Ethernet shim DIF for Recursive Inter-Network Architecture Simulator (RINASim)

Karl H. Totland

Thesis submitted for the degree of
Master in Informatics: Programming and System architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2020

Ethernet shim DIF for Recursive Inter-Network Architecture Simulator (RINASim)

Karl H. Totland

Abstract

In a rapidly changing world with larger and larger volumes of data being transmitted each day, the internet should be continually improved to be as efficient as possible. However, fundamental problems with the architecture have been revealed, that require additional solutions to be built on top of the already existing infrastructure.

Recursive Inter-Network Architecture (RINA) is a new clean-slate architecture designed with the aim of solving many of the fundamental problems that exist in the Internet today, with several working implementations that are already being used as part of research projects. RINA is built on the concept that all networking is Inter-Process Communication (IPC), where the central component that delivers IPC is the Distributed IPC Facility (DIF), which is analogous to the layers of TCP/IP and can be stacked recursively over different scopes. Adopting RINA as an architecture requires additional work however, which is where the work presented in this thesis comes in.

The aim of this thesis is to define and implement a translation layer for Ethernet that works with the interface supplied by a DIF in the simulation model library RINASim. The shim DIF is required for reusing legacy infrastructure in new networks, and as an addition to RINASim it will be useful for simulating realistic migration use-cases.

Preface

First of all, I want to thank my supervisor Michael Welzl and my co-supervisor Marcel Marek for their unlimited patience and willingness to help even when I barely contacted them until the end.

I also want to thank Thor and Jess for feeding me during the weeks before the deadline, and the help I got with proof-reading along the way.

Contents

List of Figures

Chapter 1

Introduction

In the second half of the 20th century, multiple organisations and academic institutions were heavily involved in developing systems for distributed computing. At the forefront of these systems was the network ARPANET, which is widely regarded as the foundation of the current Internet. The initial goal for ARPANET was to demonstrate the viability of packet-switched networks for allowing simultaneous data transfer to different nodes on a network[1], but this goal alone did not incentivise correct architectural decisions. Even though ARPANET seemed to have several deficiencies in its architecture, the TCP/IP protocol suite[1] adopted many of the same principles.

The Internet has changed surprisingly little over the last few decades. There are only two transport protocols[2] that are supported universally, and middleboxes[2] make it very difficult to make use of non-standard protocols. Since there is little to no room for further developing or changing existing features[3], growing routing tables[4], a very limited address space[5], and no inherent security mechanisms as part of its core architecture[6], the current Internet is not well-suited to tackle networking on a global scale. The separated scopes of the layers appear to be convenient and straightforward, but many of the mechanisms supplied could benefit from not being limited to only one scope. Many protocols have also become needlessly complicated due to shortcomings of the services delivered by lower layers.

Rather than supplying new mechanisms by building overlays to fit onto the existing Internet, fundamental changes are needed. This is where Recursive Inter-Network Architecture (RINA) comes in: a clean-slate architecture

[1]More commonly known as “the Internet protocol suite”, or just “the Internet”.
[2]Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).

intended to solve the issues imposed by the Internet by focusing on the fundamentals of networking.

RINA is a network architecture intended to solve the shortcomings of the Internet at a fundamental level. It is built from the ground up on the principles that constitute Inter-Process Communication (IPC), and aims to be a complete replacement for TCP/IP. The mechanisms that RINA supplies are configurable through an extensible set of policies, and facilitating the addition of new policies is a central part of the architecture. For RINA to ever stand the chance of being deployed in real networks however, adoption strategies need to be in place, as well as a set of translation layers so previous infrastructure is not completely invalidated.

1.1 Motivation and purpose

Employing a new architecture in a landscape of rigid networking systems requires more than just deploying them to the existing systems. Without translation layers between new and legacy architectures, utilisation of the already existing infrastructure would be impossible, making any new architecture very difficult to adopt on a global scale. Since most computers and servers today are equipped with Ethernet interfaces, these translation layers could be vital for architectures that deploy different addressing schemes, as they allow a host to adopt the new architecture with almost no compromises regarding their existing setup. In other words, they do not have to use architecture-specific hardware.

Another important factor for allowing new architectures to gain interest and trust, is the presence of accurate simulation frameworks for evaluating and visualising its concepts. Any such architecture would benefit from having robust, feature complete, and precise ways of testing their mechanisms and real-world applications. The simulation model library RINASim was made for and modelled after RINA with this in mind, and is used to test various policies to evaluate their usefulness.

However, RINASim lacks the possibility of simulating realistic scenarios where legacy networks are in use. Instead, existing components in the simulation scenarios are marked as “shims” where applicable, and their Quality of Service (QoS) capabilities are limited accordingly. Implementing a translation layer for Ethernet is the first step towards allowing RINASim to accurately simulate scenarios concerning the deployment of RINA networks over legacy networks.

This thesis has two primary goals. The first is to examine the viability of interfacing RINASim with INET, which is the TCP/IP stack implementation for OMNeT++. The second goal is to design and implement a shim layer over Ethernet in RINASim to provide the first step towards demonstrating the ease of adopting RINA as an architecture for use in real networks.

1.2 Structure

The chapters are organised in a way that first lays out necessary theoretical concepts, before describing the frameworks used and the implementation.

Chapter 2 will provide information on the technological background and specifications of RINA.

Chapter 3 explains the design of the Ethernet shim Distributed IPC Facility (DIF). It also explains Ethernet in light of RINA, which makes it easier to understand the requirements of the shim layer.

Chapter 4 describes the simulation framework OMNeT++, and the implementation of the RINA stack in this simulator framework: RINASim. It also describes INET, and includes a discussion on how it is included as part of RINASim.

Chapter 5 delves into the implementation of the Ethernet shim DIF in RINASim, as well as explaining additional changes that need to be made in the existing codebase to facilitate the implementation.

Chapter 6 is the evaluation chapter, which various examples of the Ethernet shim DIF in action. At the end of the chapter is a discussion of the implementation results.

Chapter 7 provides a conclusion, closing remarks, and some future work that could be based on the work of this thesis.

Chapter 2

Recursive Inter-Network Architecture

RINA is a network architecture designed to be a complete replacement for the Internet. It is built upon the principle that networking is IPC, and only IPC. Similarly to how the Internet works, RINA is built upon a set of layers, but where the Internet separates mechanisms by scope, RINA provides the same set of mechanisms within each scope[1].

As a consequence of separating mechanisms by scope, the networking stack of RINA only consists of one generic layer that delivers IPC: the DIF. Each stacked DIF provides services to its (N+1) DIF[2], and the mechanisms provided are completely isolated within each DIF. The DIF consists of a set of one or more IPC Processes (IPCPs) that are enrolled. Enrollment happens through the Common Application Connection Establishment (CACE) phase, where an IPCP either asks a member of a DIF to join the existing DIF, or attempts to create a new DIF if no supporting DIF is found. The CACE phase is carried out using a unified and simplistic management protocol called the Common Distributed Application Protocol (CDAP). These mechanisms are handled by various submodules, the most important of which will be described in section 2.4. The recursive scoping can allow a single DIF to span several networks, as seen in fig. 2.1.

In RINA, any type of network node is denoted as a Computing System (CS), whether they are hosts or routers. RINA also operates with the terms Service

[1]Scope in this context means communication ranges: in the Internet, layers 1 & 2 provide communication between directly connected nodes, while layers 3 & 4 provide communication between indirectly connected nodes through routing and control mechanisms.

[2]Note that a local member of the (N+1) DIF may sometimes be referred to as an "application" in the context of an underlying DIF.

Data Unit (SDU) and Protocol Data Unit (PDU). SDU signifies a packet that arrives from an upper DIF or Distributed Application Facility (DAF), before the protocol data of the current DIF is added, while PDU represents the packet after the header is added. SDUs are passed to an (N+1) IPCP or application, while PDUs are passed to an (N-1) IPCP.

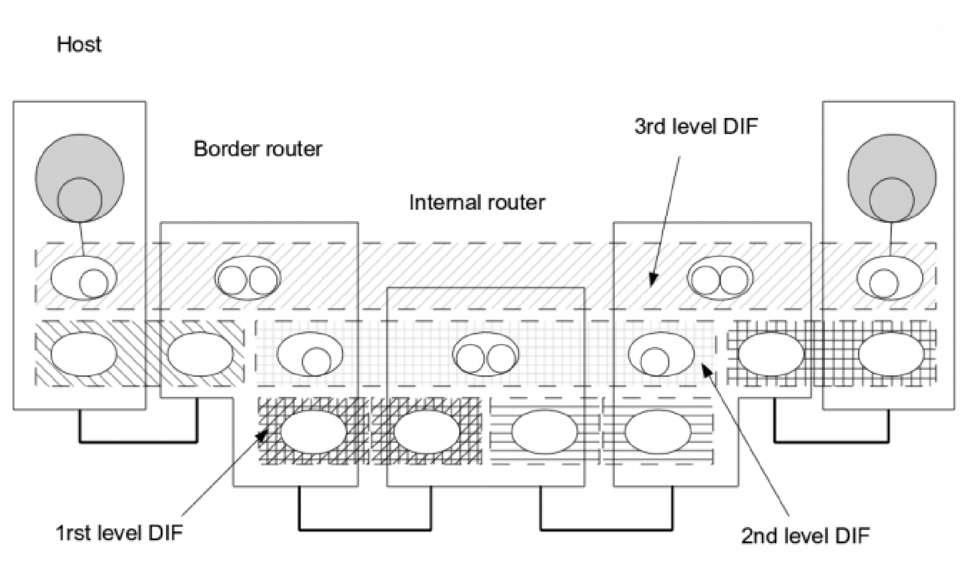


Figure 2.1: An example of a RINA network spanning multiple scopes, from Trouva, Grasa, Day, *et al.* [7]

Allowing for the use of the same set of mechanisms over different scopes has a number of advantages compared to the Internet. If the application requires it, a PDU that is lost in transit may be retransmitted from the last intermediary node instead of its originating end-point, which could reduce traffic, which in turn could result in lower congestion, as well as reducing the time it takes for handling of packet loss. There are also several security features built into the architecture, e.g. authentication when joining a DIF and encryption of PDUs. Although the Internet has similar security mechanisms through Transport Layer Security (TLS) and IPSec, they were built on top of the architecture, and not as part of it[8].

The mechanisms that deliver these features are controlled by policies, which in a sense are the parameters of the mechanisms, and specific setts of policies can be requested by the applications that use a DIF to fit the requirements that the application has for data transfer. An application can for example configure whether or not a connection requires the receiver to ACK, if it requires encryption of PDUs, or if it requires additional mechanisms like retransmission and flow control.

2.1 Nature of Applications and Application Protocols

There is a more generalised version of the DIF that is used to maintain shared state between applications, which is called the DAF. A DAF consists of two or more Application Processes (APs) that want to complete some function. The parts of the AP that are considered to be inside the network, and that directly utilise underlying DIFs, are called Application Entities (AEs). An AP can have several AEs to facilitate different kinds of communications for a given connection. For any form of communication, the stateless application protocol CDAP is used. CDAP has a simple set of pairs of primitives: `create/delete`, `start/stop`, and `read/write`. These may be used to construct any form of distributed application regardless of its needs, whether they are voice traffic, streaming, online video games, or file transfer. Voice traffic and online video games may for instance have real-time requirements where packet loss is accepted, while streaming and file transfer requires the connection to be reliable with its associated overhead.

To access underlying DIFs, an AP uses the IPC Resource Manager (IRM). The IRM delegates flow allocation calls to appropriate IPCP, polls the DIF Allocator (DA) about where to find a requested application, creation of DIFs and DAFs, and manages the use of flows by AEs. The DA is responsible for returning a list of DIFs where a destination application can be found.

2.2 Naming and addressing

While many of the proposed early network architectures had more complete naming schemes[3], the incomplete solution of ARPANET was the one that ended up being the inspiration for the Internet[9]. In ARPANET, there was a single address used for the interface, which was also used for routing. The Internet took this further by adding another address as a mapping onto the data link layer interface address through the network layer: Internet Protocol version 4 (IPv4) addresses. What makes this problematic, is that there is no specific node address, making it very difficult to know if two Internet Protocol (IPv4) addresses belong to the same node. This leads to the routing tables becoming unnecessarily, since every interface address would need to be recorded instead of just a node name. Additionally, multihoming is very difficult to implement, as it requires some additional external mapping of multiple interface addresses to a unique node address.

[3]Like CYCLADES, XNS, and OSI[7].

With directory structures, indirection could be provided to fix the growing routing tables. The Internet achieves this to some extent through the use of Network Address Translation (NAT)[10] and private networks, but this requires the use of additional infrastructure and is not part of the network layer protocol.

RINA has four identifiers that together form a more complete naming scheme:

- **Application Process Name (APN):** A globally unambiguous, but system dependent name used to reach remote applications of a specific type.
- **Application Process Instance Identifier (API-id):** When used together with the APN, this can specify a specific instance of an AP.
- **Application Entity Name (AEN):** Identifies a type of AE within an AP. AEs may have different requirements for data transfer, which is why they are separated by type.
- **Application Entity Instance Identifier (AEI-id):** Distinguishes between different instances of an AE when used together with the AEN. This could be useful for communication where several streams are used.

Application Naming Information (ANI) is a four-tuple which contains these names, forming a complete set of identifiers for naming an application. However, only the APN must be defined within the ANI and the others can be left unspecified, but defining these identifiers as well allows more granularity in regards to controlling exactly where data should be sent. This set of names provides the IPCP with the means to construct a complete set of identifiers, which together name the node in the context of a DIF:

- **Address:** An APN that is unambiguous within the layer[4], which can be used to reach the IPCP.
- **Port ID:** The identifier of the allocation AE instance, which manages the flow state through the use of CDAP. This identifier is also used as a handle for the application utilising the flow.
- **Connection End-Point Identifier (CEP-id):** The identifier of an Error and Flow Control Protocol (EFCP) instance. The source and destination CEP-ids together with the identifier of the QoS cube in use form the connection identifier, which is used to distinguish between different flows between the same set of IPCP.

[4]Layer meaning the set of DIFs that this IPCP can enroll to.

Routing is done by utilising the Address of an IPCP, and the Routing AE of an IPCP can return a set of underlying ports which can be used to easily achieve multihoming if required.

2.3 Security

The Internet does not provide any inherent security mechanisms in its architecture[11]. While IPSec secures data, the nature of how the Internet is built exposes the addresses and ports, which means that anyone listening in on the traffic can figure out where it is going. Ports are also not allocated when needed, but rather subscribed to by an application, and many services use well-known ports to allow applications to connect to them. While some of these issues may be mitigated by using overlay networks or Virtual Private Networks (VPNs), these additional mechanisms are built on top of the architecture, and not as part of it.

In RINA there are several security mechanisms to mitigate these issues that are built into the architecture. A few of them are:

- **Authentication procedure:** To gain access to a DIF, an authentication procedure is necessary. Only IPCP that are enrolled to the DIF through this allocation procedure may be used to access an application that is registered to the DIF.
- **Encryption of PDUs:** Before being passed to an (N-1) IPCP, PDUs can be encrypted to mask both the user and protocol data. When an encrypted PDU passes through an intermediary node with an IPCP enrolled in the same DIF, that IPCP will need to decrypt the PDU to figure out where to send it next. This makes it very difficult to figure out where a PDU is going.
- **Random port and CEP-id allocation:** The port and CEP-id used for a flow are randomly allocated. An agent trying to impersonate an enrolled IPCP would have to know both the port and CEP-id for both end-points to deliver a SDU to a registered application.

2.4 Networking in RINA

As mentioned in the beginning of this section, RINA is built upon the principle that networking is only IPC. Through the use of a more specialised set of mechanisms, the IPCP can transfer data in any form that an overlying application may require. In addition to the mechanisms provided for APs, the IPCP has a Flow Allocator (FA), an internal Resource Allocator (RA), and a set of modules that allow data transfer. All of these can be seen as AEs responsible for different aspects of management within the IPCP. The IPCP has a set of five Application Programming Interface (API) calls, that can be used to manage or utilise connections: `allocateRequest`, `allocateResponse`, `deallocate`, `send`, and `receive`.

