we develop its mathematical formulation. Wishing to optimise the SFC latency and benefit from the agility offered by micro-services, we propose an approach allowing to: (1) mutualize micro-services through a pre-processing algorithm; (2) place and chain micro-services through a MILP that also (3) efficiently manages internal and external parallelization of micro-services according to the network configuration (number of nodes and links as well as their available resources).

**Problem Statement Definition:** The micro-services placement and routing problem we study, is defined on a network graph  $G = (N, L)$ , where  $N$  is a set of nodes and  $L$  a set of links between nodes.  $Q$  is a set of SFC requests, with each request  $q$  in  $Q$  being characterized by a source  $S_q$ , a destination  $D_q$ , a nominal bandwidth  $B_q$  statistically representative for request  $q$ , a maximum execution latency  $\Lambda_q$  and a set of micro-services of different types to be traversed by an edge flow. The micro-services placement and chaining optimization problem consists in finding:

- The placement of micro-services over network nodes;
- The assignment of requests to micro-services already placed;
- The chaining for each request,

subject to:

- Memory node capacity constraints;
- Micro-services forwarding and execution latency constraints;
- Parallelism execution constraints.

The optimization objective chosen in our work is the minimization of the latency delay on service requests according to their latency specification. Indeed, we consider that:

- The available nodes have no usage cost but the available resources expressed in terms of CPU and memory are limited;
- The available links have no usage cost but the flow rate on each one is limited;
- The memory required for the deployment of micro-services is similar for all micro-services;
- Sharing the usage of micro-services between different SFC is not allowed.

**Overall Approach** Our approach is developed with the aim of taking advantage of the micro-service features in order to reduce the end-to-end latency of SFC. It consists in two parts: (1) a pre-processing algorithm that performs the mutualization, a pre-parallelization on the set of SFC, and provides the  $Q$  set and parallelism parameters to be used in our model, and (2) a Mixed Integer Linear Programming (MILP) translating the problem of placement and chaining of micro-services, which is solved by taking into consideration the parallelism parameters among other constraints.

**SFC Pre-processing** The pre-processing phase needs two-dimensional parallelism and mutualization tables as inputs, which indicate whether two micro-services are mutualizable or parallelizable as developed in the literature [18] [26]. To understand the pre-processing, let us take the example depicted in Figure 2.34, where a request  $q$  is made up of an original SFC containing 5 sequential micro-services. Supposing that the tables indicate that micro-services 1 and 5 can be mutualized and micro-services 2, 3 and 4 can be parallelized, our algorithm builds a new SFC made up of 4 micro-services as pictured in the second part of Figure 2.34. Note that depending on the infrastructure constraints, the MILP of the second phase may not necessarily lead to the placement and execution of micro-services 2, 3 and 4 in parallel, even if it is it has been allowed in the pre-processing phase.

**Managing Parallelized Micro-services** The pre-processed SFC are provided to the model with some parallelism parameters to carry out a placement and chaining. The aim here is to minimize the sum of the gaps between the required and achieved latency after deployment, for each SFC while taking into account the network configuration. The key-point stands in the choice of service parallelization, that is managed by the model through the generation of forks and mergers of two types:

- internal forking: a packet is duplicated within a node to reach a set of further parallelized micro-services located on the same node. This kind of internal parallelism does not require copying data, nor merging, since, as technologies such as DPDK [13] allow shared memory to be used.
- external forking: a packet outgoing from one node is duplicated to subsequent NF located on two or more successor nodes. Such external parallelism implies a copy of the packets to be sent to the different micro-services deployed on different nodes and then a merging of the resulting packets too.

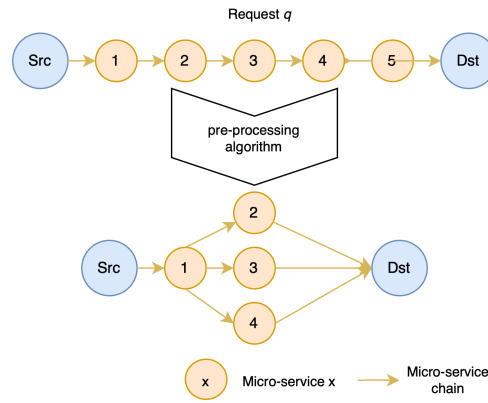


Figure 2.34: Pre-processing example

This latter mechanism leads us to add a latency cost for the external fork corresponding to the copy and merge time, unlike the internal fork. The originality of the model lies then in the fact that depending on its configuration, it will decide which micro-services to parallelize internally, externally or not at all. However, this complicates the computation of the induced latency. Indeed, when the model defines a placement, it also has to indicate whether the micro-services on the nodes (if parallelizable) are visited consecutively or not. In order to know which micro-services are running in parallel within a node, we use groups that gather them together.

To illustrate this principle, the example in Figure 2.35 shows 3 different possible deployments for request  $q$ , as introduced in Figure 2.34. The first one (A) does not parallelize any micro-services: although two are on the same node (i.e. micro-services 2 and 3), they do not belong to the same group. In this case, the resulting SFC is composed of a single group ( $g_1$ ) on nodes  $B$  and  $D$ , and two groups ( $g_1$  and  $g_2$ ) on node  $E$ . The second deployment (B) proposes an external parallelization of micro-services 2 and 3 on two different nodes, thus a single group ( $g_1$ ) per node. This induces an additional latency corresponding to the copying and merging operations. Finally, deployment (C) parallelizes micro-services 2, 3 and 4 by performing internal and external parallelism. This time, a single group ( $g_1$ ) is assigned to each of nodes  $B$ ,  $C$  and  $D$ . In this case also a fork cost is accounted because the latter is external. Beyond, this SFC example and underlying infrastructure may lead to other possibilities, actually depending on the memory capacities of each node.

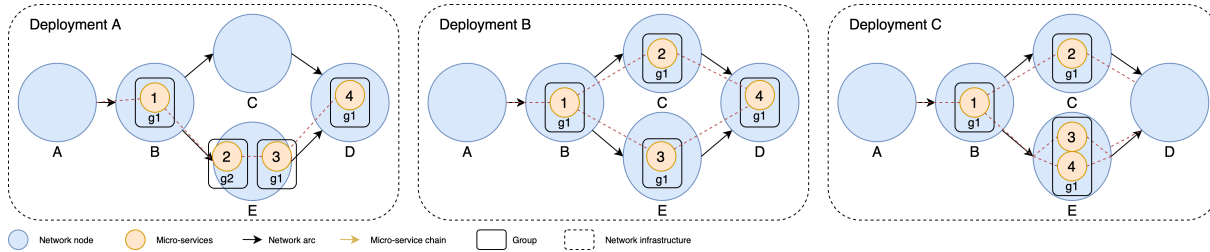


Figure 2.35: A deployment example

**Mathematical Formulation** Table 2.3 provides the set of notations used for the formulation of our MILP formalizing the problem of placement and chaining of micro-services.

In this model, we define seven decision variables.  $x_{nfq}$  are used to decide which micro-service  $f \in F$  of request  $q \in Q$  should be placed on which node  $n \in N$ .  $y_{nfq}$  allow to know whether micro-service  $f \in F$  is placed on the nodes preceding  $n \in N$  ( $n$  included) on the path related to request  $q \in Q$ . Variables  $a_{n_1n_2q}$  are used to decide whether the link between nodes  $n_1$  and  $n_2$  from set  $N$  is used for service request  $q \in Q$ , all links being oriented in this model. Having allowed the forks, we introduce variables  $l_{n_1n_2qh}$  to decide whether the link between nodes  $n_1$  and  $n_2$  from set  $N$  is used within part  $ch \in F_q$  of the chaining of request  $q \in Q$ . This allows to define the set of possible paths for the chaining of each SFC. Variables  $b_{f_nqh}$  indicate the existence of a fork on node  $n \in N$  relative to a request  $q \in Q$  and path  $h \in F_q$ . Then, the latency of request  $q \in Q$  finishing its process at node  $n \in N$  is equal to the greatest latency of the different relative paths. This requires to define a longest path, managed the notion of group as introduced

Sets	
$N$	Nodes
$L$ in $N \times N$	Links
$Q$	Service requests
$F$	Micro-services
Micro-service parameters	
$lex_f$	Execution latency
$Rm_f$	CPU resource requirements
Demand parameters	
$S_q$	Source
$D_q$	Destination
$\Lambda_q$	Latency
$B_q$	Throughput
$F_q$ in $Q$	Micro-services composition
Infrastructure parameters	
$M_n$	Node memory
$R_n$	Node CPU resource
$LatA_{n_1n_2}$	Link latency
$Da_{n_1n_2}$	Link flow rate
$CB$	Bifurcation cost
Parallelism parameters	
$T_{f_1f_2}$	Parallelism possibility
$P_{f_1f_2q}$	Parallelism possibility into service request
Model parameters	
$m$	Constant value guaranteeing compliance with constraints
Binary variables	
$x_{nfq}$	= 1 if function $f$ is placed on node $n$ for request $q$
$y_{nfq}$	= 1 if function $f$ is placed on node $n$ or before for request $q$
$a_{n_1n_2q}$	= 1 if link $(n_1, n_2)$ is activated for request $q$
$l_{n_1n_2qh}$	= 1 if link $(n_1, n_2)$ is activated for request $q$ for path $h$
$\gamma_{nqgf}$	= 1 if function $f$ related to request $q$ placed on node $n$ belongs to group $g$
$bf_{nqh}$	= 1 if there is a fork on node $n$ for request $q$ and for path $h$
$p_{nfq}$	Intermediate variable
Continuous non-negative variables	
$r_q$	Gaps between the required and achieved latency for request $q$ after deployment
$lgg_{nqgh}$	Latency of group $g$ related to request $q$ , node $n$ and path $h$
$o_{nq}$	Order of node $n$ in the chaining of request $q$

Table 2.3: Notation table

in section 2.5.1. A group  $g \in F_q$  is composed of a set of a single or several parallelized micro-services assigned to node  $n \in N$  for request  $q \in Q$  and for path  $ch \in F_q$ . Since all the micro-services belonging to the same group run in parallel, we use variable  $\gamma_{nqgf}$  to indicate whether micro-service  $f \in F$  related to request  $q \in Q$  is placed in group  $g \in F_q$  attached to node  $n \in N$ , and variable  $lgg_{nqgh}$  which corresponds to the latency of group  $g \in F_q$  relative to request  $q \in Q$  and path  $h \in F_q$ . Finally, in order to minimize the overall gap between the required and actual latency we introduce variable  $r_q$  which represents the gap latency for each request  $q \in Q$ .

This leads to the following model, consisting in minimizing an objective function relative to the lag behind expected latency and subject to the two dozen of constraints that we describe subsequently.

The objective function (2.2) minimizes the latency delay,  $r_q$  of request  $q$ . The second term is relative to additional variables used in the constraints.

$$\min \sum_{q \in Q} r_q + \sum_{n \in N} \sum_{q \in Q} \sum_{f \in F_q} \sum_{h \in F_q} y_{nfq} + bf_{nqh} + p_{nfq} \quad (2.2)$$

Constraints (2.3) and (2.4) guarantee that if there is an incoming flow in a node for a certain service request  $q$ , and the node is neither a source nor a destination node, an outgoing flow must exist and

vice-versa. Unlike a standard chaining problem, outgoing flows can be greater or lower than incoming flows and vice-versa, due to the fork-merge managed by the model.

$$\sum_{e \in N} a_{enq} \geq a_{nsq} \quad \forall q \in Q, n, s \in N \neq S_q, D_q \quad (2.3)$$

$$\sum_{s \in N} a_{nsq} \geq a_{enq} \quad \forall q \in Q, n, e \in N \neq S_q, D_q \quad (2.4)$$

In order to avoid cycles in the chain and to force it to be elementary, constraint (2.5) ensures that each node has an order in the chain and that this order is respected.

$$o_{n_1q} \geq o_{n_2q} + a_{n_2n_1q} - m(1 - a_{n_2n_1q}) \quad \forall n_1, n_2 \in N \quad (2.5)$$

The set of constraints (2.6) to (2.11) allow to set variable  $y_{nfq}$  to 1 in case function  $f$  is placed on node  $n$  or before for request  $q$ .

$$y_{nfq} \geq x_{nfq} \quad \forall n \in N, f \in F, q \in Q \quad (2.6)$$

$$a_{n_1n_2q} - 1 + x_{n_1fq} - x_{n_2fq} \leq y_{n_2fq} \quad \forall n_1, n_2 \in N, \forall q \in Q, \forall f \in F_q \quad (2.7)$$

$$y_{n_1fq} - 1 + a_{n_1n_2q} \leq y_{n_2fq} \quad \forall n_1, n_2 \in N, \forall q \in Q, \forall f \in F_q \quad (2.8)$$

$$\sum_{n_1 \in N} y_{n_1fq} + a_{n_1nq} \leq p_{nfq} \quad \forall n \in N, \forall q \in Q, \forall f \in F_q \quad (2.9)$$

$$x_{nfq} \leq p_{nfq} \quad \forall n \in N, \forall q \in Q, \forall f \in F_q \quad (2.10)$$

$$y_{nfq} \leq p_{nfq} \quad \forall n \in N, \forall q \in Q, \forall f \in F_q \quad (2.11)$$

More specifically, constraint (2.6) allows to set variable  $y_{nfq}$  to 1 if  $x_{nfq}$  is equal to 1. Constraint (2.7) aims at setting variable  $y_{n_2fq}$  to 1 if the relative function  $f$  is placed on node  $n_1$ , preceding node  $n_2$  and concerning request  $q$ . Constraint (2.8) allows to set variable  $y_{n_2fq}$  to 1 if  $y_{n_1fq}$  equals to 1 and if  $n_1$  is preceding node  $n_2$  concerning request  $q$ . The set of constraints (2.9), (2.10) and (2.11) allow to set variable  $y_{nfq}$  to 0 if function  $f$  is not placed on node  $n$  and it is not placed on any of the preceding nodes of  $n$  as well as their respective preceding nodes.

Constraint (2.12) ensures that all micro-services related to each request  $q$  are placed on the destination node or before on the chain related to  $q$ .

$$y_{nfq} \geq y_{D_qfq} \quad \forall q \in Q, \forall f \in F_q, \forall n \in N \quad (2.12)$$

Constraints (2.13) to (2.15) guarantee that if two micro-services are placed in the same group, they can be processed in parallel, and this in two ways: the first states that the two micro-services can be technically parallelizable, the second states that, in the SFC, the two micro-services are parallelizable, i.e. none of them is in the precedence list of the other.

$$\gamma_{nqgf_1} + \gamma_{nqgf_2} \leq T_{f_1f_2} + 1 \quad \forall n \in N, \forall q \in Q, \forall g, f_1, f_2 \in F_q \quad (2.13)$$

$$\gamma_{nqgf_1} + \gamma_{nqgf_2} \leq P_{f_1f_2q} + 1 \quad \forall n \in N, \forall q \in Q, \forall g, f_1, f_2 \in F_q \quad (2.14)$$

$$l_{gqgh} \geq lex_f(\gamma_{nqgf} + l_{nn_2qh} - 1) \quad \forall q \in Q, f \in F_q, \forall n, n_2 \in N, \forall h, g \in F_q \quad (2.15)$$

Constraint (2.15) ensures that variable  $l_{gqgh}$  is greater than or equal to the execution latency of all the micro-services belonging to group  $g$  on path  $h$ .

