POLITECNICO DI TORINO

Master Degree course in Ingegneria Informatica (Computer Engineering) - Computer Networks and Cloud Computing

Master Degree Thesis

Evaluation Of FPGA-based In-band Telemetry Methodologies For The Responsible Internet

Supervisors

Prof. Riccardo Sisto

Prof. Fulvio Valenza

Prof. Paola Grosso (UvA)

Dr. Anestis Dalgkitsis (UvA)

Candidate

Luca Cetino

Academic Year 2024-2025

Acknowledgements

I would like to express my sincere gratitude to my technical supervisor, Dr. Anestis Dalgkitsis, for his guidance throughout every step of this project and for his enthusiasm for my progresses, which ultimately made me feel like a valuable part of the Multiscale Networked System research group.

Many thanks to Professor Paola Grosso, who made this international exchange possible, providing me with an invaluable experience for both my engineering education and personal growth. Several times, her speech and interventions resulted captivating, inspiring me to dream of becoming a researcher at her level one day.

I also express my gratitude to my supervisor and co-supervisor, Professors Riccardo Sisto and Fulvio Valenza who believed in me and gave me the opportunity to follow my interests.

Last but certainly not least, I sincerely thank my loved ones: my family, for shaping the person I am today and for their unfailing support throughout my academic career; my life and adventure companion, Sara, whose bright positivity always brings light to my eyes and heart; and my closest friends, Eugenio and Alessandro, who have grown up with me and given me priceless memories over the years.

I owe you all so much.

Abstract

As digital infrastructures become increasingly critical to society, concerns over their trustworthiness and reliability have gained attention among European policymakers. The concept of Responsible Internet emerges as the next pivotal phase in their evolution, underlining the necessity of transparency in networks granting both critical service providers and everyday users the ability to know where the data they inject into the network are routed and which chain of providers handles them. To achieve such a goal and contribute to dismantling the *black box* nature of networks, we explored In-band Network Telemetry, a framework for collecting fine-grained metadata about the network's state with low and fixed transmission overhead. This technique relies on programmable data planes supported by modern yet uncommon network devices, such as the FPGA-based SmartNICs. It modifies packet processing by embedding telemetry operations into the forwarding plane, thus enabling telemetry data collection without the need for network sampling. In this thesis, our focus was on developing data plane telemetry solutions utilizing the P4 programming language, which allows for the flexible programming of network devices, such as FPGAs, to enable custom packet processing and real-time telemetry data collection directly in the forwarding plane.

Experimental validation was conducted within the FABRIC International Testbed, which enables research at scale across scientific domains. This infrastructure supported the deployment of the experimental programmable topology and allowed to conduct tests to assess the ability of the telemetry system to operate efficiently while introducing a low and negligible processing overhead compared to traditional packet processing, with a particular focus on impact on latency and packet loss. This work contributes to bridging the gap between network transparency and practical deployment, leveraging cutting-edge testbed environments and paving the way for more transparent and accountable infrastructures.

Contents

Chapter 1

Introduction

The Internet is the technology that evolved into the backbone of modern communication. Over the decades it became ubiquitous seamlessly interconnecting people and services with unprecedented performances. Its scale has grown exponentially in terms of interconnected devices and the reach of its infrastructure. From its conception, this evolution has influenced the complexity of computer networks, which face scalability challenges: the explosive growth in the number of connected devices, the rise of cloud computing and the emerging needs of ultra-low-latency applications push network infrastructures to their limits. The digitalization of services which heavily rely on interconnected systems has made the Internet a critical infrastructure for our society.

The inspiration for this thesis originates from the observation that, despite the diffusion and evolution of Internet technology over the past decades, it still lacks important properties often overlooked by everyday users and other stakeholders. It is taken for granted as the technology that supports communications and serves as the foundation for applications and services, an achievement that represents a remarkable success from a *plug-and-play design* perspective. On the other hand, it is the very entity to which we entrust the management of sensitive data and critical services, despite having little insight into how it operates and fulfils its duty of delivering packets, much like what is commonly referred to as a *black box.* The Internet is not secure by design. It also was not conceived to be accountable, controllable or, most importantly, transparent in how it handles data flows. These limitations have driven the need for advanced tools, yet there remains room for improvement.

The research community proposed the concept of Knowledge Defined Networking (KDN) with the idea of embedding AI and cognitive systems into the so-called *Knowledge Plane* of a network [**?**]. The knowledge plane's purpose is to construct an advanced model of the entire system, resulting in an essentially autonomous entity that understands what a network is, its purpose (*how* and *why* data must flow through it) and its users. This approach aimed to achieve a self-configurable network based on a set of high-level instructions, a network capable of automatically recognizing changes and issues, as well as implementing or suggesting possible solutions. However, KDN is not the only approach that heavily relies on knowledge about the network's state. Extracting information from the network is a fundamental aspect of implementing higher-level management frameworks.

From a different perspective, the concept of Responsible Internet has emerged with a more user-centred focus, aiming to establish a global infrastructure that enforces mechanisms to enhance transparency, accountability and controllability [?]. A key challenge in this vision is gaining insights into how data flows traverse a network or a chain of networks while simultaneously empowering users to choose their dependencies—that is, the chain of operators handling their data flows, including ISPs, DNS providers, and cloud services—when relying on the Internet infrastructure. Such a responsible infrastructure would also enable the verification and enforcement of compliance with policies and other constraints expressed by critical service providers (i.e. healthcare system, the transportation industry and other strategic sectors interconnected through the Internet) and policymakers. At the same time, it would help dismantle the *black box* nature of today's networks, providing greater transparency and control.

From a technical perspective, the responsible Internet's architecture introduces a Network Inspection Plane (NIP) and a Network Control Plane (NCP). The NIP is a distributed system designed to observe, analyze, and describe the data-carrying network with a particular focus on properties such as the jurisdiction of its providers and the security measures they implement. The NCP, on the other hand, serves as a sub-system that enables users to define and express preferences regarding how their data should be handled and by whom while in transit through the network. Hesselman et al. [?] designed these components in alignment with the three high-level objectives that structure their work: transparency and accountability are addressed by the NIP through the network description it provides, while controllability is the primary goal of the NCP, allowing users to specify how they want their traffic to be managed. This distinction in roles highlights how the NIP relies on extracting network insights using different techniques, to generate and share a transparent network description. Defined as a "machine-readable specification of the properties and relations of a group of interrelated network operators", including an operator description, this network description is made available to users. To achieve this, network telemetry functions have been identified as a viable solution and serve as one of the two proposed data sources for the NIP, alongside independent observers.

The introduction of the Responsible Internet concept has opened multiple research directions and provides the context in which this thesis has been developed. In particular, the focus is on the data extraction methods to analyse the network from different perspectives and for various objectives such as providing input for the NIP.

1.1 Existing Network Telemetry Techniques

Network telemetry encompasses a set of techniques designed to offer visibility into network activity and performance. A subset of this vast field focuses on data flow tracing, while telemetry can also be applied for monitoring network performance and health status. Network telemetry is implemented through a wide range of tools, commonly referred to as Network Management Systems (NMSs), which include:

- SNMP (Simple Network Management Protocol), a widely adopted standard for managing network devices and monitoring their health using a structured database of management objects called Management Information Base (MIB).
- NetFlow/IPFIX, a protocol introduced by Cisco in 1996 to analyze and monitor IP traffic while providing network traffic models for bandwidth optimization. It does not provide packet-level granularity.
- sFlow (sampled flow), a layer 2 packet sampler for network analysis and resource optimization. It allows the export of truncated packets captured at a low level, along with interface counters.

Despite being widely used in the industry for network resource optimization and remote equipment management, traditional NMSs have several limitations, particularly in real-time, fine-grained data collection.

1. The periodic polling/sampling introduces a stringent trade-off between the overhead (caused by high sampling frequencies) and the resulting update latency.
2. As networks expand and their topology becomes more complex, monitoring systems face scalability and overhead challenges, especially as the volume of the analyzed traffic increases. Implementing a fully distributed, global-scale internet monitoring system remains unfeasible.
3. Due to their different design goals, traditional NMSs may lack the granularity required to trace single data flows, which is essential for Responsible Internet observability.

4. Above all, they typically require intervention from the control (or management, in the case of SNMP) plane of the network to export telemetry data.

These constraints highlight the need for an alternative solution to provide more accurate and scalable data for the NIP. In response to these challenges, the next section introduces an alternative telemetry approach designed to enhance observability while maintaining efficiency and scalability.

1.1.1 A Different Approach: In-band Network Telemetry

In-band Network Telemetry (INT) is an innovative and relatively recent framework which aims to collect real-time, detailed telemetry information, enabling network insights with a low and fixed processing overhead. It provides fine-grained visibility down to a per-packet level and allows the extraction of information as

- **Path Trace**: the path traversed by a packet or a flow of packets
- **Network State**: at each hop, a snapshot of the local network conditions experienced by the packet can be collected
- **Contributions to the State**: information about other flows that have influenced the observed network state can be included

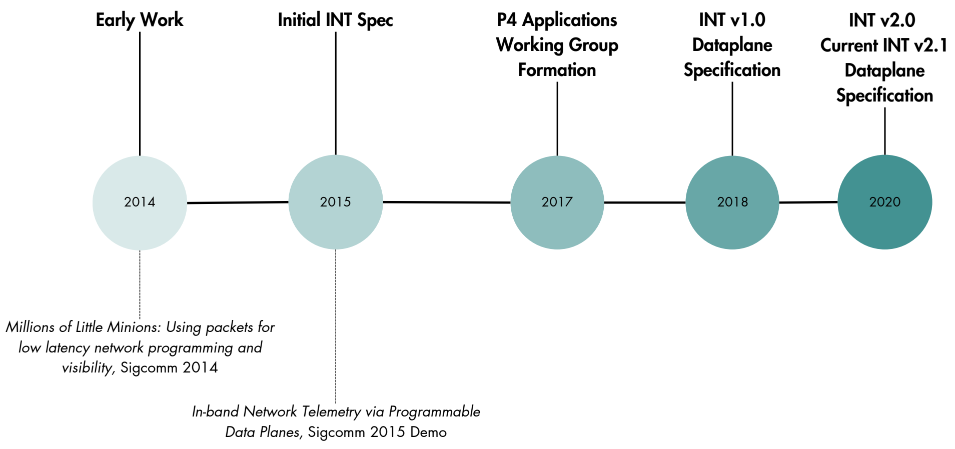

Early Work
Initial INT Spec
P4 Applications Working Group Formation
INT v1.0 Dataplane Specification
INT v2.0 Current INT v2.1 Dataplane Specification
2014
2015
2017
2018
2020
Millions of Little Minions: Using packets for low latency network programming and visibility, Sigcomm 2014
In-band Network Telemetry via Programmable Data Planes, Sigcomm 2015 Demo

Figure 1.1. Highlights In INT Framework Evolution - Image Reproduced from [**?**]

The in-band nature of the telemetry refers to the insertion of metadata and values within the data-carrying packets that traverse the network as dictated forwarding operations, even though three different modes of operation have been identified within this framework (for this and other detailed aspects, see the dedicated section 3.1). By embedding telemetry information directly into the data packets, INT eliminates the need for an external monitoring channel and operates independently of the control and management plane, significantly enhancing the scalability of the approach. INT also allows the application's endpoints to remain completely agnostic to the additional operations performed during packet forwarding; this is crucial for enabling a gradual deployment and adoption by network operators, minimizing the impact on the existing infrastructures and the standard networking functions.

As shown in the timeline in Figure 1.1, which outlines the key milestones in the evolution of the framework, the development of INT is closely tied to the P4 Applications Working Group. This organization released the first INT specification 2015 and continued to maintain and update it (see chapter 3). This close relationship stems from a fundamental architectural choice: INT operations are executed directly within the data plane of network devices, requiring the ability to customize its behaviour and implement ad hoc operations. This is where P4, a domain-specific language for programming packet processors (data plane functionalities) becomes essential.

1.2 Programmability Flavors in Networking

1.2.1 Bottom Up Design

Network programmability is a paradigm that emerged in the context of high-complexity network monitoring and management. It proposes using abstractions from the underlying infrastructure to enable automation and dynamicity in network configurations, prioritizing efficiency. This approach is often referred to as Software Defined Networking (SDN). SDN enables the centralization of the network "intelligence" into a logical, software-based component called a controller, decoupling the forwarding process of network packets (the *data plane*) from the routing process and the network control (the *control plane*). In this distinction between the two architectural components of a network, the data plane executes fixed and relatively simple functions, while protocol instances and auxiliary functions that complement data forwarding are assigned to the flexible control plane. The immediate advantage of this approach is the enhanced programmability of network control, allowing administrators to optimize, secure, and efficiently manage network resources. while SDN offers a powerful and flexible architecture for confined networks (LANs), it has the limitation of leaving the data plane fixed and abstracted from the manager's point of view.

1.2.2 Top Down Design

A different approach to enriching networks with programmability emerged with the introduction of programmable hardware complemented by domain-specific languages such as P4. These technologies allow changes to the way the data plane behaves, based on the custom functions expressed in the code. This opened the door to

- The introduction of new features (e.g. new protocols)
- The reduction of complexity, as unused and legacy protocols can be removed
- Greater efficiency in resource usage, with a flexible use of tables
- Enhanced visibility thanks to new diagnostic and telemetry techniques, as the one inspected in this thesis
- A software-driven approach with all its benefits, including rapid design cycle, fast innovation, and the ability to fix data plane bugs in the field

This level of flexibility in the network is what enables the implementation of in-band network telemetry on hardware, capable of meeting the stringent performance requirements imposed by modern network speeds. Programmable network devices include:

- **Flexible Match+Action switch ASICs** (e.g. Intel Flexpipe, Cisco Doppler, Cavium (Xpliant), Barefoot Tofino and others)
- **DPU** (Data Processing Unit), IPU (Infrastructure Processing Unit) and NPU (Network Processing Unit); examples include devices from Pensando, NVIDIA, Netronome, and Intel.
- **FPGA** (Field Programmable Gate Arrays) which are discussed later in chapter 2. This category includes the device used in our INT experiments, manufactured by Xilinx, although other manufacturers exist, such as Altera.
- **traditional CPUs** (Control Processing Unit), with applications and tools such as Open vSwitch, eBPF, DPDK, and VPP.

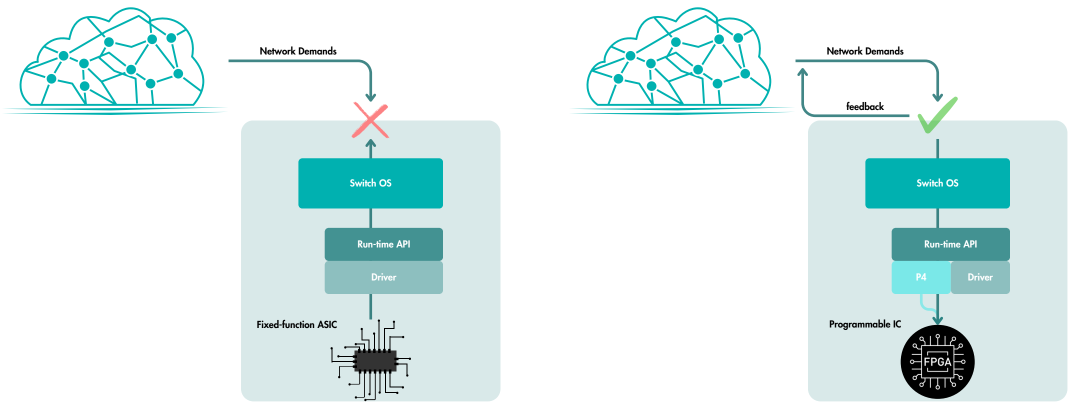


Figure 1.2. Network Programmability: Bottom-up vs. Top-down. Image Reproduced From [?]

1.3 Thesis Outline

The main contribution of this thesis is the implementation and evaluation of an In-band Network Telemetry solution on FPGA-based hardware. I designed and deployed a P4 application compliant with the P4.org INT specification and tested it within a research testbed. This work builds upon previous research by W. S. Petri [?], who inspected P4 programming and porting of P4 compiled artifacts on specific FPGA targets, outlining a foundational framework for implementing INT on FPGAs. While Petri demonstrated how an FPGA could support basic operations, this study extends this work by expanding the network topology and developing a more comprehensive P4 application aligned with the P4.org INT specification. In conclusion of this study, performance evaluations to assess the time overhead introduced by INT-specific operations have been conducted.

Beyond the implementation, this thesis explores a set of techniques and tools for extracting data flow tracing along with other real-time network insights. The selected framework, enables the collection and reporting of such information by embedding telemetry operations directly within the data plane. This approach allows state information and network metadata to be piggybacked onto the actual traffic traversing the network, eliminating the need for a separate monitoring channel. The P4 INT application was deployed on two FPGAs, whose performances were evaluated in terms of time overhead and their ability to sustain high traffic rates.

This thesis ultimately aims to develop a proof of concept and demonstrate (INT) in action through an experimental approach. To achieve this, a network topology is deployed within the FABRIC testbed, a research infrastructure that enables controlled experimentation. Within this environment, we define an INT core composed of hardware-programmable switches, ensuring a practical evaluation of the proposed solution.

1.3.1 Thesis Structure

The following chapters are organized and structured as follows: chapter 2 provides a technical background on each component of the testbed as well as the tools used to design and build the INT proof of concept, chapter 3 discusses the INT application developed to run the experiments, which are extensively presented in chapter 4. I assessed the performance impact caused by the telemetry through two performance experiments, which are presented in chapter 5. Finally, chapter 6 draws conclusions. Useful resources referenced throughout the document are fully reported in chapter 7, which contains the Python and Bash scripts developed.

Chapter 2

Technical Background

This chapter provides the technical background necessary to replicate and potentially extend the experiments conducted, which are later presented in chapter 4 and chapter 5. It outlines the process of designing and deploying an INT domain using programmable hardware to implement the data plane of network devices.

2.1 Field Programmable Gate Arrays and SmartNICs

A Field-Programmable Gate Array is a type of configurable integrated circuit which can be programmed several times after the manufacturing process. It is part of the wider class of *programmable logic devices* (PLDs) and it is essentially composed of several programmable hardware blocks disposed in an array, with a connecting grid. Such blocks can act as simple logic gates (AND, XOR) or even implement complex combinational functions. Some memory elements are also included in the logic blocks. The hardware configuration of an FPGA is written in a Hardware Description Language (HDL) an example is VHDL or Verilog, which are widely used also in traditional ASIC design.

These devices allow a re-definition of the hardware configuration, which translates into the capability of reprogramming the functions they implement. To understand why this feature is crucial for our application, there's the need to briefly present the traditional architecture on which a network device, and in general a network, is based. As shown in Figure 2.1, it can be divided into three main components:

- **Control Plane**: is the part of the architecture responsible for running complex tasks and algorithms. It's the implementation of the network protocols, once coded into algorithms, that usually run on general-purpose CPUs. It results in populating

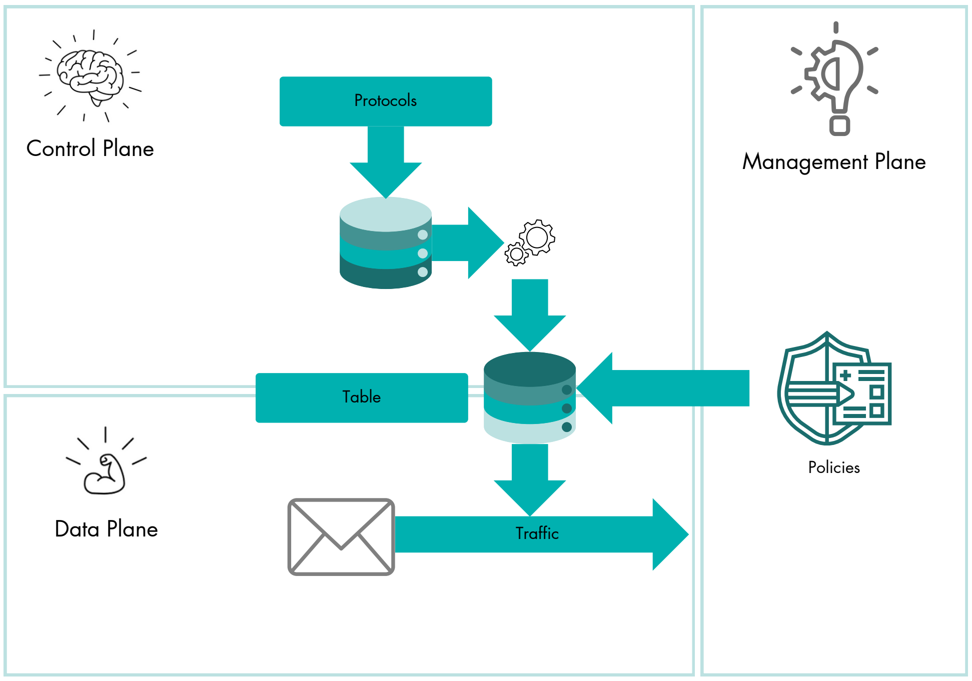


Figure 2.1. General Architecture of a Network Device

the routing/forwarding tables of the most common network devices, during a transient state of the network (e.g. when the topology is determined or changes). Its operations can be seen as conceptually articulated but executed once in a while, at low frequencies. It is also often referred to as the *slow path* a packet can take within the device.

