UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

Design, implementation and benchmark of a RINA-based virtual networking solution for distributed VNFs

Master Thesis
submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by

Sergio Giménez Antón

In partial fulfillment
of the requirements for the master in
Master's degree in Advanced Telecommunication Technologies (MATT)

Advisor: Jordi Perelló
Director: Eduard Grasa
Barcelona, January 2022

Contents

List of Figures

Acknowledgements

I would like to express my deepest appreciation to Eduard Grasa (Fundació i2CAT), who already two years ago started guiding me towards learning an amazing and bleeding edge technology such is RINA. He has been (and still is) the best mentor I could have had. Almost all the knowledge inside this thesis, is here thanks to his great ability to teach and tackle all the issues that have appeared throughout this journey.

I'm deeply grateful to thank Vincenzo Maffione (Università degli Studi di Pisa) for all the support he gave regarding all the issues I encountered implementing the netmap-based data path of the RINA stack prototype.

I also wish to thank Jordi Perelló (Universitat Politècnica de Catalunya) for his willingness to be the advisor of this work, and all the support he gave during the journey of writing this thesis.

Last but certainly not least. Thank all my family, for always being there. Especially my father, for instilling in me from a young age the passion, curiosity and admiration for the greatness of science and technology; and my mother, for her unconditional support in absolutely all the decisions that I have made throughout this journey.

This research was supported by the Spanish Center for the Development of Industrial Technology (CDTI) and the Ministry of Economy, Industry and Competitiveness under grant/project CER-20191015/Open, Virtualized Technology Demonstrators for Smart Networks (Open-VERSO)

Abstract

Scenarios where network functions are getting decoupled from the hardware they are running on is appearing as a common use case nowadays. Datacenters distributed in different network segments need to run this functions virtually and orchestrate them, which is not a simple task. A promising solution to such challenging environment is to use RINA as a network virtualization framework to support applications in the cloud, which natively brings capabilities that current network architectures need to add as ad-hoc solutions, such as support for mobility, multi-homing and flexible support for mapping application QoS requests in internal network policies.

This thesis has carried out the design, implementation and initial benchmarking of a software-based, performing RINA implementation that provides a network solution to provide connectivity in a distributed VNF context.

1 Introduction

Network operators are moving towards a life cycle model (design, implementation, operation) generally known as "Carrier Cloud" or "Telco Cloud" [2]. This model provides the ability to decouple instantiation of network functions from the platforms where they will run, which can be thought of as a set of Data Centers (DC) of different sizes, distributed across multiple network segments (edge, aggregation and core). Telco clouds enable operators to adopt agile operating practices similar to those of large application service providers and the web-scale cloud players [3]. However, to implement this model, network operators must overcome important challenges.

Current Virtual Network Function (VNF) orchestration solutions are hampered by the inability of current network protocols to dynamically create adequate connectivity [1] and the legacy of current standards that encourage the proliferation of protocols that attack point solutions; complicating dynamic management of connectivity. Orchestration of network resources, based on intention, is required; but performed efficiently and based on open standards, including: isolation of connectivity between different services, security management, dynamic allocation of directions, route selection and application of quality of service.

VNFs within Telco Clouds require secure connectivity environments with the right amount of quality, isolated from interference from services owned by other tenants. A promising solution to such challenging environment is to use RINA - the Recursive InterNetwork Architecture - as a network virtualization framework to support applications in the cloud, combining traditional functions of network protocols and service meshes [34] functions in a single integrated solution. This approach leverages the benefits of RINA to efficiently address VNF connectivity requirements, such as: application independent namespaces [5], dynamic, secure and customizable connectivity environments [6], support for mobility and multi-homing without the need for dedicated protocols and flexible support for mapping application QoS requests in internal network policies [7].

All existing implementations of the RINA architecture to date are software-based and not designed for high-performance systems. IRATI [9] and rlite [33] are C/C++ based implementations for Linux hosts, mostly focused on Linux-based servers, laptops and Virtual Machines. ProtoRINA [31] is a Java-based RINA implementation, mostly designed for education purposes and quick prototyping in academic environments. Last but not least, RINASim [32] is an OMNeT++ based simulation framework for RINA networks.

The main contribution of this thesis is the design, prototype implementation and initial benchmarking of a software-based, performing RINA implementation that provides a virtual network substrate in a distributed VNF context. The thesis is structured as follows: section II introduces the distributed VNF use case; section III describes the design and implementation for the RINA virtual networking solution; section IV provides initial benchmarking results, and section V states conclusions and discusses future work.

Part of the outcomes of this project have been included in the preparation of a scientific article submitted for presentation in the 2022 EuCNC & 6G Summit held in Grenoble, France on June 2022, currently under review.

2 State of the Art of network technologies to communicate distributed VNFs

2.1 Distributed telco cloud challenges and use of RINA to mitigate them

2.1.1 Challenge 1: Instantiation and reconfiguration of the network infrastructure of the distributed Telco Cloud

As we stated previously in Section 1, VNFs within Telco Clouds require secure connectivity environments with the right amount of quality, isolated from interference from services owned by other tenants.

The Telco Cloud distributed network fabric must be dynamically reconfigured to track the creation and destruction of VNFs throughout their lifetime, regardless of their location. Traditional network protocols and services, focused on connecting devices in a mostly static configuration, do not adapt well to the requirements of native application networks in the cloud [4], due to:

1. The lack of space for names independent of location and under application control.
2. The need for dedicated protocols to support multi-homing and mobility.
3. The difficulty of guaranteeing the performance of distributed applications.
4. The complexities related to address management and its scope in deployments of functions in native environments in the cloud, such as virtual machines or containers.

Therefore, cloud applications generally require middleware that mitigates the limitations of current network protocols. Service Meshes [30] are the latest incarnation of this type of middleware, and they support a cloud application development model known as "microservices architecture". Since network protocols and service mesh proxy servers are designed and implemented independently, this approach decreases the efficiency of network functions due to redundant or misaligned functionality between service meshes and network protocols. Also, the performance of service meshes is a problem for VNFs that play a role in the data plane [3].

The work exposed in this thesis is tested in Open-VERSO, a 5G network targeting evolution to 6G and focusing on the delivery of bandwidth-demanding and latency-constrained services, modelled through VNFs. In this context, RINA is applied as a network virtualization framework to support applications in the cloud, integrating traditional functions of network protocols and service meshes functions in a single integrated solution. The project leverages the benefits of RINA to efficiently address application networking requirements, such as

1. Application independent namespaces [5].
2. Dynamic, secure and customizable connectivity environments [6].
3. Support for mobility and multi-homing without the need for dedicated protocols.

4. Flexible support for mapping application QoS requests in internal network policies [7].

2.1.2 Challenge 2: Have the ability to respond to multiple applications with differentiated quality of service requirements, using a common infrastructure that supports dynamic instantiation of network slices

Until relatively recently, the quality of service in networks has been synonymous with concepts such as "speed", "bandwidth reservation" or the ability to guarantee a minimum or maximum data transfer speed (peak). However, this view is changing, and network quality metrics are evolving to better describe what applications need. The industry is gradually recognizing the need for standard metrics that relate the performance of applications to the behavior of the network, so that the network can:

1. The lack of space for names independent of location.
2. Fine-tune their internal mechanisms to provide adequate amounts of quality for these applications (not too much, not too little).
3. Measure the quality that is being delivered in each network service, to assess whether it is achieving its objectives. This metric is called quality attenuation (it is symbolized by ΔQ) and it is gaining traction in the industry [8].

To be able to apply ΔQ metric in practice, the network must offer mechanisms to the applications so that they can express their requirements dynamically, each time they request a network service.

As stated in the introduction (section 1), RINA provides a network service API that allows you to express this information, and also allows you to add and move it as you go down the network layers to the physical medium. Research work prior to this project developed a resource allocation policy for RINA based on the ΔQ framework - called QTAMux. This policy allows a DIF to support various levels of quality of service while making efficient use of DIF resources [7].

2.2 Network virtualization architectures for distributed telco cloud services

To date there have been a number of European research projects (FP7 and H2020) whose objective has been to develop technology based on RINA; working on your specifications, use cases, network designs, and prototype implementations. The FP7 IRATI project developed the first open source RINA prototype for the Linux operating system [9], allowing RINA to be deployed over Ethernet, TCP or UDP with simple policies for DIFs . The FP7 PRISTINE project improved the RINA implementation developed in IRATI by making it programmable through an SDK (Software Development Kit). PRISTINE designed more sophisticated policies for congestion control [10], scheduling [7] and security [11] functions.

Finally, the H2020 ARCFIRE project contributed to issues related to network management [12], designed and implemented policies for quality of service, distributed mobility

management [6] and the dynamic change of network addresses [13]. ARCFIRE improved the IRATI prototype with new policies, support for deploying RINA over Wi-Fi and stability improvements to support test scenarios with up to 100 nodes using the FIRE program testbeds [14].

The use case that this work tries to cover, is the Open-VERSO project [15], which requires a software-based implementation of RINA, but with performance (in terms of packet-per-second processing capacity) higher than current RINA prototypes can provide. IRATI and rlite, RINA's two main prototypes for the Linux operating system, are designed in which part of the components reside in user space and part in the operating system kernel. This design is well suited to the requirements of generic applications, allowing it to integrate well with Linux network subsystems and provide a consistent API to all applications.

However, when the use case is focused on supporting high-performance VNFs, another type of design is needed. This RINA prototype is based on a software framework designed to process packets with high speed and low latency. There are different frameworks focused on providing this functionality. For this RINA implementation it has been decided to use Netmap, since it supports several operating systems (Linux and FreeBSD) and it allows packets to be forwarded from an interface to the network subsystem in the kernel (which DPDK does not allow).

2.3 Netmap: a Software framework for the implementation of high-performance network functions

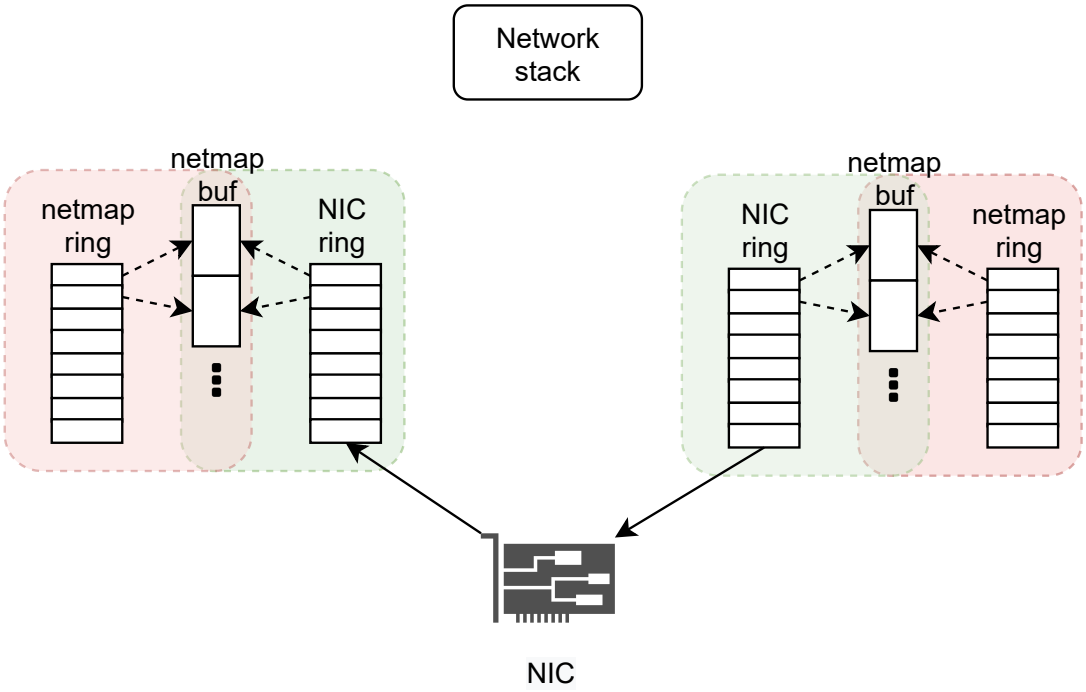


Figure 1: Diagram representing what happens when a netmap interface is set in netmap mode. The green area represents kernel space and the red one user space

Netmap [16] is a hardware-independent tool, which provides an API to have direct access to the NIC functionalities from user space of the operating system. In addition, netmap provides a set of optimizations techniques that provide better performance for packet processing (in terms of number of packets processed per second and latency) than the conventional implementation of network subsystems in the operating system.

When an interface is set in netmap mode, packets are intercepted before they are processed by the operating system's network subsystem (called "network stack" in Figure 1).

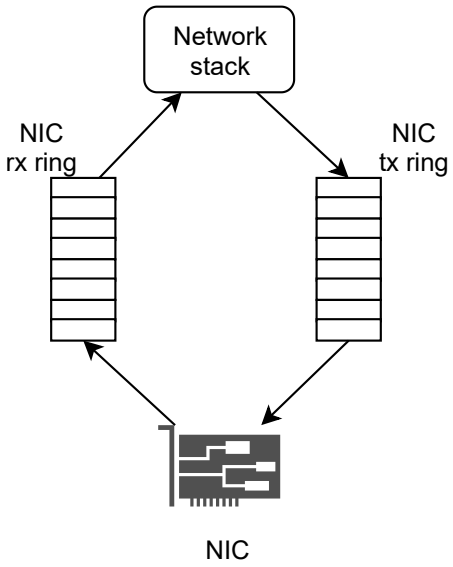


Figure 2: Conventional flow of traffic from the NIC to the OS

The conventional packet transmission and reception scheme is illustrated in Figure 2. On the other hand, in Figure 1, an interface is shown in netmap mode that, as previously mentioned, is disconnected from the network subsystem when the NIC is set in netmap mode.

When an interface is in netmap mode, netmap buffers and netmap rings are allocated in a shared memory region (zones represented in green are only accessible from kernel space, while red ones are only accessible from user space). A netmap ring is just an abstraction of a real NIC ring. The applications that operate in the netmap rings are synchronized with the actual rings belonging to the NIC through calls to the operating system.

Netmap is implemented as a kernel module for FreeBSD (native) and Linux (out of the tree) systems. Netmap offers native support for several NICs through slightly modified drivers. For all other NICs, netmap provides a generic emulator that allows to use netmap on top of the standard drivers.

The netmap API provides a number of optimizations de facto. One of the most relevant is the amortization of system calls through batching. Here is a summary of the numbers that netmap can provide:

- 10G speed with minimal packets using a fraction of a core.
- 30 Mpps on 40G NICs (limited by NIC hardware).

- 20 Mpps on VALE ports (netmap kernel software switch).
- 100 Mpps in netmap pipes.
- Similar performance on both physical machines and VMs.

3 RINA (Recursive InterNetwork Architecture)

RINA (the Recursive InterNetwork Architecture) is an architecture of computer network protocols that unifies distributed computing with telecommunications. The fundamental principle of RINA is to model the communications between computers as a generalization of the communication between processes (Inter Process Communication, IPC). RINA reconstructs the architecture of the Internet forming a model composed of a single type of repeating layer – the DIF (Distributed IPC Facility). The DIF provides the minimum number of components necessary to allow the distributed communication between application instances. RINA supports multi-homing and mobility natively, provides quality of service without need for additional protocols - beyond those already provided by the architecture -, offers a safe and programmable environment, facilitates a more competitive market than the current one and makes possible a gradual deployment of the technology - interoperating with today's existing network protocols.