Constraint (2.16) allows variable  $r_q$  to be equal to the number of SFC which do not respect their latency specification.

$$\sum_{n_1 \in N} \sum_{n_2 \in N} l_{n_1n_2qh} * lat_{A_{n_1n_2}} + \sum_{n \in N} \sum_{g \in F_q} l_{gqgh} + \sum_{n \in N} b_{fn} * CB - \Lambda_q \leq r_q \quad \forall q \in Q, \forall h \in F_q \quad (2.16)$$



Constraints (2.17) and (2.18) respectively ensure that the CPU resources and the available flow rate on each link are respected for each node.

$$\sum_{q \in Q} \sum_{f \in F_q} x_{nfq} * Rm_f \leq R_n \quad \forall n \in N \quad (2.17)$$

$$\sum_{q \in Q} \sum_{f \in F_q} a_{n_1 n_2 q} * Db_q \leq Da_{n_1 n_2} \quad \forall n_1, n_2 \in N \quad (2.18)$$

Finally, the model includes the following types of variables.

$$x_{nfq} \in \{0, 1\} \forall n \in N, f \in F, q \in Q \quad (2.19)$$

$$y_{nfq} \in \{0, 1\} \forall n \in N, f \in F, q \in Q \quad (2.20)$$

$$a_{n_1 n_2 q} \in \{0, 1\} \forall n_1, n_2 \in N, q \in Q \quad (2.21)$$

$$l_{n_1 n_2 q h} \in \{0, 1\} \forall n_1, n_2 \in N, q \in Q, h \in F_q \quad (2.22)$$

$$\gamma_{nqfg} \in \{0, 1\} \forall n \in N, q \in Q, f \in F, g \in F_q \quad (2.23)$$

$$bf_{nqh} \in \{0, 1\} \forall n \in N, q \in Q, h \in F_q \quad (2.24)$$

$$p_{nfq} \in \{0, 1\} \forall n \in N, q \in Q, f \in F \quad (2.25)$$

$$r_q \geq 0 \forall q \in Q, l_{ggnqgh} \geq 0 \forall n \in N, q \in Q, g \in F_q, h \in F_q \quad (2.26)$$

$$o_{nq} \geq 0 \forall n \in N, q \in Q \quad (2.27)$$

Besides, the model also includes some additional constraints, not detailed here due to space constraints. Briefly, they ensure that: (a) the placement of all the micro-services related to each request is achieved; (b) the chaining passes through the necessary micro-services; (c) the memory capacity of each node is respected; (d) each chaining starts with the source node and ends with the destination node related to each request; (e) variable  $bf_{nqh}$  is activated when an external fork takes place on node  $n$  for request  $q$  and on the path  $h$ ; (f) the links related to each path  $ch$  are only active if they are active in the request chain; (g) the flows for each path are respected; (h) each path related to a request  $q$  passes through at least one micro-services related to its request; (i) each deployed micro-services belongs to a group. One can lastly notice that the constraints of nodes ordering respect and micro-services ordering respect were inspired by the work carried out in [5, 19].

## Evaluation

We present hereafter the evaluation of our model, using the CPLEX solver, evaluated with various conditions.

**Implementation** In order to validate the correctness of our model and eventually its performance under various conditions, we have implemented it into CPLEX v20.1.0 using the C++ environment. The code is 1135 lines long. The correctness checking has consisted in verifying the following bias: (1) absence of circuits, (2) deployment of all micro-services related to each request, (3) respect of the ordering between micro-services, (4) respect of memory resource consumption, (5) correctness of latency computation, and finally (6) correctness of parallelism and mutualization in relation to the mutualization and parallelism tables. All experiments were performed on a machine hosting a 11th gen. Intel(r) Core(tm) i7-1165g7@2.80GHz 1.69GHz processors and 16GB of RAM, using Windows 10 Professional Education as the underlying operating system.

**Evaluation Scenarios** Our performance evaluation scenarios aims at (1) understanding the performance of our model in realistic situations and (2) comparing it with current competitors. We have especially considered those standing for acknowledged approaches in the literature, namely: (a) Monolithic VNF placement (Mono); (b) Micro-services placement with neither mutualization nor parallelization enhancements (Micro); (c) Micro-services placement with mutualization enhancement (MicroM), and finally (d) Micro-services placement with both mutualization and parallelization enhancements (MicroMP); the latter standing for our model.

Parameter	Range or value
Topology	DFN-Verein European Telco
VNF	Firewall, NAT, Traffic monitor, IPS
Micro-services	Read (Rd), Header Classifier (HC), Modifier (Md), Alert (Al), Drop (Dp), Check IP Header (CIH), HTTP Classifier (HC), Count URL (CU), Payload Classifier (PC), Output (Out)
SFC latency	5-10ms according to the SFC
Link latency	1ms
Proc. latency	1ms
#SFC	1-3
SFC length	5-14 micro-services
#Nodes	4-10
Node capacity	3-10 instances of micro-services

Table 2.4: Evaluation parameters

	Rd	HC	Md	Al	Dp	CIH	HC	CU	PC	Out
Rd	x									
HC		x		x	x	x		x		
Md			x	x	x	x		x		
Al				x	x	x		x		
Dp				x	x	x		x		
CIH				x	x	x		x		
HC							x			
CU				x	x	x		x		
PC									x	
Out										x

	Rd	HC	Md	Al	Dp	CIH	HC	CU	PC	Out
Rd	x									x
HC	x	x	x	x	x	x	x	x	x	x
Md	x			x	x	x	x	x	x	x
Al	x	x		x	x	x	x	x	x	x
Dp	x			x	x	x	x	x		x
CIH				x	x	x		x		
HC	x	x		x		x	x	x		x
CU	x		x	x	x	x	x	x	x	x
PC	x			x	x	x	x	x		x
Out	x									x

Table 2.5: Parallelism (left) and mutualization (right) tables

All the parameters we considered in our evaluation are summarized in Table 2.4 and motivated subsequently. The implemented topology, extracted from the SNDlib<sup>7</sup> library, is that of the DFN-Verein European telco. We have partitioned it by selecting only some Point of Presence (PoP) for a given region. Then, each region has been split into two layers: one node acting as a regional PoP connected to other regional PoPs according to the telco topology, but also acting as an aggregation point for a few local nodes connected to it through a regional loop forming a ring sub-topology. The different SFC we consider are the reflect of those that can be found in the dedicated literature [19, 17] (Firewall, NAT, Traffic monitor and IPS), each of them being splittable into micro-services. As the core benefits of micro-services, we have considered the mutualization and parallelization tables illustrated in Table 2.5, where a "X" in a cell means that the related row and column micro-services can be mutualized or parallelized, respectively. Finally, regarding the varying parameters of our experiments, we have deliberately chosen to limit the computation time, which may be very long in case of exact-resolution, to a realistic order of magnitude to respect what the usage of such a placement algorithm in a real maintenance process of a virtual telco infrastructure could be. As such, the computation time of each experiment has been limited to 500s and, given this limit, the computable instances of placement covers the scales provided in Table 2.4 for the number of nodes, SFC numbers, SFC lengths and nodes capacities. Overall, the results exposed subsequently required more than 30 hours of computation to encompass all cases. Finally, the prescribed latency for SFC ranges from 5 to 10ms, thus standing for those of realistic ultra-low latency use-cases.

**Results Analysis** The different metrics we measure in our performance evaluation campaign aims at first considering the overall latency of service chains as a prerequisite and then the resource consumption to instantiate SFC over the telco infrastructure. More precisely, this stands for (1) the sum of the differences between the latency required by each SFC and the effective latency after deployment, as depicted in first line of Figure 2.36; (2) the number of nodes activated for the deployment of all the SFC, an activated node being a node on which at least one micro-service is instantiated, as depicted in second line of Figure 2.36; and (3) the number of links activated for the deployment of all the SFC, as depicted

<sup>7</sup>Survivable fixed telecommunication Network Design – <http://sndlib.zib.de/>

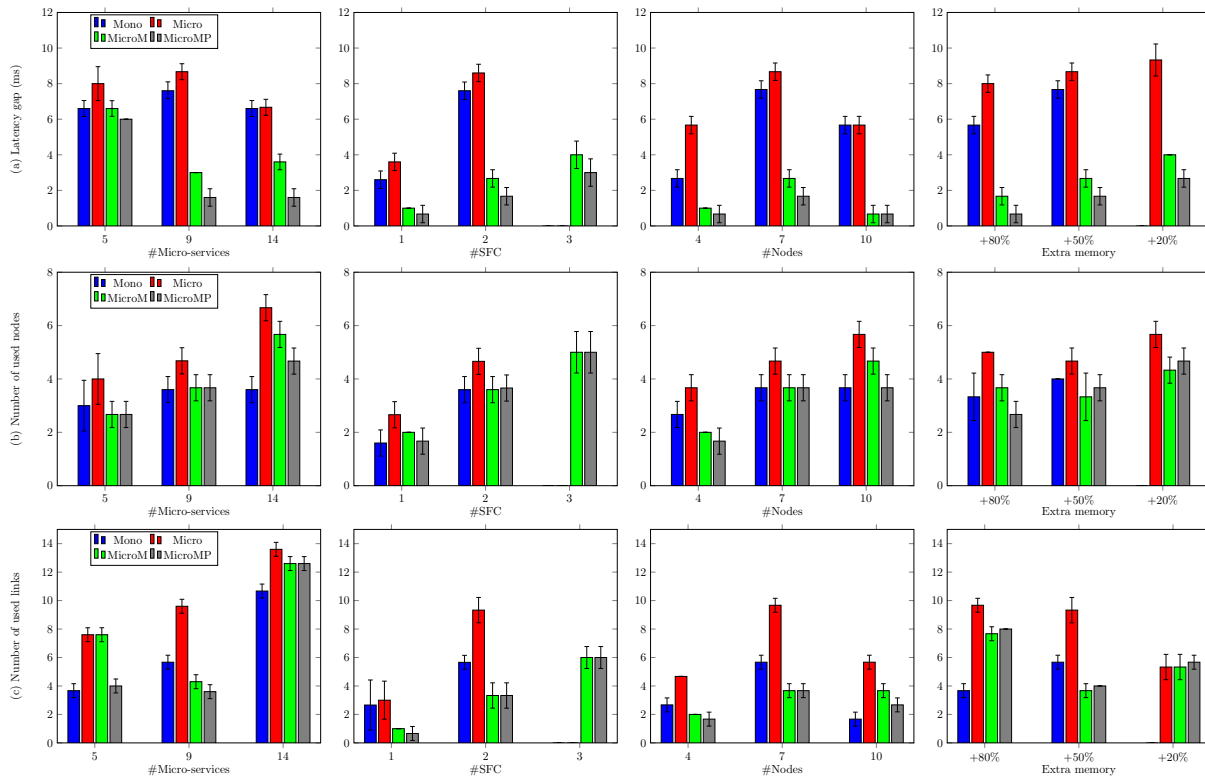


Figure 2.36: Histogram of the different measurements achieved to feature the performance of our model (MicroMP) against three competitors (Mono, Micro and MicroM). (a) Gap of latency between the SFC requirement and that computed by the placement algorithm; (b) Number of nodes required to place the requested SFC; and (c) Number of links required to place the requested SFC, according to (from left to right sub-figures): The number of micro-services, number of SFC, number of nodes and the extra allocated memory resources. Each result is the average of three repetitions bounded with 90% confidence intervals.

in third line of Figure 2.36.

**Latency Benefit** The first line of Figure 2.36 shows that MicroMP performs better in terms of latency gap in the different configurations under test. Moreover, this latency gain does not require a higher consumption of resources. We also notice that the more the system increases in size and load, the greater the benefit of MicroMP is. The reason is that the more micro-services there are, the more chances of parallelism and mutualization there are, these two aspects allowing the actual latency reduction. One can also see that Micro is the approach with the largest latency gap for all configurations. This is due to the basic decomposition into micro-services which generates a latency overhead since there are more entities to deploy. This confirms us that the decomposition into micro-services is relevant in terms of latency gain only when using mutualization and parallelism enhancements.

**Usage of Infrastructure Resources** The second and third lines of Figure 2.36 represent the nodes and links usage of the four approaches according to the different configurations. One can see that the usage of resources is more important when switching to the micro-services approach and this for all the configurations. The reason is again that with the micro-services approach, more entities have to be deployed, A VNF being composed of 4 micro-services on average [18]. Nevertheless, with the mutualization, we manage to reduce the consumption of resources by reducing the number of micro-services to be deployed, which brings us back to a consumption equivalent to Mono. Moreover, parallelism forces the parallelized micro-services to be deployed on the same node, thus bringing a slight gain for MicroMP as compared to MicroM and which, in some cases, even exceeds Mono.

**Deployment Agility** The third aspect important to notice deals with the deployment agility. Indeed, one can see that for certain instances and under certain configurations (e.g. #SFC = 3, node capacity +20%), the model cannot not find a solution for Mono and Micro, despite a sufficient

	Mono			Micro			MicroM			MicroMP		
I	Obj	Dur	Gap	Obj	Dur	Gap	Obj	Dur	Gap	Obj	Dur	Gap
1	7	2	0	7	46	0	7	45	0	6	22	0
2	8	2	0	9	500	47	3	500	7	2	157	0
3	7	2	0	9	500	47	4	500	28	2	500	16
4	3	1	0	4	500	21	0	13	0	0	12	0
5	8	2	0	5	500	47	3	500	7	2	312	0
6	-	-	-	-	-	-	4	500	27	3	500	23
7	3	1	0	6	500	37	0	17	0	0	17	0
8	8	1	0	9	500	47	3	500	7	2	472	0
9	6	1	0	6	500	37	0	4	0	0	4	0
10	6	2	0	8	500	44	2	334	0	1	177	0
11	8	1	0	9	500	47	3	500	7	2	500	8
12	-	-	-	10	500	49	4	500	28	3	500	7

Table 2.6: Features of the different model computations, with I: Instance, Obj (ms): Objective function, Dur (ms): Computation duration, Gap (%): Latency gap.

amount of infrastructure resources for a deployment. Indeed, under MicroM and MicroMP, the length of SFC is shortened and in terms of memory resources, an SFC under the MicroM and MicroMP approach thus takes up less space and is consequently easier to deploy. In addition, the micro-service breakdown allows a better agility as compared to Mono because the components are lighter and the management of memory space is easier to achieve.