- **Data Plane**: is the part of the architecture that will manage the majority of the incoming packets by repeating simple and fixed operations at a very high rate. Usually, it is implemented in hardware with ASICs (Application Specific Integrated Circuits, e.g. Longest Prefix Matching lookup in the routing table). Its low complexity and focus on performances made it gain the title of *fast path* within the architecture.

- **Management Plane**: is a section of the architecture which gets used for configuration and management purposes. It allows the enforcement of security policies as well as the monitoring, troubleshooting and configuring of a network device. It is usually implemented with a dedicated network interface and it is the component polled by the traditional Network Monitoring Systems to collect statistics and telemetry metadata.

The INT framework proposes embedding telemetry operations in the data plane of a device.

However, this approach is not feasible when considering a data plane which is implemented with an Application Specific Integrated Circuit ASIC, which is optimized for fixed and simple operations such as routing and forwarding packets. ASIC-based data planes are inherently rigid and lack the flexibility needed to support advanced packet processing tasks, such as telemetry, which go beyond the standard functions typically implemented in network hardware. As a result, ASICs do not provide native support for these operations. On the other hand, offloading telemetry processing to a different architectural component, such as the control plane, would require handling packets in the slow path, introducing significant processing overhead [**?**]. Given these considerations, the advantage of leveraging a programmable data plane, particularly one based on FPGA technology, becomes evident. The compromise it strikes between the high performance needed in the data plane and the flexibility and programmability required by complex processing in terms of packet manipulation perfectly fits our context. Programmability in network device data planes has already been introduced in existing SmartNICs (that is, smart Network Interface Cards). These network interfaces were initially designed with a general-purpose CPU to offload networking tasks from the main processor and, in some cases, included hardware accelerators for specific functions (e.g. compression, encryption) [**?**]. This evolution led to the development of FPGA-based SmartNICs, where the general-purpose CPU is replaced with a reconfigurable FPGA chip, as seen in our target device, the AMD Alveo U280.

2.1.1 AMD Alveo U280 Datacenter Accelerator SmartNIC

The hardware platform selected for our INT operations, the AMD Alveo U280 Datacenter Accelerator Card, is specifically designed to provide the flexibility required by the modern, dynamically evolving needs of datacenters; this adaptability represents the main strength as well as the key innovation compared to traditional networking equipment. It has a considerable amount of onboard hardware resources, with a particular emphasis on its FPGA (Field Programmable Gate Array) chip, which serves as the foundation of our implementation. The hardware specifications of the AU280 card are summarized in Table 2.1 [1]:

All the detailed technical characteristics of the board and its FPGA chip can be found in [**?**]. From the networking perspective, our focus is specifically on exploring the switching architecture of the FPGA of the SmartNIC. Understanding this architecture will later help us understand how network traffic interacts with the device's components and which logical blocks are responsible for processing and forwarding the packets.

[1]Data sourced from AMD portal: `https://docs.amd.com`

Specification	**Active Cooling Version**	**Passive Cooling Version**
Product SKU	A-U280-A32G-DEV-G	A-U280-P32G-PQ-G
Total electrical card load	215W	215W
Thermal cooling solution	Active	Passive
Weight	1187g	1130g
Network interface	2x QSFP28	
PCIe Interface	Gen3 x16, Gen4 x8, CCIX	
HBM2 total capacity	8 GB	
HBM2 total bandwidth	460 GB/s	
Look-up tables (LUTs)	1,304K	
Registers	2,607K	
DSP slices	9,024	
Block RAMs	2,016	
UltraRAMs	960	
DDR total capacity	32 GB	
DDR maximum data rate	2400 MT/s	
DDR total bandwidth	38 GB/s	

Table 2.1. AMD Alveo U280 card details

2.1.2 Programmable Hardware Switch

Figure 2.2 presents a schematic hardware block diagram for the programmable switching architecture implemented by the FPGA. The diagram illustrates the four ingress ports on the left and the respective egress on the right. These ports are referred to as *PF0*, *PF1*, *CMAC0* and *CMAC1*. The first two are Physical Functions which connect at high speed the card with the host (server) via the PCIe bus. This means that, from the host's perspective, the smartNIC is mapped as a PCIe device. The latter two ports are 100Gb Ethernet Medium Access Controllers (CMACs), they enable connections with external networks and implement the Layer 2 (Data Link Layer) functions of the OSI model for Ethernet interfaces.

Another crucial component is the *Smartnic 322MHz App* block, which represents the FPGA fabric programmed to implement a custom packet processor. A set of interfaces and FIFO queues interconnect the ingress/egress stages with this processor. The two *AXI* (Advanced eXtensible Interface) *Switches* dynamically adjust traffic routing within the device based on the SmartNIC's configuration, managing both ingress and egress paths. Additionally, the SmartNIC provides integrated probe counters, which are extremely useful for tracking traffic statistics, including the number of packets received or sent through each port, the flow of packets towards the internal packet processor, and metrics such as packet errors and dropped packets.

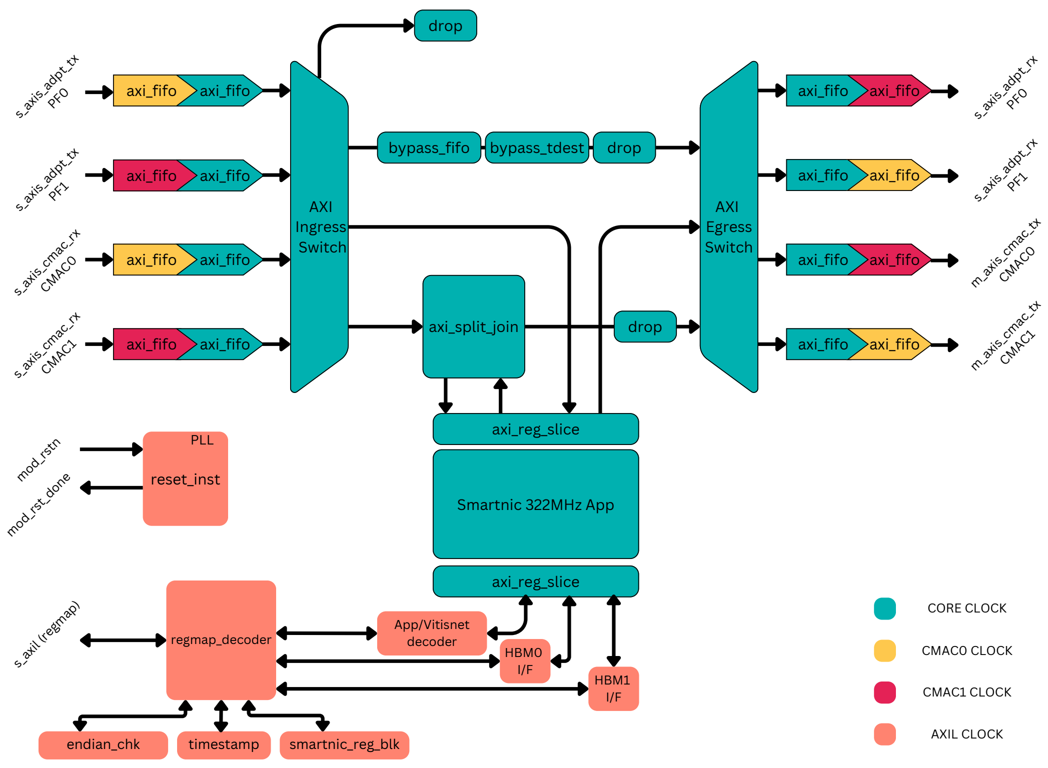


Figure 2.2. Programmable Switch Architecture - Scheme Reproduced by [?]

2.2 P4: A Language for "Speaking" To Networks

The previous section 2.1 presented the FPGA technology and its role in enabling the programmability of high-performance data planes for network telemetry functions, with focus on the fact that an FPGA gets programmed through a Hardware Description Language. In this section, it is introduced the actual language used for programming the packet processor, as well as the tools needed to translate the source code into a bitfile for programming the FPGAs.

P4 (Programming Protocol-independent Packet Processors) is a domain-specific programming language, specifically designed for defining the behaviour of packet processors such as switches, routers, smartNICs, load balancers, packet filters and many others. Unlike generic programming languages (*C*, *Rust*, ...) it offers a set of constructs and abstractions suited for expressing networking functions. It allows the developers to describe the programmable data planes that target various technologies, including CPUs, FPGAs and NPUs [?]. By design, P4 is not dependent on the target platform, to ensure the maximum portability of the source code. Its modular approach requires some additional components to function, which can be different for the various target platforms. Figure 2.3 is a graphical representation of the modular components involved in the development and deployment of P4 code on a target device.

- **P4 Language** and **Core Library**: are the common modules provided by the community P4.org. They define the common abstractions and the constructs optimized for the specific class of problems (the domain of reference: networking), independently from the target platform.
- **Architecture Definition** and **Extern Libraries**: these components are vendor-supplied and specific to a single target platform. The architecture definition is the abstract model of the target device (*i.e.* the kind of device that will be programmed: a switch, a router, a NIC and so on), each model defines different device components (parser, match-action pipelines, deparser). Extern libraries are architecture-specific software modules that allow us to interact with hardware functionalities specific of each device.

Also, the P4.org community standardized some Architecture Definitions to abstract different possible targets from the low-level hardware implementation, in order to simplify the development process. Such standard P4 architectures are the **PSA** (Portable Switch Architecture) which refers to a generic switching device; the **PNA** (Portable NIC Architecture) is an abstraction of a generic network interface card; **V1Model** is the p4 architecture referred to software-emulated switching devices which usually run on generic purpose CPUs (as the BMv2 switch). When it comes to the target of our interest, the FPGA-based SmartNIC from Alveo, the architecture for abstracting such hardware is provided by the vendor: Xilinx. This architecture model presents several programmable blocks and fixed functions and is described in the following dedicated subsection 2.2.1.

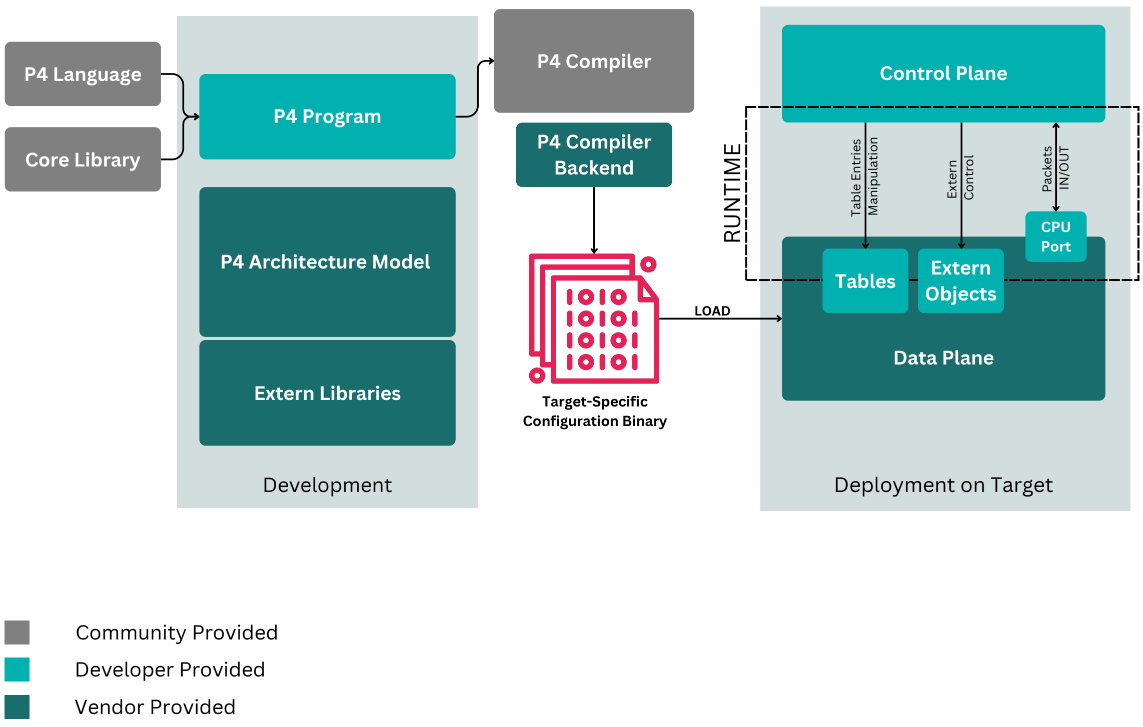


Figure 2.3. P4 Workflow and Its Components, Image Reproduced From [?]

2.2.1 XSA - Xilinx Switch Architecture for P4

The elements defined in a P4 program which targets an Alveo U280 board are similar to the ones from other standard architecture models, even though they are specifically adapted for this target device. Such elements are later mapped to the engines implemented on the programmable hardware, during the compilation process (see subsection 2.2.2) and are the following:

- **Parser**: this is the first stage of the processing pipeline and implements the parsing of the incoming packet. This happens with the construction of a parse graph, the headers and their fields within each packet are recognized based on what is expressed in the program. The parser is described as a finite-state machine.
- **Match-Action Pipeline**: this component holds the core part of the program, where the actual processing (once the packet is received and already parsed) takes place. It is used to describe the forwarding or processing tables later populated with control plane-defined entries. Here are also defined the Actions to be taken by the packet

processor based on certain fields and the respective values written in the packet.

- **Deparser**: the last stage allows us to define the packet structure as it will look on the egress link.
- **Pipeline Definition**: this represents the main pipeline definition and puts pieces together expressing how the different components of the architecture are interconnected, this directly reflects on which stages the packet gets processed by and in which order. A graphical representation of the main pipeline can be found in Figure 2.4

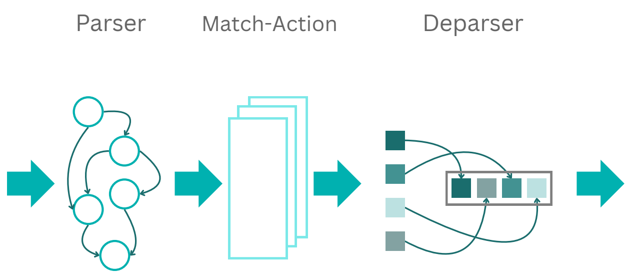


Figure 2.4. XSA Architecture Definition

2.2.2 Compiler and Synthesizer

Another set of crucial components needed before being able to deploy a P4 application on a specific target, as depicted in Figure 2.3, is a P4 compiler backend and a set of target-specific libraries. In this case, they are provided by the target vendor, Xilinx.

The tools that enable the translation from a P4 application to a target-specific configuration binary (or bitfile), are proprietary and need a license to be used. In the case of research and academic needs, evaluation licenses can be requested. For this work, evaluation licenses were provided by the University of Amsterdam. Xilinx disposes of a hardware-design suite comprehensive of two development tools: Vitis Networking P4 (VNP4) [?] and Vivado [?].

- **VNP4**: is a tool to convert high-level P4 code design requirements, expressed coherently to the XSA architecture model, into an AMD FPGA design solution. This design suite lets P4 developers leverage all the flexibility offered by the language in designing new data planes with easy and powerful constructs to design packet headers, data structures, and new processing implementations. A mapping is performed by VNP4 towards a custom data plane architecture composed of Parsing, Deparsing, Action and Look-up Engines. These components implement respectively the extraction of header information from the packets, the insertion/deletion/manipulation of packet data, the manipulation of metadata derived from the packet both locally or from other engines and the instantiation of memory search IP cores (Intelectual Properties) to manage tables and look-ups with optimized memory configurations. This tool ultimately abstracts the actual design at the hardware level with the already presented HDLs (VHDL, Verilog and others), and its output is a set of IP blocks implementing our design.

- **Vivado**: is a hardware design environment to create and implement digital circuits for Xilinx FPGAs. It works in synergy with VNP4 allowing us to import the IPs in a hardware design as pre-designed components. The following phases are the synthesis of a netlist for the FPGA followed by the mapping of such netlist on the physical FPGA resources. The process prepares a bitstream artifact ready to be loaded and deployed on the target.

This compiler and synthesizer are complex and professional tools for hardware design and engineering, with a steep learning curve. This is the reason why such tools were embedded in an even higher level framework, presented in the next section, which permitted us to concentrate on P4 development by automating the compilation, synthesis and implementation processes.

2.3 ESnet Framework

ESnet[2] is a specialized, high-speed network infrastructure operated by the U.S. Department of Energy, tailored to support the unique requirements of scientific research. It ensures fast and secure data exchange and enables researchers to efficiently analyze and share results, bridging the gap between experiments and discovery.

The ESnet SmartNIC Framework is a set of tools to simplify and improve SmartNIC-oriented operations; it is possible to divide the set of resources it disposes into the Development Workflow and the Deployment Workflow. It provides a complete workflow to

[2] `https://www.es.net/`

program Alveo cards using P4, seamlessly integrating vendor tools (see subsection 2.2.2) with various SmartNIC utilities. The framework is under active development, for this reason, dependencies and versions of the tools may vary over time.

2.3.1 Development Workflow

All the tools that contribute to the development of P4 applications, the testing (and debugging), the simulation and finally the compilation for porting them on Alveo FPGAs, are included in the **esnet-smartnic-hw** repository. This is a collection of components maintained independently as separate tools and included as submodules. The repository structure includes the following modules and tools:

- *Open NIC Shell*: an FPGA-based NIC shell forked from the original provided by Xilinx. It is designed to run on the Alveo family of boards and contains customizations for the ESnet SmartNIC Platform.
- *ESnet Smartnic Hardware*: a directory containing tools for the hardware design of the ESnet SmartNIC Platform.
- *ESnet Smartnic Firmware*: utilities for designing and packing the firmware design of the SmartNIC.
- *ESnet FPGA Library*: a directory supporting structured FPGA design methodology providing general-purpose components.
- *SVUnit*: an open-source System Verilog verification framework for FPGAs.
- *ESnet Regio*: a tool enabling the automation of the implementation of FPGA register map logic and software code.

This repository needs to be installed and all its submodules need to be initialized. Then, on the same machine, it's necessary to install Vivado (with VNP4 extension) and configure its runtime environment. A customization of the Makefile contained in the repository is required for setting appropriately the environment variables. Then it is possible to simply build the P4 design by executing the application Makefile. If everything succeeds a compressed archive containing the bitfile artifact will be generated. A complete tutorial to replicate these steps and fully understand the workflow is made available by the ESnet SmartNIC Team and can be found at [?] [?].

2.3.2 Deployment Workflow

This last phase of the workflow permits the deployment of the compiled and zipped bitfile (the hardware description artifact) on the FPGA card. It is composed of three essential Docker images, which interoperate to grant access to a powerful set of experimental tools to interact with the FPGA and hide the load and flash details managing the interaction with the card drivers.

- *xilinx-labtools-docker*: this docker image can be built starting from the homonymous GitHub repository. It is a crucial part of the setup as it will hold an instance of Vivado Lab, a non-licensed software whose functionalities are hugely reduced when compared to the full installation, but is still enough to manage the load of new programs on the FPGA.
- *smartnic-dpdk-docker*: this image, similarly to the previous one, can be built based on the respective GitHub repository and serves as an essential component for interacting with the FPGA at runtime once it is flashed with the hardware description bitfile, in particular granting access to DPDK (Data Plane Development Kit), an application for high-performance processing and efficient data plane operations.
- *esnet-smartnic-fw*: this image is bitfile dependent, which implies it needs to be rebuilt every time a new artifact is compiled starting from a P4 application. This image will be run as a Docker container disposing of the largest set of CLI tools to manage and interact with the FPGA once it is ready. They are later on referred to as ESnet CLI Tools.