RINA is the result of scientific work that seeks to describe what are the general principles of computer networks that apply to any type of network and environment. RINA is also the specific architecture, implementation, test platform and deployment of this theory. The theory informally known as the IPC-based network model [26], [27]; although it also defines concepts and results that are applicable to any type of distributed application (not only to computer networks).

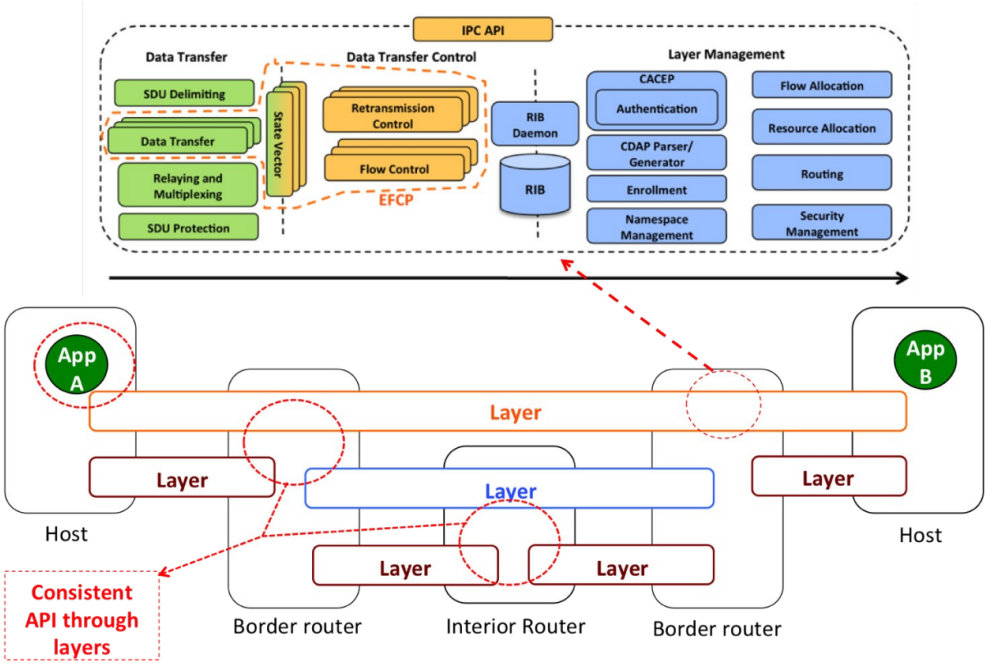


Figure 3: RINA structure: DIFs between systems (below) and IPC components (on top)

RINA is structured from a single layer type – the DIF – which is repeated as many times as the network designer needs (Figure 3, below). in RINA, layers (DIFs) are distributed applications that provide the same service (communication flows between application

instances) and have the same internal structure (Figure 3, top). The instantiation of a layer in a system is a process called IPC Process (IPCP). All IPCPs have the same functions, which can be grouped into the following categories:

1. Data transfer (delimited, addressing, sequencing, multiplexing, time-to-live control, error control, encryption)
2. Data transfer control (flow control, retransmission control)
3. Layer management (routing, allocation of flows, management of the name space, resource allocation, security management)

The functions of an IPCP can be programmed and customized through policies, so that each DIF can be optimally adapted to its environment of operation and the requirements of different applications.

3.1 Definition of the service offered by the DIF

The service definition of a DIF provides an abstract description of the API offered by the DIF to the applications that use your services – the specific APIs depend on each operating system). The application using the DIF can also be an IPC Process belonging to another higher level DIF, thus reflecting the recursive nature of RINA. All DIFs provide the same service, called flow. A flow is the instantiation of a communication service between two or more instances of an application. The API offered by the DIF allows to operate on the flows through the following operations:

- **Allocate**: Allows an instance of an application to request a flow to communicate with other instance(s) of an application. To allocate a flow, the application instance provides the name of the application destination and flow quality requirements if necessary (limits statistics of latency and data losses, minimum capacity, etc.). If the allocation is successful, the DIF returns a local identifier of the flow, called port-id.
- **Write**: Allows you to send a data unit (SDU, Service Data Unit) to through the flow identified by the port-id. The application sends an SDU - made up of N bytes of data - atomically. The DIF maintains the integrity of the SDU, trying to deliver it as it is to the instances of target applications. Applications may agree to receive SDUs incompletely or even partially (if so specified in the flow allocation). The DIF API can block the application or return an error if the internal flow control of the DIF does not allow send data at a certain time.
- **Read**: Allows to receive a data unit (SDU) through a flow identified by the port-id. If no data is available the DIF API can block the application until receiving them (or a timeout expires) or return an error code.
- **Deallocate**: The DIF removes the flow and releases all associated resources to it (such as internal memory, scheduling capacity, etc.)

3.2 The Nature of the RINA layers (the DIFs)

In contrast to traditional network architectures where layers have been defined as units of modularity (different layers carry out different functions), in RINA the layers (DIFs) are modeled as units of resource allocation [11]: all layers perform the same functions, but in different regions of operation (a point-to-point link, a backbone network, an access network, an Internet, a virtual private network, etc.).

The scope of each layer is designed to manage a certain range of bandwidth, quality of service and scale; applying the principle of "*divide and conquer*". Layers manage a certain range of resources. the policies of each layer (DIF) are selected to optimize this management, allowing each function performed by the layer to adapt to its operational environment [29].

How many layers are necessary in a given network? Depends on bandwidth range, quality of service and scale: networks very simple can have enough with two levels of layers; internet very simple with three; larger and more complicated networks may have more. It is a question that is determined at the time of designing a network determined with specific requirements; not at the time of specifying the protocol architecture.

3.3 Internal Functions and Protocols of the DIF

One of RINA's main design principles is to maximize the invariances and minimize discontinuities. In other words, try to the architecture has the minimum possible functions and mechanisms, without the need to generate special cases. Applying the principle of design in the operating systems to separate mechanism and policy - first to the data transfer protocols and then to the data management machinery layer – it turns out that in one layer (DIF) only two abstract protocols are needed (completed by different policies, thus generating various specific protocols with a very large common part) [26]:

- An abstract data transfer protocol that supports different policies and a variety of concrete syntaxes (differentiated by the lengths of the header fields used by the protocol). This protocol is called EFCP – Error and Flow Control Protocol.
- An application protocol that allows performing operations on objects remotes, used by all layer management functions (routing, allocation of flows, resources, etc.). This protocol is called CDAP – Common Distributed Application Protocol.

The separation of mechanism and policy also made it possible to structure the functions within the layer, as illustrated in Figure 3. The components of an IPC Process or IPCP can be divided into three categories: Data transfer, decoupled via a state vector of b) Data transfer control, decoupled through a Resource Information Base (RIB) of c) Layer management. These three groups of functions are characterized by a decreasing duty cycle and an increasing computational complexity (data transfer functions are the simplest ones and those that are executed more often, etc.).

- **Delimitation of SDUs**. The integrity of an SDU transmitted through of the stream is preserved by the DIF through the delimit function. This function also adapts the

SDU to the maximum packet size (PDU, Protocol Data Unit) with which the DIF works. The function of delimited uses the mechanisms of fragmentation, reassembly, concatenation and separation. In the prototype implementation presented in this work, this functionality is implemented by the Protection and Adaptation module (see Section 6.2)

- **Error and Flow Control Protocol (EFCP)**. There is an instance of this protocol for each flow that originates or terminates in the IPC Process. The protocol is divided into two large functional blocks, related through a state vector: Data Transfer (with sequencing mechanisms, packet detection of duplicated or missing, identification of parallel EFCP instances) and Data transfer control (with data transfer control mechanisms stream and relay).
- **Relaying and Multiplexing Task (RMT)**. It takes decisions regarding packet forwarding for all packets entering the IPCP through the flows provided by the lower level DIFs, and multiplexes the multi-stream packets provided by the IPCP that they belong to on the flows provided by the lower level DIFs. There is an instance of the RMT for each IPC Process.
- **SDU protection**. This component performs the functions of verification of the integrity of the PDUs, error detection, compression, encryption and packet lifecycle management. There can be a different protection policy for each flow provided by lower level DIFs. In the prototype implementation presented in this work, this functionality is implemented by the Protection and Adaptation module (see Section 6.2)

The state of an IPCP is modeled as a series of objects that are stored in the RIB (Resource Information Base), access to which is controlled by the RIB Daemon. The RIB imposes a schema on objects that model the state of the IPCP, defining which of the operations available in the CDAP protocol can be executed on each object, and what are its effects. The RIB Daemon provides all the functions of layer management the common mechanism for interacting with the RIBs of other IPCPs on the same DIF (instead of each layer management function using its own independent protocol, as is traditional). This behaviour is illustrated in Figure 18, in Section 6.

3.4 Naming and addressing

Figure 4 illustrates the main entities that receive names in the RINA architecture. Applications are given a name that is independent of their location in the network (this way they can move without losing their identity). Applications can have a name that

1. Identifies all the distributed application (it is called DAF name, Distributed Application Facilities).
2. Identifies a subset of the process instances, members of the distributed application.
3. Identifies specific members of the distributed application (process instances of this application).

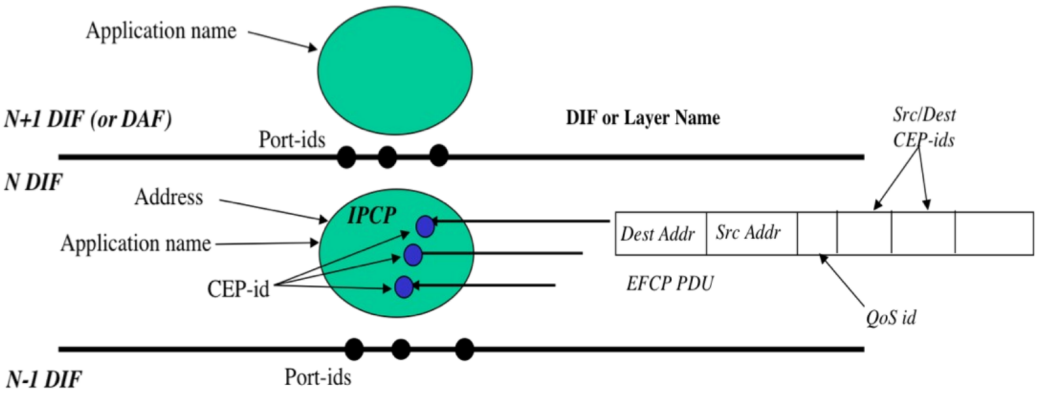


Figure 4: Naming and addressing in RINA

Application names are unique within the application namespace (although there may be multiple namespaces independent of each other). When an application requests a stream through the DIF API, the application provides the target application name as one of the arguments. If the stream allocation is successful, the application receives a port-id, which is the local identifier of the stream.

The IPC Processes are instances of applications, so they have an **application process name** that uniquely identifies them; as defined in the first paragraph of this section. However, the size of the space names of IPCPs can be much larger than the number of IPCPs within a given DIF. Also, the application names are not designed to facilitate the routing of packets within a DIF. Therefore, it is useful to assign a synonym to the IPC Process, called an **address**. The address must be dependent on the location of the IPCP within the DIF – in other words, it should make it easier to search for and find the IPCP in the DIF. However, the direction also has to be independent of the route taken to get there to the IPCP – so that an IPCP can be reached efficiently by different routes [28]. Multiple addresses can be assigned to the same IPCP – since the addresses do not express the identity of the IPCP – that only have meaning inside the DIF (each DIF keeps its space of addresses, which is independent of the address spaces of other DIFs). IPCPs exchange traffic with higher or lower level DIFs through the flows identified by the port-ids – in the same way as do the "*normal*" applications.

Each stream offered by a DIF is implemented internally through a "*connection*" between two instances of the EFCP protocol. Each EFCP connection is identified through a pair of "*Connection Endpoint ids*" (CEP-ids), which identify the source and destination EFCP instance for that connection. The port-ids and the CEP-ids are locally associated in each IPC Process; an association that can change during the lifetime of the flow. The QoS-ids identify the class to which the packets in a flow belong. All packages of the same class receive the same treatment within the DIF, thus allowing a differential treatment between classes (affecting variables such as latency, capacity, etc.).

4 Applicability of RINA for distributed VNFs

The prototype described in this work targets the core needs of the common use case in NFV-enabled platforms, where every compute node supports the deployment of multiple VNFs that are interconnected to provide services to users (see Figure 5). The work is tested in the Open-VERSO federated infrastructure, a 5G network targeting evolution to 6G and focusing on the delivery of bandwidth-demanding and latency-constrained services, modelled through VNFs. In this context, RINA provides Virtual Private Networks (VPNs) to connect VNFs belonging to the same service, while isolating VNFs belonging to different services.

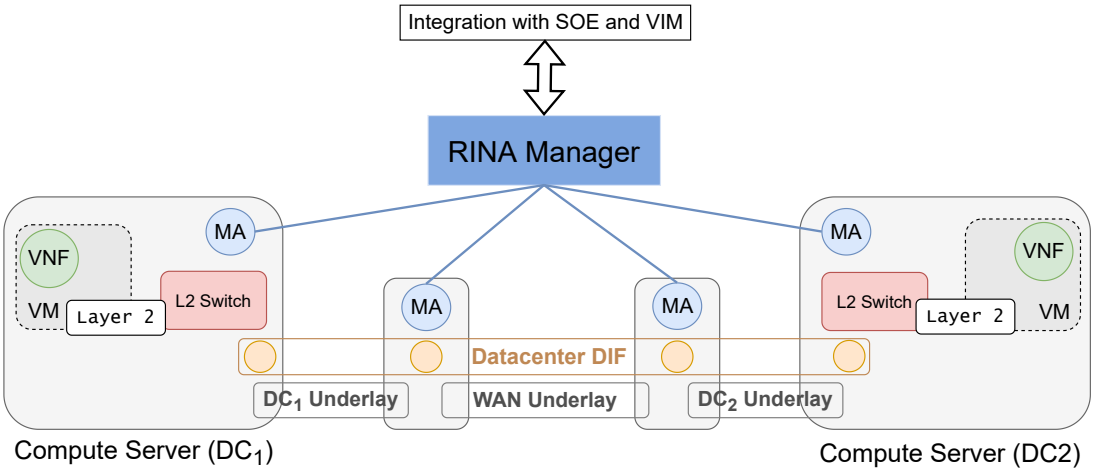


Figure 5: High level design diagram of RINA as a virtual network solution to interconnect VNFs

RINA offers virtual private networks of layer 2 between compute servers in the same or different datacenter. This model states that each VNF is located in a dedicated VM, that way it is ensured that isolation is provided between VNFs. These VNFs encapsulated in VMs are the endpoints of the network. Those endpoints located in the same compute server are interconnected through a Layer 2 software switch. The connection between endpoints distributed in different servers (in the same or in a different datacenter) is handled by a DIF, the underlying facility that provides communication services to the VNFs. The applications that are responsible to provide and manage the communication between the endpoints are the IPCPs (the orange dots in Figure 5 represent the IPCPs). It is important to note that, although only one DIF is enough to handle communication between the applications, it is mandatory to have IPCPs in the gateways of the datacenters in order to add the necessary underlay that the WAN requires.