**Analysis of Computation Features** Table 2.6 summarises the average computation features of CPLEX for the different instances. Each instance represents a configuration setup with the values presented in Table 2.4, and it has been repeated three times with different SFC to bring some randomness. Regarding the computation time, limited to 500s, we notice that for the Mono approach, the optimal solutions were found much earlier (1,5 seconds on average) contrary to the Micro approach for which the resolution time is over our limit for almost all instances. In a similar way, the gap with respect to the optimal solution of the Mono approach is zero while that of Micro is around 45%. This indicates that the solutions obtained for the Micro approach can be improved if we allowed more computation time. Nevertheless, with such a limited computation time, the Micro approach is penalized. By contrast, regarding the MicroM and MicroMP approaches, the computation time is below the limit for more than 55% of the cases. In comparison to the Mono approach, this reveals two insights: (1) with MicroM and MicroMP the obtained solutions are not necessarily optimal and, in spite of that, they are better than the optimal solutions of Mono. This means that with more computing time we could have still better solutions for the MicroM and MicroMP approach; (2) the switch to MicroM and MicroMP requires more computing time due to the complexity of the approach, but less than the Micro approach since there are fewer micro-services to deploy, thus demonstrating the reasonable cost of these approaches.

**Conclusion** The development of ultra-low latency applications has required the development of different models optimizing the placement and chaining of VNF, but also the transition to the micro-services approach with the aim of reducing the latency even more. However, considering the micro-services approach without optimization controversially augments the latency and the mutualization of redundant micro-services and their parallelization appears as the expected enhancement to reveal the actual performance gain of micro-services. In this section, we have proposed a micro-services orchestration approach composed of a pre-processing algorithm and a comprehensive mathematical model allowing the placement and chaining of micro-services taking into account their mutualization and parallelism and having as a main objective the reduction of the SFC latency to reach strong prerequisites. With extensive computation of the model, we have demonstrated that it exhibits the best latency performance as compared to monolithic approaches and also various micro-services alternatives. We have also highlighted its respect of infrastructure resources in terms of computation and link usage that do not exhibit any overhead.

## 2.5.2 Orchestration of micro-services running on GPU

The proliferation of programmable SDN and NFV pipelines in networking has introduced commodity hardware such as GPU to parallel packet processing. However, several limitations currently hinder the use of GPU for networking tasks. First, there is a lack of NFV-compliant architecture to deploy GPU-accelerated NFs in a cross-platform manner. Second, while GPUs are well-suited for parallel applications because of multiple cores, the devices have high latency when interfacing with CPUs. Recent studies have addressed this performance issue by using, e.g., integrated GPUs and optimization methods in work scheduling using continuous threads and multi-buffering. Related work has proven that GPUs can speed up many packet processing applications. However, solutions tend to be specific and reliant on proprietary API or frameworks that limit their adoption as orchestrated functions.

This work proposes an architecture based on the Vulkan API, which allows fine-grained control over GPU resources, but in a interoperable way. With Vulkan, it is possible to test different GPU-accelerated approaches for packet processing but keep compatibility over any setup capable of using Vulkan. Notably, Vulkan compatible setups include integrated and discrete AMD and Nvidia cards and host platforms on ARM and x86. The originality of this work is double. To our knowledge, we are the first to leverage the Vulkan API and SPIR-V (Standard Portable Intermediate Representation) with Kubernetes orchestration as a common ground to deploy GPU-accelerated network functions. We also open-source all the software components constituting our architecture to the community<sup>8</sup>.

The API we use is quite new. Indeed, the open GPU standards working group called Khronos, which is the organization behind GLSL and OpenCL, has released a new cross-platform graphics and compute API focused on performance, called the Vulkan API. Vulkan's new capabilities include a cross-platform and formally verified memory model called the Vulkan memory model and a class of cross-platform SIMD operands for non-uniform group operations called subgroups. Still, while released in 2016, the Vulkan API has not seemingly yet become the leading API in the GPGPU space, plausibly affected by a design decision to only support a new open standard IR called SPIR-V. Thus, any application which wishes to adopt Vulkan's features would first need to update all the GPU code to SPIR-V. In practice, this is done via cross-compilers, but translations may not necessarily produce the most performant code nor be able to use the latest features. E.g., the Vulkan memory model was released as recently as September 2019 and adds new visibility semantics for variables use, paramount for deterministic computation results, but insofar languages that expose selection over these visibility levels seem non-existent. As such, evaluation of new Vulkan features tends to mean writing SPIR-V by hand, which serves as partial motivation for this study regarding to performance requirements.

### Overview of the architecture

Our key contribution to the system framework is to leverage the Vulkan API and the latest SPIR-V versions. These technologies reduce the interoperability problems of past GPGPU approaches because Vulkan, as an open-source standard, facilitates integration. Regardless of the platform of choice, each device and architecture understands the same IR code in SPIR-V. In §2.5.2, we notice that SPIR-V is a valid IR for another reason: the produced binary files are only a few kilobytes in size and can hence be inlined within configuration text files. We use this exciting property to propose an efficient way to orchestrate GPU NF on Kubernetes using Vulkan.

Our approach to writing the GPU implementation (in SPIR-V) of network functions is not direct. We preferred to use another language called APL (A Programming Language) as a modeling language first. So, we first write the implementation in APL and derive the SPIR-V code from APL. This intermediate step is not mandatory but helps highlighting sequential part of the code. Fig. 2.37 depicts our software architecture per a GPU node, and Fig. 2.38 shows how it situates and integrates within a Kubernetes cluster. To elaborate, in Fig. 2.37, the SPIR-V code translated manually from APL is loaded on any GPU thanks to our Vulkan loader program. In Fig. 2.38, we see how our approach processes the network flows captured by the host through the CNI abstraction, preferably via Cilium or other kernel-passthrough abstractions such as DPDK to accelerate packet processing further.

Like previous NFV approaches, such as Netbricks, in this study, we run the NF unvirtualized. Here, we leverage low-level Vulkan API bindings to allow us to refine the GPU computation pipeline to better accustom to the performance of each hardware. To elaborate, we declare static resources that exist along the complete lifetime of the program, with other parts, such as program loading, working dynamically.

<sup>8</sup><https://github.com/jhvst/haavisto2021vulkan/>

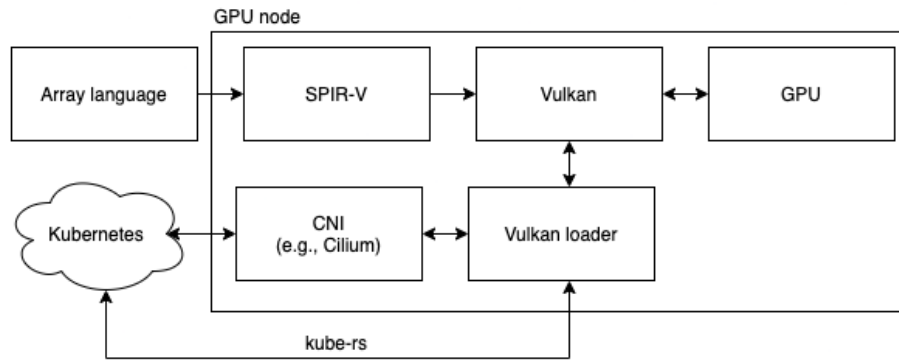


Figure 2.37: Our proposed software stack of a GPU node.

Our Vulkan-based loader program uses a Rust-wrapper library called `ash`<sup>9</sup> to interface with the GPU. In the abstract, `ash` provides conveniences, e.g., wrapping returned structures in vectors and providing default values for all API call structures. `ash` is considered low-level in the sense that all operations are “unsafe” in Rust, which means that the programmer must consult the official Vulkan API documentation to avoid undefined behavior. In turn, the Rust compiler is less useful than usually on memory safety: unsafe calls are meant as code blocks where the programmer surpasses the type system, and everything in `ash` that interfaces with the GPU is unsafe.

### Program Loading and Execution

The program flow of the loader is the following. We start by declaring static variables, i.e., ones that extend to the program’s complete lifetime. Once initiated, the shader modules and pipeline layouts are retrieved from Kubernetes via the CNI using a Rust client library to Kubernetes called `kube-rs`<sup>10</sup>. This could be considered analogous to pulling a container image from a registry for deployment. After this initialization, we open the ports specified by the Kubernetes Services and start waiting for input to the functions. Once input is received, it goes through a standard Vulkan compute pipeline (detailed in our open-source code). In the final step, once the result is copied back to CPU memory, the result is written back to the network socket specified by the Kubernetes Service file. Here on, it is the job of Kubernetes CNI to forward the response. As such, it could be argued that this way, our approach yields itself well to the working principles of chained NFs. This allows such NFV chaining to be modeled in Kubernetes, spanning many Services with GPU NFs and traditional CPU containers complementing each other. However, we yield to the fact that our Vulkan pipeline is not state-of-the-art. Efficient use of Vulkan is a subject of many books, thus out-of-scope of what could have been prepared for this study. As such, further pipeline optimizations are left for future studies. Nonetheless, the Vulkan loader is the corner stone of our architecture and is highly technical. We decided to distribute it as an open source library<sup>11</sup>.

### Orchestration

As mentioned, Kubernetes is an orchestrator system for containers and can be considered as a system that conglomerates many physical servers into a single abstracted computing unit. Kubernetes is designed to schedule software packaged into containers, but it can be retrofitted to serve other purposes due to its modular design. We propose a way to retrofit Kubernetes to orchestrate GPU microservices in a container cluster. The GPU instances use the networking infrastructure to communicate SPIR-V binaries. This way, the GPU programs are managed as Kubernetes Services, and further, the SPIR-V binaries correspond to GPU containers. We may consider our approach “non-invasive,” as we do not limit the functionality of the standard Kubernetes.

Next, to allow GPU NFs to be orchestrated, we integrate our Rust-loader application with Kubernetes Service abstractions (see: Fig. 2.39). In particular, the novelty of our approach here comes from the fact that SPIR-V programs can be inlined within Kubernetes Service abstraction as string-valued

<sup>9</sup><https://github.com/MaikKlein/ash>

<sup>10</sup><https://github.com/clux/kube-rs>

<sup>11</sup><https://github.com/periferia-labs/rivi-loader>



metadata. This is possible because SPIR-V kernels are small in size: our example of random forest prediction algorithm weights 2kb without compression. Furthermore, we achieve a non-virtualized and non-container approach while still managing to leverage Kubernetes APIs by defining the GPU nodes as non-schedulable (by not having CRI installed, see: Fig. 2.38) and the Service abstractions as services without selectors. This way, we achieve two things: 1) Kubernetes does not try to schedule any existing worker nodes to spawn containers for the GPU Services as the Service declarations lack selectors, and 2) the Services are still exposing as a cluster-wide DNS entry via CNI. This keeps our proposed approach non-invasive to existing Kubernetes installations. Hence, our proposal to orchestrate GPU NFs this way can be viable for existing Kubernetes setups. The primary benefit of integrating with the standard Kubernetes scheduling workflow, oriented around the Service abstraction, is that the GPU Services in our proposed architecture are visible, routed, and exposed within the cluster as any other standard container-based Service. This way, we can simplify our loader program: CNI handles the networking to the primary node where CoreDNS handles Service discovery. As mentioned above, for better networking performance, CNI integrations like Cilium can leverage kernel-passthrough technologies like BPF and DPDK. Such an approach might be helpful when running the GPU NFs on the network's edge, providing even lower latency to data inference and higher packet throughput.

### Practical deployment and orchestration of GPU compute resources with Kubernetes

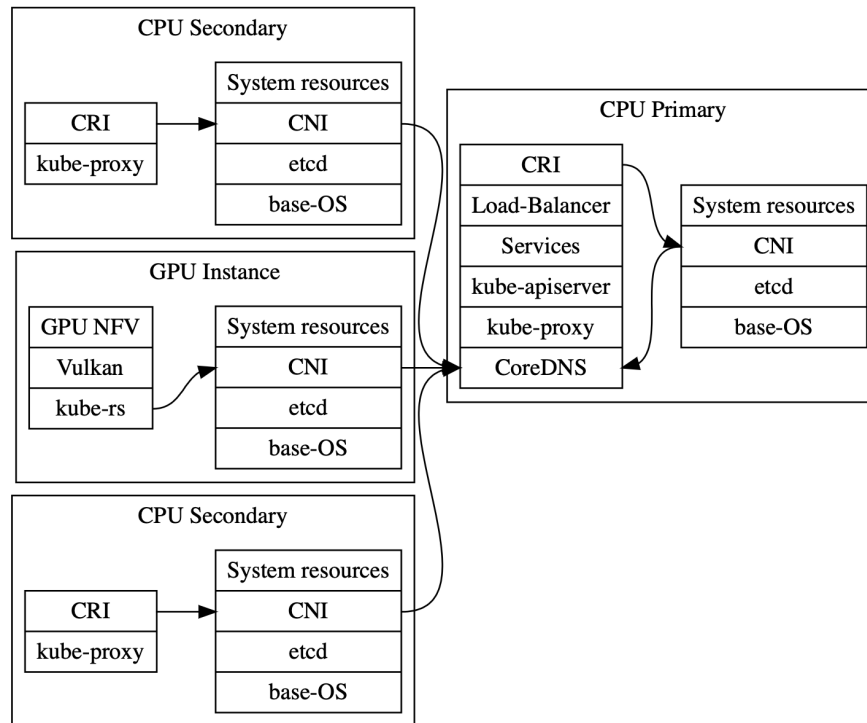


Figure 2.38: Kubernetes integration. Here, four instances form the Kubernetes abstraction. The bottom parts of the records enable the abstraction on top, which, in turn, communicate with other nodes in the cluster.

Due to Kubernetes' modular design, the architecture of each installation may vary. Here, we propose a barebone installation without many assumptions on the underlying modules. To illustrate our idea, we consider a network of a four-node system with one primary node and three secondaries (pictured in Fig. 2.38). Here, of the three secondaries, two are container hosts, whereas one is a GPU host. We propose that the GPU host remains non-schedulable for containers, per arguments on NF performance. On the cluster level, we do not cordon the GPU hosts, but instead, never install a CRI on the host. This is a possible solution when the Kubernetes cluster installation is done modularly, e.g., per instructions on [14]. I.e., even though the GPU instance cannot schedule containers, it remains cluster-accessible when the CNI is installed adequately. The CNI is just another module on the Kubernetes stack and can be, e.g.,

**flannel**. The CNIs then rely on **etcd**, a distributed kv-storage and one of the most basic requirements for a Kubernetes instance. The hierarchy in Fig. 2.38 is bottom-up: the bottom part enables the use of higher abstractions, situated higher on the stack, which then interface cluster-wide. We insist that the CNI (networking), CRI (container runtime), and the GPU host operating system may be whatever in our proposed architecture. The independence is achieved by relying on the already modular interfaces provided by Kubernetes.