2.4 FABRIC Testbed

FABRIC (FABRIC is Adaptive ProgrammaBle Research Infrastructure for Computer Science and Science Applications) [?] is an international research infrastructure that enables cutting-edge experimentation in many areas of computer science including the one of interest for this study, which is networking. The infrastructure is a distributed set of equipment at commercial collocation spaces, national labs and campuses. Its core resides in the U.S. and each of the 29 FABRIC sites has large amounts of computing and storage resources, interconnected by high-speed, dedicated optical links. It also connects to the Internet and recently, FABRIC Across Borders (FAB) extended the network with 4 additional nodes in Asia and Europe, among which, there is a node in the University of Amsterdam. Figure 2.5 shows the global network of the FABRIC research testbed, with the terabit core links highlighted in yellow and the 100G links displayed in blue. This particular testbed was selected because it suits the needs of our experiments: it allows us to test non-standard networking operations as INT, with custom network topologies. Still, it most importantly grants access to advanced hardware such as FPGA-based SmartNICs, in particular AMD/Xilinx Alveo U280 boards.

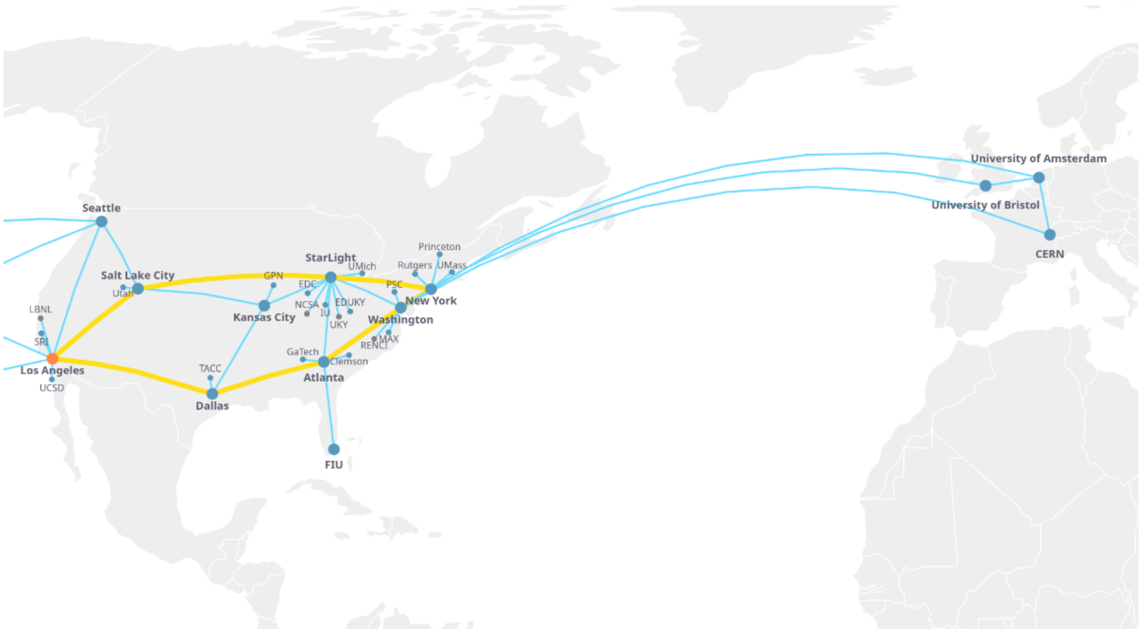


Figure 2.5. FABRIC Testbed - Image Sourced From [?]

2.4.1 FABlib API

The creation of an isolated, controlled test environment is managed through the Python APIs exposed by the FABRIC Testbed, FABlib. These are typically called using a Jupyter Notebook on FABRIC JupyterHub and make use of the following key concepts:

- **slice**: it is the container object describing a single network topology. It holds all the VMs and defines how they are interconnected. This object gets first created and then submitted to the orchestrator, on this later phase it is automatically validated and if the validation succeeds it is deployed by the FABRIC Orchestrator.

- **node**: it is a single Virtual Machine. It is added to a slice and configured with the image it is based on, the amount of computing or storage resources it has and the eventual special hardware it will have access to. The Virtual Machines will expose a management plane used by the experimenter to log in and download any software from the Internet. They have also the data plane which implements the experiment topology.

- **network**: it's the set of networking equipment used to interconnect the VMs on their data plane.
- **component**: it is a piece of special hardware (GPUs, Network Cards, NVMe drives, **FPGAs**) dedicated to a particular VM. Such components are attached to the PCIe bus of the servers which host the virtual machines and are exclusively dedicated to a node with the *device passthrough* technology which implies minimal VMM involvement.

Before the creation of a topology, the availability of resources in the physical sites must be checked using the *resource overview portal* of FABRIC. The FABlib APIs also include functions to execute commands on specific nodes, upload files on them and retrieve information: this ensures considerable flexibility and a great level of automation of the topology creation and management. Slices last for a default time interval of 24 hours, to ensure resources are freed up whenever they are not needed anymore for an experiment.

2.4.2 FABRIC Strengths and Limitations

This research environment, which allowed exploration of In-band Network Telemetry, provides a powerful and flexible testbed for networking experiments, offering access to customizable topologies and advanced hardware. The APIs exposed by FABRIC enable a good level of automation of the topology provisioning and deployment. Finally, the hosts composing the topology have the possibility to run custom scripts to realize non-standard network operations, which reduces the complexity of the INT implementation as it can be tested against specifically created network traffic. On the other hand, it also comes with several limitations to be considered while designing and executing experiments. Not every physical site of the FABRIC network has a SmartNIC and this translates sometimes into a resource shortage which can make it difficult to reserve some for an experiment. Moreover, if a slice needs more than a single FPGA, it will need to unfold across different physical sites, which can make the topology cover large geographical distances. This may not be ideal for experiments requiring low-latency communication between FPGAs, where the added delay from geographically distributed resources could negatively impact performance. However, this characteristic can be advantageous for experiments that aim to study wide-area network behaviours.

This chapter outlined the technological foundations on which the In-band Network Telemetry proof of concept is based. We explored the peculiarities of network devices to evict their limitations and the solutions offered by modern technologies (programmable data plane devices) to the structural limitations and requirements imposed by networking operations (mainly performances, difficult to scale with software emulations). A complete workflow to program FPGAs with P4 applications was proposed and finally, the testbed which hosted the experiments was presented. Figure 2.6 summarizes the technological embedding this work is based on and clarifies how the presented tools interact with each other. The following chapter will present the INT implementation deployed leveraging all these technologies.

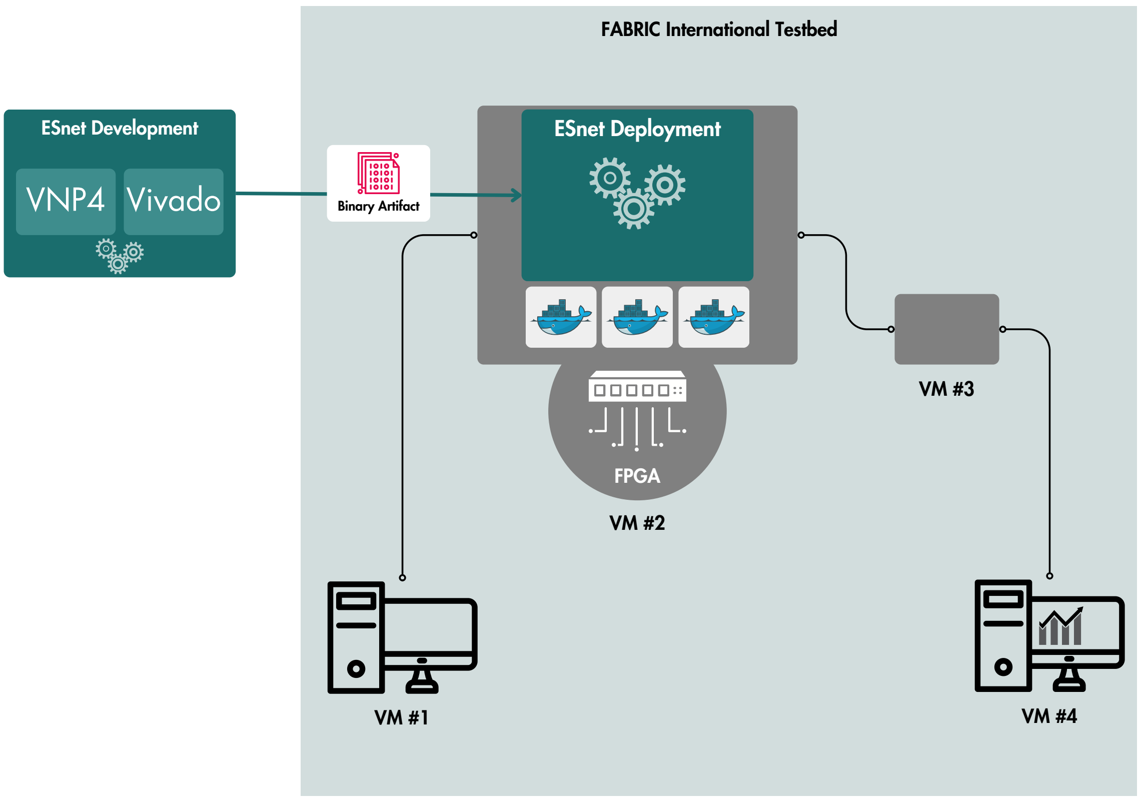


Figure 2.6. Technologies And Software Utilized

Chapter 3

In-band Network Telemetry Application

Telemetry enables real-time metrics collection from the compatible components of the network, also referred to as INT hops/devices. Telemetry information is collected by the devices' data planes, while the packets are processed with traditional networking operations. The approach is based on pushing such information into the headers of the packets that pass by. Such manipulation requires a specialized set of operations to be performed on the packets, which are not yet defined in any standard. New headers and their placement into a common packet must be defined. To address these problems, the P4.org community released an INT specification, which provided guidelines that can be followed when developing or testing an INT application.

The discussion in this chapter is based on the P4 INT Specification v2.1 [**?**]. The specification defines

- Modes of operation (INT-MD, INT-MX, INT-XD)
- Header structure
- Header location within the header stack of the packet
- Set of metadata each INT device could potentially export

Before diving into the technical definition presented in the specification, it is useful to define some terminology coherently with the document:

- **INT Source**: It is the trusted component that inserts INT headers into the packets it sends. This insertion can be based on the flows it routes and in this case, a Flow Watchlist is configured in this node.

- **Flow Watchlist**: It is a match-action table configured in the data plane of the devices, can recognize the flows based on a set of header fields and apply or insert INT instructions based on the configuration it holds.
- **INT Packet**: A packet containing an INT header. The three possible INT header types are defined below in section 3.1.
- **INT Node**: An INT-capable network device that regularly inserts, adds to, removes or processes INT instructions from INT headers. It can be a router, a switch, or a NIC.
- **INT Instruction**: Instructions to be executed by the INT Nodes. They indicate which INT data to export and can be either configured in a Flow Watchlist or carried in the INT header.
- **INT Transit Hop**: A trusted entity that extracts and exports telemetry data based on the instructions it executes. The data can be embedded into the packets processed or directly exported towards a Monitoring System, based on the mode of operation.
- **Monitoring System**: A trusted entity that collects the telemetry data received by different devices. It can be physically distributed but logically centralized.
- **INT Sink**: It's the counterpart of the INT Source, it extracts the INT Header to make the whole process transparent to traditional network devices and systems. It does not preclude the possibility of nested INT domains. Upon stripping INT headers, it decides whether to send or not the per-flow metadata collected to the monitoring system.
- **INT Domain**: Represents a set of interconnected and consistently configured INT nodes under the same administration. The edges of a domain should implement INT Source/Sink capabilities to prevent INT data from leaking out. The P4.org specification defines packet formats and device behaviour to guarantee interoperability between different manufactured devices.

3.1 Modes of Operation

Since its introduction, a large number of INT variations have been presented and have been developed in research environments and industry. The original one provided both INT instructions to be executed by network devices and INT data collected to be embedded in the packets traversing the network. The P4 specification for INT defines 3 different modes of operation for INT distinguishing them based on the grade of manipulation the packet experiences.

3.1.1 INT-XD (eXport Data)

No packet modification is required, as each hop exports the telemetry data from its data plane towards the monitoring system based on the matches configured in its internal Flow Watchlists. This means that when a packet is received and recognized to belong to a specific flow configured at one of these table entries, the corresponding INT instruction is executed and the data is exported. The packet leaves the INT hop unchanged.

3.1.2 INT-MX (eMbed instruct(X)ions)

Minor and fixed packet modification is required at the INT source, which is embedding into the packet header the instructions to be executed on the subsequent nodes. The other INT hops will then export their telemetry data to the monitoring system accordingly while forwarding the packet. The last INT hop, or INT Sink, has to remove the INT instructions header before forwarding it to the recipient. This mode of operation also supports the embedding of metadata from the source node into the Domain Specific Instruction field of the header.

3.1.3 INT-MD (eMbed Data)

The packet is modified the most in this case, as both the instructions and the metadata are embedded into the header. The instructions are inserted by the source node. While the packet proceeds in its path, each INT Transit node appends its telemetry data to the stack that follows the INT header. The INT Sink node strips both instructions and metadata from the packet before delivering it and then can selectively send the collected and aggregated data to the monitoring system. This way the packet size increases as it traverses more and more INT hops, which requires an appropriate setup of the MTU (Maximum Transmission Unit) allowed for the network segment implementing INT as well as limitations about the maximum number of hops that can insert data into the packet header. This mode of operation can reduce the overhead on the monitoring system as it doesn't receive reports from different nodes.

For this study, INT-MD mode was preferred, as it allows maximum flexibility in terms of instructions to execute on each hop by simply adjusting the bits encoded in the instruction field of the header. Moreover, this is what best implements the concept of "in-band" telemetry as data is directly inserted into the traffic without any report packets generated and directed to the monitoring system.

3.2 INT Metadata Header

The P4.org specification defines three different types of INT headers: the INT-MD type, the INT-MX type, and the Destination type. In this section, only the INT-MD Header type is presented and explained, as it is the one chosen for implementation. The INT-MD Metadata Header is 12 bytes long and it is followed by a stack of metadata populated by the hops that process the packet. Each metadata trace is either 4 or 8 bytes long, depending on what is indicated in the *hop_ml* field of the Metadata Header. Each hop adds the same amount of metadata, but the overall length of the INT-MD Header can vary, as different packets may traverse a different number of INT hops.

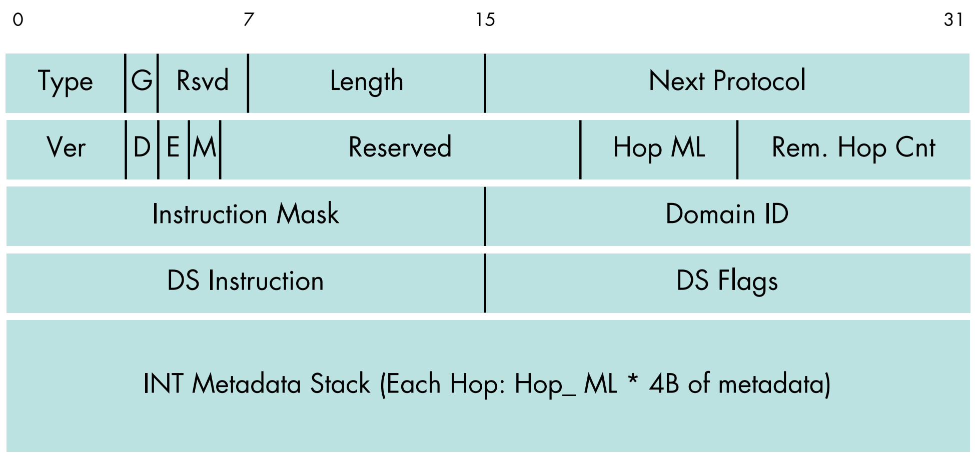


Figure 3.1. INT-MD Header Structure

The following provides a detailed breakdown for each field of the header shown in Figure 3.1:

- **Ver (4b)**: Indicates the version of the INT metadata header, it is set to 2 in this implementation.

- **D (1b) - Discard**: a flag which indicates whether the INT Sink should discard the packet after extracting the telemetry metadata, or deliver it to the recipient. This is useful in case INT is applied to synthetic rather than live traffic, which might indeed be discarded.
- **E (1b) - Exceeded**: Indicates whether an INT hop was unable to prepend its metadata due to the maximum hop count being exceeded (Remaining Hop Cnt reached 0). This must be set to 0 by the INT Source.
- **M (1b) - MTU Exceeded**: This bit has to be set if an INT hop cannot add its metadata without exceeding the egress link's MTU. In such cases, the hop does not add any metadata and sets the bit instead. This indicates that one or more hops didn't append their metadata traces, but does not specify which nodes encountered the issue.
- **Reserved (12b)**: These bits are reserved for future use, they should be set to 0 by the INT Source and ignored by other nodes.
- **Hop ML (5b) - Per hop Metadata Length**: Set by the INT Source, this field indicates to other nodes the amount of metadata, in 4B words, to insert. It is limited to a maximum of 31 and it includes the Domain Specific Metadata, unless the INT domain employs the 'source-only Domain Specific Metadata' (defined below).
- **Remaining Hop Count (8b)**: Specifies the number of INT hops that can append their metadata to the stack. Each INT node adding metadata, including the INT Source, must decrement this value by 1. If a packet arrives with this field already set to 0, the INT instructions should be ignored, and no data should be added. This field is set by the INT Source to the maximum number of hops allowed.
- **Instruction Bitmap (16b)**: Encodes of the INT instructions to be executed by the hops. Each bit represents a specific standard metadata value, as defined in [?]:
  - bit 0 (The Most Significant Bit - MSB): Node ID
  - bit 1: Level 1 Ingress Interface ID (16 bits) + Egress Interface ID (16 bits)
  - bit 2: Hop latency
  - bit 3: Queue ID (8 bits) + Queue occupancy (24 bits)
  - bit 4: Ingress timestamp (8 bytes)
  - bit 5: Egress timestamp (8 bytes)
  - bit 6: Level 2 Ingress Interface ID + Egress Interface ID (4 bytes each)

  - bit 7: Egress interface Tx utilization
  - bit 8: Buffer ID (8 bits) + Buffer occupancy (24 bits)
  - bit 15: Checksum Complement
  - The remaining bits are reserved.

  Each bit set instructs INT hops to insert 4B of metadata, except for the bits 4 to 6, which require 8 bytes each. The Hop ML is set accordingly by the INT Source.

- **Domain Specific ID (16b)**: Uniquely identifies the INT domain. If this value matches with any of the IDs known by the node, further processing of the following Domain Specific fields is performed.
- **Domain Specific Instruction (16b)**: Bitmap of instructions specific to the INT Domain, identified by the DS ID field. Each bit requests a specific DS Metadata. The amount of DS metadata added by each hop must be a multiple of 4 bytes and consistent with the value set in the Hop ML field.

In the application described later, the DS Instruction field is used to encode the operation requested by the source to each node (see chapter 4, section 4.4).

3.3 INT Header Placement

The P4.org specification outlines different options for the header placement within a packet's header stack. The INT header can indeed be inserted in different locations, either as an option or an encapsulation payload for any encapsulation protocol currently available. In this section, INT over GRE (Generic Routing Encapsulation) is presented as it is the preferred header placement for this application. The synthetic traffic used in the experiments include a GRE header. Figure 3.2 illustrates the final header stack a packet will have when it leaves the source.

The added GRE Header includes the following fields:

- **Type (4b)**: Specifies the type of INT header following the GRE header. The possible values, as defined in the P4.org specification, are discussed in section 3.2. In this application the header is always INT-MD, which corresponds to type 2.
- **G (1b) - GRE Encapsulation Indicator**: Indicates whether the original packet received at the INT Source was already GRE encapsulated (0) or not (1). This helps the INT Sink discriminate when it should remove the GRE header, as an encapsulation terminator.

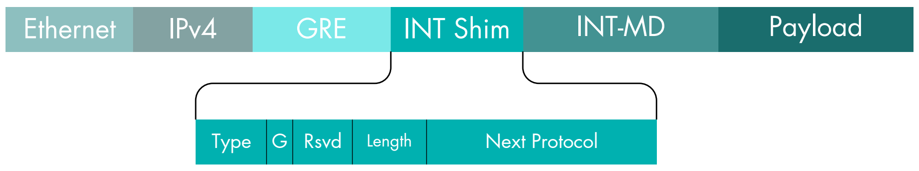


Figure 3.2. INT Header location - INT over IPv4/GRE

- **Rsvd (3b)**: Reserved bits for future use, set to 0 and ignored.
- **Length (8b)**: Specifies the total length of the INT Metadata header and INT metadata stack, excluding the shim header, measured in 4B words. This can help non-INT devices skip INT headers during parsing.
- **Next Protocol (16b)**: Contains an EtherType value, as defined by IANA [1], indicating the protocol that comes after the INT metadata stack.