This *RINA as a virtual network solution* model must integrate with advanced 5G infrastructures. Hence, it shall communicate with the infrastructure layer of the architecture in order to support the orchestration layer with the capabilities of instantiating VNFs and RINA-based networks. This integration is handled by two main elements: the RINA Management Agent (MA) and the RINA Manager (blue elements in Figure 5). A RINA Management Agent runs in every compute server and is directly attached to a centralized

RINA Manager - which shall be able to interact with a Virtual Infrastructure Manager (VIM). In the work presented in this paper, we use OpenStack as a VIM. To that end, the RINA Manager is implemented as a mechanism driver for the Modular Layer 2 (ML2) plugin for Neutron, the network system of OpenStack.

4.1 VNFs running in the same server

Regarding communication between VNFs running in the same server, the hypothesis shown in Figure 6 is considered. There is an instance of a MAC learning software switch (L2 Switch in Figure 6 and Figure 5) for each service that interconnects the corresponding VNFs. In other words, VNFs belonging to the same service are connected between them and isolated from those that do not belong to the same server. The L2 switch is implemented with the netmap API, and its design and implementation is described in section 5.

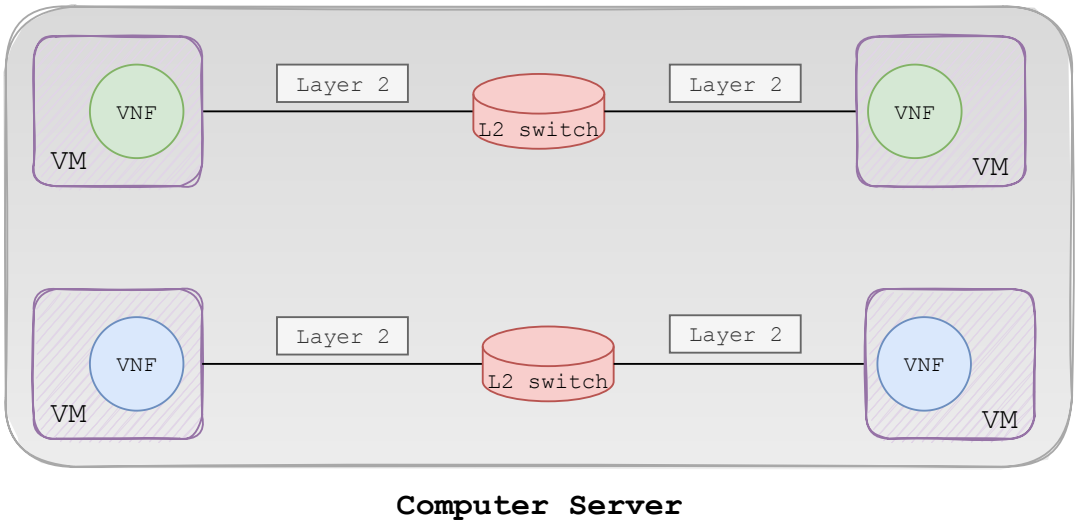


Figure 6: Work hypothesis for VNFs running in the same server

4.2 RINA in the compute nodes (1 Global DIF)

Figure 7 illustrates the two main use cases of RINA as a network solution for an NFV context. On top, Figure 7 shows the case in which all VNFs are located in the same datacenter or - what would be equivalent - the compute servers are connected by the same underlay network (with Ethernet technology, IP, MPLS, etc.). VNFs - running isolated in a VM or a container - connect to an instance of a virtual Layer 2 switch (shown in red in Figure) inside the compute server.

The virtual Ethernet switches of different compute servers associated with VNFs belonging to the same service are connected to each other throughout a DIF, the underlying facility

that provides communication services to the VNFs; in a configuration where RINA offers Layer-2 virtual private networks.

Case B of Figure 7 reflects a scenario where the VNFs belonging to the same service are hosted in different datacenters, therefore they do not have a common underlay. In this case, the DIF also needs instances of IPC Processes on the gateway nodes, so these IPC Processes can route the traffic of the VNFs through the different network segments.

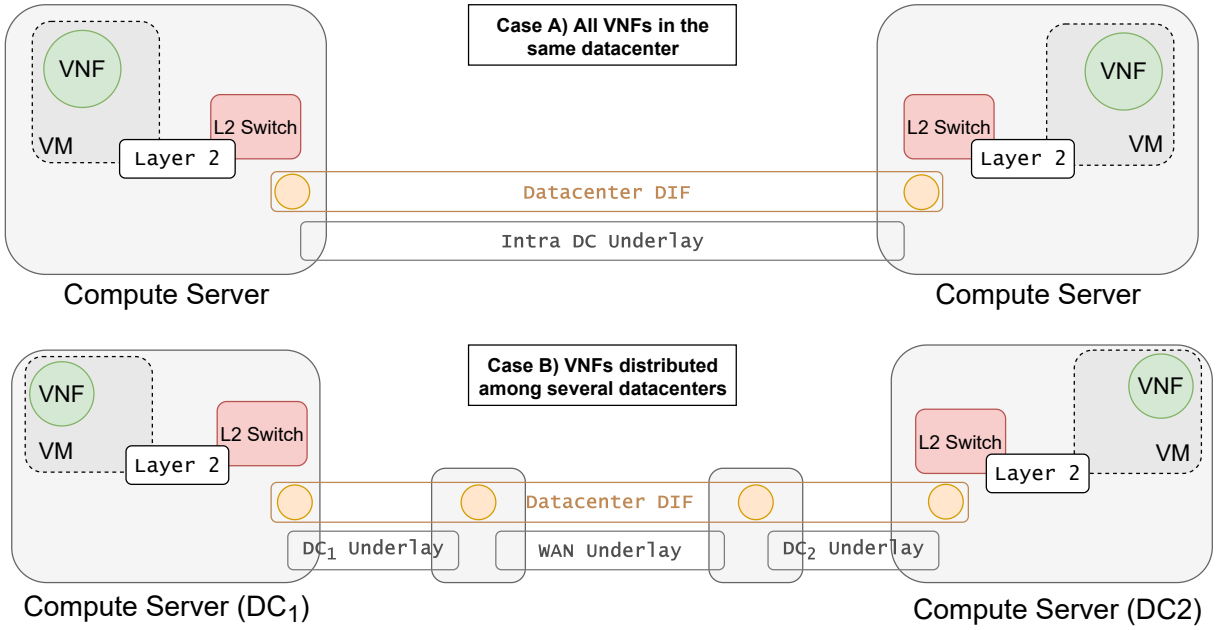


Figure 7: Use Cases of 1 Global DIF topology

4.3 RINA in the VNFs and the compute nodes (2 DIFs)

In the design described previously in Section 4.2, the VMs do not run any RINA stack. In other words, there is only RINA running in the host, and the VNFs are generic applications that use a *legacy* stack (Ethernet).

Another possible design approach is to run RINA not only in the host, but also in the VMs. With this approach it comes a new *Application DIF* that uses the services of the *Datacenter DIF*, i.e. the N - 1 DIF (see Figure 8). This topology allows that the application processes inside the same *Application DIF*, can use the services of the DIF to have a more tailored QoS requirements or different policies depending on the services they offer. Furthermore, VNFs could benefit from a cleaner API, while having better support multi-homing and mobility, without any ad-hoc solutions [35].

Regarding a possible testing scenario, since the applications running in the VMs are generic, a path towards the implementation of this scenario could be to run IRATI [9] inside the VMs (blue IPCPs in Figure 8) and the netmap-based implementation proposed in this work in the hosts (orange IPCPs in Figure 8).

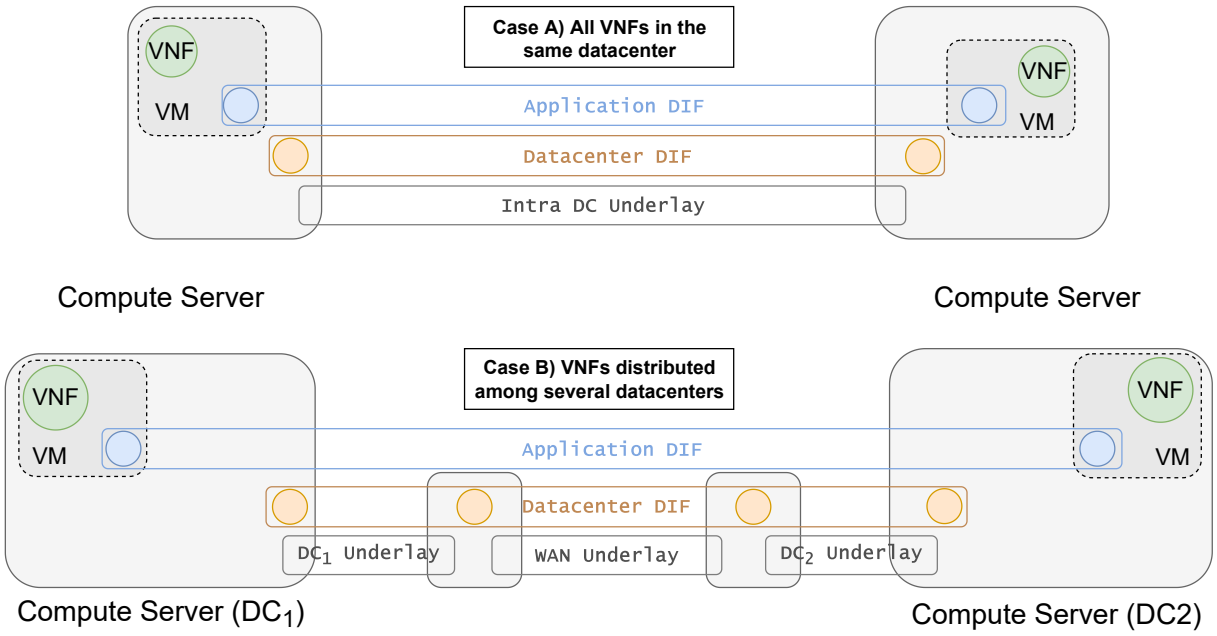


Figure 8: Use Cases of 2 DIF topology

4.4 Comparison of both scenarios

While the 2 DIF solution brings more tailored resources for each service that the application processes bring, it has two main drawbacks. In the first place, the application processes must interact with the RINA stack, which is yet not supported by many applications. In fact, one of the PoC is going to be carried on top of the RINA stack prototype presented on this work, is to run a disaggregated Open5Gs core on top of the RINA network and see how it performs with respect to current network architectures.

5 RINA Prototype Design and Implementation

This section presents the design and implementation of a minimal RINA stack written in C, as well as the needed components to integrate the implementation with an advanced 5G federated architecture. As shown in Figure 5, those main components are the Layer 2 Switch, the Management Agent, the IPCP and the data path implementation.

5.1 Netmap Data Path Implementation

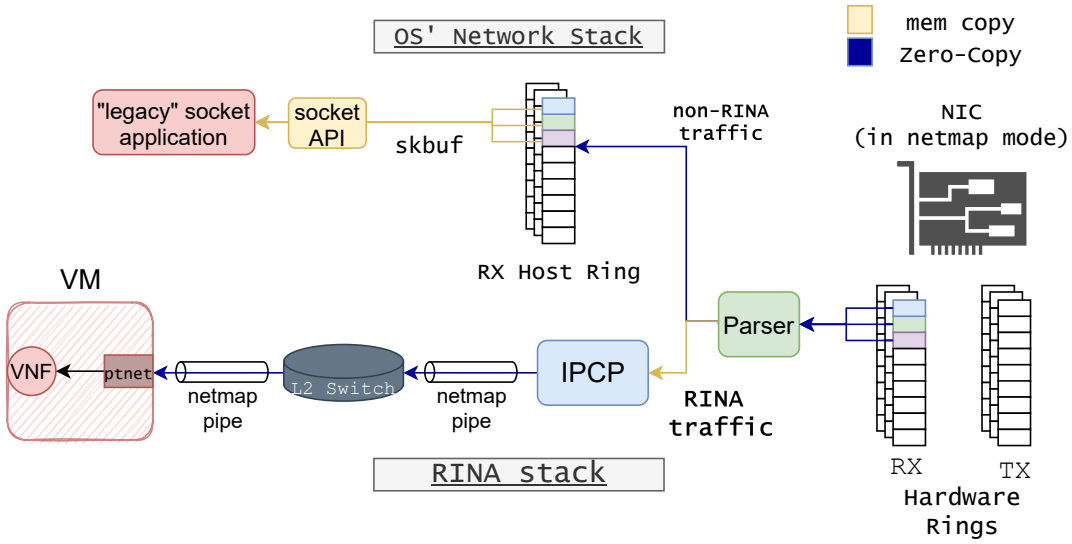


Figure 9: Prototype's netmap-based pipeline datapath workflow diagram

One potential path towards a high-performance RINA stack implementation is to use a fast packet I/O software framework that can run on generic compute hardware, such as netmap [16] or Intel DPDK [20]. The advantages of this approach are twofold. First, it provides relatively high throughput with a great amount of flexibility; since the implementation of the software program is only bound by the constraints of compute hardware instruction sets. Second, the diversity of supported available hardware platforms. As shown in Figure 9, all the data path implementation is netmap based. Figure 16 also shows a workflow illustrating the *RINA stack* path and all the interactions between the different components that compose the data plane, shown in Figure 9. On the other hand, in Figure 10, is shown the Parser workflow for a use case where the underlay is VLAN traffic.

5.1.1 Netmap Pipes and Netmap-Passthrough as NFV Networking Enablers

A network I/O virtualization solution must be in place in order to attach VMs to the network of the host (where VMs contain VNFs), allowing inter-VM communication. There are several state-of-the-art network back-ends: TAP (to inject/receive packets from host TCP/IP stack), socket (packets forwarded through a TCP/UDP socket), netmap/DPDK

(packets injected/received from a high-performance user space networking framework). A traditional deployment could be a virtio-net frontend attached to a TAP back-end connected to either an OpenVswitch instance or a standard in-kernel L2 bridge. However, this kind of deployment has several issues, such as not-batched read/writes to TAP interfaces or virtual switch processing [21].

Netmap brings a network I/O virtualization solution among the aforementioned by following an alternative approach: netmap-passthrough [19]. Netmap provides `ptnet`, which defines a virtual network driver that performs I/O. Briefly, the `ptnet` driver allows the VM to connect to a device that exposes the netmap API such as physical ports (NICs), VALE [23] ports (in-kernel virtual L2 switch) and netmap pipes. In Figure 11, is shown how the VM bypasses the hypervisor and both netmap and legacy applications are able to benefit from netmap optimizations. Netmap pipes are fast, zero-copy point-to-point virtual links that make it easy to set up high speed VNF chains. They behave like Unix pipes, where two processes can communicate between them. In a VNF context, with virtual networks made up with `ptnet` as front-end, plus netmap pipes as back-end to interconnect VMs (see Figure 12), even traditional network stack applications are able to benefit from netmap optimizations. Finally, netmap pipes make it easy to set up high speed NVF chains, and besides the simplicity they bring, they also overpass the performance of the state of the art I/O virtualization solutions such as `VirtIO` [19], [21].

5.1.2 Processing non-RINA Traffic: Communicating with the OS Network Stack

It is desired that the compute servers that are running a RINA stack, are also able to process non-RINA traffic. That way, the RINA implementation allow compute servers still be able to work with legacy non-RINA applications. This is an important feature that the prototype must implement, so we make sure that all applications that work with current internet architectures also can be deployed on top of a RINA network.

