RINASim

Vladimír Veselý, Marcel Marek, Kamil Jeřábek

Abstract RINASim is a framework for simulation of networks following principles of clean-slate Recursive InterNetwork Architecture. RINASim should help others to understand RINA concepts that are often far beyond current TCP/IP networking craftsmanship. This chapter introduces basic theory behind RINA and outlines RINASim mechanisms and policies. RINASim contains a plethora of high-level and low-level models offering behavior strictly in compliance with RINA specifications. In order to understand RINASim capabilities, the chapter also presents narrative demonstration scenarios.

1 Introduction

Recursive InterNetwork Architecture (RINA) is the clean-slate architecture aimed to change the whole Internet unlike just temporary fixes for current status quo. The RINA concept is based on John Day's thoughts, lectures and book [2] regarding ISO/OSI initiative failure, TCP/IP development, commercial adoption of the Internet and other technical/political events in Internet history.

The architecture as proposed by RINA is fundamentally different from the current TCP/IP networking. The RINA approach focuses on a few principles instead of a broad and complex eco-system of the modern Internet. The idea of the recursive composition of layers arises naturally from the structure of repeating computer networking patterns. Instead of strictly separating network functions into a predefined set of layers, RINA enables to compose a stack of layers that may offer a nearly the

Vladimír Veselý, e-mail: veselyv@fit.vutbr.cz
Brno University of Technology, Božetěchova 2, Brno 61266, Czech Republic

Marcel Marek, e-mail: imarek@fit.vutbr.cz
Brno University of Technology, Božetěchova 2, Brno 61266, Czech Republic

Kamil Jeřábek, e-mail: ijerabek@fit.vutbr.cz
Brno University of Technology, Božetěchova 2, Brno 61266, Czech Republic

same set of functions. All RINA layers employ the same protocols which contrast to the TCP/IP model, where each layer defines its set of protocols. RINA was designed to provide a simpler and efficient alternative to the current Internet architecture.

Section 2 provides brief information how to install RINASim and start working with it. Section 3 starts with a description of high-level RINA network nodes. Section 4 goes even deeper and outlines various components and policies. The whole content of Section 5 is dedicated to a thorough description of few simulations, which illustrate basic scenarios of RINA network operation. This chapter is summarized in Section 6.

2 Installation

RINASim [22] is a stand-alone framework for the OMNeT++ discrete event simulator environment. RINASim is coded from scratch and independent of another library. The main purpose is to offer the community a reliable and the most up-to-date tool (in the sense of RINA specification compliance) for simulating RINA-based computer networks. RINASim at its current state represents an entirely working implementation of the simulation environment for RINA. The simulator contains all mechanisms of RINA according to the current specification.

RINASim installation is a straightforward process with two phases: 1) obtain the source codes; 2) compile the project, which creates one static library (*librinasimcore* containing simulation core) and one dynamic library (*librinasim* binding together core and implementation of various policies).

RINASim is developed in OMNeT++ 5.2.1. RINASim August 2016's release is the last one compatible with OMNeT++ 4.6. The current trend is to make RINASim operatable on any OMNeT++ versions that support C++ 11 language standard and GCC 4.9.2 compiler. All source codes (including master and other thematic branches) are publicly available on the project's GitHub repository [23]. Manual installation and building RINASim from the source code is pretty simple. Just clone or download the main branch, import the project into OMNeT++ (it is named `rina`), compile it and start playing with RINASim. The user should be prepared for rather long initial compilation time.

3 High-level Design

The purpose of this section is to provide future RINASim user with a short introduction (or more accurately executive summary) to RINA concepts. These concepts and ideas formulate the design and development of the whole RINASim.

3.1 Overview

This subsection introduces the theoretical background. However, explanation of the whole Recursive Internet Architecture is far beyond the scope of this paper. Hence, only parts relevant to the current RINASim functionality are captured. Synthesis of RINA information provided below comes from the following sources: [11], [12], [14], [8] and [9].

Nature of Applications and Application Protocols

The set of Internet applications was rather simplistic before WWW – one application with a single instance using only one protocol. Hence, there is nearly no distinction between an application and its networking part. However, the web completely changed this situation – one application protocol may be used by more than one application (e.g., HTTP is being over-employed as a communication protocol), and also one application may have many application protocols (e.g., web browsers, mail clients).

Following terms are recognized in the frame of RINA, and their relationship is depicted in Figure 1 below:

- *Application Process (AP)* – Program instantiation to accomplish some purpose;
- *Application Entity (AE)* – AE is the part of AP, which represents application protocol and application aspects concerned with communication.

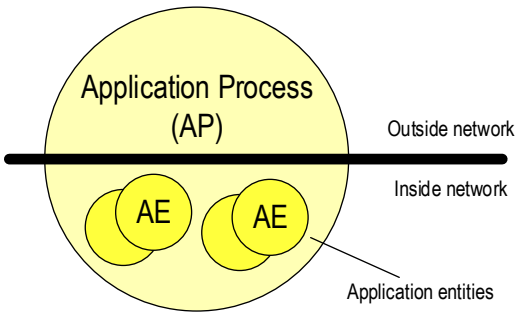

Fig. 1: Application Protocol and Application Entities relationship

There may be multiple instances of the Application Process in the same system. AP may have multiple AEs; each one may process a different application protocol. There also may be more than one instance of each AE type within a single AP.

All application protocols are stateless; the state is and should be maintained in the application. Thus, all *application protocols* modify shared state external to the protocol itself on various objects (e.g., data, file, HW peripherals). Because of that, there is only one application protocol that contains trivial operations (e.g., read/write, start/stop). *Data transfer protocols* modify state internal to the protocol; the only external effect is the delivery of Service Data Unit (SDU).

Core Terms

The data transport and internetworking tasks together (generally known as networking) constitute *inter-process communication (IPC)*. IPC between two APs on the same operating system needs to locate processes, evaluate permission, pass data, schedule tasks and manage memory. IPC between two APs on different systems works similarly plus adding functionality to overcome the lack of shared memory.

In traditional networking stack, the layer provides a service to the layer immediately above it. The recursion (and repeating of patterns) is the main feature of the whole architecture. Layer recursion became more popular even in TCP/IP with technologies like Virtual Private Networks (VPNs) or overlay networks (e.g., OTV, TOR). Recursion is a natural thing whenever we need to affect the scope of communicating parties. However, so far it was just recursion of repeating functions in existing layers.

In ISO/OSI or TCP/IP, there is a set of layers each with completely different functions. RINA on the other hand yields the idea of the single generic layer with fixed mechanisms but configurable policies. This layer is in RINA called *Distributed IPC Facility (DIF)* – a set of cooperating APs providing IPC. There is not a fixed number of DIFs in RINA; we can stack them according to the application or network needs. From the DIF point of view actual stack depth is irrelevant, DIF must know only (N+1)-layer above and (N-1)-layer below. DIF stacking partitions network into smaller, thus, more manageable parts.

The concept of RINA layer could be further generalized to *Distributed Application Facility (DAF)* – a set of cooperating APs in one or more computing systems, which exchange information using IPC and maintain shared state. A DIF is a DAF that does only IPC. *Distributed Application Process (DAP)* is a member of a DAF. The *IPC Process (IPCP)* is special AP within DIF delivering inter-process communication. IPCP is an instantiation of DIF membership; computing system can perform IPC with other DIF members via its IPC process within this DIF. An IPCP is specialized DAP. The relationship between all newly defined terms is depicted inFigure 2.

DIF limits and encloses cooperating processes in the one scope. However, its functionality is more general and versatile apart from rigid TCP/IP layers with dedicated functionality (i.e., datalink layer for adjacent node communication, a transport layer for reliable data transfer between applications). DIF provides IPC to either another DIF or to DAF. Therefore, DIF uses a single application protocol with generic primitive operations to support inter-DIF communication.

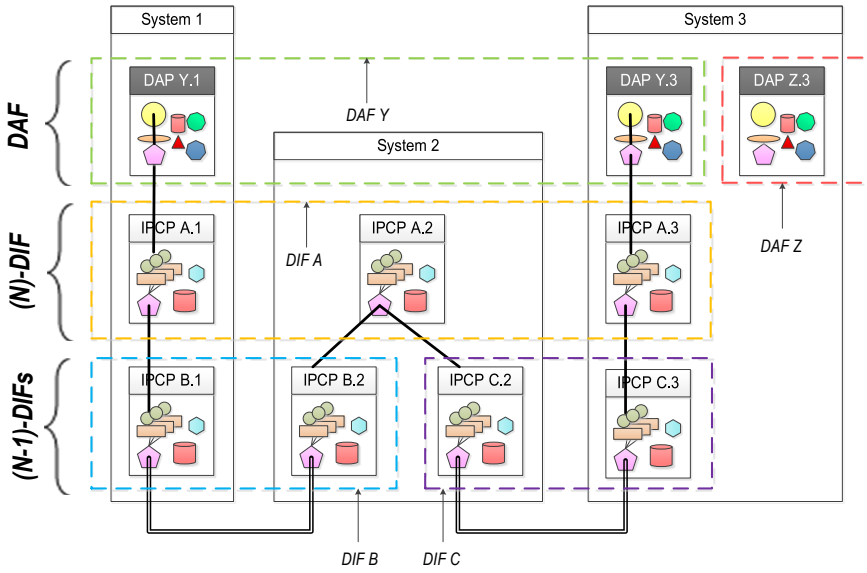

Fig. 2: DIF, DAF, DAP and IPCP illustration

Connection-oriented vs. Connectionless

The clash between connection-oriented and connectionless approaches (that also corrupted ISO/OSI tendencies) is from RINA perspective quite easy to settle. Connection-oriented and connectionless communication are both just functions of the layer that should not be visible to applications. Both approaches are equal, and it depends on application requirements which one to use. On the one hand, connectionless is characterized by the maximal dissemination of the state information and dynamic resource allocation. On the other hand, connection-oriented limits the dissemination and tends toward static resource allocation. The first one is good for low volume stochastic traffic. The second one is useful for scenarios with deterministic traffic flows.

If the applications request the allocation of communication resources, then the layer determines what mechanisms and policies to use. Allocation is accompanied with access rights and description of Quality of Service (QoS) (e.g., what minimum bandwidth or delay is needed for correct operation of application). QoS demands are then translated into the appropriate QoS class called *QoS-cube*.

Delta-t Synchronization

All properly designed data transfer protocols are soft-state. There is no need for explicit state synchronization (hard-state) and tools like SYNs and FINs are unnecessary.

Initial synchronization of communicating parties is done with the help of the Delta-t protocol with the main variable denoted as Δt (see [27] and [15]). Delta-t was developed by Richard Watson, who proposed time-based synchronization technique. He proved that conditions for distributed synchronization were met if the following three timers are realized: a) Maximum Packet Lifetime (MPL); b) Maximum time to attempt retransmission a.k.a. maximum period during sender is holding Protocol Data Unit (PDU) for retransmission while waiting for a positive acknowledgment (a.k.a. R); c) Maximum time before Acknowledgement (a.k.a. A).

Delta-t assumes that all connections exist all the time. Synchronization state is maintained only during the activity. The activity is defined as $2 \cdot \Delta t$ from the receiver side and $3 \cdot \Delta t$ from the sender side, where $\Delta t = R + MPL + A$. After 2-3 Δt periods without any traffic, the state may be discarded which effectively resets the connection. Because of that, there are no hard-state (with explicit synchronization) protocols only soft-state ones. Delta-t postulates that port allocation and synchronization are distinct.

Separation of Mechanism and Policy

We understand the term mechanism as the fixed part and policy as the flexible part of IPC.

If we clearly separate them, we discover that there are two types of mechanisms:

- *tightly-bound* that must be associated with every PDU, which handles fundamental aspects of data transfers;
- *loosely-bound* that may be associated with some data transfer PDUs, which provide additional features (namely reliability and flow control).

Both groups are coupled through a state vector maintained separately per flow; every active flow has its state-vector holding state information. For instance, the behavior of retransmission and flow control can be heavily influenced by chosen policies, and they can be used independently of each other.

This implies that only a single generic data transfer protocol based on Delta-t is needed, which may be governed by different transfer control policies. This data transfer protocol modifies the state internally, where the application protocol (carried inside) modifies state externally.