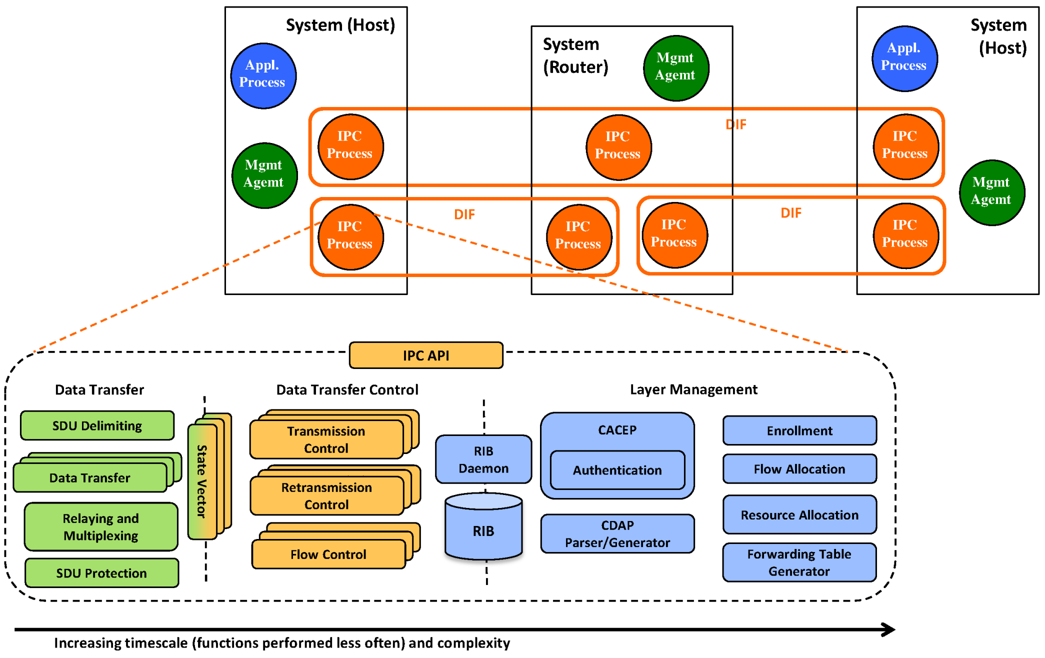


Figure 2.2: Components of an IPCP, from [12]

The FA is responsible for the creation and management of flows, and it is the module that handles the `Allocate_Request`, `Allocate_Response` and `Deallocate` API primitives. The `Allocate_Request` call takes source and destination APN, as well as a set of QoS requirements, and returns a port ID to use as a handle to refer to the allocated flow. The given set of QoS requirements dictates the mechanisms that the flow will utilise. An `Allocate_Response` is returned after flow allocation completes, indicating whether it failed or succeeded. When allocating a flow, the FA creates an FA Instance (FAI) that manages a flow from creation until deletion.

A central element in RINA is the use of QoS cubes to keep track of the capabilities of a flow. A QoS cube contains information such as how much data throughput a flow supports, how many SDUs it can send, how high the error rate is, and more. An application may request a certain set of QoS capabilities through the QoS requirements parameter supplied with the `allocateRequest` primitive, but it is up to the IPCP to decide which resources are allocated based on the current load.

The QoS cubes are managed by the RA, which is also responsible for reallocating resources within the IPCP. The RA communicates with the components of an IPCP to manage mechanisms like maximum throughput of ports, to which underlying ports flows are assigned, and regulating the capabilities of individual flows.

The module that enables data transfer within the DIF is the EFCP. On finalisation of flow creation, an EFCP Instance (EFCPI) is created to manage the state of data transfer in a connection. An EFCPI has two submodules: Data Transfer Protocol (DTP) and Data Transfer Control Protocol (DTCP). DTP is mandatory, and is responsible for mechanisms that are tightly bound to the PDU. Tightly bound means that the same state needs to be maintained on both ends of a connection, and therefore stored as part of the protocol header data of the PDU, like the PDU sequence number. DTCP however, is not mandatory, and is responsible for loosely bound mechanisms, which are maintained by each IPCP. These mechanisms are not directly related to the PDU, such as retransmission and flow control.

EFCP is heavily inspired by Richard Watson's Delta-t protocol[13], where connection and synchronisation are completely decoupled. Its essence is that connections always exist, and that synchronisation only requires an upper bound on three timers: Maximum Packet Lifetime (MPL), maximum time to wait before sending ACK (A), and the maximum time before not sending any more retries (R). $\Delta t = MPL + A + R$, and synchronisation is computed as $3\Delta t$ for the sending end, and $2\Delta t$ for the receiving end. If synchronisation is broken due to the expiration of these timers, the DTCP state information is discarded. After this expiration, resynchronisation can easily be achieved by sending a PDU with the Data Run Flag (DRF) set, which signals the start of a new "data run".

There are two additional modules that directly operate on incoming SDUs and PDUs: the delimiter module, and the Relaying and Multiplexing Task (RMT). The delimiter module makes sure SDUs from an overlying application do not exceed the size permitted by the QoS cubes of an IPCP. It is also instantiated as part of a flow, and depending on the QoS requirements imposed on the flow, additional mechanisms like partial delivery[5] can be enabled.

The RMT is the last step[6] before passing a PDU to the medium or an underlying IPCP. It can also multiplex PDUs to achieve multihoming, where the PDUs may be routed over several different paths, which may be desirable when redundancy is required.

[5]Where partial delivery denotes passing fragmented SDUs to an upper IPCP or application without waiting for an entire SDU to be reassembled.

[6]PDU encryption happens later still if it is enabled.

2.5 Deployment

While a full-scale deployment of this networking architecture is unlikely to be seen in the coming years, gradual migration is possible. Through the use of translation layers, the RINA stack could be deployed on top of legacy architecture. There are several translation layers — or shims — specified as part of the RINA specification[12], two of which support translation over Ethernet and TCP/IP. With the implementation of these shims, nodes supporting the RINA stack could communicate with each other over the already existing Internet infrastructure.

Chapter 3

Design of the Ethernet shim DIF

The goal of this chapter is to provide a thorough description of the design requirements for the Ethernet shim DIF. It must deliver an identical API to the API of a normal DIF, and should not require an upper IPCP to act differently in any way. The structure of the chapter is based on and set similarly to the design document for the Ethernet shim DIF[12], and will first give a description of the Ethernet protocol in terms of RINA, before describing the design of various components and the state transitions that are expected within the shim IPCP.

3.1 Inter-Process Communication in Ethernet

Ethernet, which encapsulates the link and physical layers of the TCP/IP stack, can be seen as a DIF in terms of RINA. In its most basic use-case, it allows communication between two end-points directly connected over medium, similar to how the lowermost DIF in a normal RINA configuration would. The next set of layers of the TCP/IP stack, the network and transport layers, can be seen as yet another DIF.

Ethernet is very minimal, and supplies no explicit control mechanisms on its own apart from checksum verification, through the use of which a datagram may be dropped if the computed checksum does not match the one supplied as part of a frame. At its core, only source and destination Media Access Control (MAC) addresses, Ethernet type, and the aforementioned checksum are part of the datagram, but an IEEE 802.1Q tag may optionally be added to support Virtual Local Area Networks (VLANs). A VLAN allows traffic to be isolated from other VLANs spanning the same Ethernet segment. In terms of RINA, this allows several DIFs to exist over the same Ethernet segment,

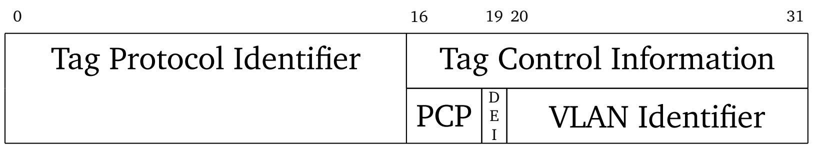


Figure 3.1: The IEEE 802.1Q VLAN tag

which is relevant when utilising switches or bridges. The VLAN tag can be seen in fig. 3.1. The "Tag Protocol Identifier" is set to `0x8100` in order to identify the tag, and otherwise only the "VLAN identifier" is relevant.

There is no explicit enrollment phase in Ethernet, and datagrams may be sent to any host connected over the same Ethernet segment, provided the destination MAC address is known. MAC addresses can be thought of as CEP-ids, but while CEP-ids in an IPCP are set upon flow creation, a MAC address is set when an interface is manufactured. Additionally, there is no mechanism for the allocation of flows, and only the destination MAC address is required to be supplied for a datagram to reach its destination. This means that any "member" of an Ethernet segment can freely send to other members, which has security implications.

Since the only identifiers for traffic are source and destination addresses, Ethernet only supports one flow per destination, and this flow has no guarantees for reliability apart from dropping SDUs if the Frame Check Sequence (FCS) does not match with computed Cyclic Redundancy Check (CRC) of the frame. An Ethernet interface therefore only supports one QoS cube that is unreliable, where many of the QoS parameters are dependent on the standard that it supports, like average bandwidth and delay.

The unreliable QoS cube provided by the Ethernet shim DIF can be seen in fig. 3.2. There are no guarantees in regard to whether any Ethernet frames will be received in order, and all supplied SDUs from an application must be within the Maximum Transmission Unit (MTU) provided by the Ethernet interface, which is normally 1500 bytes.

Logical Link Control (LLC) can be utilised to let several applications use the same DIF through the use of Service Access Point (SAP) identifiers. This mechanism could also be used to support multiple flows between two hosts, and the Destination Service Access Point (DSAP)/Source Service Access Point (SSAP) pair can be seen as CEP-ids instead of the MAC address. The header format used for LLC can be seen in fig. 3.3. LLC will not be used in this specifications, but a new shim DIF could be defined in the future that makes use of it.

Name	unreliable
Average bandwidth	Depends on standard
Average SDU bandwidth	Depends on standard
Peak bandwidth duration	Depends on standard
Peak SDU bandwidth duration	Depends on standard
Burst period	Depends on standard
Burst duration	Depends on standard
Undetected bit error rate	Depends on standard
Partial delivery	No
Incomplete delivery	No
Order	No
Max allowable gap in SDUs	Any
Delay	Depends on standard
Jitter	Depends on standard

Table 3.2: Unreliable QoS cube supplied by Ethernet shim DIF, based on the QoS cube definition of the shim DIF over IEEE 802.1Q of Investigating RINA as an alternative to TCP/IP (IRATI)[14].

DSAP	SSAP	Control	Information
1 byte	1 byte	1–2 bytes	M bytes ($M \geq 0$)

Table 3.3: An LLC header

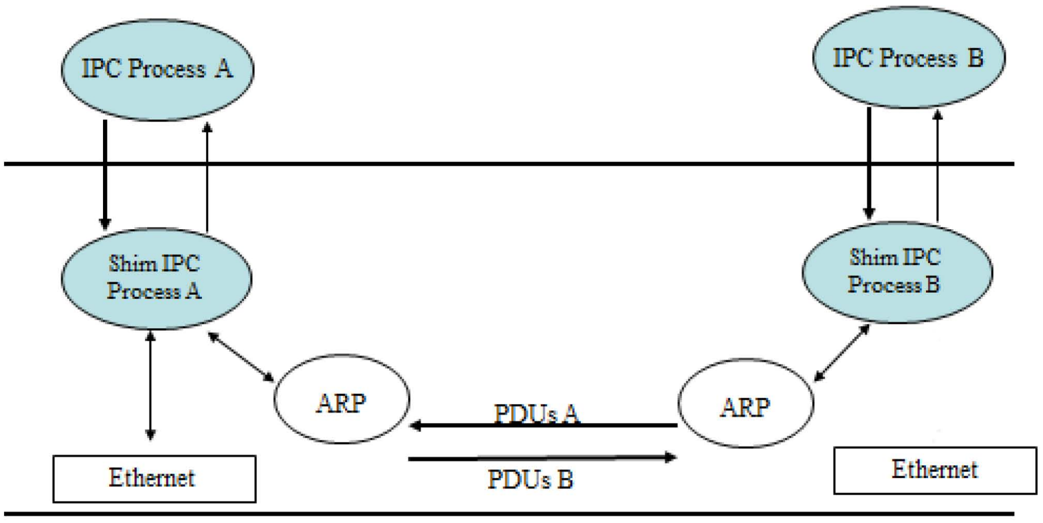


Figure 3.4: An illustration of an Ethernet shim DIF as a peer-to-peer DIF[12]

3.2 Ethernet shim layer

Based on the previous section, we can set the requirements for a shim layer over Ethernet. The aim of the shim DIF should not be to deliver more than what the Ethernet standard in use is able to manage, and this particular shim DIF is meant to support running over interfaces that support the IEEE 802.3 standard. The QoS that the shim DIF is able to deliver is dependent on the specific IEEE 802.3 standard supplied, and is therefore limited by the capabilities of the physical medium. Each shim DIF is specified by a VLAN (IEEE 802.1Q) tag, and each VLAN is a separate DIF. It is also assumed that all traffic passing within this VLAN is exclusively shim DIF traffic. All members of a VLAN are assumed to be members of the same shim DIF, and as such any shim IPCP that joins a VLAN is enrolled in the shim DIF.

As specified in the previous section, the only identifiers supplied by Ethernet are source and destination addresses. It is therefore only possible to distinguish between different destination and source end-points. The consequence of this in terms of RINA, is that only one flow is supported per end-point in the Ethernet shim DIF. Support for multiple flows per end-point could be added through the implementation of a shim DIF over LLC, but this shim DIF will make use of Ethernet directly.

A direct implication of only supporting one flow per peer is that just one (N+1) IPCP may make use of an underlying Ethernet shim IPCP at a time. This is because there is no way to distinguish between several end-points when an incoming frame only supplies source and destination MAC addresses.

MAC dst	MAC src	802.1Q tag	Ethertype	Payload	FCS
6 bytes	6 bytes	4 bytes	2 bytes	42–1500 bytes	4 bytes

Table 3.5: Ethernet II frame header format

Supporting only one application also directly ties in with the directory mechanism, as the shim DIF will utilise the Address Resolution Protocol (ARP)[1] in request/response mode to manage address translation. ARP binds the application name of the overlying IPCP to the MAC address of the Ethernet interface that the shim IPCP works on top of. This mapping of application name to MAC address is essentially the same as the directory functionality already supplied in RINA through the DA, wherein an upper application name is mapped to a lower IPCP application name.

For an application to be discoverable, it needs to explicitly register with the Ethernet shim IPCP. The binding of an application name to a MAC address happens during application registration, and when no application is registered the shim IPCP will discard all incoming SDUs. The application name is registered as a static entry with ARP on registration, and as such no ARP responses will be sent if no application is registered.

Since communication over the shim DIF only spans one Ethernet segment, it is only possible to communicate with directly connected nodes — where "directly" in this sense may encapsulate switches or bridges. Switches may allow many shim IPCPs to be part of the same segment, and subsequently the same DIF.

It should be noted that this design specification is minimal and does not describe how to secure the network. Security is a very central part of RINA, but due to the shim layer only providing the smallest possible DIF API on top of Ethernet, other layers need to enforce security mechanisms. The Ethernet protocol family provides very little in terms of security[15], but if PDU encryption is enforced by the upper DIF this does not result in any significant security risk.

3.2.1 Ethernet frame header

This is an explanation of the fields of an Ethernet frame in terms of RINA[12]:

[1]ARP is traditionally used for address translation between IPv4 and MAC addresses in IPv4 networks.

- **Destination MAC address:** The MAC address assigned to the Ethernet interface that the destination shim IPCP is bound to.
- **Source MAC address:** The MAC address assigned to the local Ethernet interface that the source IPCP is bound to.
- **802.1Q tag:** The DIF name.
- **Ethertype:** While not explicitly necessary for the operation of the shim DIF as all the traffic in the VLAN is assumed to be shim DIF traffic, `0xD1F0` is supplied as the type for Ethernet shim DIF traffic.
- **Payload:** The SDUs of the upper DIF are carried here. It is imperative that the MTU is enforced by the upper DIF, as no fragmentation or reassembly functions are performed by the Ethernet shim DIF. The MTU is normally 1500 bytes, but may be larger if jumbo frames are used.
- **FCS:** This acts as it normally would, and works as one of the only control mechanisms in the shim DIF since invalid frames are discarded.