When a NIC is set in netmap mode, the NIC is disconnected from the host stack, and made directly accessible to applications that use the netmap API. The operating system, however, still believes that the NIC is present and available, so it will try to send and receive traffic from it. In order to make the NIC rings still accessible by the operating system, we need to make use of host rings, which are two software netmap rings that are directly connected to the host stack, making it accessible using the netmap API. [22].

5.2 A Netmap Based Layer 2 Virtual Switch

Existing VALE virtual L2 switch has a very efficient and scalable switching logic [23], but it forces working in the kernel, with a limited environment (limited libraries, fixed threading and locking scheme, etc.). Since all the prototype is implemented in user-space, a new implementation of a netmap-based L2 switch has been developed.

The L2 Switch (L2-SW) follows the netmap `bridge` logic [24] - which forwards traffic between two network interfaces - generalized to N ports. However, the L2-SW implemen-

tation is by now a minimal implementation with the only goal to validate feasibility of the prototype: it is not yet multithreaded, and lacks code optimizations that make L2-SW still a bit behind of the performance brought by the `bridge` tool when the number of ports is increased to more than two.

5.2.1 The MAC Learning Algorithm

The L2-SW uses a very simple MAC learning algorithm. When an Ethernet packet is received, the L2-SW extracts the source and destination mac addresses. To do so, it gets the MAC addresses and translate them to an unsigned 64 bytes integer:

```
uint64_t
mac2int(uint8_t hwaddr[])
{
    int8_t i;
    uint64_t ret = 0;
    uint8_t *p   = hwaddr;

    for (i = 5; i >= 0; i--) {
        ret |= (uint64_t)*p++ << (CHAR_BIT * i);
    }
    return ret;
}
```

Once the L2-SW has the translation of the source MAC, it uses is to do a `lookup(src_addr)` function to the MAC learning table, where it is stored the `{hash(addr), L2-SW interface}` mapping. Note that, since it is a netmap-based switch, the interfaces are stored as a `nmport_d` structure, which contains all the needed information to use the netmap API. If this lookup to the MAC table results to be `NULL`, then the `{hash(src_addr)` and the `L2-SW interface` from where the packet came is stored in the MAC table. If the lookup is successful, then a `lookup(dst_addr)` is performed. If this operation returns `NULL`, then the L2-SW needs to do a broadcast, otherwise it already knows where to forward the incoming packet.

All this proceeding is illustrated in a workflow on Figure 13.

5.2.2 Batching Techniques in the L2-SW Implementation

Batching is one of the key aspects to take in account when implementing a netmap application. We need to make sure that we are amortizing system calls, i.e., for every `poll()` call, the L2-SW must have processed as many packets as possible due to the fact that system calls are computationally very expensive.

The batching algorithm implemented in L2-SW is illustrated in Figure 14, and is described as it follows:

If no packets are coming (i.e., the RX-rings are empty), the L2-SW remains blocked by `poll()`. When a batch of packets is copied to any RX ring, the L2-SW reads the first one and stores its memory address (i.e., a pointer to the packet) and the destination MAC address - for convenience, the MAC addresses are also converted to integers as explained in Section 5.2.1 -. Then, the Netmap `cur` index is updated and the `pkts_rcvd` counter - which represents the packets in the received batch left to be processed - is decremented by one. The following consecutive packets of the batch that has the same destination MAC address as the first packet, are also `cur ++` and `pkts_rcvd --`. Once there are no more packets in the RX-ring or the destination MAC address change (i.e., the incoming packet has a different destination than the previous one), the L2-SW marks the destination netmap port associated to the destination MAC with the `POLLOUT` flag and calls `poll()`. This will force netmap to wait for egress space in the destination port before copying the packets. Once there is space in the TX-ring of the destination port, the L2-SW copies `(cur - head)` packets from the source port RX-ring to the destination port TX-ring. Finally, the L2-SW updates the `head` index to `cur` in the RX-ring of the source port in order to indicate that ingress space is again available.

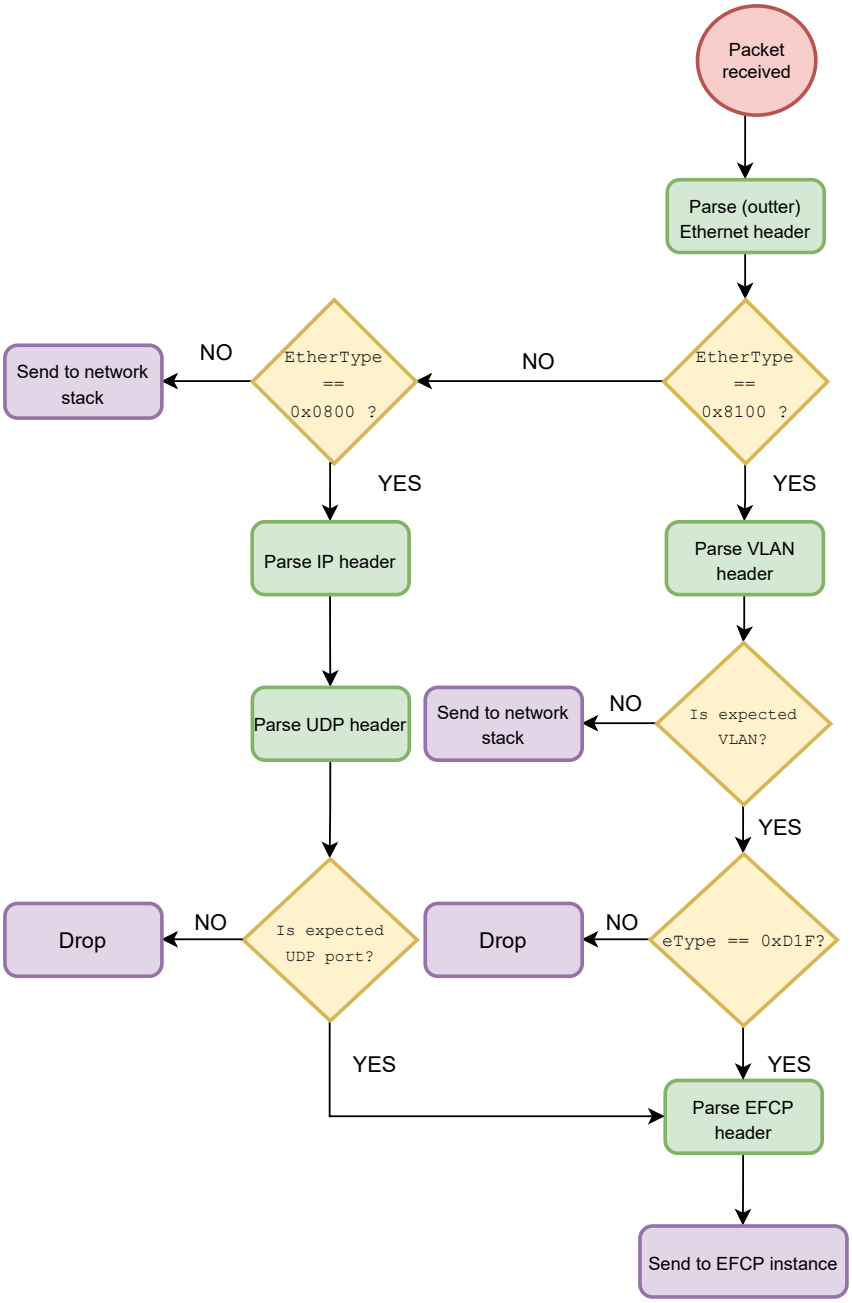


Figure 10: Parser that distributes traffic to be further processed

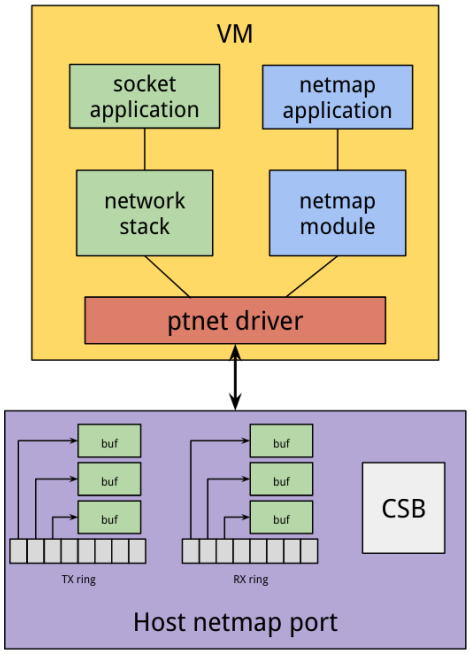


Figure 11: Both traditional socket applications and applications running in netmap mode benefit from netmap-passthrough [17]

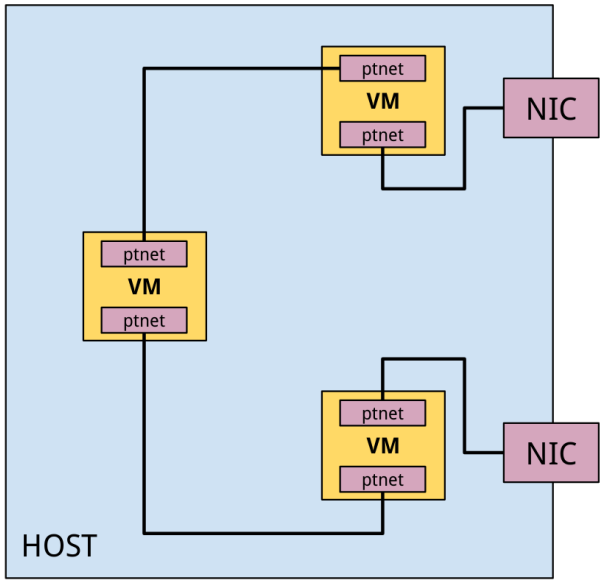


Figure 12: Generic NFV scenario. VMs running ptnet drivers, wired through netmap pipes. Source: [17]