Naming and Addressing

The Application Process communicates in order to share states. We mentioned that AP consists of AEs. We need to differentiate between different APs and also different AEs within the same AP. Thus, RINA is using following set of identifiers to achieve desired naming properties:

- *Distributed-Application-Name (DAN)* is name that identifies a distributed application, and it is globally unambiguous. One DAF might have assigned more than one DAN with different access control properties;

- *Application Process Name (APN)* is a globally unambiguous synonym for an AP of DAF;
- *Application Process Instance Identifier (API-id)* is an identifier bound to an AP Instance to distinguish multiple AP Instances. It is unambiguous within the AP;
- *Application Entity Name (AEN)* is unambiguous within the scope of the AP.
- *Application Entity Instance Identifier (AEI-id)* is an identifier that is also unambiguous within a single AP, and it helps us to identify different AE instances.
- *Application Naming Information (ANI)* references a complete set of identifiers to name particular application; it consists of a four-tuple APN, API-id, AEN, and AEI-id. The only required part of ANI is APN; others are optional.

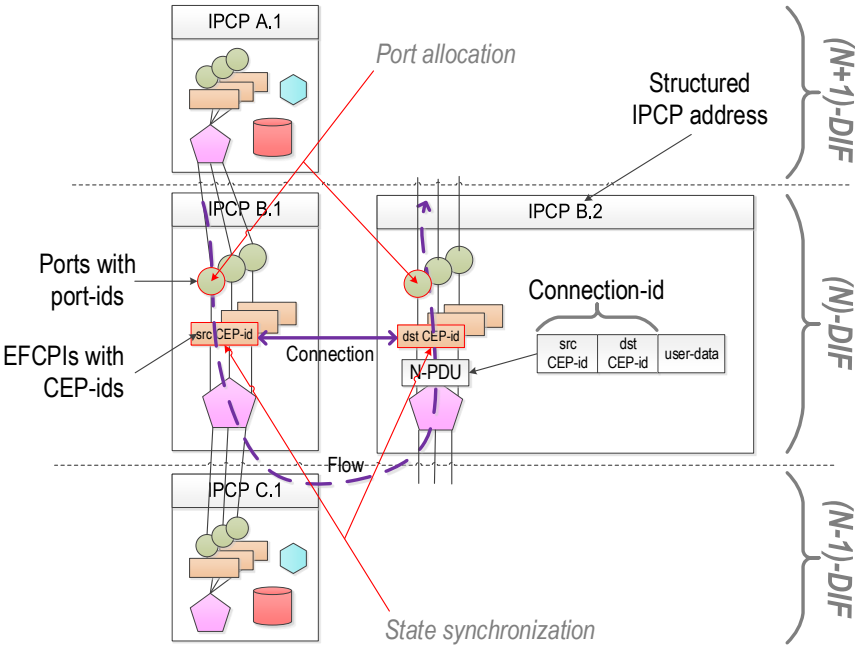

Fig. 3: Overview of IPCP local identifiers

IPC Process has APN to identify it among other DIF members. A RINA *address* is a synonym for IPCP's APN with a scope limited to the layer and structured to facilitate forwarding. APN is useful for management purposes but not for forwarding. Address structure may be topologically dependent (indicating the nearness of IPCPs). APN and address are simply two different means to locate an object in different context. There are two local identifiers important for IPCP functionality – port-id and connection-endpoint-id. *Port-id* binds this (N)-IPCP and (N+1)-IPCP/AP; both of them use the same port-id when passing messages. Port-id is returned as a handle to the communication allocator and is unambiguous within a computing system.

Connection-endpoint-id (CEP-id) identifies a shared state of one communication endpoint. Since there may be more than one flow between the same IPCP pair, it is necessary to distinguish them. For this purpose, Connection-id is formed by combining source and destination CEP-ids with QoS requirements descriptor. CEP-id is unambiguous within IPCP and Connection-id is unambiguous between a given pair of IPCPs. Figure 3 depicts all relevant identifiers between two IPCPs.

Watson's Delta-t implies port-id and CEP-id in order to help separate port allocation and synchronization. RINA's *connection* is a shared state between ends identified by CEP-ids. RINA's flow is when connection ends are bound to ports identified by port-ids. The lifetimes of flow and its connection(s) are independent of each other.

The relationship between node and Point of Attachment (PoA) is relative – node address is (N)-address, and its PoA is (N-1)-address. Routes are sequences of (N)-addresses, where (N)-layer routes based on this addresses (not according to (N-1)-addresses). Hence, the layer itself should assign addresses because it understands address structure.

3.2 Nodes

There are only three basic kinds of nodes in a RINA network (illustrated in Figure 4). Each kind represents a computing system running RINA:

- *Hosts* – end-devices for IPC containing AEs in the top layer; they employ two or more DIF levels;
- *Interior routers* – interim devices, which are interconnecting (N)-DIF neighbors via multiple (N-1)-DIFs; they employ two or more DIF levels;
- *Border routers* – interim devices, which are interconnecting (N)-DIF neighbors via (N-1)-DIFs, where some of (N-1)-DIFs are reachable only through (N-2)-DIFs; they employ three or more DIF levels.

As seen in Figure 4, the main difference between node kinds is in an overall number of DIF levels present in a computing system. Due to the limited number of network interface cards (NIC), Hosts usually have a single 0-DIF (connected to the physical medium) and a few 1-DIFs leveraging on this lowest level DIF. Interior routers have potentially a lot of 0-DIFs (for each interface) but only a few relaying 1-DIFs. Border routers also perform relaying but serve as gateways between those (N-1)-IPCs, which are not connected directly. Thus, a (N-2)-DIF is needed to reach the physical medium.

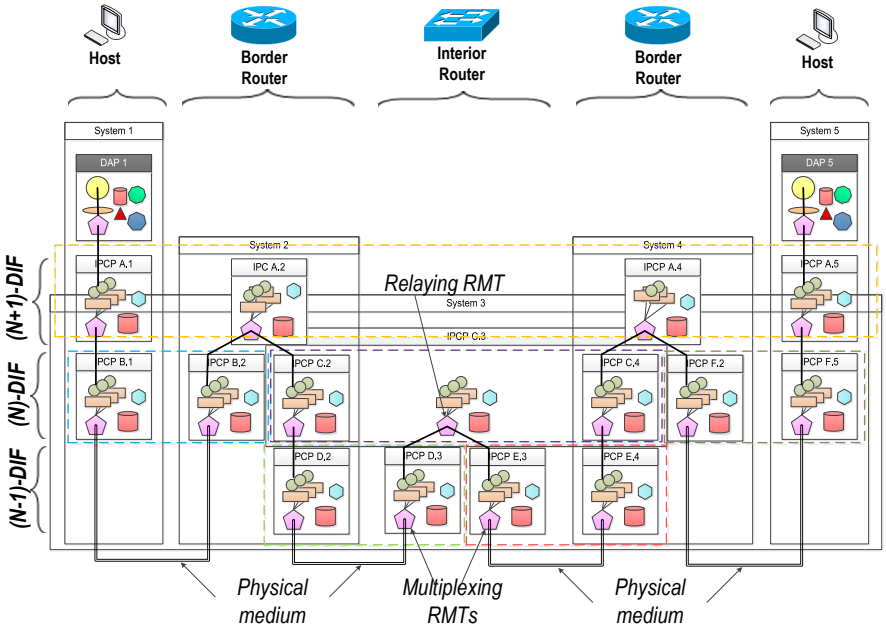

Fig. 4: Example of RINA network with three levels of DIFs and different nodes

3.3 DAF Design

DIF Allocator Interface

The primary task of the *DIF Allocator (DA)* is to return a list of DIFs where the destination application may be found based on ANI reference and access control information. An additional and more complex DA description is available in [25]. The DA contains and works with multiple mapping tables to provide its services:

- Naming information table – provides an association between APN and its synonyms;
- Search table – provides a mapping between requested APN and the list of DAs where to search for it next;
- Neighbor table – maintains a list of adjacent peers when trying to reach other DAs;
- Directory – contains records mapping APNs with access rights to the list of supporting DIFs including DIF's name, access control information and provided QoS.

IPC Resource Manager

IPC Resource Manager (IRM) (see specification [6]) as its name suggests manages DAF resources. This involves multiple different tasks:

- IRM processes allocate calls by delegating them to appropriate local IPCPs in relevant DIFs;
- IRM manages DA queries and acts upon their responses. When the DA response contains more than one DIF, IRM chooses which DIF to use;
- IRM administers the use of flows between AEs and DIFs. IRM may choose to multiplex a single or multiple AE flows into single/multiple flows to a set of DIFs;
- IRM initiates joining or creating DAF and/or DIF. IRM acts upon the DAF, or DIF lost (e.g., sending notifications or perform subsequent actions).

Application Process

The AP is a program intended to accomplish some purpose, which can be instantiated on a processing system. An AP contains one or more AE(s) introduced in the following section. AP manages some of system resources, such as the processor, storage, and IPC.

An AP must have at least one AE. Otherwise, the AP would have no input/output and would lack the state-sharing purpose.

Application Entity

An AE is a component of an AP (task). An AP needs to communicate with other APs for multiple purposes, potentially at the same time. The goal of AEs is to create and manage these application connections.

An Application Entity implements an application protocol. Such protocol provides a shared understanding of the purpose of communication, protocol, set of objects and their meaning that the two AEs exchange. There is only one application protocol called *Common Distributed Application Protocol (CDAP)* used for communication.

AEs could be implemented by subroutine libraries which should be hidden to application programmers – they would control it via API calls (e.g., similar to BSD sockets). The example should provide a better understanding of the purpose of the AEs. Imagine an application that contains two different AEs. One AE should be responsible for serving requests for web pages, while another AE might be involved in communicating with network database server. Each of these AEs has its defined purpose, set of objects they communicate, and communication protocol.

Instances of Application Processes and Application Entities

The *Application Process Instance (AP Instance)* and the *Application Entity Instance (AE Instance)* are instantiations of the AP and AE tasks. One processing system may

contain multiple instances of the same AP. Also, it is possible to have many instances of the same AE in one AP.

It is possible to create an application connection specifically to one of the instances of AP and AE. Each instance can manipulate with unique data, and it can have unique parameters.

A video conference is a good use-case of how the AP and the AE Instances might be distinguished. Imagine, that one AP Instance is a video call with more than one participant. An AE implements a video-streaming protocol, and an AE Instance represents feed from one camera of a single participant.

Common Distributed Application Protocol

CDAP is the only required application protocol in RINA. It provides a platform for building all distributed applications. CDAP allows distributed applications to deal with communication at the object level without the need to do an explicit serialization and other input/output operations. The CDAP unifies the approach of sharing data over the network, so we do not need to create any additional specialized protocols.

From the application perspective, the only operations, which can be performed on objects, are create/delete, read/write, and start/stop (execute/suspend). These operations are primarily supported by the CDAP.

CDAP consists of three subparts. *Common Application Connection Establishment (CACE)* that is involved in connection initialization. *Authentication* which is responsible for authentication of communication parties. The last one, *Common Distributed Application Protocol* is processing all other messages.

Enrollment

Enrollment is the phase of communication that follows right after the CDAP connection is established with a member of a DIF/DAF. In RINA, there exist two types of Enrollment - within DIF and within DAF. They differ mainly from the perspective of the information that is exchanged during this phase. An AP must always be enrolled to become a full-featured member of a DAF.

The Enrollment may perform following operations:

- determine the current state of the member AP (if AP is joining the DAF for the first time, or if it is a returning member);
- assign capabilities and synonyms to the new member relevant within the DAF;
- initialize static aspects, policies and synchronize RIBs;
- create additional connections.

Resource Information Base

Resource Information Base (RIB) is the logical representation of the local repository of objects in DAF. Each member of a DAF has its portion of the information stored in the local RIB. All objects are accessible via the RIB Daemon, that is responsible for managing and maintaining them.

The RIB is a storage (from the operating system perspective) and there are no restrictions how to implement it. In the DAF, it should most likely be implemented as some (key-value/relational/temporal) database of application objects.

In current TCP/IP based networks, the RIB can be compared to Management Information Base of Simple Network Management Protocol (SNMP) that is also used for the storing objects.

Objects

The *object* is the designation for a structured data that the CDAP is dealing with. Objects are the primary building blocks of the RIB. Two communicating AEs create a shared object space, and they provide access to the portion of the application's RIB. All objects enforce some access rights.

There are two types of RIB objects - passive (that contain static data) and active (which trigger various control events).

RIB Daemon

RIB Daemon (RIBd) is one of the key components of the AP. It is responsible for managing and maintaining objects in the RIB within the DAF. The RIBd monitors all events occurring within the DAF and notifies the subscribers.

AP of a DAF may have several tasks (threads), which share a state via RIB with the help of RIBd. If any task has requirements for information from another participant in a distributed application, it uses RIBd to get the information.

The RIBd accepts subscriptions from tasks. The subscription for an object is a mechanism to manage (read or write) data objects in the RIB. The subscription requests should be time driven, event-driven or direct. The RIBd should process the subscriptions as efficiently as possible.