Once these essential services are installed on each node, a DNS layer for adequately addressing the cluster is needed. Usually, this is done via CoreDNS as it has a Kubernetes integration, but it may also be some other DNS server. In our proposal, the DNS server is required to route traffic to GPU-based Kubernetes Services. To quote the Kubernetes documentation, a Service is "an abstraction which defines a logical set of Pods and a policy by which to access them (sometimes this pattern is called a micro-service)." In other words, Service is the Kubernetes-native way of defining a microservice. With the proper abstractions detailed above, this promotes the GPU-based NFs to microservices. As such, it is logical for our proposal on the orchestration of GPU NFs as microservices to interface with the Service abstraction.

<pre> apiVersion: v1 kind: Service metadata:   name: my-service spec:   selector:     app: MyApp   ports:     - protocol: TCP       port: 80       targetPort: 9376 </pre>	<pre> apiVersion: v1 kind: Service metadata:   name: my-service   spirv: AwIjBwAFAQA... spec:   ports:     - protocol: TCP       port: 80       targetPort: 9376 -- -- --  apiVersion: v1 kind: Endpoints metadata:   name: my-service subsets:   - addresses:     - ip: 192.0.2.42     ports:     - port: 9376 </pre>
--	--

Figure 2.39: Kubernetes Service declarations. On the left, one with selectors, which would spawn container image MyApp. On the right, a Service without a selector, which would not spawn any containers. Our proposal uses the right-hand side version to spawn GPU NFs as microservices.

However, we do not use the standard declaration of Service because we do not want to run the GPU NF microservices inside a container. To elaborate, using containers for GPU applications is tricky and opinionated. For reference, the Kubernetes documentation on using GPUs<sup>12</sup> lists that to use, e.g., Nvidia GPUs, the cluster has to: 1) have Nvidia drivers pre-installed, 2) have nvidia-docker installed (which only works on Linux), 3) use Docker as the CRI. The steps include similar tweaks for AMD, including allowing the nodes to run in a privileged mode. In essence, these approaches limit the flexibility of the GPU node installations and require elevated execution modes for containers, which are usually meant to run unprivileged. Instead, we prefer declaring the Service abstraction without a node selector. To compare these declarations, consider Fig. 2.39. To avoid the Service becoming an orphan, we must declare an Endpoint abstraction. This can be done in a single command by separating the configuration declarations with three dashes, as shown in Fig. 2.39. We note that in our proposal, drivers still have to be installed to support Vulkan, but our proposal allows the GPU nodes to remain operating system agnostic while not relying on containers.

As can be seen in Fig. 2.39, our proposal for declaring GPU microservices within Kubernetes requires the SPIR-V binary file to be inlined within the metadata description. The binaries are encoded in base64. After the creation of the Service file, CoreDNS triggers a CNAME entry creation for the Service. We clarify that this is a standard procedure in Kubernetes, triggered by the orchestrator on Service creation. By default, this would expose an endpoint by name `my-service.default.svc.cluster.local` in each of the cluster's nodes' CNI routing table. In our proposal, what follows is that the Service creation events

<sup>12</sup><https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/>

Table 2.7: Runtime comparison of random forest model of 150x6000x300 trees between Cython and SPIR-V.

	Device name	Runtime
CPU (Cython)	Intel Core i7-9700	380ms
GPU (SPIR-V)	NVIDIA GeForce GTX 1080 Ti	318ms
	AMD Radeon RX 6900 XT	136ms
	Apple M1	201ms

must be listened to by the GPU nodes using a Kubernetes client-wrapper, e.g., `kube-rs`. This means that each GPU node listens to the primary Kubernetes node to announce changes in the cluster Service entries. One such is found, the GPU nodes would pull the declarations using the kube-apiserver API. This would communicate the SPIR-V binaries required to load the particular GPU microservice into memory. Finally, if the Endpoints match the GPU node's local IP address, the microservice is provisioned using Vulkan. Once the microservice is initialized, the corresponding port found in the Service declaration is opened on the node by CNI. When the port receives packets, the contents are unmarshaled in Rust and passed to the GPU. Once done, the result is written back to the connection from which it came. As such, it is the responsibility of the Kubernetes CNI to route data in and out. Such reliance cuts two ways: on the one hand, our proposal only works with Kubernetes. But, we do not make assumptions about what the Kubernetes installation has to be like. I.e., it is possible to leverage Kubernetes abstractions on top of the GPU NFs, such as LoadBalancer, to balance the load among many GPU nodes or any other networking constructs. For example, to create a function chain of NFs, we would encourage the chaining to be declared on the Kubernetes level. This way, the function-chain may mix both GPU NFs and CPU NFs by interfacing via CNAME entries. By this reasoning, we consider our proposal capable of introducing GPU NFs as part of a heterogeneous system consisting of CPU and GPU nodes.

This section describes the use-case we selected for our evaluation and the related results.

**Use case description** The empirical part of the work is a development to [10], in which RF prediction is used to label encrypted hypertext data. We use the paper's application in this study by extracting the prediction algorithm as a GPU microservice, using the well optimized CPU implementation of scikit-learn RF (written in Cython) execution time as the baseline. We choose this application also because the binary decision tree traversed by RF algorithms can be parallelized as each three is data-independent. Further, the number of trees to traverse is usually well above the usual logical threads that a CPU may have. As a thesis, such workload should see performance increase on GPUs, as GPUs can have up to thousands of cores. To elaborate, the physical processor of GPU is less likely to get throttled by its physical capabilities compared to its CPU counterpart, assuming that the program's execution time takes long enough time.

For the sake of reproducibility and to allow the community to further build upon our proposed architecture, all software components have been released in open-source on GitHub<sup>13</sup>. This includes:

- the Vulkan bootloader library written in Rust and used to load the SPIR-V code<sup>14</sup>;
- Kubernetes configuration files to orchestrate a set of GPU-compute nodes;
- the SPIR-V assembly code of our network flow classifier to serve as an example (and the related but optional APL model).

**Results** The benchmarks' (Table 2.7) baseline was run on an Intel Core i7-9700 processor (8 cores). As introduced before, the application was a RF prediction over 150x6000x300 dimensional trees. As shown in Table 2.7, the Cython code on OpenMP resulted in the baseline of 380ms. The results are gathered independently of the Kubernetes workflow as neither did the baseline run within Kubernetes. Yet, we do not expect any particular bottleneck to be introduced by Kubernetes itself. On the GPU side, all devices executed the identical SPIR-V code, which is possible thanks to the Vulkan-based architecture. To evaluate this aspect, we used different GPU manufacturers (Nvidia, AMD, Apple) and different processor

<sup>13</sup><https://github.com/jhvst/haavisto2021vulkan/>

<sup>14</sup><https://github.com/periferia-labs/rivi-loader>

architectures (x86, ARM). While not evaluated at this time, the same architecture also allows different GPUs to be mixed on a single computer to achieve manufacturer-independent multi-GPU support.

Regarding the numerical results, the main point is that in all cases the GPUs are faster than the Cython code despite the delay of memory copies (especially for discrete GPUs), which proves the potential of our architecture. If we look more precisely at the different GPUs results, they are easily justifiable. The AMD card is the most powerful of the three and it also performs the best in our tests. On the other hand, the M1, while initially expected to lack in computation power in comparison, for example, to the GTX 1080 Ti, performs however significantly better than the latter. This can be explained because the chip's memory is integrated with the CPU thus what is lost in raw performance is gained in memory copy latency. We stipulate that with other more complex applications, such as neural networks, the M1 would start falling behind in performance as the memory copying becomes less relevant compared to core count and other physical properties. We remind that the purpose of this paper was not to optimize the memory copy delay but that this topic is already covered in related work.

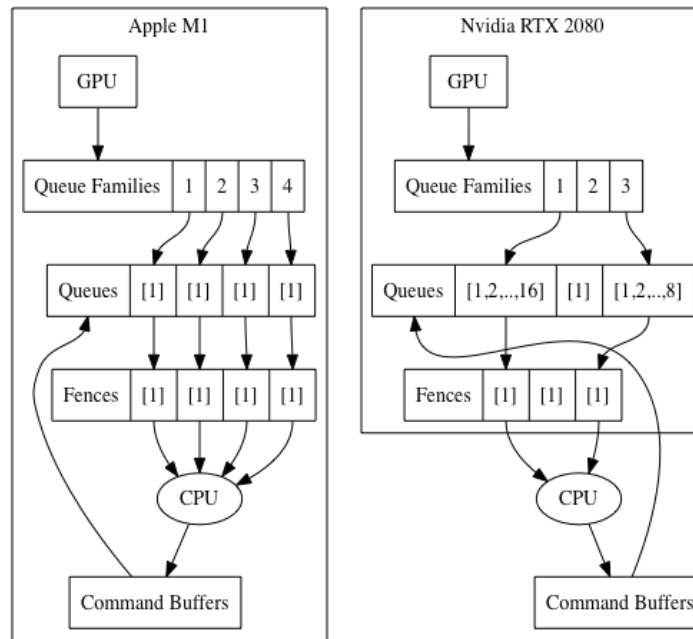


Figure 2.40: Example of two GPU architectures layout leading to different optimizations

## Evaluation

Per GPU architecture optimization left aside (it is encouraged thanks to Vulkan low-level scheduling that can adapt to the specific design of any GPU architecture as illustrated in Figure 2.40), the results prove that our architecture is effective and able to deploy and orchestrate GPU accelerated network functions on multiple heterogeneous hardware (in a same deployment) by relying solely on open API and software, which constitutes a real advance in the field.

## 2.6 Closed-loop solutions

Given the individual monitoring components as well as the orchestration algorithm previously exposed, the question of their virtuous integration in a global closed-loop solution is the point we address in this section. Indeed, for low-latency services especially, it is required to react very diligently to abnormal conditions or specific events, which alters the quality of the service. For this, it is necessary to have an efficient monitoring plane but the reaction part is also crucial. For this, we think closed-loop model is very promising. In this section, we introduce the closed-loop concept and present the results of experiments we have performed in order to know where and when closed-loop mechanisms should be activated.

### 2.6.1 Closed-loop for QoS

Closed-loop in networking gives opportunity to use feedback for a better control. In general, networks are managed in open loop with no real time feedback about the quality of service. Of course, operators use various management system to monitor their networks and verify that everything work as expected. In case of problem, deep inspection is necessary to locate and remedy to the problem. This could take time, in particular when the issue is not obvious (e.g. packet loss on some links). However, a closed loop mechanism is mandatory when an operator sells services with QoS guarantees. This is related to the well-know Service Level Agreement (SLA) and the technical details as Service Level Specification (SLS). A closed loop system is then a good tool to guarantee the QoS. Indeed, a closed loop is composed of a monitoring or probing system that collects measurements about the service and provides these measurements to a system that compared them against a reference. If the measurement differs from the reference, a third part of the closed loop system is able to modify the network configuration in order to re-enable the performance of the service and thus guarantee the expected QoS.

The available classic methods to monitor the network are as follow:

- Signaling is a form of closed loop. The signaling protocol sends regularly packets which aim to monitor the end-to-end connectivity and verify that the characteristics of the connection are already met. In case of problem, signaling protocol is able to find another path to properly re-establish the connectivity or inform the management system about the problem which in turn could instruct the signaling protocol to use an alternate path.
- Operating And Maintenance (OAM) packets. The is another in-band solution to monitor connectivities. In case of problem, the IP router is able to report the issue to the management system which in turn makes appropriate correction.
- Dynamic probes that measure the quality of service of the connectivities. However, this mechanism is particularly costly on large scale network where operators should deploy many probes. Another disadvantage is that it is complicated, and thus costly, to monitor all services.
- Backup path could also be configured on complement to the above systems. In this case, the Edge router could shift packets from the nominal connectivity to the backup path to continue to guarantee the expected QoS in case of failure.

More recently, a new mechanism appears in the toolbox for the operators opening a new area for measurements. The telemetry is completely changing the way to make measurements and monitoring traffic in an IP network. Instead of deploying expensive probes, each IP router becomes a probe where it is possible to perform some measurements. All of this measurements are sent to a Telemetry Collector which could be used to compute some statistics but also to monitor the state of the network. By adding some triggers to the Telemetry Collector, it becomes easy to close the loop and perform some corrective actions in case of failure or when some measurements exceed some threshold e.g. the delay on link between R1 and R2 exceeds the nominal 10 ms value.

Finally, IGP routing protocol with Traffic Engineering offers an alternative for closed loop. Indeed, the routing protocol is able to flood metrics that could be used to react on modification. In particular, Extended Metrics have been standardized by IETF to propagate delay, jitter, packet loss and bandwidth not as an administrative values, but as parameters which result of a measurement. E.g., a router is able to measure the delay on its links up to its neighbors and then advertises these delays within the Traffic Engineering parameters. A SDN controller that listens to these Traffic Engineering topology (e.g. with BGP Link State) could also react on failure and take appropriate actions. Note also that delay and jitter parameters could be flood with a specific bit set to 1. Named 'Anomalous', this bit serves the router to announce a abnormal measurement. Such indication could be used by an SDN controller to take action immediately.

### 2.6.2 Evaluation of closed-loop scenarios

Within MOSAICO, we performed a certain number of evaluation of closed loop mechanisms to determine which one is the most appropriate solution for low-latency services and how to configure it. The following sections describe the tested mechanisms which have been evaluated regarding a precise use case.

## Delay control

For this test, a VPN service is configured. Even if a VPN is not really a low-latency service, it offers some similitude in terms of QoS:

- Dedicated service: connect two or more customers for dedicated traffic
- QoS: guarantees quality of service for the VPN traffic, in particular the delay between two customer edges as key parameters for this use case

This use case starts by setting up a L3VPN service between two customer edges (CE) and then monitors the delay between these two CEs. An orchestrator (Ansible and AWX) is in charge to configure the L3VPN service. First, Provider Edges (PE) routers are configured to route the CEs traffic within the VPN (1). Then, a dedicated connectivity is deployed between these 2 PEs with a constraint on the end-to-end delay in order to meet the expected QoS (2). This dedicated connection is enforced by means of a Segment Routing (SR) path (3). The SR path with Traffic Engineering constraints is computed and enforced between the PEs by an SDN Controller. For that purpose, the Path Computation Element Protocol (PCEP) is used between the SDN Controller (OpenDaylight), acting as a Path Computation Element (PCE) server and the routers, acting as a Path Computation Client (PCC), to enforce the SR-TE path. Finally, an external probe monitors the delay of the network links and report measurements to a collector system. This collector is based on an influxDB database which stores all the measurements.