3.2.2 IPCP API in the Ethernet shim DIF

The core API provided by the shim DIF should be identical to a normal DIF. RINA supports a wide range of mechanisms that can be configured, and a normal IPCP can supply the same mechanisms as the Ethernet shim IPCP if required. The shim IPCP can be seen as a specialised form of a normal IPCP. The core API consists of these five functions[16]:

`allocateRequest(destination, source, QoS, port-id)` $\Rightarrow$ `reason`

- When issued by a registered application on the shim IPCP, this will attempt to resolve an address. When the address resolution either fails or completes, an `allocateResponse` is issued with the appropriate result, resulting in either failed or successful flow allocation.
- When issued by the shim IPCP on the registered application, flow allocation happens like it normally would. This happens whenever an Ethernet frame arrives from a new shim IPCP.

`allocateResponse(destination, QoS, port-id)` $\Rightarrow$ `reason`

- When issued by the shim IPCP, this signifies that address resolution is completed and has either succeeded or failed, and the application responds accordingly.
- When issued by the registered application, a positive response will result in the shim IPCP passing waiting SDUs, while a negative one will discard all waiting SDUs.

`deallocate(port-id)` ⇒ `reason`

- When issued by the registered application, this will remove all state associated with the flow in the shim IPCP.
- When issued by the shim IPCP to the registered application[2], the flow will be deallocated normally.

`send(port-id, buffer)` ⇒ `reason`

- When issued by the registered application, an SDU will be passed to the shim IPCP and the shim IPCP will wrap the SDU in an Ethernet frame and pass it to the Network Interface Card (NIC).
- When issued by the shim IPCP, an SDU is passed on to the application.

`receive(port-id, buffer)` ⇒ `reason`

- When issued by the registered application, a waiting SDU will be passed to the application by the shim IPCP.
- When issued by the shim IPCP, a waiting PDU will be passed to shim IPCP by the application.

The Ethernet shim IPCP is also required to supply some registration mechanism to make an application discoverable through the ARP directory mechanism. This could be added as the function `registerApplication(name)`, which returns false if an application is already is registered, and true if not, where it subsequently adds a static entry to the ARP cache that are used as part of ARP responses and requests. This also requires the upper IPCP to have some awareness of whether the underlying IPCP is specifically an Ethernet shim IPCP.

The next subsection will describe these API calls in terms of the state machine supplied in fig. 3.6 to make the transitions clearer.

3.2.3 State diagram

There are four states with corresponding transitions that need to be implemented as seen in fig. 3.6: `NULL`, `INITIATOR ALLOCATE PENDING`, `RECIPIENT ALLOCATE PENDING`, and `ALLOCATED`. A port ID that corresponds with a flow will also have an associated state variable, which determines the outcome of invoked actions. This state diagram is based on the one found in Vrijders, Trouva, Day, *et al.* [12], but has some added clarifications. Note that while `deallocate` is omitted from the state diagram, the primitive may be invoked in any state aside from `NULL`. In the state diagram, a `submit` suffix denotes any primitive invoked *on* the shim IPCP.

[2]This may happen when an ARP request fails after attempting address resolution after the flow is allocated

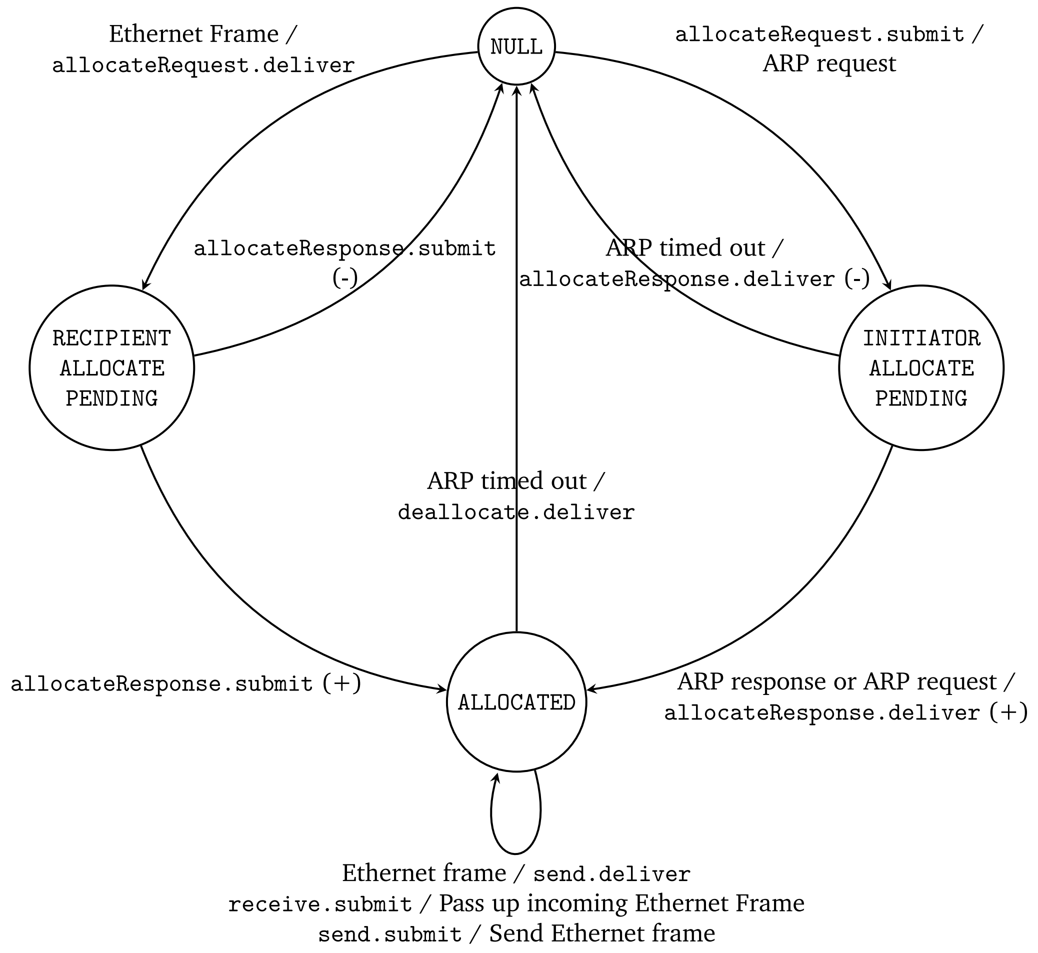

NULL
RECIPIENT ALLOCATE PENDING
INITIATOR ALLOCATE PENDING
ALLOCATED
Ethernet Frame / allocateRequest.deliver
allocateRequest.submit / ARP request
allocateResponse.submit (-)
ARP timed out / allocateResponse.deliver (-)
ARP timed out / deallocate.deliver
allocateResponse.submit (+)
ARP response or ARP request / allocateResponse.deliver (+)
Ethernet frame / send.deliver
receive.submit / Pass up incoming Ethernet Frame
send.submit / Send Ethernet frame

Figure 3.6: Ethernet shim DIF state diagram

When any port transitions to the `NULL` state, all corresponding data including queues and their contents are removed.

The following elements are specifications of the state transitions part of the state diagram from fig. 3.6.

`allocateRequest(namingInfo).submit`

This primitive is called by a registered source application on the shim IPCP to request a new flow. It requires the port to be in the `NULL` state, otherwise a negative `allocateResponse.deliver` primitive will be returned.

If a port ID for a flow already exists with the given destination application and is in the `ALLOCATED` or `INITIATOR ALLOCATE PENDING`, a negative `allocateResponse.deliver` is returned. Otherwise, the shim IPCP asks ARP to resolve the destination `namingInfo`. If an entry exists in the ARP table, the port ID transitions to `ALLOCATED` state, and data transfer may begin. If no entry is found, ARP sends an ARP request, and the port ID transitions to the `INITIATOR ALLOCATE PENDING`.

`allocateResponse(reason).submit`

This primitive is called by a destination application in response to an `allocateRequest(namingInfo).deliver` from a shim IPCP.

If the response is positive, a flow has been established in the upper IPCP, and the port ID transitions from `RECIPIENT ALLOCATE PENDING` to the `ALLOCATED` state. If the response is negative, flow allocation has failed, as indicated by the `reason` parameter.

ARP request

This action is emitted from the ARP protocol machine when address resolution has been attempted but no corresponding mapping was found.

If a port ID is in the `INITIATOR ALLOCATE PENDING` state and an ARP request arrives from the corresponding destination application, the state transitions to `ALLOCATED`.

ARP response

This action is emitted by the ARP protocol machine after an `ARPRequest` has been received with a destination application that matches the application that is registered to this shim IPCP.

If a port ID corresponding to the source application is in the `INITIATOR ALLOCATE PENDING` state, ARP populates the directory with a new entry, mapping the source application name to the source MAC address. The port ID then transitions to the `ALLOCATED` state, and a positive `allocateResponse.deliver` is delivered to the application.

ARP timed out

This action is emitted by the ARP protocol machine after no ARP response has been received.

If the port ID is in the `INITIATOR ALLOCATE PENDING` state, a negative `allocateResponse.deliver` is returned, and the port ID transitions to the `NULL` state. If the port ID is in the `ALLOCATED` state, a `deallocate.deliver` primitive is emitted to the registered IPCP, and the port ID transitions to the `NULL` state.

Ethernet frame

An Ethernet frame is sent by another shim IPCP, holding an SDU a remote application.

If an Ethernet frame is received from a MAC address that does not correspond to any port ID, the packet is queued and a port ID is allocated and set to the `RECIPIENT ALLOCATE PENDING` state. An allocation request is then passed to the registered IPCP. If the allocation response is positive, the port is set to the `ALLOCATED` state and queued packets are passed onwards to the application. If the allocation response is negative, the port is deallocated and relevant data discarded.

`receive.submit`

This primitive is invoked when an application wants to receive an SDU.

A `receive.submit` request can be called when the port ID is in the `ALLOCATED` state, otherwise it will be discarded[3]. Each frame only transports *one* SDU, as the IPCP does not provide mechanisms like concatenation and fragmentation.

`send.submit`

This primitive is invoked when an application wants to send one or more PDUs.

A `send.submit` request can be called when the port ID is in the `ALLOCATED` state. The shim IPCP will create an Ethernet frame and pass it to the NIC. An SDU must not exceed the MTU of the bound NIC, as the shim IPCP does not fragment outgoing SDUs.

`deallocate.submit`

This primitive is invoked whenever an application needs to deallocate a flow. This discards the state associated with the port ID in the local shim IPCP, and sets it to the `NULL` state. It may be invoked when the port ID is in any state apart from `NULL`.

Note that the shim IPCP has no way of passing a deallocation request to a remote shim IPCP, and any Ethernet frames arriving from the

[3]Note that a read primitive invocation should never happen in a normal IPCP when flow allocation is not completed for a given flow. The same principle goes for the write primitive.

same remote shim IPCP that had its associated state discarded will trigger a new `allocateRequest.submit` from the local shim IPCP to the application. It is therefore the responsibility of both the source and destination applications to call the deallocation primitive concurrently when required.

3.2.4 Notes on state machine

The original design document for the shim IPCP over Ethernet[12] specifies that when a port ID receives a negative `allocateResponse.submit`, all future frames from the remote application will be ignored until the remote flow is deallocated. Since there is no way to notify another shim IPCP that deallocation has occurred, this is impossible, and any new incoming Ethernet frames will reinitiate the flow allocation procedure. For this reason, it has been omitted from the state transition description.

Chapter 4

Simulation framework and model libraries

4.1 OMNeT++

OMNeT++ is an open-source discrete event simulation framework written in C++ that supports accurately modelling networking scenarios and protocols. Since everything in discrete event simulation happens as a sequence of events, networking code can be reduced to its bare essentials, unhindered by complex processing systems' requirements for critical sections and hardware limitations. This makes it ideal for accurately modelling proposed experimental protocols, and discovering edge cases.

A vast array of features is provided by OMNeT++. Its core functionality is the modularisation of components, where each module has the possibility to create gates that can be bound to other modules' gates. When bound, the modules are able to send any form of data directly through objects with the classes `cMessage` or `cPacket`. While they can be directly defined through creating classes that inherit from `cMessage` or `cPacket`, the preferred method is to use a specialised syntax for message definitions that automatically generate C++ code compiled from `.msg` files with the tool `opp_msgtool`. These message types can also be further customised in C++ source files by explicitly defining a `@customize` tag.

OMNeT++ allows developers to create modules that are connected through gates, that can send each other packets, similar to a real network. This means that one could have various protocol machines, each responsible for one protocol. In RINASim, each of the major components (as seen in fig. 2.2 for an IPCP) have their own module or set of modules that communicate with each other through sending messages, direct method calls or signals. The interactions between the modules can be visualised in the `Qtenv` tool. The simulation is processed as a set of events in discrete time, where each event is the transmission of a message from one module to another using the `send()` method.

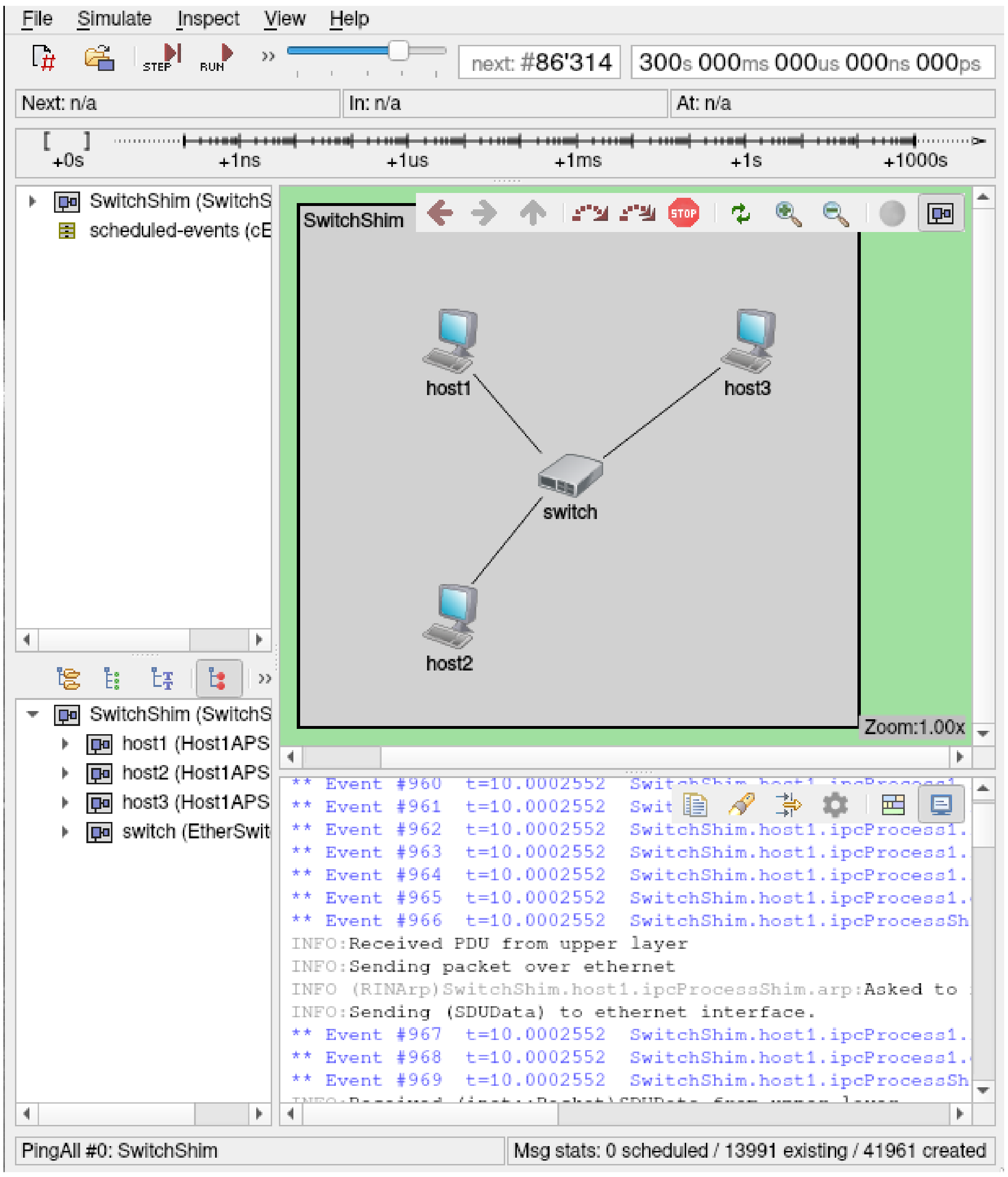
The view supplied by `Qtenv` has many fields that let a user monitor the state of a simulation, as seen in fig. 4.1. The view consists of an event log, a view that contains the parameters of a module, the simulation view, and a toolbar with various buttons to control the simulation. Attributes in the C++ source code can be added to the parameter view by utilising the `WATCH` family of macros, with `WATCH_MAP` being utilised to view the contents of maps. The simulation view shows packets being transmitted when an event occurs, other interactions between modules, and changes to the topology during simulation runtime.