The main functions of the RIBd within a DAF are:

- notify sets of members about the current value of selected objects;
- monitor events occurring within the DAF;
- provide the RIBd API that can be used by APs sub-tasks;
- respond to requests for information from other members of the DAF;
- maintain a mandatory log of received events;

3.4 DIF Design

Delimiting

The SDU in RINA is a contiguous chunk of data. IPC might fragment the SDU (when passing it down to the (N-1)-DIF) or combine user-data (when passing it up to the (N+1)-DIF). Hence, the operation performed by the Delimiting module (for specification see [3] and [10]) is to delimit the SDU into/from the PDU's user-data

preserving its identity. Employed mechanism indicates the beginning and/or the end of SDUs. Either internal (special pattern) or external (SDU length) delimiting could be used. The delimiting module can perform both fragmentation and concatenation.

Encapsulation/Decapsulation of data messages happens in the RINA components lying in the data path. Figure 5 depicts this process in DIF/DAF together with messages nomenclature.

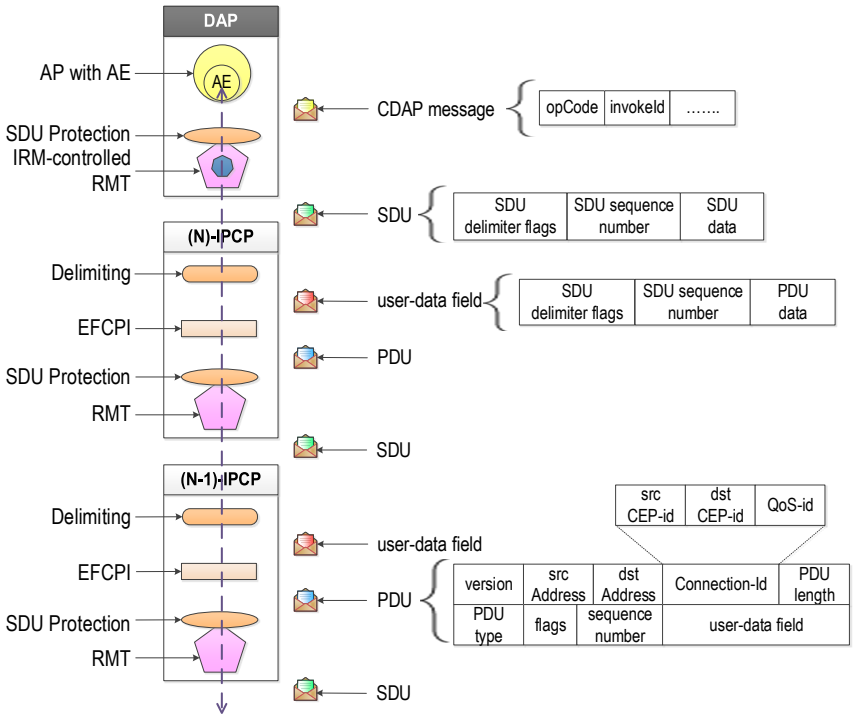

Fig. 5: Message passing between RINA components

Data Transfer with Error and Flow Control

The *Error and Flow Control Protocol (EFCP)* is split into two independent protocol machines coupled and coordinated through a state vector. EFCP guarantees data transfer and data control. The full EFCP functionality is described in [13]. The *Data Transfer Protocol (DTP)* implements mechanisms tightly coupled with the transported SDUs, e.g., reassembly, sequencing. The DTP protocol machine operates on data PDU's fields requiring minimal processing – source/destination addresses, QoS requirements, Connection-id, optionally sequence number or checksum. DTP carries the user-data in the Data-Transfer PDU (DT-PDU).

The *Data Transfer Control Protocol (DTCP)* implements mechanisms that are loosely coupled with the transported SDUs, e.g., (re)transmission control using various acknowledgment schemes and flow control with data-rate limiting. DTCP functionality is based on Delta-t, and DTCP processes control PDUs (ControlPDU). DTCP provides error and flow control over user-data.

There is an *EFCP Instance (EFCPI)* module per every active flow. EFCPI consists of DTP and DTCP submodules. The quality of service demands drive DTCP policies. The DTCP submodule is unnecessary for flows that do not need it, i.e., flows without any requirements for reliability or flow control. The relationship between DTP and DTCP is illustrated in the see Figure 6. Depicted are also data transfer and data control transfer paths. The control traffic stays out of the main data transfer.

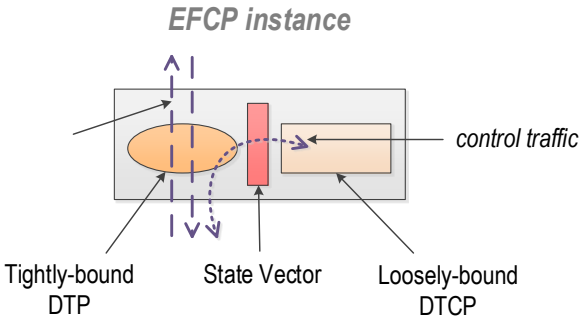

Fig. 6: EFCP instance divided into DTP and DTCP part

Relaying and Multiplexing Task

The *Relaying and Multiplexing Task (RMT)* modules have two main responsibilities – relaying and multiplexing as characterized in [7]. The goal of multiplexing is to pass PDUs from the EFCPIs and RIBd to appropriate (N-1)-flows and reverse of that. Relaying handles incoming PDUs from (N-1)-ports that are not directed to its IPCP and forwards them to other (N-1)-ports using the information provided by its forwarding policy.

RMT Instances in hosts and bottom layers of routers usually perform just the multiplexing task, while RMTs in the top layers of interior/border routers do both multiplexing and relaying. In addition to that, RMTs in top layers of border routers perform flow aggregation.

Each (N-1)-port handled by the RMT has its set of input and output buffers. The number of buffers, their monitoring, their scheduling discipline and classification of traffic into distinct buffers are all matter of policies.

The RMT is a straightforward high-speed component. As such, most of its management (state configuration, forwarding policy input, buffer allocation, and data rate

regulation) is handled by the Resource Allocator, which makes the decisions based on the observed IPC process performance.

SDU Protection

The *SDU Protection* is the last part of the IPCP data path, before an SDU is handed over to an underlying DIF. It is responsible for protecting SDUs from untrusted (N-1)-DIFs by providing mechanisms for lifetime limiting, error checking, data integrity protection and data encryption. The SDU Protection also provides mechanisms for data compression or other two-way manipulations that depend on the (N-1)-flow.

Due to different levels of trust, the SDU Protection handles each (N-1)-flow on its own. This gives us the ability to skip some SDU Protection mechanisms in favor of performance for trusted networks while still being protected from untrusted networks. This is controlled by using different policies that may protect SDU content with the help of integrity checks or encryption.

Flow Allocator

Flow Allocator (FA) as specified in [5] processes allocate/deallocate IPC API calls and further management of all IPCP's flows. FA instantiates a Flow Allocator Instance to manage each flow; FA is a controller/container for all Flow Allocator Instances.

A *Flow Allocator Instance (FAI)* is created upon allocate request call, and it manages a given flow for its whole lifetime. FAI handles creating/deleting EFCPI(s) while managing a single flow's connection. FAI returns port-id to the allocation requestor upon successful allocation as a referencing handle. FAI participates only on port allocation, not on synchronization, which is the responsibility of the EFCPI. The FAI maintains a mapping between flow's local port-id and connection's local CEP-id.

FA contains *Namespace Management (NSM)* interface for assigning and resolving names (including synonyms) within a DIF. This activity involves maintaining the table with entries that map a requested ANI to the IPCP's address.

The *Flow object* contains all information necessary to manage any given flow between communicating parties. It is carried inside create/delete flow request/response messages controlling FA and FAI operation. A Flow object contains: source and destination ANI, source and destination port-ids, connection-id, source and destination address, QoS requirements, a set of policies, access control information, hop-count, current and maximal retries of *create flow requests*.

Resource Allocator

If a DIF has to support different qualities of service, then different flows will have to be allocated to different policies and traffic for them will be treated differently. The *Resource Allocator (RA)* delineated in [4] is a component accomplishing this goal by handling the management of various IPCP resources, namely it:

- controls creating/deleting and enlarging/shrinking of RMT queues;

- modifies EFCPI's DTCP policy parameters;
- controls creating/deleting of (N-1)-flows and their assignment to appropriate RMT queue(s);
- manages QoS classes and their assignment to RMT queue(s);
- manages routing information affecting RMT's relaying or initiates congestion control.

RA maintains a catalog of meters and dials by monitoring various management resources. Each catalog item can be manipulated and shared with other IPC processes within the DIF.

4 Components and Policies

This section provides a general overview of the components design, which includes high-level abstract models of computing systems (like hosts and routers) and also their low-level submodules (like IPCP). In general, a structure of RINASim models follows the structure proposed in the RINA specification. This intentional correspondence enables anyone understanding the RINA specifications to orient easily also in RINASim. Though this structure does not always stand for the most natural representation of the RINA concepts in simulation models, it provides a framework for evaluating properties of the architecture and to identify missing or inaccurate information in the original specification.

The RINA specifications present the proposed network architecture as a generic framework, where mechanisms are intended to perform basic common functionality and policies are defined to select the most appropriate implementation of variable functionality. Rather than providing an exhaustive implementation of policies for each parameterized function, RINASim provides interfaces that are used by the core implementation to call functions defined by the selected policies.

The RINASim policy framework is based on the OMNeT++ NED module interfaces [20], which helps to minimize the need for modifying existing C++/NED source code. Instead of placing a simple module with a policy implementation inside the simulation network graph, a placeholder interface module is used. This design allows the potentially unlimited amount of user policy implementations to be defined and easily switchable via the configuration files (by setting a proper parameter of the encompassing module). Each policy consists of a NED module interface and a base C++ class.

4.1 Nodes

RINASim offers a variety of high-level models simulating the behavior of independent computing system (examples of all three types provided in Fig. 7). These models can be employed to quickly set up simulation experiments. Through parameterization and extension, it is possible to test different deployments and settings. Based on the RINA specifications, we can distinguish between the following node types:

- Host nodes, which represent devices or systems that run distributed applications. These nodes implement the full RINA stack and, also, contain an application process(es). AP instances are configured to communicate with each other to simulate the behavior of an arbitrary RINA application. Currently, there are several predefined host nodes depending on a count of APs and AEs.
- Routers (intermediate nodes), which can be either interior or border. A router is a device that interconnects different underlying DIFs and often does not run user applications. Just as in the RINA specification, there are either interior or border

routers depending on the DIF stack depth (influenced partially also by a count of interfaces).

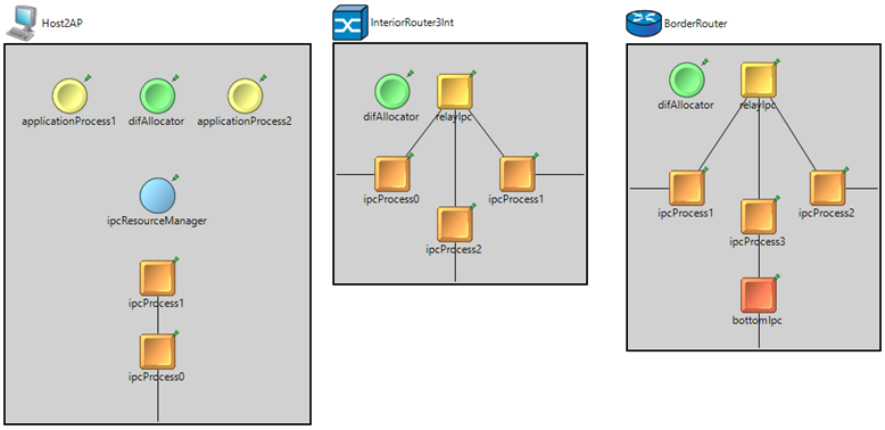

Fig. 7: Examples of RINASim node modules of different types

4.2 DAF Modules

DAF components can be divided into submodules: a) representing IPC endpoints; b) interconnecting APs and available IPCPs; c) discovering APNs. The internal structure of these components and their relationship are described below and depicted in Figure 8.

DIF Allocator

The `difAllocator` module handles locating a destination application based on its name. The DA is a component of the DAP's IPC Management that takes the ANI and access control information and returns a list of DIF-names through which the requested application is available. Moreover, the `difAllocator` module provides statically configured knowledge about the simulation network graph. RINASim's DA overloads any NSM calls.