Several scenarios have been tested for the closed loop:

- At Orchestration level

Closed Loop at the orchestration level is depicted in Figure 2.41.

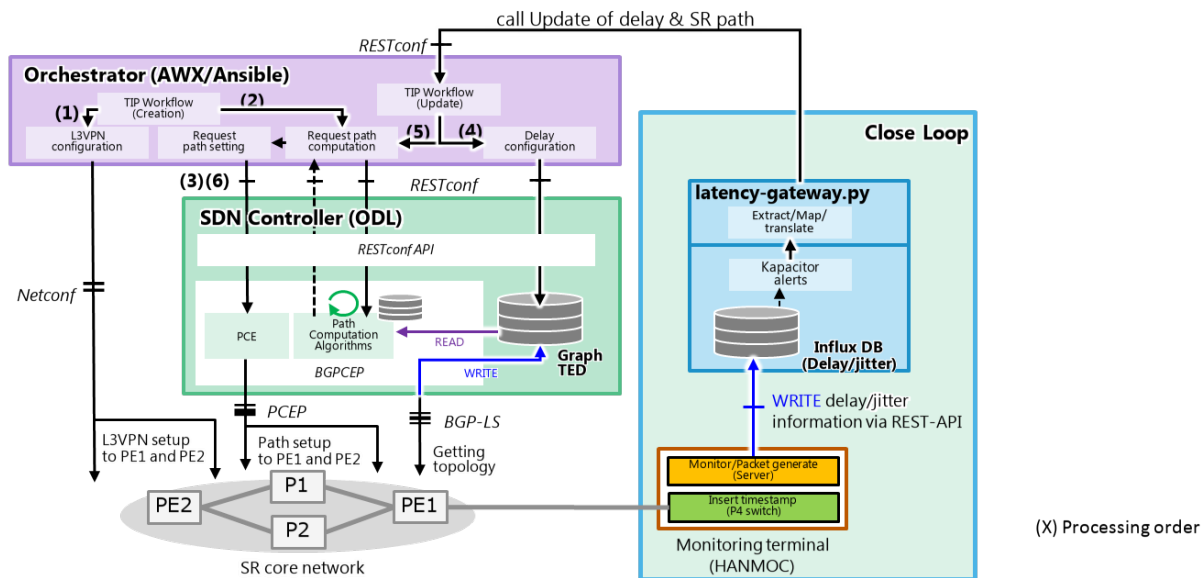


Figure 2.41: Closed Loop at Orchestration level

A trigger (python script) is configured on the influxDB to send a message to the Orchestrator when delay exceeds the L3VPN service constraints (4). On reception, the Orchestrator gets L3VPN service description and triggers the SDN Controller to recompute a new path based on a network topology which takes into account this new delay measurement (5). The new path is then enforced by means of PCEP protocol on the PEs routers.

We measure the reaction time of the closed loop as the time necessary to setup the new path. More precisely, by measuring the time elapsed between the detection of the exceeded delay by the InfluxDB and the enforcement of the new path. It takes between 3 and 10 seconds to setup the new path. Most of the time is taken by the orchestrator for service processing. The part of the SDN controller is negligible. Indeed, we are able to measure the time elapsed between the call to the OpenDaylight Rest API and the enforcement of the new path on the router (OpenDaylight logs are stamps with milli-second precision).



- At SDN Controller level

Closed Loop at SDN Controller is depicted in Figure 2.42.

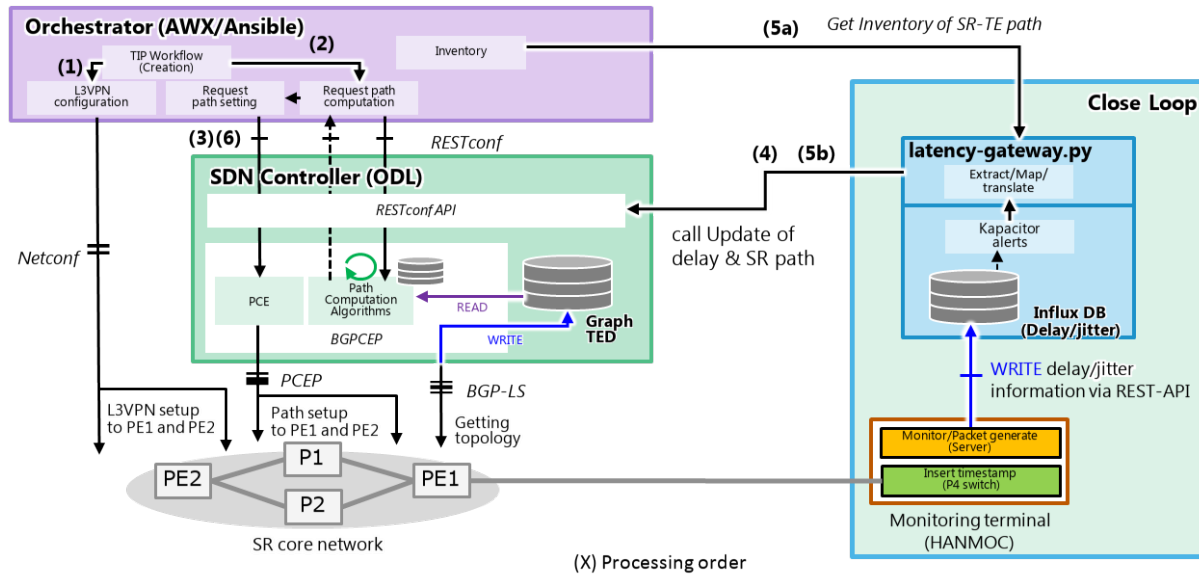


Figure 2.42: Closed Loop at SDN Controller level

A new trigger (python script) is configured on the influxDB in place of the previous one. This time, the python script uses directly the REST API interface of the SDN Controller (4 5b). Since the python script has not all parameters related to the L3VPN service, a call to the Orchestration Inventory module is performed in order to collect suitable information to modify the path(5a) i.e. source and destination IP addresses and QoS constraints.

Same measurement as before is performed. This time, the reaction time is in the range of 50 to 100 ms, around 100 faster than with the Orchestration scenario.

## Bandwidth control

As for the previous test, the VPN service is still used. Indeed, the second constraint after the delay that a VPN service could request, is the bandwidth. In this use case, the closed loop is used to verify that the reserved bandwidth for the L2VPN service is always guaranteed. For that purpose, telemetry is configured to monitor the bandwidth usage of the links in the network and monitors the traffic on the connectivity that has been enforced between PEs for the L2VPN service. As for the delay use case, an SR Path with Traffic Engineering is used to provide this connectivity. An SDN controller, i.e. a PCE server, is used to compute and enforce the SR-TE path on the PEs. A collector centralizes and stores all telemetry measurements sent by the IP routers deployed on the network. Instead of triggering actions when delay exceeds the constraints, here, an average window is used to verify that the SR-TE path could guarantee the requested bandwidth and that other traffics will not consume more than the links capacity minus the reserved bandwidth. This average window is used by the telemetry collector to report regularly bandwidth consumption to the SDN controller. In case of congestion or insufficient bandwidth guarantee, the SDN controller re-computes the path and enforces it in order to ensure that the requested bandwidth for this SR-TE path is always guaranteed.

Again, two scenarios have been evaluated to measure the closed loop reaction time:

- Global trigger where reported bandwidth for all links are verified against a global threshold
- Per link trigger where reported bandwidth for a given link is verified against a configured per link threshold

This time, the evaluation has been done on a commercial SDN controller (Juniper Northstar) since such closed loop mechanism based on telemetry is not yet available with OpenDaylight controller (see Figure 2.43).

Closed Loop reaction time is measured in the same condition as for the delay use case by measuring the time elapsed between the injection of extra traffic in the network and the enforcement of the new

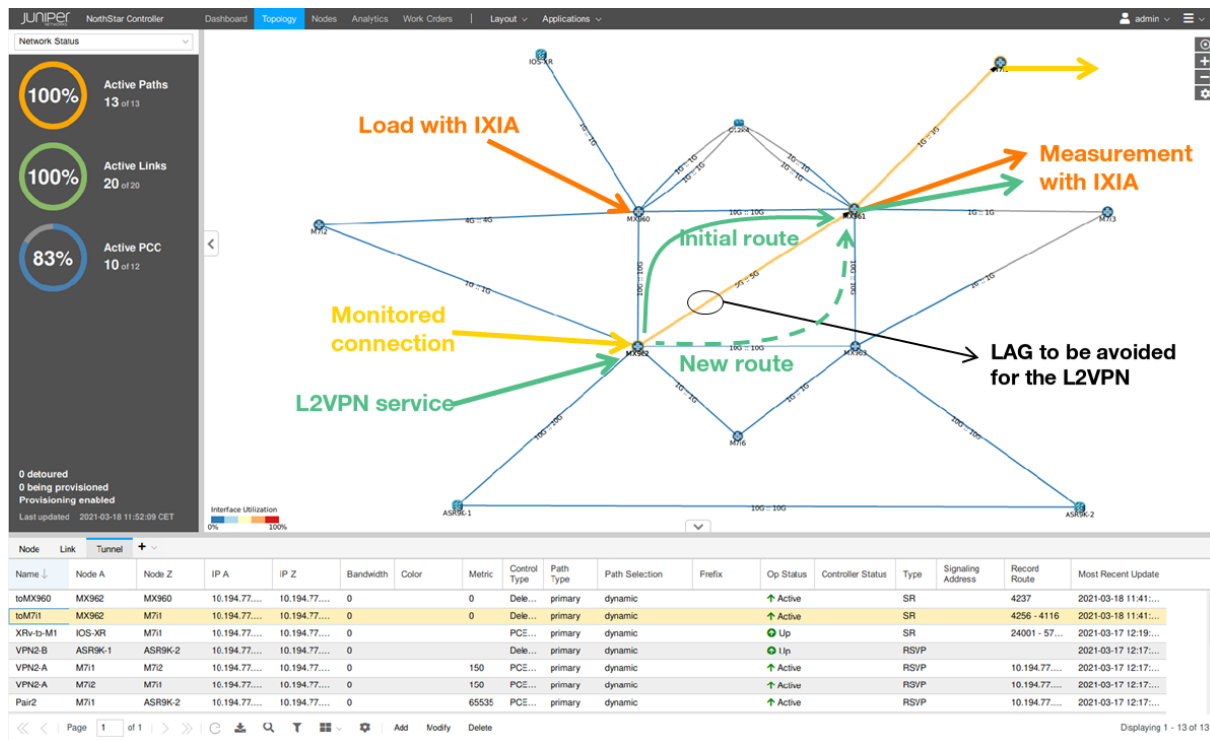


Figure 2.43: Juniper Northstar Controller and Telemetry closed Loop

path on the PE routers. With the global trigger, the reaction time is important, mostly due to the size of the average windows (1 min.). More precisely, as larger is the average window, as slower is the closed loop reaction. We are unable to measure the exact time necessary for the SDN controller to re-compute and enforce the new path once it detected that there is a problem with bandwidth, but, this is not important. Indeed, packet loss starts as soon as the traffic load is injected into the network, and the QoS of the L2VPN service is degraded during the whole average window period, which is too impacting. It is possible to reduce the size of the average window down to 5 sec. However, as it concern the global trigger, for large scale network, there is a great risk of instability or to frozen the SDN Controller as the latter will not finish to process all tunnels and all links within this 5 sec. period. With the per link trigger, the reaction time is slower due to the larger size of the average window (15 min.). Unfortunately, the SDN Controller does not offer the possibility to set an average window lower than the 15 min.

## IGP extension metric

The third evaluation scenario is not a closed loop system like the two previous ones but worth being mentioned. The main idea is to use the capacity of the IGP routing protocol with Traffic Engineering to flood extended metrics as measurements of what happens in the network. Indeed, RFC 7471 for OSPF and RFC 7810 - 8570 for IS-IS defines new set of Traffic Engineering Extended Metrics:

- Delay (average, min, max), Jitter and Packet Loss with Anomalous flag
- Residual, Available and Used Bandwidth

These extended metrics could be administrative or measured.

A SDN controller needs to acquire the network topology to be able to perform path computation with constraints. BGP Link State (BGP-LS) is the de-facto protocol which is used for that purpose. Of course, all IGP extended metrics have been standardized in addition with the standard TE metrics for BGP-LS. Thus, a SDN Controller could collect in "real-time" all these new measurements. It is necessary to precise what we understand behind "real-time". In fact, a router will not flood immediately a new OSPF LSA or a new IS-IS LSP to announce a new TE metrics. In fact, these TE metrics are considered as not mandatory to compute the routing table and thus are considered with less priority. But, we measured that TE metrics are flooded within the seconds once modified by the routers. Then, it is necessary to add the convergence time to determine when the SDN Controller could received new information at the

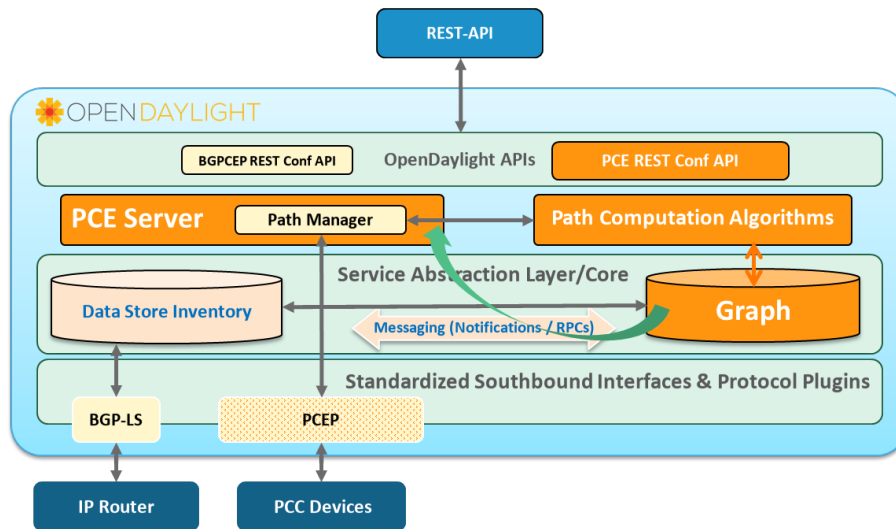


Figure 2.44: OpenDaylight closed Loop based on TED

latest. This greatly depends of the implementation of the IGP routing protocol, but also from the nature of the network topology itself and where the BGP-LS speaker (i.e. the router that convert the IGP TE metrics to BGP Update message) is located, thus when it will received the updated LSA or LSP. IGP and BGP timers could be tune to speed up this process and, regarding the anomalous flag for delay, jitter and packet loss, we could expected that OSPF LSA or IS-IS LSP are sent immediately without any delay, but we are not able to measure this point.