For setting up modules and their connections, a specialised language called Network Description (NED) is used. Modules defined in NED files can be compound modules or simple modules, and each module can have a set of parameters, gates, submodules, and definitions of gate connections. While defined in NED files, their interaction needs to be defined in C++ source files. Compound modules may consist of any numbers of submodules, which in turn may be either simple or compound modules. A compound module does not need to be defined in C++ sources, as packets may be passed directly to submodules for processing.

The main form of communication between modules, and what earns the simulation framework the status of being a simulator, is the `send()` function, which takes a gate and an object inherited from `cMessage`. Modules implement the `handleMessage()` function to control what happens to incoming messages, and these messages may contain any form of information.

Channels may also be explicitly defined between gates to allow specific parameters to be set for connections. There are three basic types of channels that can be extended, where parameters for delay, data rate and bit error rate may be customised. This way, unreliable links may be modelled, which allows the simulation of protocols that supply some form of reliability mechanisms, such as TCP.

OMNeT++ also supplies a signalling mechanism, where modules can subscribe to and emit signals. Signals are declared in NED files using the `@signal` tag, and emitted through the use of the `emit` function call in C++ source files. They are used for collecting statistics, publish/subscribe style communication between modules, and more[17]. Signals can be defined as



```
** Event #960  t=10.0002552  SwitchShim.host1.ipcProcess1.
** Event #961  t=10.0002552  Swit
** Event #962  t=10.0002552  SwitchShim.host1.ipcProcess1.
** Event #963  t=10.0002552  SwitchShim.host1.ipcProcess1.
** Event #964  t=10.0002552  SwitchShim.host1.ipcProcess1.
** Event #965  t=10.0002552  SwitchShim.host1.ipcProcess1.
** Event #966  t=10.0002552  SwitchShim.host1.ipcProcessSh
INFO:Received PDU from upper layer
INFO:Sending packet over ethernet
INFO (RINArp)SwitchShim.host1.ipcProcessShim.arp:Asked to
INFO:Sending (SDUData) to ethernet interface.
** Event #967  t=10.0002552  SwitchShim.host1.ipcProcess1.
** Event #968  t=10.0002552  SwitchShim.host1.ipcProcess1.
** Event #969  t=10.0002552  SwitchShim.host1.ipcProcessSh
```

Figure 4.1: The Qtenv simulation environment

statistics by using the `@statistic` tag in NED files, which allows features such as controllable detail level, aggregation, and more. The behaviour of signals in OMNeT++ is very similar to the behaviour of function calls aside from the publish/subscribe, as a signal emission results in immediate execution of code in the subscribed modules.

4.2 RINASim

RINASim is an open-source[18] simulation model library for OMNeT++ that implements the components of RINA. It has two main goals: to allow researchers to prototype new policies, and to allow anyone with an interest in the concepts of RINA to visualise and understand them. It began as a deliverable of the FP7 EU Pristine project, which aims to implement RINA on various platforms, and to demonstrate the benefits that may be achieved compared to the traditional TCP/IP stack.

RINASim has a significant amount of user-configurable policies included as part of the project. There are many example network configurations making use of various policies, and several different CS models that can be used to easily set up networks. Separation of policy and mechanism is achieved through inheritance, where policy modules have a set of fixed API functions that are called by the fixed parts of an IPCP, allowing the user to specify which policies to use when configuring the network. The core set of mechanisms are defined in the "core" library, which is maintained in the `src` subdirectory. During the linking phase, the policies — stored in the `policies` subdirectory — are linked together with the core library to create the `librinasim` library.

Many function calls between modules in RINASim are made through the use of signal mechanism provided in OMNeT++, including the `allocateResponse` primitive. Since several modules may be subscribed to one signal, a signal invocation may also be useful to notify several modules.

Names in RINASim must be statically defined as part of the network configuration. The `DIFAllocator` module[1] must also explicitly be provided with information about which underlying IPCPs should be used.

RINASim attempts to model RINA closely, and the core functionality is partitioned into the two main concepts: the DIF and the DAF. The various interacting submodules of IPCP and APs are laid out in a way that allows the user to see which parts interact with each other in a clear way during a

[1]Which is the representation of the DA in RINASim.

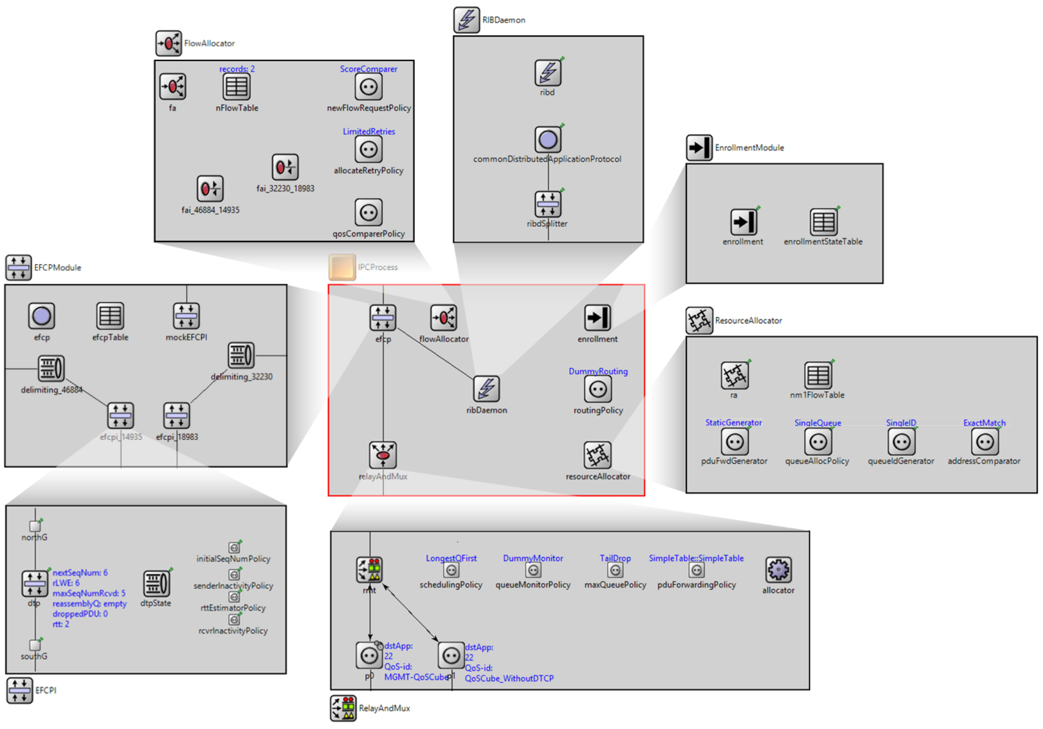


Figure 4.2: The components of an `IPCProcess` in RINASim[20]

simulation run. Since only the DIF implementation provided is relevant to the scope of this thesis, the implementation of the DAF will not be covered here. The following sections are based on the information found in Veselý, Marek, and Jeřábek [19], section 5.4. The figures are taken directly from `Qtenv` when running simulations in RINASim.

4.2.1 Distributed IPC Facility in RINASim

The `IPCProcess` module in RINASim is a compound module that consists of a set of submodules: `EFCPModule`, `RelayAndMux`, `FlowAllocator`, `ResourceAllocator`, `RIBDaemon`, and `Enrollment`. These submodules in turn have their own set of submodules that represents different functionality in RINA. The modules are laid out in a layout that provides clear visualisation of their interaction, similar to the ordering of components in fig. 2.2, with data transfer components on the left-hand side and layer management components on the right. All of the components of the `IPCProcess` module can be seen in fig. 4.2.

None of the API calls of the DIF are called explicitly on the `IPCProcess`, but instead on the relevant submodules. The primitive `allocateRequest` is represented as the function `receiveAllocateRequest()` in the `FlowAllocator`

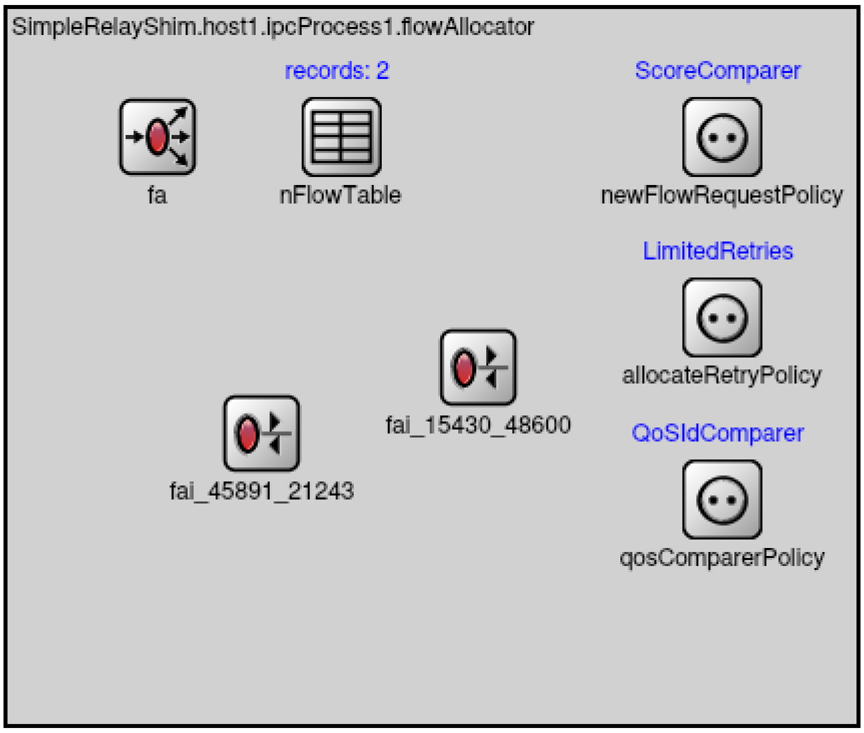


Figure 4.3: The `FlowAllocator` module, with two `FAIs`

module, while the `allocateResponse` and `deallocate` primitives are signals and caught by the targeted `IPCProcess` module in a CS. The `send` primitive is encapsulated by the `send()` function that is part of OMNeT++, while the `receive` primitive is implemented by specialising the `handleMessage()` function inherited from the `cModule` type.

The `IPCProcess` module allows a user to specify the QoS parameters that are wanted for flows, which will also dictate which mechanisms are enabled in the submodules. This is supplied as an argument for the `ResourceAllocator` on network configuration.

Flow Allocator

The `FlowAllocator` module seen in fig. 4.3 is a representation of the mechanisms supplied by the FA. It is responsible for the allocation and deallocation primitives that are part of the IPC API of the DIF. When flow allocation or deallocation is requested, the `FlowAllocator` handles the request. It then initiates the `cace` phase through the `Enrollment` module if not already enrolled with a neighbour on the path to the target CS. If a management flow already exists, it proceeds to flow allocation, where an `FAI` is created that will manage the flow through its lifetime.

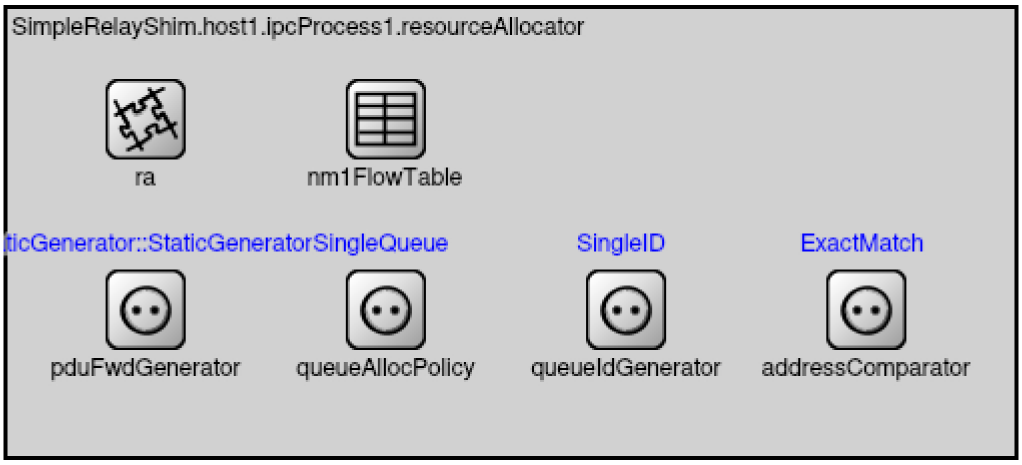


Figure 4.4: The `ResourceAllocator` module

The `FAI` module is responsible for creating gates and binding them, creating EFCPI, and handling deallocation. An `FAI` will listen to a set of signals associated with the `allocateResponse` primitive, which are emitted by a `FAI` submodule of an underlying `FlowAllocator` upon either successful or failed allocation.

Resource Allocator

The `ResourceAllocator` module seen in fig. 4.4 manages allocation of underlying flows, and starts the CACE phase if an underlying management flow has been successfully allocated. It maintains information about underlying flows in a table called the `NM1FlowTable`. Whenever a flow is allocated by the `FlowAllocator` it will check the `NM1FlowTable` for a flow that matches the QoS requirements in an underlying DIF that can be used to reach a remote `IPCProcess`. It is also responsible for notifying various other components about changes in network capabilities, for instance notifying the `RelayAndMux` module about changes in the PDU forwarding policy.

It also has responsibility for activating the required policies for flows based on which QoS cube is in use, and also keeps track of the QoS cubes that the `IPCProcess supports.`

Error and Flow Control Protocol

The `EFCPModule` module seen in fig. 4.5 manages created `EFCPIs`, which create PDUs for transmission. An `EFCPI` submodule contains the `DTP` and `DTPState` submodules, where `DTP` is responsible for constructing PDUs, and

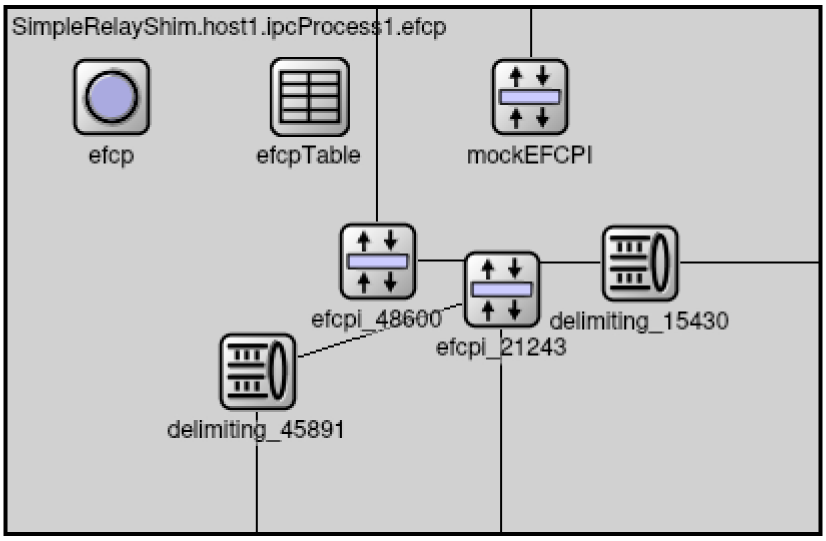


Figure 4.5: The EFCPModule module

DTPState contains the state associated with the connection. EFCPIs can have varying requirements for state. These requirements are defined when an application passes its QoS requirements, which will control whether a DTCP module is created. The DTP module will ask the DTCP module to perform retransmission and flow control if required. The DTPState contains information that must be shared between the DTP and DTCP modules.