Submodules

- `da` – Delivers the core functionality of DA;
- `namingInformation` – Provides mapping between APN synonyms;
- `directory` – Provides mapping between APN and DIF-names;
- `searchTable` – Provides mapping between APN and peer DA for authoritative search;
- `neighborTable` – Provides mapping between peer DA and neighboring DA instances.

IPC Resource Manager

The `ipcResourceManager` module currently queries the DA module to find suitable IPCP and relays communication between the AE and the IPCP.

Submodules

- `irm` – Acts as a broker between APs and IPCs and handles AP flow (de)allocation calls;
- `connectionTable` – Maintains the state of the N-1 flows.

Enrollment

The `enrollment` module manages all communication within the Common Application Connection Establishment Phase and during Enrollment phase. It is responsible for managing the states of communication and exchanging initial application objects.
Application Process

The `applicationProcess` module currently handles all application communication from initialization of the first connection to deallocation of all resources. The module acts as an independent unit within the DAF. The `applicationProcess` module also provides statically configured information about its name within the DAF.

Submodules

- `enrollment` – Delivers the core functionality of Enrollment;
- `enrollmentStateTable` – Provides state information about connections.

Submodules

- `apInst` – Spawns a running application;
- `rib` – Provides an interface for object management in the database;
- `ribDaemon` – Handles subscription for objects and manages them within DAF members;
- `enrollment` – Handles initial phases of communication;
- `applicationEntity` – Wrapper for standard AE Instances;
- `aeManagement` – Wrapper for AE Management Instances.

AE Monitor Instance

The `AEMonitor` module is an instance of a specialized AE that mimics a ping-like application.

Submodules

- `iae` – Delivers the core functionality of AE Instance;
- `commonDistributedApplicationProtocol` – Handles all CDAP messages;
- `socket` – Application buffer for read/write operations cooperating with (N-1)-EFCP.

AE Management Instance

The `mgmtae` module is used for handling the management communication. It is mainly used for enrollment and for maintaining RIB updates.

Submodules

- `aemgmt` – Delivers the core functionality;
- `commonDistributedApplicationProtocol` – Handles all CDAP messages;
- `enrollmentNotifier` – Serves as the mediator in communication between `enrollment` and `aemgmt`;
- `socket` – Application buffer for read/write operations cooperating with (N-1)-EFCP.

RIB Daemon

The `ribDaemon` module manages objects in the local RIB repository. It provides an interface for manipulating objects (e.g., read/write/delete) in the RIB.

RIB

The `rib` module acts as a database of application objects. It has an interface for searching, writing, deleting objects. The RIB is primarily accessed by the RIBd that manages these objects within the AP.

Common Distributed Application Protocol

The `commonDistributedApplicationProtocol` module accepts and sends all CDAP messages. Submodules help to differ between types and purposes of CDAP messages and allows their logging.

Submodules

- `cace` – Handles all messages that belong to CACE;
- `auth` – Handles messages that belong to authentication phase;
- `cdap` – Accepts and sends all other messages;
- `cdapSplitter` – Forwards messages to appropriate module;
- `cdapMsgLog` – Provides logging of all messages.

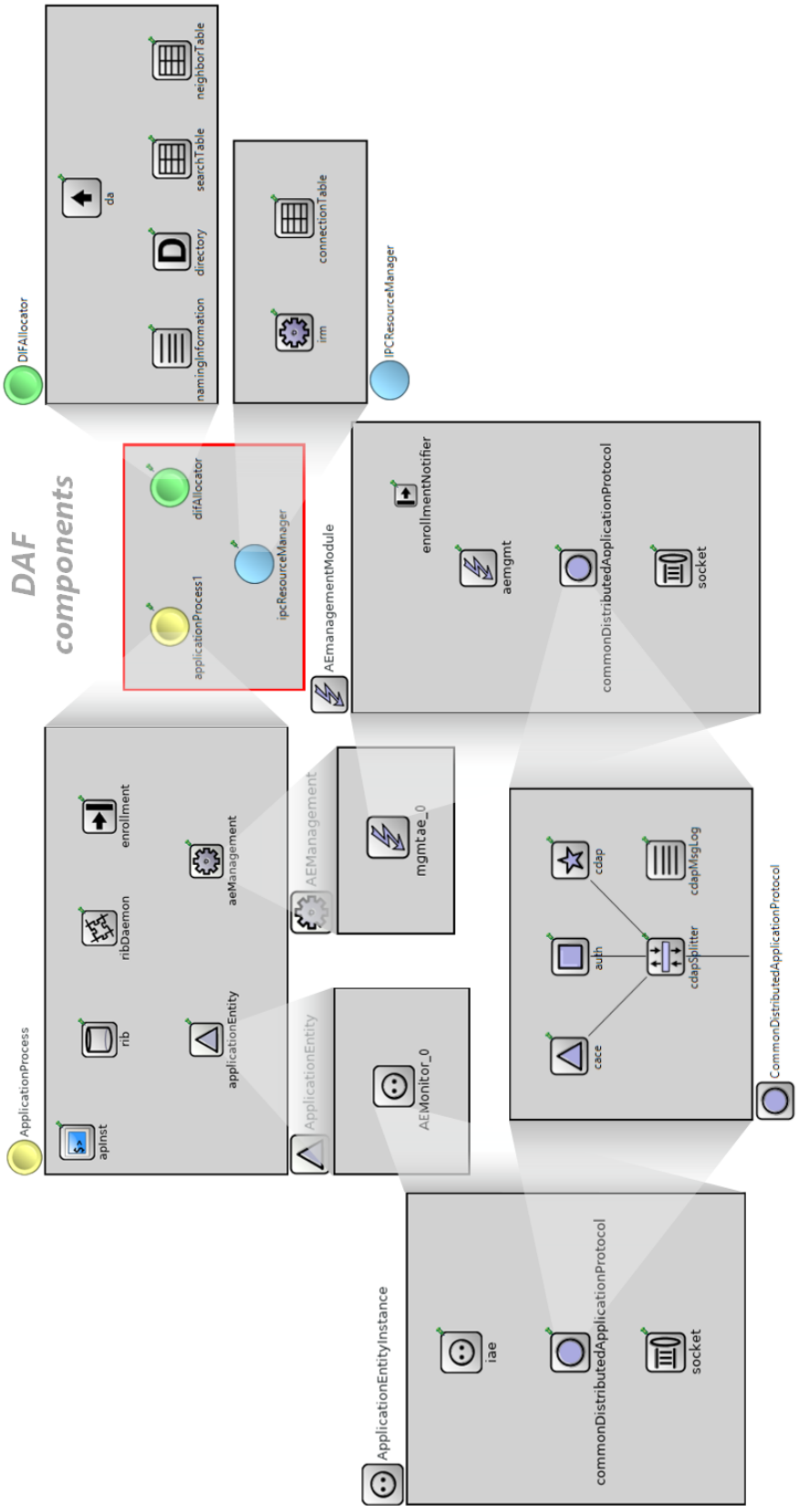

Fig. 8: DAF modules

4.3 DIF Modules

All currently implemented DIF components are enclosed to the `IPCProcess` container module (IPCP instance). The `IPCProcess` contains the following sub-modules, and the overall structure is shown in Figure 9.

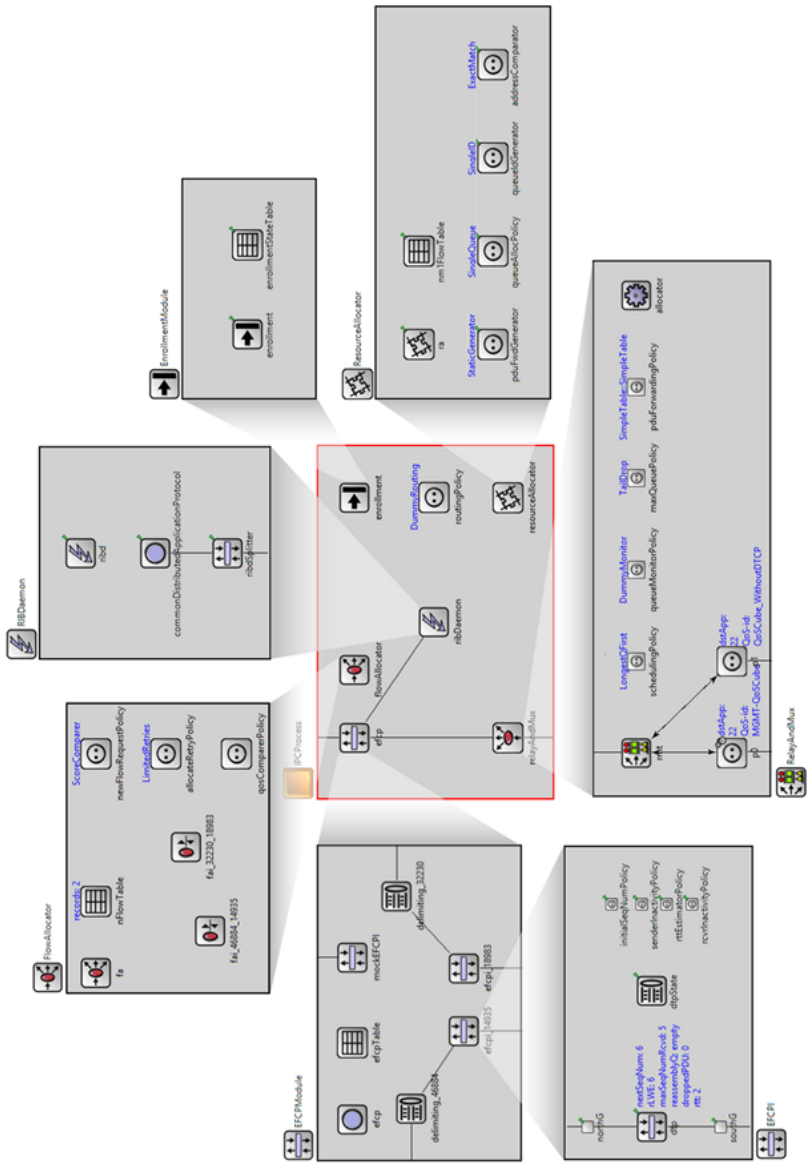

Fig. 9: IPCP's DIF modules

Delimiting

The delimiting module handles SDUs in the form of `SDUData` from the N+1 DIF and produces `UserDataField` for the EFCP instance module. In the opposite direction, it accepts `UserDataField` and produces `SDUData` to the N+1 DIF.

It is capable of fragmenting and concatenating incoming SDUs. Fragmentation is based on `maxPDUsize`. Concatenation takes the incoming SDUs and puts them in single `PDUData` until `maxPDUSize` is met or until the Delimiting Timer expires.

The delimiting module does not contain any submodules. No policies are currently associated with this module.

EFCP Compound Module

The `efcp` compound module handles the data transfer and the associated state vectors. It takes the SDUs from the (N+1)-IPCP or CDAP message from RIBd and creates PDUs.

This module dynamically spawns EFCP Instances. Dynamic modules consist of one Delimiting module (`delimiting_<portId>`) and (possibly) multiple EFCPI modules (`efcp_<cepId>`) per one flow. The EFCPI module itself is also a compound module and contains the static modules `DTP` and `DTPState`. If the flow (QoS demands) requires control, then there are `DTCP` and `DTCPState` modules. It also includes `efcpTable` to store binding between instances of Delimiting and EFCP. Moreover, there is a `mockEFCPI` module containing simplified EFCPI with DTP-like only capabilities for Management flows.

Submodules

There are three static submodules:

- `efcp` – Creates and deletes EFCP instances and Delimiting modules;
- `efcpTable` – Contains bindings between Delimiting and EFCPI (DTP and DTCP);
- `mockEFCPI` – Is simplified EFCPI with DTP-like only capabilities;

Furthermore, it may contain dynamically created pairs of Delimiting and EFCPI modules.

- `delimiting_<portId>` – Handles fragmentation/concatenation;
- `efcpi_<cepId>` – Handles data transfer and control loop functions.

Policies

Policies related to EFCP are specified in DTP and DTCP subsections.

EFCPinstance

An EFCP Instance locally manages the established flow. The `efcpi_<cepId>` module contains `DTP` and `DTPState` submodules. Any necessary policy submodules associated with the flow are part of this compound module as well.

Submodules

- `dtp` – The module implements the Data Transfer Protocol. The `dtp` module accepts user data content from the `Delimiting` module, generates PDUs, and pass them to `relayAndMux`. If necessary, it asks `dtcp` for reliable data transfer. DTP policies are configurable via `config.xml` file by specifying EFCP Policy Set in QoS-cube;
- `dtpState` – The module holds properties related to the actual data transfer. In RINASim, `dtpState` module stores all necessary variables and queues;
- `dtcp` – The module implements Data Transfer Control Protocol. The `dtcp` handles retransmission and flow control related tasks. From the perspective of RINASim, `dtcp` executes policies to update the `dtcpState`. Policies reacts to a situations when error recovery and/or flow control are expected. The current implementation supports retransmission, window-based flow control, allowed gap and A-Time;
- `dtcpState` – Maintains DTCP related variables;
- `northG, southG` – These are pass-through modules that serve for better link visualization.