3.4 Telemetry Metadata

The type of metadata which can be exported using the INT framework is a rich and assorted set, as potentially any device-level information can be collected. The use case to which INT can be applied strongly influences the kind of data to extract from the INT domain. Devices can be heterogeneous in the architecture as well as the supported set of defined and available metadata. The INT Specification [?] divides the telemetry metadata into three different classes: Node Level, Ingress Level and Egress Level.

[1]IANA, "IEEE 802 Numbers" available at `https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml`, accessed December 6, 2024.

3.4.1 Node-level Metadata

Device-level metadata includes all the information that can be related to a packet which is processed by a device and that is not specifically related to the ingress/egress interfaces. Some metrics of interest can be regarding the time the packet spent in the device's queues. Others can regard the device itself.

- **Node ID**: A unique identifier within the INT domain. This value is usually administratively assigned and can be useful to trace the path each packet took when traversing the network.
- **Control Plane State Version Number**: A value holding the version of the control plane state. This can be automatically updated by the device's control plane whenever the version of its state changes, for example, the IP Forwarding Information Base for a router.
- **Queue ID**: This identifier is related to the queue the device used to serve the INT packet.
- **Instantaneous Queue Length**: A snapshot indicating the queue length, in bytes, cells or packets. This can give insights into the state of the device for example regarding its congestion state.
- **Average Queue Length**: This is an aggregated metric that can reflect the trend over a period for the device's internal state.
- **Queue Drop Count**: the dropped packet counter for the specific queue.
- **Buffer ID**: It identifies to identify the physical buffer the packet traversed. This is useful when a buffer is shared among different queues.
- **Instantaneous/Average Buffer Length**: The instantaneous or average value in bytes or cells for the physical buffer occupancy.

3.4.2 Ingress or Egress-level Metadata

Metadata can specifically refer to ingress or egress statistics about the device. Ingress information can be used to infer the previous hop the packet was processed by or the time at which it was received. Other metrics can regard the resource utilization for the ingress queue/buffer of the device. The egress counterparts are also defined and can be supported by devices.

- **Interface ID**: The identifier of the interface on which the packet was received. The specification supports a stack of up to 2 interfaces and each device can utilize a specific semantics for this metadata.
- **Timestamp**: The device's local time reference when the packet was received or emitted. A time delta between the ingress and egress timestamp allows us to calculate the latency experienced by the packet for the processing into each device.
- **Packet Counter**: A counter for the received/emitted packet can be held by the device for the specific logical (or physical) interface by which the INT packet passed. Also dropped packets can be counted.
- **Byte Counter**: A counter to add information about the number of processed/-dropped bytes.
- **Ingress RX Utilization**: Even if the exact mechanism is vendor-defined, the INT specification supports this metadata which can give information about the resource utilization of the physical/logical interface on which the INT packet was received. The same is also defined for Egress TX Utilization.

3.5 P4 application for Alveo Cards

In our scenario, the programmable switches will be required by the INT Source to export a limited set of device-specific metadata based on what will be encoded in the instruction field of each packet. In the present section, it is described the P4 application developed to implement these operations in the data plane. As presented in subsection 2.2.1 the Xilinx Alveo U280 board is based on a specific architecture to be matched by the P4 application. Indeed, the architecture reflects directly on the design of the code. The XSA architecture the FPGA is based on defines three components: a **Parsing Engine**, a **Match-Action Engine** and a **Deparsing Engine**.

3.5.1 Headers and Structures

The headers section of the code, as shown in Listing 3.1, contains the definition of the headers and the respective fields that will be recognized and extracted when parsing an INT packet. Standard headers (Ethernet: line 1, IPv4: line 7, GRE: line 22) as well as INT header (line 35) and metadata stack (line 57 and 63) are defined in this section. To maintain coherence with the P4 specification, the INT header is divided into a *shim* header and a proper metadata stack, pre-allocated to contain the telemetry metadata. The stack is composed of two switch report headers.

```
header ethernet_t {
    bit<48> dstAddr;
    bit<48> srcAddr;
    bit<16> etherType;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;     // set to 0x2F for GRE encapsulated packets
    bit<16>   hdrChecksum;
    ipv4_addr_t srcAddr;
    ipv4_addr_t dstAddr;
}

header gre_t{
    bit<1>  C;              // tied to 0 as no checksum is used
    bit<1>  R;
    bit<1>  K;              // set to 0 for basic GRE
    bit<1>  S;
    bit<1>  s;
    bit<3>  recursion;
    bit<5>  flags;
    bit<3>  version;
    bit<16> protocol_type;  // TBD_INT in P4 specification
    /*other fields are optional*/
}

header int_shim_t{
    bit<4> type;            // INT-MD has type 1 in P4 specification
    bit<1> G;               // GRE flag - if 0 original packet had GRE encapsulation
    bit<3> Rsvd;
    bit<8> length;          // length without the int data and header!
    bit<16> next_protocol;  //ethernet type
}

header int_t{
    bit<4> ver;
    bit<1> D;
    bit<1> E;
    bit<1> M;
    bit<12> reserved;
    bit<5> hop_ml;
    bit<8> remaining_hop_cnt;
    bit<16> instruction_mask;
    bit<16> domain_id;
    bit<16> ds_instr;
    bit<16> ds_flags;
}

header switch_int0_t {
    bit<16> swid;
    bit<16> trust_level;
    bit<64> timestamp;
}

header switch_int1_t {
    bit<16> swid;
    bit<16> trust_level;
    bit<64> timestamp;
}
```

Listing 3.1. Headers Section of the P4 Application

Immediately after the headers section, there is the definition of the *headers* and the *smartnic metadata* data structures. The former (see line 1 of Listing 3.2) is be populated upon packet arrival and parsing (described later in subsection 3.5.2), and enables manipulation of header fields during packet processing. The latter (defined at line 12) is designed to store metadata associated with each packet, automatically populated with data retrieved from the SmartNIC, that can also potentially be used for telemetry purposes.

```
 1  struct headers {
 2      ethernet_t ethernet;
 3      ipv4_t ipv4;
 4      gre_t gre;
 5      int_shim_t int_shim_header;
 6      int_t int_header;
 7      switch_int0_t int_data_sw0;
 8      switch_int1_t int_data_sw1;
 9  }
10
11
12  struct smartnic_metadata {
13      bit<64> timestamp_ns;    // in ns, set at packet arrival time.
14      bit<16> pid;             // 16b packet id (read only)
15      bit<3>  ingress_port;    // 3b ingress port
```

```
    bit<3>  egress_port;      // 3b egress port
    bit<1>  truncate_enable; // 1b set to 1 to enable truncation
    bit<16> truncate_length; // 16b set to desired length of egress packet
    bit<1>  rss_enable;      // 1b set to 1 to override open-nic-shell rss
    hash result with 'rss_entropy' value.
    bit<12> rss_entropy;     // 12b set to rss_entropy hash value (used for
     open-nic-shell qdma qid selection).
    bit<4>  drop_reason;     // reserved (tied to 0).
    bit<32> scratch;         // reserved (tied to 0).
}
```

Listing 3.2. Headers and Metadata Structures

3.5.2 Parser

The parser is the logic part in charge of recognizing packet header fields and extracting information from its bytes to be used for later processing. The parser will allow the generation of the parse graph, where vertices are the parsing states while edges represent the transition between states. Each parsing starts from a **start** state, from this common root different state transitions are selected based on values contained in the extracted header fields and finally, each path in the graph terminates with either the **accept** or **reject**.

```
parser ParserImpl( packet_in packet,
                   out headers hdr,
                   inout smartnic_metadata sn_meta,
                   inout standard_metadata_t std_meta) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            ETHERTYPE_IPv4 : parse_ipv4;
            default: accept; //NOT | ethernet |+| IP |+| ... | -> don't
    care
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol) {
            PROTOCOL_GRE : parse_gre_encapsulation;
            default : accept; // If not an encapsulated packet -> don't
    care
```

```
        }
    }

    state parse_gre_encapsulation{
        packet.extract(hdr.gre);
        transition select(hdr.gre.protocol_type){
            TBD_INT: parse_int_shim;
            default: accept; // If encapsulated packet NOT containing INT
    -> don't care
        }
    }

    state parse_int_shim{
        packet.extract(hdr.int_shim_header);
        transition parse_in_band;
    }

    state parse_in_band{
        packet.extract(hdr.int_header);
        transition parse_int_sw0;
    }
    state parse_int_sw0{
        packet.extract(hdr.int_data_sw0);
        transition parse_int_sw1;
    }

    state parse_int_sw1{
        packet.extract(hdr.int_data_sw1);
        transition accept;
    }
}
```

Listing 3.3. The Parser

A representation of the parsing graph generated can be found in Figure 3.3. As illustrated, the parser designed for this INT application starts from the Ethernet header, it then expects an IPv4 header (based on the value of the *ethernet type* field) and proceeds parsing that. Then, following the same logic, it looks for a GRE header containing an INT shim header and finally the INT metadata stack. If any of these layers is missing in the headers, the packet is not actively managed by the application. This means that for the proposed implementation, it is out of our scope to handle every class of traffic and the focus is on managing only properly formatted INT traffic.

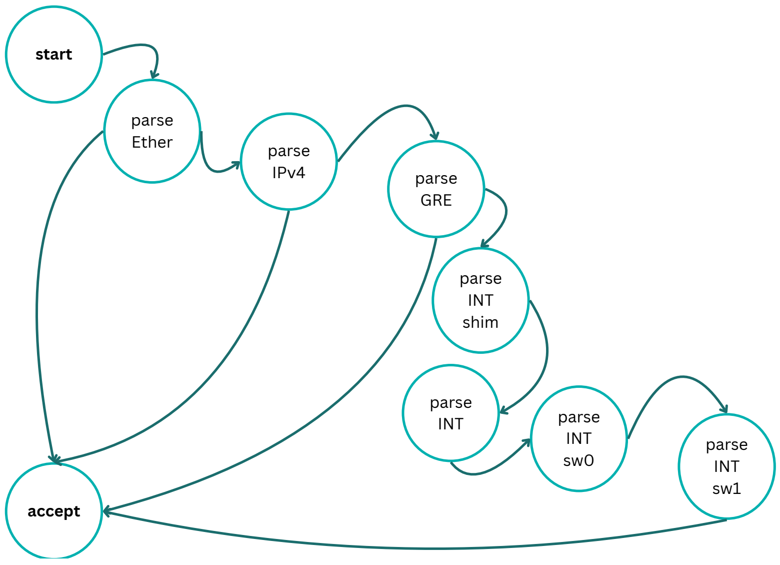


Figure 3.3. Parse Graph

3.5.3 Processing

The processing section holds the definition of the control block for the Match-Action Pipeline. Here, all the actions are coded together with their parameters (received by the control plane). This allows us to define how packets will be manipulated by the custom switch. In the same control block, there is the definition of the tables, which require a *key* the look-up will be based on as well as the set of actions that will be eventually taken as a response to a look-up. Of course, these tables are later populated at switch runtime by the control plane (or in this case, by the management plane). In this implementation, three tables are defined:

- **int operation** (shown in Listing 3.4): performs a lookup based on the domain-specific instruction field of the INT header based on an exact matching and defines the possible INT actions to be taken in response. These actions represent the collection of telemetry metadata in-band, with the insertion of metadata directly in the correct metadata stack field of the packet.
- **ipv4 routing**: implements a simple routing table, the lookup is based on the destination IP address with the longest prefix-matching logic. The implementation of all the routing functionalities is beyond the scope of this work.
- **l2 forwarding** (shown in Listing 3.5): this table implements a switching logic based on physical addresses derived from the Ethernet header. The lookup is based on the

destination MAC address to select an egress port, based on an exact matching.

```
table int_operation{
    key = {
        hdr.int_header.ds_instr:
   exact;
    }
    actions = {
        sw0_push_id;
        sw1_push_id;
        sw0_push_ingress_timestamp;
        sw1_push_ingress_timestamp;
        sw0_push_trust_level;
        sw1_push_trust_level;
        drop;
    }
    default_action = drop;
    size = 32;
}
```

Listing 3.4. INT Operation Table

```
table l2_forwarding{
    key = {
        hdr.ethernet.dstAddr : exact
   ;
    }
    actions = {
        l2_forward;
        //backward_learning;
        //flooding;
        NoAction;
        drop;
    }
    default_action = NoAction();
    size = 16;
}
```

Listing 3.5. L2 Forwarding Table

At the end of this control block, it is possible to find the **apply** control which allows us to define in which order and based on which conditions the different tables are applied to a packet. It is also possible to check for metadata values automatically updated by the SmartNIC, such as the presence of errors detected by the Parser. In this section of the code, given in Listing 3.6, note that INT operation table is applied only if the hop counter permits so, coherently with the maximum number of allowed devices.

```
apply {
    // 1. errors in the packet
    if (std_meta.parser_error != error.NoError) {
        drop();
        return;
    }
    // 2. valid ethernet and valid INT header -> apply INT and forward
    if (hdr.ethernet.isValid() && hdr.int_header.isValid()) {
        sn_meta.rss_entropy = 12w0;
        sn_meta.rss_enable = 1w0;
        if(hdr.int_shim_header.type == INT_MD && hdr.int_header.
    remaining_hop_cnt > 0){
            hdr.int_header.remaining_hop_cnt = hdr.int_header.
    remaining_hop_cnt - 1;
            int_operation.apply();
        }
        if (hdr.int_header.remaining_hop_cnt == 0) {
            hdr.int_header.M = 1;
        }
        l2_forwarding.apply();
    }
    // 3. valid ethernet without INT -> apply forward
    else if(hdr.ethernet.isValid() && hdr.ipv4.isValid()){
        l2_forwarding.apply();
    }
    else{
        drop();
    }
}
```

Listing 3.6. Apply Logic

3.5.4 Deparser and Pipeline Definition

At the end of the code is possible to find the de-parser control block which simply indicates how to put together all the pieces that compose a packet to output it on the egress interface. This step allows us to eventually modify the structure of the packet as it will look on the wire, but in our application, it is emitted with the very same structure as the incoming one because the metadata are inserted in the right field of the INT header. The definition of the Xilinx Pipeline is useful to indicate which components are requested for the hardware implementation and in which order they will be traversed by the packet. Our pipeline is composed by the three elements: Parser, Match-Action, Deparser.

```
control DeparserImpl( packet_out packet,
                        in headers hdr,
                        inout smartnic_metadata sn_meta,
                        inout standard_metadata_t std_meta) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.gre);
        packet.emit(hdr.int_shim_header);
        packet.emit(hdr.int_header);
        packet.emit(hdr.int_data_sw0);
        packet.emit(hdr.int_data_sw1);
    }
}

XilinxPipeline(
    ParserImpl(),
                MatchActionImpl(),
                DeparserImpl()
) main;
```

Listing 3.7. Deparser and Pipeline Definition

Chapter 4

Experimental Evaluation

This chapter provides an in-depth and technical description of the experiment. Our methodology follows the existing P4 specification for INT [?], aiming to simplify the architecture while preserving generality and reproducibility. The experiment's primary goal is to demonstrate a proof of concept for an INT domain implemented on real infrastructure, leveraging the computational acceleration provided by FPGA-based smartNICs on the FABRIC Testbed. These smartNICs execute the INT operations encoded in the INT header using the INT-MD mode of operation. The traffic is enriched with telemetry metadata and analyzed on the last hop of the domain, highlighting the potential of FPGA acceleration in real-world INT deployments.

4.1 FABRIC Testbed Topology

The first step in setting up the experiment is the design, setup and deployment of the topology that will host the machines and network devices emulating the INT Domain. All the components of the experimental network deployed on FABRIC are assigned distinct roles for the experiments, in this sense the topology is not symmetrical. The traffic flows from a source to a sink, or destination, passing by the two INT devices. The roles are clarified below as well as in Figure 4.1:

- **source**: The INT Source node, on which the traffic is crafted with the desired encoding of the Domain Specific Instruction (refer to section 4.4). This node represents one edge of the INT domain.
- **fpga1, fpga2**: The two INT devices implemented with the programmable data planes. They will process the packets, executing the DS Instruction requested by the source.

- **sink**: The INT Sink node, collecting the INT traffic and breaking down the packets into their components to analyse the received metadata. In a real scenario, this device would be also in charge of crafting report packets towards an INT Monitoring System, which is beyond the scope of this study.

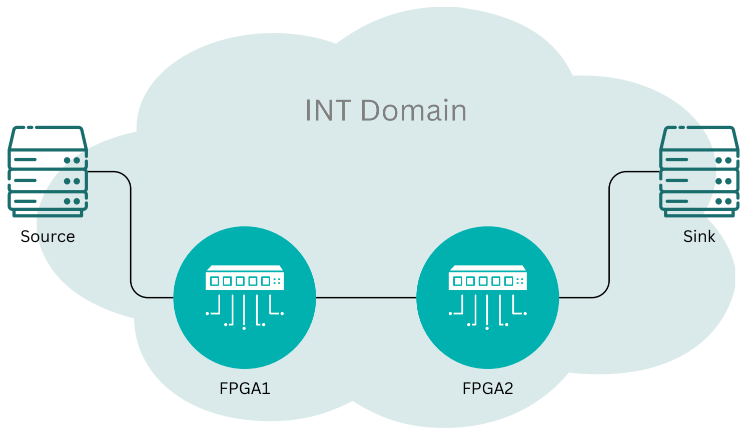


Figure 4.1. Logical topology of the INT Domain

Such a set of resources is created and managed through a Jupyter Notebook interacting with the FABRIC orchestrator using the Fablib API. The full Jupyter Notebook developed to deploy the network can be found in Listing 7.1. The following Figure 4.2 shows the topology created on FABRIC testbed.

This topology emulates the INT domain needed for the experiment. It includes four virtual machines interconnected by three traditional layer 2 switches. Since each site (or node) of the FABRIC Testbed international network has access to at most one SmartNIC, the slice is distributed over two different sites. Each VM has the following set of resources allocated:

- Ubuntu 20.04 Focal Fossa as image
- 8 CPU cores
- 16 GB of RAM
- 100 GB of secondary storage

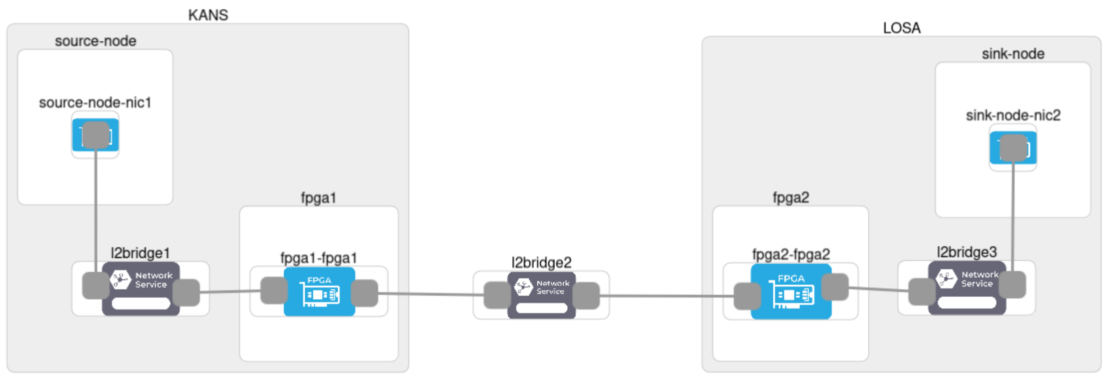


Figure 4.2. FABRIC Testbed Slice Implementing The INT Domain

The two VMs in the middle, referred to as *fpga1* and *fpga2* in Figure 4.2, also have access to the SmartNIC (Alveo U280) as a dedicated component, passed through the virtualization stack. The FPGAs are accessed and managed through the virtual machines which will no longer be directly involved in the experiment after the preparation phase.

Traditional networking equipment (*l2bridge1*, *l2bridge2*, *l2bridge3* in Figure 4.2) is needed to interconnect the machines but is transparent to the experiment's purpose. It is indeed only in charge of bridging the traffic from the output interface of one of our INT hosts to the input interface of the next one as if they were directly connected.