By monitoring its Traffic Engineering Database (TED), the SDN Controller is able to automatically take action to correct the connectivities it manages. E.g. it could detect automatically a modification of a delay on a link and re-compute a path if the new delay value increases the total delay budget which in turn does no more respect the QoS constraints. Such automatic closed loop based on network topology can be integrated in OpenDayLight as depicted in Figure 2.44.

### IntMon: a P4-based INT implementation

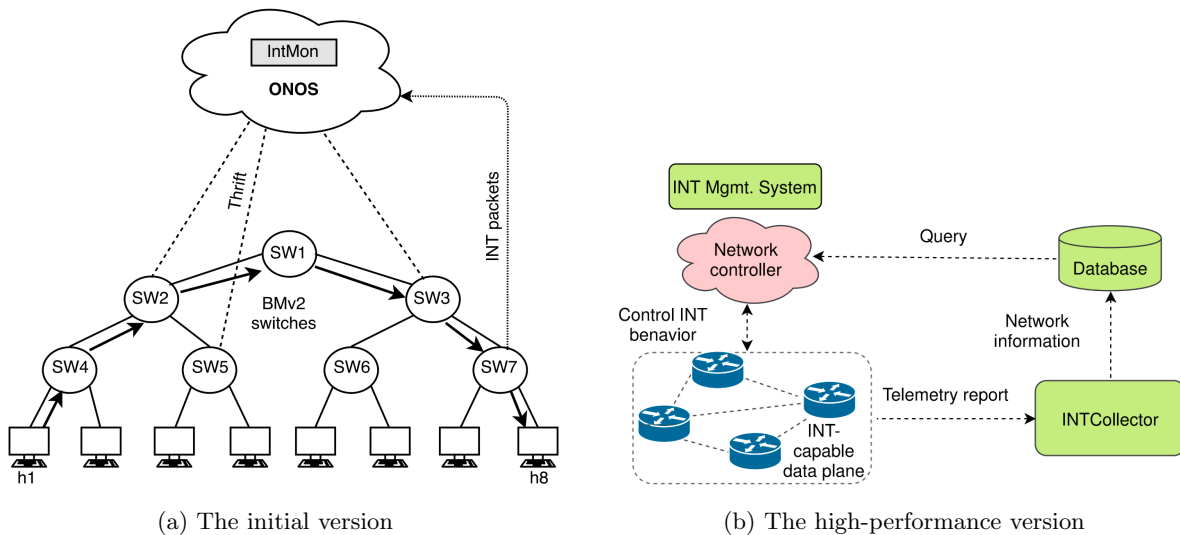


Figure 2.45: IntMon – a P4-based INT implementation

In this last paragraph, we present IntMon [24], an SDN network application in Open Network Operating System (ONOS). It implements the P4-based in-band network telemetry technology to monitor the network performance. The first implementation of INT encapsulated in UDP packets using ONOS controller and BMv2 software switches that are configured using P4 language and evaluated using Mininet

in a simple tree topology as shown in Figure 2.45a. In this version, INT packets are sent from INT sinks to the ONOS controller through the Thrift protocol. The packets are then processed at the low-level layer of the ONOS core by the INT data processor.

Since INT monitoring works directly at the data plane, high INT report rates in real-time and complex scenarios imply the need for a high performance INT collector that has been implemented in the next version of IntMon. The INT collector replaces the INT controller services implemented by the ONOS controller and stores INT metric values in a database as shown in Figure 2.45b.

The evolution of IntMon shows that the direct communication between an INT sink node and a SDN controller is not suitable for high-performance network monitoring. Still the next version is not suitable for detecting and solving in real-time, e.g. closed loop, due to the fact that IntMon pushes events to InfluxDB first then the network controller needs to pull the information from InfluxDB, consequently a delay is inevitable.

### 2.6.3 Closed-loop recommendations

From the different experiments and measurements we performed, recommendations could be established in order to build the closed loop system of MOSAICO for low-latency service:

- Closed loop must be performed at the SDN Controller level if the problem concerns only the data plane , e.g. network failure, exceeded delay .... and must be performed at the Orchestration level only if the issue concerns the service itself. Indeed, if the issue impacts the service itself, only the Orchestration level has the knowledge of all parameters and the global view of the service, thus being able to modify it to perform appropriate corrections. In addition, it is also preferable to perform appropriate corrections by the network devices themselves when possible, and inform afterwards the SDN Controller / Orchestration e.g. switch traffic to the backup path when failure is detected.
- For bandwidth monitoring, telemetry system is not well adapted for closed loop but works well to monitor the traffic, checks that the network is working well and optimizes global bandwidth consumption in an IP network. Increasing the telemetry measurement in order to decrease the average window measurement is feasible, but this will not scale on large network. Indeed, the collector system may be quickly saturated and it will be time consuming to check everything due to the amount of measurements. As a consequence, closed loop reaction time is too slow to re-route SR-TE path to avoid packet loss.
- As monitoring or telemetry could generate a huge amount of data, measurement frequency should be configured as a compromise between a faster closed loop reaction time and the number of measurements regarding the size of the network to keep the system scale. In addition, closed loop system should privilege short circuit for a faster reaction time e.g. by limiting the number of intermediate boxes between network devices and SDN Controller. The closed loop system must be stable. Indeed, a too high precision monitoring, or too frequent measurements, could conduct to oscillation: the reaction mechanism should not re-route path on micro-events.
- IGP routing protocol with extended metrics could be used to reports the state of the network i.e. topology with link state information to the SDN controller. In turn, the SDN controller could automatically trigger path re-computation if necessary to ensure that the QoS of the impacted path continue to be guaranteed.

## Chapter 3

# Ongoing and Future work

As a second chapter of this mid-term project report, we introduce here the piece of work which is either ongoing or planned as next project steps. It essentially concerns the consideration of 5G traffic delivery with L4S as described in Section 3.1, the integration of telemetry features to make the monitoring and control planes compliant with low-latency constraints as presented in Section 3.2 and finally the implementation of a modular Cloud Gaming platform in Section 3.3. Finally, since the project aims at implementing a global testbed integrating some components previously considered and evaluated or implemented, we provide some early elements in Section 3.4. One can notice that we do not report elements on very recent work, such as the detection of low-latency attacks since, at the time of the report writing, it stands for an immature work that need further development before being exposed.

### 3.1 Improvements for Cellular Networks

We have detected some issues of current solutions such as L4S and the delivery of cloud gaming platforms when using it under degraded network conditions such as cellular networks. This will then be a work undertaken by the project in the coming months, to find some improvements for use in cellular networks.

For L4S, one envisioned option is the make L4S evolve to be usable with 5G architecture as defined by the 3GPP and mainly with the split of the base station into 3 components, namely a radio unit (RU), a distributed unit (DU), and a centralized unit (DU). For this, an option is to adapt L4S to be usable with the PDCP (Packet Data Convergence Protocol) protocol, instead of the IP protocol as it currently is. Our future work will refine this direction.

In order to evaluate and confirm our work for representative cellular networks, it is required to perform tests under real and realistic conditions. However, cellular networks are very variable and some cells can be overloaded, others very few, depending on the location of end-users with regards to the location of the antenna, regarding the number of concurrent users on the same cell, etc.

The french actor ARCEP, french regulator of communications, regularly makes measurements to compare the network operators in various situations. For cellular network, it defines 3 zones, namely dense, intermediate, rural zones<sup>1</sup>. In our work, we decided to use a finer categorization scale with 5 categories based on the downlink throughput, namely excellent, very good, good, average and bad. Furthermore, we also want to evaluate our solution in mobility conditions (e.g., inside a car traveling on a highway).

For being able to perform tests in a reproducible way while being representative of real network conditions, we opted to measure transmission opportunities of commercial deployed base stations and then replay this behaviour for our experiments. For this, we will use 2 tools, namely saturator<sup>2</sup> and mahimahi linkshell<sup>3</sup>.

Saturator will be used to capture transmission opportunities (txops) traces. As its name implies, Saturator saturates the radio link of a User Equipment (UE) with a Mobile Network. During txops captures, the client sends timestamped data packets to the server which are used to compute the RTT upon reception of the corresponding Acks. This RTT value is used to adjust the client congestion window.

<sup>1</sup>Qualité des services mobiles Arcep: <https://www.arcep.fr/actualites/les-communiqués-de-presse/detail/n/qualite-des-services-mobiles-191121.html>

<sup>2</sup>The Saturator tool code is available in the GitHub repository (GitHub - keithw/multisend: <https://github.com/keithw/multisend/>).

<sup>3</sup>The LinkShell tool is available in the GitHub repository (GitHub - ravinet/mahimahi: Web performance measurement toolkit: <https://github.com/ravinet/mahimahi/tree/master/>).

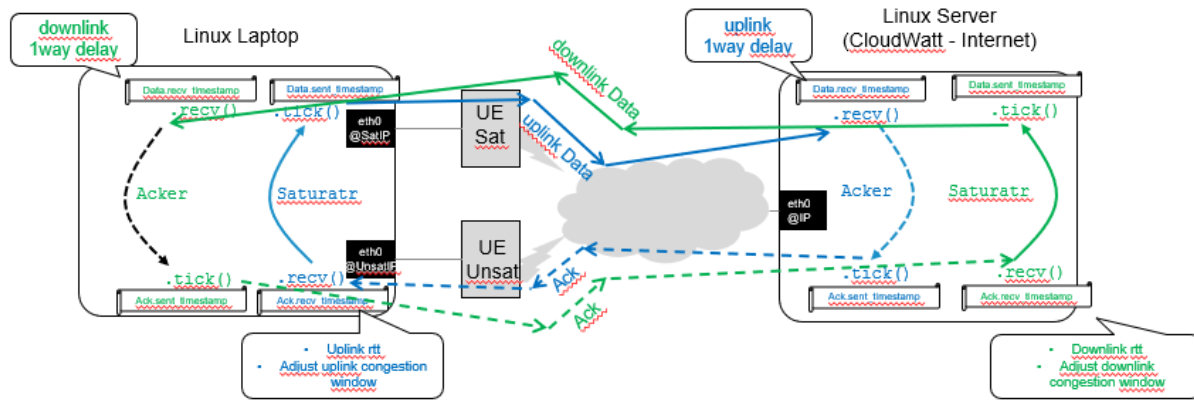


Figure 3.1: Saturator – principle of operation

The same process is applied on the server-side, so that it can modify its congestion window accordingly and hence saturate the link radio of the base station toward the UE. The functioning of Saturator is depicted in Figure 3.1.

At the end of the experiment, raw Saturator data captured on the client side is aggregated and processed with the corresponding data on the server side in order to generate two txops file, one for downlink and one for uplink. Each line in the txops file represents a packet delivery opportunity, the time at which an MTU-sized packet can be delivered. If several MTU-sized packets can be sent during the same millisecond, several lines are present in the file with the same value of the time. If no packet can be sent during few milliseconds, there is a time gap between two successive lines of the file. Such txops file can then be used by Mahimahi's LinkShell tool, to emulate a time varying cellular network exhibiting the same transmission opportunities as during the measurement.

We will make measurements on Orange commercial network for the 6 network previously described conditions and these txops files will be used for our work related to L4S and the cloud gaming use-case, and eventually other services.

## 3.2 Closed-Loop in MOSAICO Architecture

In the Mosaico project, we are aiming for high-performance network monitoring for detecting and solving issues in real-time and think that closed-loop model is really an innovative solution. The initial elements of the envisioned framework are depicted in Figure 3.2. On the first hand, the monitor needs to communicate directly with SDN controllers/orchestrators to be able to notify in real-time the detected events. The communication can be done using network sockets or a high scalable event bus such as Kafka. On the other hand, the monitor must filter out the non-essential events before sending them to the controllers/orchestrators to avoid overloading them.

As seen in section 2.6.3, a model with two levels is required, for reacting more quickly to events, and thus ensuring low-latency. In our envisioned architecture, we called them the global closed loop and the micro closed loop.

The interaction between the monitor and the controllers form a global closed loop reaction. These reactions are required perhaps when we need to have a global view of the system to perform them. For example, we need the controller to perform a reaction that requires deploying a new SDN/NF application to share payload within an existing application because we can only see the interaction between the new application and the existing one at the controller level.

Beside the global closed loop reactions, there exist the micro closed loop reactions that are suitable for the reactions that involve only a local/single application and are suitable for real-time requirements. For example the micro closed loop reaction requires updating an entry of a firewall application. In such a case, the monitor may directly notify the metric values to the firewall application. Another example is the path re-computation by the router itself (PCEP protocol allows that), the switch of packets from nominal to backup path if end-to-end delay measurement exceeds a given limit, etc.. This direct notification allows to reduce interaction time a lot because it avoids communicating through the controller and P4 is really a good target to implement these functions.



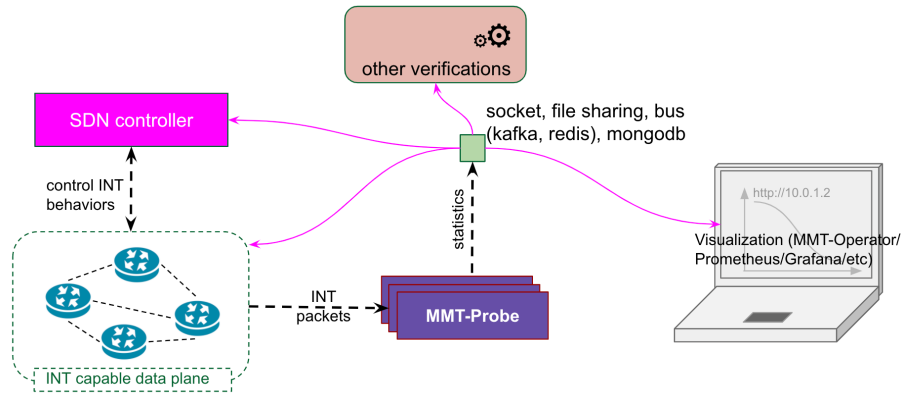


Figure 3.2: In-band network telemetry monitoring framework

The first integration of closed loop reactions are proposed in the testbed as shown in Figure 3.7. The communication APIs between monitors and controllers/orchestrators should be detailed later when we are going into the implementation phase.