DTP Policies

The `dtp` module is associated with the following policy types:

- `InitialSeqNumPolicy` – It allows some discretion in selecting the initial sequence number when DRF is going to be sent. (default: Sets the new sequential number to 1);
- `RcvrInactivityPolicy` – If no PDUs arrive in the ewatched period, the receiver should expect a DRF in the next Transfer PDU. Policy represents a timer, which should be set to $2 \cdot (MPL + R + A)$. (default: Resets all receiver-side variables and queues);
- `SenderInactivityPolicy` – Policy represents a timer, which detects long periods of no traffic. It indicates that a DRF should be sent. Δt should be set to $3 \cdot (MPL + R + A)$. (default: Resets all sender-side variables and queues);
- `RTTEstimatorPolicy` – Policy is executed by the sender to estimate the duration of the retransmission timer. This policy is usually based on an estimate of Round-Trip Time (RTT) and received/lost acknowledgements. (default: Computes RTT as an average from the current RTT and the last computed estimate).

RMT

The Relaying and Multiplexing Task represents a stateless function that takes incoming PDUs and relays them within current IPC or passes them to the outgoing port. In particular the RMT takes PDUs from (N-1)-port ids, consults their address fields and performs one of the following actions:

- If the address is not an address (or a synonym) for this IPC Process (which is determined by RA's AddressComparator policy), it consults the PDU Forwarding policy and posts it to the appropriate (N-1)-port(s).
- If the address is one assigned to this IPC Process, the PDU is delivered either to the appropriate EFCP flow or the RIB Daemon (via a mock EFCP instance).
- Outgoing PDUs from EFCP instances or the RIB Daemon are posted to the appropriate (N-1)-port-id(s).

DTCP Policies

- `ECNPolicy` – Handles the ECN bit in incoming DT-PDUs. (default: Sets inner variable based on bit in DT-PDU header);
- `ECNSlowDownPolicy` – Is executed after IPCP's RA receives Congestion Notification. (default: No action);
- `LostControlPDUPolicy` – Determines what action to take when the protocol machine detects that a control PDU (Ack or Flow Control) may have been lost. (default: Sends `ControlAck` and empty DT-PDU);
- `NoOverridePeakPolicy` – Allows rate-based flow control to exceed its nominal rate for presumably short period of time. (default: Puts DT-PDU on `ClosedWindowQ`);
- `NoRateSlowDownPolicy` – Is used to lower momentarily the send rate below the allowed rate. (default: No action);
- `RateReductionPolicy` – Is executed in case of rate-based flow control. When local shortage of buffers occurs or when buffers are less full than a given threshold, policy increases the rate agreed during the connection establishment. (default: Slows down by 10% if buffers are getting clogged);
- `RcvFCOverrunPolicy` – Determines what action to take if the receiver receives PDUs, but the credit or rate has been exceeded. (default: Drops the PDU and sends control PDU as response);
- `RcvrAckPolicy` – Is executed by the receiver of the DT-PDU and provides some discretion in the action taken. (default: Either acknowledges immediately or starts the A-Timer and acknowledges the `RcvLeftWindowEdge` when it expires);
- `RcvrControlAckPolicy` – Is executed upon reception of `ControlAckPDU`. (default: Checks the values and if necessary sends back control PDU with updated information);
- `RcvrFCPolicy` – Policy is invoked when a DT-PDU is received to give the receiving protocol machine an opportunity to update the flow control allocations. (default: Increments receiver's right window edge);
- `ReceivingFCPolicy` – Is invoked by the receiver of a DT-PDU in case there is a flow control present, but no retransmission control. (default: Sends `FlowControlPDU`).
- `ReconcileFCPolicy` – Is invoked when both credit and rate-based flow control are in use, and they disagree on whether the protocol machine can send or receive data. If this is the case, then the protocol machine can send or receive; otherwise, it cannot;
- `RxTimerExpiryPolicy` – Is executed by the sender when a retransmission timer expires. This policy must be run in less than the maximum time to Ack. (default: Retransmits DT-PDU with `seqNum` equal to the one in `RXTimer`);
- `SenderAckPolicy` – Is executed by the sender when PDUs might be deleted from the retransmission queue. It is useful for multicast-like use-cases, when it is feasable to delay discardation of PDUs from the retransmission queue. (default: Removes DT-PDU from retransmission queue up to Acked sequence number);
- `SenderAckListPolicy` – Is executed by the sender when PDUs might be deleted from the retransmission queue. The policy is used in conjunction with the selective acknowledgement aspects. It is useful for multicast transfers just like the previous policy. (default: Removes specified `seqNum` ranges from retransmission queue);
- `SendingAckPolicy` – Is executed upon A-Timer expiration in case there is DTCP present. (default: Updates receiver's left window edge and sends Ack/FlowControlPDU;
- `SndFCOverrunPolicy` – Determines what action to take if the sender has DT-PDU ready for dispatch but values of `SndRightWindowEdge` or `SndRate` are blocking them. (default: Puts DT-PDU in `ClosedWindowQueue`);
- `TxControlPolicy` – Is used when there are conditions that warrant sending fewer PDUs than allowed by the sliding window flow control. (default: Puts as many DT-PDUs from `generatedPDUQ` to `postablePDUQ`).

In RINASim, all functionality of the RMT including a policy architecture is encompassed in a single compound module named `relayAndMux`, which is present in every IPC process.

Submodules

- `relayAndMux` – Consists of multiple simple modules of various types, some of which are instantiated dynamically at runtime;
- Static modules:
  - `rmt` – Implement fundamental RMT logic that decides what should be done with messages passing through the module;
  - `allocator` – A manager unit for dynamic modules that provides an API for adding, deleting and reconfiguration of RMT ports and queues;
- Dynamic modules:
  - `RMTPort` (encompassed in `RMTPortWrapper`) – A single endpoint of an (N-1)-flow;
  - `RMTQueue` – A representation of either an input or an output queue. The number of `RMTQueues` per (N-1)-port is a matter of RA policies;
  - `sdup` – Performs SDU protection.

Policies

RMT provides following policies:

- `schedulingPolicy` – Is invoked each time some (N-1)-port has data to send. The policy employs the algorithm to make a decision about which of port's queues should be handled first;
- `queueMonitorPolicy` – Is a stateful policy that manages variables used by other RMT policies. It is invoked by various events happening inside RMT and its ports and queues;
- `maxQueuePolicy` – Is a policy used for deciding what to do when queue lengths are overflowing their threshold lengths. It is invoked whenever the size of items in a queue reaches a threshold;
- `pduForwardingPolicy` – Is a policy deciding where to forward a PDU. It accepts the PDU as an argument, does a lookup in its internal structures (usually a forwarding table populated by the PDUFG policy) and returns a set of (N-1)-ports.

Routing

The Routing module is a policy that serves for exchanging information with other IPCPs in DIF in order to generate a set of routing information. It indirectly provides input for populating the RMT `PDUForwardingPolicy`.

Routing policies are used to propagate information about routing in the DIF and are dependent on PDU Forwarding Generator (PDUFG).

Policies

Consist of a single replaceable policy that conducts routing within the DIF. There are examples of policies (e.g., `DummyRouting`, `DomainRouting`, `SimpleRouting`) leveraging distance-vector, link-state or native RINA approaches for routing PDUs.

Flow Allocator

The `flowAllocator` module handles (de)allocation request and response calls from the `IRM`, `RIBDaemon`, `DAFEnrollment` or `AE`. FA itself is structured into the flow table and flow management modules.

Submodules

- `fa` – Provides the core functionality involving instantiation of FAIs;
- `nFlowTable` – Maintains mappings between (N)-flow and bound FAI;
- `fai_<portId>_<CEPid>` – Manages the whole flow lifecycle.

Policies

- `allocateRetryPolicy` – Occurs whenever initiating FAI receives negative create flow response. It allows FAI to reformulate the request and/or to recover properly from failure;
- `qosComparePolicy` – Checks if existing (N-1)-flows can be used to support an (N)-flow;
- `newFlowRequestPolicy` – Is invoked after FAI's instantiation. Policy subtasks involve both 1) evaluation of access control rights; and 2) translation of QoS requirements specified in allocate request to appropriate RA's QoS-cubes.

Resource Allocator

The `resourceAllocator` is one of the most important components of an IPC Process. It monitors the operation of the IPC Process and makes adjustments to its operation to keep it within the specified operational range.

Submodules

- `ra` – Provides core functionality managing RMT and connections to other IPCPs via (N-1)-flows;
- `nm1FlowTable` – Maintains a table containing information about the active (N-1)-flows;

Policies

- `pduFwdGenerator` – A policy, which manages the RMT's PDU Forwarding policy;
- `queueAllocPolicy` – A policy handling RMT queue allocation strategy;
- `queueIdGenerator` – A policy generating queue IDs from Flow information and PDUs;
- `addressComparator` – A policy matching PDU address and IPCP address.

5 Demonstrations

This section outlines some of the scenarios where RINA is employed as the native network stack. General instructions (how to setup and run the examples) are provided to the reader. Furthermore, a detailed description tries to reveal advantages of adopting RINA for certain Internet use-cases. Demonstration source codes are located in `/examples/` folder, and each one includes following files (which may be reused as templates when creating other RINASim scenarios):

- `<name>.ned` – OMNeT++ simulation network, which contains definitions of nodes and their interconnections;
- `omnetpp.ini` – The scheduled setup including model configurations (e.g., node addresses, ANI for AEs, references to XML configuration) applied during initialization of scenario;
- `config.xml` – The file contains more complex / structured configurations (e.g., DA's mappings, RA's QoS-cubes, pre-allocation and pre-enrollment settings) in the form of XML data, which are mostly applied during initialization;
- `*.anf` – The file(s) is describing which statistics should be collected and evaluated during simulation run;
- `./results/*` – Scalar/vector results of various simulation runs.

5.1 Demo Network

Simulation source codes relevant for this scenario are located in the folder `/examples/Demos/UseCase5`.

The motivation behind this particular simulation is to show a ping-like application within the simple network consisting of all different node types. The topology contains two host nodes (called `HostA` and `HostB`), two border routers (called `BorderRouterA` and `BorderRouterB`) and one interior router (identified as `InteriorRouter`) interconnected together.

There are totally six *named* DIFs of three different ranks (the network is depicted in Figure 10). Please notice the addressing scheme where the same node may use the same address on a different DIF as long as this address is unambiguous within the layer's scope. The RINA address length and syntax is policy-dependent (comparing to IP or MAC). The demonstration uses a flat address space with simple strings as addresses. The addresses are mentioned in blue color to highlight them in the text.

- Top most *TopLayer* DIF is common to `HostA` (with address *hA*), `BorderRouterA` (address *rA* and self-enrolled), `BorderRouterB` (address *rB*) and `HostB` (*hB*). Self-enrolled APs/IPCPs in RINASim are ones that are automatically members of some DAF/DIF and they are skipping Enrollment phase during any communication.
- Three middle DIFs *MediumLayerA*, *MediumLayerAB* and *MediumLayerB*. *MediumLayerA* is common to `HostA` (*ha*) and `BorderRouterA` (address *ra*

and self-enrolled). *MediumLayerAB* is common to `BorderRouterA` (*rA*), `InteriorRouter` (address *rC* and self-enrolled) and `BorderRouterB` (*rB*). *MediumLayerB* is common to `BorderRouterB` (address *rb* and self-enrolled) and `HostB` (*hb*).

- Two bottom most DIFs *BottomLayerA* and *BottomLayerB*. *BottomLayerA* is common to `BorderRouterA` (*ra*) and `InteriorRouter` (address *rc* and self-enrolled). *BottomLayerB* is common to `InteriorRouter` (address *rc* and self-enrolled) and `BorderRouterB` (*rb*).

By default, every RA contains an implicit QoS-cube (with QoS-id *MGMT-QoSCube*) that defines QoS parameters (e.g., reliability, minimum bandwidth) for management traffic and guarantees successful mapping of management SDUs onto the appropriate (N)-flow. Apart from this default QoS-cube, each RA loads the QoS-cube set according to the simulation configuration. For demonstration, there are two more QoS-cubes available for each RA called *QoSCube-RELIABLE* and *QoSCube-UNRELIABLE* (with the same QoS parameters differing only in data transfer reliability).