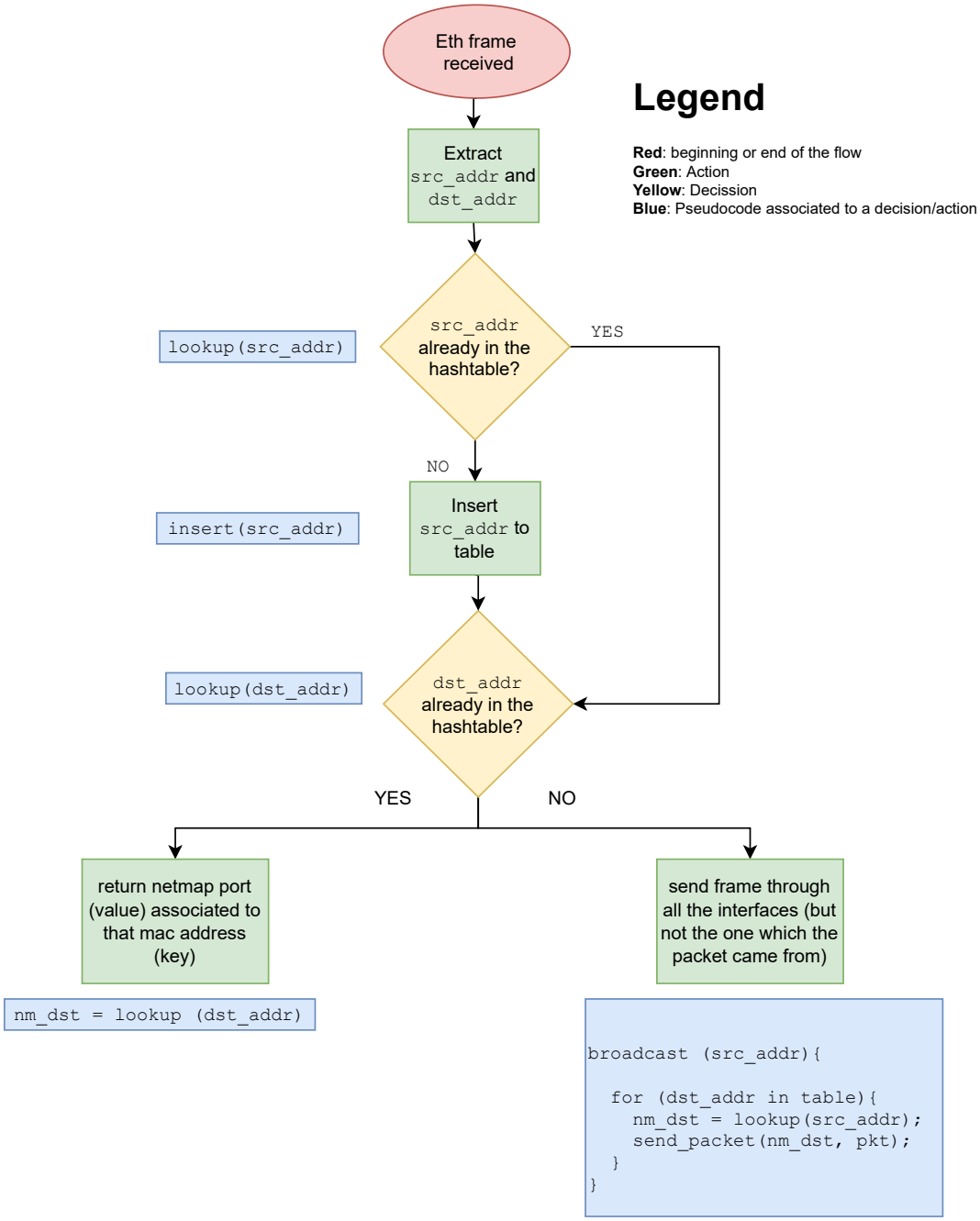


Figure 13: MAC learning algorithm implemented in the Layer 2 Switch

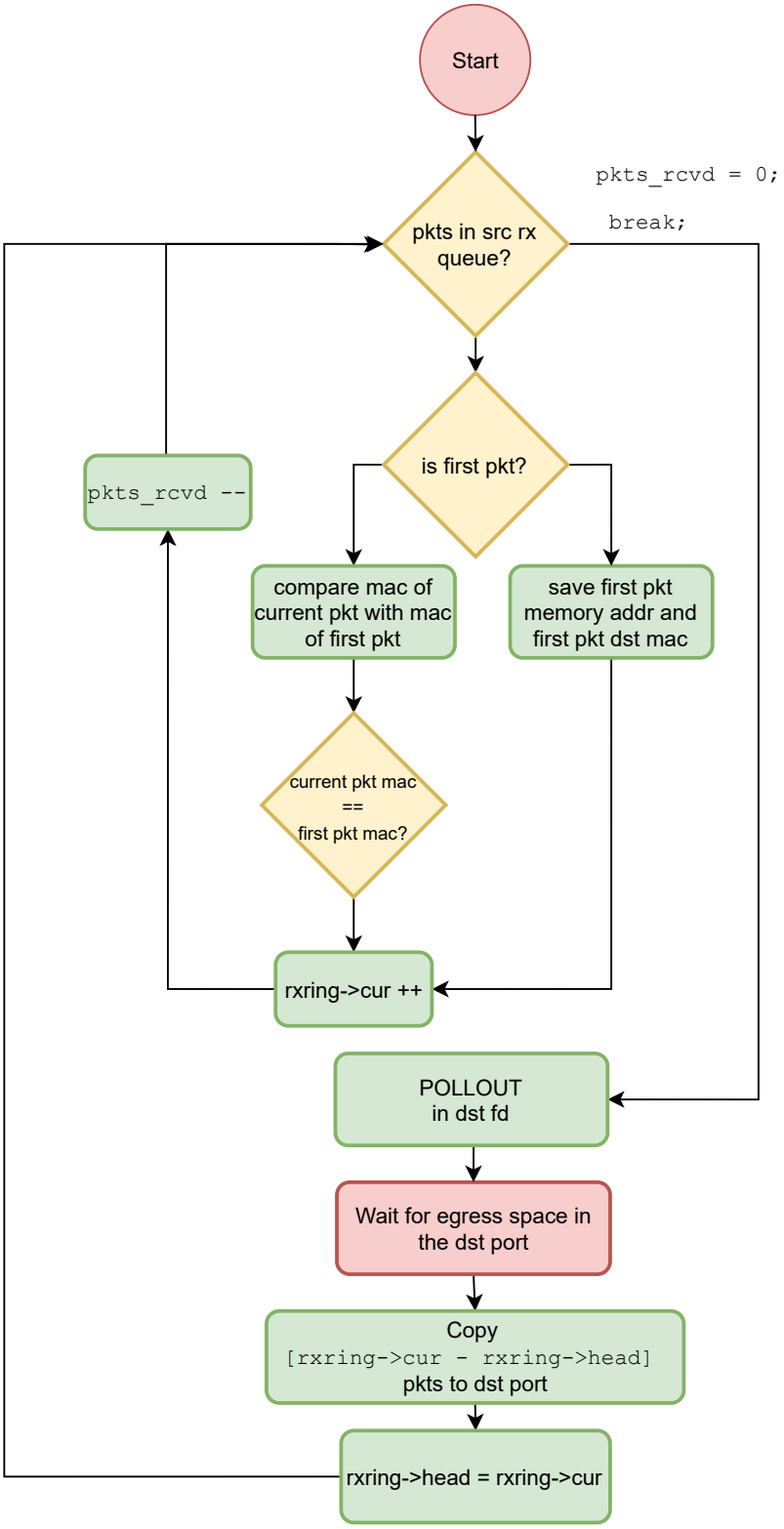


Figure 14: Batching algorithm implemented in L2-SW

6 The IPCP Implementation

This section explains in detail how the IPCP is implemented. As mentioned in previous sections, this project implements a prototype of an IPCP, implementing only the necessary functionalities that lead to evaluate the experiments shown in Section 7. to be able to run the IPCP. For a generic implementation, please refer to the IRATI project [9].

6.1 IPCP Architecture Overview

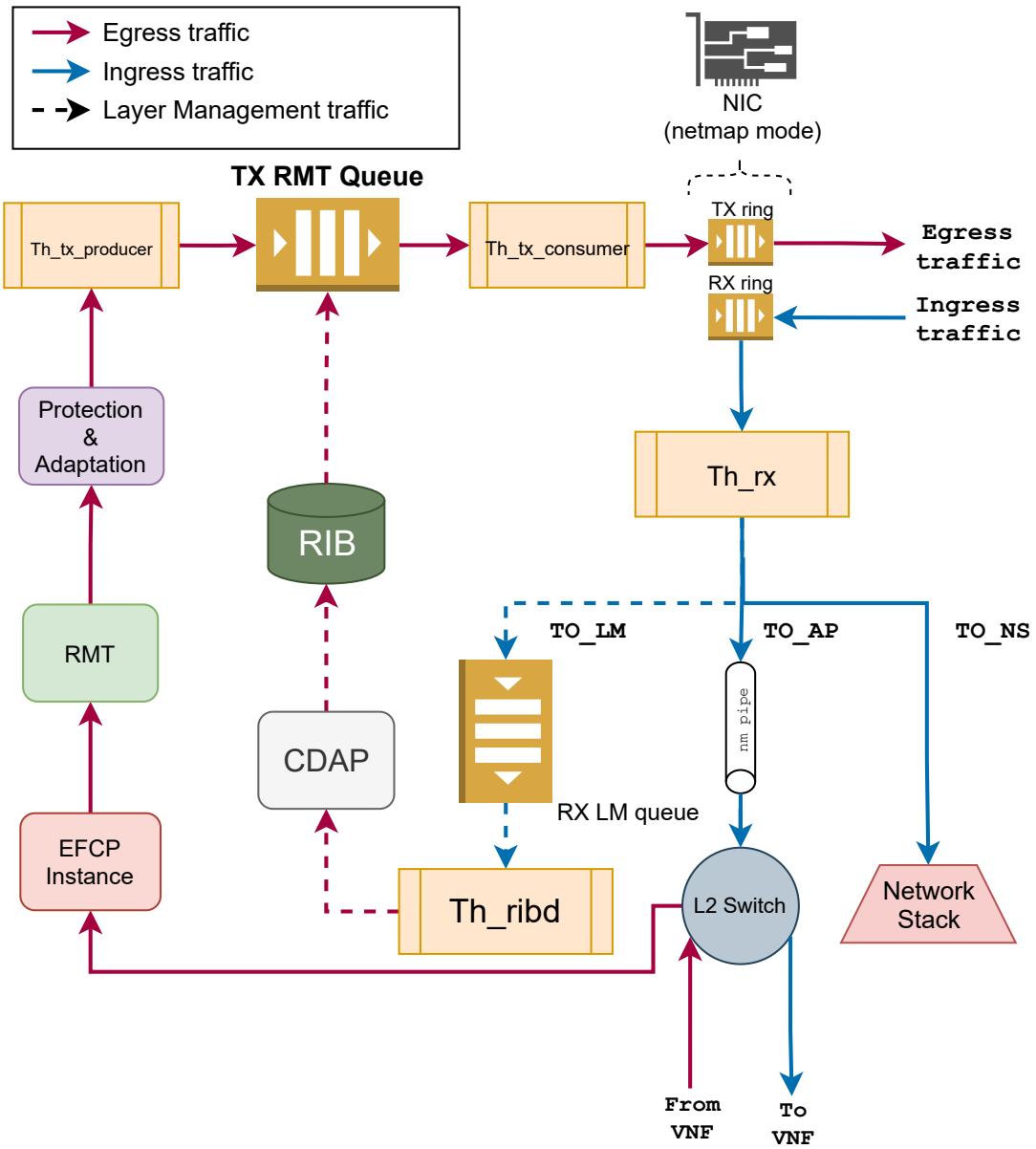


Figure 15: Generic overview of the IPCP architecture of the implementation

In Figure 15 is shown the architecture of the different internal modules of the IPCP

(rounded corner squares), the threads that move along the modules (orange boxes with stripes in the sides) and the queues (all the queues are thread-safe FIFO queues).

The thread `th_rx` parses all the ingress traffic and sends it either to layer management processing, the network stack or the application process (which is always a VNF in this implementation). On the other hand, the egress traffic starts being processed by the `th_tx_producer` thread. It reads the netmap RX ring of the L2-Switch, passes through the required data transfer modules and finally enqueues the egress traffic of the VNF to the RMT, hence, exists one `th_tx_producer` per every EFCP instance and VNF pair. All these producers compete to queue traffic in the RMT queue, which is finally dequeued and copied to the NIC TX ring by the `th_tx_consumer`.

Finally, the `th_ribd` thread implements the functionalities of the RIB daemon: it dequeues layer management traffic, decodes it using the CDAP module, queries the RIB in order to apply operations on its RIB objects and at the end queues the traffic in the RMT to be further copied to the NIC by the `th_tx_consumer`.

6.2 Data Transfer Implementation

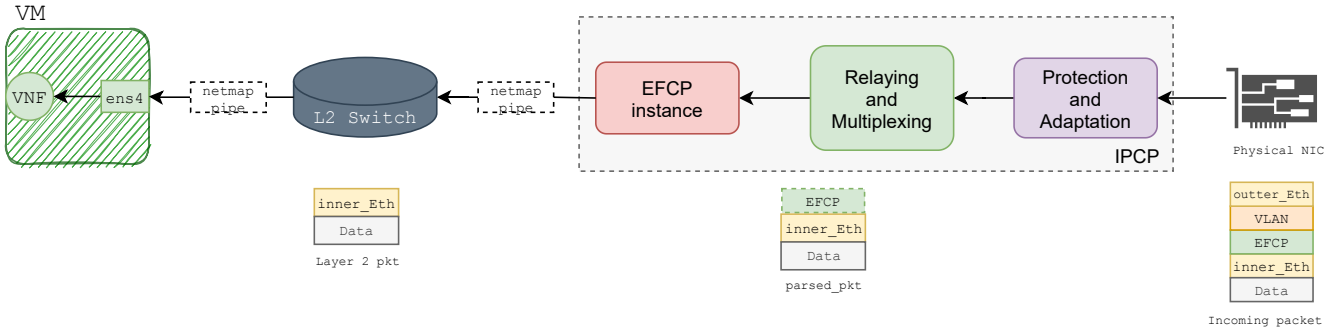


Figure 16: Overview of the RX pipeline of the prototype

As shown in Figure 15, there are three modules involved in data transfer functionality: Error and Flow Control Protocol (EFCP) module, Relay and Multiplexing Task (RMT) and the Protection and Adaptation module (PA).

- The EFCP module stores the EFCP objects and is responsible for performing operations on them. It also stores the mappings between EFCP objects, VM virtual interfaces file descriptors and flow allocator instances.
- The RMT stores the egress queues, each one mapped to the N - 1 port-id associated to the N - 1 DIF. On the other hand, it maps the EFCP destination addresses to its corresponding N - 1 port.
- The PA module stores all the information regarding the network underlay, so it is responsible for adding/parsing the overhead of the egress/ingress traffic. In other words, it implements all the operations related to SDU protection/delimitation described in Section 3.3

Figure 16 shows the modules just described previously and how they attach to the whole RX pipeline.

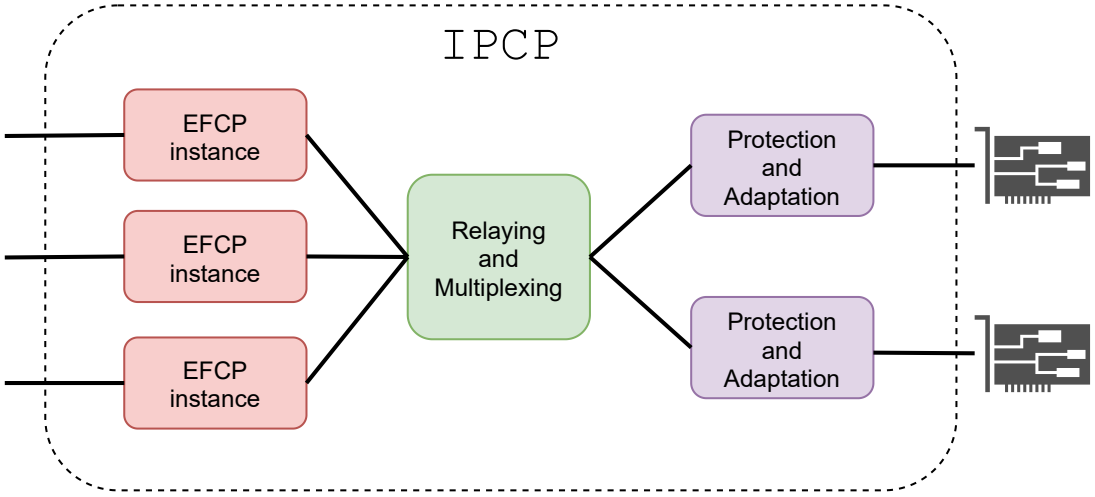


Figure 17: Generic overview of the modules of the IPCP data path: EFCP, RMT and PA

6.3 Layer Management Implementation

The IPCP provides a minimal yet scalable layer management implementation. It only implements the enrollment task, the flow allocator module - since those are the bare minimum modules needed to establish a connection between IPCPs and application processes -, the namespace manager and the routing module. However, namespace manager and the routing module have not been implemented yet. This is due to the fact that all the tests have been carried on with two IPCPs maximum, so the neighbor object in the neighbor database in the IPCP is always the remote IPCP, thus there is no need to implement the namespace manager. Same with the routing, since both IPCPs are directly attached with an Ethernet link, there is no routing further than one hop yet.

6.3.1 The RIB Daemon

The RIB Daemon is the main component in layer management functionalities. It stores the `th_ribd` thread, which, as shown in Figure 15, is in charge of processing all the layer management PDUs by passing through the required modules.

It is responsible to encode and decode all the layer management traffic, which is encoded using a Common Distributed Application Protocol (CDAP). CDAP is the common application protocol that operates on remote objects. In this particular implementation, the CDAP encodings are performed via protocol buffers, due to its efficiency and portability to other languages.

When a CDAP request is received, the RIB Daemon will fetch an object from the RIB based on its object name. The RIB is a database where RIB objects are stored. These

objects point to operations that perform actions in the IPCP. In other words, an IPCP execute operations on a remote IPCP by sending CDAP PDUs. On the other hand, when a request is sent, the RIB Daemon creates and stores a function callback. That way, the callback will be called by the RIB Daemon once the response corresponding to the previously sent request is received. This approach is illustrated in Figure 18. Figure 19 illustrates all the description above

The RIB Daemon also implements the Common Application Connection Establishment Phase (CACEP). During the CACEP, two IPCPs exchange naming information and the RIB Daemon stores and updates all this information in an application connection object.

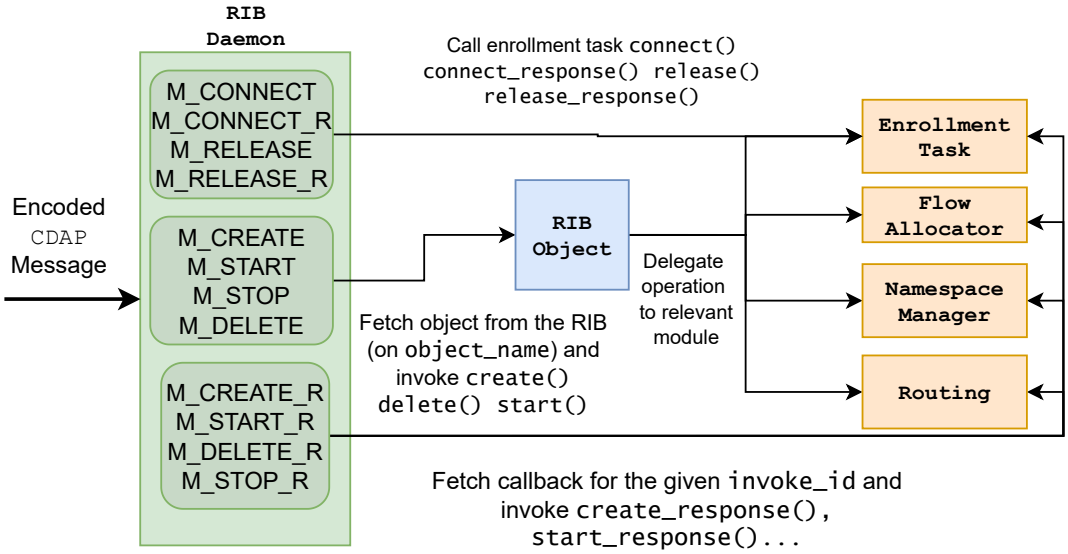


Figure 18: RIB Daemon processing of incoming CDAP messages

6.3.2 The Enrollment Task

In order to establish a connection between two VNFs (which more generically is an application process) it is needed to first establish a shim DIF over the underlay network, then use the services of the shim DIF (the N - 1 DIF) to proceed with the CACEP.