### 3.3 Modular Cloud Gaming Platform

Since the main available cloud gaming platforms are not open source, we can not add adjustments for our advanced experimental needs, for instance changing the congestion control of the cloud gamig traffic with priority queuing, ECN markers, intelligent dropping of packets, etc. So we decided to build a our own CG platform. It only supports the streaming of a single game to a single client. In this regard it is more lightweight than a full market-ready platform but nonetheless sufficient for our needs. It can be seen as an open source alternative to software like Parsec or Shadow in the way it is used. In the literature, there has been one former open source attempt of cloud gaming platform that is named GamingAnywhere<sup>4</sup> but even if its design philosophy allow more or less easy modifications, it is based on old version of libraries with some deprecated functions like FFmpeg's but also not necessarily up-to-date ways to transmit data. It would take more time to fully understand the code base and then drastically change the way it works rather than making a new one platform based on up-to-date libraries and transport protocols, but we can still take our inspiration from this project at the architectural level.

Based on this, we decided to create our own implementation of cloud gaming platform which could be also considered as a high quality remote desktop application regarding refresh rate and latency. Currently, the application is a simple client-server service where the server sends encoded audio and video when the client sends its inputs so the server can take in account any actions performed by the user. It is still in development and the final design is not fixed yet but it is built as a modular program so the fundamental bricks are easily reusable if any change in the design occurs. For the development, we selected use FFmpeg libraries for the encoding part, the SDL library for the display, audio and user input parts and X11 library for the server input part (the last one is specific most of the time). The major drawback of FFmpeg libraries is that they are poorly documented, there is only a doxygen API and some example but nothing to understand how to use the API. An alternative that might be used later if needed, or as a second version, is the GStreamer framework<sup>5</sup> (first version is at least to better understand the underlying logic).

In Figure 3.3, we can see the different modules that compose the server side of our service. The blue ones are the permanent modules i.e. the ones required for a full experience while the green ones are more for debugging and measurements. As we can see, the server side is still specialized for Linux system and more specifically those that use X11. The modules are organized as pipelines that look fairly simple but we do not need much more for this side. So in a general way, the pipelines are in the form: grab  $\Rightarrow$  convert (like scaling and sampling)  $\Rightarrow$  encode  $\Rightarrow$  send. At different steps we can optionally add some recorders or display what the server sends to the client the same way it will do. Each modules implements one or more interfaces so that they do not need to know which kind of modules are before or after them in the pipeline but only which kind of object(s) they expect to send or receive. In our case, it is mostly

<sup>4</sup><https://gaminganywhere.org/>

<sup>5</sup><https://gstreamer.freedesktop.org/>

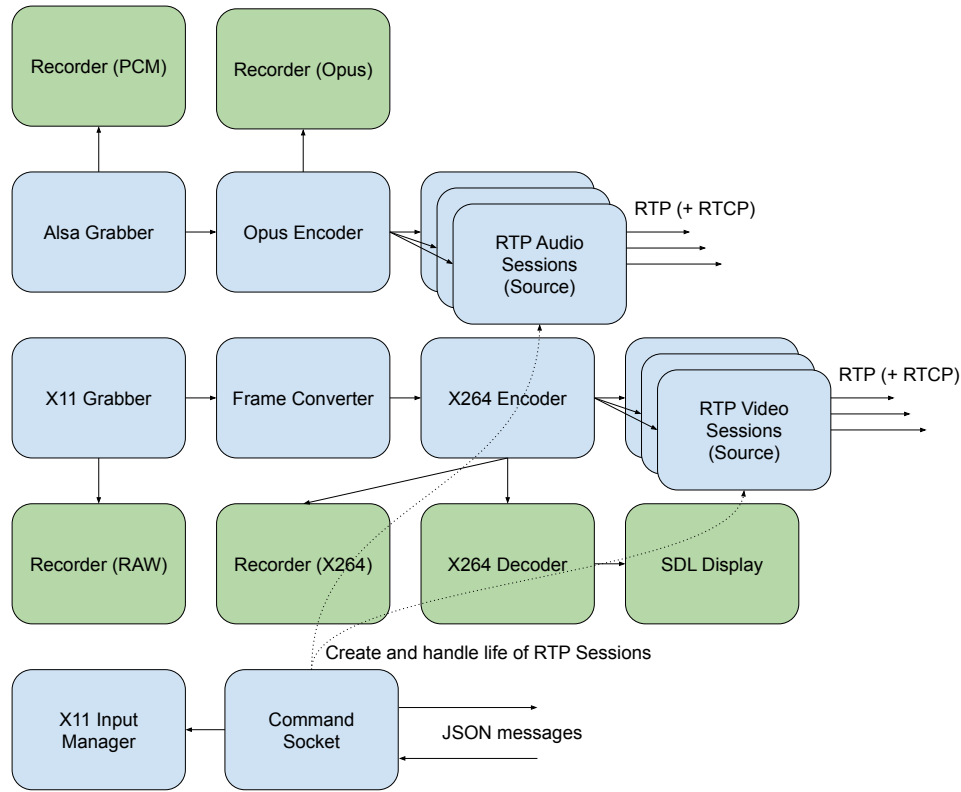


Figure 3.3: Current high-level design of the server side

AVPacket and AVFrame objects since we use FFmpeg libraries. The main communication channel is handled by the "command socket" module, which receives user inputs but also any request made by the users. In fact, the server is built in a way so that it can send data to multiple users since the added complexity from one to many is negligible. On user request, the "command socket" can spawn new RTP sessions and it is this module that will handle the life of these sessions based on a heartbeat logic. When user inputs are received, it will forward them to a specific module that send fake events to the X11 server thanks to an extension named XTEST (needed because sending XEvent directly is considered as a flaw and most applications ignore them, it is still possible that some programs ignore the ones we send). A word about the codecs used: for video we currently use X264 for simplicity but X265, VP9 (or AV1 if relevant) can be used instead with simple changes; for audio we decide to use Opus, a very efficient codec that can include in-band FEC (error correction for the previous packet only). For video we expect to fully use hardware acceleration but currently only the encoding is performed on GPU but we expect to add frame conversion and frame-buffer grabbing as future works to decrease latency. Other future works for this side include security improvements since we use RTP and UDP for communications, congestion control with, for example, the usage of SCReAM<sup>6</sup> and dynamic bitrate adaptation based on network conditions.

The client-side high-level architecture is depicted in Figure 3.4. We can see that it is simpler than the server-side. It also has some optional modules that have the same purpose as the server ones. In fact, the pipeline logic is very straightforward: packets received via the RTP sessions are decoded and played in real-time in a SDL window. Currently no mechanism is deployed to avoid artifacts due to packet loss. "SDL Display" module is also tasked to listen any event reported by the SDL and extract those related to user inputs. It will generate a JSON string if needed every 10ms (arbitrarily defined) to report the state of the keyboard and mouse (some down keys and buttons are reported periodically to maintain their state on the server side since it has an auto release logic). Most of the keys are correctly bound but some minor keys are still not well handled (mapping between SDL and X11 are not the same). The "command socket" module is then tasked to send the messages. It can also send specific messages on its own like heartbeat when nothing is sent for a given time. The client also use hardware acceleration (tested with Intel GPU) even if the task is not very intensive (a drawback is that we still need a copy

<sup>6</sup><https://github.com/EricssonResearch/scream>

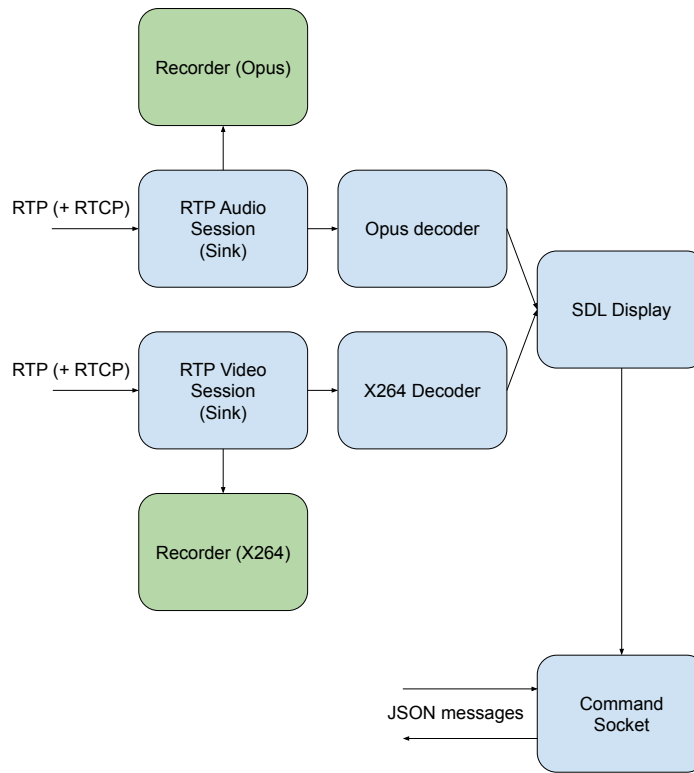


Figure 3.4: Current high-level design of the client side

from GPU to system memory but might be skipped). Client side has not much to do as future work, there is a bug with audio to solve and to add some jitter buffers but not much more at the time being.

In a global view, the two programs are multi-threaded since most modules we introduced execute their own thread(s) and have decent performances. The main logic is mostly finished but some details will still take some time to resolve. Some improvements or new features are also possible but not of prime importance like WebRTC support or a more "cloud platform"-ish service with the addition of third party programs and the usage of virtualization technologies to share hardware resources on the server between multiple clients.

### 3.4 Global Testbed

A 5G testbed based on a Software-Defined Radio (SDR) board, open source 5G core, integrating P4-based switches and using the MMT monitoring/security framework is currently being set up. This testbed will be continuously updated during the MOSAICO project with new components developed such as the L4S solution, security modules, cloud gaming servers, etc. The testbed will be used to evaluate different scenarios and key performance indicators (KPI) defined during the project. The monitoring solution will allow measuring the quantitative KPIs, allowing to determine if they are respected or if further research efforts need to be made to eliminate any bottlenecks. The testbed will also be used to determine the efficiency of the components to detect and react to security breaches.

#### 3.4.1 The 5G-in-a-Box platform

The testbed relies on the 5G-in-a-Box platform, currently supporting 4G LTE, 5G non-stand alone and lately 5G stand alone, and open source network cores or one commercialised by Montimage and CumuCore. It is a ready-to-use appliance allowing to create a full end-to-end 4G/5G network in 5 minutes.

The overall architecture of the 5G-in-a-Box platform is depicted in Figure 3.5. It basically consists of 3 main building blocks: Radio Access Network (RAN), Core network (4G EPC or 5G Core) and the Montimage Monitoring Tool (MMT).

Once deployed, the testbed platform creates a 4G/5G network allowing commercial off the shelf (COTS) User Equipment (UEs) to connect. After being successfully attached, the UEs have access to

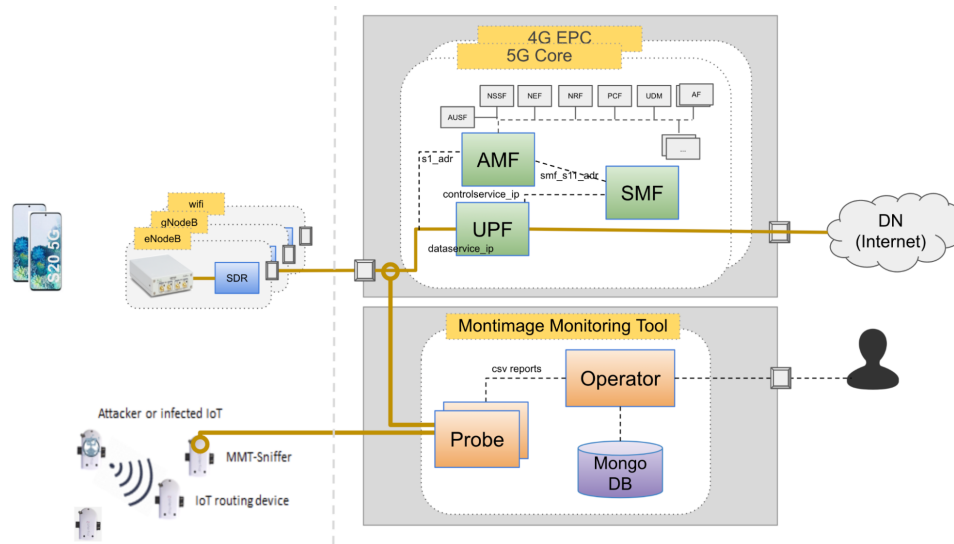


Figure 3.5: General architecture of 4G/5G-in-a-Box components

the IP services in DN or from the Internet through browsers, Web applications, Cloud gaming, VoIP, etc. All the traffic between the RAN and the Core is captured and analysed in real-time by MMT to ensure that the defined security properties and Service Level Agreements (SLAs) are respected. MMT supports automated decision and reaction in the case an anomaly is detected.

The RAN is constructed by using open source SDR. Currently we use 2 front ends: Ettus USRP B210 connecting via USB 3.0 and Ettus USRP X310 connecting via 10Gbps Ethernet; and two eNodeB softwares: OpenAirInterface (3GPP LTE Rel-10/12 PHY layer / 3GPP NR Rel-15 layer) and srsLTE (3GPP LTE Rel-10). A Non-StandAlone (NSA) version gNodeB of OpenAirInterface has also been tested to create a 5G NSA network. The SA gNodeB version is currently being installed and tested in the testbed.

The EPC is initially installed using CumuCore's vEPC with whom Montimage has established a partnership. Many other Core network open source solutions have also been installed and tested in the testbed, such as, openair-cn (3GPP Rel-10), nextEPC (3GPP Rel-13), free5Gc (3GPP Rel-15), and open5Gs (3GPP Rel-16). 5G RAN can also be simulated using UERANSim (3GPP Rel-15), which is an open source simulator of gNodeB and UEs. It allows quickly deploying an end-to-end 5G prototype solution without hardware to test 5G Network Functions (NFs).

MMT is a network traffic monitoring and security analysis framework. Its probes can be located between the RAN and the EPC/5Gcore to monitor network traffic of both the data and control planes. It uses a combination of Deep Packet Inspection (DPI), statistics and Machine Learning techniques to decode S1AP traffic between RAN and EPC/5Gcore. Consequently, it is able to collect network statistics, such as, QoS Class Identifier, statistics per UE, dynamically update the topology; and, information concerning the configuration of eNBs/gNBs, EPC's and 5Gcore's components, and UEs. It also integrates a security analysis engine using rule-based and Machine Learning to detect abnormal behaviour in the network. It allows easily experimenting attack scenarios to demonstrate the effectiveness of the security mechanisms. For this an open source solution has been developed by Montimage called 5Greplay ([HTTPS://www.5Greplay.org](https://www.5Greplay.org)). MMT also supports customised dashboards for defining and viewing new collected statistics and notifications.

The 5G-in-a-Box solution enables a portable, ready-to-use, zero-touch deployment and management for both experimenting and small-scale deployment of end-to-end 4G/5G networks. It is available as a software package or an appliance allowing to quickly deploy a mobile network.