The DA implementation currently allows only the static change of its settings (namely different kinds of mappings). Hence, the necessary configuration step is to initialize the DA properly in order to provide services to FA, RA and other components depending on naming information. Namely, two DA's tables are important for overall functionality – `Directory` (helps to search target IPCP for a given APN) and `NeighborTable` (used by FA to find a neighbor IPCP for a given IPCP).

The simulation description is divided into two phases:

1. Enrollment Phase – If another IPCP wants to communicate within a given DIF, then, it needs to be enrolled by a DIF member. Self-enrolled IPCPs are members of certain DIFs from the beginning of the simulation, and they help other IPCPs to join a DIF. In order to allow communication between any node, the simulation is scheduled to commence enrollment of: `BorderRouterA` into *BottomLayerA* at $t = 1s$; `BorderRouterA` into *MediumLayerAB* at $t = 1.5s$; `BorderRouterB` into *TopLayer* at $t = 2s$; and `HostB` into *TopLayer* at $t = 5s$. The enrollment usually involves recursive calls of enrollment procedures in lower ranked DIFs.
2. Data Transfer Phase – The IPC comprises of flow allocation, data transfer, and optional flow deallocation. `HostA` and `HostB` are configured to exchange messages via a ping-like protocol (measuring one-way and round-trip delays). In this case, flow allocation is initiated at $t = 10s$, first ping is sent at $t = 15s$ and flow deallocation occurs at $t = 20s$.

All steps related to the Enrollment Phase are described on example of application connection in Section 5.2. For now, let us skip Enrollment Phase by stating that except `HostA` all other DIF/DAF members were successfully enrolled. All flows created during Enrollment Phase carry only CACE messages (for connection establishment), and they are intended for direct RIBd-to-RIBd communication employing various management messages. Thus, these flows are called *management flows*.

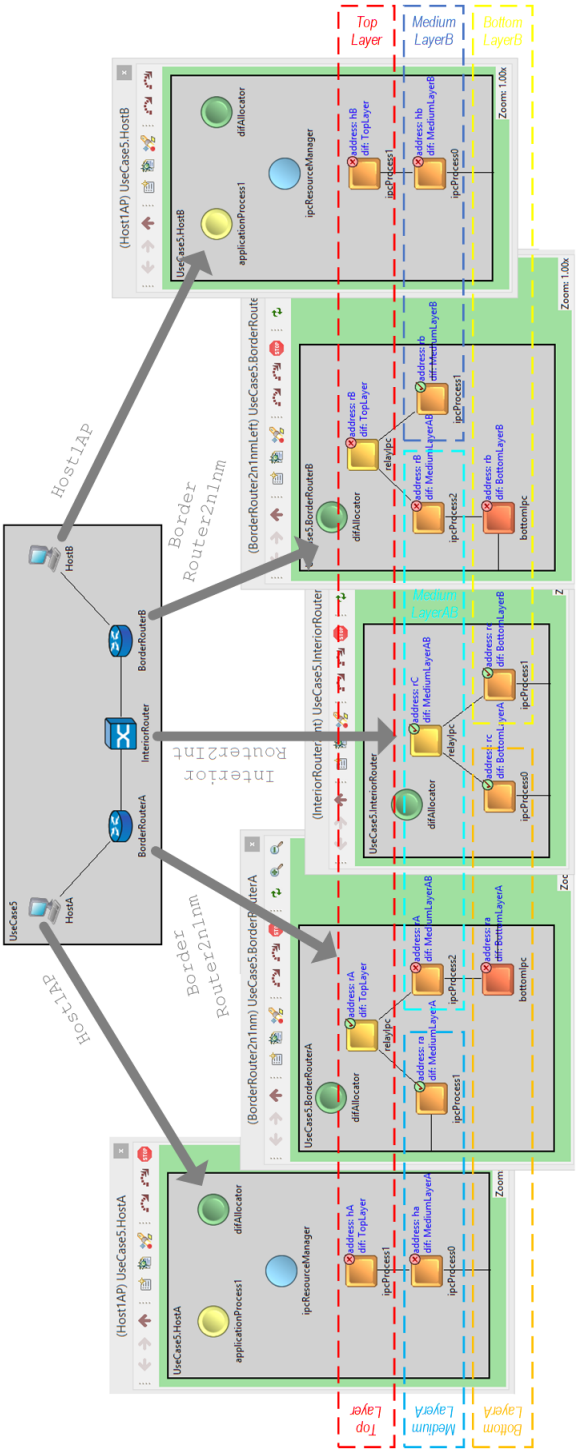

Fig. 10: Demo network diagram

The main outcome of this scenario is a simulation of data transfer events between ping-like applications (`APPing`) run on `HostA` and `HostB`. This ping-like protocol sends probe request (M_READ) from `HostA` to `HostB`, where `HostB` replies with the response (M_READ_R). One-way and round-trip time delays are measured according to timestamp differences of these messages.

The Data flow allocation starts at $t = 10s$. `HostA`'s `applicationProcess1` requests flow for communication with `HostB`'s `applicationProcess1`. The event goes through following set of steps:

#1) *Allocate request* is delivered to IRM. Over there, DA is asked to resolve the destination ANI into an IPC address within certain DIF available to `HostA`. The following result is returned yielding that destination is reachable via IPCP *hB* in *TopLayer*;

#2) `HostA` can access *TopLayer* leveraging `ipcProcess1`. Hence, IRM delegates a *allocate request call* to `ipcProcess1`'s FA. FA instantiates EFCPI and verifies whether IPCP is enrolled into DIF before any attempt for sending *create request flow*. A couple of enrollment events recursively repats: a) enrollment of `HostA`'s `ipcProcess0` into *MediumLayerA* by `BorderRouterA`; b) creation of management flow between IPCP *ha* and IPCP *ra* within *MediumLayerA*; c) enrollment of `HostA`'s `ipcProcess1` into *TopLayer* by `BorderRouterA`;

#3) After successful enrollment of `ipcProcess1`, FA may continue with the flow allocation. FA exchanges *create request/respond flow* with `HostB`. This includes the creation of an (N-1)-flow between *ha* and *ra* in *MediumLayerA* and creation of the(N)-flow between *hA* and *hB* in *TopLayer*. It gets more complex in *TopLayer* because M_CREATE and M_CREATE_R messages must be relayed by the border routers to reach `HostB`, which causes additional recursive flow allocations between interim devices (i.e., *BorderRouterA*, *InteriorRouter*, *BorderRouterB*). All interim devices are already enrolled into their DIFs, thus the established flows serve as carriers for `HostA` and `HostB` data transfer;

#4) M_CREATE from `HostA` to `HostB` is received by `BorderRouterA`'s `relayIpc`. `BorderRouterA` inspects *create request flow* and determines `BorderRouterB` with the help of DA as the next-hop. Because border routers are not directly connected, they can communicate via `InteriorRouter` as a proxy. Therefore, `BorderRouterA` establishes flow between *ra* and *rc* of *BottomLayerA* and sends a *create request flow* in *MediumLayerAB*;

#5) M_CREATE from `BorderRouterA` to `BorderRouterB` is received by `InteriorRouter`'s `relayIpc`. The message needs to be relayed to `BorderRouterB`. Hence, the flow is created between *rc* and *rb* in *BottomLayerB*. Then, *create request flow* is forwarded within this DIF;

#6) M_CREATE from `BorderRouterA` to `BorderRouterB` within *MediumLayerAB* is received by `BorderRouterB`'s `ipcProcess2`. `BorderRouterB` accepts the flow and sends *create respond flow* that travels back to `BorderRouterA`. Because the flow connecting both border routers (*rA* and *rB* within *MediumLayerAB*) is established, flow allocation from #4 may continue;

#7) M_CREATE from HostA to HostB is received by BorderRouterB's relayIpc after passing through flows the created during #5 and #6. *BorderRouterB* inspects *create request flow* and determines that HostB is reachable via its *MediumLayerB*. In order to successfully relay M_CREATE to its final destination, BorderRouterB allocates the flow between *rb* and *hb* in *MediumLayerB*. Subsequently, M_CREATE is forwarded to HostB;
#8) M_CREATE is received by HostB's ipcProcess1. The FA notifies applicationProcess1 about the current flow allocation. applicationProcess1 accepts flow for data transfer between APs. The decision is returned to ipcProcess1's FA. The IRM is asked to create bindings between AP and IPCP. The FA instantiates the EFCPI, updates the Flow object and replies back to the requestor with M_CREATE_R;
#9) M_CREATE_R is relayed via all flows formed during #4-#7 to HostA until ipcProcess1's FA receives this message. The FA updates the Flow object and notifies applicationProcess1 about successful flow allocation. Then the IRM adds the missing bindings, and the whole data path between HostA and HostB is ready. The (N)-flow in *TopLayer* can carry data traffic between AEs with the help of all underlying flows.

The next event is a transfer of data traffic between AEs. HostA sends five ping-like probing objects inside M_READ message starting at $t = 15s$. Upon reception of these messages, HostB replies with probe response (in form of dedicated M_READ_R message). Data path and flows are depicted in Figure 11 with different colors acompanied by the following description. The event consists of five repetitions of these two steps:

#1) HostA's applicationProcess1 sends a M_READ message, which is passed through IRM into ipcProcess1 to flow prepared during the previous event and descends to ipcProcess0. The message travels through the medium and flow connecting HostA with BorderRouterA within *MediumLayerA*, where it is received by ipcProcess1. It is relayed by BorderRouterA's relayIpc to ipcProcess2 and flow interconnecting BorderRouterA and BorderRouterB in *MediumLayerAB*. Because border routers are not directly connected, the message is passed to a lower bottomIpc into flow interconnecting BorderRouterA with the neighboring InteriorRouter in *BottomLayerA*. Message traverses through the medium, and it reaches InteriorRouter's ipcProcess0. Over there, the message ascends to relayIpc, where it is relayed within *MediumLayerAB*. Then it descends to ipcProcess1 into flow interconnecting InteriorRouter and BorderRouterB in *BottomLayerB*. The message travels through medium to BorderRouterB's bottomIpc. It ascends to ipcProcess2 and is relayed by relayIpc to ipcProcess1. Finally, the message reaches HostB's ipcProcess0 through medium inside flow within *MediumLayerB*. It ascends to the flow in ipcProcess1 (member of *TopLayerB*) and through IRM to HostB's applicationProcess1 as the recipient;

#2) `HostB`'s `applicationProcess1` responds with M_READ_R message that returns to `HostA` traveling in opposite direction through the same data (marked with violet line) path as in #1. Refering to Figure 11, the message is either encapsulated (from `HostA` to `HostB` green circles) or decapsulated (from `HostA` to `HostB` orange circles) into/from PDU or relayed (brown circles) depending on direction.

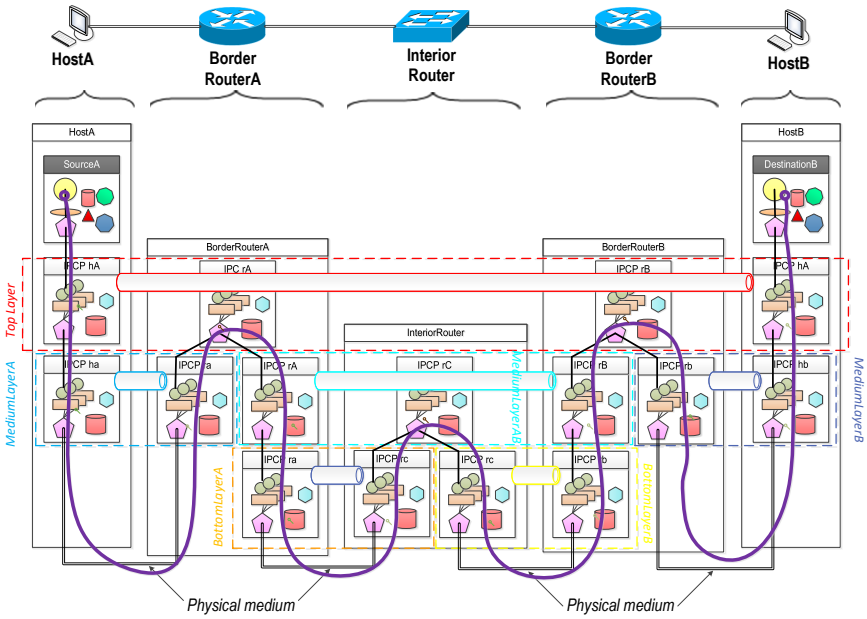

Fig. 11: Data transfer path illustration for Demo network

After the APs exchanged pings, `HostA`'s AE closes the connection and sends *deallocate submit* to `HostB` at $t = 20s$. Deallocation affects only the flow present in the *TopLayer*. The current RINASim implementation leaves the underlying (N-1/2)-flows (i.e., those not directly connected with APs) intact because they may be reused later by other applications. This event is accompanied by following steps:

#1) `HostA`'s `applicationProcess1` tells IRM to deliver a *deallocate submit*. IRM unbounds the port from its side. Then, the IRM delegates *flow deallocation* to `ipcProcess1`'s FA;
#2) This FA generates a M_DELETE message with an updated `Flow` object state inside and sends the object towards `HostB` through the flow in *TopLayer*. The message follows the data path leveraging existing management flows created during the Enrollment Phase;

#3) HostB's ipcProcess1 receives M_DELETE. The FA updates its version of the Flow object. FA delivers *deallocation submit* to HostB's applicationProcess1, which tells the IRM to remove bindings;
#4) ipcProcess1's FA on HostB then replies with M_DELETE_R acknowledging the successful flow deallocation. This message is carried back to HostA;
#5) HostA's ipcProcess1 receives M_DELETE_R. FA marks the flow as deallocated and disconnects remaining bindings between IPCP and IRM.