4.2 Flashing The Bitfile On The Smartnics

Once the topology is ready, the SmartnNICs need to be programmed with the bitfiles produced by the compilation and synthesis processes. The *Vivado Vitis Networking P4* invoked by means of the *ESnet Development Workflow* zips them in hardware description artifacts which can be flashed on the Alveo cards to configure and customize their data planes. The procedure makes use of a set of tools available on the management VMs which have access to the FPGAs, presented in section 2.3. First, the build environment must be prepared by creating the two Docker images `xilinx-labtools-docker`[1] and

[1] GitHub repository for Xilinx LabTools Docker image: `https://github.com/esnet/xilinx-labtools-docker/tree/main`

`smartnic-dpdk-docker`[2]. Then, it is possible to build a new firmware image, which depends on the previously built hardware description artifact. This results in a third Docker image named `esnet-smartnic-fw`[3]. This process results in the provisioning of the powerful set of tools made available by the framework, which include:

- **Data Plane Development Kit (DPDK)**: an application that allows bypassing the kernel of the server hosting the FPGA to connect directly to it, for high-performance packet processing and data plane operations.
- **Pktgen**: a DPDK application to easily transmit packets from the FPGA through its 2x100G ports. It also enables communication with the management VM via the PCIe bus.
- **ESnet CLI Tools**: set of command-line tools that provide control over the QDMA queues, access to the probe counters to trace the packets and perform statistics, management of parameters and configuration of the control plane rules of the P4 application and eventually control over the FPGA internal switch.

.

4.3 Smartnic Configuration

The first configuration to apply concerns the internal switch of the SmartNICs. By default, once the device is programmed, the logical ingress interfaces of the SmartNICs are directly connected to the *drop* stage, meaning that every packet received by the two 100G ports gets dropped without being processed by the data plane application. For the purpose of this experiment, it is necessary to attach these logical input interfaces to the processing pipeline. This is achieved using the command:

```
sn-cfg batch configure-switch -i port0:app0 -i port1:app0}
```

While the output interface for each packet is determined by the P4 logic and its defined rules, it is worth mentioning that it is possible to override the egress port decision with a similar command. This feature proves useful for accommodating the specific needs of different experiments. Other configurations help modify the default values and ensure that the switch reaches the expected final state, aligning with our assumptions about the

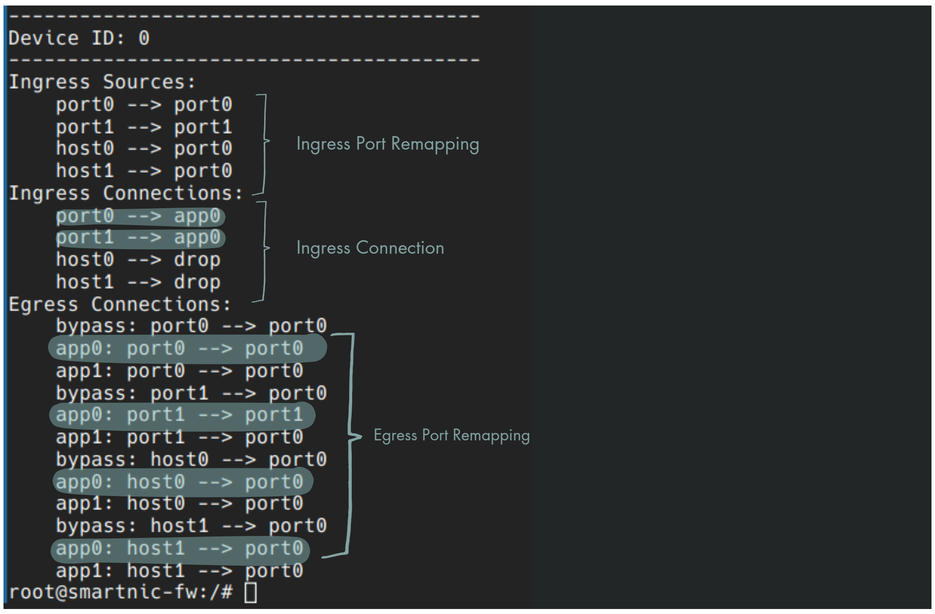
[2]GitHub repository for SmartNIC DPDK Docker image: https://github.com/esnet/smartnic-dpdk-docker

[3]GitHub repository for Smartnic Firmware image:https://github.com/esnet/esnet-smartnic-fw

internal paths followed by packets once they are received by the SmartNIC. The full switch configuration is available in the script reported in Listing 7.2, while the final state of the switch can be inspected using the command:

```
sn-cfg batch show-switch-config
```

whose output is shown in the following Figure 4.3.



```
----------------------------------------
Device ID: 0
----------------------------------------
Ingress Sources:
    port0 --> port0
    port1 --> port1
    host0 --> port0
    host1 --> port0
Ingress Connections:
    port0 --> app0
    port1 --> app0
    host0 --> drop
    host1 --> drop
Egress Connections:
    bypass: port0 --> port0
    app0: port0 --> port0
    app1: port0 --> port0
    bypass: port1 --> port0
    app0: port1 --> port1
    app1: port1 --> port0
    bypass: host0 --> port0
    app0: host0 --> port0
    app1: host0 --> port0
    bypass: host1 --> port0
    app0: host1 --> port0
    app1: host1 --> port0
root@smartnic-fw:/# 
```

Figure 4.3. Output from the command `sn-cfg batch show-switch-config`

The final step is to populate the control plane rules for the loaded P4 application. This requires inserting entries in the appropriate tables of the processing pipeline, defining the values to match against, the actions to take and the related parameters, if any. The command to insert such rules is:

```
sn-p4 insert table rule -t <TAB-NAME> -m <MATCH-VALUE> --action <ACTION>
    [--param "P1 P2 ..."]
```

Since the P4 application processes packets by inserting INT metadata according to the received instruction, the two FPGAs need to be configured with custom parameters to be exported based on the same instruction values. Table 4.1 presents the configuration for each switch.

INT Instruction	**FPGA1**	**FPGA2**
push switch ID	0x101	0x201
push switch TL	trust level	trust level
push ingress timestamp	timestamp	timestamp

Table 4.1. Mapping of INT Instructions to FPGA1 and FPGA2 Parameters

The complete script containing the parameters for each P4 rule can be found in Listing 7.3. The three types of available metadata differ significantly in nature, as clarified below. **Switch identifiers** are administratively assigned in this step of the configuration, after the SmartNICs are programmed, by differentiating the parameter associated with the same action. In contrast, the **trust levels** are hard-coded into the application, defined directly within the appropriate actions in the P4 code. While the precise semantics of these values fall outside the scope of this work, their purpose is to enable the collection of metadata from switches that cannot be administratively altered. These trust values could potentially represent a combination of various metrics, such as the confidentiality level or the reliability of a given switch in processing specific classes of packets. Lastly, the **timestamps** are directly exported by the SmartNICs based on their local clock, without requiring manual assignment or additional configuration.

4.4 Generating Synthetic Traffic

The network traffic generated for the experiment consists of a set of synthetic packets originating from the INT Source node. This traffic is generated using a *Python* script and the powerful packet manipulation library *Scapy*, which is available in Listing 7.4. The script first defines custom packet classes to represent the header formats recognized by the P4 application. Among these, one corresponds to the INT Shim header, another to the INT header, and another the INT Metadata stack. Next, the script includes a layer-binding section, where the header stack is constructed using Ethernet, IPv4, GRE and the custom INT headers. Finally it creates the packet assigning specific values to the following header fields:

- destination MAC address of the INT sink node
- source MAC address of the INT source node
- Ethernet type 0x1717, used to indicate the encapsulated payload is an INT packet
- Initial value for the INT Hop Count field of 32, which is decremented at each hop

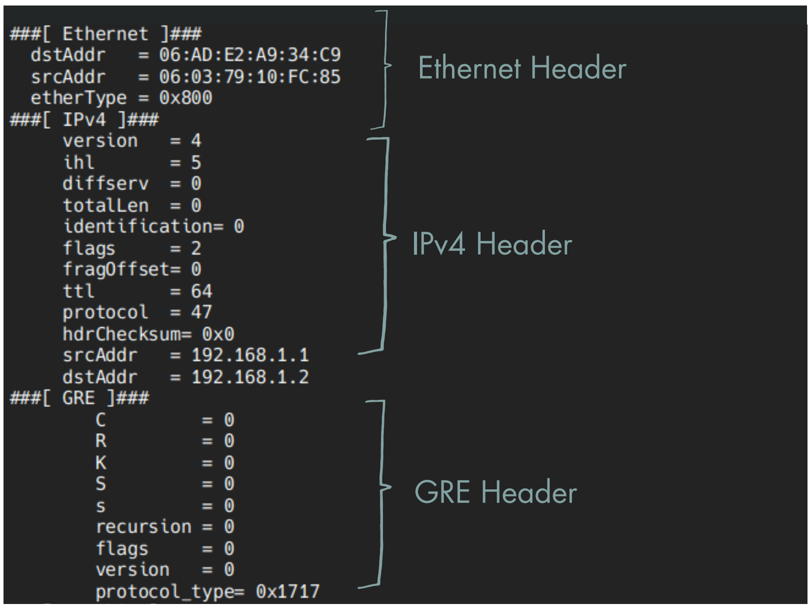
- The Domain Specific Instruction field, set to request different metadata from the devices, which will be appended to the packet.

The outgoing packet is displayed by the script both as raw bytes and in a formatted (pretty-printed) view and is then sent out from the source node's interface using the *srp1* function offered by Scapy. An example of a packet generated using this method for the experiment can be found in Figure 4.4. The packet structure is displayed with each header clearly separated, and every field is labelled with its corresponding name to facilitate the interpretation of the packet's composition. In this visualization note the INT headers are stacked and composed coherently with the P4 application's parsing logic.

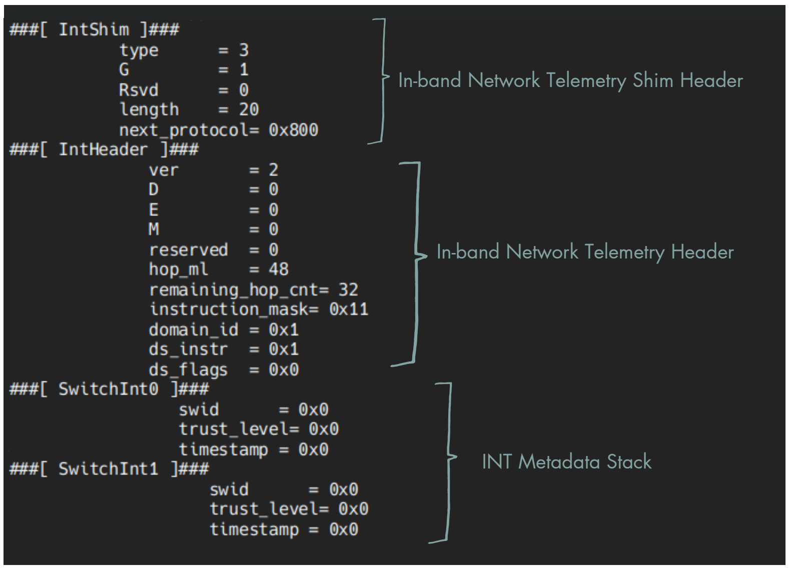
4.5 Sniffing and Analyzing INT Packets

The INT sink represents the final hop of the INT domain, meaning that its role is logically opposite to that of the source. In a real scenario, where telemetry metadata would be carried by application-generated traffic, this hop would be required to further process the packets to normalize and deliver them to their intended recipients, while simultaneously extracting the collected metadata. Additionally, such metadata would have to be prepared for the INT Monitoring System entity not present in this experiment. In this implementation, this node listens for packets arriving at its interface, parses them, and analyses the fields composing each header. Since the traffic used in this experiment does not carry payload data intended for delivery, the packets are not forwarded beyond this point. The analysis of incoming traffic is conducted using *Tcpdump* as packet-sniffing tool, allowing us to capture packets and display their raw byte format. Figure 4.5 presents the output for the command `sudo tcpdump -nlvvx -i enp7s0` executed on the sink node when a packet is received. The automatic analysis performed by Tcpdump detects inconsistencies in the packet structure, such as missing bytes or invalid checksums as can be noticed in Figure 4.5. These anomalies arise from the traffic generation script, which does not ensure full compliance with real-world packet structures. Nevertheless, packet capture confirms that the two SmartNICs successfully manipulated the packet to embed their metadata. At the sink node, the packet contains the data requested by the source to each INT hop while other metadata fields reflect changes introduced at each stage of the topology. The captured packet, presented in Figure 4.5, includes the following metadata:

- **switch ID 1**: this field is inserted by the first INT device, *fpga1*, which exports the expected value **0x101** based on its control plane configuration section 4.3.
- **switch ID 2**: The second INT device, fpga2, adds its unique identifier within the INT domain, which holds the value **0x201**, consistent with its configuration.



```
###[ Ethernet ]###
  dstAddr   = 06:AD:E2:A9:34:C9
  srcAddr   = 06:03:79:10:FC:85
  etherType = 0x800
###[ IPv4 ]###
     version   = 4
     ihl       = 5
     diffserv  = 0
     totalLen  = 0
     identification= 0
     flags     = 2
     fragOffset= 0
     ttl       = 64
     protocol  = 47
     hdrChecksum= 0x0
     srcAddr   = 192.168.1.1
     dstAddr   = 192.168.1.2
###[ GRE ]###
        C         = 0
        R         = 0
        K         = 0
        S         = 0
        s         = 0
        recursion = 0
        flags     = 0
        version   = 0
        protocol_type= 0x1717
```

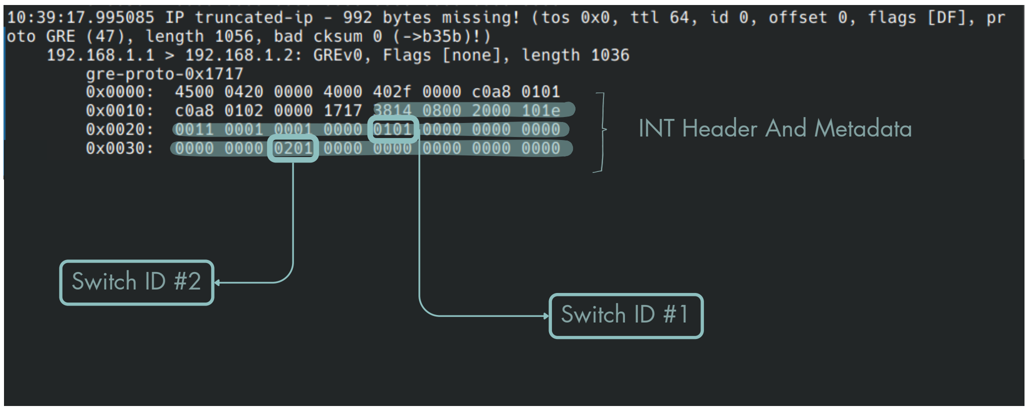

In-band Network Telemetry Shim Header
In-band Network Telemetry Header
INT Metadata Stack

```
###[ IntShim ]###
           type      = 3
           G         = 1
           Rsvd      = 0
           length    = 20
           next_protocol= 0x800
###[ IntHeader ]###
              ver       = 2
              D         = 0
              E         = 0
              M         = 0
              reserved  = 0
              hop_ml    = 48
              remaining_hop_cnt= 32
              instruction_mask= 0x11
              domain_id = 0x1
              ds_instr  = 0x1
              ds_flags  = 0x0
###[ SwitchInt0 ]###
                 swid      = 0x0
                 trust_level= 0x0
                 timestamp = 0x0
###[ SwitchInt1 ]###
                    swid      = 0x0
                    trust_level= 0x0
                    timestamp = 0x0
```

Figure 4.4. Breakdown of an INT packet crafted by *Scapy* script

- **Remaining Hop Count**: Each INT device decrements this field at every hop. Initially set to 32 (0x20), it reaches the sink node with a final value of 30 (0x1E) after two hops as can be observed in the highlighted bytes in Figure 4.5.

The analyzed packet is generated by the Python script described in section 4.4 (available



```
10:39:17.995085 IP truncated-ip - 992 bytes missing! (tos 0x0, ttl 64, id 0, offset 0, flags [DF], pr
oto GRE (47), length 1056, bad cksum 0 (->b35b)!)
    192.168.1.1 > 192.168.1.2: GREv0, Flags [none], length 1036
        gre-proto-0x1717
        0x0000:  4500 0420 0000 4000 402f 0000 c0a8 0101
        0x0010:  c0a8 0102 0000 1717 3814 0800 2000 101e
        0x0020:  0011 0001 0001 0000 0101 0000 0000 0000
        0x0030:  0000 0000 0201 0000 0000 0000 0000 0000
```

Figure 4.5. A packet captured by *Tcpdump*, INT portion of the packet and metadata are highlighted

in Listing 7.4), with a Domain Specific Instruction field value of **0x0001**, corresponding to the *push switch ID* action configured on the SmartNICs. This instruction enables the INT domain to reconstruct the traffic path with packet-level granularity. Each INT device leaves a trace in the packet, allowing the final node to identify all INT switches that have processed it.

Chapter 5

Performance Evaluation

The efficiency of the INT technique presented strongly depends on the system's ability to maintain high performance while ensuring reliability and accuracy in measurements. When executing telemetry operations on programmable hardware such as the AU280 board, it is crucial to evaluate the potential impact on performance. In particular, this study focuses on determining the proportion of packets correctly enriched with switch traces as they traverse the topology and on assessing the additional cost in terms of network latency incurred by these packets. This chapter presents two experiments designed to provide an objective assessment of the efficiency and the limitations of the application deployed on FPGAs, contributing to a better understanding of its performance in an experimental network.

5.1 Path Tracing Accuracy

Accuracy in tracing the path followed by a flow is defined as the ability of INT devices to mark the packets they process with the metadata entries requested by the telemetry instruction. A low tracing accuracy would result in some packets being lost while traversing the topology or in some packets not carrying the expected INT data. Recalling the logic of the packet processor implemented in P4 and the architecture of the target device, several factors could cause packet loss:

- Packets dropped at the ingress stage of the SmartNIC's internal processing pipeline.
- Packets dropped due to the default action in the P4 logic when a match lookup results in a miss.
- Packets dropped at the egress stage of the SmartNIC's internal processing pipeline.

- Packets discarded upon the error check failure (*parser_error* field of the standard metadata structure set).
- Packets with an unexpected header structure are discarded by the application after parsing.
- Packets overflow a FIFO queue interconnecting different processing stages in the pipeline.
- Packets lost in traditional networking equipment, such as L2 switches or the L2 site-to-site switches of the FABRIC topology

Given the variety of potential causes impacting the tracing accuracy, a key metric is the ratio of packets received at the INT sink to the number of packets generated by the INT source for the test flow. Additionally, after packet inspection performed by the INT sink node, the percentage of packets carrying the expected metadata extracted from the network is computed. To obtain these measurements, the in-band network telemetry experiment from chapter 4 was repeated using the same topology and SmartNIC configuration. As previously presented in Figure 4.2, the topology consists of an INT source, two chained FPGAs and an INT sink at the opposite end. This experiment requires an additional software component running on the sink node to count and parse incoming packets and perform statistical analysis. A *Python + Scapy* script is responsible for the following tasks:

- Sniffing network packets on the interface receiving traffic from the node labelled *fpga2*.
- Processing each incoming packet through a callback function to analyze its raw bytes and search for the expected telemetry metadata. Since the INT instruction encoded in the header queries the switches for their identifiers, these are the values searched within the bytes composing the INT metadata stack using a regular expression.
- Maintaining counters for the total number of packets received on the interface and the number of packets where the *regex* pattern matched.

This experiment confirms that no packet loss occurs within the INT domain, as every packet is correctly processed by the INT nodes and reaches the destination carrying the expected metadata. This meets the expectations given the low data rate. The computed packet loss percentage, defined in Equation 5.1, is **0%**.

$$\frac{(Num_{sent} - Num_{recv})}{Num_{sent}} * 100 \tag{5.1}$$

While this result confirms that the INT processing pipeline successfully embeds telemetry data without discarding packets under these conditions, it does not provide insights into the system's behavior under higher traffic loads. Additionally, this experiment does not evaluate the potential impact of telemetry operations on end-to-end latency, which is instead analysed in the next section

5.2 RTT Degradation Due To Telemetry Operations

The second performance experiment involves a slightly more complex topology configuration and aims to examine the degradation of the round-trip time experienced by packets when they are forwarded by the switches performing additional telemetry operations, along with basic L2 forwarding. The logic of the experiment is to generate traffic while keeping a detailed timestamp of when each packet leaves the interface of the source host. The packets are then injected into the INT domain and traverse the topology as usual, collecting switch traces in the telemetry metadata. For this experiment, the packet routing was modified to create a logically circular topology, ensuring that the traffic returns to the same host where it was generated. This setup, illustrated in Figure 5.1 allows to timestamp the received packets again and compute the latency between the *time_tx* and *time_rx* timestamps.