On creation of an EFCPI, a Delimiting module instance is also created. The Delimiting module is responsible for fragmentation and encapsulation, and produces UserDataField packets from SDUs from the upper layer, which is passed on to an associated EFCPI. Similarly, a EFCPI module passes UserDataField packets to its associated Delimiting module for concatenation[2]. To determine if an incoming SDU from an upper IPCProcess has to be fragmented, the maxSDUsize parameter of the EFCPModule is used.

The EFCPModule module also contains a module MockEFCPI, which handles incoming management requests from remote systems directed to the containing IPCProcess. MockEFCPI will pass on received messages to the RIBDaemon module. This module is required to allow the IPCProcess to receive CDAP management messages.

Enrollment

The Enrollment module seen in fig. 4.6 handles the initial part of communications between two IPCProcess modules, which means it starts the CACE

[2]Concatenation will only be enforced if required by the associated QoS cube.

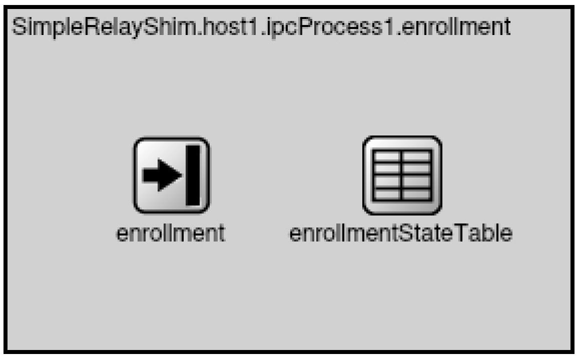


Figure 4.6: The `Enrollment` module

phase. It utilises a table to hold information about various neighbouring `IPCProcess` it is connected to, which is called the `EnrollmentStateTable`. The `Enrollment` module constructs CDAP messages, which it passes on to the `CommonDistributedApplicationProtocol` module for sending, which is part of the `RIBDaemon` module.

RIB Daemon

The `RIBDaemon` module seen in fig. 4.7 handles all management requests, and notifies relevant modules about incoming management requests that concern them through a set of "notifiers". Incoming messages are parsed and forwarded to the relevant notifier, e.g. flow creation requests/responses to the `FANotifier`, or connection and enrollment requests/responses to the `EnrollmentNotifier`.

Relaying and Multiplexing task

The `RelayAndMux` module seen in fig. 4.8 is a representation of RMT, and its main task is to either relay PDUs to modules within the current `IPCProcess`, or to pass them to the correct outgoing port. An incoming PDU from a port to an (N-1) DIF will be subject to one of the following actions:

- If the address is not the address of this IPCP, it sends the PDU according to the PDU Forwarding policy that is in use. This also goes for outgoing PDUs originating from this IPCP.
- If the address is the same as the address of the containing IPCP, the PDU is passed to the EFCPI module that corresponds with the (N+1) port in the PDU header.

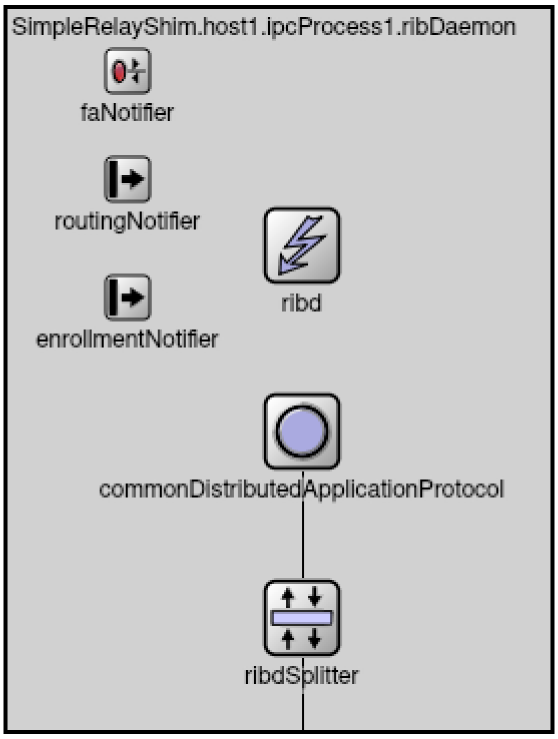

SimpleRelayShim.host1.ipcProcess1.ribDaemon
faNotifier
routingNotifier
ribd
enrollmentNotifier
commonDistributedApplicationProtocol
ribdSplitter

Figure 4.7: The `RIBDaemon` module

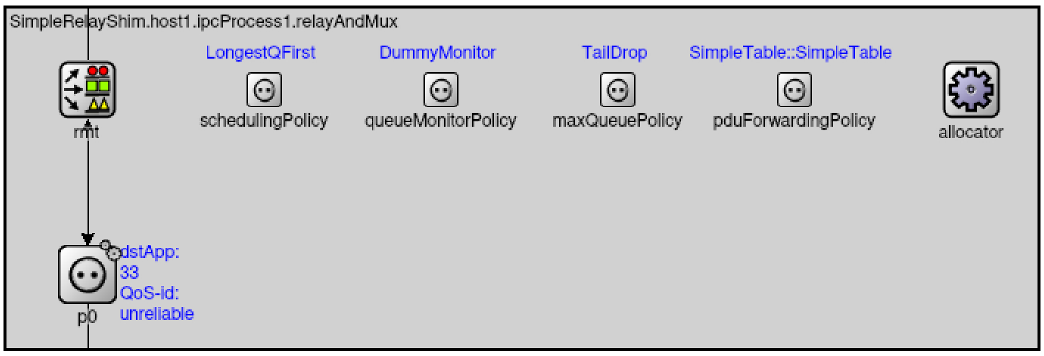


Figure 4.8: The `RelayAndMux` module

4.3 INET

INET is an open-source model library that utilises OMNeT++. It aims to supply a realistic simulation environment for the TCP/IP stack[21]. INET supplies a set of models for TCP, UDP, IPv4, Internet Protocol version 6 (IPv6), and more. It also supports wired and wireless link layer protocols such as Ethernet, Point-to-Point Protocol (PPP), IEEE 802.11[3], and more. There are also several simulation model libraries that utilise models from INET, where SimuLTE is one example for simulation of Long-Term Evolution (LTE) networks.

INET supplies extensions on top of the packet API of OMNeT++, adding the use of "chunks", which essentially represent header fields to more easily manage construction, duplication, fragmentation and serialisation of packets[22]. These packets are sent ordinarily through the `send()` function from OMNeT++, but can also easily be connected with real interfaces to achieve Hardware-In-the-Loop (HIL) simulation, which can be useful for validating simulation models. This only works when all the elements of a chunk supply their own serialisation functions.

A central part of this packet API is the addition of packet tags, which is metadata that protocol machine modules in INET can utilise to add specific information to the headers they supply. For instance, a `MacAddressReq` tag must be added to supply information about a destination MAC address before passing a packet to a NIC module.

4.3.1 Integration into RINASim

Initially the Ethernet shim DIF implementation was only meant to be integrated with INET version 3.6.7, which uses the original packet API of OMNeT++. This implementation worked, but as VLANs were not supported as part of this INET version, it was not feature-complete according to the specifications defined in section 3.2. The reason that this version was picked first, was that the new packet API seemed to be very difficult to integrate with RINASim because packets needed to be compatible with the new `Chunk` types, which the packets that were part of RINASim were not. The `cPacketChunk` was eventually discovered as part of the newer INET version 4 through its documentation, which allows a normal `cPacket` to be wrapped in a `Chunk` packet type. This ended up being used for `SDUData` packets passed to the Ethernet shim IPCP in RINASim.

[3]The standard for wireless interfaces.

There was also a choice of whether the INET modules that were needed should instead be migrated to RINASim. Both including INET and fully migrating the modules has their own set of advantages and challenges. Migration requires additional development work, but could likely make it easier to integrate RINA with the self-made Ethernet services, and binary sizes and compilation times remain smaller than with the alternative. Including the INET framework makes it so RINASim is no longer a stand-alone framework, and additional work may be needed to fit RINASim modules onto the API of INET modules. In terms of functionality however, it is preferable to integrate RINASim with INET, since INET is continually being developed to support an exhaustive list of protocols that are in use in real TCP/IP networks. With full integration between the libraries, advanced simulation scenarios could be constructed that more accurately demonstrate the challenges with adopting RINA on existing infrastructure.

The modules that are used from INET in the implementation of this thesis are the following:

`EthernetInterface`
: This module represents an Ethernet interface. It may be represented as Ethernet II, or any of the main IEEE 802.3 standards. It provides a simple interface that acts identically to an Ethernet NIC, including discarding packets if carried FCS is different from computed FCS. An instance of this module with the IEEE 802.1Q standard enabled through the use of the `qEncap` module can be seen in fig. 4.9.

`EtherSwitch`
: This module is a normal Ethernet switch, allowing multiple computing systems to communicate over the same Ethernet segment.

`MessageDispatcher`
: This module relays packets to the correct module based on protocol information, which is controlled through a set of static functions `registerService()` and `registerProtocol()` that need to be called by modules on network initialisation. Protocols are registered through these calls by corresponding modules, and the `MessageDispatcher` passes them based on target output and input gates.

There are also several classes from INET that are used in various parts of the implementation that will be defined in the next chapter. Among the most relevant ones are:

`MacAddress`
: The representation of a MAC address from INET. Supplies a set of static instantiations `UNSPECIFIED_ADDRESS` and `BROADCAST_ADDRESS`.

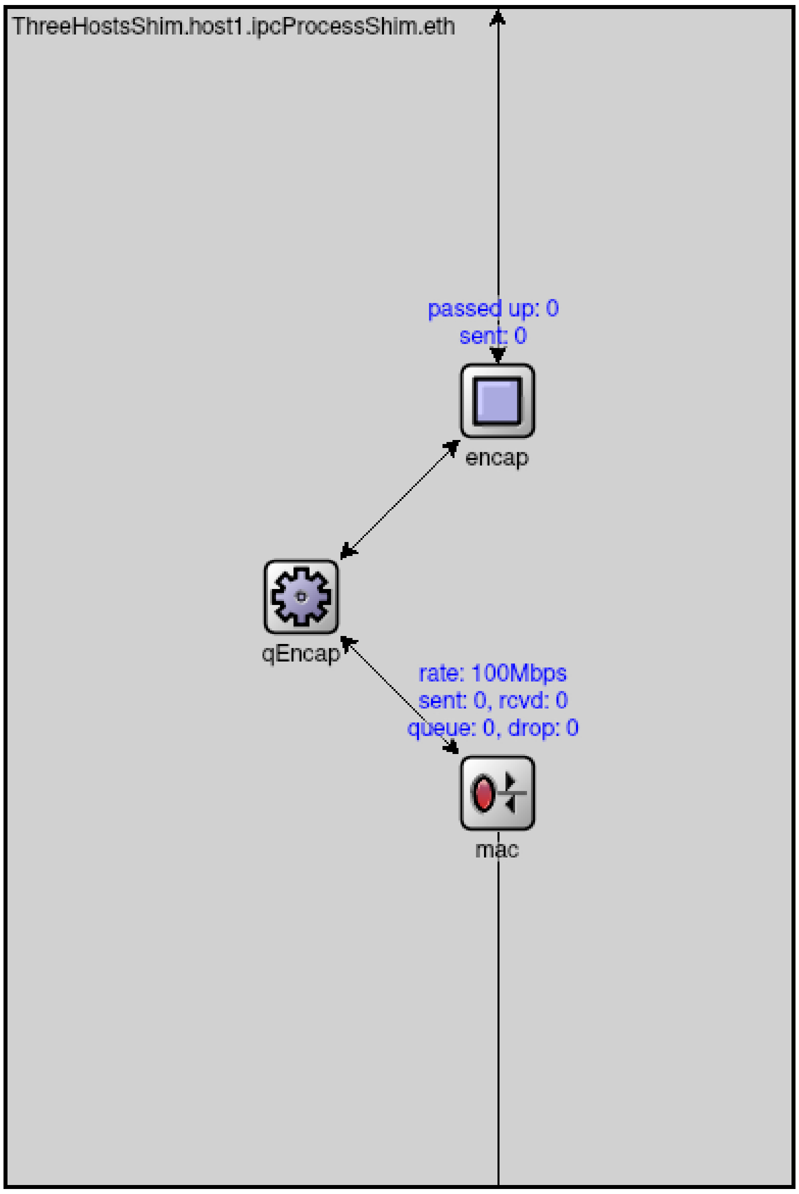

ThreeHostsShim.host1.ipcProcessShim.eth
passed up: 0
sent: 0
encap
qEncap
rate: 100Mbps
sent: 0, rcvd: 0
queue: 0, drop: 0
mac

Figure 4.9

`VlanReq`
A type of packet tag that supplies information about which VLAN ID should be used for a VLAN tag. This is supplied before passing a packet to an Ethernet NIC.

`MacAddressReq`
Another type of packet tag, which allows the addition of source and destination MAC addresses. This must be supplied for packets before they are sent to a NIC.

`PacketProtocolTag`
This needs to be added for protocols that have protocol information fields in their header that supplies information about the protocol used from the upper layer. For protocols utilising Ethernet for instance, this tag must be set to a valid member of `ProtocolGroup::ethertype`.

`ProtocolGroup::ethertype`
An Ethernet NIC will check whether a packet passed from an upper layer contains a valid ethertype, and will drop packets if they do not contain an ethertype that corresponds to this group. Any module may register new protocols as part of `ProtocolGroup::ethertype`, which is relevant for RINASim with the ethertype `0xD1F0`, as defined in the Ethernet shim IPCP design chapter.

Chapter 5

Implementation of the Ethernet shim DIF

This chapter describes the implementation of the Ethernet shim DIF, including changes that were made to RINASim during the implementation process. First, the interactions between the components of the Ethernet shim IPCP will be explained, and then each module will be explained separately, including the public API that the modules supply.

In this chapter the (N+1) application is referred to as the “registered application” for convenience, as only one application may be registered to the shim IPCP.

5.1 Required changes to RINASim

While most of the functionality is generic enough to facilitate the creation of different IPCPs with a unified API, some changes were made to flow allocation to allow several management flows to coexist. This section will clarify why these changes are necessary.

5.1.1 Testing

A testing facility was added to provide regression testing for all working examples already provided in RINASim. With the OMNeT++ simulation fingerprint mechanism, making sure that the examples have the same sequence of events through each execution becomes trivial. The caveat of exclusively utilising this method, is that since fingerprints are expected to be the same every time, the seed for the random number generators that OMNeT++ uses need to be set, which could result in testing being non-exhaustive. This can be mitigated to some extent by seeding the random number generator that OMNeT++ uses with the run number, but this also results in many more configuration runs. In this process, a few issues that were present in RINASim at the start of development that invalidated some examples were also fixed.

The fingerprints were compared by a test script developed for INET, where a large test-suite is included. This test script takes a CSV file as input, where each element contains a configuration, additional flags to be passed to `opp_run`, and a range of expected fingerprints. Through this method, any adverse effects from changes within the existing codebase could be detected easily, which significantly eased development.

5.1.2 Required changes

When flow allocation is requested for an IPCP, the DA is first polled to see which IPCP in the current or prospective DIF can be used to reach the requested application. If a remote IPCP is successfully returned, the originating IPCP will attempt to find which immediate neighbour can be used to reach the remote IPCP, and the enrollment status with the neighbouring IPCP is also checked. If they are not connected, the enrollment procedure is begun and the flow allocation is postponed, otherwise the flow allocation with the target IPCP is started.

RINASim does it slightly differently, as the flow allocation procedure does not check if the originating `IPCProcess` is connected to a neighbouring `IPCProcess`, but only checks if it is enrolled to *anything*. This was likely done as a shortcut to skip the CACE phase being initiated for intermediary hosts in large network configurations. However, it carries with it some consequences in regards to compliance, as the CACE phase will not be initiated between the originating `IPCProcess` and the neighbouring `IPCProcess` if the originating `IPCProcess` is enrolled from before.