The result of flow (de)allocation and flow's state is maintained in ipcProcess1's NFlowTable of HostA and HostB.

5.2 Simple application

Simulation source codes relevant for this scenario are located in the folder /examples/Demos/UseCase2.

There is only one *named* DIF Layer0 connecting hostA and hostB. RINA address length and syntax is policy-dependent (comparing to IP or MAC). The demonstration uses flat address space with simple string as addresses. The IPCP at hostA has the address 1 and IPCP at hostB has the address 2. The application on hostA has the APN SourceA and the destination application on hostB has conveniently the APN DestinationB.

Every application starts with the creation of an application connection to an AP that is part of a DAF. This process contains several events such as AE Instance creation, flow allocation, management connection establishment, and enrollment that need to proceed before the application connection is created. The application goes through these steps to be able to communicate within the DAF:

#1) In the HostA's applicationProcess1, the request for a connection to HostB's applicationProcess1 is induced via an API call. The HostA's applicationProcess1 is not a member of a DAF yet, so it needs to create a management connection to the destination AP first. The HostA's applicationProcess1 spawns a management AEI inside AEManagement that is used to handle the management communication between APs in a DAF. The flow to the destination AP is allocated. HostB's applicationProcess1 spawns the corresponding management AEI when it accepts flow allocation. The connection is successfully allocated, and CACE between the APs may proceed;
#2) HostA's management AEI sends a M_CONNECT request carrying the authentication information. The HostB's management AEI receives the M_CONNECT, validates the authentication data, and sends the M_CONNECT_R back. The application connection is successfully established upon reception of the positive response (the CACE phase proceeded successfully);
#3) The next phase of communication is the Enrollment. The HostA's applicationProcess1 sends M_START message containing object(s) re-

lated to the enrollment. The `HostB`'s `applicationProcess1` replies with M_START_R. Subsequent exchange of M_CREATE / M_CREATE_R initiated by `HostB`'s `applicationProcess1` (that is a member of the DAF) may proceed. These messages should synchronize essential objects in RIBs, which would allow `HostA`'s `applicationProcess1` to operate as a full-fledged member of the DAF. RINASim has placeholders to leverage this synchronization property of enrollment;

#4) When the necessary objects are created, a M_STOP message is generated by the `HostB`'s AP. `HostA`'s `applicationProcess1` sends M_STOP_R indicating that no more messages need to be exchanged. Otherwise, the `HostA`'s `applicationProcess1` might send several M_READs to obtain information from the member AP (i.e., `HostB`). Then the `HostA`'s `applicationProcess1` is a new member of the DAF.

Described Enrollment Phase is the enrollment applied on DAF membership. Enrollment within DIF follows the same message order and states except for the data in the objects that are exchanged. In case of DIF, RIBd processes above-mentioned messages directly instead of AEs (because AEs are not components of any DIF).

Once the management connection is established, and Enrollment Phase finished, the AP starts to operate as a full-fledged member of a DAF. The simple ping-like application follows these steps:

#1) `HostA`'s `applicalicationProcess1` spawns the instance of `AEMonitor` (which is a specialized AE offering ping-like behavior). The `AEMonitor` instance requests the underlying IPCP (i.e., *ipcProcess1*) for the flow with a specified application QoS. The flow is then allocated (See Section 5.1 for more details);

#2) `HostB`'s `applicationProcess1` instantiates the `AEMonitor` in response to positive allocation of the requested flow. The `HostA`'s `applicationProcess1` generates a M_CONNECT carrying the authentication information and `HostB`'s `applicationProcess1` sends a M_CONNECT_R with a positive response back. This establishes the application connection (CACE phase is completed), and the data objects may be transferred over this connection;

#3) Generally, AEs support two types of communication - either via the RIB or direct messaging omitting the RIB completely. Our ping-like application uses the direct messaging to keep the demonstration as simple as possible. The `HostA`'s `applicationProcess1` sends a M_READ message request with a specific object. The request is received by `HostB`'s `applicationProcess1` (namely by an instance of `AEMonitor`), which replies with M_READ_R;

#4) The `HostA`'s `applicationProcess1` receives the M_READ_R. The application then displays a result to a user (i.e., console text). Another round of M_READ/M_READ_R exchange follows later. The object inside these messages contains just an integer value that is incremented by each host.

This simple ping-like application shows a direct communication between two APs. The communication in real ping should be the same.

5.3 Reliable Data Transfer

Simulation source codes are same as in the case of Section 5.2 and they are in folder `/examples/Demos/UseCase2`.

In this scenario, we demonstrate reliable data transfer with a detailed description of EFCP events. We use a minimal topology with just two hosts (see Figure 12), each one with a single IPCP for a sake of simplicity. We omit the description of events prior to EFCP instance creation such as Enrollment Phase or flow allocation.

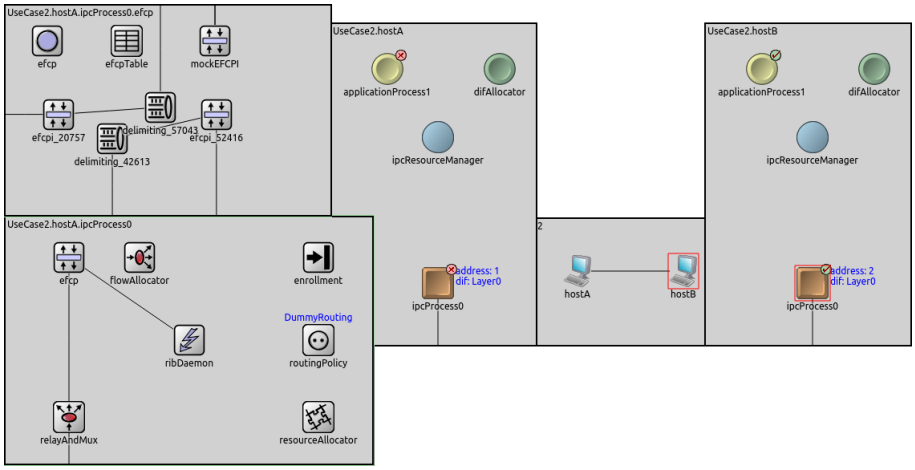

Fig. 12: Reliable data transfer illustration of two directly connected hosts

We demonstrate the reliable data transfer using a ping-like application between `hostA` and `hostB` (see Figure 12). The transfer over the link between `hostA` and `hostB` is delayed by 0.4 s in both directions. As described previously, the ping-like application exchanges ping request and response (i.e., M_READ and M_READ_R messages containing appropriate object referencing time). Both messages are sent inside the `DataTransferPDU` and acknowledged by the `ControlPDU` type. The EFCP connection is configured with the default policy set. Both flow control (i.e., speed administration) and retransmission control (i.e., acknowledgements and retransmits guarantee that nothing gets lost) are active.

At time $t = 5s$ the `applicationProcess1` on `hostA` initiates the communication. The Data Transfer starts after the successful Enrollment Phase and he flow allocation at $t = 14.8s$. The reliable data transfer includes following notable steps:

#1) The module `delimiting_57043` encapsulates the incoming SDU (i.e., *ping request* in form of M_READ) into `PDUData`, fragments if necessary, and forwards it to the EFCP instance `efcpi_20757` as `UserDataField`. In the EFCP instance it is handled by the `dtp` module. The `dtp` module initialize `dtpState` upon `UserDataField` receipt.

The UserDataField is encapsulated in the DataTransferPDU with a DT-PDU header. The SequenceNumber is set to the value produced by initialSeqNumPolicy which generates a random number by default. For simplicity, we use 1 as the initial sequence number in our example. The first PDU is labeled with the Data Run Flag (DRF) indicating the start of a new connection. After the complete PDU have been generated, the control is handed over to the DTCP which tries to send it. With the window-based flow control active (with the maximum windows size of 10 PDUs), dtcp checks if the Sender's Right Window Edge (SRWE) is bigger then the sequence number of the DT-PDU it is trying to send. The initial value of SRWE is set to the sum of nextSeqNum and intialSenderCredit.

All DT-PDUs that satisfy the flow control check are put in the postablePDUs queue. Then DTP iterates over the set of postablePDUs and sends them to RMT. Simultaneously, DTP also puts their copies into the retransmissionQ queue, and starts the RetransmissionTimer. The timer is initiated with a default RTT value of the first PDU in the connection. The RTT value is adjusted during the connection lifetime using ControlPDUs and the RTTEstimatorPolicy. After passing the PDU to the RMT, the SenderInactivityTimer is started.

#2) At time $t = 15.2s$, the DataTransferPDU is delivered to the EFCP instance in the ipcProcess0 on hostB. Since the PDU has the DRF set, it indicates the (re)start of a (new) connection. In the case of a restart, DTP dequeues as many PDUs from reassemblyQ to the delimiting module and deletes the rest. Subsequently, DTP sets the Receiver's Left Window Edge (RLWE) to the seqNum of the incoming PDU (in our example to 1) and adds the PDU to the reassemblyQ. Simultaneously, DTP updates the state vector. The SVUpdate initiates RcvrAckPolicy which sends an AckFlowPDU as confirmation of the DT-PDU (the one with seqNum = 1). The acknowledgement received by hostA updates the SRWE. Then, the UserDataField is passed on hostB to the delimiting module and the rcvrInactivityTimer is started.

#3) The AP on hostB sends back a *ping response* (i.e., M_READ_R) message. The same set of steps (see #1) related to the first DT-PDU applies to hostB's EFCPI as in case of hostA's corresponding EFCPI - EFCP generates the initial sequence number, sets the DRF and starts senderInactivityTimer.

#4) At time $t = 15.6s$, the EFCP on hostA receives the AckFlowPDU. It trigger both RTTEstimatorPolicy (updating the RTT value) and SenderAckPolicy. The SenderAckPolicy removes a subset of DT-PDUs from retransmissionQ with the sequence numbers up to the value in the AckFlowPDU.

At the same time, hostA also processes the DT-PDU with the ping response message. The same steps as in #2 are repeated on hostA.

#5) At time $t = 15.6s$, hostA sends a new *ping request*. The EFCP fills the header with sequenceNumber = 2, puts the PDU in retransmissioQ with associated RetransmissionTimer, and restarts the SenderInactiveTimer.

#6) At time $t = 16s$, `hostB` receives an acknowledgement of the last *ping response*. As a result, the DT-PDU with `sequenceNumber` = 1 is removed from `retransmissionQ`.
#7) Simultaneously to #6 at the time $t = 16s$, `ipcProcess0` on `hostB` receives DT-PDU with the second *ping request*, and sends back an acknowledgement. The process starts to repeat.

The summary of both state vectors of the hosts is shown in Table 1 with all sliding windows including Receiver's Right Window Edge (RRWE) and Sender's Left Window Edge (SLWE), which are counterparts of RLWE and SRWE. The values indicate the state after the corresponding event.

Variable	**Value**			
Event	#1	#4	#5	#6
Time	14.8 s	15.6 s	15.6 s	16 s
Next SeqNum	2	2	2	3
RLWE	0	0	1	1
RRWE	∞	∞	11	11
SLWE	0	2	2	2
SRWE	10	11	11	11
retransmissionQ	1	∅	∅	2

Variable	**Value**			
Event	#2	#3	#7	#8
Time	15.2 s	15.2 s	16 s	16 s
Next SeqNum	1	2	2	2
RLWE	1	1	1	2
RRWE	11	11	11	12
SLWE	0	2	2	2
SRWE	10	10	11	11
retransmissionQ	∅	1	∅	∅

Table 1: Values of the hostA and the hostB EFCP

Both state vectors continue to evolve in a similar fashion throughout the rest of the communication. After the last `DataTransferPDU` is sent, the inactivity timers are restarted one last time. After they expire, the State-Vectors are deallocated.