To avoid inconsistencies and obtain accurate timing measurements, any potential source of application-level processing was excluded by recording timestamps at the lowest possible level: the Ethernet layer. This was achieved using *Tcpdump* packet sniffer features. This command offers the option `-tttt`, useful to obtain the maximum timestamp resolution down to one microsecond. Once the packets reach the first and second FPGA switches, they are forwarded back to the source via the bypass paths of the two INT switches rather than continuing toward the sink node. This behaviour depends on the FPGA switch configuration and can be adjusted with the commands reported below. The node labelled *fpga1* is configured by linking its *port0* to the application logic (*app0)*, ensuring that the traffic received from the INT source is processed. On the other hand, the ingress *port1* is connected to the egress *port0* via the bypass path. Similarly, the *fpga2* node is configured with the ingress *port0* connected to the P4 logic. However, instead of forwarding traffic to *port1* for delivery to the sink node, the configuration overwrites the application's egress selection with the same port, ensuring that traffic is delivered back to the source without additional processing

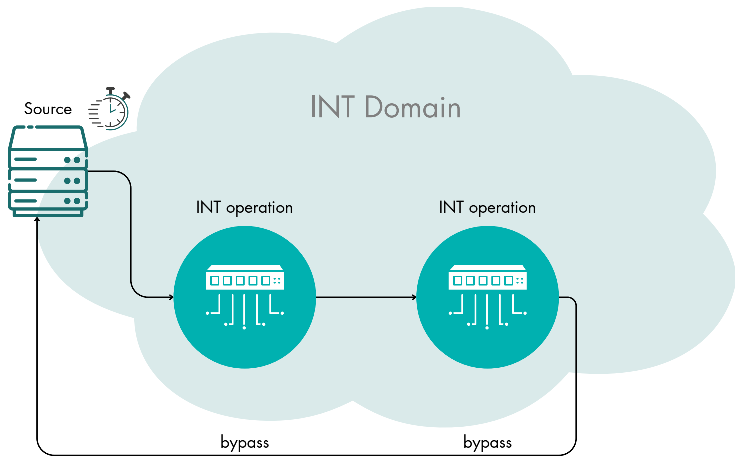


Figure 5.1. Topology Configuration For RTT Experiment

```
sn-cfg batch configure-switch -i port0:app0 -i port1:bypass
sn-cfg batch configure-switch -e app0:port1:port1 -e bypass:port1:port0
```

This setup eliminates the need to synchronize the source and destination clocks for coherent timing measurements, as the traffic flow originates and terminates on the same machine (the source).

Half of the total generated packets are used to establish a baseline latency measurement. These packets are not INT-formatted and instead, they contain a dummy UDP payload. According to the P4 application behaviour, these packets are not enriched with metadata but are simply forwarded. The experiment then continues with INT packets, repeating the latency measurements to assess the impact of telemetry operations. A *Python* script generates the packets using the *Scapy* library, while *Tcpdump* operates at a lower level, capturing traffic on the interface and storing the results in a file. The file is then parsed and processed by a separate program responsible for computing latency, specifically analyzing the *inter-packet-gap*, defined as the time delta between two consecutive lines of the capture file. To mitigate the impact of fluctuations and reduce the noise in the graph, a moving average with a sliding window of 10 data points was applied. The resulting graph is reported in Figure 5.2, showing the smoothed latency trends for normal (in blue) and INT packets (in red). The x-axis represents the packet index while the y-axis shows the measured RTT in milliseconds. From the figure, it is possible to observe

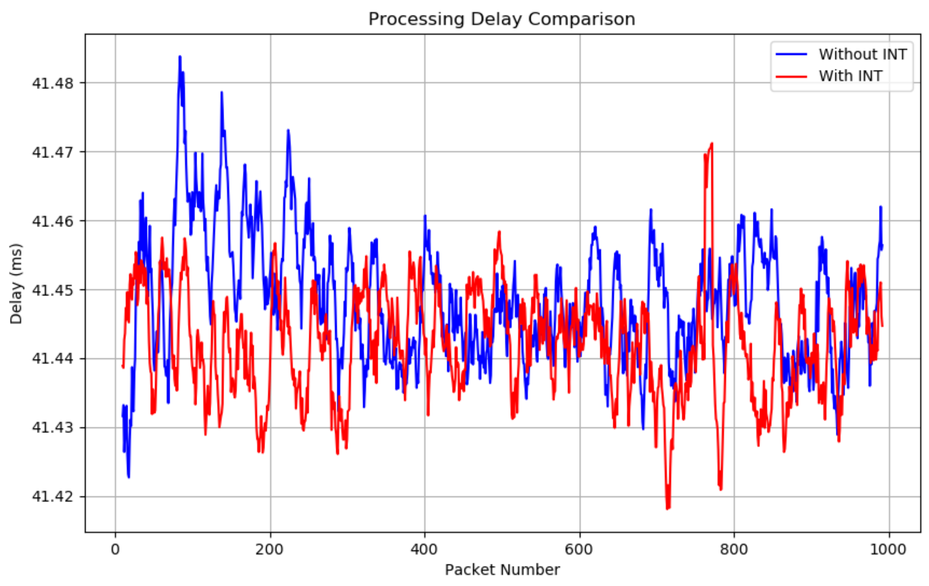


Figure 5.2. Moving average of normal and INT packets RTT

that, on average, the round-trip time measured in this experiment is not significantly affected by the telemetry operations. The mean RTT for INT packets remains close to that of non-INT packets, suggesting that the additional processing does not introduce appreciable overhead. However, the graph reveals some fluctuations. In certain areas, the lines show higher latency values for INT packets compared to standard-formatted packets, possibly due to temporary queuing periods in the FPGA switches. In contrast, in other sections of the graph, INT packets appear to exhibit a lower round-trip time. Despite these fluctuations, the overall trend does not indicate a systematic performance degradation caused by INT processing. The latency remains in an acceptable range and this suggests that the FPGA implementation can efficiently handle the additional processing to execute telemetry functions without significantly impacting network performance.

To gain further insight into the results, additional statistics were extracted from the same packet capture recorded in the tcpdump output file. These statistics are presented in the following.

1. The **Cumulative Distribution Function** of the delays, shown in Figure 5.3, illustrates the probability that a given round-trip time falls within a specific range. The x-axis represents the delay in milliseconds while the y-axis shows the cumulative

probability. The CDF curves for both normal and INT packets allow us to compare their latency distributions. From this graph, we observe that both distributions are very similar with most delays falling within the range of 41.35 ms to 41.55 ms. However, the red curve (representing packets with INT) appears a little left-shifted, indicating that a fraction of packets experience lower delays. This confirms that INT processing does not introduce appreciable additional latency in most cases. At the tail of the distribution we notice both curves converging rapidly to 1, meaning that extreme delays are very few.

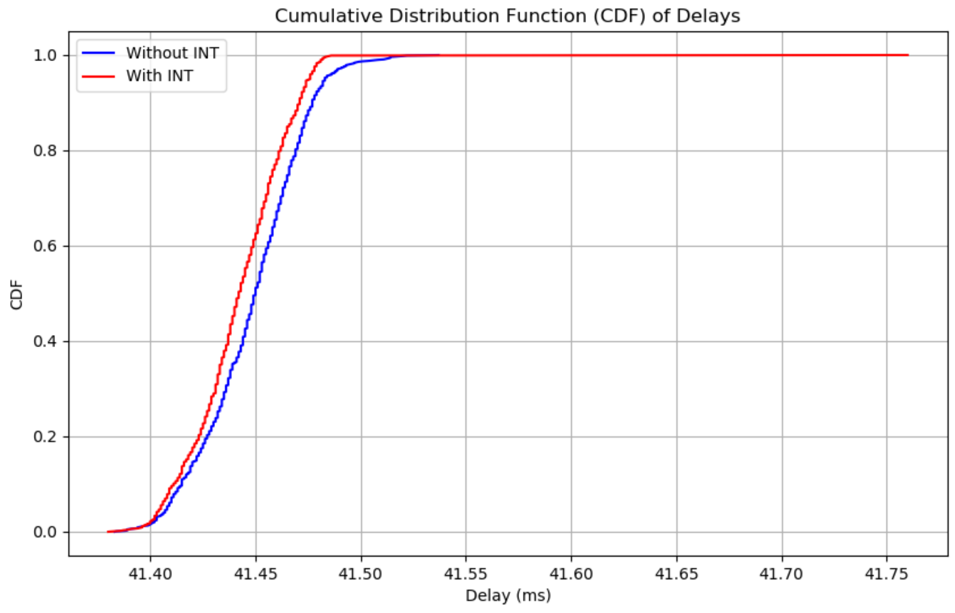


Figure 5.3. Cumulative Distribution Function of RTTs

2. The **histogram** in Figure 5.4 illustrates a detailed view of the delay distribution by showing the frequency of different values. In the plot, the x-axis represents processing delays in milliseconds, while the y-axis indicates density (or relative frequency of each value). Here, we see that both normal and INT packets (respectively in blue and red) present a peak around 41.45 ms, reinforcing the observation from the previous results that INT processing does not dramatically shift the overall latency distribution. The two distributions are nearly overlapping however there are minor differences. The red bars (INT-formatted packets) exhibit a higher density in the 41.40 to 41.45 ms range suggesting that some packets collecting INT metadata returned faster than

expected on the source. In contrast, the normal packets represented by blue bars have a slightly wider spread toward higher latencies. The variations are small, indicating that any added processing overhead introduced by INT operations remains negligible in this practical evaluation.

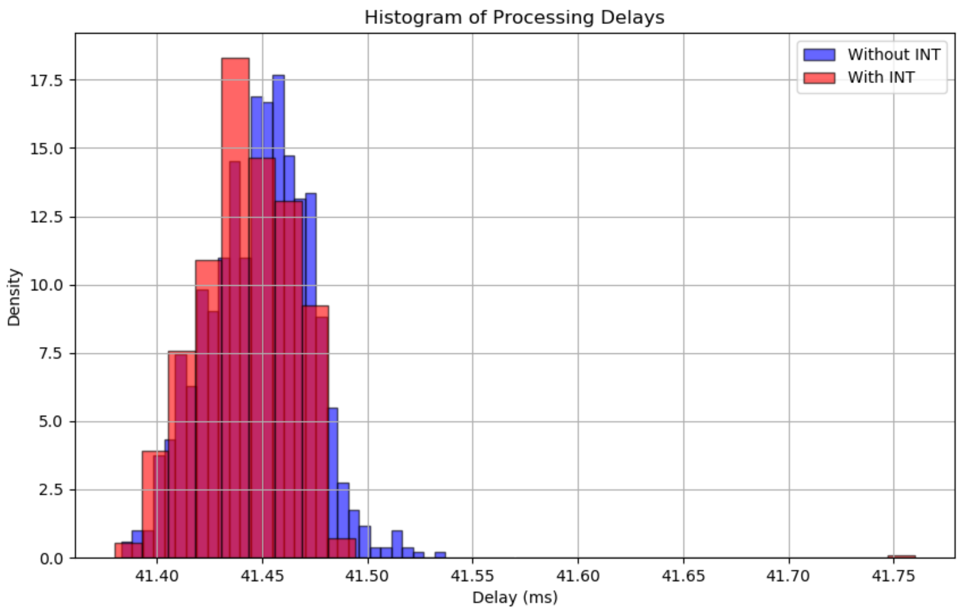


Figure 5.4. Histogram of Delay Distribution

5.2.1 Results Interpretation

The unexpected result where network latency is lower for packets processed with additional INT operations compared to traditional L2 forwarding, can be explained considering different network conditions that are not controllable in this experiment. Since FABRIC nodes are indirectly connected via shared links, the instantaneous load conditions are unpredictable and may vary. As can be observed in the presented graphs, this led to INT packets experiencing less congestion compared to regular packets, resulting in lower latency. Additionally, the implementation of telemetry on FPGA hardware is expected to introduce negligible overhead compared to delays caused by congestion and buffering in the network. If telemetry operations are executed fast enough, competing traffic in the network could have had a greater impact on the latency of regular packets than those carrying INT instructions and metadata. In conclusion, the lower latency observed for

INT packets does not necessarily mean that telemetry processing improves the overall network performance but rather that network conditions, which are not fully controllable or predictable, had a greater impact on regular packets than on those carrying INT. Some additional limitations of this experiment must be considered and are discussed in the following section.

5.2.2 Limitations of the Approach

One key observation is that the measured delays are in the order of tens of milliseconds. While this time scale is suitable for evaluating round-trip times in a network, it is considered too coarse to capture finer latency variations introduced by different packet processing paths within the hardware implementing the data plane. The predominant contribution to the observed round-trip time stems from the large geographical distances spanned by the experimental network deployed within FABRIC Testbed. The physical sites involved in this topology are CLEM (Clemson University, rack location at 340 Computer Court, Anderson, SC 29625) and KANS (Kansas City, rack location at 1100 Walnut Street, Kansas City, MO 64106). These locations are not directly connected through a dedicated link, meaning that the traffic traverses multiple intermediate segments, potentially adding further, uncontrolled, latency variations. Additionally, another limitation of this experiment is the traffic rate at which it was conducted. The test was conducted sending one packet at a time and as a result the traffic load remained low and did not stress the per-port capacity of the FPGAs. This means that potential queuing effects or congestion-related delays (factors that could become significant in a high-load scenario) are not represented in the results. A final aspect to consider is that the experiment involves two FPGA switches in the data path. Since the packets undergo telemetry processing at both FPGA nodes, the tests are effectively estimating the cumulative time overhead introduced by both devices together. However, in a more developed and realistic topology, the number of telemetry-enabled hops would likely be higher, potentially making the total processing delay more noticeable and impactful on overall network performance.

Chapter 6

Conclusion

6.1 Work Overview

As modern communication networks evolve and gain importance in the services they support, everyday users, critical service providers and policymakers have increasing concerns about the reliability of these infrastructures. This highlights the necessity of adopting advanced tools for network monitoring and traffic management. Traditional telemetry techniques are based on out-of-band methods such as SNMP or NetFlow, and they present several limitations, including the polling-based approach and their dependency on control plane operations. The In-band Network Telemetry framework represents an alternative approach that allows for the collection of information about the network state by directly embedding data within the traffic packets as they traverse the topology. This methodology offers a detailed and real-time view of the network state by leveraging data plane operations, significantly enhancing the scalability and the achievable level of detail. To implement custom telemetry functions in the data plane of network devices, a class of programmable hardware devices was investigated, with a focus on FPGA-based smartNICs. This research was supported by the implementation of a proof-of-concept experimental network capable of collecting metadata from switches as they process packets. For this purpose, the FABRIC Testbed infrastructure was used, providing the necessary hardware components, specifically the Alveo U280 Datacenter Accelerator cards. The technological framework used to interact with these remote hardware facilities included a toolchain that allowed high-level packet processing operations, formalized in a P4 program, to be transformed into a bitfile to be flashed onto the SmartNIC. Through this process, the FPGA chip was configured with the desired hardware functions. The success of the demonstrative INT domain implementation was validated through two performance tests: one to evaluate the accuracy of our system in marking INT packets with the switch identifiers and collecting the path trace, and another to measure the potential time overhead introduced by INT operations when compared to traditional L2 forwarding.

6.2 Challenges and Limitations Encountered

6.2.1 Unsupported P4 Features On Target Platform

The P4 language has some inherent limitations, and additional restrictions are imposed by the platform on which an application runs. In particular, the unsupported features from which the developed application would benefit, especially in its design and logic, are highlighted.

1. **if statements in actions** are not supported. This prevents the possibility of performing differentiated actions based on decisions made dynamically for each packet within the same action (e.g. selecting the switch identifier to be pushed dynamically, rather than defining separate actions, which would increase the number of table entries).

2. **stack of headers** cannot be implemented in P4 code. As a result, all INT metadata headers must be statically allocated at the source. Using headers stack would allow INT metadata to be treated as a homogeneous collection of data structures, each representing a switch trace. This approach would enable indexed access, reducing both table usage and code redundancy.

3. Packets cannot be **cloned or recirculated**, two actions that are respectively useful for generating reports and for re-injecting packets into the processing pipeline.

6.2.2 Unexpected Behaviour Of Metadata Interface

Vitis Networking P4 offers metadata ports, which serve as a secondary interface to facilitate the exchange of sideband packet-related data between the engines that compose the pipeline and potentially external systems. Metadata is associated with a single packet and processed concurrently with the packet itself. From the data plane of the smartNIC, within the P4 code, metadata is accessible as a data structure that must be defined according to the VNP4 documentation [?].

```
struct smartnic_metadata {
    bit<64> timestamp_ns;    // 64b timestamp (ns). Set at packet arrival
    bit<16> pid;             // 16b packet id used by platform (READ ONLY)
    bit<3>  ingress_port;    // 3b ingress port
    bit<3>  egress_port;     // 3b egress port
    bit<1>  truncate_enable; // 1b set to 1 to enable truncation
    bit<16> truncate_length; // 16b set to desired length of egress pkt
    bit<1>  rss_enable;      // 1b set to 1 to override rss hash result
    bit<12> rss_entropy;     // 12b set to rss_entropy hash value (
    bit<4>  drop_reason;     // reserved (tied to 0).
    bit<32> scratch;         // reserved (tied to 0).
}
```

Listing 6.1. Smartnic Metadata Struct

It is automatically populated by the SmartNIC upon packet arrival on *CMAC* ports. Some of its fields are write-protected, while others are reserved. During the development phase of the INT application, this structure attracted our attention because it provides a subset of the metadata relevant to our study, as presented in section 3.4. However, in all the experiments where the INT source queried the smartNICs for such metadata, the retrieved values were consistently **zero**, as if the structure had never been initialized. This unexpected behaviour suggested two possible explanations: either the metadata fields were not being populated correctly by the hardware, or additional configuration steps of the *open-nic-shell* were required to enable their extraction. To further investigate this issue, a behavioural simulation of the P4 application has been conducted. Such simulation did not exhibit the anomaly, the metadata were successfully extracted from the designated data structure.

6.2.3 Behavioural Simulation

The execution of the P4 behavioural simulation is driven by a specific Makefile available within the ESnet Development Workflow [**?**], which leverages the VNP4 software flow, depicted in Figure 6.1. The software flow involves the following components:

- **P4C_vitisnet Compiler**, which takes the P4 code as input to, compiles it and produces a JSON file as output, to be used by the Behavioural Model.

- **Behavioural Model**, a software component responsible for creating an independent replica of the operations described in the source file, allowing comparison against the expected behaviour and the RTL implementation. It consists of 2 applications: **p4bm-vitisnet**, which models the data plane, and **p4bm-vitisnet-cli** which models the control plane of the emulated device.

The behavioural model also takes as input a stream of packets described in a *pcap* file, along with an input metadata file, where each line holds the metadata values associated with a single packet. The model then generates two corresponding output files: one holding the packets modified and processed according to the P4 logic, the other containing output metadata values.

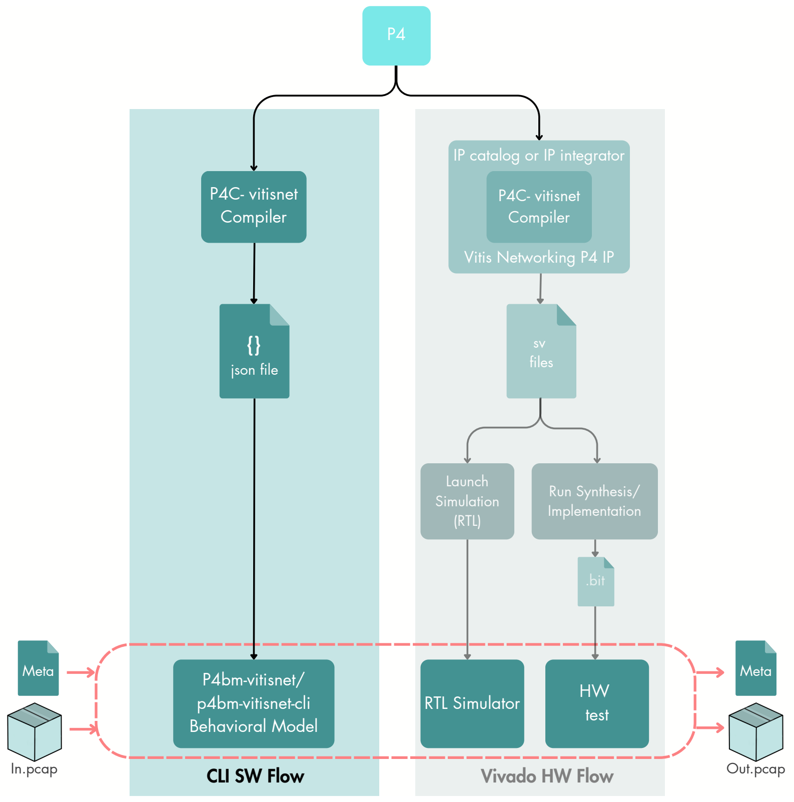


Figure 6.1. Software and Hardware Flows of VNP4. Image reproduced from [?]