The change that was introduced makes it so the flow allocation procedure now consists of checking whether an `IPCProcess` is enrolled to a neighbouring `IPCProcess` instead of just checking if it is at all enrolled in a DIF. This increases the simulation run-times for the configurations that heavily rely on the self-enrollment shortcut which introduces some extra noise, but compliance should in this case be more important to allow a greater degree of extensibility for additional improvements to the implementation in the future. There were also some inconsistencies in the handling of the forwarding of flows in comparison to normal flow allocation, as management flows were not allocated when required, meaning that an `IPCProcess` that did flow forwarding was assumed to be enrolled. The relevant changes were added to `src/DIF/FA/FA.cc`.

An additional change that was considered for this project, was to introduce an alternative API more similar to the RINA specifications that would have made flow objects exclusive to `IPCProcess` modules. The motivation with this was to make the `IPCProcess` module easier to extend, as well as making the flow allocation procedure easier to understand. In RINASim, an upper `IPCProcess` is able to make direct changes to the flow object without explicitly going through the API of the lower `IPCProcess`, which makes it harder to detect edge-cases and extend upon the protocol since changes may be made to the flow objects at unexpected points. The `ResourceAllocator` module maintains a pointer to the flow objects in an “`NM1FlowTable`”, which it will also check against for compatible flows for any new flow allocation. Allowing the underlying `IPCProcess` to find a suitable flow instead could likely make the process of implementing other types of IPCPs like shim DIFs significantly easier.

5.2 Contents of a shim IPCP

There are three core components of the Ethernet shim IPCP that work independently: an ARP module, an Ethernet interface, and a specialised shim IPCP. The shim IPCP module is called `IPCProcessShim`, and consists of the modules `EthShim` and `ShimFlowAllocator`, denoted as `shim` and `flowAllocator` in fig. 5.1. `EthShim` is responsible for recording internal state, passing packets, and querying the `RINArp` module, denoted as `arp` in the aforementioned figure. `ShimFlowAllocator` is a specialised flow allocator module tasked with keeping information about flows through an `NFLowTable`, and the implementation for this module will be covered in section 5.3. The `RINArp` module and the NIC `EthernetInterface` work as their own entities, but are still encapsulated by the `IPCProcessShim`. A host running with the `IPCProcessShim` can be seen in fig. 5.2.

An earlier stage in the implementation had the Ethernet interface external to the `IPCProcessShim` module. Since Ethernet interfaces may only be bound to one process however, it was included as a part of `IPCProcessShim`. It also allows additional host configurations to be more easily set up since the `IPCProcessShim` operates very similarly to a normal `IPCProcess` in terms of the API. Additionally, for future simulation scenarios, a host should be

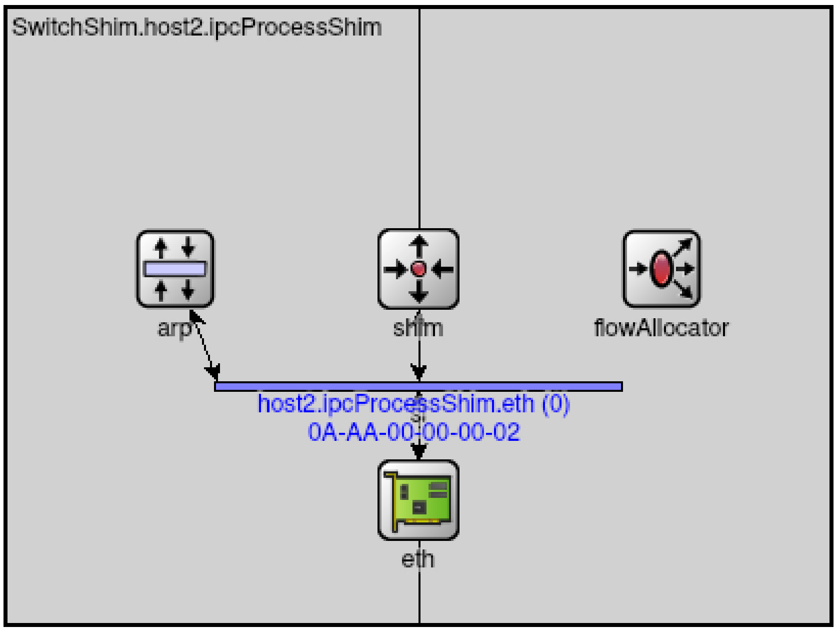


Figure 5.1: The Ethernet shim IPCP module

able to support both the RINA stack *and* the TCP/IP stack, and having the interface as an external module could lead to incompatibilities due to the `InterfaceTable` module that INET uses to keep track of the NICs of a CS. INET has a limitation where two NICs or classes inheriting from `InterfaceEntry` cannot use the same name, even if contained within completely different submodules.

The ARP module supplied in INET is too rigid and tightly coupled with IPv4, as it only allows the use of IPv4 addresses. Instead of reusing it, a new one is defined that is closer to the specifications, which allows variable length addresses[23]. The `RINArp` module could either be implemented as an external or internal module to the `IPCProcessShim`, but an internal module ended up being used because it could aid in visualisation and network configuration to a greater degree. The implementation of the `RINArp` module will be covered in section 5.5.

The DIF name is required to also be a valid VLAN ID, which means it must be able to be represented as a numerical value. Since DIF names must be defined as strings, the shim module must handle integer conversion to supply the VLAN ID for the IEEE 802.1Q tag added to outgoing SDUs.

A couple of extra parameters are set explicitly in the Ethernet Interface module `EthernetInterface` for network initialisation, as seen in fig. 5.3. The `qEncap` module is needed for adding a VLAN tag to the Ethernet frame, and will also discard packets depending on if the VLAN ID does not match the DIF name, or if there is no VLAN tag. The VLAN ID is set by the `EthShim`

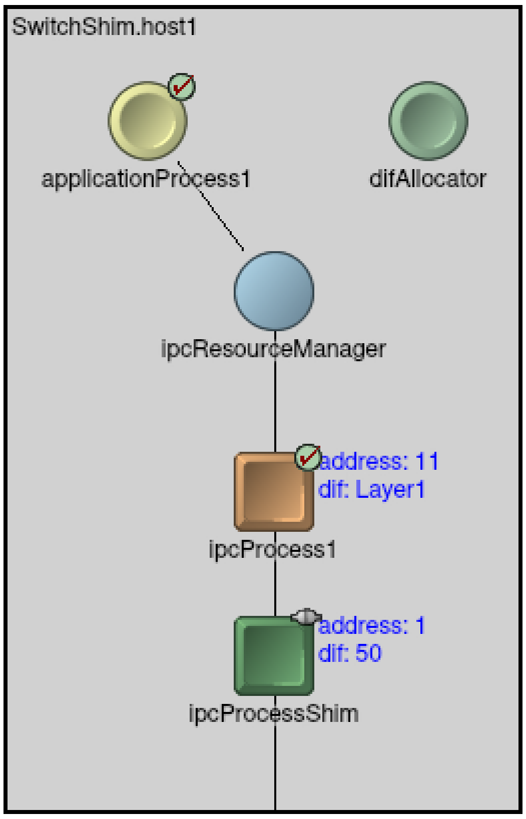


Figure 5.2: A host running over an Ethernet shim DIF, with one application process

```
eth: <default("EthernetInterface")> like IEthernetInterface {
    parameters:
        @display("p=201,240,row,60;q=txQueue");
        qEncap.typename = "Ieee8021qEncap";
        qEncap.inboundVlanIdFilter = difName;
        qEncap.outboundVlanIdFilter = difName;
        interfaceTableModule = "";
}
```

Figure 5.3: Additional parameters supplied to the `EthernetInterface` submodule.

```
**.switch.eth[*].qEncap.typename = "Ieee8021qEncap"
**.switch.eth[0].qEncap.inboundVlanFilter = "50"
**.switch.eth[0].qEncap.outboundVlanFilter = "50"
**.switch.eth[1].qEncap.inboundVlanFilter = "50"
**.switch.eth[1].qEncap.outboundVlanFilter = "50"
**.switch.eth[2].qEncap.inboundVlanFilter = "60"
**.switch.eth[2].qEncap.outboundVlanFilter = "60"
**.switch.eth[3].qEncap.inboundVlanFilter = "60"
**.switch.eth[3].qEncap.outboundVlanFilter = "60"
```

Figure 5.4: Configuration settings for VLAN-aware switch

```
enum class ConnectionState {
    null,
    initiatorAllocatePending,
    recipientAllocatePending,
    allocated
};
```

Figure 5.5: The enum containing the Ethernet shim IPCP port states

module when sending Ethernet frames by adding a `VlanReq` protocol tag to the frame. Switches and bridges do not have to be VLAN-aware, but it may be beneficial to explicitly set up VLAN filters for the switches in the configuration file as it aids visualisation since PDUs will only be forwarded by the interfaces that accept the provided VLAN IDs. The Ethernet interfaces of the switch can for instance be restricted to only accept traffic from specific VLANs like in fig. 5.4.

While a normal `IPCProcess` may have several input and output gates, the `IPCProcessShim` will only be associated to one physical link and one registered upper `IPCProcess`. Therefore, it is not necessary to declare the `southIo` and `northIo` gates as arrays.

The following sections will describe each of the central components implemented as part of this thesis. Each section has a description of the overall functionality, and then a specification of the public functions of each module and what they do. Note that any C++-specific syntax is omitted from the function parameters and return values for increased clarity. The internal states used for the ports as seen in fig. 5.5 will in these sections be prefixed by `::`.

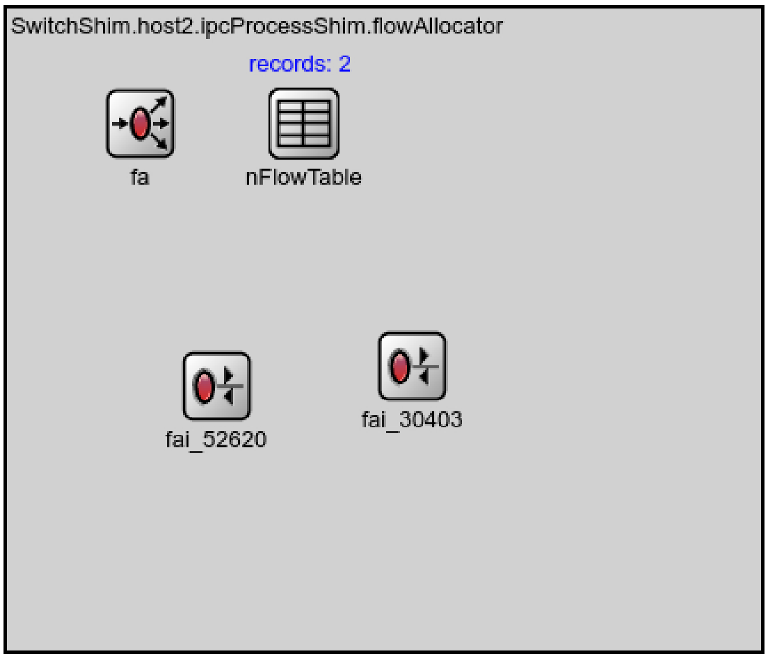


Figure 5.6: The ShimFlowAllocator module of the IPCProcessShim, with two ShimFAIs

5.3 ShimFA

As with an IPCProcess, the IPCProcessShim splits FAIs into their own modules to aid with visualising. The ShimFA module consists of an NFlowTable, and is also the container for ShimFAI modules, which manage the lifetime of their associated flows. The design philosophy by splitting the shim[1] and the FA, is that the shim handles the internal state machine for each port, while the FA handles the public facing allocation API.

ShimFA handles incoming and outgoing allocation requests, and is responsible for initiating the mechanism to create a ConnectionEntry in the EthShim module. ShimFA inherits from the generic FABase class, which is used to create a unified object API for different types of FAs. In RINASim, this is necessary since the IRM or an IPCProcess will directly access the flow allocator within an underlying IPCProcess. Flow allocation is requested by an upper IPCProcess by calling the function receiveAllocateRequest() from the ShimFA module in the IPCProcessShim, which then goes on to set up an ShimFAI and start address resolution through the EthShim module.

When allocation requests are received, the ShimFA will set the QoS cube identifier to "unreliable". The registered IPCProcess is required to accept a QoS cube with this name. For the time being, there has not been made any

[1]Where shim here means the part that directly communicates with Ethernet.

explicit enforcements for this, and any QoS requirements that are delivered to the `ShimFA` through the `receiveAllocateRequest()` function will be ignored. If a user wants to configure additional flow and retransmission control mechanisms, the registered `IPCProcess` needs to take care of this.

5.3.1 API

The API provided by the `ShimFA` module is analogous to the IPCP flow allocation and deallocation primitives that are part of the RINA specifications. The `allocateResponse` primitive is emitted as a signal that is caught by the registered upper `IPCProcess`. Similarly, the registered upper `IPCProcess` will emit allocate response signals when flow allocation is requested by the `IPCProcessShim`.

The states referenced here are defined in section 5.4, as part of the explanation of the `EthShim` module.

`receiveAllocateRequest(Flow flow)` ⇒ `bool`

Called by the registered `IPCProcess` when it wishes to allocate a flow. This function is also supplied by the normal `FA` module, and is analogous to the `allocateRequest` primitive when invoked from an (N+1) IPCP.

The `flow` parameter contains all necessary information for flow allocation, including the name of the target application and QoS requirements. The flow object is not copied, and is created by the registered `IPCProcess`. It will be utilised in the `NFlowTable`.

When this function is called, `ShimFA` will ask the `EthShim` module to create an entry through the `EthShim::createEntry()` function. If the function returns `CreateResult::error`, `receiveAllocateRequest()` returns `false`.

Otherwise, `CreateResult::pending` or `CreateResult::completed` is returned. In either case the flow allocation continues, and a `ShimFAI` is created to manage the lifetime of the flow. If `CreateResult::completed` was returned, it means that a matching ARP entry was found when the `EthShim` module attempted to resolve an address, and the `ShimFA` module will then pass the allocation request to the `ShimFAI` module, which will emit a positive `allocateResponse()` signal to the registered `IPCProcess`. The function then returns `true`.

`completedAddressResolution(APN dstApn)` ⇒ `void`

Called by the `EthShim` module after it has been notified of an

ARP response arriving, and the corresponding entry is in the `::initiatorAllocatePending` state.

This will find the `ShimFAI` corresponding to the supplied `dstApn` parameter, and call `receiveAllocateRequest()` on it. The `ShimFAI` will subsequently call `finalizeConnection()` on the `EthShim` module with the port ID that corresponds with the `ShimFAI`, and if successful it will emit a positive `allocateResponse` signal.

`failedAddressResolution(APN dstApn)` $\Rightarrow$ **`void`**
: Called by the `EthShim` module to notify the `ShimFA` that address resolution has failed. `ShimFA` will remove the associated `ShimFAI`, and subsequently ask to deallocate the flow from the registered `IPCProcess`.

`createUpperFlow(APN apn)` $\Rightarrow$ **`bool`**
: Called by the `EthShim` module when an Ethernet frame arrives from a new host. This function creates a new `ShimFAI`, and emits an `allocationRequest` signal to the registered application.

5.4 EthShim

The `EthShim` module works directly on top of the Ethernet interface. It is responsible for passing outgoing SDUs to the correct destination MAC address and incoming SDUs to the correct port, and keeping stock of the internal state of the ports. The entries are contained in a map, where destination application name is mapped to a `ConnectionEntry`. `EthShim` also handles all interactions with the `RINArp` module. It has no user-configurable fields as part of the NED file.

On network initialisation, this module subscribes to the `RINArp` signals `completedRINArpResolutionSignal` and `failedRINArpResolution Signal`. The DIF name is then parsed so it can be used in `VlanReq` tags that specify which VLAN is to be used. `0xD1F0` is also explicitly registered as an ethertype through the static `ProtocolGroup::ethertype` group.

Each port receives its own set of in- and out-queues, and gates corresponding to the registered application as seen in fig. 5.7. The gates `outGate` and `inGate` are used to send to and receive from the (N+1) port respectively. As for the queues, `outQueue` stores SDUs that will be sent to the network, for instance while waiting for address resolution from ARP, and `inQueue` stores SDUs that will be passed to an (N+1) port, for instance when waiting for upper `IPCProcess` to accept flow allocation from the IPCProcessShim.