Once the CACEP is established and both IPCPs have stored all the needed naming information, the *enrollee* IPCP starts the enrollment to the DIF by sending an `M_START` CDAP message. Within the `M_START` and `M_START_R` message exchange, both IPCPs exchange their addresses. Right after the `M_START_R` message, the *enroller* also sends an `M_CREATE` message with the *enroller* info, so the *enrollee* can store *enroller's* information as a neighbor object in the neighbors' database. With the subsequent `M_CREATE_R`, the *enrollee* sends to the enroller all the static (part of the definition of the DIF) and near-static (information that may change but very infrequently, i.e. the address) information so both IPCPs know all the necessary information from their peers. The full enrollment process, as well as all the internal function calls of the IPCP, are shown as a sequence diagram in Figure 20. In

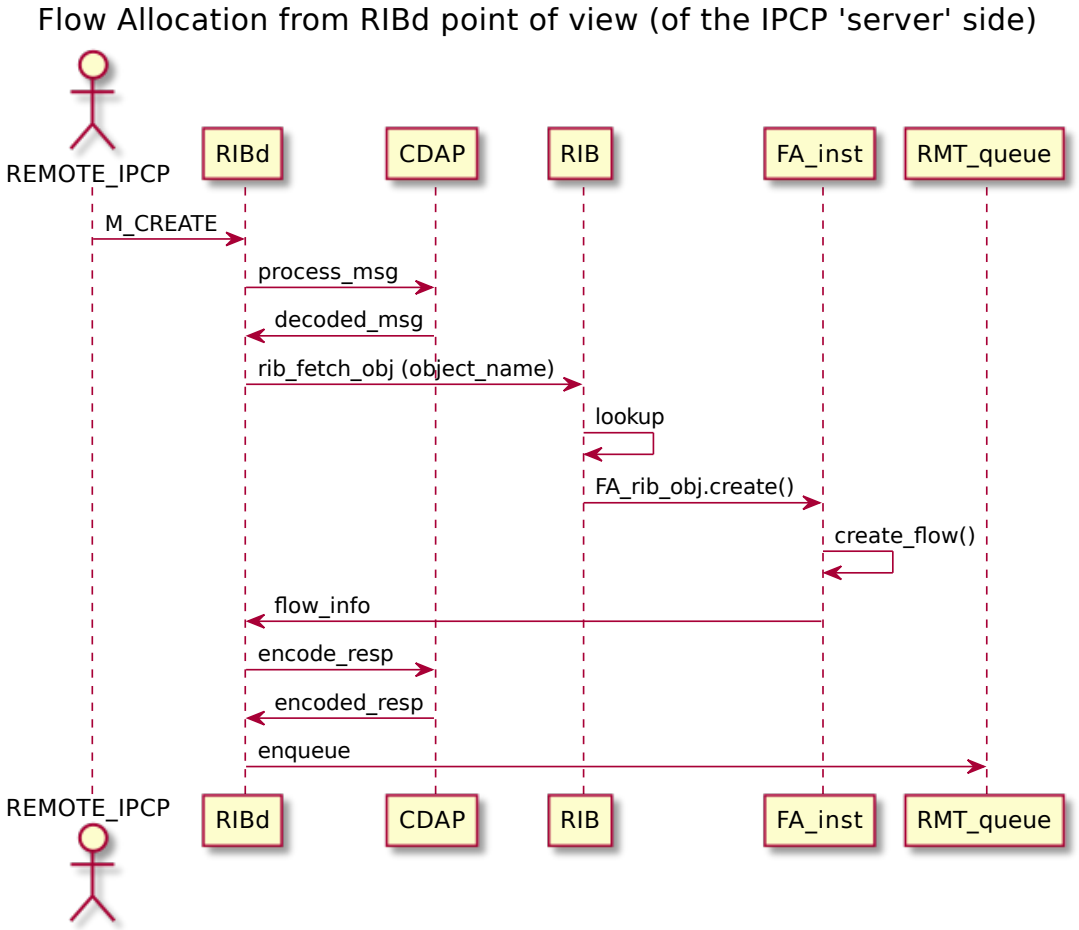


Figure 19: Workflow illustrating the RIB daemon interactions when an M CREATE CDAP message related to a flow allocation request is received

6.3.3 The Flow Allocator and the Management Agent

Once the two IPCPs are enrolled in the DIF, the application process (IPCP) on the N - DIF can request a flow to enable communication between applications on the same DIF. As stated in Section 4, in the use case covered by this work, the RINA implementation is attached with a Software Orchestration Engine (SOE) and a VIM that govern the application processes life cycles. Hence, the application process that request flow allocations to the IPCP is the RINA Management Agent.

The flow allocation workflow starts with the MA receiving a slice creation request from the infrastructure layer. Each network slice may contain several distributed VNFs, so a network slice request may ask for several flow allocation requests to the IPCP. A network slice request contains: the unique slice ID, the remote MAs that this MA should allocate a flow with, and the local port where the VNF is located. In this particular implementation, the local port is always a netmap pipe.

Once the MA request a flow allocation to the IPCP, it creates a new flow allocator instance

and sends an `M_CREATE` CDAP message to trigger a remote flow allocation in the remote IPCP. The remote IPCP will ask his MA if this flow allocation request is legit and if yes, the IPCP will establish a flow and send back an `M_CREATE_R` to confirm that the flow allocation was successful. As an example, Figure 21 illustrates all the process described above as a sequence diagram.

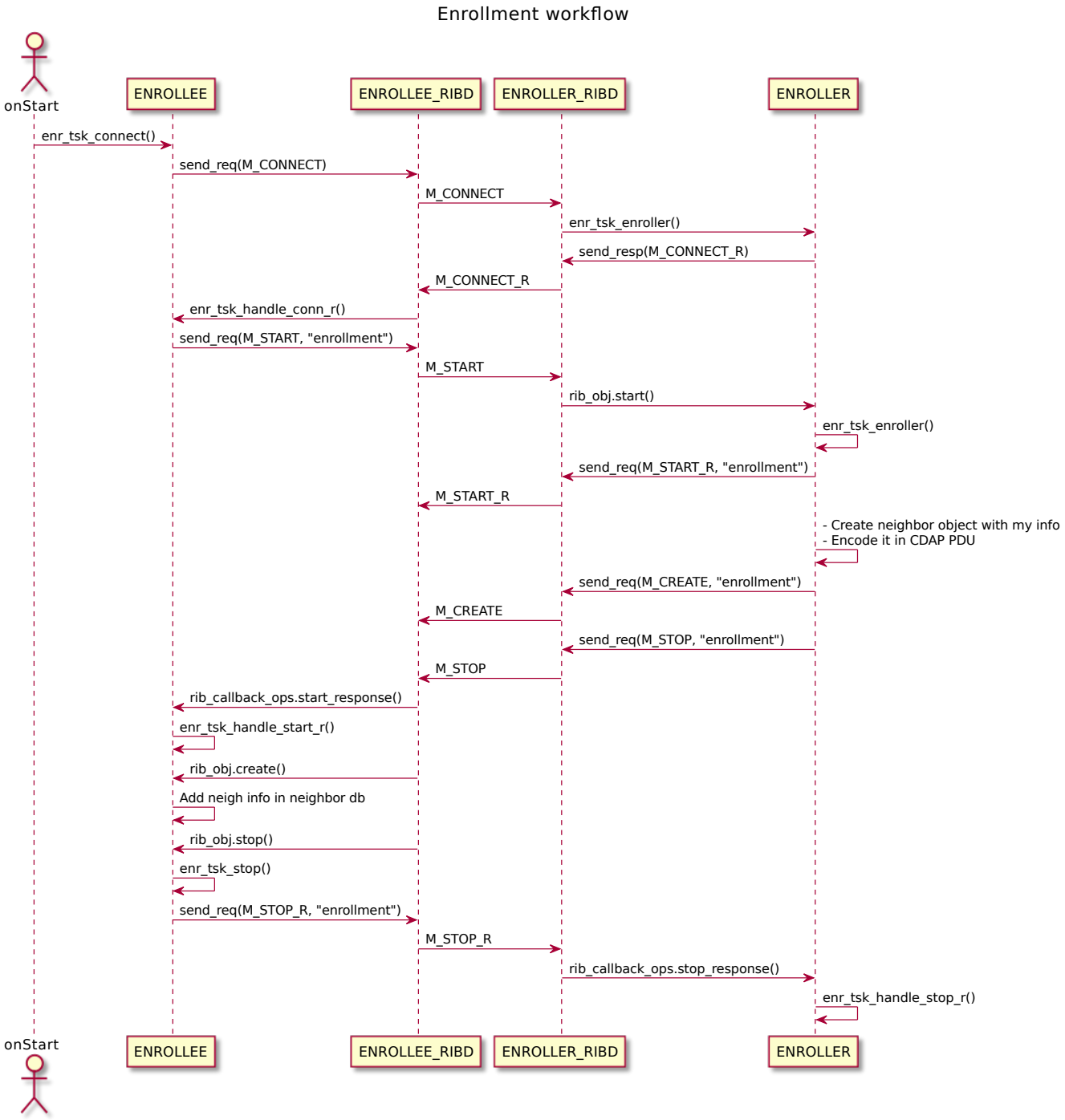


Figure 20: Full enrollment process followed by the IPCP

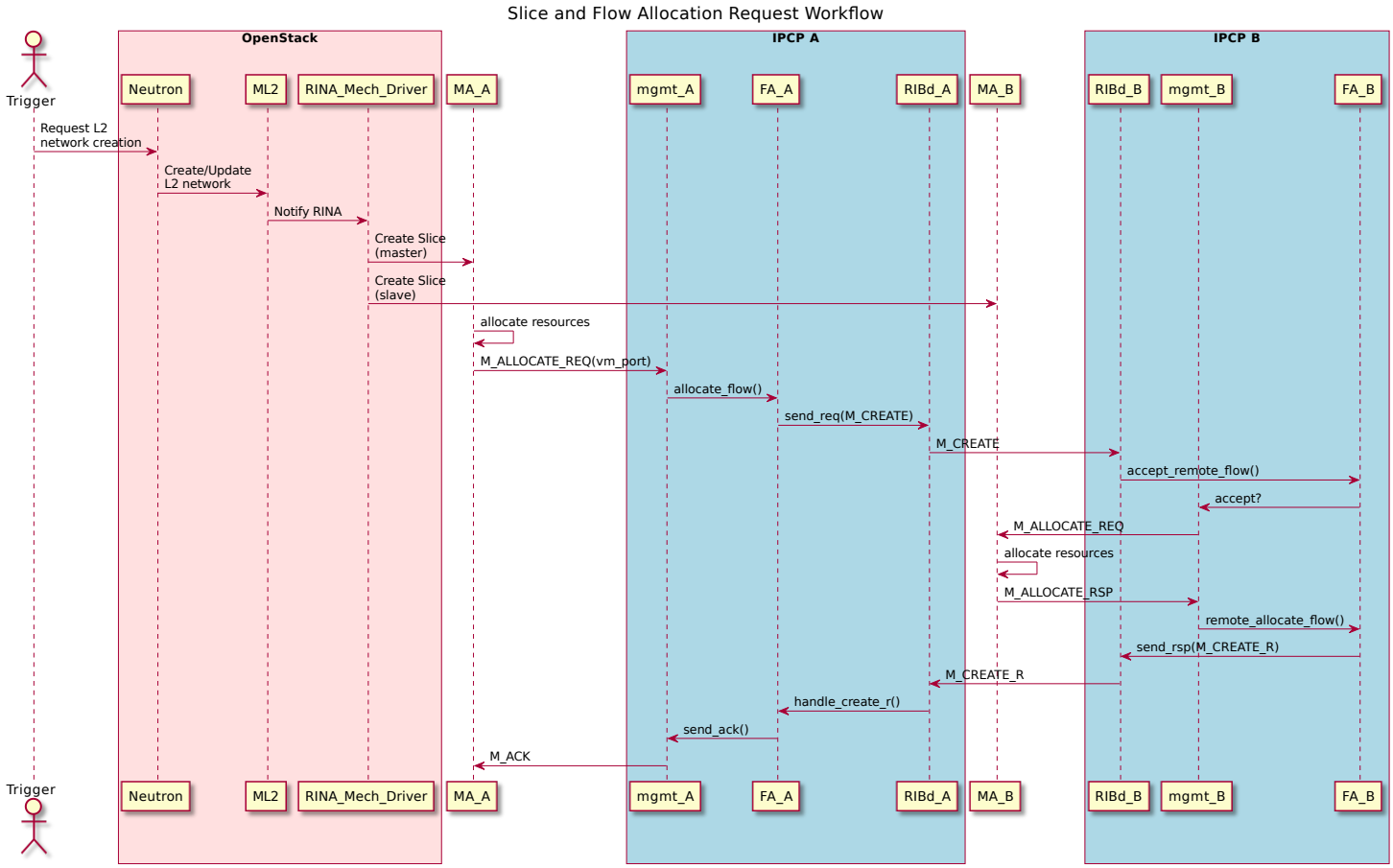


Figure 21: Detailed diagram of the virtualized testbed setup

7 Experimentation, validation and benchmarking

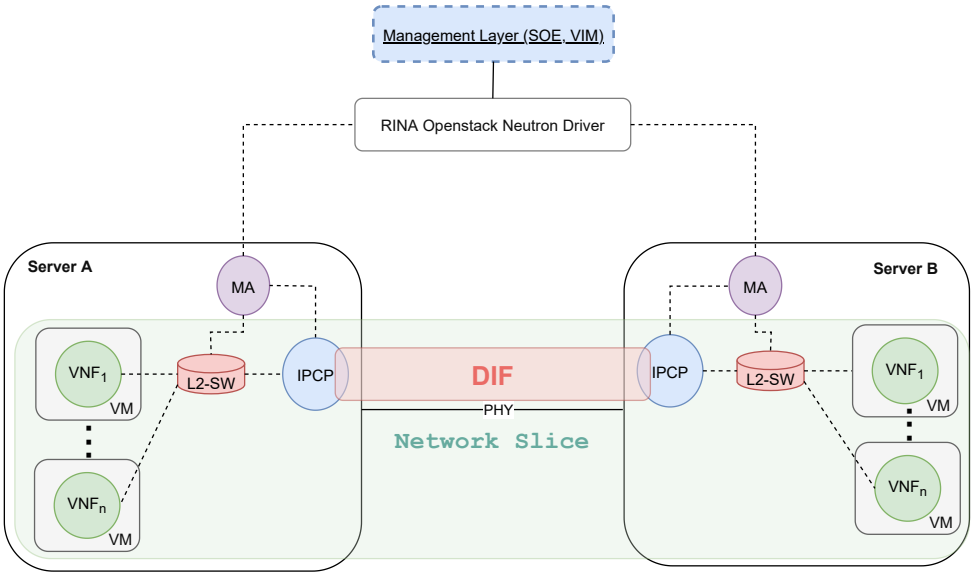


Figure 22: Physical testbed

The testbed for the experiments is shown in Figure 22. It consists of two physical servers located in the same datacenter, connected via an 1Gb Ethernet cable. The experiment aims to evaluate the performance between this prototype and IRATI [9], which is right now one of the most mature and generic RINA implementation as stated previously in Section 2.