Although the simulation was successful and the metadata extraction worked correctly while testing the telemetry application, the set of metadata provided by the device remains limited. Specifically, the **smartnic_metadata** structure does not include valuable metrics such as instantaneous and average resource utilization, for instance the queue occupancy between different processing engines. Several other potentially useful telemetry fields are not yet supported by the SmartNIC used in this study.

6.3 Future Work

While the previous section discussed the limitations and challenges of the presented application, this section outlines possible directions for future work, considering both practical enhancements and more ambitious research paths.

Implementing the INT source and INT sink on the same programmable hardware device would allow testing the telemetry methodology with real application-generated traffic, making the INT domain fully transparent to the application endpoints. This would also enable compatibility with performance testing applications (e.g. *iperf3* and *ping*), allowing evaluation of the maximum achievable throughput and identification of the potential bottleneck in the testbed. Additionally, a complete topology including all roles defined in the INT specification [**?**] would need a dedicated monitoring system, which integration could enable metadata aggregation network analysis generation.

To improve and extend the P4-based telemetry application, a future direction involves enabling the dynamic calculation of each switch's Trust Level, replacing the predefined and hard-coded values in the P4 logic. This metric could be dynamically computed based on the network administrator's intent and could include factors such as the switch's position within the network (core vs. edge), the version of the firmware running on the device and the security policies enforced by the device manufacturer (e.g. access control, anomaly detection). Moreover, once the SmartNIC's set of available and collectable hardware data is expanded, additional insights could be leveraged to refine the trust level calculation further.

The increase in packet header size due to the insertion of switch metadata can be mitigated with various techniques, one of which is the spreading of telemetry metadata across multiple packets within the same data flow. This technique, known as Per Flow Aggregation (PFA) [**?**], helps maintain a "lightweight" in-band network telemetry approach.

Beyond passively collecting network state data, INT could also be leveraged to enforce user-defined network policies, bridging the gap between transparency and control. For instance, specific fields within the packet header could encode user-defined privacy preferences, which would then be used by INT-compatible devices to refine routing decisions based on these constraints [**?**]. Such an implementation on programmable hardware switches is not yet available. As part of an ongoing research effort, we are working on a demonstration project that combines machine learning techniques for automatic path selection and congestion detection, further exploring the intersection between telemetry, network automation, and user-defined policies.

Chapter 7

Appendix

This appendix contains the scripts and complete programs used for this study and referenced in the previous chapters. They are useful for automatizing the configuration of the SmartNICs as well as the insertion of table entries from the control plane of the switches.

7.1 Jupyter Notebook For FABRIC Slice Deployment

```
# Deploying Inband Network Telemetry Domain - Fabric
# ==================================================

# Two compute nodes will include FPGAs. These devices are made available
# as FABRIC components and can be added to your nodes like any other
# component. The project must have Component.FPGA permission tag in order
# to be able to provision them.
# This experiment is deploying a topology that involves 2 FPGA nodes,
# acting as In-band Network Telemetry hardware switches. They are
# programmed using the ESnet Workflow, with a bitstream resulting as the
# compilation of the INT p4 logic. The artifact created under an
# EVALUATION LICENSE with Vivado using the Xilinx proprietary workflow,
# expires after 48 hours.

# Setup the Experiment
# --------------------
from fabrictestbed_extensions.fablib.fablib import FablibManager as
    fablib_manager

fablib = fablib_manager()

fablib.show_config();
import random

FPGA_CHOICE='FPGA_Xilinx_U280'
```

```
# don't edit - convert from FPGA type to a resource column name
# to use in filter lambda function below

choice_to_column = {
"FPGA_Xilinx_U280": "fpga_u280_available",
}

column_name = choice_to_column.get(FPGA_CHOICE, "Unknown")
# The following are the sites hosting an Alveo card pre-flashed with the ESnet Workflow bitfile
allowed_sites = ['CLEM', 'DALL', 'STAR', 'MICH', 'PRIN', 'SALT', 'SRI', 'TACC', 'NCSA', 'WASH', 'UCSD', 'LOSA', 'KANS']

fpga_sites_df = fablib.list_sites(output='pandas', quiet=True, filter_function=lambda x: x[column_name] > 0, force_refresh=True)
# note that list_sites with 'pandas' doesn't actually return a dataframe like doc sez, it returns a Styler based on the dataframe
if fpga_sites_df:
fpga_sites = fpga_sites_df.data['Name'].values.tolist()
else:
fpga_sites = []
print(f'All sites with FPGA available: {fpga_sites}')
if len(fpga_sites)<=1:
print('Warning - no enough sites with available FPGAs found')
else:
if allowed_sites and len(allowed_sites) > 1:
    fpga_sites = list(set(fpga_sites) & set(allowed_sites))
if len(fpga_sites) == 0:
    print('Unable to find sites with available FPGAs')
else:
   print('Selecting a site at random ' + f'among {allowed_sites}' if allowed_sites else '')
   site1, site2 = random.sample(fpga_sites, 2)
   print(f'Preparing to create slice in sites {site1} and {site2}')

# final site override if needed
#site = 'DALL'

# Give the slice and the nodes in it meaningful names.
slice_name=f'INT slice - {site1} + {site2}'


fpga1_name='fpga1'
fpga2_name='fpga2'
sink_node_name='sink-node'
source_node_name='source-node'
bridge1_name='l2bridge1'
bridge2_name='l2bridge2'
bridge3_name='l2bridge3'
```

```

print(f'Will create slice "{slice_name}" across sites {site1}, {site2}')

/* Create a slice with a node with FPGA at desired site
----------------------------------------------------

This slice has four VMs - two with the FPGA and the others with a simple
NIC - we will want to flow traffic across them. */
# Create Slice. Note that by default submit() call will poll for 360
    seconds every 10-20 seconds. Waiting for slice to come up. Normal
    expected time is around 2 minutes.
slice = fablib.new_slice(name=slice_name)
image = 'docker_ubuntu_20'

# Add node with a 70G drive and 8 CPU cores using Ubuntu 20 image - INT
    node 1
int_node1 = slice.add_node(name=fpga1_name, site=site1, cores=8, ram=16,
    disk=70, image=image)
fpga1_comp = int_node1.add_component(model=FPGA_CHOICE, name='fpga1')
fpga1_p1 = fpga1_comp.get_interfaces()[0]
fpga1_p2 = fpga1_comp.get_interfaces()[1]

# Add node with a 70G drive and 8 CPU cores using Ubuntu 20 image - INT
    node 2
int_node2 = slice.add_node(name=fpga2_name, site=site2, cores=8, ram=16,
    disk=70, image=image)
fpga2_comp = int_node2.add_component(model=FPGA_CHOICE, name='fpga2')
fpga2_p1 = fpga2_comp.get_interfaces()[0]
fpga2_p2 = fpga2_comp.get_interfaces()[1]

# Add two more nodes acting as source and destination of the traffice flow
src = slice.add_node(name=source_node_name, site=site1, cores=4, disk=50,
    image=image)
src_iface=src.add_component(model='NIC_Basic', name='nic1').get_interfaces
    ()[0]
src_iface.set_mode('auto')

dst = slice.add_node(name=sink_node_name, site=site2, cores=4, disk=50,
    image=image)
dst_iface=dst.add_component(model='NIC_Basic', name='nic2').get_interfaces
    ()[0]
dst_iface.set_mode('auto')

# Use L2Bridge network services to complete the topolgy
net1 = slice.add_l2network(name=bridge1_name, interfaces=[fpga1_p1,
    src_iface], type='L2Bridge')
net2 = slice.add_l2network(name=bridge2_name, interfaces=[fpga1_p2,
    fpga2_p1])
```

```
net3 = slice.add_l2network(name=bridge3_name, interfaces=[fpga2_p2,
    dst_iface], type='L2Bridge')


# Submit Slice Request
slice.submit();
/* Setup IOMMU and Hugepages
=========================

For DPDK to function properly we need to setup hugepages and IOMMU on
the VM, this is useful in case we want to generate traffic directly on
one of the VMs with the FPGA instead of using one of the simple nodes
(source and sink). */
#First of all, check if the FPGA is detected in both its PFs on the two INT
     nodes
command = "lspci -Dd 10ee:"
int_node1 = slice.get_node(name=fpga1_name)
int_node2 = slice.get_node(name=fpga2_name)
print("Node 1")
stdout, stderr = int_node1.execute(command)
print("Node 2")
stdout, stderr = int_node2.execute(command)

slice = fablib.get_slice(name=slice_name)
int_node1 = slice.get_node(name=fpga1_name)
int_node2 = slice.get_node(name=fpga2_name)

commands = list()
#commands.append("sudo sed -i 's/GRUB_CMDLINE_LINUX=\"\\(.*\\)\"/
    GRUB_CMDLINE_LINUX=\"\\1 amd_iommu=on iommu=pt default_hugepagesz=1G
    hugepagesz=1G hugepages=8\"/' /etc/default/grub")
commands.append("sudo sed -i 's/GRUB_CMDLINE_LINUX=\"\"/GRUB_CMDLINE_LINUX
    =\"amd_iommu=on iommu=pt default_hugepagesz=1G hugepagesz=1G hugepages
    =8\"/' /etc/default/grub")
commands.append("sudo grub-mkconfig -o /boot/grub/grub.cfg")
commands.append("sudo update-grub")

for command in commands:
    print(f'Executing {command}')
    stdout, stderr = int_node1.execute(command)

for command in commands:
    print(f'Executing {command}')
    stdout, stderr = int_node2.execute(command)

print('Done')
#Reboot the node (this sometimes generates an EOFError exception - ignore
it and continue)
reboot = 'sudo reboot'
```

```

print(reboot)
int_node1.execute(reboot)
print(reboot)
int_node2.execute(reboot)

slice.wait_ssh(timeout=360,interval=10,progress=True)

print("Now testing SSH abilites to reconnect...",end="")
slice.update()
slice.test_ssh()
print("Reconnected!")

# Check that IOMMU was enabled


slice = fablib.get_slice(slice_name)
int_node1=slice.get_node(fpga1_name)
int_node2=slice.get_node(fpga2_name)

command = 'dmesg | grep -i IOMMU'

print('Observe that the modifications to boot configuration took place and
    IOMMU is detected')
stdout, stderr = int_node1.execute(command)
stdout, stderr = int_node2.execute(command)


int_node1.config()
int_node2.config()

# Disable IOMMU support in VFIO (the passing through doesn't actually
work)
# Enable unsafe_noiommu_mode for the vfio module
command = "echo 1 | sudo tee /sys/module/vfio/parameters/
    enable_unsafe_noiommu_mode"

stdout, stderr = int_node1.execute(command)
stdout, stderr = int_node2.execute(command)

# Install Docker compose
# ----------------------
commands = ["sudo usermod -aG docker ubuntu",
            "newgrp docker",
            "mkdir -p ~/.docker/cli-plugins/",
            "curl -SL https://github.com/docker/compose/releases/download/
    v2.27.2/docker-compose-linux-x86_64 -o ~/.docker/cli-plugins/docker-
    compose",
            "chmod +x ~/.docker/cli-plugins/docker-compose",
            "curl -SL https://github.com/docker/buildx/releases/download/v0
    .11.2/buildx-v0.11.2.linux-amd64 -o ~/.docker/cli-plugins/docker-buildx
    ",
            "chmod +x ~/.docker/cli-plugins/docker-buildx",
            "docker compose version",
           ]

for command in commands:
    print(f'Executing {command} on node 1')
    stdout, stderr = int_node1.execute(command)
    print(f'Executing {command} on node 2')
    stdout, stderr = int_node2.execute(command)

print('Done')

/* Program FPGA and run applications on it
---------------------------------------

First we - download pre-built dpdk and xilinx-labtools containers and
install their images - download a previously built p4 artifact -
checkout the ``esnet-smartnic-fw`` code, add the artifact and build a
configuration of containers we can then execute. */
# Build dpdk and xilinx-labtools docker images, then place your artifact
    file in ~/
commands = [
    "git clone https://github.com/esnet/smartnic-dpdk-docker.git",
    "cd ~/smartnic-dpdk-docker; git submodule update --init --recursive",
    "cd ~/smartnic-dpdk-docker; docker build --pull -t smartnic-dpdk-docker
    :${USER}-dev .",
    "docker image ls"
]

for command in commands:
    print(f'Executing {command}')
    stdout, stderr = int_node1.execute(command)
    stdout, stderr = int_node2.execute(command)

print("\n\nSCP from node attached to the persistent storage, the artifact
    file and the xilinx-labtools-docker image")

# update the env_file values to match the name of the artifact file
env_file = """
SN_HW_VER=0
SN_HW_BOARD=au280
SN_HW_APP_NAME=p4_only
"""

commands = [
    "git clone https://github.com/esnet/esnet-smartnic-fw.git",
    "cd ~/esnet-smartnic-fw; git submodule init; git submodule update",
    f"echo '{env_file}' | sudo tee ~/esnet-smartnic-fw/.env",
]

for command in commands:
    print(f'Executing {command}')
    stdout, stderr = int_node1.execute(command)
    stdout, stderr = int_node2.execute(command)
print("Done")

commands = [
    "docker pull lucacet/xilinx-labtools-docker:ubuntu",
    "docker tag lucacet/xilinx-labtools-docker:ubuntu xilinx-labtools-
    docker:ubuntu-dev",
    "docker image rm lucacet/xilinx-labtools-docker:ubuntu"
]

for command in commands:
    print(f"Executing {command} - 1")
    stdout, stderr = int_node1.execute(command)
    print(f"Executing {command} - 2")
    stdout, stderr = int_node2.execute(command)

print("Done")
/* Test sn-cfg, configure CMACs
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Should see normal looking output. If everything is 0x0000 or 0xffff, the
binding to FPGA from VFIO did not work. */
command = "cd esnet-smartnic-fw/sn-stack/; docker compose exec smartnic-fw
    sn-cfg --tls-insecure show device"

stdout, stderr = int_node1.execute(command)
stdout, stderr = int_node2.execute(command)

/* Let's configure CMACs. It succeeds if we see ''MAC ENABLED/PHY UP -> UP'
    ' for both CMACs Rx and Tx. If
not, it is possible FEC is not turned off in the dataplane switch. */

# upload sn-cli config script
sn_cli_script = 'sn-cli-setup'

result = int_node1.upload_file(sn_cli_script, sn_cli_script)
result = int_node2.upload_file(sn_cli_script, sn_cli_script)

commands = [
    f"chmod a+x {sn_cli_script}",
    f"mv {sn_cli_script} ~/esnet-smartnic-fw/sn-stack/scratch",
```

```
    f"cd ~/esnet-smartnic-fw/sn-stack/; docker compose exec smartnic-fw
    scratch/{sn_cli_script}"
]

for command in commands:
    print(f'Executing {command}')
    stdout, stderr = node1.execute(command)

/* Extend the slice (as needed)
----------------------------

If you need to extend the storage slice, you can just execute the
following two cells. They display the slice expiration date and
optionally extend by 2 weeks. */



slice_name="INT slice - CLEM + KANS"
slice = fablib.get_slice(name=slice_name)
nets = slice.list_networks()
nodes = slice.get_nodes()


# Renew the slice

from datetime import datetime
from datetime import timezone
from datetime import timedelta

# Set end host to now plus 14 days
end_date = (datetime.now(timezone.utc) + timedelta(days=14)).strftime("%Y-%
    m-%d %H:%M:%S %z")

try:
    slice = fablib.get_slice(name=slice_name)

    slice.renew(end_date)
except Exception as e:
    print(f"Exception: {e}")

/* Delete the Slice (as needed)
---------------------------- */

slice = fablib.get_slice(name=slice_name)
slice.delete()
```

Listing 7.1. Jupyter Notebook for Slice Definition and Deployment

7.2 Bash Scripts for SmartNIC Confiuguration

```bash
#!/bin/bash
sn-cfg batch configure-switch -s port0:port0 -s port1:port1 -s host0:host0
    -s host1:host1
sn-cfg batch configure-switch -i port0:app0 -i port1:app0
sn-cfg batch configure-switch -e app0:port0:port0 -e app0:port1:port1

sn-cfg --tls-insecure batch configure-port -p 0 -s enable
sn-cfg --tls-insecure batch configure-port -p 1 -s enable

sn-cfg batch show-switch-config
```

Listing 7.2. Script for SmartNIC's Internal Switch Configuration

```bash
#!/bin/bash

SN_P4_CMD="sn-p4 --tls-insecure insert table rule"

if [[ $# -ne 1 || ( $1 != "s1" && $1 != "s2" ) ]]; then
  echo "Usage: $0 <s1|s2>"
  exit 1
fi

SW=$1

configure_int_operation() {
  printf "\nConfiguring int_operation table for $SW...\n"
  if [[ $SW == "s1" ]]; then
    $SN_P4_CMD -t int_operation -m 0x0001 --action sw0_push_id --param 0
   x101
    $SN_P4_CMD -t int_operation -m 0x0002 --action
   sw0_push_ingress_timestamp
    $SN_P4_CMD -t int_operation -m 0x0004 --action sw0_push_trust_level
  else
    $SN_P4_CMD -t int_operation -m 0x0001 --action sw1_push_id --param 0
   x201
    $SN_P4_CMD -t int_operation -m 0x0002 --action
   sw1_push_ingress_timestamp
    $SN_P4_CMD -t int_operation -m 0x0004 --action sw1_push_trust_level
  fi
}
configure_l2_forwarding() {
  printf "\n\nConfiguring l2_forwarding table...\n"
  if [[ $SW == "s1" ]]; then
    $SN_P4_CMD -t l2_forwarding -m 0x0A671380B76D --action l2_forward --
   param 1
    $SN_P4_CMD -t l2_forwarding -m 0x029AF99EEF7F --action l2_forward --
   param 0
  else
    $SN_P4_CMD -t l2_forwarding -m 0x0A671380B76D --action l2_forward --
   param 1
    $SN_P4_CMD -t l2_forwarding -m 0x029AF99EEF7F --action l2_forward --
   param 0
  fi
}

# Clear the table before applying configuration
sn-p4 --tls-insecure clear table
# Call configuration functions
configure_int_operation
#configure_ipv4_routing
configure_l2_forwarding

printf "\n\n\nP4 table configuration completed for $SW\n"
```