The events are mirrored on both hosts, because the application on top is sending data in both directions. The EFCP instance on both sides uses its receiver and sender state vector and finite state machine.

6 Conclusion

In this chapter, we described the core RINA principles. We summarized the RINA theory in the text that lacks any forward references. We know how hard the "mental shift" from TCP/IP concepts towards RINA is, thus we urge a reader to follow the citations in order to learn more. We described different kinds of high-level RINA nodes including hosts, interior routers, and border routers. Subsequently, we dived deep into the low-level RINA components that are being used by DIF and DAF.

RINASim at its current state represents an entirely working implementation of the simulation environment for RINA. The simulator contains all mechanisms of RINA according to the current specification. The RINASim philosophy benefits from the clever OMNeT++ module interfacing, which allows flexible change of employed policies. Furthermore, the chapter also contained a detailed illustration of the RINA

principles using three RINASim scenarios. The demonstration description show the impact of recursion and help others to understand the enrollment and the flow (de)allocation procedures in praxis. The demonstration setup may be employed as the template when creating new scenarios.

The main motivation behind the development of the RINASim is that it should allow:

- researchers to prototype and test new policies and mechanisms in a native and full-compliant RINA environment - *scientific goal*;
- others to visualize and understand the RINA principles - *educational goal*.

RINASim (first revealed in [26]) started as one of FP7 EU PRISTINE deliverables and continues beyond the end of the project. However, RINASim is just one of the independent implementations (see Section 7.2 for more) of the RINA concepts. RINASim is the open environment that can be extended with experimental features. The simulator helps to evaluate new features and to compare them with existing methods.

The future work for RINASim involves the following improvements that we would like to integrate:

- Real-life L2 simulation modules – RINASim core lacks any real-life 0-DIF medium implementations (e.g., Ethernet, LTE, serial). This severely impacts some simulation results and collected statistics because the medium's main properties (delay and bandwidth) are fixed and do not respect usual processing;
- Topology generator – Preparing a scenario and accompanied configuration is a cumbersome process even with all the help of OMNeT++'s IDE. We want to create a dedicated web-application that would allow to generate even complex RINASim topologies using a few mouse clicks and drag-and-drops;
- Hardware-in-the-loop simulation – By the time of this chapter publication, RINA will already pass ISO standardization process. Since multiple projects obey RINA specification, we would like to try connect our simulation modules with real operating system implementations;

We encourage anyone interested in RINA to step in and contribute!

7 Appendixes

7.1 CDAP Messages

Message Opcode	Purpose
M_CONNECT	Initiate a connection from a source application to a destination application
M_CONNECT_R	Response to M_CONNECT, carries connection information or an error indication
M_RELEASE	Orderly close of a connection
M_RELEASE_R	Response to M_RELEASE, carries final resolution of the close operation
M_CREATE	Create an application object
M_CREATE_R	Response to M_CREATE, carries result of the create request, including identification of the created object
M_DELETE	Delete a specified application object
M_DELETE_R	Response to M_DELETE, carries result of the deletion attempt
M_READ	Read the value of a specified application object
M_READ_R	Response to M_READ, carries part or all of object values, or error indication
M_CANCELREAD	Cancel a prior read issued using M_READ for which a value has not been completely returned
M_CANCELREAD_R	Response to M_CANCELREAD, indicates outcome of the cancellation
M_WRITE	Write a specified value to a specified application object
M_WRITE_R	Response to M_WRITE, carries result of the write operation
M_START	Start the operation of a specified application object, used when the object has operational and non-operational states
M_START_R	Response to M_START, indicates the result of the operation
M_STOP	Stop the operation of a specified application object, used when the object has operational and non-operational states
M_STOP_R	Response to M_STOP, indicates the result of the operation

Table 2: CDAP messages

7.2 RINA Adoption

Pouzin Society [21] is a formal body in charge of maintaining the RINA specifications. Any individual or organization can become a member and participate on related research and development. RINA is successfully targeted in the frame of multiple EU projects as an alternative to the traditional TCP/IP stack. Here is a list of projects and their main interests concerning RINA:

- IRATI [18] – IRATI advances the state-of-the-art of RINA towards an architecture reference model and specifications that are closer to enable implementations deployable in production scenarios. The design and implementation of IRATI prototype on top of Ethernet permits further evaluation and deployment of RINA in real computer networks;
- IRINA [16] – IRINA aims to compare RINA against TCP/IP in a lab environment using IRATI prototype. Moreover, it proposes use-cases, where RINA is a better option for big national research and educational networks;
- PRISTINE [24] – PRISTINE investigates programmability of RINA architecture, namely its separation of mechanisms and policies to achieve more flexible behavior of network components;
- OCARINA [19] - OCARINA's objectives are to research and develop new congestion control, and routing mechanisms in RINA to show that RINA, indeed is a much better solution for the Internet than TCP/IP in terms of performance;
- ARCFIRE [1] - ARCFIRE's main goal is to demonstrate the RINA benefits at large scale running experiments experimentally (e.g., deployment of virtualized infrastructure, end-to-end service provisioning, the applicability of DDoS attacks in RINA stack) on the FIRE+ experimental facilities;
- RINAiSense [17] - The RINAiSense project improves the scalability and security of wireless sensor networks while investigating the applicability of RINA to resource-constrained systems.

Moreover, notable implementations are introduced and facts about RINA readiness and deployment status:

- OpenIRATI - Open-source programmable implementation of RINA protocols for Linux. Enables Linux device to behave as a RINA-enabled host or software router;
- rlite - Open-source implementation of RINA for Linux. Implementation is divided into user-space and kernel-space parts. Kernel-space parts can be loaded as modules on the unmodified Linux kernel;
- Ouroboros - Prototype Inter-Process Communication subsystem for POSIX operating systems, that incorporates a fully decentralised packet switched transport network based on part of RINA concepts;
- ProtoRINA - One of the implementations of the RINA architecture. It serves as teaching tool, and also enables the design and development of new protocols and applications;
- RINASim - The official OMNeT++ based framework simulating RINA architecture.

Acknowledgement

This paper was supported by the Brno University of Technology organization and by the research grant FIT-S-17-3964.

RINASim is a joint international project, which would not be possible without the huge support and contribution of the following individuals:

- Tomáš Hykel – Brno University of Technnology (CZ);
- Sergio Leon Gaixas – Universitat Politècnica de Catalunya (ES);
- Ehsan Elahi – TSSG in the Waterford Institute of Technology (IE);
- Kewin Rausch – Fondazione Bruno Kessler (IT);
- Peyman Teymoori – University of Oslo (NO);
- Kleber Leal – Universidade Federal de Pernambuco (BR).

List of Acronyms

AE Application Entity
AEI-id Application Entity Instance Identifier
AEN Application Entity Name
ANI Application Naming Information
AP Application Process
APN Application Process Name
API-id Application Process Instance Identifier
CEP-id Connection-endpoint-id
CACE Common Application Connection Establishment
CDAP Common Distributed Application Protocol
DA DIF Allocator
DAF Distributed Application Facility
DAN Distributed-Application-Name
DAP Distributed Application Process
DIF Distributed IPC Facility
DRF Data Run Flag
DTP Data Transfer Protocol
DTCP Data Transfer Control Protocol
EFCP Error and Flow Control Protocol
EFCPI EFCP Instance
FA Flow Allocator
FAI Flow Allocator Instance
IPC inter-process communication
IPCP IPC Process
IRM IPC Resource Manager
NSM Namespace Management
PDU Protocol Data Unit
PDUFG PDU Forwarding Generator
RA Resource Allocator
RIB Resource Information Base
RIBd RIB Daemon
RINA Recursive InterNetwork Architecture
RLWE Receiver's Left Window Edge
RRWE Receiver's Right Window Edge
RMT Relaying and Multiplexing Task
RTT Round-Trip Time
SDU Service Data Unit
SLWE Sender's Left Window Edge
SNMP Simple Network Management Protocol
SRWE Sender's Right Window Edge
QoS Quality of Service

References

[1] ARCFIRE Consortium: ARCFIRE project - Experimenting with RINA on FIRE+. [online] http://ict-arcfire.eu/ (2018)

[2] Day, J.: Patterns in Network Architecture: A Return to Fundamentals. Pearson Education (2008). URL https://books.google.cz/books?id=5FDdAAAACAAJ
[3] Day, J.: D-Base-2010-007: Delimiting Module. Tech. rep., Pouzin Society (2009)
[4] Day, J.: RINA-RFC-2010-002: Notes on the Resource Allocator. Tech. rep., Pouzin Society (2010)
[5] Day, J.: D-Base-2011-015: Flow Allocator Specification. Tech. rep., Pouzin Society (2011)
[6] Day, J.: D-Base-2011-017: IPC Resource Manager (IRM) Specification. Tech. rep., Pouzin Society (2012)
[7] Day, J.: D-Base-2012-010: Relaying and Multiplexing Task Specification. Tech. rep., Pouzin Society (2012)
[8] Day, J.: RINARefModelPart3-1 140102: Part 3 - Distributed InterProcess Communication, Chapter 1 - Fundamental Structure. Tech. rep., Pouzin Society (2012)
[9] Day, J.: RINARefModelPart3-2 140102: Part 3 - Distributed InterProcess Communication, Chapter 2 - DIF Operations. Tech. rep., Pouzin Society (2012)
[10] Day, J.: DelimitingGeneral130904: Delimiting Module. Tech. rep., Pouzin Society (2013)
[11] Day, J.: RINARefModelPart1-0 130925: Part 1 - Basic Concepts of Distributed Systems. Tech. rep., Pouzin Society (2013)
[12] Day, J.: RINARefModelPart2-1 130925: Part 2 - Distributed Applications, Chapter 1 - Basic Concepts of Distributed Applications. Tech. rep., Pouzin Society (2013)
[13] Day, J., Marek, M., Tarzan, M., Bergesio, L.: Error and Flow Control Protocol Specification version 6.6. Tech. rep., Pouzin Society (2015)
[14] Day, J., Trouva, E.: RINARefModelPart2-2 140102: Part 2 - Distributed Applications, Chapter 2 - Introduction to Distributed Management Systems. Tech. rep., Pouzin Society (2014)
[15] Fletcher, J.G., Watson, R.W.: Mechanisms for a reliable timer-based protocol. Computer Networks (1976) **2**(4), 271 – 290 (1978). DOI http://dx.doi.org/10.1016/0376-5075(78)90006-5. URL http://www.sciencedirect.com/science/article/pii/0376507578900065
[16] GÉANT Project: IRINA. [online] https://geant3plus.archive.geant.net/opencall/Optical/Pages/IRINA.aspx (2018)
[17] IMEC-Distrinet: The Recursive Internet Architecture as a solution for optimal resource consumption, security and scalability of sensor networks. [online] https://distrinet.cs.kuleuven.be/research/projects/RINAiSense (2018)
[18] IRATI Consortium: Investigating RINA as an Alternative to TCP/IP. [online] http://irati.eu/ (2018)
[19] OCARINA: OCARINA: Optimizations to Compel Adoption of RINA. [online] http://www.mn.uio.no/ifi/english/research/projects/ocarina/ (2018)
[20] OMNeT++: OMNeT++ Simulation Manual. [online] https://www.omnetpp.org/doc/omnetpp/manual/#sec:ned-ref:module-interfaces (2018)

[21] Pouzin Society: Pouzin Society | Building a better network. [online] http://pouzinsociety.org/
[22] Pouzin Society: RINASim - Recursive InterNetwork Architecture Simulator. [online] https://rinasim.omnetpp.org/
[23] Pouzin Society: kvetak/RINA: RINA Simulator. [online] https://github.com/kvetak/RINA (2018)
[24] PRISTINE Consortium: PRISTINE | PRISTINE will take a major step forward in the integration of networking and distributed computing. [online] http://ict-pristine.eu/ (2018)
[25] Trouva, E., Grasa, E., Day, J., Bunch, S.: Layer discovery in rina networks. In: Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), 2012 IEEE 17th International Workshop on, pp. 368–372. IEEE (2012)
[26] Vesely, V., Marek, M., Hykel, T., Rysavy, O.: Skip this paper-rinasim: your recursive internetwork architecture simulator. arXiv preprint arXiv:1509.03550 (2015)
[27] Watson, R.W.: Delta-t protocol specification. UCID 19293, Lawrence Livermore National Laboratory (1983)