### 3.4.2 Evolutions of the testbed

MOSAICO advocates the use of programmable of data plane, such as P4. This capability will be developed during the MOSAICO project and after be integrated into the testbed as programmable P4-based nodes between the RAN and the Core network. It will allow early detecting and preventing security issues, such as, (D)DoS attacks and deploying lightweight mitigation mechanisms, for example, acting as a firewall to block malicious traffic sources. Different network congestion control algorithms can also

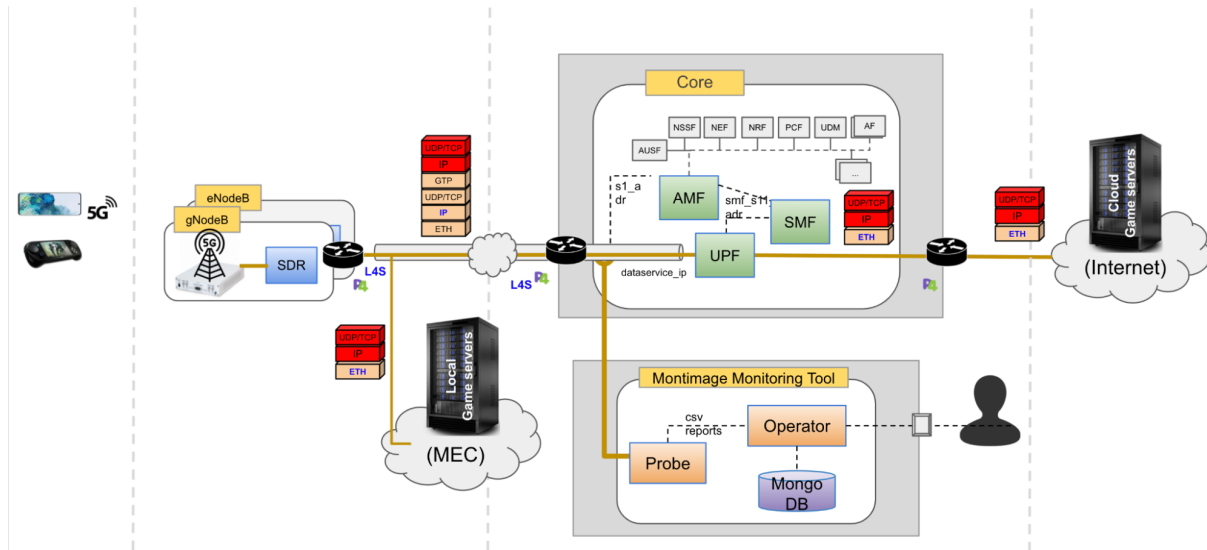


Figure 3.6: P4-enabled programmability of data plane

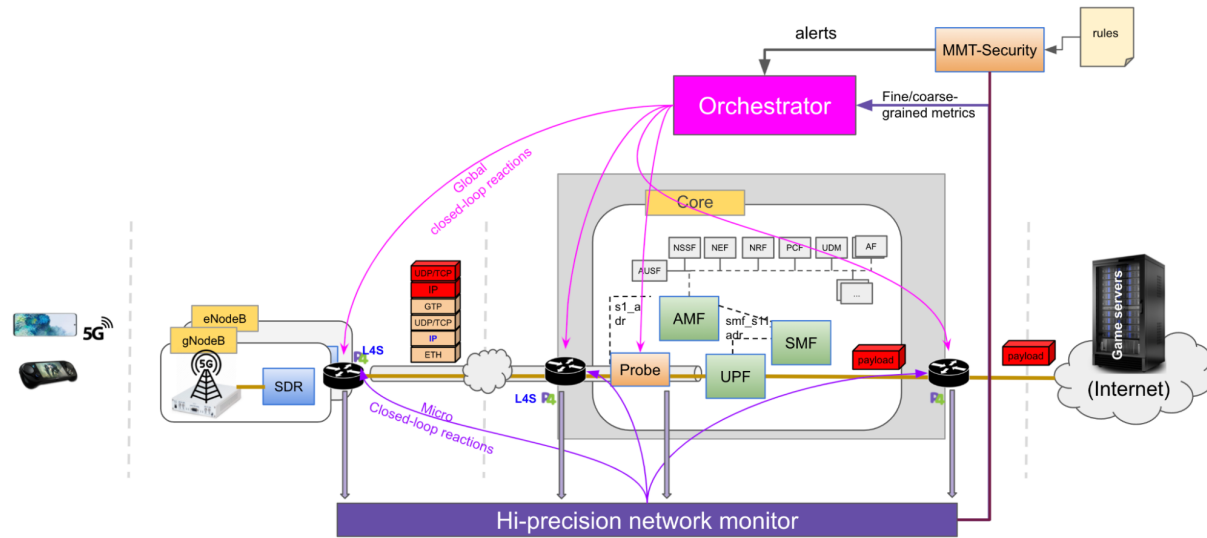


Figure 3.7: High-precision monitoring and closed-loop reactions

be implemented inside each router to ensure low latency and low loss network performance. The most important role of the routers is to allow users to reroute data plane traffic on-the-fly, without restarting the routers. This capability can be used to enable Mobile Edge Computing (MEC) to reroute traffic used by MEC services without passing through the Core network, as shown in Figure 3.6, thus increasing the performance and reducing the latency.

The high-precision monitoring and closed-loop model is another capability that will be developed during the MOSAICO project and be integrated in the tesbed. A possible general architecture of the tesbed could then be as shown in Figure 3.7. The ability of reprogrammability of the P4 routers allows implementing a high-precision network monitoring framework, such as in-band telemetry (IBT), to collect fine-grained network metrics. The collected metrics can be used by the router itself to adapt its congestion algorithm to ensure network latency and avoid loss. This micro closed-loop reduces reaction time, thus it is suitable for real time reaction requirements. The high-precision network monitor can synthesize coarse-grained metrics from the fine-grained ones, and report them to MMT-Security and a network orchestrator (or an SDN controller). The former detects anomalies and the latter can decide and then reconfigure or redeploy Network Functions (NFs) to ensure end-to-end Service Level Objectives (SLOs) requirements concerning latency and loss.

## Chapter 4

# Conclusion and Perspectives

The present deliverable stands for the mid-term milestone of the ANR MOSAICO project. Split into two chapters, it allowed a careful survey of the project achievements in the first chapter, and ongoing and prospective work in the second. The concluding remarks we formulate are consequently as follows. First of all, moving from a best-effort Internet toward an architecture intrinsically supporting low-latency features, requires major changes on all the components which traditionally form a comprehensive networking solution, namely a data-plane, accompanied by a control and management planes. As such, the project had to face, over the last period, the strong challenge to explore all this areas to carefully review, select and benchmark the candidate components which are numerous and not exhibiting the same level of maturity. Furthermore, the use-case selection also appeared as a challenging point since, to date, low-latency services are obviously not available as ready-to-use too. This motivates the approach the project considered with (1) the selection of cloud gaming as a use-case since it exhibits low-latency requirements while being already deployed with the current best effort architecture of the Internet and (2) the implementation of several individual testbeds and the operation of several experimental campaigns to consolidate the ground of individual components. Substantially, the few take-away we formulate regarding the project achievements are as follows: (1) the project has a selected use-case with a deep understanding of its networking requirements, (2) data-plane components are also selected, implemented on a programmable architecture and ready for an integration in a global architecture; (3) abnormal traffic situations have been identified and carefully reviewed to guide the design of troubleshooting and security components; (4) a theoretical design of the orchestration is also completed and ready to move toward a practical integration, and (5) monitoring and closed-loop issues have been studied to guide further developments in this area. Consequently, the second half of the project will essentially gather this set of individual components and contributions into the global MOSAICO low-latency architecture. If the project is currently disseminating and submitting these first part achievements in selected high-venue scientific journal and conferences, this second half will still induce further designs (e.g. detection algorithm of DoS attacks in low-latency, or a heuristic method for the timely resolution of the orchestration model), implementations (e.g. an in-band telemetry solution for MMT) and eventually integration in the global MOSAICO testbed.



# Bibliography

- [1] [Online; last accessed 10/2021]. 2012. URL: <https://github.com/keithw/multisend/blob/master/sender/saturatr.cc>.
- [2] [Online; last accessed 10/2021]. 2016. URL: <https://github.com/ravinet/mahimahi/tree/master/traces>.
- [3] [Online; last accessed 10/2021]. 2016. URL: <https://github.com/sdnfv/openNetVM>.
- [4] [Online; last accessed 10/2021]. 2021. URL: <https://tcpreplay.appneta.com/wiki/captures.html>.
- [5] B. Addis et al. “Virtual network functions placement and routing optimization”. In: *IEEE CloudNet*. 2015, pp. 171–177.
- [6] Dejene Boru Oljira et al. “Validating the Sharing Behavior and Latency Characteristics of the L4S Architecture”. In: *SIGCOMM Comput. Commun. Rev.* 50.2 (May 2020), pp. 37–44. ISSN: 0146-4833. DOI: [10.1145/3402413.3402419](https://doi.org/10.1145/3402413.3402419). URL: <https://doi-org.proxy.utt.fr/10.1145/3402413.3402419>.
- [7] B. Briscoe, M. Kuehlewind, and R. Scheffenegger. *More Accurate ECN Feedback in TCP*, draft-ietf-tcpm-accurate-ecn-11 (work in progress). [Online; last accessed 01/2021]. 2020. URL: <https://tools.ietf.org/html/draft-ietf-tcpm-accurate-ecn-13>.
- [8] B. Briscoe et al. “Implementing the ‘Prague Requirements’ for Low Latency Low Loss Scalable Throughput (L4S)”. In: *Netdev0x13* (2019).
- [9] Bob Briscoe et al. *Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture*. Internet-Draft draft-ietf-tsvwg-l4s-arch-10. Work in Progress. Internet Engineering Task Force, July 2021. 42 pp. URL: <https://datatracker.ietf.org/doc/html/draft-ietf-tsvwg-l4s-arch-10>.
- [10] Pierre-Olivier Brissaud et al. “Transparent and Service-Agnostic Monitoring of Encrypted Web Traffic”. In: *IEEE Transactions on Network and Service Management* 16.3 (Sept. 2019), pp. 842–856. DOI: [10.1109/TNSM.2019.2933155](https://doi.org/10.1109/TNSM.2019.2933155). URL: <https://hal.inria.fr/hal-02316644>.
- [11] Mihai Budiu and Chris Dodd. “The P416 Programming Language”. In: *SIGOPS Oper. Syst. Rev.* 51.1 (Sept. 2017), pp. 5–14. ISSN: 0163-5980. DOI: [10.1145/3139645.3139648](https://doi.org/10.1145/3139645.3139648). URL: <https://doi.org/10.1145/3139645.3139648>.
- [12] Koen De Schepper et al. “PI<sup>2</sup>: A Linearized AQM for Both Classic and Scalable TCP”. In: *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*. CoNEXT ’16. Irvine, California, USA: Association for Computing Machinery, 2016, pp. 105–119. ISBN: 9781450342926. DOI: [10.1145/2999572.2999578](https://doi.org/10.1145/2999572.2999578). URL: <https://doi.org/10.1145/2999572.2999578>.
- [13] Linux Software Foundation. *Data Plane Development Kit*. URL: <https://www.dpdk.org/> (visited on 05/07/2020).
- [14] Kelsey Hightower. *Kubernetes The Hard Way. Bootstrap Kubernetes the hard way on Google Cloud Platform. No scripts*. URL: <https://github.com/kelseyhightower/kubernetes-the-hard-way>.
- [15] Ralf Kundel et al. “P4-CoDel: Active Queue Management in Programmable Data Planes”. In: *IEEE Conference on Network Function Virtualization and Software Defined Networks, NFV-SDN 2018, Verona, Italy, November 27-29, 2018*. IEEE, 2018, pp. 1–4. DOI: [10.1109/NFV-SDN.2018.8725736](https://doi.org/10.1109/NFV-SDN.2018.8725736). URL: <https://doi.org/10.1109/NFV-SDN.2018.8725736>.
- [16] *L4S GitHub*. [Online; last accessed 10/2021]. 2021. URL: <https://github.com/L4STeam>.
- [17] M. C. Luizelli et al. “A fix-and-optimize approach for efficient and large scale virtual network function placement and chaining”. In: *Computer Communications* 102.C (2017), pp. 67–77.

- [18] Z. Meng et al. “MicroNF: An Efficient Framework for Enabling Modularized Service Chains in NFV”. In: *JSAC* 37.8 (2019), pp. 1851–1865.
- [19] A. Mouaci et al. “Virtual Network Functions Placement and Routing Problem: Path formulation”. In: *IFIP Networking*. 2020, pp. 55–63.
- [20] Ravi Netravali et al. “Mahimahi: Accurate Record-and-Replay for HTTP”. In: *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*. Ed. by Shan Lu and Erik Riedel. USENIX Association, 2015, pp. 417–429. URL: <https://www.usenix.org/conference/atc15/technical-session/presentation/netravali>.
- [21] *P4 Consortium*. [Online; last accessed 10/2021]. 2021. URL: <https://p4.org/>.
- [22] Koen De Schepper, Bob Briscoe, and Greg White. *DualQ Coupled AQMs for Low Latency, Low Loss and Scalable Throughput (L4S)*. Internet-Draft draft-ietf-tsvwg-aqm-dualq-coupled-14. Work in Progress. Internet Engineering Task Force, Mar. 2021. 54 pp. URL: <https://datatracker.ietf.org/doc/html/draft-ietf-tsvwg-aqm-dualq-coupled-14>.
- [23] Koen De Schepper, Olivier Tilmans, and Bob Briscoe. *Prague Congestion Control*. Internet-Draft draft-briscoe-iccrp-prague-congestion-control-00. Work in Progress. Internet Engineering Task Force, Mar. 2021. 30 pp. URL: <https://datatracker.ietf.org/doc/html/draft-briscoe-iccrp-prague-congestion-control-00>.
- [24] N Van Tu, J Hyun, and J W K Hong. “Towards onos-based sdn monitoring using in-band network telemetry”. In: *2017 19th Asia-Pacific Network ...* (2017). URL: <https://ieeexplore.ieee.org/abstract/document/8094182/>.
- [25] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. “Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks”. In: *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*. Ed. by Nick Feamster and Jeffrey C. Mogul. USENIX Association, 2013, pp. 459–471. URL: <https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/winstein>.
- [26] S. Xie, J. Ma, and J. Zhao. “FlexChain: Bridging Parallelism and Placement for Service Function Chains”. In: *TNSM* 18.1 (2021), pp. 195–208.
- [27] Wei Zhang et al. “OpenNetVM: A platform for high performance network service chains”. In: *Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization*. 2016, pp. 26–31.