On registration of an upper `IPCProcess`, the `IPCProcessShim` will need to

```
struct ConnectionEntry {
    ConnectionState state = ConnectionState::null;
    cGate *inGate = nullptr;
    cGate *outGate = nullptr;
    cPacketQueue outQueue = cPacketQueue("Queue for outgoing packets");
    cPacketQueue inQueue = cPacketQueue("Queue for incoming packets");
};
```

Figure 5.7: A connection entry corresponding to a port ID.

add an entry in the directory structure chosen. In RINASim specifically, application registration is implicit on network setup, which means that this step needs to happen in the initialisation stage.

5.4.1 API

These functions are callable from other modules, and will trigger a sequence of events relating to the state of a connection entry. The functions `arpResolutaionCompleted()` and `arpResolutionFailed()` are both triggered from signals emitted by the `RINArp` module.

`registerApplication(APN apnName)` ⇒ `void`

Called by the `ShimFA` module to notify both the `EthShim` and `RINArp` modules that an application has registered with the `IPCProcessShim`. This function retrieves the MAC address of the Ethernet interface, and asks `RINArp` create a static cache entry that maps the MAC address to the supplied `apnName` parameter. The `APN` typename here signifies an application name.

`createEntry(const APN &dstApn)` ⇒ `CreateResult`

Called by the `ShimFA` module to request the creation of a connection entry after a flow allocation request has been received. `CreateResult` is an `enum class` that supplies three return values: `::failed`, `::pending`, and `::completed`.

This function will initiate ARP address resolution, and if the ARP cache already contains an entry mapped to the supplied `dstApn`, `CreateResult::completed` is returned and `ShimFA` will proceed to create an FAI. If an entry does not exist, `RINArp` sends an ARP request, and `CreateResult::pending` is returned. If a connection entry already exists that is mapped to `dstApn`, `CreateResult::failed` is returned, and flow allocation has failed. The state for the

ConnectionEntry is set to ::initiatorAllocatePending, but if CreateResult::completed was returned, finalizeConnection() will be called in addition, setting the entry state to ::allocated.

deleteEntry(APN dstApn) ⇒ void
: Called by the ShimFA module when flow deallocation is requested. This may also happen when ARP address resolution has timed out when the state is ::allocated.

finalizeConnection(APN dstApn, const int portId) ⇒ bool
: Called by a ShimFAI module at the end of flow allocation, both when a flow is being requested from the IPCProcessShim, and when the IPCProcessShim requests a flow from the registered IPCProcess.

  This function uses the portId parameter in the names of the gates that are created for the ConnectionEntry mapped to the dstApn parameter.

arpResolutionCompleted(ConnectionEntry entry, APN apn, MacAddress mac) ⇒ void
: This function is invoked when a completedRINArpResolutionSignal signal has been emitted by RINArp.

  If the parameter entry is in the ::initiatorAllocatePending state, this function will call the ShimFA::completedAddressResolution() function, passing the apn parameter. The ShimFA module then proceeds to complete the flow allocation procedure.

  If the entry parameter is in the ::allocated state, any packets that are waiting for address resolution to be completed will be passed on to the EthernetInterface.

arpResolutionFailed(APN apn) ⇒ void
: This function is invoked when a failedRINArpResolutionSignal signal has been emitted from RINArp.

  If the connection entry corresponding with the apn parameter is in either the ::initiatorAllocatePending or ::allocated states, the connection entry will be destroyed, and ShimFA::failedAddressResolution() will be called, passing the apn parameter. The ShimFA will then proceed to deallocate the flow with apn as the destination application name.

If the connection entry found after a signal from RINArp has been emitted is in either the null or ::recipientAllocatePending states, neither arpResolutionCompleted() or arpResolutionFailed() will be triggered.

When an Ethernet frame arrives from an application that is not associated with any ConnectionEntry, a new one will be created,

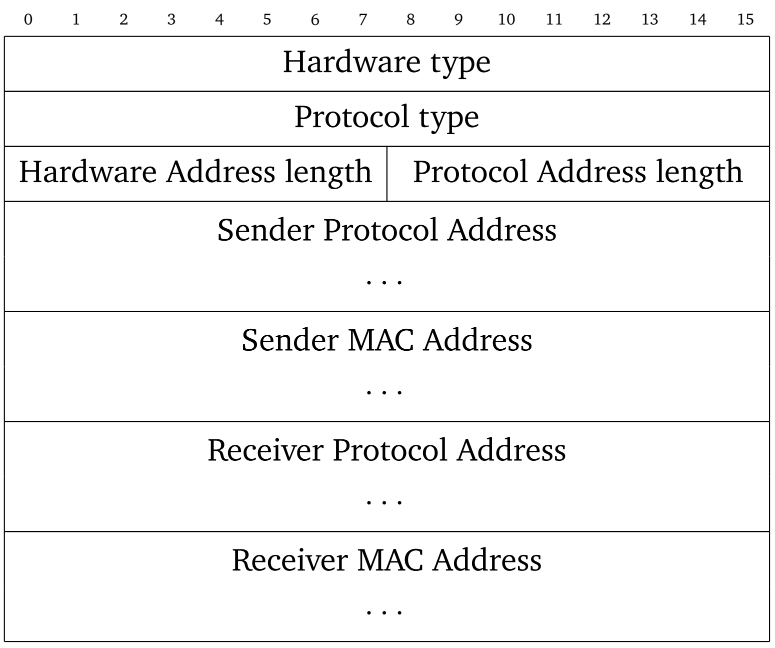


Figure 5.8: ARP packet, as per the specification[23]

and the state set to `::recipientAllocatePending`. It will then call `ShimFA::createUpperFlow()`.

5.5 RINArp

While INET has two ARP modules[2] that could be used directly, none of them are fully compliant with the ARP specification[23], and do not support variable length addresses which incidentally would be useful for an implementation that supports the naming scheme of RINA. The modules could be reused by either hashing or limiting the lengths of the application names of registered applications, but defining a separate and more generic ARP module could potentially also be useful for implementations of potential future shim IPCP specifications over Ethernet. The implementation is inspired by the `Arp` module in INET.

The module is named RINArp to distinguish it from other ARP modules. It for the most part follows the packet format of ARP as seen in fig. 5.8, although MAC addresses are still expected to be 6 bytes of length, and so the MAC address length header field is for this reason omitted in this particular implementation, as seen in fig. 5.9.

[2]`GlobalArp` and `Arp`.

```
class RINArpPacket extends inet::FieldsChunk
{
    uint8_t apnLength;
    uint8_t opcode @enum(ARPOpcode);
    inet::MacAddress srcMacAddress;
    APN srcApName;
    inet::MacAddress dstMacAddress;
    APN dstApName;
}
```

Figure 5.9: RINArp packet, with `ARPOpcode` being one of `ARPRequest` or `ARPResponse`.

`RINArp` consists of an ARP cache, which is implemented as a map, where a destination application name is mapped to an `ArpCacheEntry`, which holds a MAC address, timers, and a retry count.

5.5.1 API

The API of `RINArp` is mainly encapsulated by the functions `resolveAddress()` and `getAddressFor()`, where the former will initiate address resolution by sending an ARP request. For `RINArp` to be effective, it needs to have a static `ArpCacheEntry`, which is supplied through the `addStaticEntry()` function. Without it, it can still receive ARP requests, but will never reply with an ARP reply.

When an ARP request is received, the `RINArp` module will create an `ArpCacheEntry` if one does not already exist with the information supplied in the request. In either scenario, a new expiration timer will be set, and `RINArp` will now be able to resolve this address.

`resolveAddress(APN apn)` ⇒ `MacAddress`

This function is called by the `EthShim` module. It will search for the supplied `apn` in the ARP cache. If an entry is found, the corresponding MAC address is returned. Otherwise, an ARP request is generated, and `MacAddress::UNSPECIFIED_ADDRESS` is returned, which notifies that no corresponding address exists. The ARP request has the target MAC address set to `MacAddress::BROADCAST_ADDRESS`, which reaches every node on the segment within the same VLAN.

When an ARP request is sent, an `ArpCacheEntry` will be created in the table if an expired one does not already exist associated with the supplied `apn`. In either scenario, a new retry timer will be set for the entry corresponding to the `retryTimeout` parameter of the NED module.

`getAddressFor(MacAddress mac) ⇒ APN`
This function is called by the `EthShim` module when an Ethernet frame arrives, as it needs to find the correct `ConnectionEntry` associated with a flow to the registered `IPCProcess`. The function checks the ARP cache for an entry with a MAC address corresponding to the supplied parameter `mac`. If an entry is found, the APN corresponding with the entry is returned. Otherwise, `APN::UNSPECIFIED_APN` is returned, which notifies that there is no entry that corresponds to `mac`. In this case, `EthShim` will drop the packet.

`addStaticEntry(MacAddress mac, APN apn) ⇒ bool`
This function is called by the `EthShim` module on network initialisation to supply information about the registered application.

This must be called for this module to be able to deliver ARP responses.

`deleteStaticEntry() ⇒ void`
This function is unused for the time being, but will be relevant if RINASim at some point allows more control in regards to removing an `IPCProcess` during a simulation.

Its purpose is to remove the static entry in case an application unregisters. For the time being, there is no corresponding unregister call supplied by the `ShimFA`.

`setVlanId() ⇒ void`
This function is called by the `EthShim` module to supply the VLAN ID, which needs to be set for outgoing Ethernet frames through the `VlanReq` tag.

Chapter 6

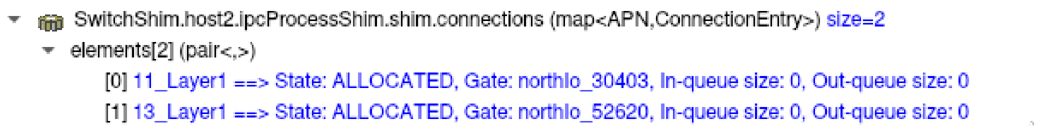
Evaluation of implementation

The efforts made in this thesis are captured most succinctly by showing the conformance of the implemented components with the specifications that were presented in the requirements section. This chapter first presents various example configurations created to show the capabilities of the Ethernet shim DIF. Then at the end of the chapter, a discussion about the implementation and test-cases is provided.

6.1 Output of a simulation

Through the use of the `WATCH` and `WATCH_MAP` macros that are part of OMNeT++, variables can be made available for inspection during simulation in `Qtenv`. The internal state of the `ConnectionEntry` map that is part of `EthShim` is made available in this way, and can be be inspected as seen in fig. 6.1.

The event logs can be used to determine the simulation success. For instance, if the VLAN ID used for an Ethernet frame is incorrect, it will be dropped. This might be notified in the event logs like in fig. 6.2.



```
SwitchShim.host2.ipcProcessShim.shim.connections (map<APN,ConnectionEntry>) size=2
  elements[2] (pair<,>)
      [0] 11_Layer1 ==> State: ALLOCATED, Gate: northIo_30403, In-queue size: 0, Out-queue size: 0
      [1] 13_Layer1 ==> State: ALLOCATED, Gate: northIo_52620, In-queue size: 0, Out-queue size: 0
```

Figure 6.1: The `connections` map from `EthShim`, with two entries in the allocated state.

```
** Event #26  t=10.00001162  TwoDIFsOneSwitch.host3.
   ipcProcessShim.eth.qEncap (Ieee8021qEncap, id=195)  on arpREQ
    (inet::Packet, id=87)

WARN:Received VLAN ID = 50 is not accepted, dropping packet.
```

Figure 6.2: Example of a dropped Ethernet frame due to incorrect VLAN ID in the Qtenv enivronment.

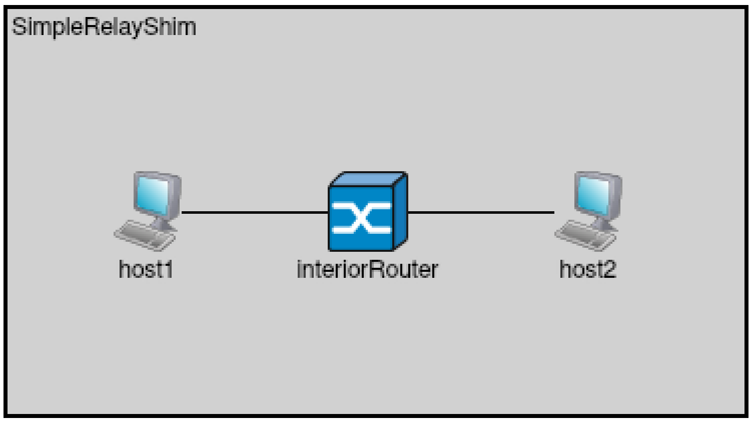


Figure 6.3: A router and two hosts utilising Ethernet shim DIFs.

6.2 Simulation configurations

The configurations shown in this section covers most use-cases of the shim IPCP over Ethernet. Each subsection has a corresponding figure that shows the network simulation in the simulation view of `Qtenv`, and a description of how the simulation works and what it tests. Since these evaluations are meant to test the shim

6.2.1 Simple relay example

This is a basic use-case where all the lowermost DIFs are provided by `IPCProcessShim` modules. In this configuration, `host1` will ping `host2` every second. This will lead to flow allocation being triggered in the `IPCProcessShim`, and also shows the normal `IPCProcess` being compatible with the `IPCProcessShim`.

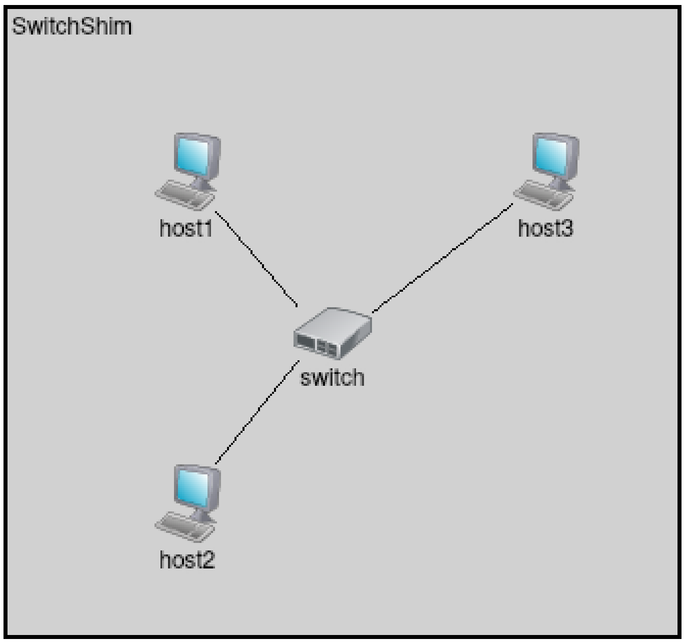


Figure 6.4: An Ethernet shim DIF over a switch with three CSs.

6.2.2 Three CSs on a switch

In this configuration example, as seen in fig. 6.4, three CSs communicate with each other utilising an Ethernet shim DIF over a switch. The configuration is set up in a way where `host1` will ping `host2` at 10000ms into the simulation, `host2` will ping `host3` at 10300ms, and host3 will ping `host1` at 10600ms. Each host subsequently pings its target CS each second until 200 seconds have passed.

The connection entries seen in fig. 6.1 are from `host2` in this example. The connection entry associated with the `11_Layer1` address first transitions from the `RECIPIENT ALLOCATE PENDING` state to the `ALLOCATED` state. The other connection entry is created when `host2` requests a connection with `host3`, and starts off in the `INITIATOR ALLOCATE PENDING`, and when an ARP response is received from `host3` it will transition to the `ALLOCATED` state.

This configuration is among the ones that required changes to the handling of flow allocation in the `IPCProcess`, since more than one neighbouring IPCP can be reached that are not yet enrolled in the DIF. Therefore the originating IPCP needs to go through several CACE phases.