Additionally, in Figure 23 the testbed is shown from a *RINA perspective* in the sense that the Figure shows how the VNF inside a VM is an application process and how hey form a Distributed Application Process (DAF). A DAF is no more than the whole distributed application, and it is formed by exchanging CDAP messages and using the IPC services from the DIF they belong to.

However, the testbed has a main caveat: the maximum line rate supported on a physical link, 1.488 for 1 Gb Ethernet. Also, other minor constraints are expected to appear in this physical testbed, such as physical medium or hardware inefficiencies. Because of this, the performance of the prototype has been tested in a fully virtualized environment, shown in Figure 24. In this virtualized experiment, a netmap pipe attached to both IPCPs acts as a physical NIC and physical medium at the same time. That way, it is possible to test the performance of the implementation without being capped by physical limitations.

Having a virtualized testbed in a single machine facilitates the quick testing of functionalities under development as well as the ability to act as a *benchmark* in the sense of a way of obtaining maximum values thanks to the lack of physical hardware/medium limitations. It is important to note that, in order to increase the simplicity of the virtualized test, we are not using virtual machines, since the performance is the same with or without using VMs due to the use of netmap-passthrough [19].

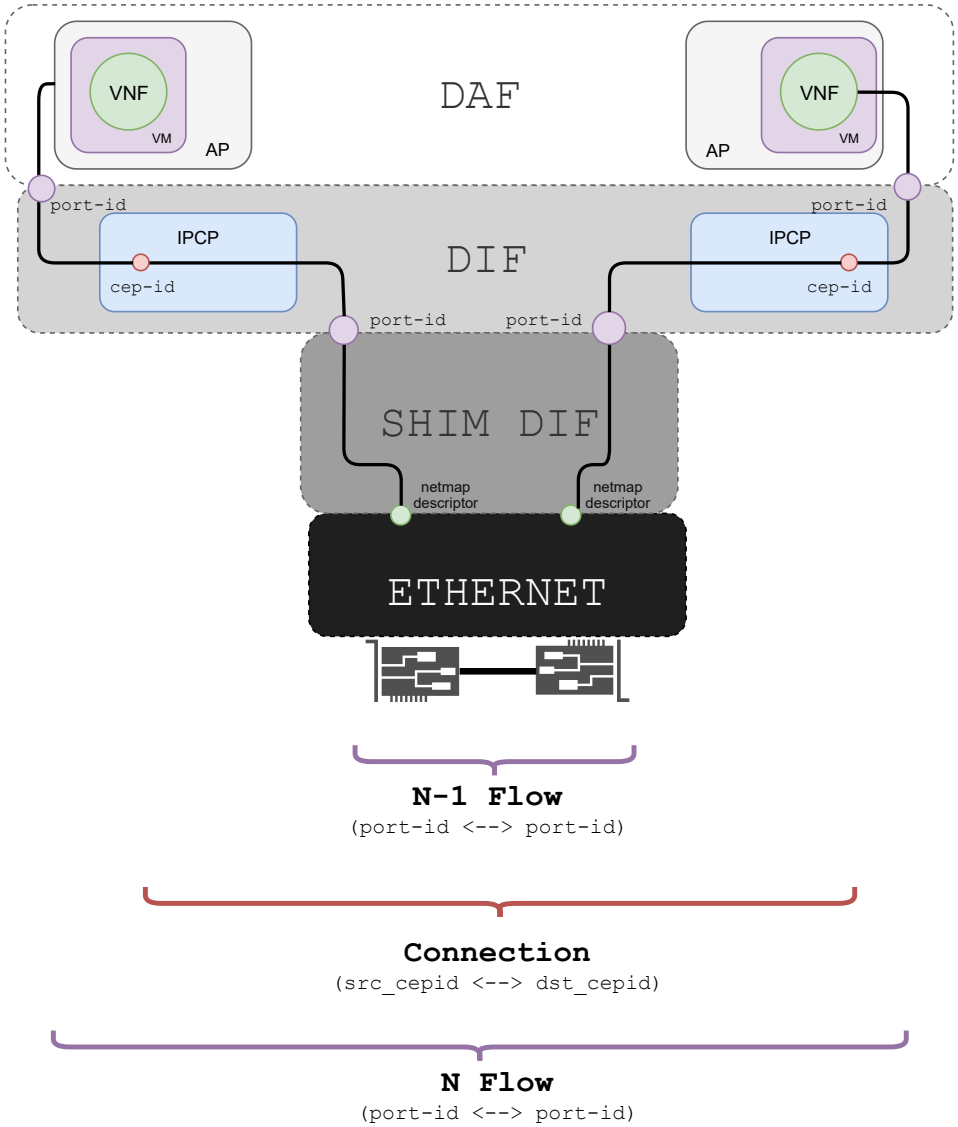


Figure 23: Experiment setup from a RINA perspective

All metrics related to speed and latency are obtained through the `pkt-gen` [25] tool, which leverages netmap to generate and receive packets in large batches, taking advantage of all netmap optimizations. Another consideration is that only one VNF runs at each server, which results into the lack of threads of execution competing for the resources of the queues, and thereby this experiment can be considered as a benchmark.

7.1 Experiment Evaluation

Figure 25 shows the speed in Mega packets per second (Mpps) and the throughput in Gbps, both as a function of the packet size. The green-dotted line represents the theoretical maximum speed in a physical 1 Gbit Ethernet wire. As stated in previous sections, due to the use of packet I/O software framework for packet processing, performance in terms of speed (Mpps) and throughput (Gbps) is the order of 100 times higher than IRATI.

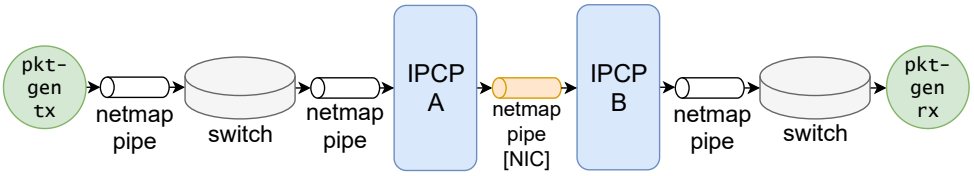
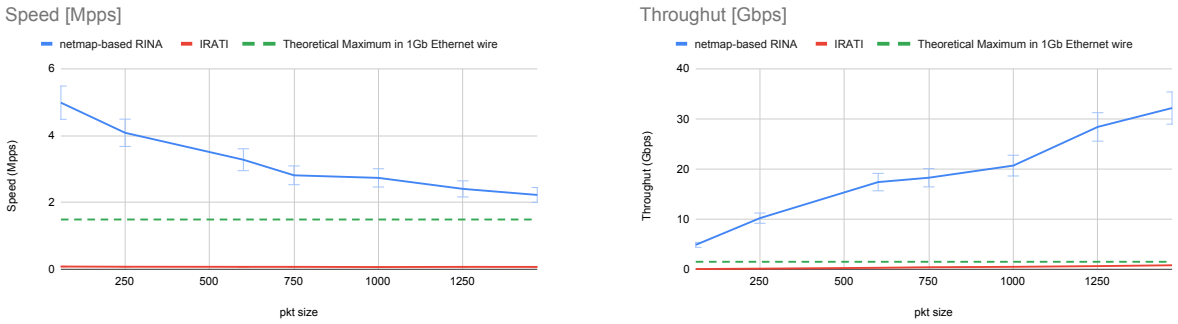


Figure 24: Virtual testbed using a netmap pipe emulaing a physical NIC and the medium



(a) Speed vs packet size in Mpps

(b) Throughput vs packet size in Gbps

Figure 25: Speed and Throughput comparison between the PoC and IRATI

However, it can be noted that the throughput is slightly higher than it should be: this is due to the addition of the WAN header (IP + UDP) and the RINA header (EFCP).

It is important noting that, among the components that are part of the pipeline shown in Figure 24, the IPCP is the bottleneck, as it can process 60 bytes packets at 4 Mpps; whereas a single L2-SW can process 60 bytes packets at 12 Mpps (using zero-copy). As the IPCP implementation can be further optimized, it is expected that performance of the pipeline could increase.

8 Conclusions and future development:

The work carried out in this thesis has proven the feasibility of implementing a RINA stack based on netmap as packet processing framework.

However, some improvements on the prototype implementations are to be done. This includes moving towards a multi-thread approach in the L2-SW, i.e., having a consumer/producer thread pair per interface that reads/forward traffic without giving any priority to any interface (or giving priorities on purpose).

On the other hand, another improvement consists on having more than one thread that performs read/write operations to the physical NICs. Whereas this is not needed in virtualized environments (due to the fact that netmap pipes only have a single TX/RX ring pair), it may affect the performance in physical NICs that have several TX/RX ring pairs.

Due to the fact that the data plane implementation is fully implemented in user-space, an implementation from scratch of the IPCP has been carried out. The IPCP is a key component in the RINA architecture, as it is responsible to expose the API to operate in remote objects in the DIF, so a more generic IPCP implementation is an important improvement to take in account and make sure it can cover use cases that may appear.

Finally, the next steps regarding the implementations are to validate the integration of the prototype with a federated 5G architecture, first validating its integration with an infrastructure layer (i.e., OpenStack) as well as validating the performance with highers layers (i.e., an orchestration layer). In order to achieve so, work towards the implementation of a mechanism driver to be integrated with Neutron's ML2 plugin in OpenStack is already being carried on.

References

[1] J. Day, "How in the heck do you lose a layer!?," in *2011 International Conference on the Network of the Future*, pp. 135–143, 2011.

[2] T. Nolle, "Cloud-native for carrier cloud." Accessed: 2021-12-6.

[3] G.-P. S. N. WG, "From webscale to telco, the cloud native journey," 2018.

[4] R. R. Kewin Rausch, "Progressive network transformation with rina," 2017.

[5] M. P. d. L. Leon, R. Ranganathan, D. Bainbridge, K. Ramanarayanan, A. Corston-Petrie, and E. Grasa, "Multi-operator ipc vpn slices: applying rina to overlay networking," in *2019 22nd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pp. 72–75, 2019.

[6] E. Grasa, L. Bergesio, M. Tarzan, D. Lopez, S. van der Meer, J. Day, and L. Chitkushev, "Mobility management in rina networks: Experimental validation of architectural properties," in *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 1–6, 2018.

[7] S. Gaixas, J. Perello, D. Careglio, E. Gras, M. Tarzan, N. Davies, and P. Thompson, "Assuring qos guarantees for heterogeneous services in rina networks with δq," pp. 584–589, 12 2016.

[8] B. Forum, "Quality of experience delivered (qed) project press release." Accessed: 2021-12-6.

[9] S. Vrijders, D. Staessens, D. Colle, F. Salvestrini, E. Grasa, M. Tarzan, and L. Bergesio, "Prototyping the recursive internet architecture: the irati project approach," *IEEE Network*, vol. 28, no. 2, pp. 20–25, 2014.

[10] P. Teymoori, M. Welzl, S. Gjessing, E. Grasa, R. Riggio, K. Rausch, and D. Siracusa, "Congestion control in the recursive internetworking architecture (rina)," pp. 1–7, 2016.

[11] E. Gras, O. Rysavy, O. Lichtner, H. Asgari, J. Day, and L. Chitkushev, "From protecting protocols to layers: Designing, implementing and experimenting with security policies in rina," 05 2016.

[12] S. van der Meer, J. Keeney, L. Fallon, S. Feghhi, and A. de Buitléir, "Large-scale experimentation with network abstraction for network configuration management," pp. 60–65, 2019.

[13] E. Grasa, L. Bergesio, M. Tarzan, D. Lopez, J. Day, and L. Chitkushev, "Seamless network renumbering in rina: Automate address changes without breaking flows!," pp. 1–6, 2017.

[14] A. F4.4, "Arcfire d4.4. execution of experiments, analysis of results and benchmarking of kpis." Accessed: 2021-12-10.

[15] "Open-verso project." Accessed: 2021-12-12.

[16] L. Rizzo, "netmap: a novel framework for fast packet i/o," pp. 101–112, 2012.

[17] V. Maffione, "Netmap as a backend for vms." Accessed: 2021-12-8.

[18] E. Security, "What's the max speed on ethernet?." Accessed: 2021-12-19.

[19] V. Maffione, L. Rizzo, and G. Lettieri, "Flexible virtual machine networking using netmap passthrough," 06 2016.

[20] M.-A. Kourtis, G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, and F. Liberal, "Enhancing vnf performance by exploiting sr-iov and dpdk packet processing acceleration," *Proceedings of 2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, 2015.

[21] V. Maffione, "Netmap as a backend for vms." Accessed: 2022-1-10.

[22] L. Rizzo, "Revisiting network i/o apis: The netmap framework," *ACM Queue*, vol. 10, p. 30, 03 2012.

[23] L. Rizzo and G. Lettieri, "Vale, a switched ethernet for virtual machines," 12 2012.

[24] M. L. Luigi Rizzo, "A netmap application to bridge two network interfaces." Accessed: 2022-1-10.

[25] L. Rizzo, "Packet generator for use with netmap." Accessed: 2022-1-13.

[26] J. Day, *Patterns in network architecture: A return to fundamentals*. Pearson Education, 2007.

[27] J. Day, I. Matta, and K. Mattar, "Networking is ipc: A guiding principle to a better internet," in *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, (New York, NY, USA), Association for Computing Machinery, 2008.

[28] J. Day, "How naming, addressing (and routing) are supposed to work," 2016.

[29] V. Maffione, F. Salvestrini, E. Grasa, L. Bergesio, and M. Tarzan, "A software development kit to exploit rina programmability," in *2016 IEEE International Conference on Communications (ICC)*, pp. 1–7, 2016.

[30] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp. 122–1225, 2019.

[31] Y. Wang, I. Matta, F. Esposito, and J. Day, "Introducing protorina: a prototype for programming recursive-networking policies," *SIGCOMM Computer Communications Review*, vol. 44, pp. 129–131, July 2014.

[32] V. Vessely, M. Marek, T. Hykel, and O. Rysavy, "Rinasim: Your recursive internetwork architecture simulator," *Proceedings of the OMNeT++ Community Summit 2015*, September 2015.

[33] V. Maffione, "A light rina implementation." Online: https://github.com/rlite/rlite, 2017.

[34] S. Fulton, "Service mesh: What it is and why it matters so much now." Accessed: 2021-12-27.

[35] E. Grasa, L. Bergesio, M. Tarzan, D. Lopez, S. van der Meer, J. Day, and L. Chitkushev, "Mobility management in rina networks: Experimental validation of architectural properties," in *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 1–6, 2018.

Appendices

A Experiment for benchmark the performance of the RINA stack implementation datapath

A.1 Experiment Setup

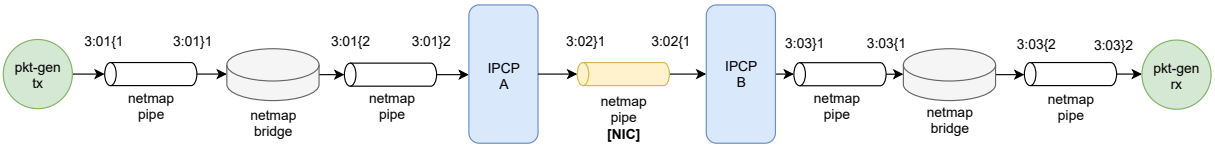


Figure 26: The experiment set up

- Maximum 2 netmap pipes per VALE port memory region. IOW, maximum pipes between you can zerocopy is 2.
- Using `pkt-gen` as the packet generator.
- There is no need to use VMs because with netmap passthrough, `pkt-gen` has the same performance in or outside a VM.