Listing 7.3. Script for Table Entries Insertion In SmartNIC's Control Plane

7.3 Python Script for Traffic Generation And Analysis

```
from scapy.all import *
import binascii

class Ethernet(Packet):
    name = "Ethernet"
    fields_desc = [
        MACField("dstAddr", "00:00:00:00:00:00"),
        MACField("srcAddr", "00:00:00:00:00:00"),
        XShortField("etherType", 0x0800),
    ]

class IPv4(Packet):
    name = "IPv4"
    fields_desc = [
        BitField("version", 4, 4),
        BitField("ihl", 5, 4),
        ByteField("diffserv", 0),
        ShortField("totalLen", 0),
        ShortField("identification", 0),
        BitField("flags", 2, 3),
        BitField("fragOffset", 0, 13),
        ByteField("ttl", 64),
        ByteField("protocol", 0x2F),  # GRE
        XShortField("hdrChecksum", 0),
        IPField("srcAddr", "10.0.0.1"),
        IPField("dstAddr", "10.0.0.2"),
    ]

class GRE(Packet):
    name = "GRE"
    fields_desc = [
        BitField("C", 0, 1),
        BitField("R", 0, 1),
        BitField("K", 0, 1),
        BitField("S", 0, 1),
        BitField("s", 0, 1),
        BitField("recursion", 0, 3),
        BitField("flags", 0, 5),
        BitField("version", 0, 3),
        XShortField("protocol_type", 0x1717), #INT custom Ethernet Type
    ]

class IntShim(Packet):
    name = "IntShim"
    fields_desc = [
        BitField("type", 0x3, 4),
        BitField("G", 1, 1),
        BitField("Rsvd", 0, 3),
        ByteField("length", 20),
        XShortField("next_protocol", 0x0800),  # IPv4 after INT stack
    ]

class IntHeader(Packet):
    name = "IntHeader"
    fields_desc = [
        BitField("ver", 0x2, 4),
        BitField("D", 0, 1),
        BitField("E", 0, 1),
        BitField("M", 0, 1),
        BitField("reserved", 0, 12),
        BitField("hop_ml", 48, 5),
        ByteField("remaining_hop_cnt", 32),
        XShortField("instruction_mask", 0x0001),
        XShortField("domain_id", 1),
        XShortField("ds_instr", 0x0001),
        XShortField("ds_flags", 0),
    ]

class SwitchInt0(Packet):
    name = "SwitchInt0"
    fields_desc = [
        XShortField("swid", 0),
        XShortField("trust_level", 0),
        XLongField("timestamp", 0),
    ]
```

```

class SwitchInt1(Packet):
    name = "SwitchInt1"
    fields_desc = [
        XShortField("swid", 0),
        XShortField("trust_level", 0),
        XLongField("timestamp", 0),
    ]

bind_layers(Ethernet, IPv4, etherType=0x0800)
bind_layers(IPv4, GRE, protocol=0x2F)
bind_layers(GRE, IntShim, protocol_type=0x1717)
bind_layers(IntShim, IntHeader)
bind_layers(IntHeader, SwitchInt0)
bind_layers(SwitchInt0, SwitchInt1)

dst_mac = "06:AD:E2:A9:34:C9"
src_mac = "06:03:79:10:FC:85"
pkt = (
    Ethernet(dstAddr=dst_mac, srcAddr=src_mac, etherType=0x0800) /
    IPv4(srcAddr="192.168.1.2", dstAddr="192.168.1.3", totalLen=0x420) /
    GRE() /
    IntShim(next_protocol=0x0800) /
    IntHeader(remaining_hop_cnt=32, ds_instr=0x0001) /
    SwitchInt0() /
    SwitchInt1()
)

print("Packet bytes in hexadecimal:")
print(binascii.hexlify(bytes(pkt)).decode('utf-8'))

pkt.show()

iface = "enp7s0"

resp = srp1(pkt, iface=iface, timeout=3, verbose=False)
```

Listing 7.4. Python Script for INT Source Emulation

```python
from scapy.all import *
import time


packet_size = 78
data_rate = 100  # Desired rate in Mbps
num_packets = 100  # Total packets to be sent

class Ethernet(Packet):
    name = "Ethernet"
    fields_desc = [
        MACField("dstAddr", "00:00:00:00:00:00"),
        MACField("srcAddr", "00:00:00:00:00:00"),
        XShortField("etherType", 0x0800),
]

class IPv4(Packet):
    name = "IPv4"
    fields_desc = [
        BitField("version", 4, 4),
        BitField("ihl", 5, 4),
        ByteField("diffserv", 0),
        ShortField("totalLen", 0),
        ShortField("identification", 0),
        BitField("flags", 2, 3),
        BitField("fragOffset", 0, 13),
        ByteField("ttl", 64),
        ByteField("protocol", 0x2F),  # GRE
        XShortField("hdrChecksum", 0),
        IPField("srcAddr", "10.0.0.1"),
        IPField("dstAddr", "10.0.0.2"),
    ]

class GRE(Packet):
    name = "GRE"
    fields_desc = [
        BitField("C", 0, 1),
        BitField("R", 0, 1),
        BitField("K", 0, 1),
        BitField("S", 0, 1),
        BitField("s", 0, 1),
        BitField("recursion", 0, 3),
        BitField("flags", 0, 5),
        BitField("version", 0, 3),
        XShortField("protocol_type", 0x1717), # custom INT ethertype
    ]

class IntShim(Packet):
    name = "IntShim"
    fields_desc = [
        BitField("type", 0x3, 4),
        BitField("G", 1, 1),
        BitField("Rsvd", 0, 3),
        ByteField("length", 20),
        XShortField("next_protocol", 0x0800),  # IPv4 after INT stack
    ]

class IntHeader(Packet):
    name = "IntHeader"
    fields_desc = [
        BitField("ver", 0x2, 4),
        BitField("D", 0, 1),
        BitField("E", 0, 1),
        BitField("M", 0, 1),
        BitField("reserved", 0, 12),
        BitField("hop_ml", 48, 5),
        ByteField("remaining_hop_cnt", 32),
        XShortField("instruction_mask", 0x0001),
        XShortField("domain_id", 1),
        XShortField("ds_instr", 0x0001),
        XShortField("ds_flags", 0),
    ]

class SwitchInt0(Packet):
    name = "SwitchInt0"
    fields_desc = [
        XShortField("swid", 0),
        XShortField("trust_level", 0),
        XLongField("timestamp", 0),
    ]

class SwitchInt1(Packet):
    name = "SwitchInt1"
    fields_desc = [
        XShortField("swid", 0),
        XShortField("trust_level", 0),
        XLongField("timestamp", 0),
    ]

bind_layers(Ethernet, IPv4, etherType=0x0800)
bind_layers(IPv4, GRE, protocol=0x2F)
bind_layers(GRE, IntShim, protocol_type=0x1717)
bind_layers(IntShim, IntHeader)
bind_layers(IntHeader, SwitchInt0)
bind_layers(SwitchInt0, SwitchInt1)

dst_mac = "0A:67:13:80:B7:6D"
src_mac = "02:9A:F9:9E:EF:7F"
pkt = (
    Ethernet(dstAddr=dst_mac, srcAddr=src_mac, etherType=0x0800) /
    IPv4(srcAddr="192.168.1.2", dstAddr="192.168.1.3", totalLen=0x4E) /
    GRE() /
    IntShim(next_protocol=0x0800) /
    IntHeader(remaining_hop_cnt=32, ds_instr=0x0001) /
    SwitchInt0() /
    SwitchInt1()
)


```

```
print(f"Sending {num_packets} packets at {data_rate} Mbps...")

# Send packets at desired data rate
start_time = time.time()
sendpfast([pkt] * num_packets, mbps=data_rate, loop=1, iface="enp7s0")
end_time = time.time()
print("ok")
duration = end_time - start_time

print(f"Packet transmission completed in {duration:.2f} seconds.")
print(f"Actual transmission rate: {packet_size * num_packets * 8 / (1000000
     * duration)}:.2f Mbps")
```

Listing 7.5. Python Script for INT Source Emulation - Controlled Bitrate For Accuracy Test

```
from scapy.all import *
import time

DST_MAC = "06:AD:E2:A9:34:C9"
SRC_MAC = "06:03:79:10:FC:85"
PAYLOAD = "Performance test packet"
NUM_PACKETS = 1000
ETHERTYPE_TBD_INT = 0x1717

class Ethernet(Packet):
    name = "Ethernet"
    fields_desc = [
        MACField("dstAddr", "00:00:00:00:00:00"),
        MACField("srcAddr", "00:00:00:00:00:00"),
        XShortField("etherType", 0x0800),
    ]

class IPv4(Packet):
    name = "IPv4"
    fields_desc = [
        BitField("version", 4, 4),
        BitField("ihl", 5, 4),
        ByteField("diffserv", 0),
        ShortField("totalLen", 0),
        ShortField("identification", 0),
        BitField("flags", 2, 3),
        BitField("fragOffset", 0, 13),
        ByteField("ttl", 64),
        ByteField("protocol", 0x2F),  # GRE
        XShortField("hdrChecksum", 0),
        IPField("srcAddr", "10.0.0.1"),
        IPField("dstAddr", "10.0.0.2"),
    ]


```

```
class GRE(Packet):
    name = "GRE"
    fields_desc = [
        BitField("C", 0, 1),
        BitField("R", 0, 1),
        BitField("K", 0, 1),
        BitField("S", 0, 1),
        BitField("s", 0, 1),
        BitField("recursion", 0, 3),
        BitField("flags", 0, 5),
        BitField("version", 0, 3),
        XShortField("protocol_type", ETHERTYPE_TBD_INT), # INT encapsulated
    ]

class IntShim(Packet):
    name = "IntShim"
    fields_desc = [
        BitField("type", 0x3, 4),
        BitField("G", 1, 1),
        BitField("Rsvd", 0, 3),
        ByteField("length", 20),
        XShortField("next_protocol", 0x0800),  # IPv4 after INT stack
    ]

class IntHeader(Packet):
    name = "IntHeader"
    fields_desc = [
        BitField("ver", 0x2, 4),
        BitField("D", 0, 1),
        BitField("E", 0, 1),
        BitField("M", 0, 1),
        BitField("reserved", 0, 12),
        BitField("hop_ml", 48, 5),
        ByteField("remaining_hop_cnt", 32),
        XShortField("instruction_mask", 0x0001),
        XShortField("domain_id", 1),
        XShortField("ds_instr", 0x0001),
        XShortField("ds_flags", 0),
    ]

class SwitchInt0(Packet):
    name = "SwitchInt0"
    fields_desc = [
        XShortField("swid", 0),
        XShortField("trust_level", 0),
        XLongField("timestamp", 0),
    ]

class SwitchInt1(Packet):
    name = "SwitchInt1"
    fields_desc = [
        XShortField("swid", 0),
        XShortField("trust_level", 0),
        XLongField("timestamp", 0),
    ]

bind_layers(Ethernet, IPv4, etherType=0x0800)
bind_layers(IPv4, GRE, protocol=0x2F)
bind_layers(GRE, IntShim, protocol_type=0x1717)
bind_layers(IntShim, IntHeader)
bind_layers(IntHeader, SwitchInt0)
bind_layers(SwitchInt0, SwitchInt1)


def send_and_measure(packet, iface):
    latencies = []
    conf.sniff_promisc = True # ensure to accept returning packets

    def filter_response(response):
            return response.haslayer(Ether) and (response[Ether].dst == "2a
    :2b:2c:2d:2e:2f" or response[Ether].dst == "06:ad:e2:a9:34:00")

    for _ in range(NUM_PACKETS):
        start_time = time.time()  # Timestamp before sending
        response = sniff(iface=iface, timeout=2, count=1, lfilter=
    filter_response, started_callback=lambda: sendp(packet, iface=iface,
    verbose=False))
        end_time = time.time()  # Timestamp after reception

        if response:
            latency = (end_time - start_time) * 1000  # in ms
            latencies.append(latency)
            print(f"Packet RTT: {latency:.2f} ms")
        else:
            print("No response received")
    return latencies

# Normal packet (no INT)
def create_normal_packet():
    return  Ethernet(dstAddr="06:AD:E2:A9:34:00", srcAddr=SRC_MAC,
    etherType=0x0800) / IPv4(srcAddr=SRC_IP, dstAddr=DST_IP, totalLen=0x420
    , protocol=0x11) / UDP() / PAYLOAD

# Packet wirh INT header - source emulation
def create_int_packet():
    return Ethernet(dstAddr="06:AD:E2:A9:34:00", srcAddr="06:03:79:10:FC:91
", etherType=0x0800) / IPv4(srcAddr=SRC_IP, dstAddr=DST_IP, totalLen=0
x420) / GRE() / IntShim(next_protocol=0x0800) / IntHeader(
remaining_hop_cnt=32, ds_instr=0x0001) / SwitchInt0() / SwitchInt1()

def main():
    iface = "enp7s0"
    print("Testing normal packets...")
    normal_packet = create_normal_packet()
    #normal_packet.show()
    normal_latencies = send_and_measure(normal_packet, iface)

    print("\nTesting INT packets...")
    int_packet = create_int_packet()
    #int_packet.show()
    int_latencies = send_and_measure(int_packet, iface)

    print("\nResults:")
    if len(normal_latencies) != 0:
        print(f"Normal Packet Avg RTT: {sum(normal_latencies) / len(
normal_latencies):.2f} ms")
    else:
         print(f"Normal Packet Avg RTT: {sum(normal_latencies) / 1:.2f} ms"
)

    if len(int_latencies) != 0:
        print(f"INT Packet Avg RTT: {sum(int_latencies) /  len(
int_latencies):.2f} ms")
    else:
         print(f"INT Packet Avg RTT: {sum(int_latencies) / 1:.2f} ms")


if __name__ == "__main__":
    main()
from scapy.all import *
import time

DST_MAC = "06:AD:E2:A9:34:C9"
SRC_MAC = "06:03:79:10:FC:85"
PAYLOAD = "Performance test packet"
NUM_PACKETS = 1000
ETHERTYPE_TBD_INT = 0x1717

class Ethernet(Packet):
    name = "Ethernet"
    fields_desc = [
        MACField("dstAddr", "00:00:00:00:00:00"),
        MACField("srcAddr", "00:00:00:00:00:00"),
        XShortField("etherType", 0x0800),
```

```python
    ]

class IPv4(Packet):
    name = "IPv4"
    fields_desc = [
        BitField("version", 4, 4),
        BitField("ihl", 5, 4),
        ByteField("diffserv", 0),
        ShortField("totalLen", 0),
        ShortField("identification", 0),
        BitField("flags", 2, 3),
        BitField("fragOffset", 0, 13),
        ByteField("ttl", 64),
        ByteField("protocol", 0x2F),  # GRE
        XShortField("hdrChecksum", 0),
        IPField("srcAddr", "10.0.0.1"),
        IPField("dstAddr", "10.0.0.2"),
    ]


class GRE(Packet):
    name = "GRE"
    fields_desc = [
        BitField("C", 0, 1),
        BitField("R", 0, 1),
        BitField("K", 0, 1),
        BitField("S", 0, 1),
        BitField("s", 0, 1),
        BitField("recursion", 0, 3),
        BitField("flags", 0, 5),
        BitField("version", 0, 3),
        XShortField("protocol_type", ETHERTYPE_TBD_INT), # INT encapsulated
    ]

class IntShim(Packet):
    name = "IntShim"
    fields_desc = [
        BitField("type", 0x3, 4),
        BitField("G", 1, 1),
        BitField("Rsvd", 0, 3),
        ByteField("length", 20),
        XShortField("next_protocol", 0x0800),  # IPv4 after INT stack
    ]

class IntHeader(Packet):
    name = "IntHeader"
    fields_desc = [
        BitField("ver", 0x2, 4),
        BitField("D", 0, 1),
        BitField("E", 0, 1),
        BitField("M", 0, 1),
        BitField("reserved", 0, 12),
        BitField("hop_ml", 48, 5),
        ByteField("remaining_hop_cnt", 32),
        XShortField("instruction_mask", 0x0001),
        XShortField("domain_id", 1),
        XShortField("ds_instr", 0x0001),
        XShortField("ds_flags", 0),
    ]

class SwitchInt0(Packet):
    name = "SwitchInt0"
    fields_desc = [
        XShortField("swid", 0),
        XShortField("trust_level", 0),
        XLongField("timestamp", 0),
    ]

class SwitchInt1(Packet):
    name = "SwitchInt1"
    fields_desc = [
        XShortField("swid", 0),
        XShortField("trust_level", 0),
        XLongField("timestamp", 0),
    ]

bind_layers(Ethernet, IPv4, etherType=0x0800)
bind_layers(IPv4, GRE, protocol=0x2F)
bind_layers(GRE, IntShim, protocol_type=0x1717)
bind_layers(IntShim, IntHeader)
bind_layers(IntHeader, SwitchInt0)
bind_layers(SwitchInt0, SwitchInt1)


def send_and_measure(packet, iface):
    latencies = []
    conf.sniff_promisc = True # ensure to accept returning packets

    def filter_response(response):
            return response.haslayer(Ether) and (response[Ether].dst == "2a
    :2b:2c:2d:2e:2f" or response[Ether].dst == "06:ad:e2:a9:34:00")

    for _ in range(NUM_PACKETS):
        start_time = time.time()  # Timestamp before sending
        response = sniff(iface=iface, timeout=2, count=1, lfilter=
    filter_response, started_callback=lambda: sendp(packet, iface=iface,
    verbose=False))
        end_time = time.time()  # Timestamp after reception
```

```python

        if response:
            latency = (end_time - start_time) * 1000  # in ms
            latencies.append(latency)
            print(f"Packet RTT: {latency:.2f} ms")
        else:
            print("No response received")
    return latencies

# Normal packet (no INT)
def create_normal_packet():
    return  Ethernet(dstAddr="06:AD:E2:A9:34:00", srcAddr=SRC_MAC, etherType=0x0800) / IPv4(srcAddr=SRC_IP, dstAddr=DST_IP, totalLen=0x420, protocol=0x11) / UDP() / PAYLOAD

# Packet wirh INT header - source emulation
def create_int_packet():
    return Ethernet(dstAddr="06:AD:E2:A9:34:00", srcAddr="06:03:79:10:FC:91", etherType=0x0800) / IPv4(srcAddr=SRC_IP, dstAddr=DST_IP, totalLen=0x420) / GRE() / IntShim(next_protocol=0x0800) / IntHeader(remaining_hop_cnt=32, ds_instr=0x0001) / SwitchInt0() / SwitchInt1()

def main():
    iface = "enp7s0"
    print("Testing normal packets...")
    normal_packet = create_normal_packet()
    #normal_packet.show()
    normal_latencies = send_and_measure(normal_packet, iface)

    print("\nTesting INT packets...")
    int_packet = create_int_packet()
    #int_packet.show()
    int_latencies = send_and_measure(int_packet, iface)

    print("\nResults:")
    if len(normal_latencies) != 0:
        print(f"Normal Packet Avg RTT: {sum(normal_latencies) / len(normal_latencies):.2f} ms")
    else:
         print(f"Normal Packet Avg RTT: {sum(normal_latencies) / 1:.2f} ms")

    if len(int_latencies) != 0:
        print(f"INT Packet Avg RTT: {sum(int_latencies) /  len(int_latencies):.2f} ms")
    else:
         print(f"INT Packet Avg RTT: {sum(int_latencies) / 1:.2f} ms")


```

```python
if __name__ == "__main__":
    main()
```

Listing 7.6. Python Script for Performance Test

```python
from datetime import datetime
import matplotlib.pyplot as plt
import numpy as np

def parse_tcpdump(file_path):
    without_int = []
    with_int = []
    with open(file_path, 'r') as f:
        packets = f.readlines()
        mid_point = len(packets) // 2
        for i, line in enumerate(packets):
            if "192.168.1.2" in line and "192.168.1.3" in line:
                ts = line.split(' ')[0] + ' ' + line.split(' ')[1]
                if i < mid_point:
                    without_int.append(ts)
                else:
                    with_int.append(ts)
    return without_int, with_int

def calculate_delay(timestamps):
    fmt = "%Y-%m-%d %H:%M:%S.%f"
    deltas = []
    for i in range(0, len(timestamps) - 1, 2):
        time1 = datetime.strptime(timestamps[i], fmt)
        time2 = datetime.strptime(timestamps[i + 1], fmt)
        deltas.append((time2 - time1).total_seconds() * 1e3)  # Convert to
milliseconds
    return deltas

def moving_average(data, window_size=10):
    return np.convolve(data, np.ones(window_size)/window_size, mode='valid'
)

def generate_plot(without_int_delays, with_int_delays, output_file):
    plt.figure(figsize=(10, 6))

    # Smoothed moving average lines
    if len(without_int_delays) >= 10:
        smoothed_without_int = moving_average(without_int_delays)
        plt.plot(range(10, len(without_int_delays) + 1),
smoothed_without_int, linestyle='-', color='blue', label="Without INT")

    if len(with_int_delays) >= 10:
        smoothed_with_int = moving_average(with_int_delays)
        plt.plot(range(10, len(with_int_delays) + 1), smoothed_with_int, linestyle='-', color='red', label="With INT")

    plt.title("Processing Delay Comparison")
    plt.xlabel("Packet Number")
    plt.ylabel("Delay (ms)")  # Now in milliseconds
    plt.legend()
    plt.grid(True)
    plt.savefig(output_file)
    print(f"Graph saved to: {output_file}")

file_path = "packets.log"
output_file = "processing_delay_comparison.png"

without_int, with_int = parse_tcpdump(file_path)
without_int_delays = calculate_delay(without_int)
with_int_delays = calculate_delay(with_int)

generate_plot(without_int_delays, with_int_delays, output_file)
```

Listing 7.7. Python Script for Parsing Tcpdump Output and Generating Plots