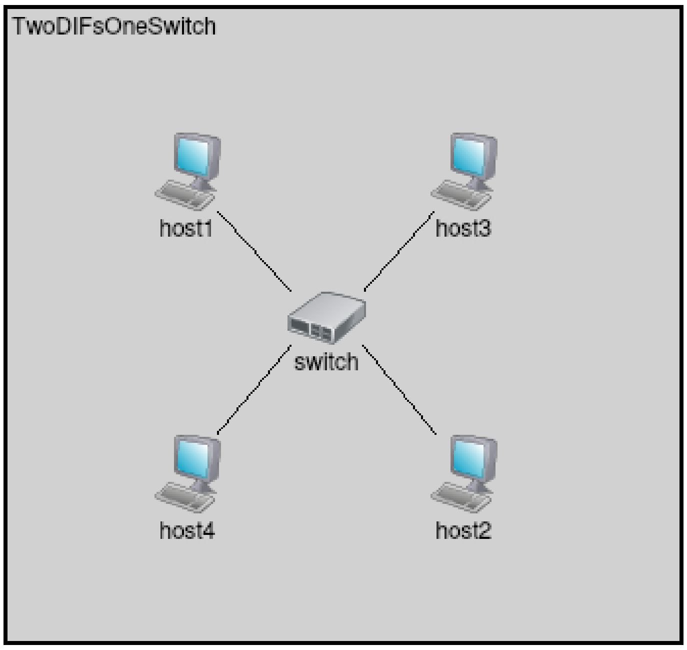


Figure 6.5: Two Ethernet shim DIFs over the same Ethernet segment.

6.2.3 Two DIFs on a switch

This network configuration provides an example of using VLANs to separate DIFs over the same Ethernet segment. It is set up in a way where `host1` will ping `host2` continually, and `host3` will ping `host4` continually. The `Qtenv` event log transcript in fig. 6.2 is from this example, and this happens at the Ethernet interface of the `host3` CS when its `IPCProcessShim` receives an Ethernet frame from a DIF it is not a member of.

The `IPCProcessShim` modules of `host1` and `host2` in this example have their DIF names set to “50”, while the `IPCProcessShim` modules of `host3` and `host4` have their DIF names set to “60”. Since these DIF names are also used as VLAN IDs, any Ethernet frames that do have a VLAN tag with a corresponding ID will be dropped at arrival.

This example can also be configured to make the switch VLAN aware. For this, each Ethernet interface has to be configured with its own filter, as seen the implementation section in fig. 5.4. This aids in visualisation by making it clear which channels are intended for a specific VLAN or shim DIF.

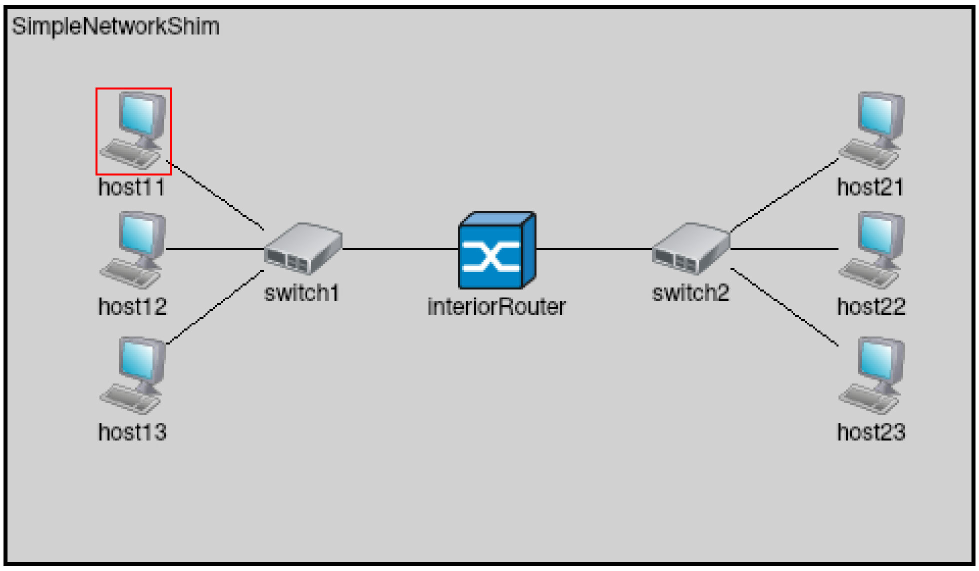


Figure 6.6: Two Ethernet shim DIFs over the same Ethernet segment.

6.2.4 DIFs over switches and a router

The example seen in fig. 6.6 provides a simple network configuration that represents a combination of the first two evaluation cases. The highlighted `host11` will ping `host23` continually, to see how SDUs and ARP messages are passed correctly over both switches.

6.3 Discussion

RINASim was part of a research project, and has been programmed as proof-of-concept. While it serves well as a simulation framework for policy implementations, extending the code has been difficult because of shortcuts taken as part of the implementation. The overt reliance upon signals as a mechanism also made the flow of execution difficult to follow, which made it difficult to fit the `IPCProcessShim` to the IPC API provided.

The solution regarding whether a management flow will be allocated is not entirely correct, as management flows should be multicast. Before the fix, enrollment would only be attempted once when trying to reach a neighbouring `IPCProcess`, which led to no management flow being allocated at the receiving end if an `IPCProcess` attempted to reach several several neighbours. This happens because the `ResourceAllocator` module is asked to allocate a management flow only once, and the CACE phase is started when an underlying requested management flow has completed allocation.

Instead, a potential solution could have been to change flow allocation so only one management flow will be allocated within an `IPCProcess`, but it will request the allocation of management flows from every underlying `IPCProcess` it uses to reach other remote `IPCProcess` modules. While the solution provided in this thesis works, it will lead to unnecessary overhead due to several management flows within an `IPCProcess`. This leads to more flow creation procedures, which creates noise in the simulation event logs.

The test-cases provided as part of this chapter only model a few scenarios for the Ethernet shim DIF. Some additional scenarios that could be valuable to show its capabilities, such as utilising both shim on different segments. This should still work as part of the implementation.

Implementing CSs that support both the RINA and TCP/IP stacks could also be a significant step towards showing more advanced migration cases, but interfacing these would need a specialised application that translates from the RINA stack to transport layer protocols such as TCP. Additional shim layers are mentioned in Vrijders, Trouva, Day, *et al.* [12], and is a good next step for RINASim as a project.

More could have been done to evaluate the practical capabilities of the shim DIF, as none of the example test-cases try to hit any limits of the protocol. As such, it is not able to tell if the implementation will hold if the throughput is high enough.

Chapter 7

Conclusion

The two main goals of the thesis are fulfilled, which were to integrate the INET framework, and implement a shim IPCP over Ethernet for RINASim. Both of these goals were successful, and several example configurations have been added to the RINASim repository that show the various use-cases of the Ethernet shim DIF.

The INET framework was integrated without much additional work required. The addition of a compatibility layer with INET allows realistic simulation scenarios where Ethernet-compatible infrastructure can be utilised and communicated with, including switches and hosts running the traditional TCP/IP stack.

The work provided in this thesis demonstrates the viability of the Ethernet shim as part of the solution for gradual migration to RINA. The implementation serves as a basis that can be used for future shim IPCPs.

A simple testing facility was added to the repository to automatically detect if changes to the codebase introduce unforeseen consequences. This was helpful during the implementation of the Ethernet shim DIF, and the author hopes that this tool can be used and eventually extended with more test configurations. Some work was also put into making the code readable and well-documented, with the intention that the `IPCProcessShim` module can be easily extended upon in the future.

7.1 Future work

The work of this thesis has laid the ground work for allowing RINASim to employ new translation layers to run advanced simulations and adoption scenarios on more realistic networks. Shim layers over TCP and UDP that connect hosts running the RINA stack could be implemented to allow overarching DIFs to exist over TCP/IP networks built using the INET framework. The Ethernet shim-layer works as a proof of concept for the viability of such an implementation. A next step could be to create a host that supports both the TCP/IP and RINA networking stacks.

Additionally, serialisation is a necessary component for allowing HIL simulation or network emulation, which could let RINASim communicate with other implementations of RINA. This would require fundamental changes to how RINASim handles construction of packets, as well as specialised functions that provide serialisation of the contents in the packets. RINASim may also benefit from using the new packet API of INET, with the added benefits listed in Varga, Bojthe, Meszaros, *et al.* [22]. However, making INET a dependency of RINASim may not be desirable.

As RINASim has a limited following with eight core contributors and a rarely updated codebase at the time of writing, it also has quite a bit of technical debt and inefficient code[1]. Efficiency is very important when running large-scale simulations with tens or hundreds of hosts, and technical debt that seems insignificant may add several hours to the time required for a simulation to finish.

RINASim could also benefit from unit testing various parts of the codebase, as this could not only lead to bugs being detected, but also more segmented code, which could be beneficial for maintenance. The added fingerprint testing framework works as a form of integration testing, as it evaluates the interaction between the components, but it cannot validate the quality of the components by themselves. The INET framework has a large testing framework that could serve as inspiration for this task.

[1]One of the best examples of which is that many signals propagate to every module that is subscribed, regardless if they are intended for that module or not.

Glossary

C++
: An object-oriented programming language used for the OMNeT++ framework. 25, 26, 44

IEEE 802.11
: Standards for wireless NICs. 35

IEEE 802.1Q
: The VLAN standard. ix, 13–16, 36, 42

IEEE 802.3
: The Ethernet standard, compatible with Ethernet II frames. 16, 36

INET
: A simulation model library for OMNeT++ that supplies a vast array of protocol and infrastructure models from TCP/IP. 3, 35, 36, 40, 42, 50, 59, 60

OMNeT++
: A discrete-event simulation framework to simulate networks. 3, 25, 26, 28, 30, 35, 39, 40, 53

RINASim
: A simulation model library for OMNeT++ that represents RINA. i, ix, 2, 3, 25, 28, 29, 35, 36, 38, 41, 45, 52, 57–60

TCP/IP
: Transmission Control Protocol/Internet Protocol, another name for the Internet. i, 1–3, 12, 13, 28, 35, 36, 42, 58–60

Acronyms

AE Application Entity 7–10
AEI-id Application Entity Instance Identifier 8
AEN Application Entity Name 8
ANI Application Naming Information 8
AP Application Process 7, 8, 10, 28
API Application Programming Interface 10, 13, 17–19, 28–30, 35, 36, 39, 41, 45, 46, 51, 57, 60
API-id Application Process Instance Identifier 8
APN Application Process Name 8, 10, 52
ARP Address Resolution Protocol x, 17, 19, 21, 22, 41, 42, 46–52, 55, 57

CACE Common Application Connection Establishment 5, 31, 32, 40, 55, 57
CDAP Common Distributed Application Protocol 5, 7, 8, 32, 33
CEP-id Connection End-Point Identifier 8, 9, 14
CRC Cyclic Redundancy Check 14
CS Computing System x, 5, 28, 30, 42, 55, 56, 58

DA DIF Allocator 7, 17, 28, 40
DAF Distributed Application Facility 6, 7, 28, 29
DIF Distributed IPC Facility i, ix, x, 3, 5–9, 11, 13–18, 20, 28–31, 33, 35, 39–43, 47, 53–60
DSAP Destination Service Access Point 14
DTCP Data Transfer Control Protocol 11, 32
DTP Data Transfer Protocol 11, 31, 32

EFCP Error and Flow Control Protocol 8, 11
EFCPI EFCP Instance 11, 31–33

FA Flow Allocator 10, 30, 45, 46
FAI FA Instance 10, 30, 31, 45, 48
FCS Frame Check Sequence 14, 17, 18, 36

HIL Hardware-In-the-Loop 35, 60

IPC Inter-Process Communication i, 2, 5, 9, 30, 57
IPCP IPC Process ix, x, 5–11, 13, 14, 16–19, 21–23, 25, 28, 33, 35, 38–42, 44, 46, 50, 54, 55, 59
IPv4 Internet Protocol version 4 7, 17, 35, 42
IPv4 Internet Protocol 7, 17
IPv6 Internet Protocol version 6 35
IRATI Investigating RINA as an alternative to TCP/IP ix, 15
IRM IPC Resource Manager 7, 45

LLC Logical Link Control ix, 14–16
LTE Long-Term Evolution 35

Bibliography

[1] S. Lukasik, “Why the ARPANET was built,” *IEEE Annals of the History of Computing*, vol. 33, no. 3, pp. 4–21, 2010.

[2] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K.-J. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, and D. Damjanovic, “De-ossifying the internet transport layer: A survey and future perspectives,” *IEEE Communications Surveys & Tutorials*, vol. 19, no. 1, pp. 619–639, 2016.

[3] M. Handley, “Why the Internet only just works,” *BT Technology Journal*, vol. 24, no. 3, pp. 119–129, 2006.

[4] V. Fuller, “Scaling issues with routing+ multihoming,” *Plenary session at APRICOT*, 2007.

[5] P. Richter, M. Allman, R. Bush, and V. Paxson, “A primer on IPv4 scarcity,” *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 2, pp. 21–31, 2015.

[6] S. M. Bellovin, “Security problems in the TCP/IP protocol suite,” *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 2, pp. 32–48, 1989.

[7] E. Trouva, E. Grasa, J. Day, I. Matta, L. T. Chitkushev, P. Phelan, M. P. De Leon, and S. Bunch, “Is the Internet an unfinished demo? Meet RINA!” In *TERENA Networking Conference*, 2010, pp. 1–12.

[8] E. Grasa, O. Rysavy, O. Lichtner, H. Asgari, J. Day, and L. Chitkushev, “From protecting protocols to protecting layers: Designing implementing and experimenting with security policies in rina,” in *IEEE ICC 2016, Communications and Informations Systems Security Symposium*, 2016.

[9] J. Day, “How Naming and Addressing (and Routing) Are Suppose to Work,” 2016.

[10] B. Edelman, “Running out of numbers: Scarcity of IP addresses and what to do about it,” in *International Conference on Auctions, Market Mechanisms and Their Applications*, Springer, 2009, pp. 95–106.

[11] G. Boddapati, J. Day, I. Matta, and L. Chitkushev, “Assessing the security of a clean-slate internet architecture,” in *2012 20th IEEE International Conference on Network Protocols (ICNP)*, IEEE, 2012, pp. 1–6, ISBN: 1-4673-2447-7.

[12] S. Vrijders, E. Trouva, J. Day, E. Grasa, D. Staessens, D. Colle, M. Pickavet, and L. Chitkushev, “Unreliable inter process communication in Ethernet: Migrating to RINA with the shim DIF,” in *2013 5th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, IEEE, 2013, pp. 215–221, ISBN: 1-4799-1177-1.

[13] R. W. Watson, “The delta-t transport protocol: Features and experience,” in *[1989] Proceedings. 14th Conference on Local Computer Networks*, IEEE, 1989, pp. 399–407, ISBN: 0-8186-1968-6.

[14] E. Grasa, F. Salvestrini, G. Carrozzo, P. Cruschelli, A. Chappel, J. Graham, S. Vrijders, D. Staessens, M. Tarzan, L. Bergesio, E. Trouva, A. Vico, and C. Bermudo, "IRATI Deliverable 3.1: First phase integrated RINA prototype over Ethernet for a UNIX-like OS," 2013.

[15] T. Kiravuo, M. Sarela, and J. Manner, "A survey of Ethernet LAN security," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 3, pp. 1477–1491, 2013.

[16] J. Day, *The Interina Reference Model - Part 3: Distributed InterProcess Communication - Chapter 1: The Distributed IPC Facility (DIF)*, 2014.

[17] A. Varga. (2020). "OMNeT++ - Simulation Manual," [Online]. Available: https://doc.omnetpp.org/omnetpp/manual/ (visited on 08/24/2020).

[18] V. Vesely, *RINASim Simulator, https://github.com/kvetak/RINA*, Nov. 11, 2019.

[19] V. Veselý, M. Marek, and K. Jeřábek, *Deliverable 2.6: RINASim-Advanced Functionality*. 2015.

[20] V. Veselỳ, M. Marek, and K. Jeřábek, "RINASim," in *Recent Advances in Network Simulation*, Springer, 2019, pp. 139–181.

[21] A. Varga, Z. Bojthe, L. Meszaros, G. Szászkő, R. Hornig, and A. Török. (2020). "INET Framework - What Is INET Framework?" [Online]. Available: https://inet.omnetpp.org/Introduction.html (visited on 08/24/2020).

[22] ——, (2020). "Migrating Code from INET 3.x — INET 4.2.0 documentation," [Online]. Available: https://inet.omnetpp.org/docs/migration-guide/index.html (visited on 08/24/2020).

[23] D. C. Plummer, "RFC 826: An ethernet address resolution protocol," *InterNet Network Working Group*, 1982.