A.1.1 Start the IPCPs

Build them if not already built:

```
mkdir build
cd build
cmake ..
make ipcp
make l2-switch
```

Now start IPCP A and IPCP B:

```
# IPCP A:
sudo ./out/ipcp ../test/11_two_ipcps_config_file/ipcp_a.json
```

```
# IPCP B:
sudo ./out/ipcp ../test/11_two_ipcps_config_file/ipcp_b.json
```

If everything went fine, you should see an output telling that enrollment went fine in each IPCP:

```
[INFO] enr_tsk_handle_stop_r (/home/sergio/i2cat/netmap-rina-router/ipcp/enrollment-ta
```

A.1.2 Start the Management Agents

Read the MA README to know how to install dependencies and start the management agents.

Start MA A and wait for slice request:

```
python management_agent.py -c ./test/config_a.json
```

```
MA_A>: --start
MA_A>: -c -id 1 -rapn MA_B -r slave -p vale3:01}2
```

```
{management_agent.py:155} INFO - ManagementAgent: Creating slice
```

Start MA B and create a slice request:

```
python management_agent.py -c ./test/config_b.json
```

```
MA_B>: --start
MA_B>: -c -id 1 -rapn MA_B -r master -p vale3:03}1
```

If all the flow allocation workflow went fine, you should see an ACK coming from IPCP B to MA B:

```
MA_B>: Received message: opcode: M_ACK
```

Note that the MA will complain because it cannot find the layer 2 process. It's fine, I just removed some lines to avoid the MA to invoke the L2-SW.

A.1.3 Invoke the bridges / L2-SW

Invoke the "IPCP A" bridge:

```
sudo bridge -i vale3:01}1 -i vale3:01{2
```

Invoke the "IPCP B" bridge:

```
sudo bridge -i vale3:03{1 -i vale3:03{2
```

A.2 Run the experiment

On the IPCP B side, run a the netmap `pkt-gen` tool in receive mode:

```
sudo pkt-gen -i vale3:03}2 -f rx
```

Finally, on the IPCP A side, run a the netmap 'pkt-gen' tool in transmit mode:

```
sudo pkt-gen -i vale3:01{1 -f tx
```

A.3 Core pinning

The experiment might run faster if the affinity of the process is constrained to a CPU core.

```
$ pgrep ipcp & pgrep bridge
50748
```

```
50761
49243
49251

$ sudo taskset -cp 0,1 50748
pid 50748's current affinity list: 0-7
pid 50748's new affinity list: 0,1

$ sudo taskset -cp 2,3 50761
pid 50761's current affinity list: 0-7
pid 50761's new affinity list: 2,3

$ sudo taskset -cp 4,5 49243
pid 49243's current affinity list: 0-7
pid 49243's new affinity list: 4,5

$ sudo taskset -cp 6,7 49251
pid 49251's current affinity list: 0-7
pid 49251's new affinity list: 6,7
```

B Physical deployment. 1 IPCP in each server in same datacenter

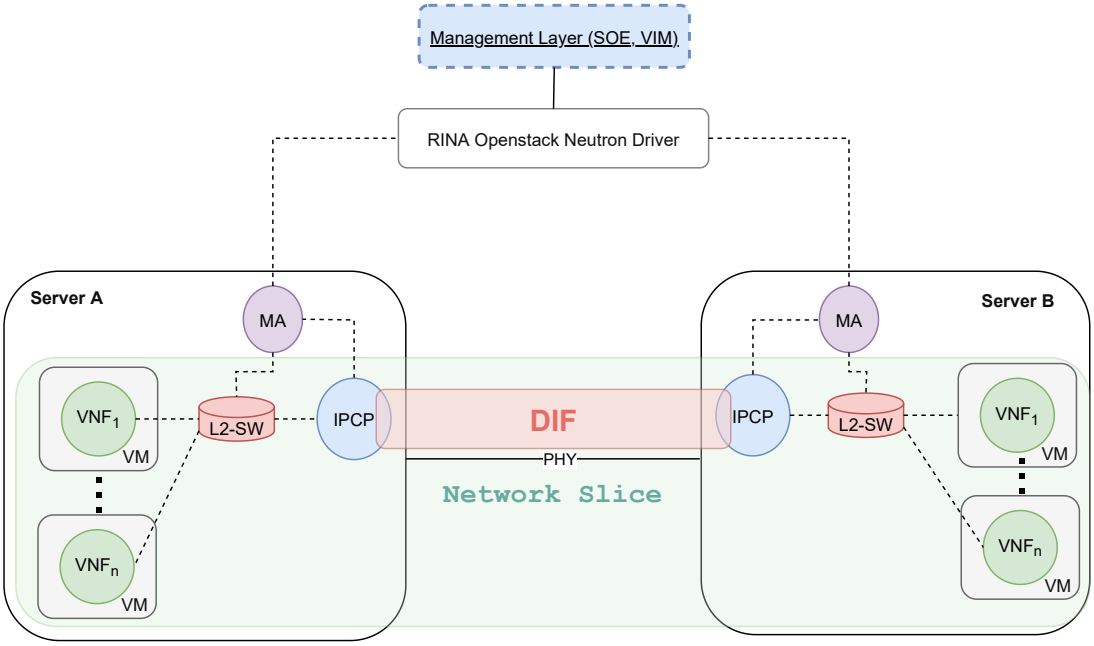


Figure 27: Physical testbed setup

B.1 Netmap

First thing we need to do is to install netmap-patched drivers in the server.

> In `espriuX` case, NIC drivers are `igb` in both servers, then all the commands are related to the `igb` driver.

First, clone the netmap repo in any directory:

```
git clone https://github.com/luigirizzo/netmap
```

Now, move to the netmap directory, and run the `configure` command. Make sure you have kernel headers matching your installed kernel. For some drivers (`e1000`, `veth`, `forcedeth`, `virtio-net` or `r8169`). The easiest way I found for downloading the sources is to download the sources from launchpad.net. To do so, in the google search bar type: site:launchpad.net `linux-image-4.4.0-31-generic`. That will look only for results in launchpad.net and for exact name search.

```
./configure --drivers=igb
```

> We specify the driver we want to use in order to not download all the sources of the drivers that netmap supports (which is the default option). In this case, we are using the `igb` driver.

Now, compile and install the netmap kernel module with:

```
make
sudo make install
```

Now we have the netmap kernel module installed in the kernel (you can double-check with `lsmod | grep netmap`).

Now we have to replace the current NIC driver with the netmap-patched driver we just compiled. Please do that in a physical tty (not ssh or remote access), because when the NIC driver is removed, you will automatically lose connection to the remote machine. Later I'll put also a "remote install method" that worked for me.

Assuming you are in the netmap directory:

- Remove the `igb` driver:

```
rmmod igb
```

- Install `netmap` module if not done previously:

```
insmod ./netmap.ko
```

- Install the netmap-patched driver:

```
insmod igb-5.3.5.20/src/igb.ko
```

B.1.1 Install drivers the remote way

If there is not physical access to the server, you can install the netmap-patched driver in the remote machine using the following commands (at least that worked for me).

First create a bash script that will be executed in the remote machine. The script looks like that:

```
rmmod igb
modprobe netmap
insmod path/to/netmap/igb-5.3.5.20/src/igb.ko
```

Now, run the script in the remote machine. **Make sure it is run in background! Otherwise, the process will stop in the `rmmod igb` step, and you'll lose the ssh access**

```
sudo ./remote_install.sh &
```

B.1.2 Final Netmap configurations

Enable communication within the network stack In order to enable communication within the network stack, we need to disable NIC offloads. netmap does not program the NICs to perform offloadings such as TSO, UFO, RX/TX checksum offloadings, etc. As a result, in order to let netmap applications **correctly interact with the host rings**, you need to disable these offloadings:

```
ethtool -K eth0 tx off rx off gso off tso off gro off lro off
```

Wait time for the NIC to be ready Sometimes when netmap opens a physical NIC in netmap mode it needs to wait a bit for negotiation and the link might last a few seconds to be up. Beware of that, because a timer may be useful when talking to physical ports to let link negotiation complete before starting transmission (netmap `bridge` application leaves 4 seconds for negotiation in physical links).

However, I've not encountered the need for a wait time with virtual netmap ports such as netmap pipes.

C Detailed Virtualized testbed Setup

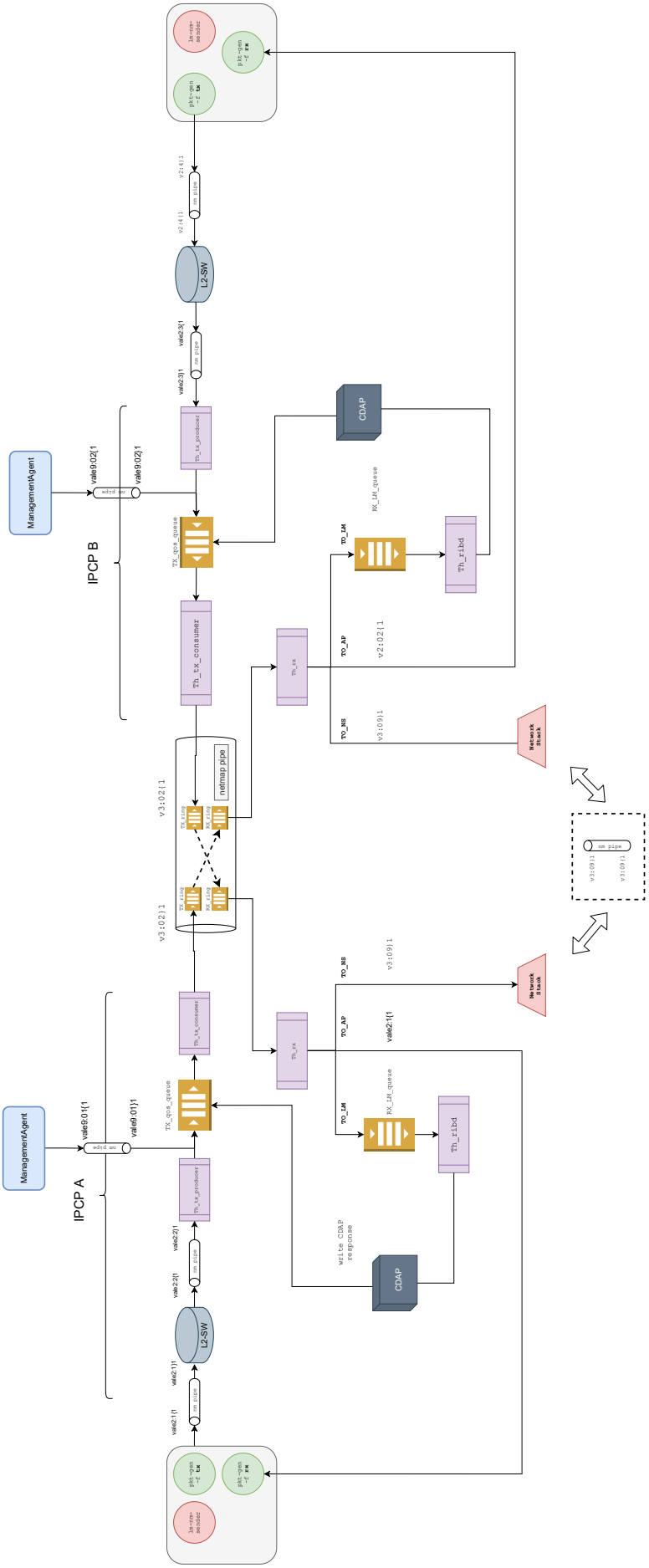


Figure 28: Detailed diagram of the virtualized testbed setup

D UML of the IPCP

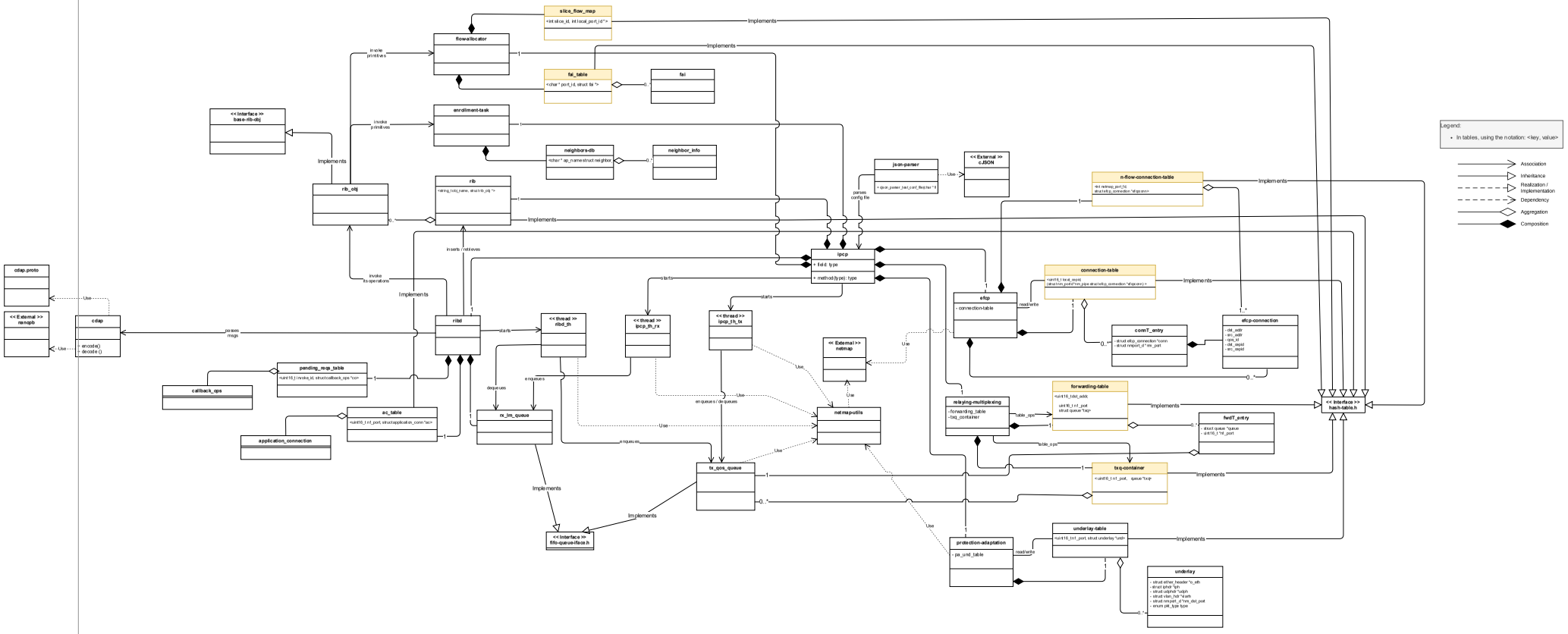


Figure 29: UML Diagram of the IPCP with its main components. In yellow, hash tables that contain internal mappings